Mathematical techniques for shape modelling in computer graphics: A distance-based approach

Dimitrios TSOUBELIS

March 1995

Dissertation submitted in fulfilment of the requirements for the degree of Doctor of Philosophy at the London School of Economics, University of London.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346


BRITISH LIBRARY
OF
POLITICAL
AND
ECONOMIC
SCIENCE

Στήν Αγγελική

Abstract

This research is concerned with shape modelling in computer graphics. The dissertation provides a review of the main research topics and developments in shape modelling and discusses current visualisation techniques required for the display of the models produced. In computer graphics surfaces are normally defined using analytic functions. Geometry however, supplies many shapes without providing their analytic descriptions. These are defined implicitly through fundamental relationships between primitive geometrical objects. Transferring this approach in computer graphics, opens new directions in shape modelling by enabling the definition of new objects or supplying a rigorous alternative to analytical definitions of objects with complex analytical descriptions. We review, in this dissertation, relevant works in the area of implicit modelling. Based on our observations on the shortcomings of these works, we develop an implicit modelling approach which draws on a seminal technique in this area: the distance based object definition. We investigate the principles, potential and applications of this technique both in conceptual terms (modelling aspects) and on technical merit (visualisation issues). This is the context of this PhD research. The conceptual and technological frameworks developed are presented in terms of a comprehensive investigation of an object's constituent primitives and modelling constraints on the one hand, and software visualisation platforms on the other. Finally, we adopt a critical perspective of our work to discuss possible directions for further improvements and exploitation for the modelling approach we have developed.

Table of contents

List of figures

List of tables

List of plates

Chapter 1 An introduction to computer graphics

1.1 A brief history of computer graphics

Computer graphics is the part of Information Technology concerned with the visual representation of data. We can identify two major categories of such data; data which is the output of other systems and data that is constructed for the purpose of producing a particular image.

Data from other systems is either the output of software packages like databases, spreadsheets and other statistical and scientific applications, or from other specifically designed input peripherals such as two or three-dimensional scanners, and other sensors. The benefit from visualising such data sets — typically of numerical nature — is that through the effective utilisation of shapes and colours it is much easier to present an overall 'picture' of this data. It is anticipated that such representation will reveal to the user potential areas of interest in the data set under study. The most critical factors for such an application are the accuracy of the input data, the effective use of colours and shapes (e.g. bar graphs, scatter plots, histograms, colour coded dimensions etc.), and the adequate preparation of the input data for visualisation purposes (e.g. isolation of the required sub-set, scaling, various consistency checks, formatting of data and other types of preprocessing).

For many applications, however, the emphasis is not in the collection of data, but in the construction of the necessary data to produce a required image. Such images are usually reproductions of real life objects, and their shape is approximated at a certain level of detail with the use of geometrical methods that may vary significantly in complexity. The input data in such an application will usually be descriptions of the shapes and colours of the details of all the real life objects that need to be visualised.

Recent advances in hardware technologies have played a very important role in enabling computer graphics techniques to be used in a great range of applications. The computational demands of computer graphics techniques have been met by innovations in computer

architectures including RISC processor sets (e.g. IBM's POWER), dedicated video bus (e.g. VL VESA bus), additional graphics processors (e.g. XGA chipset), application specific graphics boards (e.g. IBM's hipfmgex 3-D High Performance Graphics Processor for polygon rendering, and z-buffers for hidden object elimination) and multiprocessor arrangements (e.g. transputer clusters, distributed computing environments). As a result, nowadays computer graphics applications include *business graphics*, *computer aided design (CAD)*, *human - computer interaction (HCI)*, *computer aided education (CAE)*, *multimedia (MM)*, *computer animation*, *medical imaging*, *document image processing (DIP)*, and *virtual reality (VR)*. These categories of applications are classified according to the type, or types, of data they primarily manipulate.

For example, the domain of business graphics is concerned with the graphical representation of numerical data and relates to graphical forms such as bar graphs, line drawings and pie charts. The visualisation of more complex sets of data geared towards the needs of a particular problem are encompassed under the general term of *scientific visualisation*. Such an application area is that of medical imaging, where data, usually scanned from the human body, are visualised and processed to depict pathological areas accurately. Another approach to visualisation is taken with computer aided design (CAD) which is concerned with the design and subsequent visualisation of geometrical shapes, usually models of real-life objects. There, the user is able to have a preview of an object yet to be manufactured. Additional software may be used to enable a thorough testing of some of the physical properties of the modelled objects. Usually the implementation of computer aided design applications is complemented by suitable computer aided manufacturing (CAM) applications, where CAD designs are being transformed into instructions for manufacturing tools in order to facilitate efficient and accurate manufacture of the CAD models.

Another less ambitious application for computer graphics, but of great commercial interest, is that of computer animation, where the effect of object motion is added to that of object visualisation. This approach has been successfully introduced in the production of television (TV) advertisements. Recently it has also entered the courts of justice where (usually) the defence presents its case of events via computer generated animation that re-creates the alleged crime from the viewpoint of the accused, the victim, or other witnesses involved.

The domain of human - computer interaction (HCI) is orientated towards the study of the presentation of both textual and numerical information in order to facilitate efficient interaction between computer software and users. Here important issues include the choice of colours used depending on the importance of the information presented, the density of characters presented on the computer screen at any given time, the character sets used, the number, location and size of windows used, the use of pointing devices and other equipment necessary for user interaction.

Computer aided education and document image processing are both involved with the manipulation and sophisticated retrieval mechanisms of information in the form of pictures. Here the emphasis is on the presentation of images that are relevant to the user's search queries. In computer aided education systems, the user is presented with pictorial information, usually for the purpose of a tutorial, where context sensitive help can be provided. In document image processing applications, the user is not only presented with images of the documents that need to be consulted but in many cases the user is also expected to amend them, thus progressing towards the realization of the dream of the 'paperless office'.

Recently, a new application domain that encompasses the utilities of most of the above has emerged. This is 'virtual reality', that involves the design and visualisation of models of objects, their motion through a virtual space, and the active participation of users who are able to 'wander around' this space. The potential of such an application has been appreciated in fields where the training of new personnel necessitates realistic conditions that are too expensive, too dangerous or impossible to control, for example for the training of pilots, astronauts, special army personnel etc.

Despite the differences the above application domains may present, both in terms of the nature of data used and the purpose of the images produced, they all share the common need of data *preparation*, or *modelling*, before data *visualisation*. This observation illustrates the fundamental principle that computer graphics is a two phase process that involves the *modelling* and the subsequent *visualisation* of models of objects. Depending on the application, the emphasis on either phase (i.e. modelling, visualisation) may vary, but both

phases are necessary in all applications. For each phase there exist a number of different approaches such as *polygonal mesh approximation* and *analytical modelling* with regard to modelling, and *scan line* and *ray tracing* with regard to visualisation, all of which will be presented in subsequent chapters.

The linkage between modelling and visualisation approaches will determine the balance between *response time* and *image quality* for a given application. Response time is an objective measure that is defined by the time it takes a particular installation (hardware / software) to produce the required image. Image quality is a rather subjective criterion for assessing how (photo-) 'realistic' the produced image is. The demands of the particular application in terms of image quality and response time will effectively dictate the hardware platform and the combination of modelling and visualisation approaches best used.

In this dissertation, the emphasis is given to the techniques used in computer graphics and especially in modelling, and therefore the particulars of an individual application domain will not be thoroughly examined unless they are an essential ingredient to our approach. More specifically, the aim of this thesis is to develop a technique that defines a new family of geometric shapes. Our approach is to exploit the power of computer graphics in order to achieve the visualisation and initial study of geometrical objects that are too complex, or impossible, to be described analytically.

We define such objects as the sets of points that fulfil a number of constraints (i.e. geometrical loci). We will use a simple but powerful constraint that emerges from extensions to the measure of distance, as will be explained in subsequent chapters. A formalisation of linear combinations of such an extended measure of distance will construct a generic definition of several new types of geometrical objects. Usually these objects are sets of surfaces but there are special cases where they degenerate into discrete points or even empty sets. The nature of the produced objects (i.e. points, surfaces, ...) as well as their characteristics (i.e. size, area, curvature, ...) are parameterized thus enabling the generation of families of such objects.

This technique for surface generation extends contemporary methods for defining simple geometric objects like the circle and the ellipsoid. Its power comes by enabling visualisation and initial study of geometric objects that are impossible to be described and therefore studied by traditional analytical means. Nevertheless, this new technique will employ modifications of currently used computer graphics methods for both phases of modelling and visualisation, in order to enable an initial study of this new family of geometric objects.

In this chapter, an introduction to the fundamental principles of computer graphics will be presented. This will include an explanation of the relevant terminology, a presentation of the necessary mathematics concerning n–dimensional Euclidean spaces, a breakdown analysis of the comprising stages of a (typical) computer graphics application, and, finally, a presentation of the actual hardware and software platforms we used to develop and test the implementation of our proposed object generation technique. The next two chapters will be devoted to the presentation and analysis of the main modelling and visualisation techniques currently used in computer graphics applications. The presentation of these techniques will be followed by our evaluation and criticisms concerning their suitability to the object generation technique that this dissertation proposes.

Having presented all the necessary background information needed for establishing common ground of understanding with the reader, in chapter four we will present recent advances in modelling including research relevant to our proposed technique; namely *implicit modelling*. Chapter five will be devoted to the definition of the proposed object generation technique. This will include the rationalisation of an extended measure of distance and the presentation of a generic mathematical definition out of which a number of interesting special cases will be isolated and investigated further. In chapter six we will examine the challenges we met during visualisation of such defined objects. There, an analysis of various alternative visualisation techniques will be discussed and followed by the presentation of the preferred alternatives. Finally, chapter seven will entail a demonstration of the potential of the proposed object generation technique. There, some special cases of the proposed technique will be used to illustrate its power and its potential with regard to the contemporary techniques. This contrast will be analysed to show future directions for further generalisations and expansions of the proposed approach.

1.2 Definitions

The aim of any computer graphics application is to produce an image (or a series of images) on display device. Such a device could be the monitor of a computer, a television screen, a film surface etc. [Smarte 1988]. Except for the class of direct volume display devices (DVDD) [Clifton III and Wefer 1993], this display device is assumed to be a two-dimensional area that for the remainder of this dissertation will be called the *viewport.*

As mentioned earlier, the underlying principle in all computer graphics applications is that the process of producing an image via a computer involves two phases. In the first phase, *modelling*, the description of what needs to be displayed is produced. This description is called the *model* of the image or the *scene*. The constituent parts of the scene are also called the *objects* of that scene. The model will provide information about the shape of the component objects in the scene, their relative size and positions, and, very often, their colours.

In most applications the shape of the modelled objects is approximated by simple geometrical objects such as planar polygons, spheres, boxes etc. The mathematics used for shape approximation will be presented in this chapter. Nevertheless, the model of a scene may also contain information about processes that need to be used to determine the shape of an object. Such processes may include the use of fractals, pseudo-random number generators, or other geometrical or physical constraints such as the geometrical loci, gravity or elasticity. This kind of modelling is called *implicit* or, *procedural modelling* and will be studied in subsequent chapters.

In the second phase, *visualisation*, an imaginary observer is introduced, and its view of the modelled scene is reconstructed onto the display device as a picture. To add realism into the produced image, a number of light sources that are assumed to illuminate the scene, is also introduced. During visualisation, optical phenomena such as light reflection, light refraction, radiosity etc. may also need to be simulated. However, what is actually displayed on the viewport is not a picture of a real world scene, as a photographic camera would capture, but an image of some mathematical model describing that scene.

A variety of mathematical tools such as topology, metric spaces, matrix algebra, calculus, trigonometry and numerical analysis are used in both phases of a computer graphics application. Nevertheless, the area of mathematics that is the most fundamental in computer graphics is coordinate geometry. This is because both the model and the displayed image are expressed via Euclidean spaces.

1.3 The Euclidean space

It is essential, therefore, to illustrate the use of Cartesian coordinate geometry as a means of representing the Euclidean space. Since the same principles apply in both the representation of the model and the representation of the image on the viewport, it is appropriate first to describe the general case of the Cartesian coordinate system of two dimensions and then to show how it can be adjusted to represent the model and the resulting image.

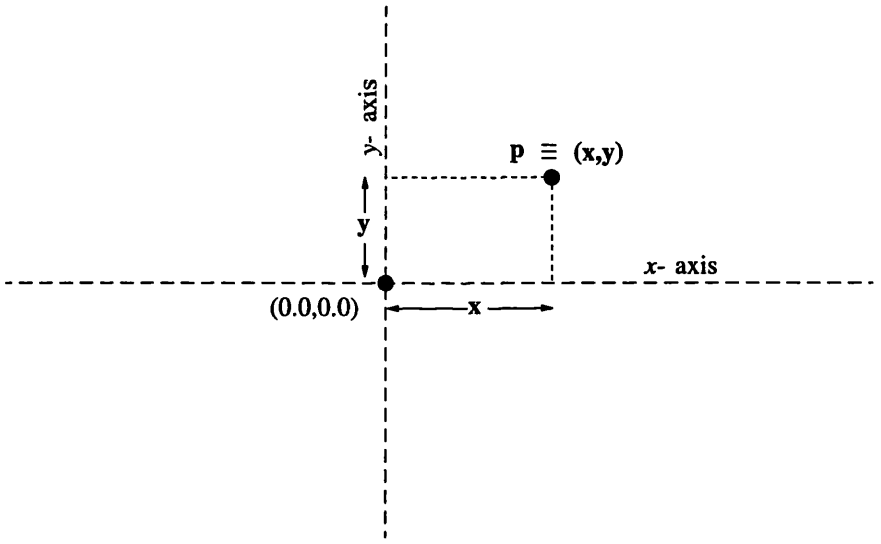


Figure 1.1 The two-dimensional Cartesian coordinate system

We may imagine two-dimensional space as the plane of this page, as Figure 1.1 shows, but extending to infinity in all directions. In order to specify the position of points uniquely, we have to impose a Cartesian coordinate system on the plane. We start by arbitrarily choosing a fixed point in this space, which is called the *coordinate origin*, or *origin* for short. A line that extends to infinity in both directions is drawn through the origin – this is the *x-axis*.

The normal convention, which we follow, is to imagine that we are looking at the page so that the x-axis appears from left to right on the page (the horizontal). Another two-way infinite axis, the *y-axis*, is drawn through the origin perpendicular to the x-axis; hence conventionally this is placed from the top to the bottom of the page (the vertical). We now draw a scale along each axis; unit distances need not be the same on both axes or even linearly distributed along the axes, but this is normally the case. We assume that values on the x-axis are positive to the right of the origin and negative to the left: values on the y-axis are positive above the origin and negative below.

We can now uniquely fix the position of point $\boldsymbol{p}$ in space with reference to this coordinate system by specifying its *coordinates*. The *x coordinate*, $\boldsymbol{x}$ say, is that distance along the x-axis (positive on the right-hand half-axis, and negative on the left) at which the line perpendicular to the x-axis, that passes through $\boldsymbol{p}$, cuts the axis. The *y coordinate*, $\boldsymbol{y}$ say, is correspondingly defined by using the y-axis. These two values, called a *coordinate pair* or *two-dimensional point vector*, are normally written in brackets thus: (x, y), the x coordinate coming before the y coordinate. We shall usually refer to the pair as a *vector* – the *dimension* (in this case dimension two) will be understood from the context in which we use the term. A vector, as well as defining a point (x, y) in two-dimensional space, may also be used to specify a *direction*, namely the direction that is parallel to the line joining the origin to the point (x, y).

Having defined the two-dimensional Cartesian coordinate system, it becomes apparent how the three-dimensional system can be determined. We can imagine a third two-way infinite axis, the z-axis, passing through the origin and being perpendicular to both x-axis and y-axis. Following our conventions therefore, the z-axis will be perpendicular to the plane of the page, and its positive part will either be coming towards us thus defining the *right-handed coordinate system*, or away from us (*left-handed coordinate system*). Assuming the unit distances have also been defined along the z-axis, a point in space (three-dimensional) will be determined as a *three-dimensional coordinate vector*, normally written as a triple of its corresponding x, y and z coordinates: (x, y, z). The left-handed three-dimensional Cartesian coordinate system, as defined here, will be used in the remainder of this dissertation to describe all our three-dimensional objects of our modelled scenes.

It must be realized that the coordinate values of a point in space are totally dependent on the choice of coordinate system. During our analysis of computer graphics modelling and visualisation techniques we will be using a number of different coordinate systems to represent the same objects in space, and so a single point in space may have a number of different vector coordinate representations. For example, if we have two coordinate systems with parallel axes but different origins – say separated by a distance 1 in the x direction, and 2 in the y direction – then the point (0 , 0) in one system (its origin) could be (1 , 2) in the other: the same point in space but different vector coordinates (Figure 1.2).

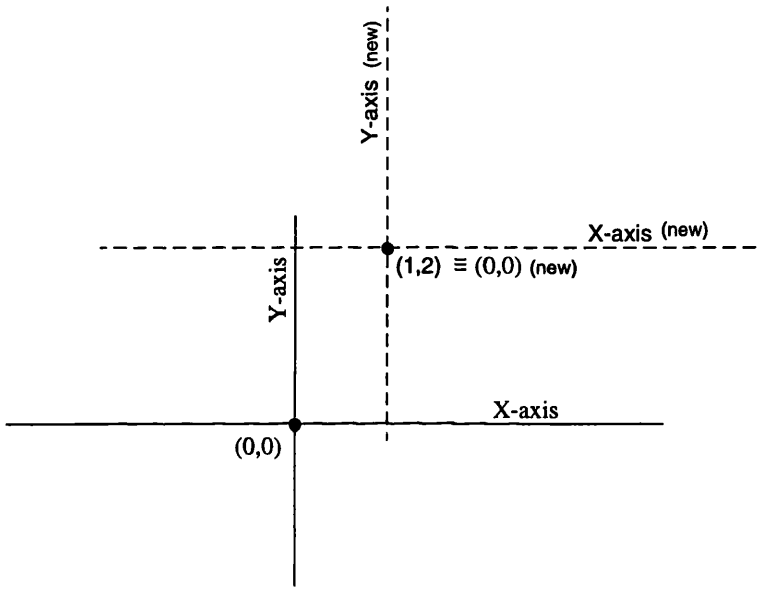


Figure 1.2 Using two coordinate systems

The transition from one coordinate system to another is achieved by the use of matrices. In coordinate geometry simple *affine transformations* like the translation, rotation and scaling of axes are represented by matrices. Furthermore, combinations of such transformations can also be represented by (products of) matrices [Angell & Tsoubelis 1992]. Nevertheless, transformations exist, not only for the substitution of coordinate systems of the same dimension (as in the above example) but also for systems with different ones. The latter case, is usually encountered in the transformation of a (usually) three-dimensional space (used to describe the scene) to the two-dimensional space of the viewport. This special type of transformation that reduces the dimension of space is called *projection.*

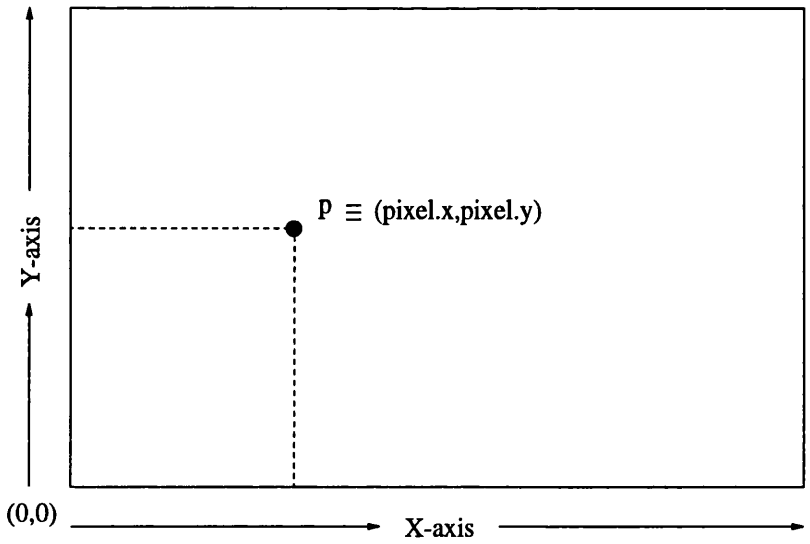


Figure 1.3 The viewport's Cartesian system

In order to use the display device, we will also introduce a very special case of a two-dimensional Cartesian coordinate system which is assumed to have points of integer coordinates. The reason is that the viewport is assumed to be composed of a rectangular array of points called *pixels* (picture elements). As a result, points on this space will be the pixels, and unit distances on the x-axis and y-axis are assumed to equal the horizontal and vertical size of the constituent pixels. And therefore, according to this observation, the coordinates of any pixel in the viewport may only be integer multiplicands of the corresponding unit distances. We can simplify the viewport's coordinate system by ignoring the pixel's unit sizes and instead, measure pixel coordinates as number of pixels from a predetermined origin. And assuming that the origin of the system is the bottom left corner of the viewport, as Figure 1.3 shows, the coordinates of any pixel on the viewport will be the number of pixels to the left and below it.

1.4 From mathematical models to images

In order to clarify the relationships between different systems we ought to work with one fixed coordinate system only. In computer graphics, however, it proves to be very convenient to use more than one coordinate systems. Therefore, for our clarification, we will use at least four different coordinate systems, namely the *ABSOLUTE*, the *OBSERVER*, the

WINDOW, and the *VIEWPORT* systems. In particular cases, however, as we will see in the next two chapters, other coordinate systems like the *LIGHT* systems may also be used.

The ABSOLUTE system will be used for describing our model and the OBSERVER system for calculating the view of the scene as seen by a particular observer. The LIGHT systems will be used to describe the scene 'as seen' by each particular light source. The WINDOW coordinate system will be used to represent the resulting image onto a model output device (realized by the viewport), while the VIEWPORT system will be used to describe the image on the screen of a particular display device. The WINDOW system will use real numbers for point coordinates where the VIEWPORT will use pixel units for determining pixel coordinates (Figure 1.2).

All objects in a scene are therefore described using the ABSOLUTE system and we name their position the *ACTUAL* position. For convenience, however, each individual object may be described in a simple way usually around the origin of the ABSOLUTE system. This we call the *SETUP* position for that particular object. Therefore, objects are first individually defined with reference to their own SETUP position and then they are moved to their ACTUAL position, thus constructing the required scene. It is implied here that both positions are described with reference to the ABSOLUTE system and the transformation from one position to the other is achieved by the appropriate matrices ($\boldsymbol{P}_i$) for every object i in the scene, as Figure 1.4 shows.

Usually, after the scene is defined, an imaginary observer is introduced and the observer's view of the scene is required to be reconstructed on the viewport. The observer's position is described with reference to the ABSOLUTE system, but since this viewpoint becomes the most critical point during visualisation a new coordinate system is introduced. This is the OBSERVER system that has its origin where the eye of the observer is, and its orientation is determined by the observer's *direction of view*. It is implied here that we use a single-eye observer! For a 'realistic' two-eye observer two different views (one for each eye) need to be calculated and therefore two OBSERVER systems should be determined.

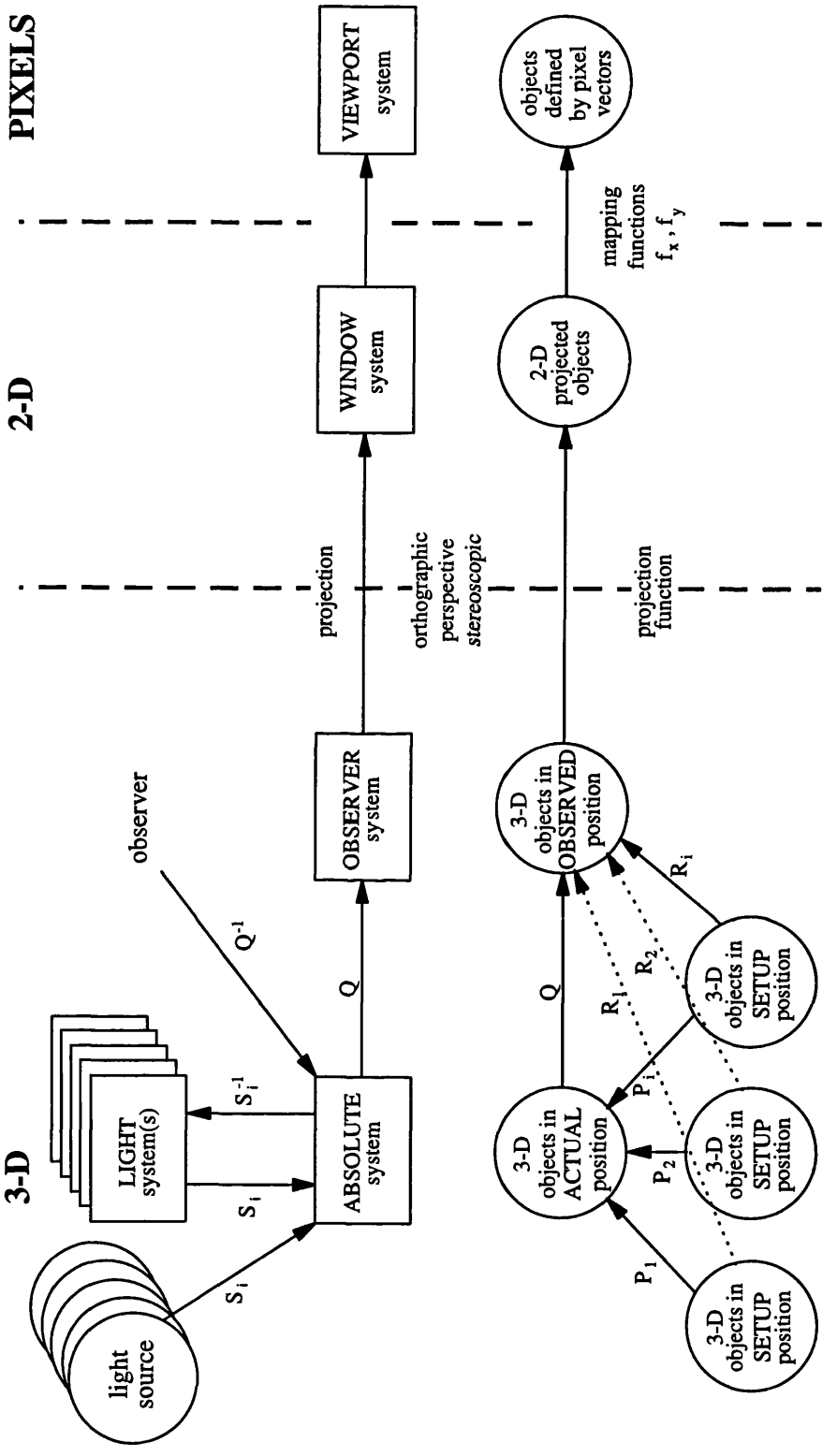


Figure 1.4 From models to images

At this stage, all objects that are currently described in their ACTUAL position with reference to the ABSOLUTE system will be transformed (matrix $\boldsymbol{Q}$) to their OBSERVED position with reference to the OBSERVED coordinate system. As a short-cut it is not uncommon to avoid using the ACTUAL position of objects and instead use for every object i in the scene, the combined transformation (given by the product matrix $\boldsymbol{R_i} = \boldsymbol{Q} \times \boldsymbol{P_i}$) from the SETUP directly to the OBSERVED position.

Another addition to the model may occur at this stage. This is the introduction of light sources that illuminate the scene. The light sources are described with reference to the ABSOLUTE system and usually a transformation to the OBSERVED system is also needed. However, for additional functionality such as the representation of shadows, some computer graphics applications define for every light source, a *LIGHT* coordinate system. Its origin is the light source and its orientation is appropriately chosen to ease the calculation of shadows produced by that source. Transformations between the ABSOLUTE and each LIGHT system are achieved by the matrices $\boldsymbol{S_i}$ and $\boldsymbol{S_i^{-1}}$ uniquely defined for each source.

Having described all the constituents parts of a scene, usually in three dimensions, the next stage is the reconstruction of the observer's view. This implies that we need to calculate the image of the scene as this appears on the retina of the observer's eye. We must reiterate here that we use single-eye view. Since this view is built on the two-dimensional retina, we need to define and use a new coordinate system of dimension two; the *WINDOW* system. Very often, its origin is determined by the direction of view and its orientation is parallel to the *X-Y* plane of the OBSERVER system.

The transformation from the OBSERVER to the WINDOW system involves a reduction in dimensions, usually from three to two. Such a transformation is called *projection*. Depending on the application, a variety of projections can be used. These include the *orthographic*, the *perspective*, the *stereoscopic* etc. [Angell & Tsoubelis 1992]. The most 'realistic' type of projection is perspective. This produces two-dimensional views in a way similar to those of the natural eye or the photographic camera. Projections are also represented by matrices thus enabling a homogenous approach for the complete transformation of the scene description from the OBSERVER to the WINDOW system.

Once the required image is described in the WINDOW system, the final step is to depict it on the graphics viewport. Therefore, a conversion from vector coordinates (real numbers) to pixel coordinates (pixel counts) is necessary. The image description in terms of pixels will be determined via the VIEWPORT system. To achieve this conversion, we first isolate a finite rectangular area (or *window*) within the WINDOW system. This rectangular window is centred around the origin of the WINDOW system and is to be identified with the graphics viewport. Therefore, the mapping functions from the WINDOW to the VIEWPORT systems will be determined from the window and the viewport used. We assume that such a viewport is composed of a rectangular array of points (the pixels, or picture elements). This matrix of points measures ***nxpix*** pixels horizontally by ***nypix*** pixels vertically, counting from the bottom left corner of the viewport (Figure 1.5).

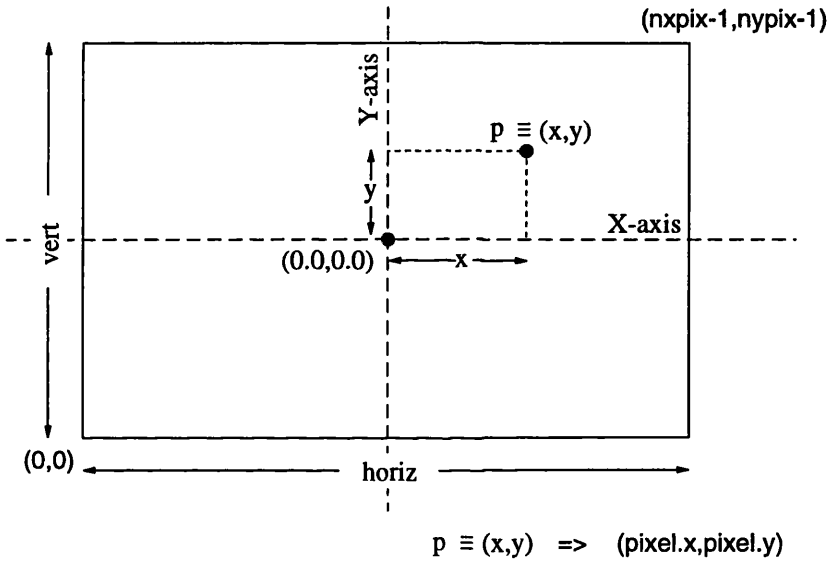


Figure 1.5 Relating points to pixels

The mapping from real window coordinates to integer coordinates (multiples of pixel units) is achieved by dividing the window into *nxpix* x *nypix* equal sized rectangular areas called *sub-windows*, that correspond to the *nxpix* x *nypix* pixels of the viewport. Therefore, for a given point inside the window, we determine the sub-window to which it belongs, and map it to the corresponding pixel on the viewport. As a result, all points that belong to the same sub-window will be mapped to the same pixel. For example, as Figure 1.5 shows, for a window of ***horiz*** x ***vert*** size and a viewport of *nxpix* x *nypix* pixels, the coordinates of point ***p*** ≡ ***(x , y)*** will be mapped onto the pixel ***(fx(x) , fy(y))*** ≡ ***(pixel.x , pixel.y)*** via the functions:

$$fx(x) = \left[\frac{2x + horiz}{2\,nxpix}\right], \qquad fy(y) = \left[\frac{2y + vert}{2\,nypix}\right]$$

where $[r]$ denotes the integer part of the expression r.

1.5 The model display device

Consequently, we can draw two-dimensional views of projected scenes on the graphics device, by simply relating the real coordinates of points in the window with their corresponding pixels in the viewport. For reasons of portability and flexibility, however, the graphics device is not assumed to be the particular screen, monitor or plotter that is part of the currently used hardware installation. Instead, a *model graphics display device* is introduced. This is assumed to be a virtual viewport of an arbitrary but fixed size with the additional capability of discriminating among 16,777,216 different colours.[1]

The role of such a model graphics device is to hold an as accurate an image description as possible, given the constraints of the particular installation (e.g. memory capacity), and the demands of the application (e.g. image resolution). This form of the image is usually held in secondary storage. This will enable further manipulation of the image, such as archiving, post-editing, viewing, plotting on microfilm etc. as Figure 1.6 shows. The main advantage of using such a generic form for the produced image is therefore portability; the same image can be viewed on a variety of different hardware viewport devices with minimal effort. In this way the image is realized on any viewport by utilising fully the underlying hardware.

In terms of image quality, the nearest the specifications (i.e. screen resolution, number of available colours) of the real viewport are to the model device, the better. It follows, however, that a real viewport with specifications considerably lower than the model device's will result into loss of information and significant degradation of image quality. Before

[1] Usually modern viewports, as we shall see in later sections, provide support for an 8-bit description for each of the red, green and blue colour components. Consequently the total number of possible colour combinations is $(2^8)^3$.

discussing the challenge of transferring an image from such a model display device onto a real hardware viewport, it is essential to explain the format that our images are described.

Image description on the viewport may take one of the following two forms; *vector* or *raster*. A vector image is the one described by a set of line segments (the vectors) of the appropriate colours at the appropriate locations inside the viewport. In contrast to vector images, a raster image is described in terms of the colour of the viewport's constituent pixels. In most of our applications we will be using raster images since this format of image description is inherent to most of the visualisation techniques we will be using (e.g. octree, ray tracing).

However, the use of the model graphics device as a means for representing raster images imposes a considerable demand in storage requirements. The two main factors determining the storage needs are the *pixel* and *colour resolution* of the model device. The pixel resolution determines the size of the device in terms of pixels. Referring back to Figure 1.5, the maximum allowable values for ***nxpix*** and ***nypix*** will determine the maximum size (measured in pixels) of an image that can be represented in the model device. Therefore, for a raster image of the maximum size, we need to keep information of the colour of all the model device's ***nxpix*** × ***nypix*** pixels.

An issue that is often underestimated here is the actual size of the pixel. This relates the *aspect ratio* of the real viewport with that of the model device. The aspect ratio of a viewport is the fraction of the physical dimensions of its displayable surface over those of its pixel resolution. In other words, the aspect ratio of a viewport is the fraction of the horizontal over the vertical physical size of a pixel. It follows that a square pixel (when displayed on the viewport) yields an aspect ratio of 1:1. For an accurate reproduction of shapes, these ratios must be equal; a circle will be distorted into an ellipse, lines will change their slope etc. Fortunately, most hardware viewports use square pixels. Nevertheless, there are common purpose machines, like the IBM personal system, that use screen modes of 6:5 aspect ratio (VGA mode 19). For reasons of convenience, we use a model graphics display device of square pixels. This affects the actual choice of values for ***nxpix*** and ***nypix***, since their ratio has to equal that of the horizontal (***horiz***) over vertical (***vert***) dimensions of the

window used (Figure 1.5). Viewport arrangements that do not match the ratio of the dimensions of the window, are addressed by the introduction of appropriate non-uniform scaling functions.

The second major characteristic of the model device is that of colour resolution which determines how many different colour values a pixel can take. In other words, it determines the number of different colours that the model device is capable of depicting. However, since the model device is a virtual viewport, this maximum number can be arbitrarily set to any value. Research [Wyszecki *et al.* 1982] has shown that the human eye cannot differentiate between intensities that differ by less than 1% for a black and white image, and so perceives them as a continuous tone. Therefore, for a high quality 'photo-realistic' image, no more than 256 (= 2^8) shades of any particular colour will be needed[2]. This observation results in technical specifications that fall well within the limits of the dynamic range of the most common display devices used. For example, a typical video monitor that uses the cathode ray tube technology is capable of depicting between 400 to 530 different intensity levels, and a typical photographic film can go up to 700 [Foley *et al.* 1990].

Furthermore, colorimetry informs us that any visible colour can be expressed, hence approximated, by the combination of three *primary independent variables*. This has resulted in a worldwide accepted standard called the *Commission Internationale de l' Éclairage (CIE) chromaticity diagram* that maps the complete colour range as a linear combination of these three primaries which have been assigned the informative names *X*, *Y* and *Z*. Although the CIE diagram has been accepted as the worldwide standard for colour description, there is no exact (i.e. analytical) description that would map any given colour spectrum into the corresponding X, Y and Z primaries and vice versa. Nevertheless, data tables, that approximate these functions at 1nm intervals of visible light frequencies as perceived from a 2° and a 10° field of view on the retina, have been constructed via laborious measurement experiments.

[2] The choice of value 256 is convenient since we can exactly use one byte (= 8 bits) of computer memory to represent the intensity value of any colour. The choice of a smaller value, say 128 (= 2^7) may yield an image of similar quality but the manipulation of seven-bit values may become a very troublesome process.

This inconvenience, however, has led to the construction of a number of alternative *colour models*, all attempting to describe a significant subset of the complete *colour space* (i.e. set of all conceivable colours) as a combination of measurable parameters like hue, saturation, brightness, or relative colour intensities. For example, the *RGB colour model* is based on the tri-stimulus theory for colour perception[3] and describes colour as the combination of intensity values of three primary pure colours; the red, the green and the blue. Other colour models include the *CMY* (cyan, magenta, yellow), the *CMYK* (cyan, magenta, yellow, black), the *YIQ* (luminance, chromaticity), the *HSV* (hue, saturation, value), the *HLS* (hue, lightness, saturation) etc. For a complete study of the most frequently used colour models the reader is referred to [Hall 1989; Meyer *et al.* 1980]. In order to retain compatibility with the CIE standard, transformation algorithms have been devised that convert the defining parameters between the RGB model and the CIE as well as between the RGB and the rest of the colour models. In our applications, we will be using the RGB colour model. The rationale for our choice is convenience since the RGB model is used in all hardware viewports we will be using, including computer monitors, paper printers and film plotters.

However, by using full colour (totalling 24 bits of data) information for an image we cannot always ensure its accurate realization on a hardware viewport as the technical characteristics of viewports may vary drastically. Such characteristics include the maximum number of colours that can be represented simultaneously on the viewport, the dynamic range of the viewport, its gamma correction mechanism etc. These difficulties, that stem from the differing technical specifications of the various viewport devices, may be clustered into two broad categories:

- using a limited amount of colours to represent an image
- using the 'right' colours to achieve accurate visual impression of the image

The aim in the first category is to use the viewport's available colours in order to approximate the image. Here techniques like *halftoning* and *random dithering* are being used mainly for paper printers [Holladay 1980; Knuth 1987], while adaptive algorithms like the

[3] This theory is based on the hypothesis that the retina has three kinds of colour sensors called *cones*, with peak sensitivity to red, green and blue light. Experiments have shown that the peak sensitivities are at 580nm, 545nm and 440nm of the visible spectrum accordingly.

popularity, the *median-cut* [Heckbert 1982] and the *agglomerative clustering* [Xiang & Joy 1994] are commonly used for cathode ray tube devices. The former techniques build a wide palette of shades by combining together the device's limited set of colours (e.g. the dot size of a black and white laser printer). In this way, the colour gamut of the device is used to approximate the required colours. In the latter techniques, however, the emphasis is in determining the exact amount of appropriate colours (e.g. 256 in VGA mode 19) that will be used to represent the image's *colour gamut*. The, effectiveness of these techniques therefore depends on knowledge of the required colour gamut before visualisation commences.

The aim of the second category of challenges is to achieve accurate image reproduction on a variety of different viewports. Depending on the technology used for a particular hardware viewport, significant differences in the colours of the reproduced image may be perceived. Attempts for device independent colour reproduction include monitor calibrating devices, predetermined colour palettes and ANSI-standard calibration targets [McMillan 1992].

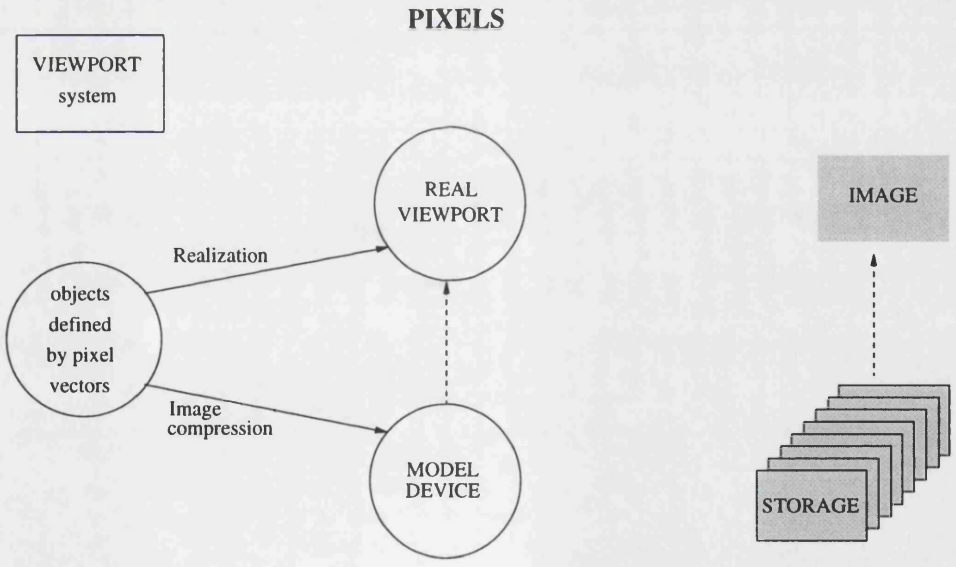


Figure 1.6 Manipulation of image files

To recapitulate, in our computer graphics applications we will be using the concept of the model graphics display device and all image files will (eventually) follow the raster format. These images will either be archived in the computer's secondary storage devices (i.e. disks,

tapes) for further manipulation and subsequent approximation to a viewport, or immediately realized on a hardware viewport (Figure 1.6). This model device will use the VIEWPORT pixel coordinate system to describe raster images of up to a maximum *nxpix* × *nypix* square pixel resolution. The colour of each pixel will follow the RGB colour model and will use up to 256 different values for each primary colour (i.e. red, green, blue), thus using three bytes of colour information per pixel. This results in a huge storage demand even for images of a moderate size.

For example, a typical image of 1024 × 768 pixels and 24 bit (3 × 8) colour resolution will need 2,359,296 bytes (= 2.25 Mbytes approximately) of memory space. For the production of a 35mm colour slide at ISO 100 however, we will need an image of 4096 × 2730 pixels in size and of 24 bit colour resolution that will result into a 14-fold increase in storage demands (approximately 32 Mbytes). Furthermore, technical constraints impose the need to convert a binary stored image into a character-based one to ensure its safe transmission to a variety of communications links and hardware platforms[4] thus resulting in doubling the size of the image file. It is essential, therefore, that transferring such a model device raster image onto the secondary storage should involve an extra phase of processing, namely *image compression* that will be presented in the next section.

1.6 Image compression

Data compression techniques aim at reducing the size of the necessary amount of data used, while keeping their information content intact. They establish their effectiveness on the amount of redundancy that exist in a particular set of data. The techniques aiming at reducing the size of data files that hold images, whatever their particular format may be, fall into the broad category of information theory, namely *data compression.* Especially with images, however, it may not even be necessary to preserve all the information of the original

[4] Using all 256 possible values of a byte implies that the control characters from the relevant character set (ASCII, EBCDIC) are also being used. However, when transmitting such an image file via computer networks some of the non-printable characters may be incorrectly interpreted as control characters by the network circuits thus permanently distorting the contents of the image. The safest approach to overcome this possibility is to use only printable characters (ie. alphanumeric) thus doubling the size of the image file.

un-compressed image. In many applications image details may get corrupted, or lost, for the benefit of achieving a greater reduction on the size of the original image file. The effectiveness of such techniques can be measured with the *compression ratio* they achieve, the speed of actually applying the compression and de-compression algorithms and the similarity of the de-compressed data file when compared with the original.

The compression ratio is defined by the ratio of the original to the compressed file, while the degree of similarity is a subjective measure of how close to the original the de-compressed image 'looks'. More objective measures, like the exact number of bytes the images differ, cannot be applied since the importance of the image is the information it carries and not the exact bytes that describe it. In other words, we are interested in the preservation of the important details of an individual image.

Fortunately, raster images offer a great proportion of redundancy. It is not uncommon that adjacent pixels may hold exactly the same or, very similar colours. This is true especially on the background coloured pixels. Furthermore, very often only a small proportion of the complete colour space is needed for any particular image. Therefore, many techniques are based on the above observations and depending on the contents of an image file, produce astonishing compression rates ranging from one (10:1) up to four orders of magnitude (10000:1).

Data compression techniques for raster images may be based on the actual number of colours used by a particular image, the frequency distribution of the actual colours used, or on the colour information of adjacent pixels (context sensitive data compression). Apart from some Image Processing applications, where the frequency distribution of the actual colours used for a particular image are known (or, can be calculated at real time) the rest of the computer graphics applications use context sensitive data compression techniques. Such techniques are based on information (or, rather redundant information) of the colour of neighbouring pixels.

Since in our applications image quality is more significant than storage space, we adopted an image compression technique that preserves the original image. It is a context sensitive

technique, called *run length encoding*, and works as follows: the raster image is scanned in a pre-determined direction, say from the top left corner to the bottom right one by moving from left to right first. Once two or more consecutive (according to this pre-determined direction) pixels are found to be of the same colour, then replication of information is avoided by recording the number of consecutive pixels that have same colour.

Therefore, the format of the compressed image file is a series of pairs of pixel counts followed by their common colour value. For pixel counts up to 256 pixels, one byte may be used, thus in the best case of 256 consecutive pixels all having the same colour only four bytes are necessary instead of the 768 (= 256 × 3) on the original raster image. However, in the worst case, of consecutive pixels of different colours, one extra byte is added per pixel (for the pixel count) thus increasing the size of the image file by 33%. Yet, experience has shown that, on average, run length encoding can achieve a compression ratio of 100:1. Apart from preserving the original image, run length encoding was chosen because it is also being used in the software driving all the hardware graphics viewports of our installation (i.e. computer monitor, paper printer, and film plotter drivers).

Nevertheless, not all of them are capable of preserving the original image, when decompressed. It is not uncommon that a compromise on the image detail (with reference to the original un-compressed image) may take place in order to achieve a considerably better compression ratio. One such technique is implemented by the Iterated Function Systems (IFS) algorithm [Barnsley *et al.* 1988; Horn 1989; Barnsley 1989] that achieves compression ratios of 1000:1 or higher. It is based on *the collage theorem* that claims to construct an image from a union of sub-images and uses two-dimensional transformations that contract space. The aim of the IFS technique is for a given image to derive the necessary set of transformation matrices. Depending on the geometric regularity of the image (i.e. well defined shapes, not 'random patterns') and the required degree of similarity with the original, this method may expand the image file ('random patterns', exact copy) or reduce it up to four orders of magnitude (geometrical regularity, low degree of similarity to the original).

Barnsley also announced that the complete set of the IFS compression - decompression algorithms has been embedded into microchips thus enabling a real time response. This performance characteristic is essential when transmitting in real time images or animation sequences (e.g. videoconferencing) via any communications channel from the local storage bus to a Wide Area Network-ed host. Other image compression techniques that are frequently used include CCITT huffman, LZH, LZW, PKZIP etc. For a complete account of the various image compression schemata the reader is referred to Murray & vanRyper [1994].

1.7 The nature of light

In order to understand the foundations of computer graphics, the definition of *light* is necessary. What also needs to be defined is what a light ray consists of and what is meant by the word colour. In computer graphics, light is assumed to consist of an infinite number of closely packed rays (*light rays*) that can be represented as vectors in the three-dimensional space. Additionally, from physics, it is assumed that a light ray consists of 'packets of energy' called *photons*. The energy (E) that the photons carry is modelled as electromagnetic waves and relates to its frequency (f) as the following equation shows: $E = f \times h$, where h is the Planck's constant that is $h \approx 6.63 \times 10^{-34}$ Joules × seconds.

Another way to express the frequency f is by the wavelength λ. Frequency and wavelength are linked together with the following equation $f \times \lambda = c$, where c is the velocity of light in the medium that it passes through. For example, in a vacuum it is $c \approx 3 \times 10^8$ meters/second.

What we perceive as colour is the frequency of that electromagnetic wave when the photons that carry it hit the receptor cells on the retina of our eyes. When a photon hits our retina it gives off its energy. If this energy is approximately of frequency in the range of 360 - 830 THz (1THz = 10^{12} cycles per second), the receptor cells are 'tuned in' and stimulated, thus passing the appropriate signals (stimuli) to the human brain. In terms of wavelengths, the range of visible photons is between 360 and 830 nanometres (1nm = 10^{-12} meters). Mapping this range to the colours we actually perceive, what we see as red is near 360 THz, while blue is at the other end of the spectrum (i.e. 830 THz).

Another parameter that characterizes a light ray is the *intensity* it carries. Intensity of light at a given wavelength is a measure of the amount of photons of that wavelength that are travelling along that ray. It is expressed in terms of energy E and is measured in Joules. A plot that depicts the distribution of intensity over all visible wavelengths for a given light ray is called *frequency spectrum plot* or simply *spectrum* of that light ray. The spectrum of many light sources has been studied (e.g. CIE Standard Illuminant D6500 represents an approximation of the sun's spectrum on a cloudy day) since it acts as an indicator of the chemical composition of that source or of the media it travels through. Between individuals, the range of visible frequencies may vary, therefore, the limits of the visible spectrum we mentioned are to be treated as approximations. Moreover, the above theory for explaining the nature of light does not answer all the questions in physics, but it is still sufficient for most of the computer graphics applications.

1.8 Our installation

The rationale behind our research is to provide a visualisation system to aid researchers not only in the field of computer graphics but also in mathematics, physics, biology, to name but a few. Therefore it is essential to prove that all our methods can be implemented on a common purpose computer system and not on a very specialized, and therefore very expensive, graphics engine.

Our installation consists of one workstation, black & white and colour printers, a microfilm plotter and a microfilm recorder. The workstation is a general purpose IBM Personal System Model 95 XP with adequate memory (8 Mbytes RAM, 1.5 Gbytes disk) and a tape archiving system for long term storage and back up purposes. It is a single processor (Intel 80486DX 33MHz) system with an accelerated video sub-system (XGA). The XGA adapter is used on its high resolution 1024×768 mode and is capable of depicting up to 256 different colours on the computer screen at any time from a choice of $2^{16} = 65536$ different colour shades, using the RGB colour model.

Most of the software used in this dissertation is implemented in the Object Oriented C language extensions (C++) and in particular it is developed in the Borland C++ environment under the DOS operating system. Although some DOS memory extending facilities and other operating systems are available that provide excellent memory management functionality, great effort has been put to contain the complete code inside the 640K memory limits of the DOS operating system. Again the rationale behind this decision is portability along different small sized common purpose computer systems.

For the same reason, the software produces machine independent viewport commands, the *viewport primitives*, that are traced and recorded by implementations of the model display device driver. For every real hardware viewport a different model display device driver has been implemented in order to exploit fully the particular capabilities of the hardware involved. Specifically, there has been developed a device driver for the XGA and the VGA modes of the computer's monitor, one for devices that accept the Adobe Systems PostScript® language and one for devices that accept the Hewlett Packard Graphics Language (HP-GL®) for pen plotters. All these device drivers are interchangeable and all communicate with the main code via the common set of device independent viewport primitives.

Image files follow the raster format and are stored compressed with the run length encoding algorithm. Depending on the visualisation method used, and the operating system's limitations (DOS 640Kbytes barrier), some post-processing may be necessary. For example, images produced with the oct-tree method need to be temporarily stored in a *meta-language* format. This format is actually an encoding of the device independent graphics primitive commands directed to the model display device implementations (i.e. device drivers for particular hardware viewports). This meta-language file is subsequently processed in order to produce the raster image file necessary for the rest of our manipulations.

The use of paper printers is achieved via the PostScript® device driver which is capable of producing *Encapsulated PostScript® files (EPS)* holding the complete raster image in the form of a bitmap. The EPS files are realized either on a IBM 4029 PS black & white 600dpi (dots per inch) laser printer, or on a QMS100 dye sublimination colour printer. Where appropriate, the meta-language format is used to transform images from our *model display*

device language (i.e. set of device independent primitive commands) into the PostScript® or the HP-GL ones for the inclusion of draft sketches (usually wire-frames) into word-processed documents for further manipulation.

Furthermore, for higher quality hardcopies of our images, two film-based viewports are used, the Dicomed and the Montage. They are both able to handle 35mm film of ISO 100 resolution which effectively gives an area of 4096 x 2730 addressable pixels. The Montage recorder accepts raster images only. Vector graphics image formats are being rasterized at a pre-processing stage before they are mapped on to the film area. This facility is local, accepts among other formats Encapsulated PostScript files, and is sufficient for most of our needs.

The Dicomed D48C, however, offers the additional advantage of vector plotting, thus enabling us to explore the 'additive' behaviour of the photographic film. This equipment is attached to a complex of three CONVEX 220 supercomputers (under a dialect of the UNIX operating system) located at ULCC and is accessed via the X.25 network links of JANET. Here the appropriate device driver had to be written. It was implemented in the FORTRAN programming language in order to link to the DIMFILM library of subroutines. In order to avoid possible corruptions of the image files while being transmitted via the network links, and in order to achieve a file description independent of the character set differences between the computers used (ASCII vs. EBCDIC) it was essential that image files were converted in order to use only printable characters (i.e. alphanumeric) instead of their original pure binary form.

Another manipulation to the image files was the arbitrary change of their pixel resolution in order to match the particulars of a hardware viewport. This involved the use of interpolation techniques that enabled the transfer from the (orthogonal) grid of square pixels of the model display device to any other orthogonal grid of arbitrary density (pixel resolution) and aspect ratio (i.e. rectangular pixels). It is anticipated that such transformations may alter details of the original image, since a simplistic linear interpolation model is used [Tsoubelis 1985]. Nevertheless, such a facility has proved to be convenient while previewing a great number of archived images.

Chapter 2 Modelling in computer graphics

2.1 Introduction

The first phase in any computer graphics application, as mentioned in the previous chapter, is modelling, where the description of what needs to be visualised is constructed. The model description is then properly encoded (e.g. in a data file) in order to provide all the necessary information for the subsequent phase of visualisation.

In this chapter we will take an overview of the most frequently used approaches to modelling that have been implemented in the field of computer graphics. First, we will present and discuss the constituent parts of a model in terms of both necessary and optional characteristics. Then, we will discuss the two basic categories of model in the computer graphics domain. Finally, we will present a number of approaches to modelling that are considered representative in terms of their underlying philosophy, main characteristics, application areas, strengths and weaknesses.

However, before commencing our presentation of the various issues regarding modelling, it is essential that we give some definitions about the terminology that we will be using, in order to establish a common understanding with the reader. This will clarify the use of terms that, although they have been widely used in the relevant bibliography, often have been assigned a variety of different meanings. All these terms relate to the word *model* and our interpretations will be given in the following section.

2.2 Terminology - definitions

Since the aim of modelling in computer graphics is to construct the description of the required scene, and the most prominent feature is the shape of the objects in the scene, we will start our definitions with that of the *geometrical object.*

- *Geometrical object.* A primitive entity, such as a point, a line segment, a planar polygon, a circle, an ellipsoid, a cone, a cylinder, etc., as defined and used in all common geometry textbooks such as Euclid's Elements [Papanickolaou 1978].

- *Object.* The constituent parts of a scene. An intuitive term to describe a logical entity in the computer graphics scene (e.g. a ball, a table). With a few exceptions that will be discussed later in this chapter, we assume that a scene consists of objects, and that any object can be decomposed into a combination of geometrical objects that we will also call *primitives.*

- *Surface.* The locus of points that have a common property. For example, the surface of sphere is the set of all points in three-dimensional space that are an equal distance away from a fixed centre point. In some cases, a surface is also considered as the boundaries between an object and its surrounding space.

- *Solid object.* The volume of space that is enclosed by a 'closed' surface. We assume that the surface is also part of the solid object.

- *Scene model* or, *model.* The data describing the computer graphics scene. These include information about the geometrical properties of the constituent objects such as shape, size, and location in the scene. Additionally, they may include information about the colour properties of the materials that these objects are assumed to be made of, such as reflection and refraction parameters of all object surfaces, and any other supplementary data necessary for the visualisation algorithms.

- *Colour model.* The set of rules that permits the description of colour. The set of all perceivable colours define the *colour space.* There are a few colour models that are widely accepted. Depending on the colour model used a different subset of the colour space can be described.

- *Shading model.* The set of rules that permits the analytical description of optical phenomena. Optics define colour in a dual nature; electromagnetic wave, and quantum.

As a wave, colour is determined by a spectrum and as a quantum it is defined by a ray. For the purpose of computer graphics we also use the concept of perception from theories of psychology. A distillation of all these theories has resulted into a variety of shading models that describe the interaction of colour with matter.

- *Modelling*. The process of determining all the data needed to construct the model of a particular scene.

2.3 Characteristics of modelling

Using the above terminology, the purpose of modelling in computer graphics is twofold; first, it is the process of describing the necessary primitive geometrical objects and second, it is the description of the way these geometrical objects should be combined together in order to construct the required scene.

The process of defining the model of an object, however, is not straightforward. There are several types of objects that do not have a unique model. There are two important reasons; the first is that the model may not be an exact description of the object but an approximation to it; and the second reason is that different combinations of primitive geometrical objects may be used to construct the same object in a scene. Consequently, various degrees of approximation will result into different models of the same object.

Choosing therefore, a particular modelling approach entails considerations regarding the required degree of approximation to the object, and the type of primitives that a particular visualisation algorithm can handle. This choice of the appropriate degree of approximation (if any), the type of primitives and the suitability of their combination can be judged against a set of general criteria. Such criteria should not be treated as compulsory features that any modelling approach should conform with, but as a set of properties that very often aid the designer and improve the speed of visualisation. The following list presents some of the most frequently used criteria:

- *Generality.* A modelling approach should be *generic.* As such, similar objects should generate similar model descriptions. Similar objects are therefore described once, say in their SETUP position, and the scene's model is then constructed by instantiations of that generic forms. For example, all 'boxes' in a scene may be described (i.e. modelled) as parallepipeds and their description should differ only in the values of the parameters describing the boxes' exact dimensions and surface properties.

- *Controllability.* A modelling technique should offer as many degrees of freedom as possible. Such a property will enable the designer to adjust the modelled shapes both globally and locally [Barsky 1981; Forsey *et al.* 1988]. This property complements the property of generality.

- *Invariance.* By applying a transformation (such as those introduced in the first chapter) to specific points (usually possible centres of symmetry or other control points) the whole shape should be transformed. Such a property would significantly simplify and enhance generality and controllability and can be achieved via appropriate SETUP to ACTUAL transformation matrices. There are transformations, however, for which this property is not 'always valid', and where 'intuitive' adjustments are necessary. Take, for example, the application of Minkowski operators on polygons detailed in chapter four. The M-difference does not always produce a polygon, and a post-processing stage needs to be followed after the M-difference operation.

- *Continuity.* A surface should be (at least) geometrically and/or analytically continuous [Barsky 1984]. Although it is not essential, such a property would ease the calculations needed for the determination of the vector perpendicular to any point on the surface (i.e. the *normal vector*). Moreover, since most of the visualisation algorithms are based on the assumption of surface continuity, discontinuous surfaces should be treated with care.

- *Bounding volume information.* Information about volumes bounding the modelled object should be easy to calculate. The tighter a *bounding volume* (or *extent*, or *enclosure*) fits the modelled object, the better such spatial information can be exploited [Whitted 1980]. Such an observation aims at improving the efficiency of various visualisation algorithms.

In certain implementations of such bounding volume acceleration techniques, the process of visualisation may be improved by an order of magnitude or more. [Weghorst *et al.* 1984; Kay *et al.* 1986; Arvo *et al.* 1989].

- *Ease of computation.* This is an all encompassing title for all the issues regarding the difficulties encountered during the implementation of any modelling approach. Relevant issues are the exploitation of recursiveness [Meagher 1982] and parallelism [Dippe *et al.* 1984; Kobayashi *et al.* 1987; Nishimura *et al.* 1983], the re–usability of pre-calculated values and the exploitation of bounding volume information [Nemoto *et al.* 1986].

It can be observed that the above list only covers issues that refer to the geometry of the models. However, in most applications, the model also contains information about the colour properties of the surfaces of the modelled objects; such information is necessary for the rendering algorithm during the visualisation phase.

For a typical computer graphics application, where 'photorealism' is a high priority, the rendering algorithm should simulate a number of optical phenomena such as specular and diffuse reflection, shadows, penumbra, ambient illumination, light refraction, radiosity, etc. The set of optical phenomena that a computer graphics application will simulate constructs a *shading model* that the rendering algorithm will have to implement. Depending on the complexity of the shading model and the colour model used, a number of different parameters will have to be defined for all the surfaces of the modelled objects.

For example, the colour of a surface may be defined by the proportion of the light this surface reflects. In order to simplify the rendering algorithm, these proportions are not measured in all frequencies of visible light but they are sampled in a few frequencies only, usually the primary components of a convenient colour model. For the RGB colour model, the proportion of reflected over incident light on a particular surface will be measured for the pure red, green and blue light only. These proportions will normally depend on the direction of the incident light and the viewing direction. With a few exceptions found

elsewhere,[1] such details are roughly approximated since they put a heavy strain on the computational demands of the computer system. Other parameters involved in the implementation of a shading model may include the *shine*, *gloss*, *degree of transparency*, *index of refraction*, colour and intensity of the *ambient light* etc.

2.4 The nature of models

So far we have seen that the constituent parts of a model give a *static* description of the geometrical and optical properties of the modelled object. Furthermore, we implicitly made the assumption that we know exactly the shapes and colours of the modelled objects. However, these observations may not be true in all applications and for all the objects we may need to model: there are objects like the sea waves, clouds or fire, that do not have a specific form (i.e. geometrical shape) and any instantiation of their appearance may suit our purpose. Moreover, there are objects that we cannot describe analytically, but we know a way (i.e. the procedures) to construct them. Apart from geometry, the optical properties (e.g. colour, reflection, transmission properties, etc.) of the surfaces of the modelled objects may also vary, as is the case of the textured surfaces of objects like marble, wood, textiles, etc.

Therefore, a more thorough investigation of the different types of objects and models is necessary. As a result, we will call an object *deterministic* or *stochastic* depending on the nature of its properties that describe its shape and its optical behaviour. With regard to a given property, an object will be called *deterministic* if this property's value has to be uniquely determined (by a formula or process) and *stochastic* if its exact value is not important but an instantiation of it is sufficient for the application. With regard to models, we will call a model *static* if the data describing a particular property of an object are known to us, *constrained* or, *constraint–based* if the state of that property can be determined by a set of given constraints (e.g. gravity, geometrical locus, etc.) or *procedural* if the state of that property is not known but can be calculated via a known procedure.

[1] The shading model of luminaire design software that determines the light flux distribution in three-dimensional space takes into account the exact physical properties of the light sources and the materials used for the appropriate reflectors [FiELD 1992; Ward 1994].

In order to understand this binary classification of objects and models, we will present in this sections a few representative examples for each of the above categories. We will start with objects with a deterministic shape and examine the different types of models we can have. Then, we will look at stochastically shaped objects and their resulting models.

Deterministic objects

- Static model: a cube of given dimensions can be modelled as parallepiped. The only information we must include in the model is the length of its edges. Alternatively, the cube may be built by a set of polygonal facets and therefore information about its constituent vertices has to be included in the model.
- Constrained model: a sphere can be modelled as the locus of all points in three–dimensional space that are equidistant (i.e. the radius) from a given fixed point (i.e. the centre). Another more complex example would be the shape of a table-cloth covering an uneven surface. Here, the exact location of the table-cloth has to be calculated by the physical properties (*physically-based modelling*) of the cloth (i.e. weight, density, elasticity, etc.) and this information will be included in the model for the subsequent phase of visualisation. It follows that such information should be determined by applying the appropriate laws of physics (e.g. gravity). Physically-based modelling is already a recognised research area within computer graphics and the reader is referred to Terzopoulos' [1989] inelastic (plasticine like) objects that get permanently deformed after a collision, or Barr's [1989] chains that are affected by gravity forces.
- Procedural model: a finite cylinder is produced by intersecting an infinite cylinder with two half–spaces perpendicular to the cylinder's axis. Here the emphasis is on the construction of the shape of the object by using surface generation procedures like that of extrusion, rotation, envelope, etc.

Stochastic objects

- Static model: a 'close–up' view of a rough surface (e.g. a wall). If we look from a distance, a wall looks like a flat surface which could be modelled by an appropriately shaped planar polygon. However, a 'close-up' view of a wall will reveal its roughness. But 'anomalies' on this wall surface are not distinguishable. Consequently, the model of such a rough surface need not be, and cannot be, an exact description of all the observed

'anomalies'. A random generation of a polygonal mesh that resembles a similar degree of roughness would therefore suffice to model this surface.

- Constrained model: a, for example, fractally produced ivy as it grew over a particular fence structure. The constraints here come from the interaction of gravity (the flower is unable to sustain its weight) and the geometry of the fence. Here the randomness of the object(s) will be simulated by pseudo-random number generators, therefore some control parameters (e.g. seed, magnitude, random proportion) have to be supplied. As a result, the main characteristics of an object will be determined by the constraints, but the details will be randomly chosen.
- Procedural model: the object produced by the random displacement of three-dimensional points from a given flat polygon. Another example would be the three-dimensional object that is produced by the rotation of the Mandelbrot Set with given parameters (i.e. initial values, threshold value) around its imaginary axis.

The following table (Table 2.1) summarises the above examples on a 2 by 3 matrix.

Objects / **Models**	**Deterministic**	**Stochastic**
Static	box – parallepiped	wall – random polygonal mesh
Constraint–based	sphere – geometric locus	ivy – controlled random growth
Procedural	cylinder – set theoretic operations	random object – random displacement

Table 2.1 Examples of the object – model classification schema

The discrimination between constrained and procedural models of stochastic objects is somewhat analogous because procedural models imply the use of pseudo–random number generators that have to be controlled (i.e. constrained) by a set of defining parameters (e.g. seeds). Another difficulty that emerges from this classification arises from the observation that there may be more than one model for describing the same object. Consequently, the same deterministic object may have both constrained and static models. In the following example we will illustrate how the deterministic object 'ring' (also called torus or, doughnut) may have models in all three categories (i.e. static, constrained, and procedural).

A ring is defined by the formula $(x^2 + y^2 + z^2 - (a^2 + b^2))^2 - 4a^2(b^2 - z^2) = 0$. This formula will construct a static model of the ring. However, the locus of three-dimensional points that are equidistant from a given circle, will also define a ring. Such a model is constrained and is described as: $\{p \in \mathbb{R}^3 \mid \|p - circle\| = c\}$. Furthermore, by revolving a circle around a circular trajectory we also define a ring. This model of a ring is procedural and consists of an accurate description of the construction process.

Following analogous steps, we can conceive a similar schema by examining shading models and the way light interacts with the surfaces of objects. For example, *deterministic shading* may result in *static*, *constrained* and *procedural shading models.*

- Static shading model: the colour of a surface is assumed to have a particular value irrespective of any illumination sources in the scene. This assumes a trivial shading model using fixed colours and is frequently used in applications where the number of available colours on the viewport is extremely small (e.g. VGA mode 18 offers 16 colours only).
- Constrained model: the colour of a surface is determined by its primary colour but is adjusted in order to simulate optical phenomena like diffuse and/or specular reflection, transparency, etc. Shading models like the *Gouraud* and *Phong shading* belong to this category and will be presented in detail in the next chapter.
- Procedural model: the colour of a surface is determined by the mapping of another image (usually in the form of a *bitmap*) onto that surface. Another example of a procedural model for shading is the introduction of random colour perturbation.

In an analogous way, *stochastic shading* describes shading models that accept as an input parameter a (pseudo-) randomly chosen primary colour. This type of shading does not produce 'intuitive' results. An example of a stochastic shading model is, in certain circumstances, that of *false colouring.*

2.5 Approaches to geometrical modelling

In this section, we will take an overview of the existing approaches to modelling, that have been implemented in the field of computer graphics. For each one, apart from the description of its main characteristics, we will discuss its advantages and disadvantages, and we will categorize it following the previously described model classification.

In geometry, there are two basic methods for describing a curve or a surface, namely *classification* and *enumeration.* With both methods the underlying assumption is that an object (curve, surface, etc.) is represented by the set of it constituent points, (the locus of which describe the surface). Therefore, the task of defining a shape, is expressed as the task of determining the set of points (in the appropriate n–dimensional space) that the particular object consists of.

As a result, in the classification method, the object's constituent points are determined by an appropriate function ($F(p)$), that for a given point p determines whether that particular point belongs to the desired object or not. According to Hanrahan [1989] the description of an object is determined by a *point-membership classification function (PMCF).* This is either a formula or a procedure that decides whether a certain point (input) is *inside*, *outside* or *on* the surface of the required object (output) :

$$F(p)\begin{cases} < 0 & \textit{inside} \\ = 0 & \textit{on} \\ > 0 & \textit{outside} \end{cases}$$

However, since there are objects where the meaning of 'inside' and 'outside' is not well defined, we would suggest a better definition: in the classification category, there exists a mathematical *test* (T) — which may be either a formula or a procedure — that the locus of the points p that evaluate the *defining test* function $T(p)$ to zero, define the required object (usually a surface if no degeneracies occur). Since a point is not known to belong to an object unless the above test has been performed, objects generated with this method are also called *implicit.* If the above test is expressed as a mathematical function, then according to the description of the classification function T, objects can be called *algebraic* if T is expressed by polynomials (of a finite degree) only, and if T is a differentiable (i.e. smooth) function, they are called *analytic.*

For example, the perimeter of a circle of radius one unit in two–dimensional space may be defined analytically by the classification test: $(T(p)=0) \equiv (x^2+y^2-1=0)$, where x and y denote the circle's coordinates using a two–dimensional Cartesian coordinate system imposed on the two-dimensional space. In this example the analytic function $x^2 + y^2 - 1$ is used to determine the test $(T(p)=0)$ that will distinguish all points that evaluate it to zero.

The second mathematical method for object description, namely enumeration, assumes that all the constituent points of an object can be generated by mapping to them a set of parameters. Hence this method is also called *explicit*, or *parametric*. Such an assumption implies that there exist both a set of input parameters with certain values (or range of values) and an appropriate mapping function (or functions) that calculates the coordinates of all the points of the required object. In a more precise manner, a three–dimensional surface will take the form: $(x(u,v),y(u,v),z(u,v))$ where x, y and z are independent mapping functions of the parameters u and v. Therefore, knowledge of the mapping function and of the ranges of the values of the input parameters is sufficient for us to determine an object.

For example, the perimeter of a circle of radius one unit, in two–dimensional space, may be defined explicitly by one input parameter φ and the following functions:
$(x(\varphi) , y(\varphi)) \equiv (\cos(\varphi) , \sin(\varphi))$ when $\varphi \in [0,2\pi)$.

From the above example, it becomes obvious that both methods may be used to determine the same object. However, as we will see in the next chapters, there are objects that can be described by a test function (i.e. classification) but they may be too complex to be described by their equivalent enumeration function. The choice of the appropriateness of each method may be judged against the set of criteria we presented earlier, and the particular requirements of the application.

In the next subsections, we will present the most significant modelling approaches in computer graphics. They will be classified according to the category in which they are most often used (i.e. enumeration, classification). We will begin with the enumeration category and their main representative *interpolation*. Then, representatives of the classification

category will follow starting with *polygonal mesh*. Then, the approaches of *analytic functions* and that of *volumetric arrays* will be presented. Finally, another pair of approaches, namely *constructive solid geometry* and *procedurally defined surfaces*, that are used to combine object descriptions from any other modelling approach in order to build more complex ones, will be discussed.

2.5.1 Interpolation

Interpolation methods have been studied extensively for many purposes. Application areas includes image processing, remote sensing, ship building, metallurgy, etc. Depending on the requirements and the assumptions of a given problem, a great variety of interpolation techniques exist. Apostolatos [1981] discusses the fundamental mathematical theory behind Lagrange, Everett and Tschebychev interpolation techniques. One approach to the problem of interpolation that is identified with computer graphics is *splines*. It originated in the ship building industry, where one of the main tasks of the builders was to determine the curvature of metal arms that would connect together the basic skeleton of a ship.

With regard to splines, a typical interpolation problem in two dimensions is expressed as follows: given a sequence of *control points* (in two-dimensional space), define (a function that produces) a curve which passes through or nearby these points. It would be desirable for this function to have minimum curvature and be analytically continuous over the interval where the points are defined. In three–dimensional space, the equivalent of the control point sequence is a lattice arrangement called *control grid* and the required curve becomes a smooth surface that passes through or near that grid.

With regard to modelling in computer graphics, the interpolation problem is expressed in a slightly different manner. What is required is a shape (usually a continuous curve or surface) that needs to be approximated. Therefore, the user will have to determine the desired (enumeration) function that approximates the given shape. As a result, it is the user's responsibility to define the appropriate control points (or grid) that, when interpolated, will determine the required function. The guessing of the appropriate control points is aided by the following:

- The ability of creating surfaces by connecting together little patches (*patchwork*) [Coons 1964; Coons 1967; Forrest 1972].
- The extensive control that current interpolation techniques offer. Specifically, there is
  - *Local control by point displacement*, where shape alterations may be achieved in small areas of the produced surface by displacing the nearest control point(s). This is the case of most spline techniques like *B-splines* [Barsky 1984], [Schoenberg 1946; Carry & Schoenberg 1947; Carry & Schoenberg 1966], *Bézier splines* [Bézier 1972; 1974; 1977], *non-rational B-splines* [Riesenfeld 1973], etc.
  - *Local control by weight adjustment*, where shape alterations of small areas of the produced surface may be achieved by the weight factor of the nearest control point(s). This is the case of *rational B-splines* [Versprille 1975; Tiller 1983].
  - *Local control by bias and tension*, where local alterations of the degree of bias (or symmetry) and the amount of symmetric tension applied on a surface may be achieved by the adjustment of θ_1 and θ_2 (bias, tension) of the *Beta-splines* [Barsky 1981; Barsky *et al.* 1982; Barsky *et al.* 1983].
- Fast algorithms for evaluating splines like, for example, the Cox-deBoor [Cox 1972; deboor 1972] recursive algorithm.
- Customized types of splines that show specific properties. The main family of such splines was introduced by Catmull and Rom [1974], where the control points were replaced with (control) functions. The type of splines used, and the type of the control functions will predicate a set of properties on the resulting surface (or curve). Barry and Goldman [1988] showed how Lagrange interpolation polynomials can be used as control functions, and presented a recursive algorithm for their evaluation.
- Advanced spline editors that allow real time editing of spline curves and surfaces.

With this approach, a shape is described by a (interpolation) function, or a combination of more than one function (i.e. patchwork), together with the necessary parameters (i.e. the arrangement of the control points, and/or their weights, bias, tension, etc.). For example, a circle may be defined by a B-spline using as control points the corner and midpoints of the circle's circumscribing square. Accordingly, in three dimensions, the corresponding sphere may be obtained by a B-spline and a control lattice as defined by the vertices and the midpoints of the sphere's circumscribing cube.

2.5.2 Polygonal mesh

Apart from functions, surfaces can be described using building blocks to construct them. Surfaces are approximated by a 'patchwork' of simple planar geometric objects namely, *flat polygons*, or *facets*. The reason for classifying this approach under the 'implicitly defined shapes' category is that it is based on the assumption that a polygon is described implicitly; a polygon is assumed to be the locus of co-planar points that fall inside its edges.

This implies the existence of the appropriate *point in polygon* classification tests. Two such tests are the *non-zero winding number rule* and the *even-odd rule* and are presented in the [Postscript® 1987] reference manual. Both techniques assume that the vertices of the polygon have been named in such a way that there exist an order with which all vertices can be visited by a given algorithm. Assuming a two-dimensional space, according to the first technique, an infinite line that passes from the given point is (conceptually) drawn with any direction not parallel to any of the polygon's edges. Then starting with a counter of zero, we add one (1) if an edge crossed that line from left to right, according to our pre-determined naming convention, and subtract one for opposite direction (right to left) intersections. If the final count is zero the given point is outside the polygon else, inside. In we illustrate how this rule is applied for the points ***p*** and ***q*** and canonical pentagon and a five pointed star shaped polygon with its edges intersecting each other respectively.

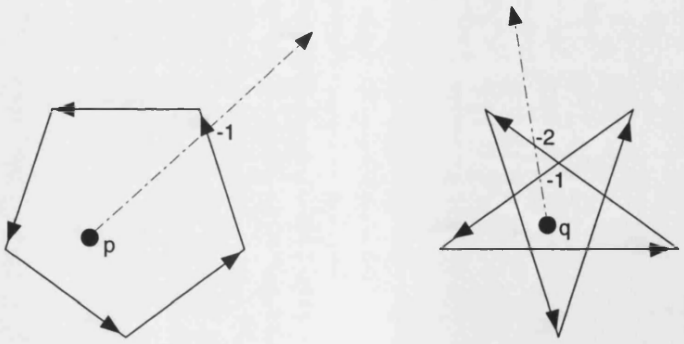


Figure 2.1 The non zero winding number rule

According to the second test, (the odd-even rule), an infinite line is again drawn over the point in question. Then we count the number of times that this line intersects with the polygon's edges. If the total is an odd number the point lies inside the polygon else, it is outside. illustrates how we apply this rule for the same objects as we did in the previous example (). Note here that the simple case of the convex pentagon gives the same results, however, the case of the five pointed star polygon does not.

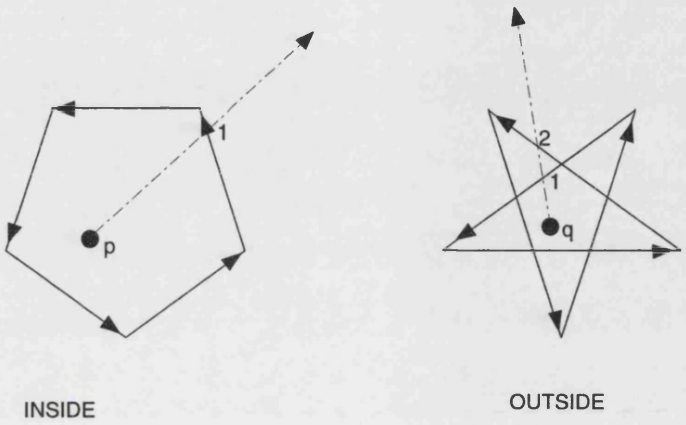


Figure 2.2 The even-odd rule

Invoking the above point-in-polygon tests on flat polygons in three-dimensional space, however, poses an additional challenge; the transformation into the equivalent two-dimensional problem. This entails the determination and use of a coordinate system that has (say) its *X-Y* plane coincident with that of the polygon, and its origin coincident with that of the point in question.

For our convenience, and without affecting the applicability of the polygonal mesh approach, we found it effective to make the following two assumptions. The first regards the use of convex planar polygons only, since the use of concave ones complicates the implementation of the point-in-polygon test. If it is necessary to use concave ones we decompose them into a set (patchwork) of smaller convex ones.

The second assumption regards information about the orientation (i.e. the 'front' and the 'back' face) of such a facet. Such information is embedded in the model by assuming a consistent way of ordering the vertices of all the facets. For example, the 'front' (or outside)

face of a polygon is described by noting its edges counterclockwise. The exact ordering is not important as long as it is consistent in all the facets of the model. This ordering will enable the visualisation algorithms to decide whether a surface is visible or not and consequently ease, or possibly avoid, the determination of the point-in-polygon test.

Polygonal mesh models can be visualised extremely quickly (rates of thousands of polygons per second are common) because of the simplicity of the building primitive used (polygons). Moreover, many intersection finding and rendering algorithms have been translated into microcode and put into the hardware (VLSI chips) thus further increasing visualisation speeds.

But polygonal mesh has some drawbacks as well:

- The approximation of shapes with high curvature necessitates the use of large numbers of tiny polygons, thus considerably increasing the size of the model description and implying the need for large databases to store them.
- Since an object is approximated by a patchwork of planar polygons, the notion of curvature is lost. Therefore, it is the responsibility of the visualisation (rendering mainly) process to remedy this. Algorithms like *intensity interpolation shading* [Gouraud 1971] and *normal vector interpolation shading* [Phong 1975] reduce but do not eliminate the problem.
- The joints of the patches (polygons) do not produce pleasing images (i.e. smooth surfaces). As a result, a jagged polygonal silhouette appears.

Figure 2.3 Various degrees of approximating a sphere

Despite the above mentioned drawbacks, polygonal mesh is the most common approach used for modelling in computer graphics because of its simplicity, its visualisation speeds and its capability to approximate (to a certain extent) any conceivable surface.

As Figure 2.3 shows, the closer to the required surface we approximate, the more facets we need to patch together. In Figure 2.3 we see three different approximations to a sphere using the polygonal mesh approach. Starting from the coarser one, we used 100, 400 and 900 polygonal facets.

2.5.3 Analytic functions

The use of analytical functions as a modelling approach is the second most significant (after the polygonal mesh) representative of the classification category. Recalling our definitions, objects are defined with a point-membership classification test which is usually in the form $T(p)=0$ for a function $T(p)$, $\forall p \in \mathbb{R}^n$. Such a function T may be an algebraic function such as a quadric, quartic, superellipsoid, superhyperboloid etc. In general, an algebraic surface in three dimensions is represented by the following function:

$$T(x,y,z) = \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} a_{ijk}\, x^i\, y^j\, z^k$$

where $i,j,k \in \mathbb{Z}$, $\forall\, i=0,\ldots,l$, $j=0,\ldots,m$, $k=0,\ldots,n$, and its degree is defined to equal to the sum $l+m+n$. For this function T, we can generate a point membership classification test as (Eq. 2.1) shows.

$$T(x,y,z) = 0 \qquad \text{(Eq. 2.1)}$$

The following table (Table 2.2) shows some examples of the function tests that describe the most common three-dimensional geometrical objects.

sphere	$x^2 + y^2 + z^2 - 1 = 0$
cylinder	$x^2 + y^2 - 1 = 0$
cone	$x^2 + y^2 - z^2 = 0$
paraboloid	$x^2 + y^2 + z = 0$
hyperboloid	$x^2 + y^2 - z^2 + 1 = 0$
torus	$(x^2 + y^2 + z^2 - (a^2 + b^2))^2 - 4a^2(b^2 - z^2)$

Table 2.2 Analytical function tests for simple geometrical objects

The use of function tests for the definition of an object (surface, curve, etc.) enables us to produce *families* of similar objects according to the following observation. Given a function, say f, the test that can be generated by the equation $f(p)=0$ will define a set of points $p \in \mathbb{R}^3$ in (say) the three–dimensional space that will construct an object. However, it is not necessary that the second part of this test is always set to zero (0). A more general definition would be:

$$f(p)=s,\ s \in \mathbb{R} \qquad \text{(Eq. 2.2)}$$

Such a test is equivalent to the $f(p)-s = 0,\ s \in \mathbb{R}$. Therefore, by 'off-setting' the initial function f we may generate an infinite number of 'similar' tests. Thus for every possible value of the real parameter s a new object (i.e. set of points) is produced. As a result, the same function f will produce a family of similar objects by adjusting the variable s as equation (Eq. 2.2) shows. Following the notation of equation (Eq. 2.2), for the rest of this document whenever the parameter s is not mentioned it will be implied that the defining function f assumes that $s=0$ and the resulting object is created by the point-membership classification test: $f(p)=0$.

Algebraic surfaces that intersect each other may also be *blended*. Take, for example, two surfaces defined by the functions f_1, f_2 and the families that they generate $f_1=s_1, f_2=s_2$ for all the possible real values of s_1 and s_2. The blending function g that blends between the surfaces f_1 and f_2 is denoted by $g(f_1,f_2) = g(s_1,s_2)$ and is expressed as a function of the parameters s_1 and s_2. Hoffman and Hopcroft [1985] suggest that a desirable blending function should intersect both surfaces f_1 and f_2, be a tangent to these surfaces along the

curve of their intersection and be smooth between these curves. Hanrahan [1989], suggests the ellipse as an suitable blending function:

$$g(s_1,s_2) = \frac{(s_1-a)^2}{a^2} + \frac{(s_2-b)^2}{b^2} - 1$$

where a,b are the values of s_1, s_2 when f_1 intersects f_2.

Another way of combining intersecting algebraic surfaces is by using *homotopies* like:

$$h(f_1,f_2) = z^2 f_1 + (z-1)^2 f_2$$

for all the real values of $z \in [0,1]$.

Splines may also be seen as analytic surfaces, but since it is the control points that determine the functions, we prefer to classify them in the enumeration definition type.

Other analytic surfaces are the *superconics* and the *superquadrics*. Blinn [1982], has used *blobs* made of superimposed density distributions like:

$$f(x,y,z) = \sum_{i=0}^{n} b_i \, exp^{-d_i} - T$$

where b_i is a weight assigned to a nucleus point i, d_i is the distance of the general input point (x,y,z) from the nucleus i, and T is an arbitrary threshold value assigned by the designer in order to control the extent of the generated blob.

2.5.4 Volumetric arrays

Defining the shape of an object can also be achieved by dividing space in small units and then describing what exists in each unit. Specifically, in three dimensions, an object is assumed to be surrounded by a cube. This cube is subdivided into $n \times n \times n$ subcubes of equal size called *voxels*. Each voxel, is then assigned a number that represents the proportion of the voxel's space that is filled with points from the object. That number is named the *density* of the space that a voxel contains or, in short, the *voxel's density*.

Thus the description of an object becomes a three–dimensional array that represents the object in terms of density values. In the literature, this modelling approach has been misleadingly named by several authors as *N-dimensional arrays*, because of the software programming conventions for declaring arrays (e.g. Array[n][n][n]).

It follows that the larger the number n (or *sampling rate*) is, the better the approximation to the object's shape becomes. On the other hand, however, the higher the sampling rate, the larger the description of the object becomes. Moreover, the process of volume subdivision becomes significantly longer as well. This is particularly important in medical imaging applications where a patient, injected with radioactive or other toxic substances, has to stand still for a lengthy period of time (usually one to two hours) in order for parts of the patient's body to be scanned. Therefore a trade-off between the sampling rate and the quality of the resulted image has to take place and this is usually resolved with the scanned model being sampled at $256 \times 256 \times 256$ voxels.

In general, volumetric arrays are treated as density functions and visualisation algorithms are used to depict iso-surfaces. But in some cases, certain details may be hidden inside the visualised iso-surfaces. For this reason, cross-sections at given angles of intersection should also be produced by using interpolation techniques. In general, a set of parallel planes is intersected (at a given orientation) with the complete volumetric array, in order to produce another volumetric array description of the same object. In this way, different images of the same part of the human body may be superimposed to each other in order to highlight their differences and enable further analysis of the visualised object.

For the purpose of interpolation we must therefore examine the range of permissible values we may use to represent densities. In the most simple case, we use two values only; the voxel either intersects with the object, denoted by 1, or it does not and we assume a density value of 0. In a typical medical imaging application, however, the density values may take any real value in the continuum [0 , 1]. This variety of density values is used to identify different matters in the human body (e.g. blood, bone, tissues, blood vessels).

During visualisation the necessary interpolation technique will be adjusted according to the range of permissible density values. For example, if we use binary density values (either 1 or 0) then the density values that after interpolation are above 0.5 (> 0.5) must be changed to 1. Similarly, density values found below 0.5 (≤ 0.5) are changed to 0.

Density values may also be used to effect the shading process (i.e. *colour-coding*). In this way, areas of interest (denoted by a range of density values) may be rendered with a different colour in order to enhance the contrast of the resulting image. These ranges of similar density values are then called to act as *thresholds.*

In the simple case of a binary set of permissible density values, various data compression schemata may also be used, thus allowing high sampling rates of $1024 \times 1024 \times 1024$ to be implemented at reasonable memory demands. The most common method for such data compression is the *octree encoding* [Meagher 1982; Doctor *et al.* 1981]. According to this technique, a region of space is called *homogenous* when it contains the same material characterised by the unique threshold value,[2] and *heterogenous* otherwise.

The starting assumption for this data compression algorithm is that a cubical shaped volume of space encompasses the whole object. This cubical space is usually aligned along the axes of an orthogonal coordinate system. If this cube is heterogenous, it is subdivided along the axes into eight equally sized subcubes called *octants.* The same division process is recursively applied to all heterogenous octants. The subdivision ends when the size of the produced octants becomes smaller than a certain limit (i.e. the size of the voxel) or when all octants are homogenous.

The results of these subdivisions are encoded into a data structure called *octree,* which is the compressed model of the object. This is a tree where each node may have up to eight children. Child expansion implies heterogenous octants (nodes). Therefore, the terminal nodes (homogenous voxels) are used to hold the density value of their corresponding area.

[2] Density values are coded in relation to the single threshold value (either above or below).

2.5.5 Constructive solid geometry

This approach is used to construct objects by performing operations from 'set theory' on a collection of simple geometrical objects that are usually called *primitives*. The main operators used are the *union* ($\cup$), *intersection* ($\cap$) and *complement* ($-$). The primitive objects are treated as sets of points and the aim of this approach is to build the scene, using the above operators.

For example, a hollow sphere is created by subtracting a solid sphere (i.e. taking the intersection of the complement) from a another concentric solid sphere but with larger radius. Another more complex example can be seen in Plate 1, where the rear suspension of a car has been modelled using only three primitive geometrical objects; the halfspace, the infinite cylinder and the infinite helix.

The combinations of functions required to generate an object can be represented in the form of a binary tree since all operators are either binary (i.e. union, intersection) or unary (i.e. complement). This *building tree*, therefore, has all its non terminal nodes holding set operators and the terminal ones holding sets of points corresponding to the primitives or their complements. This method may be used recursively, therefore primitive objects can become objects that have also been generated using this approach, based on other simpler primitives. The model description of an object using constructive solid geometry would therefore consist of the building tree and the description of the primitives.

2.5.6 Procedurally defined surfaces

Another common approach to defining objects is to determine a skeleton or an outline of them, and apply to them a procedure such as *extrusion*, *rotation*, *sweeping* etc. Extrusion is the method of assigning an extra dimension to a given object. Usually, a two-dimensional object is being used as the outline of the extruded three-dimensional one. For example, Figure 2.4 shows the outline and the extruded object that represents the letter C. Observe here that the outline of the object (i.e. letter C) is a concave polygon that was broken up into 18 convex facets. An extension to extrusion is the *translational sweeps* where a planar

curve is translated along a straight axis to produce a surface. By altering the radius of the curve while translating it, or, by using non straight axes for the sweeps a more general method is constructed.

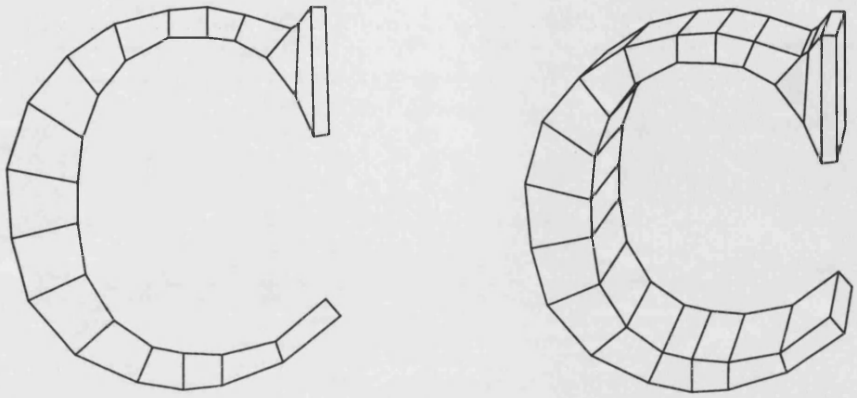


Figure 2.4 The outline and the extruded letter C

Another general method is the production of the *generalized cylinder*. Its surface is defined by sweeping a planar curve along a trajectory in three-dimensional space. The location, size and orientation of the curve, in relation to the trajectory need to be determined. Moreover, by translating three–dimensional objects along a three–dimensional trajectory even more complex surfaces are defined [Faux *et al* 1979].

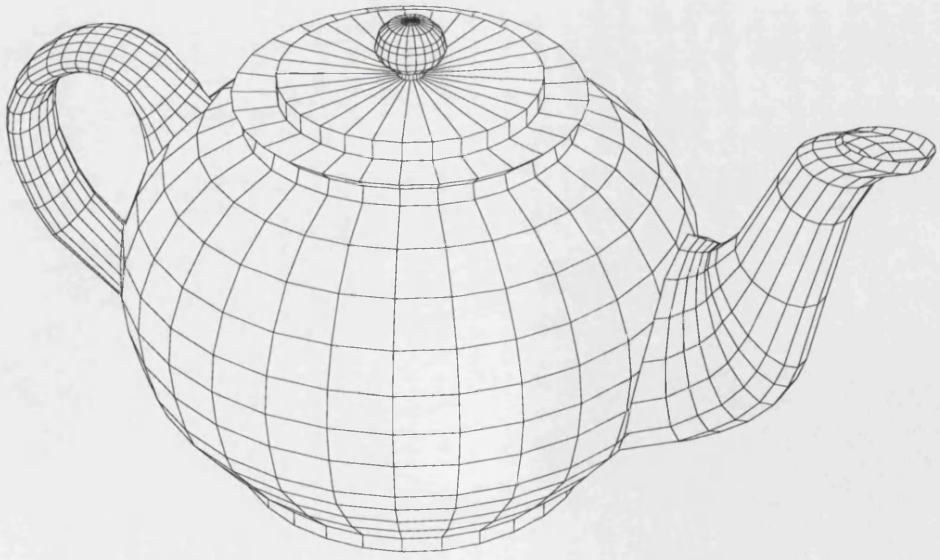


Figure 2.5 A teapot

Figure 2.5 shows a teapot that was generated by rotations and translational sweeps. The body of the teapot was produced as the body of revolution of given outline, while the neck and the handle as translational sweeps along the two–dimensional trajectories as shown in Figure 2.6.

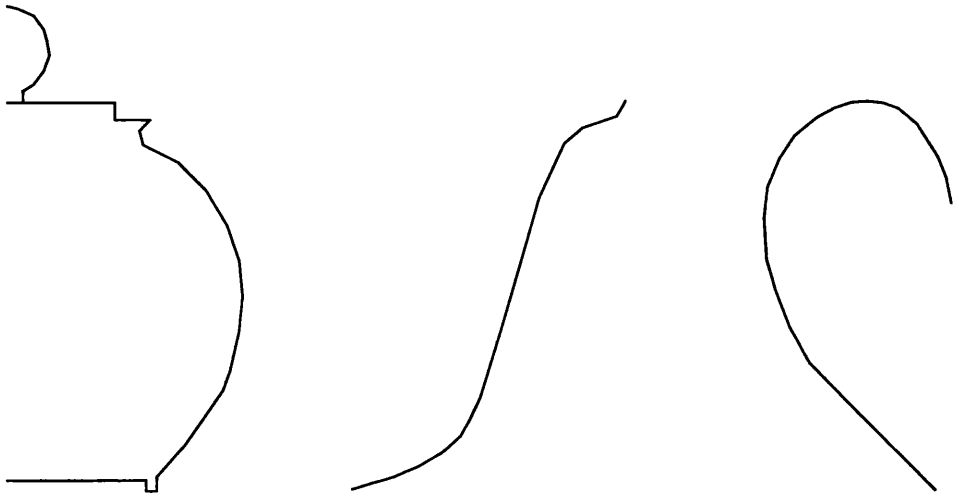


Figure 2.6 The major axes of the body, neck and handle of the teapot

Surfaces (bodies) of revolution are another type of procedurally defined surfaces. Here, a shape is defined as the envelope produced by a curve which is rotated around an axis. Methods like *strip trees* [Kajiya 1983] and *stacked cones* [Bier 1983] are used to manipulate these shapes.

Wijk [1985] studied shapes that had been defined by sweeping spheres. The method of *polyspheres* [Pickover 1989] has also been used extensively. According to Pickover a polysphere is "n spherical surfaces at given centres with specified radii". Therefore, the problem of tracing a sphere that moves along a certain trajectory may be translated into determining the union of space defined by spheres centred along that trajectory. The spacing of the spheres will determine the accuracy of the representation. We believe that the polysphere method may be expanded to model surfaces produced by sweeping other geometric objects as well. Plate 2 shows an object produced by the method of polyspheres.

2.6 Discussion

The classification of each modelling approach as static, constrained or procedural, was intentionally avoided since these approaches can be used to produce more than one type of model. For example, the approach of polyspheres may be embedded in a visualisation algorithm and the model will become the description of a procedure that needs to be used to construct an object (procedural model). Alternatively, the polyspheres approach may be used as the basis of a triangulation process to generate a polygonal mesh, hence a static model.

However, despite the above mentioned difficulty of classification, such a schema provides us with a framework for evaluating modelling approaches in computer graphics. This will help us understand the underlying principles behind the various modelling techniques.

For example, as we will see later, in chapter four, there exist a number of different techniques in literature that all have the same basis, namely the generation of *iso-surfaces*. All these techniques are presented in the literature as implicit, therefore, in our classification, they are constrained-based. But there are some details which we should observe with greater attention. These relate to the actual form of data that feed the visualisation algorithms, which in our definitions is the model of the scene. And in many cases what is provided for visualisation is a polygonal mesh that approximates to the required iso-surface, thence a static modelling approach.

But also the rest of these techniques, that do not use polygonal meshes as their intermediate stage, are still not free from false claims. A large majority of these techniques use splines or other similar analytic functions in order to approximate to the required surfaces, thence they are static modelling approaches.

The observation we make is very subtle but it clearly changes the perspective for assessing and classifying modelling approaches. The most effective argument, counter to our claim, that we can construct would be that: 'a technique that visualises implicit surfaces should belong to the constrained category of modelling'. In other words, where do we give

emphasis: to the type of the surface we intend to visualise or, in the actual form of data that are supplied to the visualisation algorithm?

We believe that it is the structures of data for describing an object that determine the nature of a modelling approach. This is consistent with the rest of our definitions and also agrees with a major segment of the literature. Besides, what we will eventually visualise will be a representation of the data structures that are supplied to the visualisation algorithm, which would approximate to the required object. As a result, the most accurate way to classify a modelling approach is to examine the visualisation approach that it intended to be paired with. This will shed light onto the nature of the data that will eventually drive the visualisation algorithm.

So, in the next chapter we will present the most commonly used visualisation techniques and examine the type of input data each of these techniques operates on. Then, we will present our own research which is the construction of new modelling approach that clearly belongs to the constrained-based category of both deterministic and stochastic objects.

The approach we propose in this dissertation uses a point-membership classification test to define objects. This test, which we will also call the *defining test*, will be used to determine a surface by visualising all points that validate it. We concentrate our efforts to construct a general test that will enable the generation of 'families' of implicit surfaces. To achieve this generality we parameterise this test appropriately.

Moreover, in order to demonstrate the constrained-based nature of this modelling approach, we will describe surfaces that are too complex or impossible to describe analytically. The modelling approach that we propose is based on the Euclidean distance between points. We extend the definition of the Euclidean distance in order to assign a meaning in the distance between two geometrical objects including points, line segments, cylinders. In this way, the models we will generate (through parameterisation of the defining test) cannot be described analytically but, as the following chapters demonstrate, can be visualised. The visualisation algorithms also proposed here, do not use convex polygons or similar surface descriptions, but directly apply the defining test in order to detect and depict the necessary points.

This way of defining shapes is not a new one; actually it was one of the first methods ever used in mathematics. Let us take a simple two-dimensional object, say, a circle. Its definition, taken from a mechanical method of construction, is 'the locus of points that are equidistant from a given point'. Similarly, objects like ellipses, parabolas, and hyperbolas were also defined by this distance-based approach. As more complex objects emerged from the use of more complicated distance-based constraints, their conceptualisation became impossible. This led to the introduction of the analytical object descriptions which is today the main method for object definition and analysis.

Computer graphics, however, as we shall illustrate in this dissertation, offers a very powerful way for studying geometrical objects in this ancient way. This is achieved by visualising the object's projection onto the plane (or three–dimensional volume) of a viewport. The additional dimensions of the object, if any, can be conceptualised by the shades that are produced when the object is illuminated, and by the additional animation aids that enable the movement (rotation, translation etc.) of the object in any directions required.

Chapter 3 Current visualisation techniques

3.1 Introduction

In this chapter, we will present the tools that are commonly used for the transformation of models into images. Following the notation that we introduced in the first chapter, depicted in Figure 1.4, we assume that all the objects in a scene are described at their ACTUAL position using the ABSOLUTE Cartesian coordinate system. Since during modelling it may be proved more convenient to use the objects' SETUP positions, we assume that the appropriate transformation matrices for the SETUP to ACTUAL position of every object and the inverse matrices are also provided. This information, together with all the data necessary for the realisation of the chosen shading model, will become the input to the visualisation phase of a typical computer graphics application.

In a similar fashion to modelling, there are a number of different visualisation approaches currently in use, all of which have some advantages and disadvantages over the rest. Most of them have been designed to match a particular modelling approach but 'cross-combinations' between modelling and visualisation algorithms have also been tried.

Despite the diversity of the visualisation approaches, as the following sections will illustrate, their underlying functionality consists of the following five stages (Figure 3.1); *clipping* the scene inside the visible area, *projecting* it onto the two–dimensional WINDOW system, *removing the hidden lines and surfaces* to determine the surfaces that are visible by the current observer, *shading* the visible surfaces according to the chosen shading model and, finally, *mapping* the image from the WINDOW system to the VIEWPORT system (of a real or model viewport device). All five stages are not necessarily explicit in all the visualisation approaches we will present. Moreover the order in which they take place in a particular visualisation algorithm need not be strictly sequential but some stages may occur concurrently.

Figure 3.1 The five stages of visualisation

In this chapter we will present the three the most frequently used visualisation approaches; *polygonal mesh*, *octree* and *ray tracing*. The polygonal mesh, as its name reveals, is optimized to handle objects that have been modelled with the polygonal mesh modelling approach. The octree visualisation algorithm is mainly used in conjunction with the analytical functions, as well as the constructive solid geometry, other procedurally defined models, and the volumetric arrays modelling approach.

Ray tracing is mainly used with models using analytical functions but it can be adapted to use almost any other modelling approach including the polygonal mesh, the constructive solid geometry and the n-dimensional arrays. The distinguishing difference between all three visualisation approaches is the manner that the five stages of visualisation (i.e. clipping, projection, hidden surface removal, shading and mapping) are implemented. In the following sections we will present for every visualisation approach the particular methods used in implementing all the above five stages.

3.2 Polygonal mesh

In the polygonal mesh visualisation approach, all five stages can be isolated and studied separately. They happen sequentially, although depending on the type of projection used, clipping may occur before (in perspective projection) or after (in orthographic projection) the projection stage. Therefore, we will first present the stage of projection before that of clipping. Unless otherwise stated, we assume that the scene is modelled in the three-dimensional space, using the ABSOLUTE coordinate system. Furthermore we assume that an observer has been introduced in the scene, and the scene model has been eventually transformed into the OBSERVER coordinate system. The eye of the observer is at the origin of the OBSERVER system and the direction of view is assumed to be along the negative z- axis. Finally we also assume that a number (one or more) of light sources have been introduced in the scene and the transformations from their corresponding LIGHT systems to the ABSOLUTE have also been determined.

3.2.1 Projection

What the eye of the observer sees when looking at a three-dimensional scene is a *projection* of the vertices, lines and facets of the objects in the scene onto a *view plane*, which is assumed to be perpendicular to the line of sight. A projection is defined by a set of lines which we call *the lines of projection*. The projection of a vertex onto a plane is the point of intersection of the plane with the unique line of projection which passes through the vertex. The projection of a line segment onto a plane is the line segment in the plane which joins the projections of its two end-points. The projection of a facet onto a plane is the polygon formed by the projection of each of its corner vertices joined in the same order. It is important to note that the sequence in which vertices, lines and facets are drawn may be critical; on some raster viewport devices such as the monitor of a computer, earlier determined vertices, lines and facets can be obscured by later over-drawing.

In the OBSERVER system the view plane is usually defined to be of the form $z = -d$ (for some $d \geq 0$) – a plane parallel to the x/y plane and perpendicular to the z- axis. Vertices are projected onto this plane by a transformation matrix that produces projected points with coordinates of the form $(xp\ ,\ yp\ ,\ -d)$, where xp and yp depend upon the type of projection and on d, the fixed perpendicular distance of the view plane from the eye.

The obvious WINDOW system to choose has the x- and y- axes parallel to the x- and y- axes (respectively) of the OBSERVER system, with the origin on the OBSERVER z- axis at $z = -d$. Then any point vector on the view plane with the OBSERVED triplet of coordinates $(xp\ ,\ yp\ ,\ -d)$ has WINDOW coordinates $(xp\ ,\ yp)$. Of course we still have to calculate the values of xp and yp for every vertex in the model. As yet we have neither defined the position of the view plane (the value d), nor have we described the type of projection of three-dimensional space onto the plane. These requirements are closely related. In this chapter we will consider two possible projections, first the *orthographic*, which sometimes called the *axonometric* or *orthogonal* projection, and then the *perspective* projection.

The orthographic projection

A *parallel projection* is characterised by having parallel lines of projection, and is a projection under which points in three-dimensional space are projected along a fixed direction onto any plane not parallel to those lines. The orthographic projection is a special case whereby the lines of projection are perpendicular to the plane (it is sometimes referred to simply as *the parallel* projection). We can choose the view plane to be any plane with normal vector along the line of sight (the line of projection). This means that we can take any plane parallel to the *x*/*y* plane of the OBSERVER system, and for simplicity we choose the plane through the origin, given by the equation $z = 0$. An OBSERVED vertex is thus projected onto the view plane by the simple expedient of setting its z coordinate to zero, and thus any two different points with OBSERVED coordinates (x , y , z) and (x , y , z') say (where $z \neq z'$), are projected onto the same point $(x , y , 0)$ on the view plane, and hence onto the point (x , y) in the WINDOW system.

The perspective projection

The orthographic projection has the property that parallel lines in three-dimensional space are projected into parallel lines on the view plane. Although they have their uses in certain scientific and architectural applications, such views do look odd! Human comprehension of spatial position is based upon *perspective*. Hence our brains attempt to interpret orthographic figures as if they are perspective views. In order to achieve visual realism, it is essential to produce a projection which displays perspective phenomena – that is, parallel lines should meet on the horizon, and an object should appear smaller as it moves away from the observer. The drawing-board methods devised by artists over the centuries are of some value to us, but the three-dimensional coordinate geometry introduced in chapter one furnishes us with a relatively straightforward technique for achieving this.

What is perspective vision?

To produce a perspective view we introduce a very simple definition of what we mean by vision. We imagine every visible point in space sending out a ray which enters the eye. Naturally the eye cannot see all of space, it is limited to a cone of rays which fall on the retina, the so-called *cone of vision*, which is outlined by the dashed lines of Figure 3.2. These rays are the lines of projection. The axis of the cone is called the *direction of vision*

(or the *straight-ahead ray*). In what follows, we assume that all coordinates relate to the OBSERVER right-handed coordinate system, with the eye at the origin and the straight-ahead ray identified with the negative z- axis.

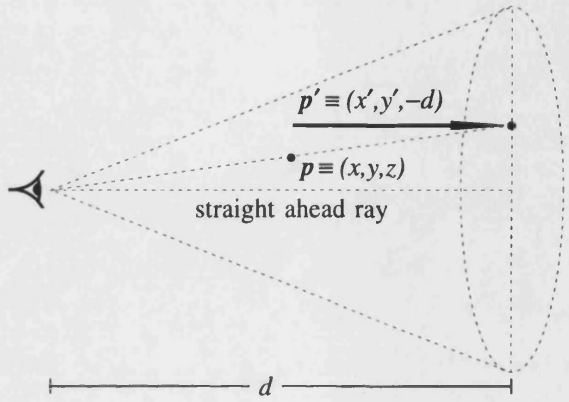


Figure 3.2 The cone of vision

We place the view plane (which we call the *perspective plane* in this special case) perpendicular to the axis of the cone of vision at a distance d from the eye (that is, the plane $z = -d$). In order to form the perspective projection we mark the points of intersection of each ray with this plane. Since there is an infinity of such rays, this appears to be an impossible task. Actually the problem is not that great because we need only consider the rays which emanate from the important points in the scene, in particular the corner vertices of polygonal facets. Once the projections of the vertices onto the perspective screen have been determined, the problem is reduced to that of representing the perspective plane (the view plane) on the graphics viewport. The solution to this problem is similar to that of the orthographic projection and will be discussed in the *mapping* stage of visualisation.

The calculation of the projected points using perspective projection is as follows. We let the perspective plane be a distance d from the eye. Consider a point $\boldsymbol{p} \equiv (x, y, z)$ (with respect to the OBSERVER system) which sends a ray into the eye. We need to calculate the point of intersection, $\boldsymbol{p}' \equiv (x', y', -d)$, where this ray cuts the view plane (the $z = -d$ plane), and thus we determine the corresponding WINDOW coordinates (x', y'). First consider the value

of y' by referring to Figure 3.2. By *similar triangles* we see that $y'/d = y/|z|$, that is $y' = -y \times d/z$ (remember that points in front of the eye in the OBSERVER system have negative z coordinates). Similarly $x' = -x \times d/z$ and hence $\boldsymbol{p}' \equiv (-x \times d/z\ ,\ -y \times d/z\ ,\ -d)$. Thus the WINDOW coordinates corresponding to $\boldsymbol{p}$ are $(-x \times d/z\ ,\ -y \times\ d/z)$. The projection makes sense only if the point has negative z coordinate (that is, it does not lie behind the eye). In the next section we will see how with three-dimensional clipping we can ensure that we will be using only the vertices that conform to this condition.

3.2.2 Clipping

Theoretically, objects may be positioned throughout space, even behind the eye. The formulae derived to represent the perspective projection deal successfully only with points that lie within the so-called *pyramid of vision.* Any attempt to apply the formulae to points outside this volume, especially those lying behind the eye, gives nonsensical results. The scene must, therefore, be *clipped* so that all vertices of the new, clipped model lie within this pyramid before the projection may be applied. The process of clipping will intersect the scene with the pyramid of vision. In this way, the clipped model of the scene may be different from the original non-clipped model. In this sub-section we will briefly explain clipping in spaces of two and three dimensions.

The clipping of vertices, lines and facets in two–dimensional space is simply the task of determining which parts lie within a rectangular window with dimensions *horiz* × *vert* (i.e. the window). This task is interpreted mathematically by calculating the intersection of the rectangle defined by the window, with all the objects of the two–dimensional scene. This mechanism is also sufficient for dealing with orthographic projections of three–dimensional scenes since the whole space can be projected onto the view plane hence, bringing the problem in two dimensions — thus projection may occur before two–dimensional clipping.

Dealing with perspective projections is rather more complex. Once again we assume that we have a view plane some distance from the eye along the negative z-axis of the right-handed OBSERVER system. A rectangular (*horiz* × *vert*) window on this plane will be identified with the graphics viewport. We may also assume that the eye is positioned in such a way

that each vertex has a strictly negative OBSERVED z coordinate. This ensures that every vertex can be projected onto the view plane by our chosen perspective projection, whence two-dimensional clipping ascertains which parts of the image lie totally within the window.

Suppose, however, that we wish to depict a scene as viewed from a position within the model, such as a point lying in a landscape with a large ground plane. Clearly, parts of the model will lie behind the eye and consequently cannot be projected on the viewplane. Such problems cannot be resolved by two-dimensional clipping and so extended methods must be developed. *Three-dimensional clipping* must determine which parts of a line or, facet, can be projected, and subsequent hidden surface removal must be executed on the clipped scene.

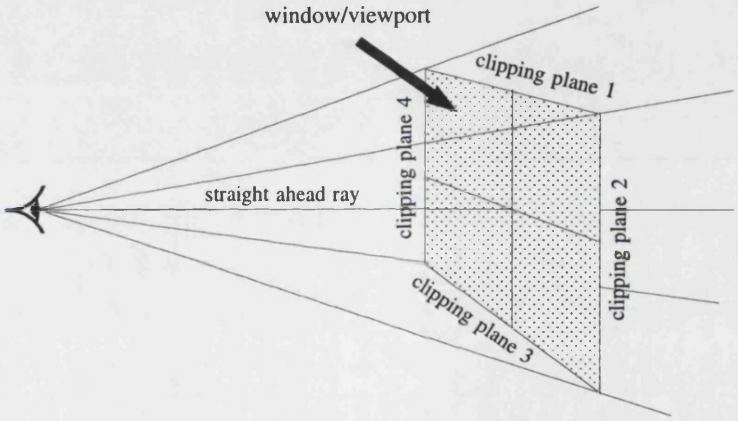


Figure 3.3 The pyramid of vision

There are consequently two problems that need to be solved. Firstly, we must determine which part, if any, of a facet lies in the volume of space projected onto the window, and secondly we must incorporate this information into the data structures representing the scene.

The volume of three-dimensional space which is projected onto the window is a rectangular pyramid with axis of infinite length. This pyramid which we call the *pyramid of vision* (a subset of the cone of vision), has its apex at the eye position (the origin of the OBSERVER coordinate system) and four infinite edges, each passing through one corner of the window on the view plane (Figure 3.3). It is thus bounded by four planes (the *clipping planes*), each of which contains the OBSERVER origin and one edge of the rectangular window.

A point vector, (x, y, z), lying within the pyramid of vision is projected, by perspective projection, onto the point $(-d \times x/z, -d \times y/z)$ in the window (d is the perpendicular distance from the eye to the view plane). Each clipping plane divides space into two halves. The half-space containing the pyramid of vision is said to be the *visible side* of the plane. The four clipping planes must be represented in such a way that we may easily determine whether a point lies on their visible side or not.

3.2.3 Hidden surface/line removal

Although a high degree of realism is achieved with regard to the geometry of the image produced, the perspective projection is not sufficient to produce images that could be directly mapped onto the viewport. The problem that arises becomes obvious with the following example: consider that a cube has to be displayed. It is modelled as a set of six square polygonal facets. When projected, all facets of the cube are visible on the screen. But this is not true in reality, unless the cube is perfectly transparent. In reality, one can see at most three facets of a cube simultaneously. Its other facets are hidden by the bulk of the cube itself. To simulate this situation, a *hidden surface removal* algorithm has to be applied to the projected polygons before they are mapped onto the viewport.

There is a variety of hidden surface removal algorithms in literature [Sutherland *et al.* 1974; Foley *et al.* 1990]. Some have enormous storage overheads and need powerful computers. In this section we will briefly present two of the most frequently used, namely the *painter's algorithm* and the *Z-buffer*.

The painter's algorithm is based on the assumption that all objects are closed (i.e. solid) and there is a way for identifying whether a facet is viewed from inside (*invisible*), or outside (*visible*) the object. After all the facets are projected, they are examined to discover whether they overlap or not. This test is restricted to the clipped and visible facets only. If they do overlap, there is a second test that determines which facet lies in front and which is behind.

After all the visible facets have been checked, they are displayed on the viewport by drawing the furthest from the observer first, and progressively drawing the nearest in front of the observer last. In complex scenes the topological ordering of the visible facets will require the use of special data structures such as the *directed graph*, that will pose great memory demands on the computer used.

This algorithm is based on the assumption that the type of viewport we use supports 'over-drawing'. Therefore, when facets are displayed, those that are drawn first will be 'over-drawn' by the facets that will be drawn later. This approach resembles that of a painter, hence its name.

Another, probably conceptually the simplest, approach, but one which is rather expensive on processing power and memory, is the so-called *Z-buffer* algorithm. This involves a rectangular array representing the totality of pixels on the screen. We imagine rays of light entering the eye through each of the pixels on the screen.[1] We consider these rays as axes of a rectangular (orthographic) or pyramidal (perspective) prism leading from the eye to the pixel, and off to minus infinity ($-\infty$). These rays naturally pass through objects in our scene and we can note the coordinates of these points of intersection. The z value of the intersection of the axis of this prism with each object is calculated in turn and compared with the buffer value. The array, the Z-buffer, will hold the 'z- coordinate' (initially minus infinity) of the nearest point of intersection. So we build up a picture by adding new objects, finding where the rays cut the object, and changing the array values (and the pixel colour on the screen) whenever the latest point of intersection is nearer to the eye than the corresponding value stored in the array, giving a simple hidden surface algorithm for each pixel.

Another approach, the *scan line algorithms*, considers one scan line of a raster screen at a time and uses information about polygonal facets in the scene, much in the same way as the Z-buffer, to colour these scan lines correctly, giving a correct hidden surface picture. Yet another way is to *seed* each facet with a single representative point. When the scene is

[1] Actually, the rays are assumed to pass through the cross-points of a rectangular grid that is formed on the window, and during the stage of mapping this lattice will correspond to the pixel organisation of the viewport.

transformed into OBSERVED position, the seed points are ordered in increasing distance from the observer's eye (i.e. the origin of the OBSERVED coordinate system). This order is then used to ascertain which facet lies in front of which: the so-called *depth-sort algorithm*. This is not a very satisfactory method because it will often give incorrect displays of scenes which contain a wide variety of facet sizes and topologies.

3.2.4 Shading

In the combination of clipping - projecting - hidden surface removal, a shading model has to be added in order to incorporate colour shades. Its purpose will be to determine the colour of every pixel of the screen, by calculating the amount of light that can be seen on any point in the scene which is visible to the observer and so would eventually be mapped onto the viewport. For pixels that do not represent any points on any facet (i.e. the sampled points do not belong to any facet), the *background* colour should be used. For the rest of the pixels, the shading model is applied — to the points represented by pixels — to calculate the amount of light that is reflected to the observer from each visible point in the scene.

There exist a number of different shading models that can be incorporated in the polygonal mesh visualisation approach. As we will also see in section 3.4.8, depending on the number of optical phenomena (i.e. specular reflection, shadows, transparency) that need to be simulated, the complexity of the mathematical models that describe them vary. Apart from a few very rudimentary shading models, the necessary data for their implementation are:

- the location of the observer (a vector)
- the location of all the light sources (an array of vectors)
- the intensity and colour of the light sources (depends on the colour model used)
- the *material properties* of all the surfaces of all the objects in the scene (absorption, reflection, transmission coefficients depending on the colour model and the shading model)
- a mechanism (e.g. function) to calculate the normal vector to any point on the surfaces of all objects in the scene.

In this section we will briefly present some of the most frequently used shading models and explain how reflections, transparency and shadows can be simulated. More complex shading models are discussed in section 3.4.8. For the rest of this dissertation, we will make the assumption that the RGB colour model, introduced in the first chapter, is being used.

Constant colour shading

The simplest and quickest way of displaying a facet using the RGB model is by *constant colour shading*. We assume that the shade is constant across any facet and once the colour of light reflected from a point on the facet is found, the facet is displayed on the window using a simple area-fill. Although constant colour shading is reasonably sufficient for scenes made up entirely of matt, planar surfaces, this method has a number of disadvantages. The results obtained on models representing curved or glossy surfaces are unsatisfactory – the polygonal facets that make up the approximation to a curved surface are clearly distinguishable, and also the specular highlights are unconvincing since they are constrained to be made up only of entire facets. Increasing the density of facets in the mesh helps to some extent, but we are able to produce far more convincing images of approximated curved surfaces by what is generally referred to as *smooth shading*. Here a surface is displayed by individually shading every pixel on each projected facet of the polygonal mesh in a way that smooths out any intensity discontinuities. We give two different interpolation methods for smooth shading: Gouraud shading and Phong shading.

Gouraud shading

Gouraud's method of *intensity interpolation shading* [Gouraud 1971] goes a long way towards solving the problems of constant shading mentioned above. The intensity of light reflected at each vertex of a facet is determined, taking into account ambient light, and diffuse and specular reflection. The intensity at each internal point of a projected facet may then be calculated by interpolation between these intensity values. The trick is in calculating the intensity at the vertices. Suppose we have a number of facets approximating to a curved surface. Each vertex lies in the real curved surface and is contained in the boundaries of a number of the approximating facets. The vertex normal may be found by averaging[2] of the

[2] Usually we assume that all facets have approximately the same area, but a weighted average according to the actual area of every facet may produce more convincing results.

surface normals of the facets containing the vertex in their boundaries. The *apparent colour* (or vertex intensity) at the vertex may then be calculated. The intensity or colour at each point within the facet is then determined, using a *scan line* approach, by interpolating between the vertex intensities as shown in Figure 3.4. The intensity at point ***A*** is found by interpolating between those at points ***1*** and ***2***, the intensity at ***B*** is found by interpolating between ***3*** and ***4***, and finally the intensity at ***C*** is found by interpolating between ***A*** and ***B***.

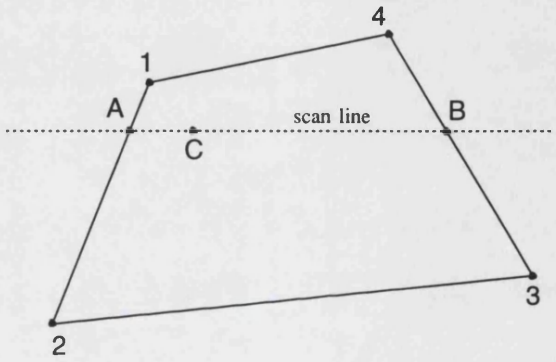


Figure 3.4 Interpolation of intensities within a facet

Phong interpolation

Some problems still remain with Gouraud shading, mainly involving facets which face almost directly towards the light source. In Figure 3.4, for example, points ***A*** and ***B*** may both have the same intensity and so interpolating between them results in a constant intensity across the surface, making it appear flat. Problems also occur with the depiction of highlights produced by specular reflection.

These problems are partially eliminated by using Phong's *normal vector interpolation* shading [Phong 1975]. This method involves the calculation of the normal vector at each point on a surface by interpolating between the normals at the vertices, and thence calculating the shade by applying a shading model at that point. This method produces considerably more accurate and pleasing results, however, it is accordingly more time-consuming to implement.

Plates 3, 4 and 5 illustrate the differences of the three shading models. These plates show the same model of a teapot as it is visualised by all three shading models; constant (plate 3), Gouraud (plate 4) and Phong (plate 5). A more detailed analysis of these shading models is found in Angell and Tsoubelis [1992].

Shadows

A facet which obscures all or part of another facet from exposure to a light source is said to *cast a shadow* onto this other facet. A shadow cast by a convex polygonal facet, say ***j***, onto another convex polygonal facet, say ***i***, is also a convex polygon which may be considered to lie on the surface of facet ***i***. This polygon is usually called a *shadow polygon.* The criterion for finding shadows is very similar to that for finding hidden surfaces and most hidden surface algorithms can be adapted accordingly. Usually the model is calculated as if it was observed by an imaginary observer located at each light source. For a variety of alternative solutions see [Crow 1977; Angell & Tsoubelis 1992].

Transparent surfaces

Many hidden surface algorithms can be adapted to allow for the inclusion in the scene of facets made of transparent materials. This is by no means a trivial exercise and a full simulation, taking into account specular reflection, refraction etc., is too complex to be implemented with the polygonal mesh visualisation approach. Nevertheless, if we accept certain limitations, then we can deal with transparent surfaces in the polygonal mesh model using a topological depth-ordering algorithm similar to that of the hidden surface removal algorithm. A simplified version of such an algorithm assumes that the index of refraction of all the transparent surfaces is always unity, and no more than one transparent surfaces may overlap if seen by the observer's viewpoint.

Reflections

Suppose one facet in a scene is a *mirror.* We should be able to see the reflection of the scene in this mirror. If we calculate the reflection of each vertex of the scene in the mirror facet, we have the *physical reflection* of the scene. Note that here we are creating a new set of points with coordinates specified in relation to the same coordinate system – the OBSERVER system.

How can we relate to this physical reflection with the reflection observed in the mirror? Imagine that the mirror facet is a window surrounded by an infinite plane. The reflection in the mirror is precisely the part of the physical reflection which can be seen through (and beyond) this window. Those parts of the physical reflection which lie in front of the mirror cannot be seen in the reflection since in the real scene they lie behind the mirror. The problem of reflection thus reduces to projecting the reflected scene onto the view plane, and then drawing only those parts which both intersect with the projection of the mirror and lie behind the mirror in reflected space.

There is a major drawback to any algorithm for finding reflections of scenes. If you sit in front of a mirror with another mirror behind you, what do you see? You see a reflection of yourself in the mirror in front of you, but you also see a reflection of the mirror behind you, in which you see a reflection of your back and of the mirror in front of you, in which you see a reflection of the mirror behind you, and so on! The process is infinite and there is no way round this. We must either insist that a scene contains no mirrors that reflect each other, or else we simply ignore infinite reflections of mirrors, allowing for only a limited number of *levels of reflection*. Usually, a limit of one level of reflection is imposed, so when reflected in another mirror, a mirror facet is considered as an ordinary, non-reflecting facet.

3.2.5 Mapping

Once we know the coordinates and the colour of all the points on the window, we are ready to map them onto the viewport. As Figure 1.5 of chapter one shows, for a window of ***horiz*** × ***vert*** size and a viewport of ***nxpix*** × ***nypix*** pixels, the coordinates of point $p \equiv (x, y)$ will be mapped onto the pixel with coordinates $(fx(x), fy(y)) \equiv (pixel.x, pixel.y)$ via the following mapping functions:

$$fx(x) = \left[\frac{nxpix\,(2x + horiz)}{2\,horiz}\right], \qquad fy(y) = \left[\frac{nypix\,(2y + vert)}{2\,vert}\right]$$

where $[r]$ denotes the integer part of the expression r.

From the above formulae we can observe that if two points on the window have a horizontal distance less than ***horiz / nxpix***, and a vertical distance less than ***vert / nypix***, it is quite possible[3] that they will be mapped onto the same pixel, hence one will 'over-draw' the other. This means that we use computer resources to calculate the colour of points that we will eventually not use. To avoid this waste of resources, before attempting any calculations regarding points belonging to the same facet, we first determine whether they will be mapped onto different pixels on the viewport. This observation optimizes the visualisation algorithm to match the resolution of the screen, thus improving its efficiency.

3.3 Octree

In the octree algorithm, the four out of the five stages of visualisation are implicit. Specifically, projection, clipping, hidden surface removal and mapping are embedded in the process or in the form of initial conditions in the octree algorithm. Therefore, before presenting how each stage is affected by the algorithm, it is vital to present firstly the algorithm itself.

The octree visualisation algorithm was originally used to match the volumetric arrays modelling approach and the octree data compression technique as we presented them in chapter two. It assumes that the scene is enclosed by an appropriately positioned *supercube* that is properly aligned with the viewplane (i.e. the plane of the window). The shape of this supercube is frequently assumed to be a geometric cube. However, as we will see in the next section, there are cases where the shape of the supercube is a trapezoid (truncated pyramid).

Given a list of objects in the scene, this supercube is called *homogenous* if it does not intersect with any of the objects, and *heterogenous* if it does. When calculating intersections, we use the solid supercube (i.e. the volume of space enclosed by the supercube) and not its facets. Paradoxically, however, we intersect the solid supercube with the surfaces of the

[3] Depending on the absolute values of their co-ordinates and the exact mapping functions (i.e. truncation, rounding, etc...) these points will be mapped onto the same or adjacent pixels.

objects in the scene. As a result, the supercube that is totally contained inside an object is assumed homogenous. The reason is that only the surfaces of the objects can contribute to the colour of the pixels on the viewport.

If the initial supercube is characterized heterogenous, it is subdivided by eight equal-sized *subcubes*. We then recursively try to characterize each of the subcubes as homogenous or heterogenous. Whenever a heterogenous subcube is found, we further subdivide it into eight equal-sized subcubes. This process of recursive subdivision is terminated when the size of the subcube becomes such that it can be mapped onto exactly one pixel on the viewport. We name this minimal-sized subcube as *cubelet*.

3.3.1 Projection

The type of projection used is dictated by the shape of the supercube and subsequently, the subdivided subcubes. Specifically, if the supercube and the subcubes have a cubic shape, then the orthographic projection is implied. However, if the initial supercube has the shape of a pyramid, similar to the pyramid of vision (i.e. the same apex and parallel facets) then the perspective projection is implied.

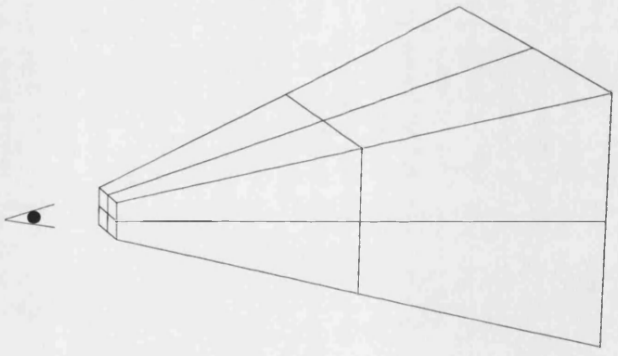


Figure 3.5 Perspective projection with the octree approach

To ease the process of subdivision, this pyramid-shaped supercube is truncated by the viewplane, resulting in a supercube with the shape of a trapezoid (a truncated pyramid). Subdivision would therefore produce trapezoid-shaped subcubes that are not similar to each other, as Figure 3.5 shows. This figure shows the first level of subdivision of a supercube

into trapezoid-shaped subcubes, as it is determined by using the midpoints of the supercube as the vertices of the newly generated subcubes.

Clearly, the perspective projection adds a considerable overhead of calculations, during subcube subdivision and tests of subcube - object(s) intersection, thence for the rest of this chapter we will assume the use of the orthographic projection and cubic-shaped supercubes for the octree visualisation approach.

Even with the orthographic projection and the use of cubical subcubes, the characterisation of a subcube as heterogenous or homogenous is a major overhead. Consider for example the intersection test of a subcube with an infinitely long helix. Usually, in such cases, a more relaxed test is implemented that replaces the subcube with a sphere that circumscribes it. Therefore, the orientation of the subcube with regard to each surface in the model is no longer critical. But the volume that the sphere covers is larger than that of the subcube it replaces, and the intersection test may provide incorrect results thus falsely subdivide the subcube and waste computer resources. However, the speed gains achieved by the use of spheres in the intersection tests more than compensates for the misleading results. Furthermore, errors incurred at one level of subdivision will be amended at the next level down. Additionally, errors at the cubelet level are insignificant since they will affect at most one pixel. At the section of shading, we will see how we can further subdivide the cubelet, thus achieving a more accurate implementation of the octree approach.

3.3.2 Clipping

With the octree visualisation algorithm, clipping is not an issue since whatever lies outside the initial supercube will never be processed. Therefore, the appropriate position of the supercube will ensure clipping. Usually, the initial supercube is located in such a way that the direction of view passes through it centre and is perpendicular to one of its sides.

Another important factor that affects clipping is the size of the supercube. If it is too small, many pixels on the viewport will not be painted at all. On the other hand, if the size of the supercube is too large, then we will be calculating the colour of pixels that will eventually

fall beyond the boundaries of the viewport. Moreover, the supercube needs to be equally subdivided in all three dimensions thus producing a lattice of $2^N \times 2^N \times 2^N$ cubelets. Such an organisation maps directly onto a square viewport, but will not necessarily match the pixel arrangement of the viewports we use.

Usually, the number of subdivision levels N is an integer between $log_2(nxpix)$ and $log_2(nypix)$. These limits on the number of subdivision levels constrain the size of the initial supercube to be between $nxpix^3$ and $nypix^3$ cubelets. Smaller values for N will generate cubelets that correspond to more than one pixel, and larger values will produce cubelets that may map to the same pixel.

To add to the complexity of this problem, we need to cater for the aspect ratio of the viewport that will result into rectangular-shaped pixels, thence needing rectangular shaped subcubes. The non unity aspect ratio can be adjusted by a transformation matrix that appropriately transforms the axes of the ABSOLUTE coordinate system according to the scaling constraints of the pixels' shape.

3.3.3 Hidden surface removal

This stage of visualisation is also implicit in the octree algorithm. Consider a supercube of size $N = 10$. This will result in a viewport of $2^{10} \times 2^{10} = 1024 \times 1024$ pixels and a lattice of $2^{10} \times 2^{10} \times 2^{10} = 2^{30}$ cubelets. It would be too time consuming, even for a powerful computer, to characterise all these cubelets as homogenous or heterogenous for a given scene. Besides, only a very small proportion of them will be eventually found as heterogenous, and out of these very much fewer will be visible by the observer and hence will have to be painted on the viewport. Therefore, the process of subdivision becomes critical to the effectiveness and efficiency of the algorithm.

With regard to hidden surface elimination, the octree algorithm is adjusted so that the subcubes nearest to the observer, along the line of projection, are processed before those furthest away. As a result, a heterogenous cubelet is detected, (i.e. a surface is found), the corresponding pixel on the viewport can be painted. Therefore there is no need — unless

transparency is to be simulated — to process any cubelets that lie behind the heterogenous ones that we have already processed because they would eventually correspond to previously painted pixels. This observation is also true for a subcube of any size provided that the corresponding area onto the viewport is totally painted.

Therefore the octree algorithm approximates the surfaces of objects within the supercube by identifying and painting those cubelets that intersect with the surfaces: note that this algorithm does not actually consider solid objects, only surfaces. Nevertheless, in order to use the functionality of Constructive Solid Geometry (chapter two), we have developed techniques that identify whether a subcube lies totally outside, totally inside, or intersects with an object. In cases where the classification of inside / outside is nonsensical (i.e. open surfaces) we only distinguish wether a subcube intersects with a surface, or not.

Determining whether a pixel has already been painted implies that there is a bi-directional communications link between the computer and the viewport. But such a link may not always be efficient. For example, there are installations where the viewport is connected via uni-directional parallel links, or other proprietary setups. Moreover, when using a model viewport device, this link may be impossible. In such a case, a copy of the viewport's image, that we will call *memory screendump*, must be stored at the immediate memory (RAM) of the computer. For a 24 bit colour, 4096 x 3072 pixels viewport the screendump is approximately 36 Mbytes; a major constraint for most computer installations. Fortunately, the size of the screendump may be considerably reduced since we do not need information about the colour but only a flag whether a pixel is painted or not. Consequently, the memory demands for the above example will be 24 times less since we only need one bit instead of 24 bits per pixel.

3.3.4 Shading

With regard to shading, any shading model that calculates ambient light, as well as diffuse and specular reflection may be used, provided the appropriate information is available. One difficulty we may encounter at this stage is the choice of the point on which the shading model will be applied. This difficulty relates to the fact that we have to sample the cubelet's

continuous space and choose a representative point for the discretised viewport. But the size of the cubelet, however small it may be, is sufficient to enclose an infinity of such candidate points, all belonging to the modelled surface. Moreover, in some cases, points from more than one surface may intersect with the same (heterogenous) cubelet. Therefore, it is essential that our sampling method, as to which point is used for the application of the shading model, must be accurate. It is very difficult to provide a universal solution, but the following rules of thumb provide the general strategy that can be followed in order to select such a candidate point.

If the cubelet contains points from one surface only, then the centre of the cubelet may be used. However, that point may not belong to the surface at all, hence making the determination of the normal to that surface problematic. In such a case, approximation techniques may be used either to estimate the normal, or determine the point on the surface nearest to the centre. In both cases, the error introduced depends highly on the curvature of the surface at that location.

If the cubelet intersects with more than one surface, a choice of which is the visible surface (i.e. the nearest to the observer) has also to be made. One method is to draw the projection line that passes through the centre of the cubelet and then determine the nearest — to the observer — intersection with all the surfaces that pass through that cubelet. Such a technique will demand extra coding, but a number of alternatives that simply re-use code already developed for the octree algorithm also exist. These alternatives use *subcubelets*, that are subcubes of a size smaller than the cubelet. One such alternative extends the subdivision process a certain number of levels (usually one or two). Then the shading model is applied for all the visible points found, and the colour of the corresponding pixel is determined by averaging the colours of these points. This method will not guarantee a unique intersection of subcubelet — surface, but the process of averaging for the final colour will ensure that the errors have been 'smoothed' (*anti-aliased*).

Another method uses subcubelets centred around the projection line, which in turn passes through the centre of the initial cubelet. If the nearest heterogenous subcubelet intersects with more than one surface, the process is repeated with subcubelets of a smaller size up

to a certain level. This method ensures that the correct visible surface will be determined eventually. Moreover, it is based on intersection algorithms between subcube and surfaces, that should have already been coded in the octree algorithm.

Once the appropriate point coordinates for a heterogenous cubelet have been determined, the shading algorithm has to be invoked. A straightforward shading model would assume that the point is visible by the light sources and would calculate the ambient light, diffuse and specular reflection components that this point would reflect back to the observer. However, such a model would fail to simulate optical phenomena like transparency, shadows and reflections.

Shadows may be simulated by the addition of a *visibility test* that establishes whether the point is visible by a light source, and adjusting the diffuse and specular reflection components accordingly. However, such a visibility test would imply the implementation of code to achieve a *line to surface* intersection.

Transparency is another phenomenon that with some extra coding can be simulated. With the restricting assumption that all the materials have a unit index of refraction, pseudo-transparency is embedded in the hidden surface removal stage. Specifically, instead of terminating the subcube subdivision as soon as the corresponding area on the viewport is found painted, we now continue the subdivision until we ensure that the subcube does not intersect with any semi-transparent surface [Doctor *et al.* 1981]. This extension to the basic octree algorithm would imply that the memory screendump should be large enough to keep the full colour of every pixel, thence transparency is simulated by the proper (proportional to the degree of transparency) averaging of colours.

3.3.5 Mapping

The octree approach to visualisation of a three-dimensional scene assumes that drawing will take place on a *superscreen* which is a $2^N \times 2^N$ discrete grid of pixels that for the purposes of our presentation we may assume $N = 10$. For the time being we will ignore any aspect ratio problems and assume that each pixel is square — therefore the superscreen is defined

as 1024 × 1024 pixels square; our original viewport of *nxpix* × *nypix* pixels is centred in this superscreen. Usually, we ensure that the size of the superscreen, thence the choice of N, is such that it will fully contain all the pixels of the viewport or, we approximate N with the formula given in the previous subsection on clipping.

We now consider each square pixel to be the front face of a small cube, or *voxel* (volume pixel). By extending the superscreen back into the third dimension, denoted by the z-axis, we can thereby create a *superblock* composed of $2^N \times 2^N \times 2^N$ voxels. An individual voxel is located within the superblock by its voxel coordinates, counting the number of voxels say, to the right (x), above (y) and into the screen (z) starting from the left, bottom, front voxel of the superblock. This apparently peculiar choice, is made so that each voxel that maps to a particular pixel in the viewport has exactly the same x- and y- pixel coordinates in the VIEWPORT system. We now re-consider each pixel on the superscreen to be the front face of just one of the 2^N voxels from the column that stretches out perpendicularly behind the superscreen. Actually, these columns are aligned with the lines of projection and the superscreen is assumed to correspond to the pixel organisation of the viewport.

We map the superblock onto a cube in three–dimensional space, that we have already called the supercube. The front face of the supercube is centred at the ACTUAL origin, and scaled appropriately. The same scaling factor will map individual voxels into the small cubelets in space, thereby mapping the superblock onto a total of $2^N \times 2^N \times 2^N$ cubelets. By this mapping we can now consider the front face of the supercube as the WINDOW onto three-dimensional space. Each column of 2^N cubelets that is perpendicular to, and behind, this WINDOW, corresponds to a column of voxels behind the superscreen.

With regard to the aspect ratio, the rectangular shaped pixels will predicate non-cubical voxels (i.e. parallepipeds). Such difficulties as we already discussed in the subsection on clipping, are overridden by the use of transformation matrices which alter the scaling of the coordinate systems used.

3.4 Ray tracing

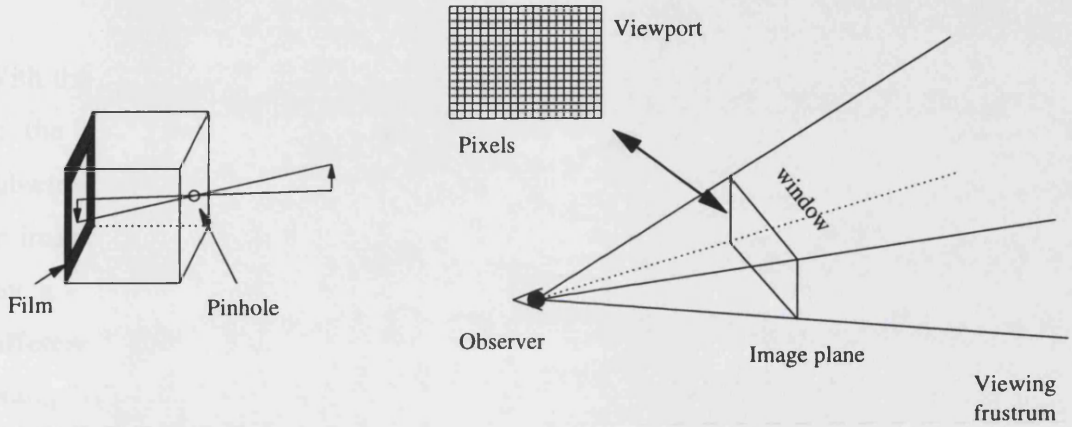
The last approach to visualisation that we will present in this chapter is ray tracing. It is based on the principles of the *pinhole camera model* and it was primarily developed as a *global illumination model* [Whitted 1980]. Compared to the rest of the visualisation approaches, it is the most capable of simulating optical phenomena. The simplicity of the ray tracing principles, its potential for achieving 'realistic images', and its adaptability to virtually all modelling approaches, make it the most promising computer graphics visualisation approach. However, ray tracing has not been adopted by many computer graphics users because its implementation demands computers with enormous power and it is best suited on parallel architectures.

In this section we will first present the pinhole camera model, followed by the two distinctly different ray tracing approaches, the *forward* and the *backward* ray tracing. Then we will continue to present the five stages of ray tracing that, in a manner similar to octree, are embedded in the algorithm.

3.4.1 The pinhole camera model

To conceptualise the pinhole camera model, we imagine a box, like the one in Figure 3.6, where in the centre of one of its facets there is a small hole (pinhole) and at the inner side of the opposite facet there is a light sensitive surface (e.g. photographic film). This system, in the history of photography is one of the oldest camera models, however, painters have used the underlying principles since at least the time of Canaletto. In the pinhole camera, light coming from outside the box (the environment) passes through the pinhole and hits the film. If the size of the pinhole is very small, then any small region on the film can only be affected by light coming along the direction that connects that area with the pinhole (Figure 3.6a).

In computer graphics,[4] the pinhole camera model is partially altered. The pinhole, is replaced by the *observer's eye* , and therefore the *image plane*, which plays the role of the film, has to be placed in front of the pinhole, as Figure 3.6b shows. In this way, the image that will be recorded on the image plane, which is the viewport, will not be inverted as is the case on the pinhole camera.



(a) (b)

Figure 3.6 The pinhole camera model and the ray tracing equivalent

In this model, the observer is restricted to 'see' only through the image plane. Therefore, the visible space is defined by the infinite pyramid which has its apex at the observer's eye and each infinite edge passes through one vertex of the image plane. This volume of visible space (i.e. infinite pyramid) is called the *pyramid of vision*. The above restriction is arbitrary and in some computer graphics applications the visible volume is further reduced to a *frustum* by excluding from the pyramid of vision the space between the observer's eye and the image plane. Therefore the pyramid of vision will be confined only in the volume which is in front of the image plane along the direction of view. This three-dimensional volume that can be projected on the image plane is also called the *viewing frustum* [Glassner 1989].

[4] With ray tracing other more complex camera models have been simulated but references to them will be given at later sections.

3.4.2 Forward ray tracing

In computer graphics, an image is determined by the colour of every pixel on the viewport. Consider such a pixel on the viewport; this will correspond to a small rectangular area on the image plane which we will call *subwindow*. What is visible through that subwindow has to be represented by a single shade (the colour of the corresponding pixel) and is a problem that "much of the work of 3D computer graphics is devoted to ..." [Glassner 1989 page 4].

With the ray tracing approach, the colour of each pixel will be determined by the averaging of the colours of all the light rays (their photons) that hit inside the corresponding subwindow on the image plane. Consider a computer graphics scene involving an observer, an image plane and inside the corresponding viewing frustum, some objects illuminated by say, a single light source. The light source will generate an infinite number of photons with different (depending on the source) colours, travelling to all possible directions. Take, for example, a photon coming out of the source and heading directly towards the observer's eye through the image plane. The observer will see light coming out of the source.

Consider now another photon coming out of the same source but heading towards an object in the scene. This photon hits the surface of that object and after interacting (exchanging energy) with the matter of that object's surface is reflected back towards, say, the image plane and the observer's eye. This is actually the reason that the observer sees the object: light (from a source) hits the surface of an object and is reflected towards the observer (passing through the image plane). Another photon may follow a totally different route, starting off from the same source, reflecting on several surfaces in the scene and then is either too weak to be noticeable, or never meets the image plane and the observer.

The mathematics describing the interaction of a light ray (stream of photons) with a surface may become very complex. A model simulating that interaction should be capable of describing the direction of a reflected and possibly a refracted ray (if the surface is transparent) and the colour (and intensity) of the light they carry. In simulating optical phenomena such as diffuse reflection, however, we need to know the complete distribution of reflected rays, hence considerably complicating the shading model algorithms.

The problem of determining the correct colour of a pixel on the viewport is transformed into averaging the colours of the light rays that hit the corresponding window on the image plane. One way of calculating these rays would be to start from a light source (since it is the only place that photons are generated) and follow the route of every ray that would eventually (after possible interactions with surfaces of objects in the scene) reach the observer. This process of following (tracing) light rays from the point they are generated until they hit the observer is called *forward ray tracing*.

Although in theory forward ray tracing produces the anticipated results, in practice it is too inefficient to use. Consider a light source; light rays emanating from that source will go to all possible directions. From all these rays, some will go off the scene directly, others will miss the observer after striking onto one or more surfaces, and only a very small percentage of initial rays will eventually reach (directly or not) the observer through the image plane. The identity (in terms of point of origin and direction) of these latter rays is known only after their complete route (through reflections and refractions) has been calculated. Therefore, simulating this approach in a computer proves inefficient since most of the CPU time will be spent in calculating the route of light rays that do not contribute anything to the final image on the viewport since they will never reach the observer. As a result, another approach (which resembles almost the inverse one) has been widely adopted as the correct implementation of ray tracing.

3.4.3 Backward ray tracing

This approach to ray tracing considers only the rays that would eventually contribute to the colouring of the pixels on the viewport. Given a pixel (on the viewport), any ray that comes from the scene and passes through the image plane and hits the observer's eye can be characterised as a vector that passes through the observer's eye and through the centre of a subwindow that would be mapped exactly to that pixel. In fact, any point inside the subwindow could be used, but for simplicity the subwindow's centre is chosen. In some sophisticated applications, where more than one point inside the subwindow are required, a deterministic and sometimes a stochastic method is employed to select them.

Therefore the only ray that determines the colour of that pixel, and which from now on will be called the *eye ray*, is assumed to pass through two known points: the subwindow's centre and the observer's eye. What remains to be determined is the light source from which this ray conveys light. If that ray comes directly from a light source in the scene, then we proceed to calculate the colour and intensity of this light contribution. If, instead, that eye ray comes from the surface of an object (i.e. intersects with that surface) then it could be the reflection or refraction (or a combination of both) of some other ray(s) that convey light. Consequently, these new *child rays* must be traced.[5]

For each of the child rays, therefore, the sources of the light they convey must also be determined. This is achieved by following every child ray along its path until we identify its origin, in a fashion similar to that of the original eye ray. In this way we construct a recursive path-finding process.

This recursive process usually terminates when a child ray does not intersect either with a light source or with any object in the scene and therefore is said to 'miss' the scene. In this case the source of that ray is assumed to be the background of the scene where we assume that uniformly scattered light (emanating from the background) illuminates all the surfaces of all objects equally (*ambient light*).

However, this recursive path-finding process is not guaranteed to terminate under all circumstances. For example, a ray that reflects between two appropriately positioned mirrors will spawn infinite generations of child rays. In such a case therefore, we have to apply some other termination criteria. The various criteria used and their advantages will be discussed with greater detail in the next subsection.

The process of backward tracing light rays from the observer, through the image plane, until they reach a light source or disappear in the background, is called *backward ray tracing*, and because of its improved efficiency, over the forward ray tracing, it is the one adopted by the overwhelming majority of computer graphics users. We have also adopted this approach, and for the rest of this dissertation ray tracing will be synonymous to backward ray tracing.

[5] Optics have modelled reflection and refraction and therefore these rays can be calculated.

3.4.4 Definitions

From the recursive process of tracing rays, a tree data structure that depicts the complete route of the initial eye ray may be defined. This tree is called the *ray tree*. The root of the ray tree describes the vector of the eye ray and its nodes hold information about the child rays. Moreover, the child rays may be classified into three different categories according to the way they were generated [Glassner 1989]. Specifically, rays that carry light directly from a light source to the surface of an object are called *illumination* or *shadow* rays.[6] Similarly, *reflection* rays are the ones that carry light reflected off by a surface, and *transparency* rays are the ones that carry light that has been transmitted through a surface.

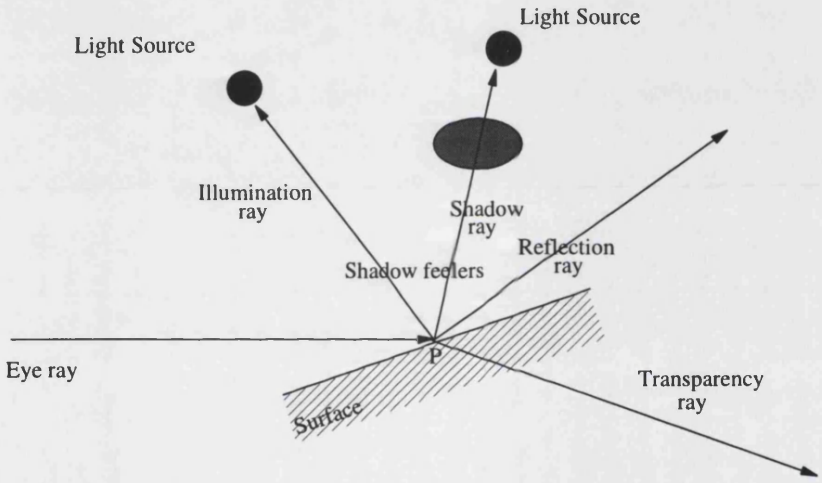


Figure 3.7 Definition of rays

Consider, in Figure 3.7, an eye ray which, in the backward ray tracing philosophy, emanates from the observer towards the viewing frustum. Assume that it strikes at the point ***P*** on the surface of an object in the scene. The light of the eye ray (that the observer perceives) will be determined by the light that illuminates (directly or not) point ***P*** and is either reflected off or, emitted through the surface at ***P*** towards the observer.

[6] Their name depends on the individual case: consider a point on the surface of an object and a light source. If there is a clear, visible, path connecting them directly, this defines an illumination ray. If this path is obstructed by another object, then that point is in shadow and therefore we are talking about a shadow ray.

Light that comes directly at point ***P*** from the light sources specifies whether ***P*** is in shadow cast by other objects in the scene. This is determined with a ray called *shadow feeler*. It is a ray connecting ***P*** with the light sources. If the shadow feeler does not intersect with any object before reaching the source, it is assumed to be an illumination ray illuminating point ***P***. Alternatively, it is a shadow ray since another object is in between ***P*** and the particular light source.

Light striking at ***P*** can be radiated in a given direction, not necessarily unique, with four main different *mechanisms* or *light transport modes*, two of which are the *perfect specular reflection* and the *perfect specular transmission*. These two modes describe the effects of reflection and transparency on a perfectly flat shiny and transparent surface. The other two mechanisms, namely the *diffuse reflection* and the *diffuse transmission*, describe the same phenomena but on rough, imperfect surfaces. The mathematics that model these (not necessarily all) phenomena will compose our shading or *illumination model* which is also called *the rendering equation*.

Therefore, in the above example, light that is reflected at ***P*** towards the eye ray direction is also important. Similarly, light transmitted through the object's surface at ***P*** going to the same eye ray direction is also taken into account. As a result, the appropriate reflection (***R***) and transparency (***T***) rays are calculated. But in order to determine the light of the eye ray (expressed in terms of colour and intensity) the rendering equation needs information about the light that the ***R*** and ***T*** rays carry. This means that the rendering equation should have been applied for these reflection and transparency rays beforehand. This observation justifies the recursive nature of the backward ray tracing algorithm. Moreover, tracing backwards, if these rays (i.e. reflection and transparency) intersect with other surfaces, more rays are involved in the rendering equation thus expanding the ray tree.

Although the expansion of the ray tree should only stop when no more rays are generated (due to lack of intersections with surfaces in the scene), there are cases where the relative location of the objects is such that infinite expansion is demanded by the rendering equation. Whitted [1980] suggested that a fixed tree depth (i.e. a maximum level of tree expansion, or maximum computer storage allocated for holding the ray tree) should be used to prune

the tree. This method is a trade-off between wrongly coloured images if the depth limit is very small and wasted CPU time if the size of the ray tree is too large. The choice of the 'correct' depth is not clear. It greatly depends on the relative position of the objects in a scene and sometimes there are areas in the image that demand great tree depths (e.g. direct involvement of the light sources) and others that need very small tree depths (e.g. surfaces in shadows). One way to overcome this is by using a different limit for every ray tree. Such a technique is called *adaptive tree depth control*; for a given ray tree, expansion will terminate when a child ray does not contribute a significant amount of colour to the corresponding pixel. This *threshold of significance* is arbitrarily chosen and in most applications is taken equal to the *colour resolution* of the viewport.

For example, in a 24 bit frame buffer, where 8 bits are used for each primary colour of the RGB model, the corresponding colour resolution is 2^{-8} of the maximum intensity used. Consequently, intensity variations of less than 2^{-8} cannot be coded, thence do not contribute to the final image. Furthermore, from psychophysics, we can deduce similar values for the threshold of significance that have been obtained from experiments measuring the ability of the human eye to discriminate between two contiguous colour intensities.

Although this technique produces acceptable images, in theory it can be proved that it may be an arbitrarily incorrect approach since we do not know the intensity of the light sources we might encounter during the ray tree expansion. In order to avoid such unexpected errors, we have to assume that the maximum allowed intensity of any light source in a computer graphics scene is set to an arbitrary value, which for convenience it represents the unit intensity $I_{max} = 1$.

Despite this assumption, when there are more than one sources in the scene, it is still possible for certain surfaces to be illuminated by more than one source thus resulting in intensity levels that are larger than the preset limit of I_{max}. Furthermore, there are also certain combinations of reflective and refractive surfaces that may converge light from a source to a particular location in the scene, thus illuminating it with intensity larger than the preset limit. Examples of such surfaces are appropriately positioned convex lenses and paraboloid mirrors. In such cases, we either increase the maximum allowed intensity I_{max} and scale all

intensity values in our calculations accordingly, or we introduce attenuation of light intensity in order to diminish the possibility of such errors, or finally we may truncate all undesired intensity values to the maximum allowed and therefore introduce a small error in the image.

Backward ray tracing, which from now on will be simply called ray tracing, was introduced by [Kay 1979a; 1979b; Whitted 1980] as an extension to the *ray casting* method for hidden surface removal [Appel 1968; Goldstein 1971]. Ray casting is used as a method for determining the visible parts of objects in a scene and is similar to ray tracing. Their difference is that in ray casting the ray tree is not generated but only the initial eye ray is used. Information about the colour of a pixel is gained from the corresponding eye ray regardless of any possible reflection and transparency rays. The shading models used in ray casting were therefore considered *local*, as opposed to the *global* ones introduced by ray tracing [Whitted 1980].

3.4.5 Projection

Since the ray tracing approach is based on the pinhole camera model, the perspective projection is implied by the algorithm; all initial eye rays emanate from the observer and are spread inside the pyramid of vision. Actually, the eye rays may be considered as the lines of the perspective projection.

To implement ray tracing with the orthographic projection, we would simply need to change the definition of the lines of projection (i.e. the eye rays). Specifically, in orthographic projection, all eye rays should be parallel with each other. Moreover, we may assume that they all hit the image plane perpendicularly. Although orthographic projection is simple to implement, it defeats the essence of the ray tracing approach and so is rarely used.

3.4.6 Clipping

Clipping is implicit in ray tracing since we will only encounter the eye rays that belong to the inside of the pyramid of vision. Consequently, all objects that fall outside this pyramid of vision will not intersect with the eye rays. However, since these implicitly clipped objects

are part of the scene, they should not be ignored because they may contribute to the rendering equation because of their reflection, refraction, or shadows cast onto the surfaces of other objects that have not been clipped.

3.4.7 Hidden surface removal

In ray tracing, the problem of eliminating hidden, invisible by the observer, surfaces is addressed by the eye rays. Obviously when an eye ray is fired, the first surface it will hit will be one that is visible by the observer. Therefore, determining whether a ray intersects with any objects in the scene is very critical and the correct expansion of the ray tree depends exclusively on it. From early experiments by Whitted [1980], the CPU time spent for intersection related calculations was in the range of 75% - 95% of the total CPU time needed for a ray traced image to be generated. Although these numbers depend very much on the complexity of the scene, the actual hardware platform and the possible acceleration techniques used, the task of ray – surface intersection still poses a major calculations overhead.

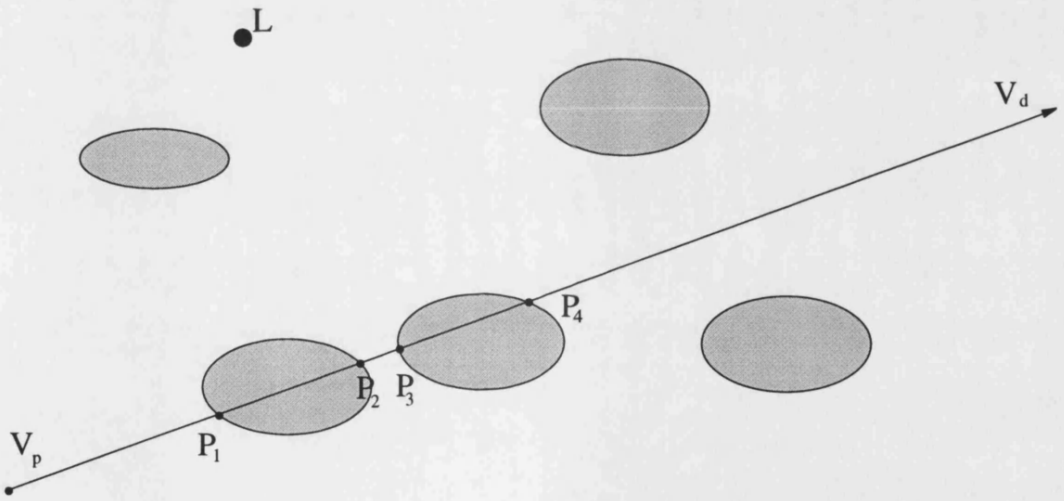


Figure 3.8 The ray - surface intersection

Consider Figure 3.8 that shows a ray passing through a scene. In geometry, this ray is defined as a vector V by its point of origin V_p and its direction V_d. Parametrically, it is defined as $V = V_p + \lambda \times V_d$, $\lambda \geq 0$. The constraint for λ denotes that the ray extends infinitely

to the V_d direction only. In a typical ray tracing implementation, all the objects in the scene will have to be checked whether they intersect with that ray V. All the points P_i of intersection ($P_i = V_p+\lambda_i \times V_d$) will then have to be collected and sorted and the one (say P_j) that the ray hits first (i.e. the nearest to Vp, P_j: $\lambda_j = min(\lambda_i)$) will become the origin for the new child rays. If the original ray V was a shadow feeler, then its direction V_d would be defined by $V_d = L-V_p$, where L is the position of a light source, and we would be interested in finding points P_k that cut the path from Vp to L (i.e. $P_k = V_p+\lambda_k \times V_d$, $0 \leq \lambda_k \leq 1$).

The mathematics involved in a ray - surface intersection calculation may vary considerably in complexity and depend mainly on the modelling approach used. Since the major geometric task of ray tracing is the intersection of objects with rays, any model may be used. In practice we can differentiate between three different approaches to the ray - surface intersection problem which are the *algebraic*, the *geometric* and the *divide and conquer.*

The algebraic approach.

The algebraic approach to intersection problem is appropriate to analytic models. The problem of intersection of a ray vector ($V = V_p+\lambda \times V_d$, as Figure 3.8 shows) with an object (an analytic function) is transformed to an equation (usually a polynomial), the roots of which need to be calculated. This equation is usually expressed parametrically with the vector's parameter λ, and its real roots (if any) will determine the points on the ray vector that intersect with the surface of the object.

For example, a sphere of unit radius, centred at the point (2 , 3 , –4) of the OBSERVED system and a ray V with V_p = (0 , 0 , 0) (eye ray) and direction V_d = (2 , 3 , –4) would produce the following equation:

Vector ray: $V = (0 , 0 , 0)+\lambda \times (2 , 3 , -4) \Rightarrow V_x = 2\lambda,\ V_y = 3\lambda,\ V_z = -4\lambda$

Sphere: $(x-2)^2 + (y-3)^2 + (z+4)^2 = 1$

Points on the vector $V \equiv (V_x , V_y , V_z)$, that also belong to the sphere should validate both equations $(V_x-2)^2 + (V_y-3)^2 + (V_z+4)^2 = 1 \Rightarrow (2\lambda-2)^2 + (3\lambda-3)^2 + (-4\lambda+4)^2 = 1$

This is a second degree equation with roots: $\lambda = 1 \pm \frac{1}{\sqrt{29}}$

For geometrical objects such as the plane, or quadrics (e.g. sphere, ellipsoid, cylinder, etc.) the intersection with a ray vector will result in first or second degree equations, which can be solved analytically. However, if the produced equation is of degree greater than two, the complexity of the problem is significant and approximation techniques are used. Geometrical objects that belong in this category include the torus and the (infinite) helix.

Numerical analysis techniques for solving equations produce approximations to their roots by iteratively guessing a solution and improving on it. This iterative process terminates when a suggested approximation is not far away from the actual root (i.e. their distance is less than a distance epsilon ε). The speed of convergence to a solution depends on the type of the technique used (e.g. Newton Raphson, Regula Falsi, Bernoulli etc.) and of the initial guess of the root.

Moreover, for such a process to converge, a set of preconditions that are not always convenient to prove, has to be met, thus adding more to the calculation overhead. For a complete survey of numerical methods see Apostolatos [1981]. Algebraic methods are generic since the solutions they produce can be parameterized, thus enabling a category of problems to be solved by assigning the appropriate values to these parameters. For example, once the algorithm for calculating the roots of a fourth degree polynomial is determined, the problem of intersecting a ray with any quartic is solved. But such a solution is difficult to determine and it involves a considerable amount of calculations, thence more CPU time.

The geometrical approach

The second approach to solving the ray - surface intersection is the geometric one. With this approach the exact conditions of each problem are exploited and the solutions (points of intersection with a given ray) are calculated only when necessary. For any given situation, certain conditions like *space coherence* and *bounding volume information* are the first to be exploited. With the first condition, space coherence, the relative position of objects in a scene is examined. For example, if a ray starts from a point V_p that lies inside the volume defined by a sphere, then an intersection point (with that sphere) will exist irrespective of the direction V_d of that ray. With the second condition, information about simple geometrical objects can be used to infer results regarding more complex ones. Take, for example a torus

and a sphere that completely covers it. If a given ray does not intersect with that sphere it would not intersect with the torus either. Apart from the above general conditions, others specific to individual intersection problems can be used to avoid unnecessary calculations if knowledge concerning lack of intersection can be gained.

To give an example, consider the problem of intersecting a ray with a sphere where V_p (the ray's origin) is outside the sphere. If the ray's direction points away from the sphere then there is no intersection. With the geometric approach, therefore, only the necessary steps towards a possible solution are actually occurring, thus reducing the amount of CPU time needed. But, on the other hand, geometric solutions are not as generic as their algebraic equivalent. This means that for every type of object a separate algorithm for solving the intersection problem has to be determined thus increasing the amount of code needed.

The 'divide and conquer' approach

This is a combination of the algebraic and geometric approaches so that the advantages of both may be exploited. Bounding volume information is used to determine whether a given ray intersects with the bounding volume of an object. If such an intersection occurs, the object is subdivided into smaller pieces, if possible, and for every piece a new bounding volume (smaller in size) is defined. Then, the same ray is checked for intersection with all the new bounding volumes. For every such intersection, the corresponding piece of the object is again subdivided into smaller pieces and intersection checks are performed again.

This recursive process ends when that ray does not intersect with any of the bounding volumes of the pieces of the object or, when the size of the intersecting bounding volume is smaller than an arbitrarily chosen limit. In that latter case, the intersection point is assumed to be in the centre of the bounding volume. This approach, is convenient to use with surfaces that are modelled by a recursive function (e.g. different types of splines). A typical example of this approach can be found in [Whitted 1980], where ray tracing is applied on bicubic patches using a recursive evaluation algorithm proposed by Catmull and Clark [1978].

3.4.8 Shading

When a ray hits the surface of an object, its direction as well as its colour will change. From optics, the geometry of reflection and refraction have been modelled, while from quantum mechanics explanations about the colour changes are given. In this section, the geometrical issues will be presented first. Next an attempt to explain some of the spectral changes will be given. Finally, an advanced shading model capable of simulating all four colour transport modes will be briefly presented.

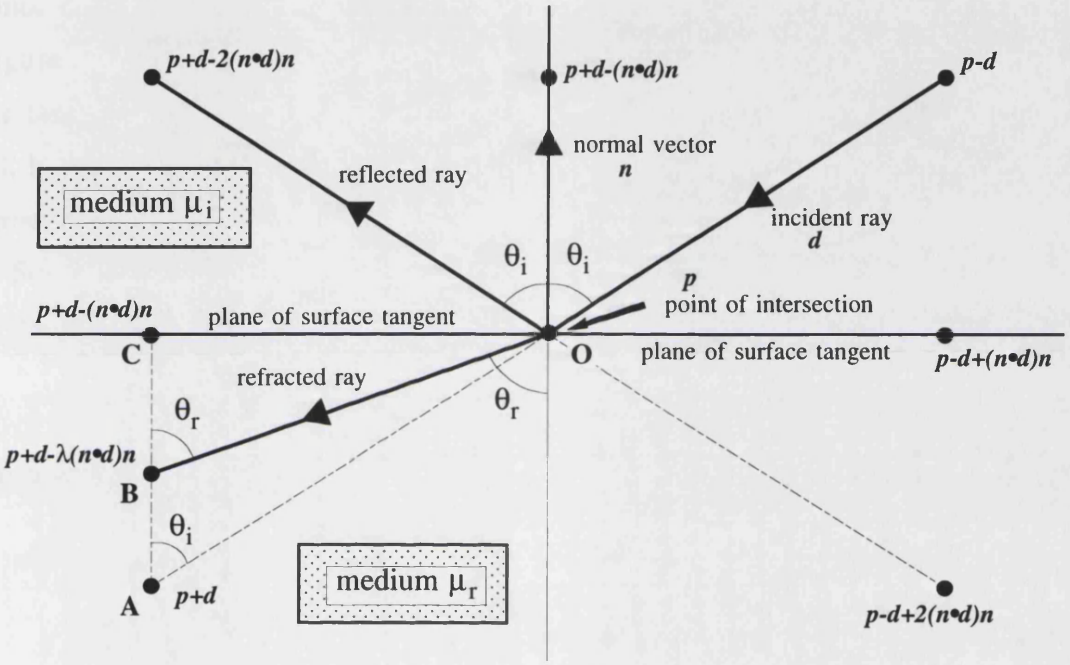


Figure 3.9 The ray - surface interaction

Consider Figure 3.9. Suppose that an incident ray with unit direction vector $\boldsymbol{d}$ hits a surface at point $\boldsymbol{p}$, where the normal pointing out of the surface is unit vector $\boldsymbol{n}$. From elementary physics we know that the *angle of incidence*, θ_i, the angle made by the ray with the normal, equals the *angle of reflection*. Assuming that the surface is acting as a plane mirror at point $\boldsymbol{p}$ (labelled $\boldsymbol{O}$ in the figure) and that the directions have the senses given in Figure 3.9, we can see that point $\boldsymbol{p}-\boldsymbol{d}$ is reflected into point $\boldsymbol{p} - \boldsymbol{d} + 2(\boldsymbol{d} \bullet \boldsymbol{n})\boldsymbol{n}$, from which we can ascertain the direction vector of the reflected ray to be $\boldsymbol{d} - 2(\boldsymbol{d} \bullet \boldsymbol{n})\boldsymbol{n}$. The function denoted by $\boldsymbol{d} \bullet \boldsymbol{n}$ is the *inner product* between vectors $\boldsymbol{d}$, $\boldsymbol{n}$.

Now we emulate refraction by considering the same incident ray as it passes through the surface. Suppose the ray passes from the incident medium with *refractive index* μ_i into the refracting medium with refractive index μ_r: we call the relative refractive index from the second to the first medium $\mu = \mu_r / \mu_i$. If the refracted ray makes an angle of refraction θ_r with the normal (in the opposite sense) $-\boldsymbol{n}$, then by Snell's Law:

$$\frac{\sin\theta_i}{\sin\theta_r} = \frac{\mu_r}{\mu_i} = \mu$$

Since the normal, and the incident and refracted rays all lie in the same plane, referring to Figure 3.9, we can treat refraction as though the *straight-through* ray is pushed up towards the tangent plane if $\mu < 1$, or down away from it if $\mu > 1$. In the figure, point $\boldsymbol{p}+\boldsymbol{d}$ (labelled $\boldsymbol{A}$), lying on the straight-through ray, is pushed up towards a typical point $\boldsymbol{p}+\boldsymbol{d}-\lambda(\boldsymbol{d} \bullet \boldsymbol{n})\boldsymbol{n}$ (labelled $\boldsymbol{B}$) that is dependent on the value of λ ($=|\boldsymbol{AB}|/|\boldsymbol{AC}|$), which in turn depends on μ. Setting λ to unity fixes a point labelled $\boldsymbol{C}$ that lies on the plane tangential to the surface at the point of incidence. Since $\boldsymbol{d}$ (and $\boldsymbol{n}$ for that matter) are unit vectors, then:

$$|\boldsymbol{OA}| = 1, \quad |\boldsymbol{OC}| = \sin\theta_i = \mu \sin\theta_r \text{ and thus } |\boldsymbol{OB}| = \mu$$

Remembering that $\boldsymbol{n}$ and $\boldsymbol{d}$ are unit vectors, we can then calculate:

$$|\boldsymbol{AC}| = |\cos\theta_i| = |(\boldsymbol{d} \cdot \boldsymbol{n})| \quad \text{and}$$

$$|\boldsymbol{BC}| = \mu \cos\theta_r = \mu\sqrt{(1 - \sin^2\theta_r)}$$

$$= \sqrt{(\mu^2 - \mu^2\sin^2\theta_r)}$$

$$= \sqrt{(\mu^2 - \sin^2\theta_i)}$$

$$= \sqrt{(\mu^2 - (1 - \cos^2\theta_i))}$$

$$= \sqrt{(\mu^2 + (\boldsymbol{d} \cdot \boldsymbol{n})^2 - 1)}$$

Hence $$\lambda = \frac{|\boldsymbol{AC}| - |\boldsymbol{BC}|}{|\boldsymbol{AC}|} = 1 - \frac{|\boldsymbol{BC}|}{|\boldsymbol{AC}|}$$

$$= 1 - \frac{\sqrt{(\mu^2 + (\boldsymbol{d} \cdot \boldsymbol{n})^2 - 1)}}{|(\boldsymbol{d} \cdot \boldsymbol{n})|}$$

We are of course assuming that both θ_i and θ_r are acute angles, $(\boldsymbol{d} \bullet \boldsymbol{n}) \neq 0$, and the values under the square root symbol are non-negative. We can now find the value of λ, and hence the refracted ray. If the values under the square root are negative then we assume *total reflection.*

Apart from the above, other solutions to the light transmission problem can be found in [Heckbert 1989]. Heckbert compares three solutions with regard to the number of primitive calculations (i.e. additions, multiplications, divisions and square roots) needed to implement thus providing a framework for choosing the quickest technique.

The above solutions model the geometric aspects of the optical phenomena only (i.e. reflection and transmission). What is still needed to compose the complete rendering equation is to simulate the reaction of light in terms of its spectral composition (i.e. intensity on every visible wavelength). These models, should be able to explain phenomena like the *colour shift* that occurs in perfect specular reflection, or the *colour absorption* and the reason that we actually see objects.

When a ray strikes on the surface of an object what actually happens is that photons interfere with the *atoms* of that object. From quantum mechanics, it is known that atoms vibrate and can be characterised by the amount of energy they carry. Their energy may only take a few specific values called *energy levels* or *energy states*. According to this theory, an atom can take or give specific amounts of energy thus moving upwards (or downwards) in the permissible energy states. When a photon approaches an atom, due to the phenomenon of *sympathetic resonance*, some energy will be exchanged: atoms with *resonant frequency* close to the photon's frequency will be more easily *excited*, and *absorb* almost all the photon's energy. Accordingly, big differences in the frequencies will result in small amounts of exchanged energy.

When a photon arrives with energy insufficient to boost an atom to the next energy level, its energy is absorbed (by the atom) and converted into heat. But if the energy of the photon (transferred sympathetically) is just enough to enable the atom to move to a higher energy state then the photon will disappear (since it gave all its energy) and the atom will oscillate

at a higher level. The atom cannot stay at its new excited level indefinitely and after a while returns back to its previous energy state thus emitting a new photon with energy equal to the difference of the two energy states and with frequency similar to absorbed one.

This phenomenon if seen from a macroscopic view appears to be the reflection of light from a surface. This is actually the reason for seeing coloured objects like, for example, a blue ball; blue photons are reflected back while all the other colours are absorbed at the ball's surface and transformed into other forms of energy (mainly heat). On the other hand, from a microscopic point of view, the following assumption is also made: the surface of an object is composed of many tiny flat reflectors also called *microfacets*. The distribution of their orientation will determine how glossy and shiny a surface is.

In a shiny flat surface, almost all the microfacets have the same orientation, while in a less smooth surface, microfacets with any orientation have the same probability to appear. If light comes from a direction almost tangential to the surface, it will be either blocked by the microfacets, or reflected by the appropriately orientated microfacet following the laws of reflection. If a light ray hits the surface with a small angle of incidence, then it will be reflected for a while amongst the microfacets before it leaves off the surface towards the appropriate (by the reflection laws) direction. In the latter case, the absorption of photons of certain frequencies will become apparent since many 'microreflections' will have occurred. This colour shift, that occurs in specular reflection, is expressed by the Fresnel function $F(\lambda , \theta)$ where λ is a given wavelength of visible light and θ is the angle of incidence at the appropriately orientated microfacet [Foley *et al.* 1990].

By modelling both the geometrical and the optical reactions of light when it hits a surface, a complete shading model emerges. In computer graphics, there exist many different shading models that simulate the above interactions. The simple ones cater only for perfect specular reflection, while the more sophisticated can simulate all four light transport modes (section 3.4.4). What all these models have in common is that they differentiate between light coming directly from a light source and light coming indirectly from other surfaces through reflections and/or transmission.

A typical global illumination model was first introduced by Whitted [1980]. This illumination model for a given point of a ray - surface intersection calculates the ambient($I_{ambient}$) light from the scene and the diffuse ($I_{diffuse,i}$) and specular ($I_{specular,i}$) reflections of the i^{th} point light source. The global illumination model is then given by the recursive equation:

$$I_\lambda = I_{ambient} + \sum_{1 \le i \le m} [I_{diffuse,i} + I_{specular,i}] + k_s I_{refl,\lambda} + k_t I_{tran,\lambda}$$

Where i denotes the one of the m point light sources and k_s, k_t are the specular reflection and transmission coefficients of the materials involved. The wavelength λ denotes that we can sample this equation in the red, green and blue primaries of the RGB colour model. Hall [1983] suggested, a more accurate equation that also accounts for Fresnel's law.

3.4.9 Mapping

The final stage in visualisation is what we called mapping of points into pixels. This stage is also implicit in the ray tracing algorithm. It takes place in the definition of the eye rays. There, we assumed that a subwindow on the image plane will correspond to exactly one pixel on the viewport. Therefore, by firing one eye ray towards the centre of each subwindow, we can determine the coordinates of the pixel we will eventually paint.

3.5 Problems with visualisation

Apart from all the merits and disadvantages we presented for each particular visualisation approach, they all suffer from the problem of *aliasing*. It is inherited by the definition of computer graphics which represents an analogue (continuous) world with digital (discretised) means. This problem appears as four different phenomena, namely, *precision*, *spatial aliasing*, *colour aliasing* and *temporal aliasing*, all of which distort the resulting image. In this section we will present all four phenomena and suggest remedies. For our convenience, we will assume the ray tracing visualisation approach, unless we state otherwise.

Precision

Each computer can handle numbers (integers or reals) up to a certain degree of exactness. With regard to integers there exist a range of permissible values beyond which we need to use special processor commands and *extended precision arithmetic.* With regard to real number representation, computers are using *interval arithmetic* where a range of real numbers is represented by one number only, the *representative*, which is usually the centre of the interval. The rest of the numbers in this interval cannot be represented precisely, but are approximated with that unique representative of the interval, thus introducing *precision* or *rounding* errors. The IEEE has produced standards for number representation and arithmetic and the rounding error (i.e. half the length of the interval) for single precision real number representation is in approximately 10^{-7}. Using extended precision representation, this error may even fall below 10^{-11} but with considerable needs in memory space and CPU time.

When calculating the intersection of a ray with an object, it is not uncommon that tens of multiplications, divisions or even square roots are involved, especially when iterative approximations are used. As a result, the tiny representation errors accumulate and increase in the intermediate stages of the calculations, thus producing (at the end) a solution of debatable accuracy (i.e. errors in the order of 10^{-2} or larger). In such cases, that are very often encountered, a point that theoretically is assumed to be on the surface of an object, can be found far away from it (either inside or outside).

Furthermore, the use of such a misplaced point as the origin for new child rays adds to the problem of accuracy since errors will amplify when propagated through the lower levels of the ray tree, and an incorrect ray tree will eventually be produced. The most common problem in ray tracing applications is that such a point is misplaced at the inside of the surface of an object and the new rays that originate from it hit the same surface again and again, thus deceiving the rendering equation and producing a peculiar and incorrect texture.

For the problem of precision many solutions have been suggested. Some treat the numbers as intervals since this is the actual representation of numbers in computers. With such a method, numbers that differ less than ε (an arbitrarily small distance called *epsilon*) are treated as being equal. This epsilon is chosen to match the computer's precision of number

representation (i.e. approximately 10^{-5}) and in many applications is assumed to be constant. But this only partially solves the problem of precision since if application distances in the order of 10^{-3} are common, a value of $\varepsilon = 10^{-5}$ is relatively significant and it will still produce problems. Adjusting ε according to the order of magnitude of the numbers (i.e. scaling) used seems to solve most of the problems but there is still no guarantee that such an epsilon will always exist to be accurately represented by the computer (i.e. if $\varepsilon = 10^{-10}$ is needed, then a common 16 bit system cannot represent it so that $1.0 + \varepsilon \neq 1.0$).

Another approach that is partially based on the existence of ε (constant or not) is the following. After a solution (i.e. an intersection point) has been found, an iterative approximation technique (of accuracy ε or smaller) is employed to improve on that solution. In such cases, the CPU time needed for eventually determining an intersection point increases considerably.

Finally, another totally different approach can also be used for the precision problem. It is based on the logical implications of what an algorithm is meant to do (e.g. to determine a point on a surface that will be used as an origin for new rays). With this method, if an intersection point is going to be used as the origin of a ray that travels outside the (intersected) surface, then it is moved an arbitrary small distance outside that surface so that the problem of hitting the same surface twice (or more) is certainly avoided. Accordingly, points used for rays that travel inside an object are moved to the interior of that object. This arbitrary dislocation of points, although succeeding to avoid the primary problem of wrong intersections, produces inaccurate images. This flaw becomes obvious in cases where objects are very near to each other, or where surfaces with high curvature are involved.

Concluding the discussion on the precision problem, we can remark that there is no unique preferred method for avoiding precision problems. The rule of thumb is to use a mixture of them all. The criteria for which particular to use more extensively should include the type of objects used, the order of magnitude of the numbers involved, and the relative importance of producing accurate images (e.g. medical/scientific versus advertisement).

Spatial aliasing

The use of a discrete medium such as the viewport (an array of a finite number of pixels) to depict a continuous analogue image of a scene will result in *jagged* edges (looking like staircases) or even lost objects. It is a typical problem of sampling which is called spatial aliasing.

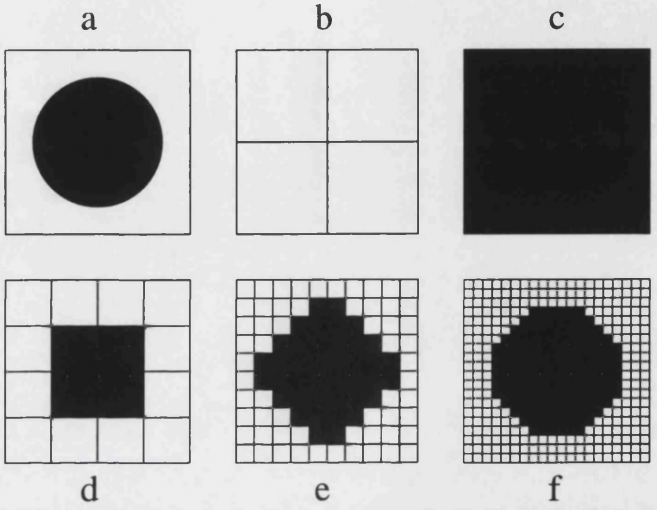


Figure 3.10 Spatial aliasing

To better understand spatial aliasing, let us consider Figure 3.10. In this figure, the image of a ray traced sphere is shown on viewports of different resolutions. Assume that the sphere's centre is projected on the centre point of the image plane and the complete image (the projected circle) falls inside it, as Figure 3.10a, shows. If the resolution of the viewport is only 2×2 pixels, a simple ray tracer would miss the sphere if its radius was smaller than a certain distance (Figure 3.10b). This problem is inherent in the nature of ray tracing, since by definition[7] all four (corresponding to the pixels) eye rays miss that sphere. Therefore, in the general case, an arbitrarily large object can disappear from the viewport if it falls inside the infinite pyramid produced by the thus produced four eye rays (Figure 3.11).

Whitted [1980], in order to avoid missing objects, suggested that for every object in the scene at least one ray should hit it unless it is hidden by other objects. With this suggestion, the image of Figure 3.10c, may emerge. By increasing the resolution of the viewport,

[7] In the standard ray tracing, as it has already been mentioned, an eye ray is defined by the vector that emanates from the observer and passes through the centre of the corresponding to the pixel subwindow on the image plane.

Figure 3.10d (4 x 4), e (10 x 10) and f (20 x 20) are produced. What is common to all these images is the quality of the perimeter of the re-produced circle which is not smooth but follows an approximated path determined by the regular grid of pixels of the viewport. This appearance of unwanted jagged edges is one of the most annoying phenomena characterizing not only ray tracing but the whole spectrum of computer graphics applications.

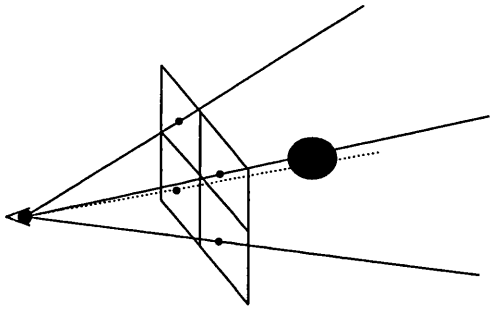


Figure 3.11 Losing objects from the scene

To avoid this problem of spatial aliasing (jagged silhouettes) many suggestions have been made. But what they all have in common is that they increase the number of eye rays that correspond to every pixel, a technique called *oversampling*. As a result, the colour of a pixel is calculated by a *weighted average* of the colour of all the rays that correspond to that pixel. With the simplest method, namely *supersampling*, between three to nine (usually 5) eye rays are fired for every pixel. They all originate from the observer, but pass through different points of the subwindow on the image plane. The pattern of these points is the same for every pixel, thus producing a regular grid of density higher than the viewport's (Figure 3.12a).

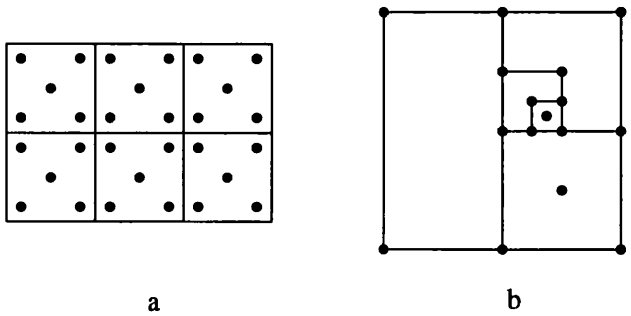


Figure 3.12 Regular and adaptive supersampling

However, there are two main disadvantages with this method. First, it does not eliminate the aliasing problem but only reduces it, since the eye rays still form a regular grid on the image plane and therefore jagged shapes may be noticeable. The second drawback is the waste of CPU time that occurs from the firing of five or nine eye rays into areas of little or no change in the resulting colours.

As a result, another method, namely *adaptive supersampling*, emerged. With this method a minimum number of eye rays, usually five, is fired for every pixel, at fixed, relative to the subwindow, locations (Figure 3.12b). If these rays return colours with significant differences, the regions of the subwindow that produce these colour changes are treated as subwindows on their own and five more eye rays are fired for each of these. As Figure 3.12b shows, the results of two of the five rays have already been calculated at the immediately higher level of subdivision and there is no need for them to be re-calculated.

This recursive subwindow subdivision and subsequent ray firing process stops after a maximum level of subdivision is reached, or when the returning colours are not different.[8] This method produces better results than the constant supersampling and, depending on the scene, may also be faster since CPU time is only spent in areas of interest (i.e. where colour changes are significant). The drawback of this method is that although being adaptive, still aliasing effects exist since all eye rays pass through a regular grid but of a higher resolution.

Another approach, that avoids firing rays at regular grid locations, is the one called *stochastic supersampling* and it is based on the Monte Carlo method [Halton 1970]. With this method, for a given pixel, several rays (usually nine) are fired passing through randomly selected points within the corresponding subwindow. As a result, jagged edges are not characterized by the patterns of a regular grid but by random patterns also known as *noise*.[9]

---

[8] The criterion of colour difference seems better compared to maximum number of subdivisions, in terms of quality, since whenever there is significant colour difference it is implied that the rays hit near the edge of a surface, or a highlight, and therefore a denser sampling rate is necessary. But in order to achieve faster results either the threshold colour difference is enlarged or a combination of both criteria is used.

[9] This method conforms to the observation that although the human eye consists of a finite number of photoreceptors [Williams 1983], they do not form a regular grid but follow a Poisson disk distribution as Yellott [1983] suggested.

With the help of digital filters, stored in look-up tables or calculated in real time, the colour of every pixel is determined [Amanatides 1987a; Cook 1989].

By extending the concept of stochastic supersampling into all types of rays (i.e. shadow, illumination, reflection and transparency) a new category of ray tracing algorithms called *stochastic*, or *distributed*, or *probabilistic* ray tracing (as opposed to the standard *deterministic* one) can be defined. With distributed ray tracing, apart from eliminating the spatial aliasing problem, a "whole range of fuzzy phenomena" [Cook 89], can be simulated: *blurry reflection*, *blurry transparency*, *penumbras*, and *depth of field* can be modelled by the distribution of reflection, transparency, shadow/illumination and eye rays over a lens configuration.

Temporal aliasing

With the distribution of eye rays over time, *motion blur* may also be modelled. However, the production of frames at discrete time intervals for the creation of an animation sequence will result in objects moving stepwise and not continuously.[10] This is especially noticeable when visualizing fast moving objects where in cinema films their silhouettes appear blurred (*motion blur*). For example, we can have the illusion that the wagon's wheels are revolving in the opposite direction of the wagon.

Colour aliasing

The third problem of sampling continuous events in discrete media is that of colour aliasing. In a similar fashion to spatial and temporal aliasing, sampling also occurs when the rendering equation must determine the colour information at a particular location. However, such information should be in the form of intensity distribution along the range of visible frequencies (i.e. visible spectrum). But this requirement would complicate the implementation of a shading model immensely since it would necessitate operations on spectra responses. Consequently, using a colour model such as the RGB, the rendering equation approximates the interaction of colour intensity spectra by determining equivalent colour intensities on the primary red, green and blue frequencies only. An obvious effect of

[10] This phenomenon was first noticed at the early cinema films. Since then, experiments have shown that the human brain cannot differentiate between images changing more often than 16 times a second (approx.).

colour aliasing is therefore that in computer graphics transparent surfaces do not exhibit the 'rainbow effect' of *light spectral analysis* due to the variation of the refractive index on different light frequencies.

3.6 Acceleration techniques

Another problem with visualisation algorithms in general but particularly with ray tracing, is the amount of computing time needed. Ray tracing of scenes is a considerably longer process compared with the rest of the visualisation approaches. Moreover, extensions like stochastic ray tracing would become significantly (e.g. more than 100 times) more time consuming than the standard deterministic ray tracing. Detailed results of timing ray tracers have shown that the time spent in intersection-related calculations amounts for most (over 60% approximately) of the total computing time needed to render an image. As a result, most of the research in accelerating ray tracing has been directed towards the ray - surface intersection.

In a broad classification of ray tracing acceleration techniques we can differentiate between two main categories that we will call *intersection-related* and *general* (or, *intersection-unrelated*). Moreover, another classification schema that will divide all acceleration techniques into *hardware oriented* and *software oriented*, may also be imposed. The boundaries of this categorisation may not always be clear. In general, a software oriented technique will be considered an algorithm that can be implemented on a general purpose computer and is transportable to any other common purpose language implementation. A hardware oriented technique will be the one that exploits the particular architecture of a special purpose computer system. In this document, we will use the first classification and, for each category, references concerning the second classification will be given.

Intersection-related acceleration techniques can be further divided into two sub-categories, namely, *faster* and *fewer intersections*. Achieving faster intersections by software can occur by fully exploiting spatial information so that only the necessary steps towards the solution of the intersection problem are calculated. Other techniques involve the transformation of

either the ray or the surface (or sometimes both) to a new coordinate system so that intersection calculations are more convenient to perform. Avoidance of expensive calculations like divisions[11] or square roots and the extensive use of look-up tables also fall into this sub-category.

Hardware techniques for faster intersections involve the utilisation of CPU registers (for holding variables that are constantly needed) and Programmable Array Logic (PAL) chips. The algorithm of ray - surface intersections is written in microcode and incorporated into a PAL chip, thus considerably reducing the execution time needed. Ullner [1983] was one of the first people who examined theoretical hardware configurations including massive use of custom-made VLSI circuits.

In the second sub-category, fewer intersections can be realized by both software and hardware. One technique is the use of *bounding volume hierarchies*; objects that are simple to intersect are introduced to bound the more complex actual objects of a scene. Single objects, aggregations of objects, or even bounding volumes can be again bounded, thus resulting into *bounding hierarchies*. If a ray does not intersect the bounding volume at a certain level in the hierarchy, objects bounded by that volume need not be examined at all for that particular ray.

Another technique aiming at fewer intersections is *spatial subdivision*. Here, the space of the viewing frustum that encloses objects of the scene is subdivided to produce 'sub-scenes' of reduced complexity. *Uniform* and *adaptive* (*non uniform*) space subdivision techniques exist. For the uniform, the scene is bounded by a rectangular parallepiped (usually a cube), or a trapezoid with one of their facets being parallel to the image plane. This bounding volume is then subdivided into $n \times n \times n = n^3$ smaller similar volumes (parallepipeds or trapezoids respectively) thus producing a three-dimensional grid of such subvolumes.

[11] In the majority of processors, division of real numbers takes more CPU cycles than multiplication. Therefore, if the division of many numbers by the same denominator is required, it is faster to calculate the inverse of the denominator and multiply this to all the numbers needed.

At a preprocessing stage, for every subvolume, a list of all the objects (namely the *object list*) that intersect or are completely enclosed by it, is generated. While ray tracing, the path of subvolumes that a ray travels through is calculated [Amanatides 1987b; Fujimoto 1985; Fujimoto 1986]. Given a ray, we first find its path along the subvolumes. Then, for every subvolume in the path — starting from the nearest to the origin of the ray — we intersect the ray with all the objects in the subvolume's object list. When we detect an intersection we no longer trace this ray along the path of subvolumes.

With this technique we considerably reduce the total number of intersections needed per ray. The speed gains depend on the complexity of the scene and the size of the subvolumes. Although this technique reduces the total execution time, some time delays are added at the preprocessing stage, and the *subvolume traversal* (while following a given ray). Moreover, time is wasted while visiting subvolumes with empty object lists.

As an improvement to the problem of visiting empty subvolumes, *non uniform space subdivision* techniques have been introduced. Here, the initial bounding volume is always a cube. At the preprocessing stage this bounding volume is initially[12] subdivided into eight equal subcubes. If for a given subcube the corresponding object list is not empty, this subcube is recursively subdivided into eight equal sized subcubes thus producing a structure similar to the octree modelling approach. This recursive subcube subdivision ends when a subcube produces an empty object list, or its corresponding object list contains less than a threshold number of objects, or when the subcubes reach a minimal size. With this technique, time spent in visiting empty subcubes is reduced. Nonetheless, the algorithms used to determine the path of a given ray through the variable size subcubes are more complex and time consuming than in the uniform case.

With spatial subdivision, there are three problems that have to be resolved or avoided. The first problem is concerned with precision. Although Amanatides [1987b] suggests a fast and effective voxel traversal algorithm for the uniform spatial subdivision, he did not explain what will happen if a ray passes through an edge or even a vertex of one of the subcubes.

[12] The only time that there is no need to subdivide the initial cube is when ray tracing the empty scene (i.e. there is no object in the scene).

We have adopted an amendment to Amanatides' algorithm which comprises an extra check to ensure that our results are always consistent with our notation (i.e. which subcube owns the points that belong on its facets).

The second problem refers to the case of finding an intersection point that belongs to another, neighbouring subcube. Here, the extra check that an intersection point belongs to the subcube currently being processed, is adopted. The third problem relates to the fact that while visiting neighbouring voxels there is danger of checking the same object with the same ray more than once. Arnaldi [1987] introduced the concept of the *mailbox* where information concerning intersection results between rays (all rays are identifiable) and objects, is kept for further use.[13]

Spatial subdivision has also been exploited from a hardware point of view. Specifically for ray tracing, Kobayashi, Nakamura and Shigei [1987] suggested a parallel configuration of *intersection processors*. Each such intersection processor is responsible for the ray - scene intersections that occur within a particular volume of space which we will call *subcube*. Specifically, each intersection processor is assigned one (or more) subcubes which are determined at a preprocessing stage by an adaptive space subdivision algorithm taking into account the spatial coherence of the scene. Then, when a ray passes through a particular subcube, the intersection processor responsible for this subcube will test for possible intersections. Information about which subcubes a ray passes through, is essential in order to assign the intersection task to appropriate processors and is facilitated by the use of *face-neighbour quadtree* data structures. In this way face-adjacent subcubes of identical size could be identified quickly. This method was named *adaptive division graph* and the resulting hardware configuration had the form of a six-dimensional hypercube. Dippé and Swenson [1984] used a uniform subdivision process to allocate one subcube to each of the $n \times n \times n$ three-dimensional processor organisation. Then according to the *load* of each processor, the size of the subcubes changed so that no processor was idle. Other approaches may be found in [Nemoto 1986; Cleary 1985].

[13] He introduced it in the context of Constructive Solid Geometry, but it can be used for this problem as well.

Additional techniques used for achieving fewer intersections are the *directional* techniques like the *light buffer*, the *ray coherence* and the *ray classification*. These techniques use the concept of the *direction cube* in order to determine a subset of all the objects that a given ray is likely to hit. A direction cube is a cube in which the normals to all its facets coincide with the axes of an orthogonal three-dimensional coordinate system (i.e. the OBSERVED system). For a more detailed presentation see Arvo [1989].

The second category (intersection-unrelated) of ray tracing acceleration techniques, can be subdivided into three sub-categories, namely, these aiming at firing *fewer rays*, these using *generalised rays* and these exploiting *parallelism* (or *concurrency*). Here the differentiation between hardware and software oriented issues is more clear. Apart from using standard computer science tricks (e.g. assembly language) to write faster code, the first and second sub-categories are software oriented. The third relates mainly to the hardware, although parallel architectures are first simulated and tested in software on general purpose machines.

First we will examine techniques that aim at firing fewer rays. The use of *adaptive tree depth control* for the expansion of the ray tree, falls into this sub-category. Another technique is the *statistical supersampling* which is related to the aliasing problem. Here, for a given pixel, a minimum number (usually three) of eye rays are initially fired at random directions inside the corresponding subwindow as in stochastic supersampling. Then, a statistical test is applied to determine whether more rays are needed. By adjusting the test, fewer rays per pixel are fired without significantly reducing the quality of the image.

Another technique in this sub-category is the *generic ray* [Bowyer *et al.* 1987; 1989]. With this technique, the ray tracing approach considers only one ray, the *generic*. This is a ray with its origin and direction treated symbolically, as parameters to a generic vector definition. The intersection problem is then solved symbolically only once for the whole scene, using a symbolic manipulation mathematical library such as NAG. In this way, the mathematical description of the projection of the scene on the viewplane is determined and described analytically. Then, by assigning the appropriate values to this parameterized symbolic solution, the complete image emerges.

The time needed to render an image is significantly smaller compared to the time needed to solve the generic intersection problem. However, such a generic solution is extremely complex to determine. Apart from this demand for enormous symbolic calculation power, the approach of the generic ray is based on the intersection of the scene with eye rays only. Consequently, it is a *ray casting* approach that is unable to implement a global illumination model.

The second sub-category of (intersection-unrelated) ray tracing acceleration techniques is concerned with the *generalized rays*. Also here the definition of a ray is changed. It is no longer considered to be a vector, but a set of vectors. The underlying assumption is that eye rays that are close to each other (both in origin and directions) are likely to hit the same objects with the same sequence, thus producing 'similar' ray trees. As a result, information gained from previous intersections may be exploited.

Amanatides [1984], introduced a new approach to ray tracing, called *cone tracing*, where a ray is assumed to be a packet of vector rays forming a cone with its apex at their common origin and with a circular cross section. Heckbert and Hanrahan [1984], in their *beam tracing*, assume that a ray is a packet of vector rays forming a cone but, unlike cone tracing, they assume an arbitrary polygonal cross section. Shinya, Takahashi and Naito [1987] introduce *pencil tracing*, where a ray is assumed to be a set of vector rays all being in the vicinity of a main vector ray called the *axial ray*. The geometrical models used with the pencil tracing method are simple object like planes, polygonal facets and spheres.

The multicomputer LINKS-1 [Nishimura 1983], is one example of hardware architecture in which ray tracing is exploited. It is a set of 64 *node computers* that are controlled by a single *root computer*. The root can dynamically re-configure the connections of all the nodes thus producing many different organisations. Goldsmith and Salmon [1985] examined different potential implementations of ray tracing onto a hypercube configuration. Atkin, Ghee and Packer [1987] examined the implementation of ray tracing onto different configurations of transputers. They noticed that in configurations between 1 and 80

transputers the performance gain was strongly linear[14] (100% - 95.5%). They also introduced a set of procedures that could make their system fault tolerant. In this way, if a transputer fails, the adjacent transputers will detect the fault, report it, and compensate for the (possibly) lost data. Another example is the *ray casting engine* which is optimized to visualize models that describe objects as sets of line segments, as we explain in chapter four.

In this chapter we first proposed a framework, consisting of five stages, as Figure 3.1 presents, for examining visualisation approaches in computer graphics. Then we presented the three most popular approaches, namely the polygonal mesh, the octree and the ray tracing. For each of these approaches we described their principles and examined their advantages and disadvantages. Additionally for every stage of our discussion framework we analysed different alternatives we may apply and discussed their suitability to specific problems.

We believe that the first three chapters of this presentation have provided the reader with the essential context within which the rest of this dissertation will evolve. In particular the first chapter gave us the necessary background knowledge and established a ground of understanding with the reader. Then in chapter two we presented the main issues of modelling, while in this chapter we addressed the issue of visualisation.

Continuing this dissertation, in the next chapter we will discuss the major research trends in the literature that address similar modelling and visualisation challenges that we intend to tackle in our own research. In chapter five we proceed with our suggested modelling approach, starting first by presenting the underlying theory and then by providing a wealth of examples. Then in chapter six we discuss the visualisation approach that we believe is suitable to match the models that we generated in chapter five. In the final chapter (chapter seven), we discuss two different but complementary issues, criticisms and further directions.

[14] The concept of linearity in the context of parallel computer architectures relates to the percentage of performance gain when increasing the number of processing units used. A 100% linear system would double its performance when the number of processing units is also doubled.

Chapter 4 Current trends in implicit modelling

4.1 Origins of our research

In this chapter we will present the research trends that have appeared in the literature and follow tracks, similar in principle, to the one we are pursuing. In most cases the techniques that we will present here are application-driven solutions to specific modelling challenges and cannot be seen to represent 'generic' approaches for modelling in computer graphics.

These modelling techniques are presented here and not in the general review of the modelling approaches of chapter two, for two reasons. First because they are conceptually closer to the modelling approach developed in this dissertation, and second because they will help us to establish — through a discussion of their inefficiencies and inadequacies — the need for our research. For this reason, we have selected an assortment of nine different modelling techniques that partially address the modelling problems we are also concerned with. Where appropriate, we will compare and contrast these techniques with the modelling approach we propose later in this dissertation.

The techniques reviewed in this chapter represent three main research trends. The first is directly related to the use of distance as a field generator. Consequently the issue of combining fields together, an operation called *interference*, or *confluence*, will also be presented. This trend characterises the techniques of *soft objects*, *skeletons*, *blends by displacement*, *distance fields* and *colour superposition*. The second trend refers to the exploitation of mainly algebraic functions that modulate a simple geometrical object or set of objects. In this category fall the techniques of *sphere plots* and *convolution*. The third research track is concerned with the issue of constraint-based tessellations and is represented by the technique of *Voronoi tessellation* and *Delaunay triangulation*. Finally we will also present the technique of *ray-representations*. This modelling technique does not fall in any of the three research trends but it provides us with a useful perspective for manipulating objects as sets of line segments.

4.2 Soft Objects

In computer graphics, a still image of a scene is not always sufficient to depict the exact nature and consistency of the materials of the objects. However, in computer animation, moving objects will interact with their environment and, depending on the material of their construction, their shape, or that of the matter they interact with, may need to be modified according to the laws of physics (e.g. gravity, elastic collision, etc.).

The definition of *soft objects* came as a response to the need of modelling objects that change their shape in order to follow the constraints of their environment, thus modelling a more natural behaviour of matter. The term *soft objects* was coined by G. Wyvill, C. McPheeters and B. Wyvill [1986a; 1986b; 1986c].

Their technique is based on the production of *iso-surfaces*[1] determined by a set of analytical functions. Specifically, they assume that there exist a set of independent control points. Each such point is responsible for generating a field according to a function $C(r)$ such as: $C(r) \in [0.0, 1.0]$, $C(0.0) = 1.0$, $C(R) = 0.0$, where r denotes the distance from a control point, and R is the maximum distance beyond which the contribution of that particular control point to the field is null. The function $C(r)$ is also assumed to be continuous in the interval $[0.0, R]$, and its first derivative at either side of the interval is zero. Another arbitrary restriction to the function is that $C(R/2) = 0.5$.

As an example, in one of their papers [G. Wyvill *et al.* 1986b], they suggest the function:

$$C(r) = -0.444\frac{r^6}{R^6} + 1.889\frac{r^4}{R^4} - 2.444\frac{r^2}{R^2} + 1$$

This function is similar to those used by Blinn [1982] to model fields of electron density as produced by atoms and propagated through molecular structures. It is also the same function that Bloomenthal and Wyvill [1990] use to generate surfaces for the modelling of skeletons, as we shall present in the next section.

[1] This term describes the locus of points that evaluate to the same value for a given property.

We can observe three major limitations to this modelling approach: global control of the model is compromised by the need to use a cut-off distance R, the generation of fields is restricted by the need to use points to define them, and there is a need to polygonise the generated surfaces before they can be visualised which results in loss of smoothness. Therefore, despite the authors' claims this approach to modelling has the characteristics of a static modelling approach and not an implicit one.

More specifically, the type of restrictions imposed for the determination of the function $C(r)$ imply that the modeller needs to know how effective each control point may be (i.e. the value of R). Such a condition aims at speeding up the calculations during the visualisation stage since the shape of the resulting soft object can only be affected locally by the fields of the neighbouring control points. This balance between global and local control, totally depends on the choice of the value of the cut-off distance R. If this is too large (i.e. comparable to the dimensions of the structure composed by all the control points) then every point on the surface of the resulting soft object will be affected by virtually all the control points, thus considerably reducing the efficiency of the visualisation algorithm. If the value of the cut-off distance R is too small, the continuity of the iso-surfaces will break, so the resulting soft object will become fragmented and will consist of several surface pieces.

Consequently, we must pay great importance to the choice of the value of R which must be large enough to ensure iso-surface continuity, but also small enough to make the visualisation stage efficient. We would recommend that every control point should be assigned a different cut-off distance value of R. This assignment could take place automatically, at a pre-processing stage, once all the control points have been determined. This pre-processing stage may also help in the determination of an interval of suitable iso-surface values that would guarantee a continuous surface of the resulting soft object.

Our research addresses the three limitations of this approach by abolishing the use of cut-off distance R, by allowing fields to be generated through the use of any geometrical object, and by avoiding polygonisation as a prerequisite for visualisation through the use of implicit modelling.

4.3 Skeletons

This section is concerned with the introduction of the *skeletons* as another tool for the modelling of computer graphics scenes. According to Bloomenthal and Wyvill [1990] and Burtnyk [1976] a skeleton consists of *points*, *splines* and *polygons*. The *points* are degenerate skeletons that serve as centres for simple quadrics such as spheres and ellipsoids, or superquadrics. The *splines* are sets of central axes for the purpose of modelling generalized cylinders with possibly varying radii or cross-sections. Moreover, the *polygons* are regarded as a mesh of flat facets and/or splines that are used to make an offset surface of the thus defined models.

The emphasis in this approach is in the interactiveness of such a modelling tool. The skeletons are shapes that the designer specifies using points, splines and polygons. In this way, we achieve an initial definition and manipulation of the skeletons. In addition to that specification the modeller has also to define a number of parameters that control how the skeletons will become a polygonized surface to feed the appropriate visualisation algorithms. In this aspect of skeleton modelling (i.e. parameter adjustment) implicit functions are determined for every skeletal part and both global and local control is exerted on the model.

The final aspect of skeleton modelling is that of *blending*. In this stage the skeletal elements are weighted together in order to determine how the resulting polygonized surface should behave when more than one skeletal elements are in proximity. A survey of blending techniques is reported by [Woodwark 1986] but further developments in computer aided design have extended the range of these techniques considerably.

To achieve a higher degree of interactiveness, Bloomenthal imposes cut-off points for all skeletal elements. In this way, the weight of any skeletal element that is further than R units of distance from any other is diminished to zero. A weight function for blending is also allowed to become negative thus permitting the subtraction as well as the addition of skeletal elements. A simple example used by Bloomenthal and Wyvill [1990] is defined below:

$$f(r) = 1 - (4/9)r^6 + (17/9)r^4 - (22/9)r^2, 0 \le r < R$$

where r is the distance of a three-dimensional point from a skeletal element.

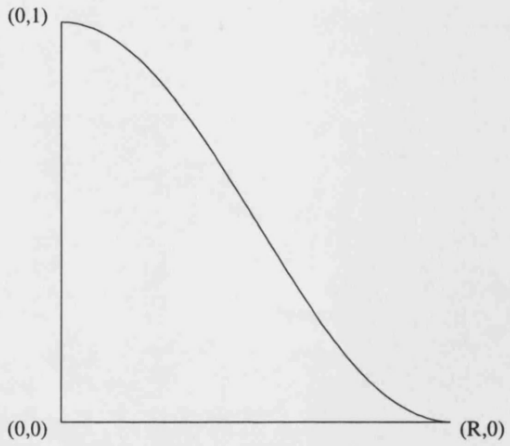


Figure 4.1 A blending function

This weighting function for blending skeletal elements is graphically shown in Figure 4.1. Its effect is demonstrated on Figure 4.2 where two skeletal splines (effectively two line segments that will become the axes of two finite cylinders of fixed circular cross-section) intersect and their resulting surface has been calculated using two-dimensional geometry.

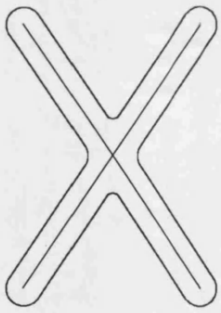


Figure 4.2 Blending contours

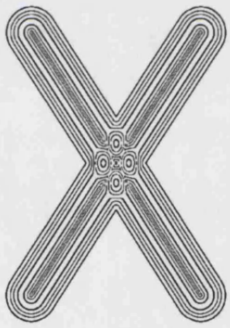


Figure 4.3 Anomalies of the contour map

This method extends the soft objects method (section 4.2) by allowing the generation of fields not only by points, but also by splines, and polygons which are the geometrical entities that the developers have chosen. In spite of these extensions, this method has also some limitations. The most important is that although the fields can be generated by several geometrical objects, the measure of distance from these objects is used in an arbitrary manner.

Moreover, our experiments depict that their proposed blending function does not generate a family of contours of 'similar shape' but, as we approach an intersection of two skeletal elements the contours become disconnected and they no longer outline this skeleton (Figure 4.3). Another limitation is that before visualisation it is essential that several approximations of the defining fields must be performed, and the result is a polygonised surface description, hence a static modelling approach.

The arbitrary manner in which the measure of distance is utilised in this approach is manifest in the conflicting treatments of this measure. In most cases, the distance of a point from a skeletal element is assumed to be the minimum Euclidean distance of the skeleton from that particular point. The use of this minimum distance definition from skeletal elements is considered by Bloomenthal as a *simple metrics arithmetic.* In other cases, however, distances are calculated from additional control points imposed by the designer; in these cases Bloomenthal considers the measure of distance to fall into the category of a *compound metrics arithmetic.*

Although Bloomenthal recognizes that most skeletal surfaces thus defined can be described by analytical functions, he prefers to treat them as procedural i.e. "defined by procedures that return a scalar value given a three-dimensional point" [Bloomenthal & Wyvill 1990]. The reason of his choice is to allow flexibility for the modelling process. Moreover, by using implicitly defined objects, a generic approach in visualisation would be more appropriate since is would enable visualisation of a much wider variety of possible shapes.

With respect to the visualisation of skeletal models Bloomenthal concludes in his survey that a trade-off needs to be made between geometrical accuracy which results into image quality, and speed of calculation which results into improved and speedier interactiveness. In his survey Bloomenthal [1990] considered several techniques which included: space subdivision of the skeletal elements (points, polygon, splines) using the octree display method, surface polygonization using simple but quicker (compared to the octree) linear interpolation, and more accurate but slower techniques such as successive binary subdivision approximation, or regula-falsi. He also looked into adaptive visualisation processes where a more detailed polygonal mesh was produced in the high curvature portions of the surface, or where the

surface was proved to be visible by the observer. The visualisation approaches that Bloomenthal has considered result in approximations rather than an accurate representation of the modelled surfaces.

In the implicit modelling approach that we propose, we elaborate on the measure of distance by applying set-theory. Such a study permits us to define an extended measure of the Euclidean distance, which in turn gives us a superior approach for defining families of surfaces. We also utilize the octree visualisation approach directly on the generated iso-surfaces rather than simply on the skeletal elements which provides a more accurate image of the modelled surfaces, as the following chapters demonstrate.

4.4 Implicit blending using displacement

This method is introduced as an "intuitive" approach to the implicit blending of surfaces [Rockwood 1989]. Fortunately the mathematical background and relevant theories for this method are also presented in the same paper. This method is not of immediate relevance to our research, however, we share several of the underlying principles.

One such common principle is the definition of implicit surfaces. The notion of the 'inside' and 'outside' of an implicitly defined object is well established. Furthermore, the use of constructive solid geometry and the Boolean operators as modelling tools are given formal mathematical definitions in the context of implicit models.

Another equally important issue that Rockwood discusses was the definition of the algebraic distance and the inefficiency of blending functions to provide continuity of the distance function. These discontinuities observed in such "pseudo-Euclidean blends" are counteracted with the displacement of the blending functions used. The roots of the blending functions are used to displace the blended surface in order to make the definition of the algebraic distance continuous over the complete space of the blend.

In the approach we propose the same principles are also utilised. For example, the concept of 'inside' is fundamental to most of the surfaces that are being generated with our proposed implicit modelling approach. Furthermore, the use of constructive solid geometry as a mechanism for building complex models out of simpler ones is also tightly related to our approach. Our method differs from this on the issue of blending and the need for polygonising the modelled surfaces before visualisation.

4.5 Distance fields in Medicine

This method is concerned with the visualisation of models of parts of the human body and especially models of the brain, that are initially described as meshes of triangles. To smooth out such defined models an alternative to smooth shading techniques was applied. This was based in the generation and visualisation of iso-surfaces that were produced by distancing away from the mesh a given distance [Payne & Toga 1992].

This distancing out procedure allowed for further manipulation of the surface produced which covered both global and local aspects of spatial control. The main processes that could take place in a such defined distance field surface were: *averaging* that entails interpolation between surfaces or surface patches, *offsets* which results in a global (or local) shifting of the complete surface (or surface patch), *blending* that amounts to connecting together surface patches in a new surface of arbitrarily chosen smoothness, and *blurring* which is intended for the reduction of surface details while keeping the overall shape of the surface almost unchanged.

Specifically for blurring, out of all four functionalities (e.g. averaging, offsetting, blending and blurring), the benefits of such a modelling approach are twofold. First is of course the minimisation of computer storage requirements. As a consequence, such a model would demand less computing time during visualisation. This would provide medical professionals with a real-time workbench for viewing such surfaces. The second stream of benefits comes from the actual minimisation of surface details which in many cases obstruct the observer's attention and depending on the actual viewpoint may cast shadows on more important features of the observed surface.

The primitive geometrical objects of this approach are, as we have already mentioned, triangles. These have been collected and put together by an automated data acquisition system. The offset zero surface is therefore the mesh itself. Nevertheless, any other manipulation (i.e. non zero offsets, blending, etc.) will demand the calculation of the distance between the mesh and any arbitrarily chosen point in three-dimensional space. The problem of finding the distance of a point from a mesh of triangles is split into calculating the minimum distance of that point from all the meshes' constituent triangles. Consequently the problem is shifted into determining the distance of a point from a triangle in three-dimensional space. This three-dimensional problem is then simplified into a two-dimensional one by transforming the triangle and the point so that the triangle is parallel to the X - Y plane of the scene's coordinate system at the $Z = 0$ level.

In this way, the accordingly transformed coordinates of the point can be used to determine its distance from the triangle. Seven cases have been identified depending on the orientation of the point's projection onto the triangle's plane. This analysis is claimed to provide an efficient method for determining the distance of any point from the mesh, however, a number of other acceleration techniques have also been proposed. These acceleration techniques include the use of spatial coherence information that can be inferred with cubes that surround portions of the model's surface. Furthermore, it was suggested that the necessary transformation matrices for moving every triangle of the mesh onto the X - Y plane should be computed once, at the beginning of the visualisation, and stored for further use. Additionally, evaluation of computationally expensive functions such as square roots had to take place only when they were absolutely necessary.

This method, although it could demonstrate its potential in the application domain of medicine, is not appropriate as a general tool because it has been fine-tuned to process triangles only. Nevertheless, it has a place in the broad field of computer graphics since nowadays there are many three-dimensional scanners in use that produce models in the form of meshes of triangles (e.g. Cyberware, post-processed CT scans, etc.).

The off-setting of polygonal meshes in order to produce a smoother surface is also used in our approach where meshes of any convex planar polygons may be used rather than triangles. Furthermore, our approach is capable of producing the same effect on any other geometrical entity that we wish to model. Moreover, the polygonal mesh that approximates to the modelled surface is not necessary for its visualisation, unlike the method of distance fields presented here.

4.6 Colour superposition

Another lead to our research is provided in the work of Firby and Stone [1987] who describe and explore the effects of superposition of families of curves. In this section we present the method of superposition, we discuss its limitations and the way we have overcome them in our research.

Firby and Stone examine the creation of interference patterns in optics and more specifically in the areas of textile manufacturing, paper-printing of patterns and computer graphics. In optics, the effects of interference are colourful patterns created when light passes through an assembly of optical lenses. As the index of refraction of the lenses is slightly different at different wavelengths of visible light, at the perimeter of such an optical assembly analysis of light occurs (rainbow colours). In an assembly, each lens will produce its own colour patterns. Furthermore, patterns produced on the first lens will also pass through the next thus eventually producing a superposition of interference patterns.

In the textile industry, the interference patterns are produced by the optical illusion which is created when several patterns of (usually multicoloured) yarn are interweaved (i.e. superpositioned) in order to construct the fabric. In a similar way, in computer graphics the interference patterns become apparent as Moiré patterns due to the spatial and colour approximations imposed by the orderly arrangement of the viewport's pixels. A simulator of the interference colour using ray tracing and the Fresnel's generalized formulae for its shading model has been recently proposed by M. Dias [1994].

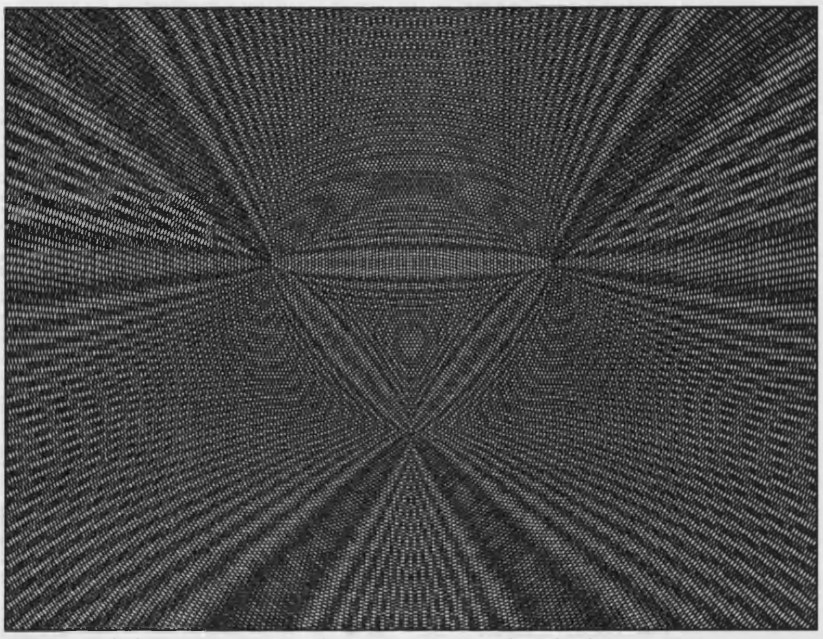


Figure 4.4 Interference due to concentric circles

In order to study these interference patterns, Firby and Stone used contour maps that were imposed on top of each other. The primary contours they used were concentric circles which produce the effect shown in Figure 4.4. In some of their experiments they have also used radial lines (Figure 4.5). In order to demonstrate the effect of superposition, Firby and Stone colour-coded the contours: for each set of contours, a colour was assigned and on a few occasions its intensity was diminishing on a linear scale as individual contours were progressing away from the centre point of the set of the concentric circles.

The results of these experiments were plotted directly on colour film. In this way, instead of processing mathematically the effects of superposition (addition), Firby and Stone exploited the properties of the photographic film. Photographic film demonstrates distinct properties that differentiate it from other display media. They are described by their effects and the additive nature of colour. In particular, if an area of the film has already been plotted, it cannot be erased, or replaced by any succeeding plot over the same area. Any such plot will result in the addition of colours. Take, for example, the RGB colour model. Using the notation of the first chapter, the colours red (1 , 0 , 0) and blue (0 , 0 , 1) when added together will produce purple (1 , 0 , 1). However, the addition of blue (0 , 0 , 1) with blue (0 , 0 , 1) will 'burn' (i.e. over-expose) the film and produce saturated blue of a degree proportional to the exposure time and the speed of the film.

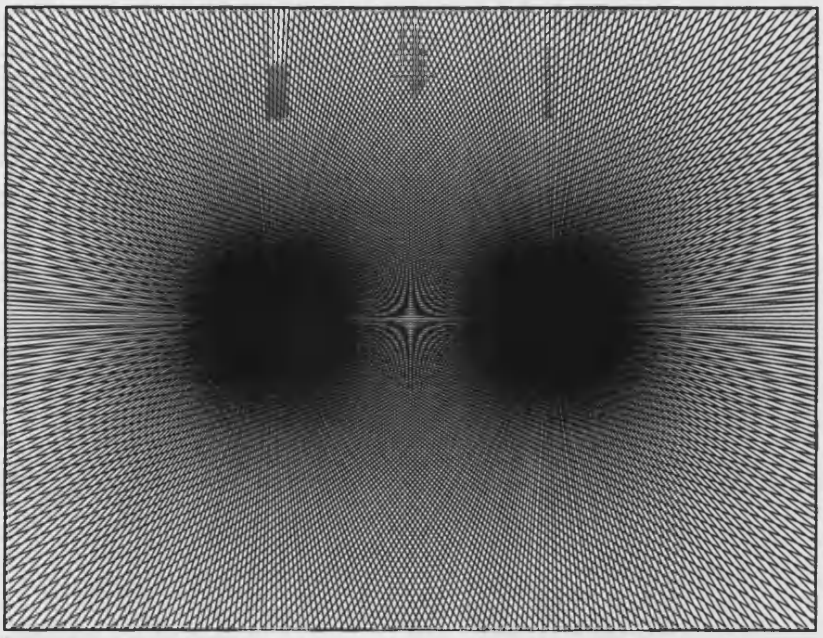


Figure 4.5 Interference due to overlapping radial lines

These properties of colour, as it is plotted on a photographic film, may be used to demonstrate not only the effects but also the contributors (individual map centres) of the confluencing contour map thus generated. It is therefore important to select appropriate colours in order to illustrate confluence of colour-coded maps without burning the film as we illustrate in Figure 4.5.

The method of superposition and its effects on colour-coded maps is based on the same theory as the modelling approach that we propose. The main difference between superposition and our implicit modelling approach is that while the superposition method views the effects of confluence (fields defined with the measure of distance) as patterns of curves, our implicit modelling approach treats the results of confluence as surfaces (iso-surfaces). Consequently, superposition restricts the visualisation of the effects of confluence in spaces of two dimensions because of the utilisation of the photographic film, whereas our implicit modelling approach permits a more intuitive representation of the results of confluence to spaces of higher dimensions as well.

The limitations of the superposition method stem from the way effects of confluence are illustrated: using differently coloured patterns imposes a number of important restrictions. First, it is based solely on two-dimensional geometry, second it relies not on analytical

descriptions but on optical illusions, and thirdly its success depends on the choice of the appropriate colour-coding schema and the particulars of the photographic film used.

The implicit modelling approach that we propose clearly provides an improved way for examining confluence since the results are not presented as patterns with a two–dimensional geometry but as surfaces which are being illuminated and shaded in order to provide a comprehensive base for studying them.

To exhibit the similarities and differences between superposition and the implicit modelling approach that we propose, let us consider the following example of interference. Let us assume that there are two points in two-dimensional space that constitute the centres of their respective contour maps (***A***, ***B***) of concentric circles. Both maps, have been assigned shades of the same primary colour, blue (0 , 0 , 1), which at their centres has minimal intensity (0 , 0 , 0) and the colour's intensity is increased as we progress further from the centres up to a maximum value of pure blue (0 , 0 , 1). This colour change is achieved by using a linear function of the distance d from the map's centre, say $Col(d) = (0 , 0 , 0.0001 \times d)$. The contours of confluence of the two contributing maps will then emerge in this example as patterns coloured with the same shade of blue. These patterns of the same shade, the *iso-shade* patterns, form ellipses that have their two foci at the centres of the two contributing contour maps. The results of this superposition (Figure 4.6) are similar to those of Figure 4.7 that were produced with our implicit modelling approach (also depicted on plates 6 - 9).

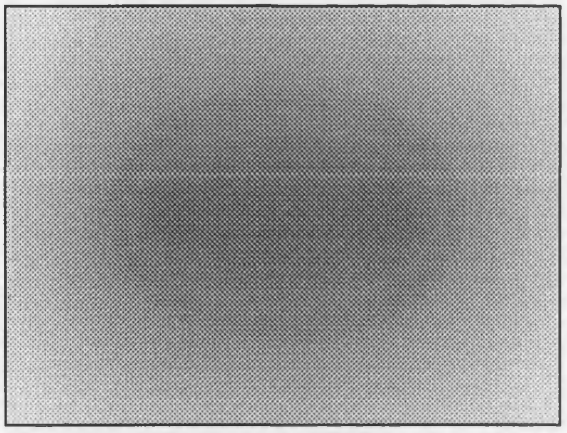


Figure 4.6 The field of A+B

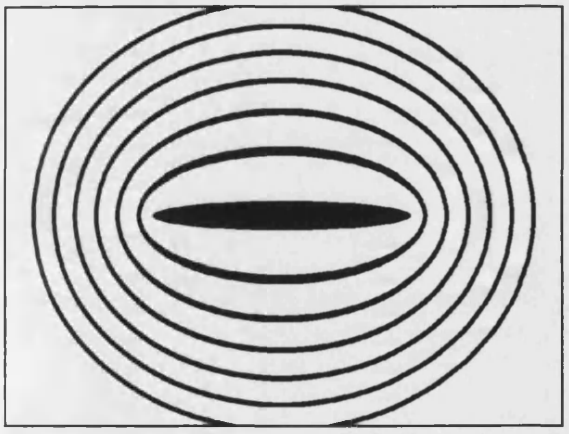


Figure 4.7 Contour map of A+B

To prove the argument, let us name these circular contour maps as ***A*** and ***B*** and let us also assume that the distances of the map centres from a randomly chosen fixed point ***p***, are d_A, and d_B accordingly. From the map ***A***, this point should be coloured as $Col(d_A)$. Similarly from the map ***B*** the contributing colour would be $Col(d_B)$. The addition of the maps ***A*** + ***B***, namely the confluencing map, will then evaluate at this point ***p*** as $Col(d_A) + Col(d_B)$ as Figure 4.8 illustrates. This result, for a particular type of functions *Col()*, is also equal to $Col(d_A+d_B)$ which is the application of the function *Col()* using one contour map with two sets of concentric circles (coincidental to the maps ***A*** and ***B*** respectively).

Any function that satisfies the relationship: $Col(a+b) = Col(a) + Col(b)$ may be used. One such function for example, is the $Col(d) = (0 , 0 , 0.0001 \times d)$ that assigns a more intense scale of blue as we progress further away from the centre of the map. Colour values that will result in a blue colour component greater than (0 , 0 , 1) will be truncated to (0 , 0 , 1), in order to avoid overexposing the film. Therefore, in the above example, the locus of points that have the same iso-colour value ***c*** will be characterised by points that the sum of the distance from the centres of the two maps (***A*** and ***B***) is fixed and validates the equation:[2] $Col(d_A+d_B) = c$. Such a locus of points also defines an ellipse with foci at the centres of the confluencing maps ***A*** and ***B***.

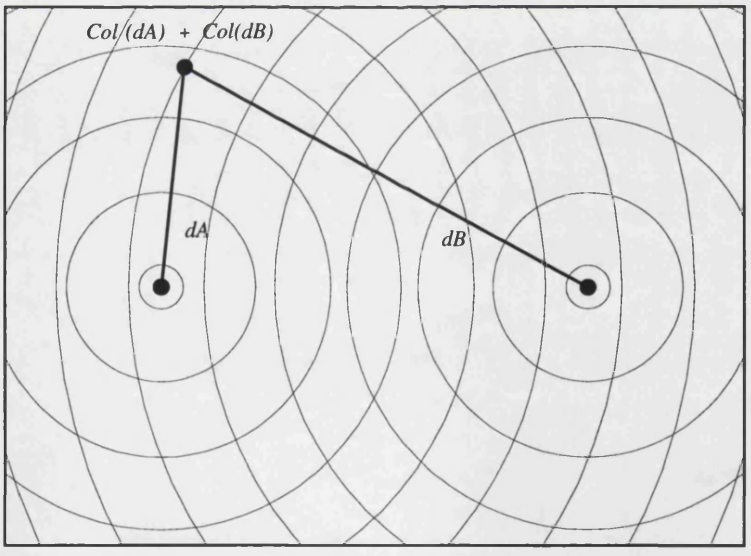


Figure 4.8 Determining the map addition

[2] To be more specific, our claim is true for all points ***p*** in the two-dimensional space that their distance from the map centres satisfy the inequality: $d_A + d_B < 10000$.

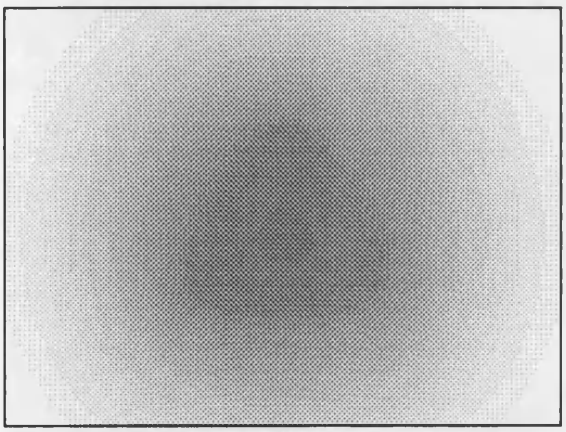


Figure 4.9 Field of three points

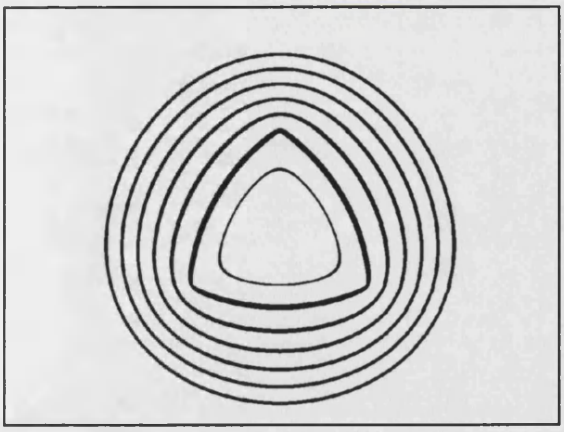


Figure 4.10 Contour map of 3 points

Using three confluencing maps and the same linear function for assigning colours, *Col(d)*, we get Figure 4.9 and Figure 4.10. The illusion of the same shape can also be obtained from Figure 4.4. However, if we extend this method in the three-dimensional space, the inadequacies of the superposition method are obvious, but can be overcome with the implicit modelling approach that we propose, as plate 11 illustrates.

4.7 Sphere plots

Sphere plots represent a modelling approach that creates surfaces which are defined on the surface of a sphere. There are two difficulties with this approach: the choice of the initial approximation to the spherical surface upon which the modelled surface will be built, and the scaling of the modelled surface so that it will not cause degeneracies.

This method is the result of a NATO and US Energy department grant,[3] aiming at analysing the effects of the global warming phenomenon [Foley *et al.* 1990]. As such, the principal object this method was the earth which was modelled as a sphere. One of the main tasks of this project was to depict, in a colour-coded schema as well as geometrically, functions that had been defined over a sphere. Ozone density, barometric pressure, temperature, and other atmospheric parameters provided the functions to be plotted. Such bivariate functions were defined along the longitude and latitude of the earth and were either described analytically, or needed to be interpolated out of a small number of observation points.

[3] NATO RG 0097/88, DE-FG02-87ER45041

Their plots had to be mapped around the surface of a sphere. In certain circumstances the points to plot had to be extruded from the sphere's surface. In general, the sphere's surface was replaced by a mesh of *structural points* and the sphere plots were based on the elevation adjustments of certain structural points from the sphere's centre. These structural points were the vertices of the triangles that had been provided by the triangulation mesh of a unit radius sphere. In order to depict the value of the sampled function, the elevation adjustments had to be proprotional to the distance of the structural points from the sphere's centre.

The problems they faced stemmed from both the method of triangulation and the degree of height adjustment. For the first problem, they abolished the longitude-latitude rectangular grid approach and experimented with the subdivision of regular canonical solids such as the tetrahedron and the icosahedron. In this way an adaptive solid subdivision method was developed in order to provide enough accuracy for mapping their data but also getting a smooth shaded image.

With regard to their second problem, a unit radius sphere although convenient for triangulation, placed a limit on the values they could plot. It is apparent that height adjustment of values in the range of (1,0] would 'squeeze' the sphere, within (0,-1] would expose its centre, and in the range [-1,-∞) would pierce the sphere and produce nonsensical results. The obvious remedy, of scaling down and shifting their measurements to fall within the range of [1,2] was adopted.

Although, as we will see in the following chapters, in our method there is also the need to scale appropriately such density measurements, we extend this modelling approach by providing a method for plotting any such scaled function on any arbitrarily defined shape, and not just the unit sphere. Moreover, we will show how the generated images produced with the approach we propose, are smooth without the need for choosing the appropriate triangulation density. This is because we do not use any such mesh, but instead we directly map the density functions onto the pre-defined surfaces.

4.8 Convolution

Following research on simulations of electric fields by Blinn [1982], the soft objects of Wyvill [*et al.* 1986] and the rounding of corners in solid models by Colburn [1990], convolution comes as another technique for producing and visualising implicitly defined surfaces [Bloomenthal & Shoemake 1991]. This technique is very similar to Blinn's and Wyvill's. The only difference is the mathematical perspective under which it is discussed and studied. Blinn and Wyvill described surface models with their geometrical properties; iso-surfaces that had to be approximated by means of a polygonal mesh. With the technique presented here, Bloomenthal and Shoemake describe their surfaces as convolutions of simple functions (of distance) along skeletons of points.

The distinctive feature of this technique is the transfer of a classical digital signal processing tool, namely *convolution*, in the modelling process of computer graphics. Initially convolution was only used in computer graphics as a signal processing tool for aliasing problems during rendering [Blinn *et al.* 1976; Feibush *et al.* 1980; Greene *et al.* 1986; Foley *et al.* 1990; Wolberg 1994]. In the context of modelling, this tool is introduced in the context of skeletons [Wyvill *et al.* 1986] and the surfaces around them that are generated as contours of fields produced with using the measure of distance.

The skeleton is assumed to be a set of discrete points that generate a surface around themselves. This is a uniform spherical surface at a given distance from each skeleton member point. If the skeleton consists of more than one point then a decision has to be made regarding the handling of the resulting surface pieces. We can either assume these surface pieces as a *union* of individual patches, in which case we will use the *maximum* as the operator for combining them together in a set-theoretic definition, or we can assume they are a smoothly connected *surface blend*, where the constituent pieces have been *add*ed together with the *addition* operator.

The first choice produces quick results that may not be analytically continuous over the resulting surface. This is the result of the set–theoretic union of the constituent surface patches which is implied by the use of the function of maximum. The second choice, which

is also the one that Bloomenthal preferred, demonstrates how individually produced surface patches are blended together to construct a smooth surface. For this method the choice of the appropriate blend function is important since it will affect the smoothness of the generated surface but also the complexity of the calculations used for its determination and therefore would affect the efficiency of the visualisation algorithms.

Bloomenthal and Shoemake [1991] define a skeleton $S(p)$ as a function that evaluates to one for all points p that constitute the skeleton and evaluates to zero elsewhere. Around each point s of the skeleton, a surface patch (described below) is assumed to be constructed:

$$f(p) = \exp\left(\frac{-\|s-p\|^2}{2}\right)$$

A surface is defined as the union of these surface patches 'around' each skeletal point. Depending on the nature of the skeleton we can distinguish two cases of surface union, the *discrete union* where the skeleton consists of a finite set of points, and the *continuous union* where there is an infinity of points that constitute the skeleton.

The discrete case of surface union would then be denoted by the sum of all surface patches.

$$f(S,p) = \sum_{s \in S} \exp\left(\frac{-\|s-p\|^2}{2}\right) \qquad \text{(Eq. 4.1)}$$

The case of a continuous skeleton piece such as a line segment or a polygon is treated as an infinite sum of patches which is achieved by integration

$$f(S,p) = \int_S \exp\left(\frac{-\|s-p\|^2}{2}\right) ds \qquad \text{(Eq. 4.2)}$$

Using this notation, Bloomenthal and Shoemake [1991] view the exponential function in (Eq. 4.1, Eq. 4.2) as the generator of a surface which is the "convolution of a spatially extended skeleton". This view is based on the observation that equations (Eq. 4.1) and (Eq. 4.2) can be perceived as the convolution (denoted by the symbol $\star$) of a skeleton $S(p)$ and the Gaussian function $h(p)$ (Eq. 4.3). In other words, $f = h \star S$ (Eq. 4.4).

$$h(p) = \exp(\frac{-\|p\|^2}{2}) \qquad \text{(Eq. 4.3)}$$

$$f(p) = (h \star S)(p) = \int_S \exp(\frac{-\|s-p\|^2}{2}) ds \qquad \text{(Eq. 4.4)}$$

To make this observation useful for the development of an algorithm out of this modelling approach, a number of approximations as well as complementary assumptions had to be made [Bloomenthal & Shoemake 1991]. The most significant approximation is the replacement of the *Gaussian h* function with a cubic spline. An additional assumption is the imposition of limits to the extent of the skeleton's contributions. With regard to the assumptions made, only simple sets of skeletons can be computed efficiently. For the more complex ones, Gaussian filters can also used but in this case the properties of these filters need to be analyzed further.

Bloomenthal identifies two such properties, the *superposition*, and the *component separation* which stem from the study of fast Fourier transformations. The first is best described by the equation $h \star (S_1 + S_2) = (h \star S_1) + (h \star S_2)$ which effectively allows the construction of complex structures out of simpler ones. The second allows the separation of the *h* function into coordinate components. This means that one can separate a three-dimensional convolution into a two–dimensional one and then multiply that with the third dimension component. Furthermore, the two-dimensional convolution can be further decomposed into two one-dimensional components. This process of decomposition reduces complexity thus making this modelling approach useful for a variety of computer graphics applications.

A variation, or rather extension, to this technique is the use of weight functions that are attached to every skeleton member. The weight function will add a considerable degree of flexibility to the modelling of such convoluted surfaces since it would allow local control of the individual surface patches before they are blended together. Another extension to the convolution modelling approach is the application of deformations which we should note, produce different results if they are applied to the skeleton than if they are applied to the final surface blend.

The benefits of this approach have not been explored fully, due to the mathematical complexity of the calculations involved and the lack of a consistent approach for the appropriate visualisation of the generated surfaces. Moreover, a thorough study of the effects that *h* functions produce when convoluted with skeleton definitions, has not been carried out yet. Such analysis would allow modellers to select and use convolutions depending on their properties. In our understanding, the approach of convolution differs from ours in the way we manipulate and subsequently visualise the modelled surfaces. The analytical nature of the convolution function *h* may become too complex for integrals to be evaluated and therefore, approximation techniques may be required for the visualisation of convolution-generated surfaces. In our approach, however, surfaces are modelled as sets of points, thence there is minimal use of analytical tools and surface approximations are not necessary.

4.9 Delaunay triangulations and Voronoi tessellations

This section is concerned with the determination of a Voronoi tessellation [Voronoi 1908; 1909]. Delaunay triangulations [Delaunay 1933] are also presented here for reasons of completeness since they represent the dual face of Voronoi tessellations. Our approach addresses the problem of Voronoi tessellations and as we show in chapter five it is capable of solving a more generalised form.

The problem of Voronoi tessellations has appeared with different names such as Dirichlet triangulations [1850], Thiessen's problem [1911] and has been studied by a number of researchers from a variety of application fields such as mathematics [Angell & Moore 1986; Green & Sibson 1978; Bowyer 1981], geophysics [Watson 1981], and aerodynamics [Jameson *et al.* 1986; Vassberg and Dailey 1990; Baker 1989].

A Voronoi tessellation starts by considering a set of points in the n–dimensional space which are usually named *nuclei.* Using Watson's [1981] 'biological' description, the n–dimensional case of Voronoi tessellation partitions the (n–dimensional) space into convex polytopes that may be thought of as expanding hyperspheres centred at the nuclei. Their surface expansion will cease when they meet with each other, thus producing the desired set

of convex polytopes. By assuming a common rate of hypersphere expansion, we can ensure that the meeting points between two hyperspheres (i.e. the faces of the polytopes) are equidistant from their respective centres (i.e. nuclei). In this way, the hyper-volume of space which is surrounded by any convex polytope is guaranteed to be closer to the only nucleus that lies inside this polytope, than to any other nucleus in space.

The above definition of a Voronoi tessellation is based on the fact that all hyperspheres grow with a common rate. This restriction ensures that the meeting points between hyperspheres (i.e. the polytopes' faces) are equidistant between their respective centres. An interesting extension to this tessellation stems by disregarding the above restriction; each hypersphere is allowed to have its own growth rate. In this way, growing hyperspheres will meet in points where the distance from their respective nuclei is proportional to their growth rate. Consequently, the faces of the generated polytopes are no longer portions of hyperplanes only, but portions of hyperspheres. The tessellations thus generated may not even be connected since areas of influence by a particular nucleus may be separated by areas of influence of other (more influential) neighbouring nuclei. These observations result from the application of the Appolonius theorem in the n–dimensional space [Angell & Moore 1986] and the corresponding tessellation is named *weighted Voronoi tessellation.*

Determining the tessellation out of a set of nuclei is not a trivial task. Angell and Moore [1986] suggest the use of quadtrees in producing two–dimensional cross-sections of tessellations. They first determine such a cross-section plane. Then, they define a unit sized square window upon which the real coordinates of the hyperspace will be mapped. Then the quadtree algorithm examines whether this appropriately sized square intersects with any points of the tessellation's polytopes. Usually, the initial cross-section window is positioned in a way to ensure that there exist such an initial intersection. Once polytope's intersection is suspected, the square window is subdivided into four equally sized square subwindows. For each of these subwindows the same *tessellation interrogation* process is applied. For subwindows with no common points with the tessellation, the subdivision process can be safely interrupted. However, for the intersecting subwindows, the subdivision process does not continue endlessly, but is interrupted once the size of a subwindow may be accurately represented by one pixel on the attached viewport.

This quadtree algorithm can be used for unweighted and weighted Voronoi tessellations. The speed of the algorithm depends on the complexity of the tessellation interrogation process, and the resolution of the viewport which dictates the total depth of the recursive subdivisions. If a subwindow is found to belong inside the volume of a tessellation polytope, there is no need for further subdivision. However, if a subwindow is found to intersect partially with the tessellation, then although further subdivision is required it is not necessarily implied that there exist tessellation points, since there may be other nuclei in the neighbourhood. Such cases are catered for by the recursive nature of the quadtree approach.

Other researchers such as Watson [1981], approach the subject of unweighted Voronoi tessellations from its geometrically dual angle. This is the case of the Delaunay triangulation where the aim is to determine a set of space filling polytopes that have their vertices on a given aggregate of nuclei (i.e. points in n-dimensional space). The requirement for the polytopes thus defined is that the circumscribing hypersphere for any polytope does not contain (i.e. intersect with) any other nucleus. Mathematically this problem is Voronoi's dual since the centres of the polytopes' circumscribing hyperspheres may become the vertices of the unweighted Voronoi tessellation for the same nuclei. With this observation, the required Delaunay polytopes can be constructed from the unweighted Voronoi tessellation since any point of the nuclei set cannot lie inside any such determined circumscribing hypersphere because this would contradict with the definition of the Voronoi tessellation.

These techniques are representative solutions to the Voronoi tessellation and Delaunay triangulation problems. They are all optimized to solve particular cases of the tessellation and triangulation problems. For example, most of them provide a solution to the unweighted problem(s) in the space of two dimensions. Furthermore, they all share a common assumption about the nature of the nuclei; the nuclei are assumed to be points.

In the implicit modelling approach that we propose in the following chapters, we demonstrate a more powerful technique for determining the Voronoi tessellation. Its power is illustrated by the expansion of the definition of the Voronoi tessellation in order to make it applicable to nuclei that are not necessarily points but also line segments, planar polygons or even three-dimensional geometrical objects such as spheres or convex polygonal meshes.

4.10 Ray representations

Ray representations, or *ray-reps*, is a new approach to modelling [Menon *et al.* 1994]. The principal idea behind this method is the visualisation process of *ray casting*. According to this method, an object can be observed by all the eye rays (emanating from an observer) that intersect with it. Following this principle, ray representations is an attempt to model geometrical objects with sets of lines, called *rays*.

The modelling process starts with a set of parallel rays that are equally spaced in the three-dimensional space in order to form a *ray grid*. This ray grid is assumed to cover all the volume of three-dimensional space that surrounds the scene. This assumption will ensure that the 'front' as well as the 'back' of the scene would intersect with the ray grid irrespective of the position of the observer. The resulting model will consist of all the ray segments of the ray grid that intersect with the geometrical objects of the scene. Consequently, in order to ensure that all the objects in the scene will intersect with at least one ray of the ray grid, the spacing between the individual rays in the ray grid is critical. Furthermore, another issue that needs appropriate consideration is the choice of the direction of the rays. A bad choice would result into rays tangential to some objects, thus resulting in null or single point intersections.

The ray representations method caters for geometrical objects that can be expressed with quadrics or similar analytical functions. In this way, the process of finding the ray segments that intersect with the objects is a straightforward task. Nevertheless, once the ray representations of the scene's objects have been computed a number of transformations may take place. The most simple transformations are those of space coordinates which Menon [1994] called "rigid motions". Sweeps along arbitrary trajectories can then be expressed by step-wise rigid motions that follow the trajectory given.

The importance of this modelling approach and its relevance to our research is that models are treated as sets of line segments. Therefore, a number of set-theoretic operations can also be applied. By allowing the intersection ($\cap$), union ($\cup$) and complement ($-$) of sets of line segments the functionality of constructive solid geometry can also be utilized. With this

rationale, the step-wise rigid motions approach to sweeps can be seen as a set-theoretic union (U) of all the resulting steps of rigid motions.

This modelling approach also lends itself for the use of the boundary representations modelling method. Once a geometrical object is described by its boundaries, the intersection of the boundaries with an appropriately defined ray grid will produce the ray representation model of the geometrical object. Although this translation seems a straightforward one, the way object boundaries have been described would impose difficulties during the ray - boundary intersections. For example, for boundaries that are represented by planar polygons or non uniform B-spline patches (NURBS) of a small degree (usually less than four) the ray -boundary intersection problem can be computed quickly and accurately. However, for boundaries described by higher degree spline patches or inferred by other processes, the intersection problem is too complex to compute (i.e. no exact analytic answer is available).

Apart from these classical set-theoretic functions, the morphology operators of Minkowski may also be used. These operators, the *M-addition*, denoted by $\oplus$, and the *M-difference* which is denoted by $\ominus$ have been interpreted intuitively [Menon *et al.* 1994] in order to be used with the ray representations and produce 'reasonable' images. The M-addition of sets $A \oplus B$ is defined as the union of all the translations of set B by all members of set A.

$$A \oplus B = \{a+b \mid a \in A, b \in B\} = \bigcup_{a \in A} B+a = \bigcup_{b \in B} A+b$$

where $a+b$ is defined as the vector addition (translation) between vectors a and b. It should be noted here that vectors may also be seen as point coordinates, thus their addition would result in a new point. In a similar manner, the M-difference is defined as

$$A \ominus B = \bigcap_{b \in B} A+b = -(-A \oplus B)$$

where $-A$ denotes the complement of the set A and $A + b$ denotes the translation of the set A by the vector b, i.e. $A + b = \bigcup_{a \in A} \{a+b\}$

Menon interprets the Minkowski operators by applying them only to the end-points of the sets of line segments and not to all points that the line segments consist of. In this way the calculations are accelerated. However, the M-difference is not free from problems. There are

cases where the result of M-difference between two sets of ray segments will produce ray segments with coincidental end-points. This stems from the fact that this operator is not closed within the sets of line segments. This observation has led Menon to introduce some modifications to the M-difference operator which he has named *regularised M-difference* and is denoted by $\ominus_{reg}$. The improved regularised operator is applied as a post-processing stage to 'clean up' the set of line segments from all the degeneracies that may have occurred after the application of the original 'non-regularised' M-difference operation.

In Figure 4.11 we illustrate how the operator M-addition works. There are two polygons; a square, denoted by $\boldsymbol{A}$ and a triangle denoted by $\boldsymbol{B}$. On the right side of the figure we see the M-addition $\boldsymbol{A} \oplus \boldsymbol{B}$ which is denoted by the dashed-line polygon. For the purpose of understanding, we have also included in this figure the intermediate stage of the M-addition operation where the triangle $\boldsymbol{B}$ has been added ($\oplus$) to every vertex of the square $\boldsymbol{A}$.

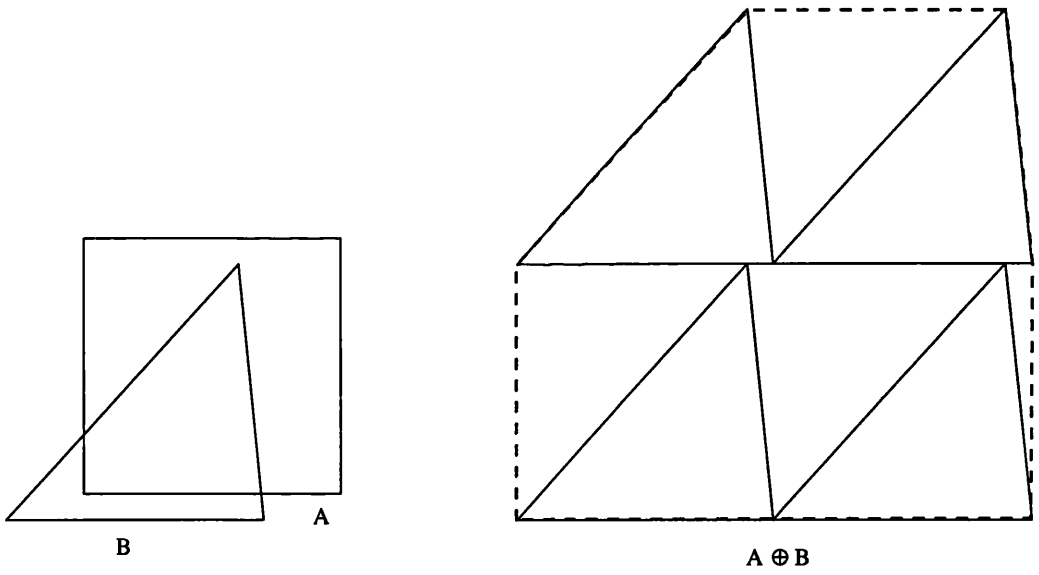


Figure 4.11 The M-addition of a square with a triangle

This modelling approach can be seen as a two stage process. The first stage, that of determining ray representations out of geometrical object descriptions, involves heavy and complex calculations. The next stage is concerned with the manipulation of the sets of line

segments using set-theoretical operators. Although this is beneficial to modelling since the application of transformations permits the building of complex geometrical representations out of very simple ones, it also puts enormous demands onto the hardware.

This approach is characterised by enormous computational costs. However, the simplicity of using line segments as the only means of describing a model will be appreciated in the visualisation stage where parallelism and spacial coherence may be exploited. To make this approach usable there has also been built a piece of hardware that implements all the set-theoretic operations algorithms, as well as the constructive solid geometry manipulations in firmware. This system is named the Ray Casting Engine [Ellis *et al.* 1991] and extensively exploits parallelism.

In contrast to implicit modelling approaches ray-reps do not assume a set of constraints to be the surface's defining test (chapter two). Therefore, this method seemingly belongs to the family of static modelling approaches as it assumes that the objects to be modelled have an analytical description. However, a conversion to an implicit definition, that describes all objects as sets of line segments (the ray-reps), is applied before visualisation, making ray-reps an implicit modelling approach.

Similarities of this method with the modelling approach that we propose stem from the treatment of surfaces as sets of points. In this way, we share the application of set-theoretic operators. Moreover, the building of surfaces of revolution and those of general sweeps along a given trajectory are also treated in an analogous way since these operations act upon the sets of points that describe the objects and are not determined analytically.

The difference between ray-reps and our approach is that ray-reps are line segment representations of objects that are visualised in a specialised computer system (the Ray Casting Engine). Our approach applies set-theoretic principles not only to line segments but to sets of points thus allowing the manipulation of a greater variety of objects.

4.11 Discussion: the need for further research in modelling

In the previous sections we have reviewed a number of methods that use, either implicitly or explicitly, the measure of distance as the means for generating a surface. Distance is a key concept in the geometry of metric spaces; it is used for descriptions and measurements in these spaces. The study of objects in these spaces has been carried out through the use of their analytical descriptions. It must be noted, however, that not all objects have an analytical description. Computer graphics also make use of analytical (i.e. explicit) models to visualise objects. The use of analytical techniques in graphics has currently reached its full potential and has began to expose the limitations of these techniques. Computer graphics offer the potential to manipulate and visualise implicitly defined objects also, thus creating opportunities to study objects without the prerequisite of their analytical description. Furthermore, implicit object definitions rely heavily on the measure of distance.

4.11.1 Criticisms of current research

The techniques reviewed in this chapter represent efforts to generate implicitly defined objects. In all the above cases, however, implicitly defined models are then approximated through analytical functions for visualisation purposes. Therefore, the 'implicitness' of the approach is compromised in every case. In the chapters that follow, we propose an approach which is free from any use of explicit (analytical) techniques in both modelling and visualisation.

The utilisation of the measure of distance varies greatly in the methods we have reviewed in this chapter. For example, in colour superposition, the measure of distance is implicit as the perceived distance between the confluencing maps, whereas in soft objects and skeletons, the measure of distance explicitly determines the shape of the modelled surfaces. Furthermore, in convolutions, where there exist a mathematically neat way to extend skeletons spatially, the measure of distance and the manner with which it is included in mathematical expressions make it the vital ingredient for the description, generation and visualisation of surfaces.

The techniques presented so far contain concepts that will also be used in defining our own implicit modelling approach. Our approach combines concepts such as the use of the measure of distance for the generation of fields of 'potential', the description of objects as iso-surfaces, the visualisation of iso-surfaces as they are defined on a given 'potential' field, the treatment of objects as sets of points (the locus of which describe the surface) and the manipulation of objects using set-theory. We will conclude this chapter by briefly introducing our use of the these concepts which aims to overcome the limitations of the modelling techniques discussed so far.

4.11.2 Requirements for an implicit modelling approach

In the approach that we propose, the measure of distance is used to create a surface by assuming that a field is propagated among the points participating in that surface. The potential of every point in the field will be calculated as a function of the distance of this point from a given set of objects. Such sets of objects were the kernels for the soft object method, or the skeletons for the skeleton-based method. The locus of points that exhibit the same potential in this field, called in our method *iso-surfaces*, defines the surfaces that we develop and visualise.

The assumption that a point may generate a field in its surrounding space (a 'potential' field), is greatly enhanced in the fifth chapter. In particular, we provide there an analysis of fields that originate from a variety of simple geometrical objects such as lines, planar polygons, spheres, cylinders etc. In this way, we are able to construct several families of new surfaces that currently are too complex or, impossible to describe otherwise. Therefore, our modelling approach re-uses old (pre- Pythagorean) surface construction methods which have been abandoned because of the complexity involved in their implementation. We demonstrate extentions to one such constrution method in plates 25 - 31.

Moreover, by perceiving surfaces as the 'locus of points with a certain property' we allow several manipulations to be performed on these surfaces. These surface manipulations can be achieved by applying set-theory (on the sets of points describing the surfaces). For example, the generation of a body of revolution or, the envelope of a surface along a given

trajectory, can be implemented as the union of sets. This approach offers significant benefits by providing a generic way to manipulate surfaces and a means for describing them without the need of analytical methods.

With regard to visualisation, we require that the surfaces generated with our method convey as much information as possible to the viewport. Therefore, approximations to the functions that define the surface and approximations to the surface with a polygonal mesh are not desirable except for the cases where the error they introduce is insignificant when compared to the error that the mapping of the ABSOLUTE space to the VIEWPORT space imposes. The visualisation approach we use in this research (chapter six) strives for maximum detail of the created images which will allow a thorough study of the models we have developed. The following chapters provide details of the modelling approach that we have developed to meet the requirements outlined in this section. We illustrate its potential through a series of examples of especially interesting cases.

Chapter 5 Distance as a tool for surface definition

5.1 Introduction

As we have seen in chapter two, one of the major modelling approaches is that of implicit modelling. According to this approach, a surface is implicitly defined as the locus of points in space that satisfy a point membership classification test. In this way, the description of a surface is not analytical, but it has the form of a set of constraints which collectively we have called the *point membership classification test.* Usually, these constraints are mathematical relationships involving point coordinates. For example, using the measure of Euclidean distance, denoted by d, points with three-dimensional coordinates (x, y, z) that belong to the set $\{ (x,y,z) \mid x^2+y^2+z^2 = 1, x,y,z \in \mathbb{R} \}$ define a sphere with centre the origin of the coordinate system (i.e. (0 , 0 , 0)) and radius of one unit.

In this chapter we will use the theory of implicit modelling to develop a modelling approach that we will then use to describe a new family of geometrical objects. These objects will be surfaces in general, usually in the three-dimensional space, unless some degeneracies occur. In order to create such objects, we will use an *extended* definition of the Euclidean distance.

We first have to justify our preference for using the implicit modelling approach. There are two reasons for our choice. Firstly, for analytically defined surfaces there already exist a number of sophisticated mathematical tools that enable their study (e.g. integration, partial differentiation). However, there are not many tools for adequately studying implicitly defined surfaces. Secondly, we believe that the implicit modelling approach is more powerful than the analytic. This observation stems from the fact that although any analytically defined surfaces may also be described implicitly, the reverse is not always true.

Take for example the surface that is determined by the three functions f_x, f_y, f_z of the two independent variables u, v: $(x,y,z) = (f_x(u,v), f_y(u,v), f_z(u,v))$, $u,v \in \mathbb{R}$.

With the implicit modelling approach, the same surface would be described by the following constraints: $\{ (x,y,z) \mid \exists\, u,v \in \mathbf{R} : x = f_x(u,v) \wedge y = f_y(u,v) \wedge z = f_z(u,v) \}$.

Such a conversion from analytical to the equivalent implicit definition may apply for any analytical expression, thence for all object definitions that use an analytical approach. But the conversion from an implicit surface definition to the equivalent analytical one is not always straightforward or feasible. The process of determining the analytical function that results from a given set of constraints may prove to be insurmountably complex. As a result, there are surfaces that can be implicitly defined but, due to the lack of an equivalent analytical description, they are impossible to study in a precise way. In the sections that follow we will see many examples of such surfaces.

Effectively, what we propose is the use of computer graphics techniques as a means of studying implicitly defined surfaces. We demonstrate the power of this approach by building a family of implicitly defined surfaces that are too complex to be described analytically, as is the case with the surfaces in plates 46, and 48. Then we will show how computer graphics methods may be used to visualize these surfaces. The appropriate position of the observer, the types of projection used and the shading models utilised will aid the conceptualisation of such surfaces. Moreover, metrics such as the area, the volume or the curvature of these surfaces may also be approximated.

Specifically, the method we propose is based on geometrical constraints that relate point coordinates with the function of the Euclidean distance. In this way, a generic definition of implicit surfaces will be formulated and analysed. In the next section we will give a precise mathematical definition for describing a mechanistic method for constructing geometrical objects. We will use this object generation method as a starting point for the construction of our modelling approach and for this reason we will call this method the *initial problem definition.*

Then, we will proceed to show how we can extend this initial definition and transform it into a significant modelling approach. This transformation will take place in two phases and is documented in sections 5.3 (phase A) and 5.4 (phase B). In the first phase (phase A), we

introduce a more 'intuitive' definition of the measure of distance; one that has been extended appropriately to be applicable in geometrical objects (as opposed to the classic Euclidean distance which is applicable to points only). In the second phase, we will then apply this new measure of distance on the initial problem, thus constructing a new definition for creating (describing) geometrical objects. The behaviour of this new modelling approach and its potential in describing new classes of geometrical objects will then be analysed. This exploration is achieved by assigning different interpretations to the constituent parts of our modelling approach. The concluding sections of this chapter will then be devoted to the application of our proposed modelling approach and the illustration of its potential using a variety of examples.

5.2 The initial problem

The modelling approach we propose is based on a simple mechanistic way of constructing geometrical objects. This is the method of *'pencil and string'* and is one of the first techniques used in geometry to construct objects. The classical example of this method is the definition of a circle where we tie one end of a piece of string to a fixed point on a given plane (two–dimensional space) and the other end around a pencil. Then we move the pencil so that its tip is always on the plane and the string is always taut. This method for defining a circle is mathematically expressed as:

$$\{\, p \mid p,q \in \mathbb{R}^2,\ d(p,q)=\delta \,\}$$

Where $d(p,q)$ is the Euclidean distance between the points p and q, q is the fixed point (centre of the circle), δ is the length of the string (the radius of the circle) and $\mathbb{R}^2$ denotes the plane on which the circle lies.

If we fix the two ends of the string to the plane, and allow the pencil to move so that its tip stays always on the plane and the string is always taut, then we construct an ellipse. The mathematical description of this construction is denoted:

$$\{\, p \mid p,p_i \in \mathbb{R}^2,\ d(p,p_1)+d(p,p_2)=\delta \,\}$$

where p_1 and p_2 denote the fixed end-points of the string, the foci of the ellipse thus generated, $d(p,p_i)$ is the Euclidean distance between points p and p_i, δ is the length of the string and $\mathbb{R}^2$ denotes the plane on which the ellipse lies.

The object definition method, that we use as the starting point for our modelling approach, is the general form of the 'pencil and string' method. In fact our initial problem definition is to calculate and subsequently visualize the 'locus of a point with the property that the sum of its distances from k given points is constant'. In other words, we study the geometrical objects (usually surfaces) that will result from the following implicit definition:

$$\{\, p \mid p,p_i \in \mathbb{R}^n,\ \sum_{i=1}^{k} d(p,p_i)=\delta \,\} \qquad \text{(Eq. 5.1)}$$

Where $d(p,p_i)$ is the Euclidean distance between the points p and p_i, k is the number of constituent points p_i and δ is a non negative real number that we will call the *defining parameter*. Moreover, $\mathbb{R}^n$ denotes the n-dimensional space used for the construction of the objects thus defined.

For this surface definition (denoted by Eq. 5.1), analytical solutions for the simple cases of $k=1$ and $k=2$ exist and the resulting surfaces have been extensively studied for both the two-dimensional and the three-dimensional space. Apart from these two cases, the majority of the geometrical objects (i.e. $k \geq 3$) generated by this object definition (Eq. 5.1) have not been studied. The reason is that the equivalent analytical definitions are very often too complex to calculate.

This initial problem definition — as expressed by equation (Eq. 5.1) — will form the basis for our implicit modelling approach. We will call the function used as the point membership classification test as the *defining constraint* and we will treat it as a *density function*. As such, we will assume that every point in space can be characterised with a density weight resulting from the application of the density function at this point. Then, for a given value of the defining parameter δ, which we will also call *density value*, we detect all points in space that evaluate to the same density weight (equal to δ). The locus of points with the same density value, which we call *iso-density contour* (or, *surface*) will then be treated as the geometrical object that we will visualise.

In two dimensions, the curvature of the produced shapes (i.e. contours) becomes apparent by the visualisation of various contours of incremental density values. In three dimensions, however, the curvature of the (iso-)surfaces may be sensed from the illumination effects (e.g. shading, highlights, etc.) that artificial light sources of a computer graphics visualisation algorithm produce. The choice of the observation point and the direction of view will be significant since in certain circumstances the front — in relation to an observer — parts of an object will obstruct the view. Additional information about the geometrical objects thus defined, could be gained if the researcher has the ability to alter his viewpoint freely with regard to the surface, preferably in real time.

The observation we need to make here for the object definition of the equation (Eq. 5.1) is that the defining constraint is expressed with the measure of the Euclidean distance. In three dimensions it is described by the formula:

$$d(p,q) = \sqrt{(p_x-q_x)^2 + (p_y-q_y)^2 + (p_z-q_z)^2}\,,\ p,q \in \mathbb{R}^3 \qquad \text{(Eq. 5.2)}$$

where $p \equiv (p_x\,,\, p_y\,,\, p_z)$, $q \equiv (q_x\,,\, q_y\,,\, q_z)$ are two points in the three-dimensional space.

The reason for concentrating our attention to this initial object definition described in the equation (Eq. 5.1) is twofold. First it demonstrates the potential of computer graphics; surfaces that are too complex to calculate and study can be (defined and) examined visually. Second, we will use this initial problem definition to explore a simple but powerful surface construction mechanism in order to provide 'extensions' to the definition of primitive geometric objects such as the sphere or the cylinder.

For these reasons, the object definition given in equation (Eq. 5.1) will be expanded so that we can describe numerous families of generalisations of known geometrical objects that in three dimensions will usually result into surfaces, provided that no degeneracies occur.

The generalization of the definition (Eq. 5.1) will evolve in two phases. First, we will expand the definition of the Euclidean distance so that we can use the distance of a point from a set of points. Then, in the second phase, we will extend the nature of the defining constraint so that it relates not only points but other more complex geometric objects as well.

5.3 Model development

5.3.1 Phase A. The extended definition of distance

Mathematically, the Euclidean distance is defined between two points (Eq. 5.2). As the definition of the equation (Eq. 5.2) shows, it maps a pair of points (p , q) onto a non negative real number which we shall denote as $d(p,q)$. In this first phase of extending the definition of the Euclidean distance we will explore how the measure of distance can be defined between a point and a set of points.

Let us consider A to be a set of points. We decompose the process of calculating the distance of a point p from the set A in two stages. In the first stage we will calculate the Euclidean distance of p from every point in the set A. The resulting values will be called *intermediate distance values*, or *intermediate values* for short. Then, in the second stage, we will use a function or a procedure that will combine the previously calculated intermediate values and will (possibly uniquely) determine the distance of p from A. Such functions and procedures could include the *minimum*, *maximum*, *average*, *the k^{th} member of a given ordering schema* etc. Let us assume, for reasons of clarity, the function of minimum. In this way, the extended distance $d_e(p,A)$ of a point p from a set of points A will be defined as the minimum Euclidean distance of that point p from all the points of set A. Specifically,

$$d_e(p,A) = \min_{x \in A} \{ d(p,x) \} \qquad \text{(Eq. 5.3)}$$

where $d(p,x)$ is the Euclidean distance between points p and x, and x is a member of set A.

The reason for using the function of minimum for the second stage of the calculation of the distance of a point from a set of points was purely for our convenience since it is straightforward to calculate and compared to the rest of the alternative functions, it is simpler to conceptualize. This choice is by no means compulsory and as we shall see in later sections it can be replaced by any other function or procedure. A selection of some alternative definitions of the extended measure of distance and their analytical and geometrical implications will be presented in sections 5.5.1 and 5.5.2.

5.3.2 Phase B. The generalized problem

In the previous section we briefly discussed a general technique to extend the definition of the Euclidean distance $d_e(p,A)$ in order to calculate the distance of a point p from a set of points A. Now, going back to our initial problem definition, as it was expressed by the equation (Eq. 5.1), we will use the extended distance definition (d_e) in order to achieve a more comprehensive surface generation method. This will become the objective of the second phase (phase B) of the construction of our modelling method, which will start by substituting in equation (Eq. 5.1) the Euclidean distance (d) with the extended distance (d_e). This will result into the following implicit surface description:

$$\{\, p \mid p \in \mathbb{R}^n, \sum_{i=1}^{k} d_e(p,A_i) = \delta \,\} \qquad \text{(Eq. 5.4)}$$

Observe that the points p_i of equation (Eq. 5.1) must now be replaced by the k sets of points A_i, where $i = 1, \ldots, k$.

This new surface definition (described in equation Eq. 5.4), is sufficient to describe the families of surfaces we are interested in. However, we will continue one step further in the refinement of this definition (Eq. 5.4) in order to make it easier to manipulate in our computer graphics visualisation algorithms. Specifically, instead of using sets of points (A_i) we intend to use collections of primitive geometrical objects in a fashion that a computer graphics designer is acquainted with. From the first sections of chapter 2 we have already agreed that the term primitive geometrical object is any geometrical object that the computer graphics designer uses as a building block to construct the required scene (e.g. a line, a line segment, a point, a plane, a torus). Combinations of such primitives from now on will be called *collections*, and will be used as building blocks for the construction of computer graphics scenes.

The utilisation of such defined collections will form the basic building blocks for a new computer graphics shape modelling approach. This approach will enable the user to produce a great variety of implicitly defined surfaces, hence expanding the applicability of computer graphics. The reason for using collections instead of primitives as primary building blocks is that in this way we can work with an arbitrary level of abstraction with regard to

modelling. This means that we have the freedom, as the next section illustrates, to customise the primary building blocks of this modelling approach according to the application needs. As a result, what seems to be a compound object in one application, may be used as a primary building block for another.

What follows in this section is the transformation of definition (Eq. 5.4) in order to accommodate collections of primitives. From (Eq. 5.4) we can calculate the distance of a point from a set. Consider now that this set is countable and finite. Such a set can then be equivalent, at least for our purposes, to the union of its members. Specifically, we assume that set A consists of m members:

$$A=\{x_1,x_2,...x_m\} \rightarrow A \equiv \bigcup_{j=1}^{m}\{x_j\}$$

In this way we can now replace the single-membered point sets ($\{x_j\}$), with primitive geometrical objects such as lines, planes, etc. (denoted as B_j). As a result the set A_i of definition (Eq. 5.4) will become a collection (C_i) of primitive objects ($B_{i,j}$).

$$A_i \rightarrow C_i = \bigcup_{j=1}^{m_i} B_{i,j}$$

Bringing all these ideas together, the families of implicit surfaces that we will investigate and visualize will be the ones formed by the definition:

$$\{\, p \mid p\in\mathbb{R}^n,\ \sum_{i=1}^{k} d(p,C_i)=\delta \,\} \qquad \text{(Eq. 5.5)}$$

Where, $C_i=\bigcup_{j=1}^{m_i} B_{i,j}$, and $B_{i,j}$ represents a primitive geometric object.

Moreover, $d(p,C_i)$ is the extended distance (as outlined in the previous section), and δ is a variable that we have already named as the *defining parameter*, and may be assigned a (usually non negative) real number which we called the *density value*.

The definition described in equation (Eq. 5.5) forms the basis for our modelling approach. It is a point membership classification test which we will use to define a variety of families of surfaces. In the following sections we will show how we can interpret the object definition of (Eq. 5.5) in order to generate families of 'intuitive' extensions to simple geometric objects, thus demonstrating the potential of the modelling approach that we propose.

We will examine the role of the defining parameter δ. Specifically, we will study the behaviour of the proposed object definition when the defining parameter δ is a constant number, a function, and a process.

5.4 Model exploration

5.4.1 The defining parameter being a constant

This is the typical case of the definition of iso-surfaces. The problem of defining iso-surfaces has been addressed by many researchers in various fields of study, like for example image processing, medical imaging, engineering and has been presented in the previous chapter. All these research streams share a heavy use of various linear or non-linear interpolation techniques. This is necessary because their problem starts off with a grid for values of the defining density function that is sampled at various locations within the space of their interest. From this grid then, they produce information about the whole of the sampled space.

Our modelling approach is also concerned with the definition of iso-surfaces, but, unlike the approaches used in other fields of study, we do not need to approximate the modelled surfaces. This benefit flows from knowing the defining function, hence we avoid any approximation in creating the modelled surfaces. Therefore, our approach resorts to approximation only when it is technically unavoidable, i.e. in visualising these surfaces where we need to convert the continuous space used in modelling into the discrete space of the pixel arrangement of the viewport (for a detailed discussion of sampling issues see section 3.5).

Consider our model as defined in definition (Eq. 5.5). First, we will examine the range of permissible values the defining parameter can have. Then, for some interesting cases we will discuss the geometrical implications of the produced surfaces. An example of such a study is depicted in plates 13 - 16).

Because we are using the Euclidean distance between points, by definition this is always a non negative real number. Therefore, the use of the extended distance definition from sets of points should also be a non negative number. As a result, the sum of non negative numbers will also be a non negative one. It follows that the first part of the defining constraint in (Eq. 5.5) will always be assigned a non negative value. Consequently, the use of a negative value in the defining parameter δ, will not make equation Eq. 5.5 true for any candidate point in space. Therefore, the resulting surface will degenerate to the empty set. But even when the defining parameter has a non negative value there are still cases where the resulting surface is empty.

For example, let us consider a model with a single collection of primitives where we use the definition of the minimum distance. In this example each non negative value of the defining parameter δ will produce a shape. In the extreme case of $\delta = 0$, the resulting shape is the (exact image of) the defining collection. This is expected because the points that are zero-distant[1] from a collection are only the constituent points of that collection. For any positive value of δ, the resulting surface will produce an approximated image of the defining collection. This approximation may be both from the outside and the inside of the collection, depending on the topology of the collection and the exact value of δ. Further details and examples of such surfaces will be presented in subsection 5.6.1.

5.4.2 The defining parameter being a function

The main characteristic of our surface definition (Eq. 5.5), is the use of a variable defining parameter δ that is determined by a function which we will call the *defining function*; for every point in space, or at least in the volume of space we are interested in, the value of δ will therefore be determined by a (defining) function which will usually accept as input (i.e. *input parameters*) the point's coordinates.

[1] We assume that we use the function of minimum for the measure of distance. Other functions such as the maximum will obviously behave differently.

Before presenting some illustrative examples, it is essential to understand how the defining function works. According to the category of implicit modelling (chapter two), a point in space is known to belong to the defined surface, only after it has been tested against a set of defining constraints which form the point membership classification test. Therefore, in the following discussion we will always assume that we are given a point p, which we will have to test against a given set of constraints shown in the definition of (Eq. 5.5).

In order to calculate the value of the defining function, however, its input parameters will first need to be determined. In most cases, these parameters are not the point's coordinates themselves, but a combination of them. This means that there exists a *mapping function* that relates these coordinates to the defining parameters. Therefore, with different mapping functions, the same defining function will produce different surfaces from the same model. Consequently, apart from the defining function, one will also have to determine the mapping function in order to give an accurate surface description.

It is very important to highlight the significance of the number of dimensions of the space we use, compared to the number of input parameters of the defining and mapping functions. This observation becomes useful during the analysis of the examples we present.

Consider a shape as defined by (Eq. 5.5). For reasons of clarity we reproduce Eq. 5.5 here.

$$\{\, p \mid p \in \mathbb{R}^n, \sum_{i=1}^{k} d(p,C_i) = \delta \,\} \qquad \text{(Eq. 5.5)}$$

In this description, the defining test will produce a curve for every permissible value of the parameter δ. The complete set of these contours will produce a contour map as we shall see in Figure 5.6. We will call this the *model's contour map*.

Let us now assume that the defining function for δ is also defined in the same space and is calculated by using all n space coordinates as equation Eq. 5.6 shows.

$$\delta = f(x_1, x_2, \ldots, x_n) \qquad \text{(Eq. 5.6)}$$

Furthermore, let us generate the contour map that is produced by all possible outcomes of the function $\delta = f(x_1, x_2, \ldots, x_n)$. We call this set of contours the *function's contour map*.

Specifically, this map is constructed by first fixing the value of $\delta = f(x_1,x_2,...,x_n)$ and then depicting all the points in space that evaluate the defining function to that pre-set value.

It should now become obvious that the resulting surface will be the intersection of these two contour maps (the model's and the function's). Such an intersection is defined by all the points in space that are characterized by the same density value at both (the model's and the function's) contour maps.

Another way of perceiving the same surface is by re-arranging the point membership classification test as definition (Eq. 5.7) shows. In this way, we calculate the *confluencing* iso-surface of the combined function for the defining contour value of zero (0). Specifically,

$$\{\, p \mid p\in\mathbb{R}^n ,\ \sum_{i=1}^{k} d(p,C_i)=f(x_1,x_2,...,x_n) \,\} \Rightarrow \{\, p \mid p\in\mathbb{R}^n ,\ \sum_{i=1}^{k} d(p,C_i)-f(x_1,x_2,...,x_n)=0 \,\} \quad \text{(Eq. 5.7)}$$

This arrangement can be achieved because function $f(\ )$ is assumed to use all n coordinates for its input parameters as the constraint function also does.

An interesting aspect of this case is that we can obtain quite complex 'objects' by applying very simple functions. See, for example, plate 20 which is generated using one rectangle as the only primitive of a model with only one collection. For this model, the defining function is the trigonometric function of *sin( )*. Further details about this model and other especially interesting cases will be presented in section 5.6.2. In that section we will also study models where the defining function does not use all n coordinates as input parameters.

5.4.3 The defining parameter being a process

So far we have investigated the case of the defining parameter δ being a constant and the general case of it being a function. By generalizing the nature of δ one step further, we can assume that its value could also be determined by a *process*. By process we mean any algorithm that, given a set of input values, will produce (determine) the value of the defining parameter δ (i.e. output). As input values, again an obvious choice would be to use (a mixture of) the coordinates of the point that we need to test against the model's constraints.

For this reason, such an algorithm should be *consistent* so that for the same input values it will always produce the same output value. With this restriction, several aspects of our visualisation approach will become easier.

When we do not use all the space coordinates of a point as input parameters for the determination of the defining process, we have some degree of freedom as to which coordinates to use. As we will explain in section 5.6.2, this issue is resolved with the use of a mapping function. In such a case, bodies of revolution as well as other types of surface modulation of the defining process can be achieved.

We have chosen to present two processes. The first process, explained in section 5.6.3, uses a pseudo-random number generator similar to the ones used for texture mapping, or sometimes to 'landscape modelling' [Angell & Tsoubelis 1992]. We use this number generator as a defining process that assigns a undulating surface around small collections of simple geometrical objects. Plate 23 illustrates our claims with a model of one collection of one primitive only.

The second process, also presented in section 5.6.3, illustrates how the definition of the Mandelbrot set [Peitgen & Richter 1986; Gleick 1988] may be adapted for our modelling approach. In particular, we perceive the definition of the Mandelbrot set as a bivariate process in order to rotate it around a particular axis as we demonstrate in plate 24.

Another type of process that we are going to use extensively for determining the defining parameter δ is another implicitly defined iso-surface. Such defined surfaces include a variety of simple known geometric objects like the parabola, where we determine the locus of points that are equidistant from both a given point and a given infinite line, as well as other more complex structures like the Voronoi tessellation. For this reason, the necessary definitions and some further analysis of our generic surface definition (Eq. 5.5) will be presented separately in the next subsection.

5.4.4 The defining parameter being another implicit definition

In this section we will make use of the definitions of the *nearest* and *second nearest* primitives. Therefore, before presenting this model, we will first discuss the formation of these definitions (i.e. nearest). From definition (Eq. 5.3) we saw that the distance from a point to a set of points can be associated with the concept of the minimum distance. In other words, with the distance of that point from the nearest point of the set. Furthermore, we also saw in this chapter that for the calculation of the distance of a point from a collection of primitives we may use the minimum distance from that collection. In such a case specifically, we first calculate the minimum distance from each primitive, and then choose the minimum of these.

This implies that an ordering amongst the primitives of each collection can occur according to their distance from any given point. Actually, this ordering is feasible because each collection consists of a countable, finite and therefore individually identifiable set of primitives. Therefore the definitions of the nearest and the second nearest primitives are attainable. The reason for utilising such an ordering schema will become obvious in the next paragraphs, where we will need to identify points that are equidistant between two different primitives of the same collection.

In this section we investigate the case where the defining parameter δ is determined by another implicit definition (Eq. 5.5). In this way, the model for our objects becomes:

$$\{\, p \mid p \in \mathbb{R}^n ,\ \sum_{i=1}^{k} d(p,CA_i) = \sum_{j=1}^{l} d(p,CB_j) \,\} \qquad \text{(Eq. 5.8)}$$

The surfaces that result from definition (Eq. 5.8) will consist of all the points that are equidistant from the nearest primitives of collections CA_i and CB_j. In other words, the definition of equation (Eq. 5.8) describes the intersection of iso-surfaces that are produced by the *CA* and *CB* sets of collections. In fact this was expected, since the defining parameter in (Eq. 5.8) is actually a process.

This model definition can be applied to generate known mathematical surfaces such as the three-dimensional paraboloid (plate 25), as described in section 5.6.4. The power of our modelling approach lies, however, in its ability to produce with no further effort generalisations of known surface definitions thus creating families of such surfaces. Examples of this case are the paraboloid-like shapes in plates 27, 29 and 31, also described in section 5.6.4. There, we will also present the models that we used to build these surfaces.

Another interesting family of surfaces comes from the same definition of equation (Eq. 5.8) with the following assumption; consider that *CA* consists of only one collection and that *CB* consists of the same collection as *CA* (i.e. $CA = CB$). For such a definition, the resulting surface would always be the whole space because both members of the constraint are identical. For this reason our generic surface definition (Eq. 5.8) is adjusted to the following

$$\{ p \mid p \in \mathbb{R}^n , d_1(p,C) = d_2(p,C) \} \qquad \text{(Eq. 5.9)}$$

where d_1 denotes the distance from the nearest primitive in collection C and d_2 the distance from the second nearest primitive of the same collection C. If C consists of points only, the resulting surface(s) is a Voronoi tessellation, or *diagram*. To recall from chapter four, the Voronoi diagram defines for each point, or nucleus, its corresponding neighbourhood so that any point in space is nearest to the owner of the neighbourhood it belongs to, than to any other (Figure 5.1).

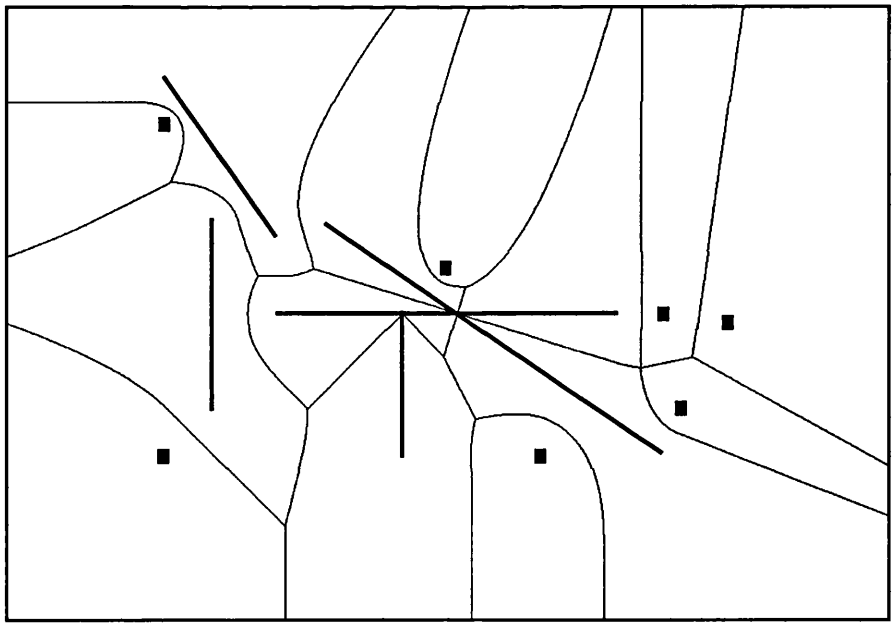


Figure 5.1 Voronoi diagram using 7 points and 5 line segments

But with the surface description expressed in equation (Eq. 5.9), the concept of the Voronoi diagrams can be extended greatly. In particular, we suggest that instead of using only points for the definition of the Voronoi nuclei, other simple geometric objects such as lines, or polygonal facets should be used. In this way, we extend the definition of the Voronoi tessellation making it applicable to virtually any geometrical object our modelling approach can describe. Plate 33 demonstrates a tessellation where line segments are used as nuclei. Moreover, using our modelling approach we can also assign weights to the nuclei of the Voronoi tessellation, and visualise the resulting surfaces (tessellations) as plates 47 and 48 demonstrate. Thus, we can study the effects that changes to these weights produce for a given nuclei arrangement. An example of this, is the sequence of plates 34 - 44, explained in section 5.6.4.

5.5 Discussion: mathematical and geometrical implications

5.5.1 Mathematical implications

In this subsection we shall analyse and evaluate the options that are available to us during the two stages of extending the definition of the measure of distance. This is useful because it will help us understand the consequences of using a particular function to extend the definition of the measure of distance. What is assumed to be known is a set of points A, and a point p that may or may not belong to set A. What we intend to determine, is a way for measuring the distance of point p from this set.

We can distinguish between two cases, regarding the nature of this set; the *finite*, where the set A consists of a finite number of points, and the *infinite*, where the set A consists of an infinity of points that is usually determined by the locus of a point that describes a geometrical object such as a line segment, or a torus. Therefore, we shall present two examples to cover both the finite and the infinite category of sets. For each such example, we will present and evaluate the alternatives for defining the measure of distance, and we shall justify our preferences. We start with the finite category of sets.

Finite category

Consider the finite set A of the following five points in two-dimensional space as Figure 5.2 shows: $A = \{x_1,x_2,x_3,x_4,x_5\} \equiv \{ (1.1 , 1.3) , (2.5 , 1.9) , (3.4 , 1.1) , (5.1 , 0.1) , (1.9 , 0.3) \}$. Let us now assume that for a given point, say $p = (3.6 , 0.6)$, we need to calculate its extended distance $d_e(p,A)$ for the set A. At the first stage of our calculations we will have to determine the intermediate values v_1, v_2, ..., v_5. These values represent the Euclidean distance of p from every point of the set A (i.e. x_1, x_2, ..., x_5). Then, in the second stage, we will have to determine the minimum of these intermediate values and use that as the extended distance $d_e(p,A)$.

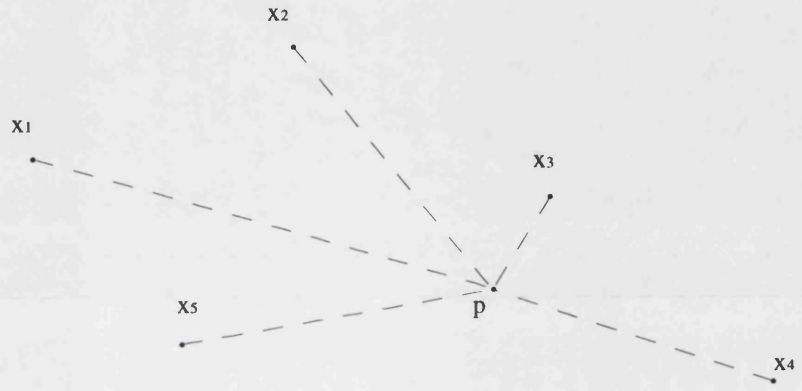


Figure 5.2 Calculating the distance of a point from a finite set

STAGE 1 Intermediate values	STAGE 2 Distance of p from A according to formula:
$v_1 = d(p,x_1) = 2.6$	*Minimum* $d_{e1} = 0.58$ (v_3)
$v_2 = d(p,x_2) = 1.7$	*Maximum* $d_{e2} = 2.6$ (v_1)
$v_3 = d(p,x_3) = 0.54$	*Average* $d_{e3} = 1.636$
$v_4 = d(p,x_4) = 1.58$	2^{nd} *in incremental order* $d_{e4} = 1.58$ (v_4)
$v_5 = d(p,x_5) = 1.73$	*Weighted average* d_{e5} = Any (depends on weight vector)

Table 5.1 The two stages for the calculation of the extended distance from p to A

The results of this example are summarised in Table 5.1, where we have calculated the extended distance $d_e(p,A)$ using the *minimum* (d_{e1}), *maximum* (d_{e2}), *average* (d_{e3}), 2^{nd} *in incremental order* (d_{e4}) and *weighted average* (d_{e5}) functions. All but the last of these alternatives can be calculated. The evaluation of the weighted average depends on the values of the weights that must be assigned to all the members of the set *A*.

Infinity category

Let us now consider another set of points, say ***B***, that unlike set ***A*** consists of an infinite number of points that are determined by the locus of a point which describe an infinite line (Figure 5.3). We shall try to calculate the extended distance $d(p,B)$ by applying the same alternatives that we used in the previous example. However, unlike the previous example, the outcome of the first stage of calculations (i.e. that of the intermediate values) does not produce a finite set of intermediate values (v). Instead, there is an infinity of intermediate values and their minimum is realized for the point x_0 which is defined as the intersection point between the perpendicular line that passes through the point p and the line represented by the set ***B*** (Figure 5.3). Consequently, all intermediate values belong to the continuous interval of real numbers $[d(p,x_0), \infty)$, where x_0 is the nearest point of ***B*** from p (Table 5.2).

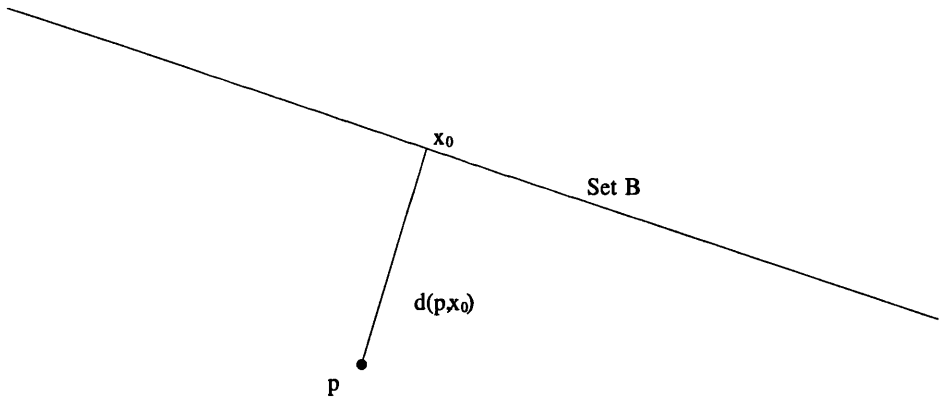


Figure 5.3 The distance of a point from an infinite set

We can observe here, in the second stage of this example, that most of the extended distance definitions $d_e(p,B)$ are unsuitable for further calculations. Specifically, d_{e4} cannot be calculated since the set of intermediate values is infinite and uncountable. Definitions d_{e3} and d_{e5} can also be inappropriate because they may become too time consuming to determine,

since they involve the calculation[2] of integrals. Consequently, from the extended distance definitions that use the functions of minimum (d_{e1}) and maximum (d_{e2}), we have the choice to use either.

STAGE 1	STAGE 2
$v \in \{ d(p,x) \mid x \in B \} =$ $= [d(p,x_0) , \infty)$ where x_0 is the nearest point in B from p.	*Minimum* $d_{e1} = d(p,x_0)$
	Maximum d_{e2} = (depends on B)
	Average d_{e3} = (depends on B)
	2nd in incremental order d_{e4} = Not applicable
	Weighted average d_{e3} = Any (depends on weight vector)

Table 5.2 The two stages for the calculation of the extended distance from p to B

In most of our examples we used definition d_{e1} (minimum) for the following three reasons. Firstly, the notion of infinite distance, as the use of maximum (d_{e2}) implies, is more difficult to conceptualize when compared with the concept of the minimum (nearest) distance. Secondly, the measure of the infinite (∞) distance is relatively more difficult to incorporate in calculations. Thirdly, by using the minimum function, the range of the permissible intermediate values, in most cases can be neither negative nor infinite. This observation stems from the mathematical definition of any 'distance' function that must

- produce non-negative results
- give the same outcome if its arguments are transposed
- verify the triangular inequality

Consequently, a definition for extending the measure of the Euclidean distance may be that of d_{e1} which calculates the minimum of all the intermediate values v. This conforms with the general mathematical definition of distance and is relatively easy to conceptualize, calculate and implement algorithmically.

[2] In specific cases, there exists a formula for calculating an (infinite) integral, but in general we will need to approximate it using numerical analysis methods.

We have to stress here that there are other alternatives that we may use for extending the definition of the measure of the Euclidean distance. Such alternatives may simply be introduced in the appropriate visualisation algorithms. For example, in chapter seven we show the contour maps produced from the use of the weighted inverse square distance function.

However, if one needs to use another (other than the minimum) function to extend the definition of distance, one needs to be aware of the specific restrictions this function imposes on its operands. For example, the distance formula d_{e4} (i.e. the 2nd in incremental order) is limited to countable sets of points where an ordering schema may be imposed. This is the only way where one can determine the k^{th} member of that order and hence, calculate the required distance. Furthermore, if two or more members evaluate to equal intermediate values, the k^{th} member in this ordering schema may be impossible to identify uniquely. Such inconveniences do not become unsurmountable problems, but may produce disconnected iso-surfaces. It is clear that a continuous range of intermediate values will not be suitable to such a definition of distance.

5.5.2 Geometrical considerations

In the previous sections we were concerned with the determination of a new modelling approach for computer graphics. These efforts started with a general concept of a problem (Eq. 5.1) and concluded with a mathematically well defined problem description as expressed in (Eq. 5.5). During this process, we defined and used a number of concepts that include the *extended distance*, the *nearest object*, the *primitive geometrical objects*, and the *collection*, which we believe are important and will prove useful to the rest of our investigation. Here, in this subsection, we will explore the geometrical implications of the treatment of the measure of distance, as we have defined in the previous sections. We illustrate its significance through a series of examples. Moreover, where appropriate, we will relate our findings to other research concerned with implicit modelling, thus producing a more comprehensive analysis.

The first example considers the line segment ***AB*** in two-dimensional space, as Figure 5.4 shows. The distance of point ***P*** from ***AB*** will be the length of the line segment ***PO***. The length of a line segment will be determined as the Euclidean distance between its two defining vertices (***A*** , ***B***).

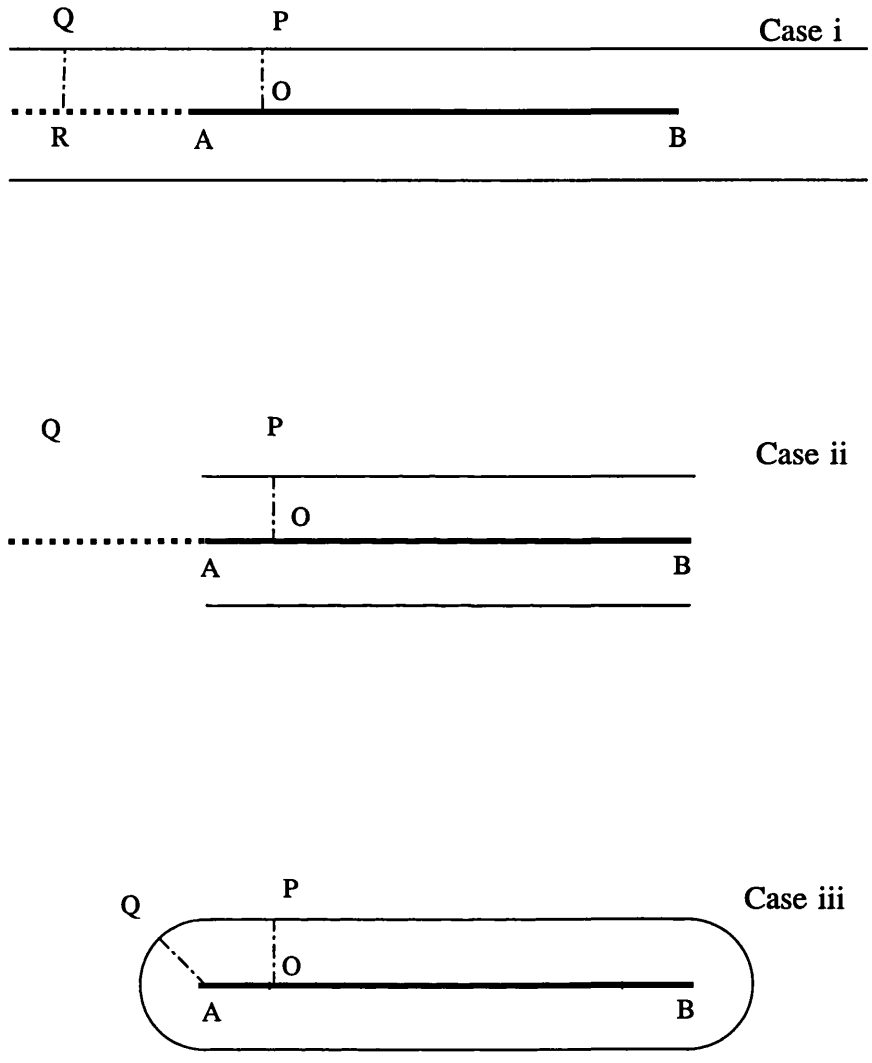


Figure 5.4 The calculation of distance ***d(P , AB)***

Consider now another point ***Q***. Its distance from ***AB*** could be defined in three different ways. The first is found in all geometry textbooks and involves the virtual extension of the line segment ***AB*** until it intersects with the line that passes through ***Q*** and is perpendicular to ***AB***. For the second, we propose to prohibit the arbitrary use of virtual extensions to the

line segment AB, thence leaving the value of the required distance from Q to AB as *undefined.* Finally, the third approach applies the extended definition of the measure of distance, as we proposed it in equation (Eq. 5.3). Table 5.3, summarises the above alternatives, and Figure 5.4 illustrates their geometrical implications.

Case	Value
i	The length of the segment QR, since this is the distance of Q from the infinite line which extends the segment AB.
ii	Undefined, since when we draw the line from Q, perpendicular to AB, it does not intersect with the segment AB.
iii	The length of the segment QA, which is the minimum distance of Q from the set of points that form the line segment AB (definition Eq. 5.3).

Table 5.3 Different cases for evaluating the distance of a point from a line segment

According to the three different cases of evaluating the distance of a point from a line segment, three different contours would emerge (Figure 5.4) for any given value of the defining parameter δ (Eq. 5.5). It becomes apparent that the first case (i) produces a disconnected contour of two parallel infinite lines. Similarly, in case (ii), we also get a disconnected contour that consists of two parallel line segments. From the definition of case (iii), however, a continuous contour emerges. This consists of two parallel line segments — similar to the second case (ii) — but now their end-points are connected with two semi-circles. Therefore, case (iii) is the one consistent with our extended distance definition as given in equation (Eq. 5.3).

The significance of case (iii) becomes apparent when using polygonal lines in the form of collections. For example, consider Figure 5.5. Here, the model consists of six line segments properly placed to form the skeleton of the capital letter 'M'. These line segments can also be seen as a collection (C) thus enabling the modelling and the effortless manipulation of a complete letterset. Following the notation established in the previous sections, the shape of Figure 5.5 will be given by the following definition:

$$\{p \mid p \in \mathbb{R}^2,\ d(p,C) = \delta\}$$

With the use of different values for the defining parameter δ, a family of shapes, all approximating to the letter 'M', can be produced. For small positive values of δ, the resulting shape resembles very closely the defining polyline (i.e. the collection). This method has already been used by Bloomenthal [1990] and others under the name of 'skeleton filling' as we discussed in chapter four.

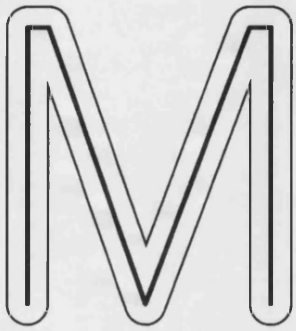


Figure 5.5 The skeleton of capital letter 'M'

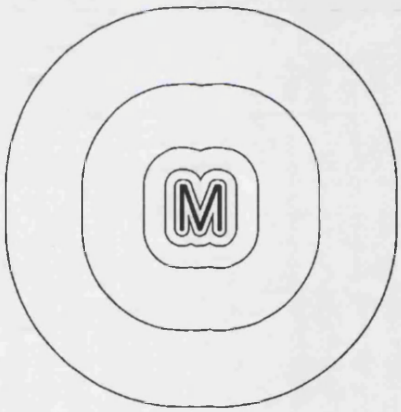


Figure 5.6 Contour maps of the capital letter 'M'

In contrast, the use of very large values of δ will produce distorted images of the defining skeleton. As a rule of thumb, in most cases, the larger the value of δ is the more the resulting contour resembles a circle. In Figure 5.6, for example, the same model of the capital letter 'M' that was used in Figure 5.5 is visualized for values of δ in the range of several orders of magnitude larger compared to the size of the defining line segments, thus producing a number of contour maps.

5.5.3 Summary

In presenting the implicit modelling approach we have developed, we begun by analyzing the measure of the Euclidean distance, and we arrived at a number of alternatives for extending it. Further analysis guided us to choose definition (Eq. 5.3) as the extended definition for the measure of distance between a point and a set of points. In this way we finally defined the distance between a point and a geometric object.

Our modelling approach initially considered a simple model for surface description which is expressed with equation (Eq. 5.1). Through a series of definitions and assumptions we derived an enhanced and significantly more 'intuitive' mathematical description for generating implicitly defined surfaces as equation (Eq. 5.5) expresses. This final model for surface generation will now be evaluated through the analysis of several interesting cases. We anticipate that with this analysis the capabilities of our modelling approach (as determined by the model of Eq. 5.5) will be demonstrated, thus giving to the reader a more comprehensive view of its significance. The conceptual schema of the final model we adopted is also depicted in Figure 5.7. There, we illustrate the decisions we take in order to construct our models and we also indicate some of the choices we preferred to follow.

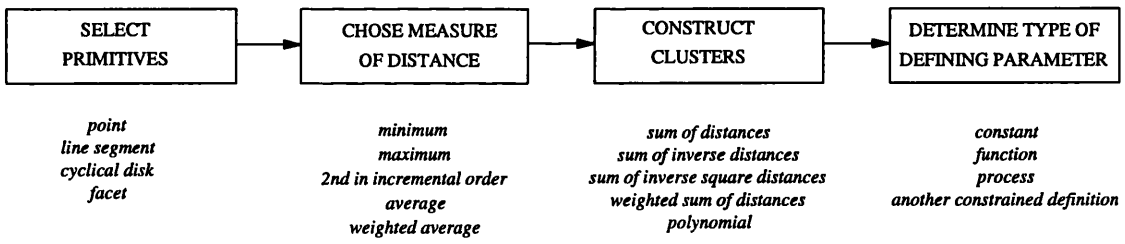


Figure 5.7 The conceptual schema for implicit model construction.

This final model is now capable of describing many simple and already known geometrical objects. This model's power lies in the ability to describe new objects and forms by perceiving them as generalizations of other more simple ones. We can envisage further enhancements for our modelling approach such as various types of surface generalisations and other more complex constraints.

The types of extensions that we applied to the initial model were necessary because they enabled us to explore a wider variety of surfaces compared to those covered by the initial model. An important issue in this research is that throughout the process of model expansion we have avoided the use of analytical means in order to demonstrate the potential of combining old but very intuitive object description techniques with the capabilities of modern computing methods in computer graphics. Such methods have been abandoned in the pre-computer era due to the complexity involved in studying them. The first 'wave of efforts' in computer graphics was focused on analytical object descriptions that definitely present several limitations when compared to the modelling approach proposed in this dissertation. It is only very recently that research in computer graphics has re-discovered the power of implicit modelling (chapter four). We hope that this research contributes to this effort, and the examples that follow will show its power.

5.6 Applying the modelling approach

The presentation of the modelling approach that we propose would not be complete unless we provide an extensive analysis of its potential. This will be achieved with the presentation of several examples that demonstrate how to construct a variety of different families of geometric objects. In some examples, we will also demonstrate the power of this modelling approach by showing its consistency in utilisation and similarity in results when compared with other modelling approaches. We form this presentation according to the nature of the defining parameter δ as we also did during the initial analysis.

5.6.1 The defining parameter being a constant

Objects that are described with a single collection model are sufficient to produce a number of known and well studied geometrical objects. Repeating a previously discussed example, a collection that consists of one point will produce a circle in two dimensions, or a sphere in three dimensions with centre the defining point, and radius the non negative value of the defining parameter δ.

When the collection represents a circle of radius r, however, the resulting surface will then become a set of two concentric circles in two dimensions. These circles will have a distance of δ from the circular collection and would reside at either side of the collection. Their radii would be $(r+\delta)$ and $(r-\delta)$. If $(r-\delta) \leq 0$ then the inner circle is degenerate. In three dimensions, the resulting surface is a torus with a defining radius equal to r (the radius of the collection), and its shape will develop around that circular collection with radius δ.

Another family of objects that can be produced with such a model, of a sole collection, is that of generalized cylinders. Generalized cylinders were first introduced by Agin and Binford [1976]. Since then, many researchers have used them in various applications such as object recognition, scene recognition, volume representation etc. For an overview of such work the reader is referred to Shani and Ballard [1984]. According to them, "a generalized cylinder is a representation of an elongated object viewed as having a main axis (spine) and a smoothly varying cross section".

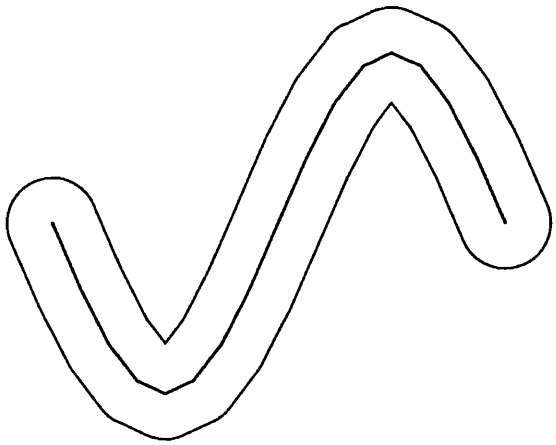


Figure 5.8 Iso-surface calculated along a polyline

In this section, our model is capable of defining generalized cylinders with spines of arbitrary shapes, but with circular cross section of constant radius δ only, because we assume models with one collection and a constant defining parameter. This poses some restrictions about the smoothness (analytical continuity) of the produced surfaces. It becomes apparent when the spine is a polygonal line and the cross section is circular of constant

radius (Figure 5.8). In such a case, the produced surface will be analytically continuous (i.e. differentiable) in all places except for the joints of the polygonal line of the spine where it is only geometrically continuous (i.e. the surface is not segmented but it is not differentiable either). In the next sections, we will see how arbitrary cross sections can also be modelled in order to produce a complete family of generalized cylinders.

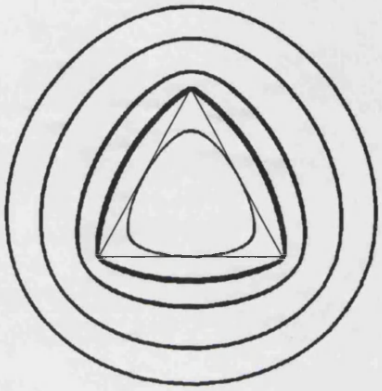


Figure 5.9 The sum of the distance from three points

By using object descriptions, such as those defined by equation (Eq. 5.5), with more than one (non empty) collection, the choice for the value of the defining parameter is more restricted. Consider, for example, an ellipsoid. This is defined by two collections each containing one of its *foci*. When δ is non negative but less than the distance between the two collections (i.e. its two foci), the resulting surface is the empty set. In this case therefore, δ has to be greater or equal to that distance. In the extreme case of δ being equal to the distance between the two foci, the resulting surface is the line segment that connects the two collections.

In plates 6 - 9 we depict the contour maps of models with one, two, three and four collections, of one point each, respectively. In all four plates we used the function of the minimum distance. However, we can use any other distance definition, like for example, the function of the inverse distance as plate 10 demonstrates.

In Figure 5.9 we also illustrate the contour maps defined by a much simpler model. It is model with three collections each of which contains one point as its only primitive. In three-dimensional space the surface generated by the same model is depicted in plates 11 and 12.

We can imagine in plate 11 the location of the three primitives; they are the vertices of the resulted triangular shape. One of these vertices, we may observe, is also responsible to the lack of surface smoothness near the lower left side of the plate. This effect is attributed to the choice of the defining parameter δ which, for this surface, was taken to be approximately equal to the sum of the distance of this vertex from the other two. In this way we can assure that the thus generated surface will pass nearby this vertex. In plate 12 we see the same surface from a different viewpoint. We positioned the observer so that he lies in the plane defined by these three primitives, so that we can see details of the curvature of surface that were not visible in the previous plate (plate 11).

When the collections are more complex than single points, or more collections are involved, the minimum permissible value for δ is more difficult to determine as the sequence of plates 13 - 16 illustrate. In this sequence, we used a model of polylines that constructed the skeleton of the capital letters 'ΕΠΥ' and varied the value of the defining parameter δ considerably. In this way the outline of the letters (plate 13) disappears by the 'inflated' surface that a high value of δ generated (plate 16).

5.6.2 The defining parameter being a function

In the following example we will demonstrate how we can generate the 'graphical representation of a univariate function that is defined along a line'. Although this may seem to be a very trivial task, the way we model the graphical representations is very important and powerful because it enables us to generate very complex, three-dimensional graphical representations that are defined not only on a Cartesian coordinate system, but along any geometrical curve or surface as well. It is only for reasons of clarity that this example is described in the two-dimensional space.

Suppose that the defining function for δ uses only one input parameter (say, u). For every point $p \equiv (x, y)$ in the two-dimensional space, in order to evaluate the defining function ($\delta(u)$), the value of u will have to be calculated first. This is achieved by somehow combining the point's coordinates. A simple choice is to use the x- coordinate (i.e. $u \equiv x$) and ignore the y- coordinate. Therefore our defining function will become $\delta(x)$. Suppose

also that (Eq. 5.5) consists of one collection of an infinite line that coincides with the horizontal (say x- axis) of the space's assumed Cartesian coordinate system. In this way, the resulting surface definition becomes:

$$\{\, p \mid p \equiv (x,y)\ ,\ d(p,C) = \delta(u) \,\} \rightarrow \{\, p \mid p \equiv (x,y)\ ,\ d(p,X_{axis}) = \delta(x) \,\}$$

Geometrically, this means that the resulting shape will be the graphical representation of the univariate defining function $\delta(u)$. Moreover, the choice of the line in the only collection of the model should not necessarily coincide with the x- axis of the assumed Cartesian coordinate system. It can be a line of any orientation, as long as the appropriate mapping to the parameter u is correct. Here, however, there are two issues that need attention: first, is the case where the defining function takes 'prohibited' values (e.g. becomes negative), and second is the fact that the resulting graphical representation will be also *mirrored* along the x- axis.

With regard to the first issue, there are two methods. The first and simpler method is to shift (i.e. translate) the defining function away from the x- axis. If the function is bounded, as for example the sine (*sin()*) function, an appropriate translation is sufficient. If, in contrast, the defining function cannot be bounded (e.g. $\delta(x) = x$) we either try to establish some local boundaries and shift accordingly, or use its absolute value that is always non negative. The alternative (second) method we may employ is to truncate and ignore all the parts of the surface that are produced by negative values of the function $\delta(u)$.

With regard to the second issue, a symmetrical (i.e. mirrored) image along the x -axis will always appear since the measure of distance does not exhibit a sense of orientation (i.e. it is irrelevant from what half-plane we approach the x -axis). If the symmetrical image is not desirable, one should modify the defining function $\delta(u)$, so that it would enable a check to determine the half-plane from which the distance is calculated:

$$p \equiv (x,y)\ :\ \delta_{modified}(p) = \begin{cases} \delta(x) & \textit{if } y \geq 0 \\ 0 & \textit{elsewhere} \end{cases}$$

We can now illustrate the effects of object definitions in the space of three dimensions. Here, there is a wider range of possibilities with regard to the number of parameters that this

function may have. Nevertheless, what seems to be important is the difference between the degrees of freedom of the defining function with respect to the dimensionality of the space in which it is defined. Therefore, in a way analogous to the two-dimensional case, when the defining function depends on all three coordinates (x , y , z), the resulting surfaces emerge from the intersection of the model's and defining function's contour maps.

However, when the defining function depends on two parameters only, say the u, v we will need a mapping function: $(x,y,z) => (u,v)$. Let us assume that the object definition consists of one collection, which is a plane, say the X - Y plane that passes from the origin of the coordinate system. The shape of the resulting surface depends on the selection of the mapping function. Suppose for example that $(u,v) \equiv (x,y)$. In this way, the z coordinate of any point will be the equal to the (Euclidean) distance of that point from the defining collection (i.e. the X - Y plane). As a result, the produced surface will be the graphical representation of the defining function, which now is defined along the X - Y plane. Again here, as in the previous example, the same two issues (i.e. 'prohibitive' function values, and 'mirrored' images) have to be taken into account.

Let us now take another model that consists of one collection which is defined by one infinite line in the three-dimensional space. Here, a mapping that provides surfaces useful in terms of computer graphics modelling applications stems from the following procedure: suppose that somewhere along the defining line there is fixed point that we will call the *origin*. Assume also that we can impose a direction along that line. Consequently, any given point on that line, apart from the origin,[3] will define a vector starting from the origin and ending at that point and its direction would be either the same or the opposite to the predefined one.

In this way, one of the parameters of the defining function (say v) may be the Euclidean distance of a point from that line. And the other, (u), may be analogous to the length of the line segment that is defined by the origin and the projection of the given point onto the line. By using this mapping to determine the parameters u, v the resulting surfaces will become

[3] If the end point of a vector coincides with its starting point, which in our example is the origin, then it has zero length and its direction is undefined.

the *body of revolution* around the defining line, as plate 17 shows. The model in this plate consists of one collection only which defines a line segment as the only primitive. Observe in this plate how the end-points of the line segment are rounded off by semi-spheres of a radius that is determined by the evaluation of the defining function at these end-points respectively.

In this sense, when the defining collection is a line, this being an infinite line or a line segment, the resulting surfaces for δ being a function, or even a constant, are bodies of revolution as plate 18 shows. In this plate we show how the trigonometric function of *sin( )* is rotated along a line segment (the primitive of a model with one collection).

Moreover, interesting shapes will also emerge when the defining collection is a polyline which is also called *a broken axis of revolution.* In this case, surfaces will be produced from rotation along different line segments. Here, attention should be given to the choice of the mapping function if the resulting shape is to become geometrically continuous. In plate 19 we see a model of a polyline (of one collection of one primitive) which has as defining function the *sin( )*. The segments of the polyline are connected to form a 'broken axis' and the defining function *sin( )* is defined along the polyline. We can observe the way the surface behaves around the joint of the constituent line segments. Surface continuity (geometrical) is assured by the way we calculate the defining function along the polyline.

Another interesting case is that depicted in plate 20, where, the same *sin( )* function is defined along the longer dimension of a rectangle. The surface of this model has been modulated along the rectangle's longer dimension with the *sin( )* function which has been extruded along the rectangle's smaller dimension.

In a similar way we can also create a model of a surface when the defining collection consists of one primitive of one point only. Here, this (defining) point can be used as the origin of an alternative coordinate system, like that of *mercator* coordinates where any point on the surface of a sphere is determined by its orientation along the equator, namely the *longitude*, and its distance from the assumed north pole which is called the *latitude*. Hence, for any given point, its mercator coordinates could be used for the parameters (u, v) of the

defining function $\delta(u,v)$. In this way, the defining function is 'mapped' onto the surface of a sphere (i.e. the one used for the corresponding calculations of the mercator coordinates). In other words, the surface produced is the result of the 'modulation' of a spherical surface by the defining function. This means that the surface that could have been produced by using a constant value for δ (i.e. a sphere), is now modulated in accordance with the defining function that replaces δ.

Consequently, research directed towards texture mapping [Heckbert 86], inverse mapping [Haines 89], or wrapping can now be used in order to exploit surface modulations further by using models similar to these presented here in the above examples. Another classical application here is the mapping of environmental variables (e.g. atmospheric temperature, ozone density) that are measured using the mercator coordinate system of the spherical-like earth. For more details on this issue the reader is referred to the spherical plots modelling approach presented in chapter four.

5.6.3 The defining parameter being a process

When using random numbers for the determination of δ, two issues arise. The first has to do with the consistency of the random numbers used, and the second with the continuity (i.e. smoothness) of the produced surface. With regard to consistency, algorithms that produce pseudo-random numbers [Kernighan & Ritchie 1988] can be used as long as their initial *seeds* remain unchanged during the visualisation process. With regard to surface smoothness however, some additional techniques have to be used. In order to control the continuity of the produced surface, randomly produced values are permitted to oscillate only between specific limits depending on the values of their neighbouring points. One way to achieve this is by first calculating the 'smooth' value through interpolation of neighbouring ones, and then by adding to that an amount of noise. Dietmar Saupe [1989] covers this subject with more detail by presenting the midpoint displacement method as well as spectral synthesis and functional based approaches. Additionally, methods using stochastic noise synthesis can also be used. Other references include [Mandelbrot 1982] and [Voss 1985].

By utilizing such a technique, a controlled but also (pseudo) random process can be defined. Plates 21 and 22 show the effects of such processes for two different values of the control parameters. In plate 21 the variation (smoothness) of the pseudo-random numbers is globally controlled, where, in plate 22 the smoothness is less apparent and more locally confined. The pseudo-random numbers produced by these processes are colour-coded, therefore continuity is expressed by colour adjacency with respect to the spectrum of visible light (i.e. the rainbow spectrum).

We can use this process in order to create a pseudo-random surface around a line segment. Plate 23 shows a model of one collection of one point only that has such a pseudo-random process as the defining constraint. It is important to observe that the produced surface is fragmented and consists of at least two separate pieces. This type of observation can only be made after the visualisation of such a model, since analytically is too difficult to determine. Once such 'discontinuities' are located, further studies on the model are possible. This pseudo-random process uses all three coordinates as input. Therefore, the surfaces produced, as explained in the previous section, are the intersection of the model's and the process's contour maps.

Benoit B. Mandelbrot in the late 1970s was the first to attempt a description of the subject of fractals [Mandelbrot 1977]. Since then the subject of fractals together with that of Chaotic Dynamical Systems [Devaney 1989] have evolved into an individual domain of research. The definition of the Mandelbrot set is given in relation to a process for generating sequences of numbers [Peitgen & Richter 1986].

In our example we are going to use the following process: Assume a point p in two-dimensional space. The n^{th} member of the sequence $(S_{n,p})$ that stems from p will be:

$$S_{n,p} = S_{n-1,p}^2 + p \ , \ S_{0,p} = 0$$

This sequence either converges to zero or approaches to infinity (∞). It has been proved, however, that this sequence does not converge to zero if the norm of at least one of its members is larger than 2. Specifically:

$$if \ \|S_{n,p}\| > 2 \rightarrow S_{n,p} \rightarrow \infty$$

and the norm of a two-dimensional point is the length of the vector it defines:

$$\|p\| = \sqrt{p_x^2 + p_y^2}\ ,\ p \in \mathbb{R}^2$$

The density value used for the produced contour maps is the integer number n that denotes the order of a specific number $S_{n,p}$ for which the above defined 'divergence' criterion is satisfied. If after a number of iterations (for the calculation of $S_{n,p}$) the sequence has not proved to diverge, the process for calculating members of that sequence terminates, and the density value for the corresponding point p is assigned to be the maximum number of iterations. This maximum number will be called the *resolution* of the set.

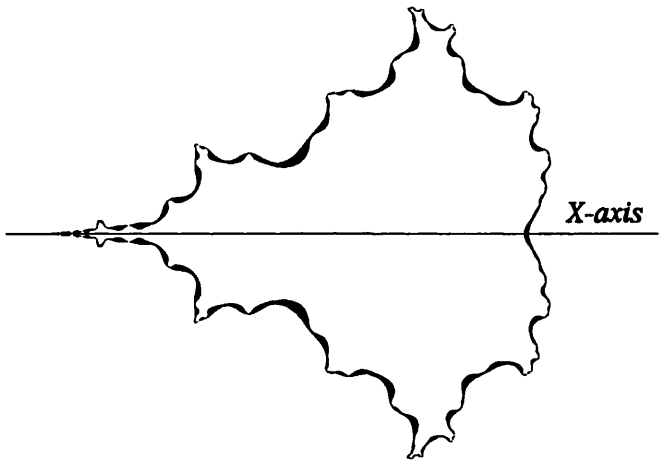


Figure 5.10 *X*– axis symmetry of the Mandelbrot set

In Figure 5.10 we can see the borders of the Mandelbrot set at resolution 12. We can observe the variations of thickness of the borders of the set which verify its fractal nature. A simple rule to observe here is that the larger the resolution is, the more complex the border appears. For this reason any attempt to approximate these borders with polygonal lines will be incorrect since for a different resolution this can be arbitrarily wrong. Because of the simplicity of its generation rules and the complexity of its resulting contour maps this set has become popular.Let us assume now such a process to be the defining process for a model consisting of one collection of one infinite line. The mapping function has to be adjusted so that the resulting surface would be the rotated Mandelbrot set along the x- axis. For this reason, the x- axis is mapped with the appropriate scaling onto the defining line thus determining the first input parameter (u). The second input parameter (v) for this fractal process will be determined by the distance from that line segment.

Let us also assume that the Mandelbrot process has a given resolution (say 12). In plate 24 we see the rotated surface thus generated. At the right top corner of the same plate we also see the contour of the borders of this Mandelbrot set. As the previous paragraph stressed, the fractal nature of the Mandelbrot contours amplifies the potential of our generic model (Eq. 5.5) to define and manipulate such complex object descriptions.

The choice of the x- axis as the axis of rotation is used because of the symmetry of the Mandelbrot set (as it was defined here) along the x- axis as Figure 5.10 shows. As a result, any other mapping from the coordinates to the input parameters of that fractal process would result into a distorted view.

5.6.4 The defining parameter being another implicit definition

This is a special case of the defining function being a process, but as we discussed in section 5.4.3 it must be presented separately. This particular treatment has also allowed us to adjust the model definition of Eq. 5.5 to that of Eq. 5.8 and the particular case of Voronoi diagrams (Eq. 5.9), that for reasons of clarity we also present here:

$$\{\, p \mid p \in \mathbb{R}^n ,\ \sum_{i=1}^{k} d(p,CA_i) = \sum_{j=1}^{l} d(p,CB_j) \,\} \qquad \text{(Eq. 5.8)}$$

$$\{\, p \mid p \in \mathbb{R}^n ,\ d_1(p,C) = d_2(p,C) \,\} \qquad \text{(Eq. 5.9)}$$

Consider, for example, the case where CA is one only collection consisting of one primitive which is one point (p), and CB is one collection that contains one primitive which is an infinite line denoted by l. Let us also assume for this example that these collections do not intersect. The definition of (Eq. 5.8), in two dimensions, will produce a parabola (Figure 5.11). This figure, shows the line l as a dotted line for reasons of clarity.

We must note here the problem of accuracy that dominates the visualisation techniques used for this category of models and which we intuitively call the *thickness of the surface*. The contours in Figure 5.11 are getting thicker as we trace the curves away from the defining collections (l and p). The reason is that for a given surface thickness t, more points satisfy

the inequality $d(q,l) - d(q,p) < t$ for all points q, as we move away from both l and p. If we reduce the level t, we will loose trace of the contour in the area near the defining primitives.

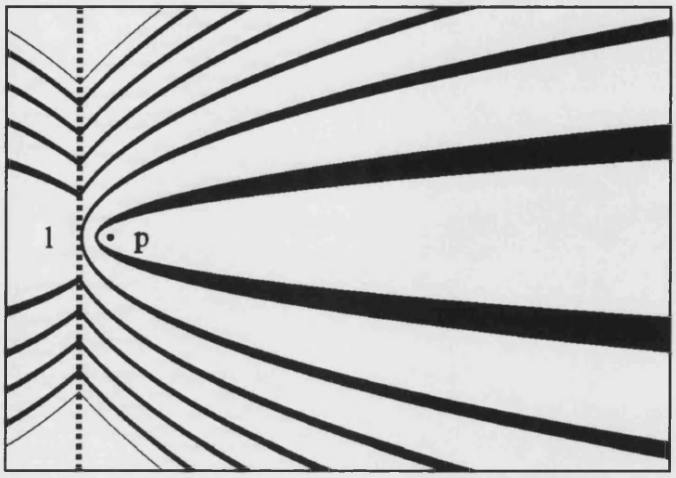


Figure 5.11 A parabola defined by an infinite line

If in the example above the *CB* collection was a line segment instead of an infinite line, then following case (ii) of Figure 5.4, the resulting shape would have been truncated accordingly. When we use the minimum distance (case iii of Figure 5.4), beyond the truncation points, the parabola shape degenerates into infinite lines (Figure 5.12). Again for reasons of clarity, the line segment ***l*** is denoted as a dotted line.

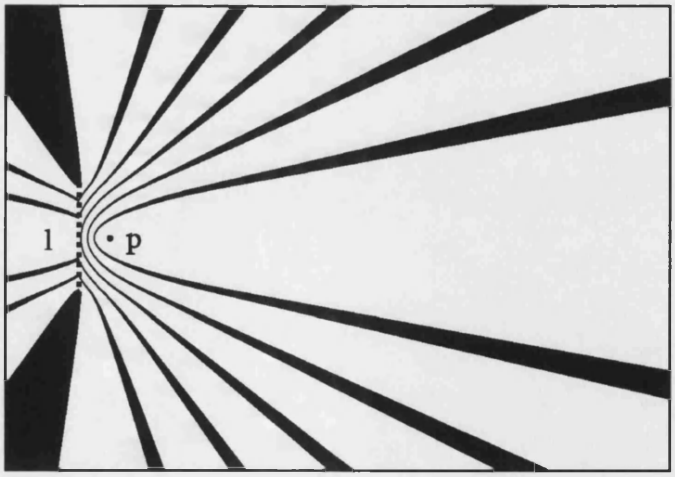


Figure 5.12 A parabola as defined by a line segment

To illustrate the capabilities of our modelling approach, we will explain how we can extend effortlessly the definition of the paraboloid in order to define a family of paraboloid-like objects, as plates 25 - 31 show. A paraboloid is defined as the locus of points that are equidistant from a given point and a given plane that does not pass through that given point. In plate 25 we can see a paraboloid such defined. Because the paraboloid extends to infinity for reasons of image clarity we had to truncate its surface when the distance of its constituent points from the given plane (and/or given point) exceeded a predefined threshold. As a result, we could replace the given plane with a planar disk of a large enough radius so that it would not affect the shape of the truncated paraboloid. In this plate (plate 25) we can see a turquoise-coloured disk and the resulting paraboloid above it. In plate 26, which is the same (plate 25) paraboloid model but, observed from a different viewpoint, we can see the 'inside' (other side) of the paraboloid where the defining point (turquoise-coloured sphere) is also visible.

Our first experiment was to generate a paraboloid-like surface by applying the two-dimensional definition of the parabola and visualise it in the three-dimensional space. In this way, the defining primitives, denoted with the turquoise colour, were a line segment and a point (plate 27). The thus generated surface is also depicted in plate 28 where we can also see that it is an extruded parabola along the direction perpendicular to the plane that passes through the defining primitives.

The next experiment on the definition of the paraboloid was to replace the defining point with a line segment. In this way we visualised, in plate 29, the locus of points that are equidistant from a given plane (denoted by the turquoise planar disk) and a given line segment (denoted as a turquoise line segment). Another view of the same object is also shown in plate 30. Following that line of experimentation we then replaced the paraboloid's defining primitives with two given line segments non-intersecting and perpendicular to each other. The thus generated surface is depicted in plate 31. For reasons of clarity the sides (faces) of the generated surfaces have been assigned different colours and three light sources have been used to provide the shading and highlights.

Another series of experiments was also conducted on the surface definition of Eq. 5.9. This definition expresses the problem of the Voronoi tessellation, but also allow us to enhance it considerably. One such example of the Voronoi tessellation is shown in plate 32 where we used nine points (denoted with turquoise spheres) as nuclei. Our first enhancement was to use as nuclei line segments instead of points. In plate 33 we show the tessellation defined by using three line segments as nuclei. Taken in pairs, all segments are perpendicular to each other but they do not intersect. Since some of the resulting surfaces may extend infinitely we had to impose an upper limit threshold in a fashion similar to that of the paraboloid examples. Moreover, in order to make the nuclei visible, at least partially, we had to impose a lower limit threshold below which no surfaces were visualised. This lower limit stops visualisation of all points on the visualised surface that are closer to the nearest primitive a distance smaller than the lower limit threshold.

The second of our enhancements to the definition of the Voronoi tessellation was the assignment of weights to all participating nuclei, these being points, lines, or any other geometric object. To better understand the effect that weights had on the thus generated tessellations, we constructed a series of tessellation images (plates 34 - 44) for the same arrangement of nuclei, but with varying weights. Specifically, we used a model of four points as nuclei which are denoted as turquoise-coloured spheres. The weights of the three nuclei (the ones on the right, below, and left) were set to the same value 1.0, while the weight of the top nucleus was let to vary between 0.75 (plate 34) and 1.25 (plate 44) in steps of 0.05. As a result, eleven plates were produced.

To conclude our experiments of this category of model descriptions, we also generated plates 45 - 48 with models of several points (4 in plate 45, 9 in plate 46) and line segments (plates 47, 48) as nuclei, all being assigned different weight values. In these plates we used a colour coding schema which assigns the same material properties, including colour, to all the surfaces that face the same nucleus they are nearest to. However, because of the definition of the Voronoi surfaces, where two nuclei are equidistant, we paint each visible surface according to the nearest or second nearest nucleus, depending on the orientation of the surface in relation to the observer.

5.6.5 Discussion on the applications of the modelling approach

In this section we have seen some examples of the models generated by the approach we have developed and described in this dissertation. We have concentrated on interesting cases of models for different categories of the defining parameter δ.

In the first category, where the defining parameter is constant, we saw how the generated shapes are affected by the exact value of δ. In the second category, we demonstrated the use of functions as a means to modulate surfaces. In other words, we saw how we can visualise a particular function that is defined on the surface of another object. We also illustrated there how we can generate bodies of revolution and envelopes of shapes that are defined along a trajectory. Then, in the third category, where the defining parameter is a process, we showed how pseudo-random or fractal processes may also be used in order to modulate the surface of geometrical objects.

Finally, in the last category, where the defining parameter is another implicit definition, we demonstrated the potential of the modelling approach that we developed through two especially interesting cases; that of surfaces that are equidistant from two implicitly defined objects, and that of the extensions that we attached to the definition of the Voronoi tessellation problem.

We believe that the examples we used are indicative of both the simplicity and the power of the modelling approach that we have presented in this dissertation. In the next chapter (six) we will turn our attention to the issue of visualisation. There, we will describe the visualisation approach that we have adopted, which matches the requirements of our proposed modelling approach, thus producing a complete tool for manipulating implicitly defined surfaces. This visualisation approach, which we also used for the production of all the plates in this dissertation, further enhances our modelling approach as it allows us to demonstrate how we can exploit its principal capabilities: its intuitive nature in form description, its power for generalising object definitions, and its support in refining our conceptualisation of new geometrical objects and shapes.

Chapter 6 Visualisation of implicit surfaces

6.1 Introduction

In the previous chapter we saw how the measure of distance can be used to create many families of surfaces, some of which have never been modelled before, while others are radical generalizations of well known sets of surfaces. Specifically, starting from the definition of the initial model described in (Eq. 5.1), we reached surface descriptions such as the ones represented by equations (Eq. 5.5), (Eq. 5.8) and (Eq. 5.9). After defining a surface using the above equations, the next task will be, as it has already been stated, the use of computer graphics techniques in order to visualise it. This visualisation process we believe should have two different but complementary targets. The first should aim at the exploration of such models and therefore should offer the user 'real time' manipulation. This means that the user should be able to vary the degree of approximation to the model's surface in exchange for quicker visualisation time. The second target should aim at the production of a 'realistic surface representation' thus broadening the range of building blocks (i.e. shapes) that are available to the current computer graphics user. As a result, a visualisation method is needed that would avoid, as much as possible, the use of arbitrary assumptions[1] about the model.

Although the above targets are complementary, there are cases where both cannot be reached at the same time. This is especially true where the highest quality picture is needed for the study of the model's details, thus delaying visualisation speed due to the enormous calculation demands. There are also cases where some coarse approximations to the surface are welcomed when visualisation speed is a necessity. In this chapter we will present the visualisation approach that offers the highest image quality. This is based on the octree model definition and representation method [Clark 1976; Meagher 1980; 1982; Doctor & Torborg 1981]. Following this approach we will illustrate how we can get a very accurate representation of the modelled surfaces. Then we will criticise our visualisation approach and discuss the issues involved in improving its speed and accuracy.

[1] In various stages of the visualisation process some assumptions about the geometry of the surface could simplify the necessary algorithms and significantly speed up the whole process.

Specifically, in this chapter we will present the visualisation approach adopted, and justify the selection. Furthermore, we will explain the adjustments we felt necessary in order to enable the effective visualisation of implicitly defined surfaces. Finally, we will elaborate on the problem of choosing the suitable point on the implicitly defined objects that will be of use as the representative for the corresponding pixel in the resulting image.

The visualisation approach used is primarily based on the octree algorithm. This algorithm was presented on the initial chapters on modelling (chapter two) and visualisation (chapter three) and we will extensively use the terminology defined there. However, in this chapter, we will examine the octree approach from the perspective of the programmer. As such, a number of new issues like programming environment, data structures, number precision and error tolerances will need consideration.

6.2 Design considerations

The programming environment we chose is that of the Object Oriented Programming [Henderson 1993] and its specific implementation is the C++ programming language [Borland 1992; Stroustrup 1987]. The C++ programming language was chosen because of its adaptability as both a high but also a very low level programming tool. The C++ extensions were also preferred for the same reason since they enable programming at an even higher level of abstraction. To be more precise, the C++ programming environment allows the control (and direct manipulation) of individual lines of hardware ports and the linkage of assembly language instructions, as well as the handling of complex abstract entities (i.e. classes or objects) like the palette, the scene and the window. This environment has proved very convenient to use, not only during the initial prototyping stages, but also in the development of the final software application. Moreover, implementations of the C++ programming environment exist on all the common hardware and operating system platforms. For example, there exist the shareware GNU compiler for all the major UNIX implementations, and the Borland C++ environment for the DOS, Windows, and OS/2 platforms of Intel ×86 compatible hardware. Many other manufacturers provide implementations of the C++ programming language such as the Microsoft C/C++, the IBM

XLC++, the Watcom C/C++, the ZortexC++, etc., all offering the ANSI standard implementations alongside their own proprietary libraries that are targeted at specific application domains (i.e. mouse manager, numerical approximation algorithms, etc.). Therefore, portability of C++ programs is not an obstacle provided only the standard function libraries are used.

Once the choice of the programming environment was made, the next important issue was the utilisation of the Object Oriented mechanisms. The initial design for the complete software implementation was very critical since it would affect not only the efficiency but also the capabilities of the resulting application. In an Object Oriented environment, the most fundamental building block is the object. This is an instantiation of an abstract data structure which is user-definable and is called class. A class describes data of a particular form and a number of methods that are processes designed to manipulate on this data. One of the most important features of a class is the inheritance. This property allows the hierarchical definition of subclasses that stem from a parent class. As such, subclasses share the data forms and methods of their parent class and consequently determine the data and methods that their own subclasses will inherit.

The advantages of using the Object Oriented approach to computer programming have been listed in many books, and can be summarized in the following key areas: modularity, ease of testing, maintainability, reusability. However, the main disadvantage of the object Oriented approach is rarely mentioned and refers to the choice of the appropriate class hierarchies that would reflect accurately, and enable the effective implementation of, a particular application domain. Fortunately (!) applications of computer graphics are the most favourable example for Object Oriented textbooks and one can anticipate a variety of alternative designs to appear. But the majority of the references prefer to treat points in space as the most important class, from which the rest of the representations of simple geometrical objects (e.g. line, circle, etc.) must inherit. The potential of this *pixel-based* design approach proved to be neither appropriate to our implementation requirements, nor powerful enough to accommodate the variety of viewports needed in a real environment.

However, apart from the pixel-based design approach, another three candidates were examined. The first is based on the *geometrical object* which is the primitive building block of the model and the second is the *subcube* or the *cubelet* which forms the primitive cubical space on which the octree algorithm is based. The third alternative, and which is the one we adopted, treats the complete modelled scene as the primitive class for our design.

6.3 The current implementation

The octree method as described in the previous chapters constructs what we will call the *traditional octree* approach. This approach will also form the basis for our implementation. Nevertheless, in order to make our implementation capable of visualising the surface descriptions given in the previous chapter in a more effective way, we have introduced some amendments to the traditional octree. For this reason, in this section we will present the most important differences of our approach compared to the traditional octree. First, with regard to modelling, we will show how our model definitions are used with the octree method. This to a great extent is aided by the introduction of two basic assumptions. Some criticisms about them and a presentation of other alternatives will follow. Finally, we will present how during visualisation all the information necessary for the rendering of the appropriate pixels (i.e. intersection point, direction of the normal, etc.) can be established.

6.3.1 The implicit to octree model conversion

The implicit surfaces defined in the previous chapter have not been described with octree data structures. Therefore, in order to use an octree based visualisation algorithm, we will have to somehow convert the implicit description of a surface into the equivalent octree based one. Specifically we will need to extract information regarding the homogeneity of cubes of space (i.e. whether a cubelet contains any points of the modelled surface). This process of model conversion, however, exhibits some complicated elements that are addressed and resolved through the introduction of two assumptions that we will present in this section. The first assumption regards the shape of the octant we will use, and the second regards the mapping of octants in three-dimensional space with pixels on the viewport. Our

rationale behind these assumptions, as we will see in the next paragraphs, emerges from the lack of knowledge about the behaviour of the constraint (and the defining) function inside a particular volume of space.

For the process of model conversion we will follow the method described in the octree modelling subsection of chapter three; the surface is surrounded by a supercube, the intersection test (i.e. whether a particular subcube intersects the surface) is applied and the subcube subdivision proceeds accordingly. Each subcube corresponds to an octant and the minimal sized subcube is called *cubelet* and corresponds to a voxel. Information about the homogeneity of the cubelet will be used to determine the colour of the voxel and subsequently the colour of the corresponding pixel onto the viewport. But the answer to the intersection test may sometimes become a prohibitively time consuming process, especially when we do not exploit or we do not have basic knowledge about the geometry of the modelled surface (e.g. bounding volume information).

Because the models that we will visualise greatly involve the function of distance, knowledge about the existence of one point on the modelled surface may help us to make similar inferences about neighbouring points with regard to the same surface. The exploitation of this observation will become obvious if we rephrase the intersection test along the lines of the following argument.

As we have already mentioned, the defining constraint of an implicit definition can also be seen as a function, namely the *constraint function*. This constraint function may take any real value depending on the value of the input coordinates, thus assigning every point in space one real number. In this way, if we allow the input coordinates to take any value inside the volume of a subcube, the constraint function will take a variety of values. If, additionally, the constraint function is continuous the resulting range of values will also be continuous and form an interval, the *constraint interval*. Since the surface consists of all the points that fulfil the defining constraint (i.e. make the constraint function equal to the value of the defining parameter) the intersection test can be transformed into: **whether the constraint interval that is defined inside the volume of a particular subcube includes the value of the defining parameter**.

In order to illustrate the calculations involved for the determination of the constraint interval let us assume an implicit surface definition that consists of one primitive geometrical object only, which is a line (l) in the three-dimensional space, as equation (Eq. 6.1) specifies.

$$\{p \,|\, p \in \mathbb{R}^3,\ d(p,l) = \delta\} \qquad \text{(Eq. 6.1)}$$

This primitive is denoted with l and is represented in Figure 6.1. From this definition we can infer that the generated surface will be a cylinder of a circular cross-section with radius equal to δ. Let us also consider a subcube with its centre denoted by C at a particular location so that is intersects with l. For that point C, we evaluate the constraint function. This value, which belongs to the constraint interval, will be by definition the distance of C from l. Let us also denote with A the point on l that this distance is calculated from.

The homogeneity of the subcube will be decided from the intersection test. The re-phrased intersection test, as it was presented earlier in this section, examines whether the value of the defining parameter δ is included in the constraint interval or not. Therefore, the limits of the constraint interval will have to be determined. In order to determine the minimum and maximum extremes of the constraint interval we use the triangulation inequality property of the distance function. For this reason we draw the line that connects A with C. The nearest to A intersection point of this line with the subcube will be the one that minimizes the constraint function. Similarly, the furthest from A intersection point with the subcube will maximize the constraint function.

The complexity of the calculations for the nearest and furthest intersection points of the line with the subcube depend on the orientation of the subcube in relation to that line as Figure 6.1 shows. A simplification to this calculation overhead comes from the first of our assumptions where instead of the cubical space that the subcube defines we use the volume defined by the subcube's circumscribing sphere. In this way, once the distance of C from the surface is calculated, the minimum and the maximum extremes of the constraint interval are calculated if we respectively add or subtract the radius of that sphere from that distance. Therefore, we do not need to draw the AC line, nor do we need to find the intersections of that line with the particular subcube.

Although this assumption simplifies the calculations involved, it is not very accurate since sometimes it misclassifies some homogenous subcubes as heterogenous, thus resulting into wasteful calculations. A thorough investigation about the advantages and disadvantages of this assumption will follow in the next subsection.

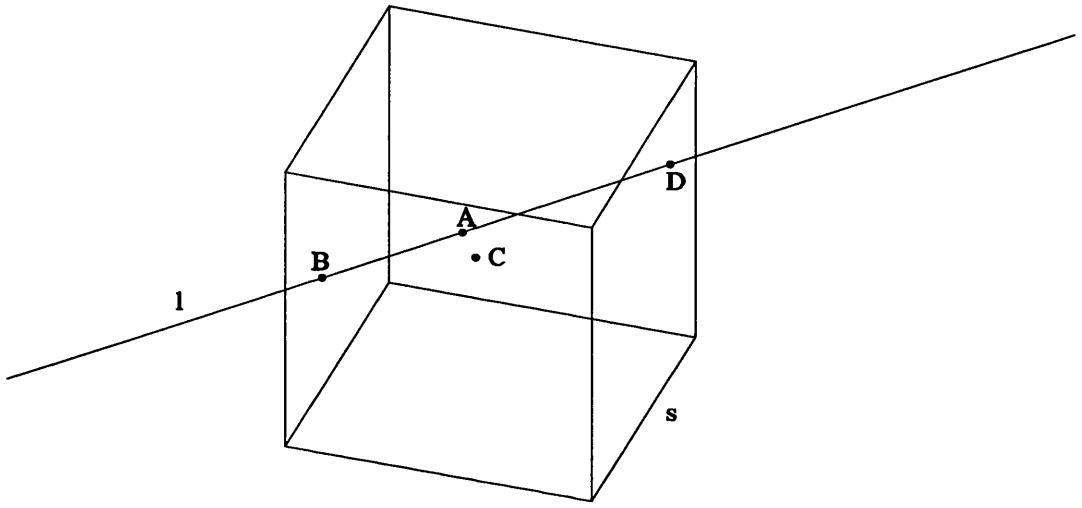


Figure 6.1 The inconvenience of using **cubical** subcubes

To analyse the effectiveness of this assumption concerning the utilisation of spherical subcubes, let us assume that this particular subcube of Figure 6.1 has an edge length equal to s. The constraint function for point C will be expressed as $d(C,l) = d(C,A) = v$ and the resulting constraint interval for the cubical space defined by the subcube will be:

$$[d(A,B),\ d(A,D)]$$

while for the spherical one will be $[(v-r),\ (v+r)]$, where r denotes the radius of the subcube's circumscribing sphere and $r = s\frac{\sqrt{3}}{2}$

So far, we used surfaces that are defined by a single primitive only. In a more general case, where there is only one collection of primitives, the primitive that is nearest to the subcube's centre C is first identified. Then we apply the calculations we have just presented for that primitive. Eventually, in the case where more than one collection of primitives are

combined, we decompose our calculations as follows. For every collection we determine the corresponding constraint interval. Then these intervals are combined appropriately, i.e. according to the constraint definition. Then the combined constraint interval is used for the application of the intersection test, as we saw in the previous example.

But not all constraint functions are as simple as that in the above example. Moreover, there are also cases where the constraint function is not continuous and therefore there is more than one constraint interval.[2] In such cases, we use the following systematic way for tracing the surface: at every intersection test between a subcube and the surface we assume that the test is true and therefore the corresponding octant needs to be subdivided. Eventually, when we reach at the level of the voxel a realistic answer to the intersection test must be given.

This comes from the second of our assumptions where: we assume that a voxel corresponds to a cubelet of space that is small enough (for the resolution of our viewport) to be treated as a single point. This point is assumed to coincide with the cubelet's centre. The rationale behind this assumption is that the voxel corresponds to a single pixel on the viewport and therefore all points inside the cubelet (that correspond to that voxel) will contribute to the colouring of a single pixel on the viewport. It is evident that this exhaustive technique for the generation of the octree structure demands enormous[3] time to implement, therefore, is has to be used only as the last resort.

Once the constraint interval has been determined, we need to apply the intersection test. Actually, we have to answer whether the value of the model's defining parameter belongs to the constraint interval or not. But as we saw in the previous chapter, the model descriptions of the surfaces we aim to visualise are categorised — according to the nature of their defining parameter — in three different classes. The classes *I*, *II*, and *III* refer to models where the defining parameter is constant, function, or process. From this last category (*III*) we will make a discrimination to differentiate between the defining parameter

[2] In the later case, all intervals are calculated and their union for the rest of our discussion will also be called *constraint interval.*

[3] For the production of a model of reasonable resolution (ie. 512 × 512 × 512) the point membership test of the constraint of the implicit definition should be calculated a maximum of 134,217,728 times (=512^3). For a high quality model, at 2048 × 2048 × 2048 resolution, the above test needs to be applied a maximum of 8,589,934,592 times.

being another implicit definition, which will form a new class of models (class *IV*), from it being any other process. Therefore, in the rest of this subsection we will concentrate on how the intersection test is evaluated for each of the four model classes. We will start with models of class *I*, because they are usually the simplest. Then for every other class, we will present a way for transforming the corresponding intersection test into a class *I* equivalent, thus giving a unified solution to the intersection test.

In class *I* models the defining parameter is described by a constant number. Therefore, the intersection test is transformed into a *point membership classification test.* In other words whether the value of the defining parameter belongs to the constraint interval. However, in order to establish the basis of a unified approach to answering the intersection test (and cater for the solution of the class *II*, *III* and *IV* models) we will assume that the value of the defining parameter (say δ) is represented by an interval, the *defining interval*, which for the sake of consistency is defined as the next equation shows : $\delta = [\delta, \delta]$.

Therefore, the intersection test is now considered to be a test of **whether the constraint interval intersects with the defining interval**. For the rest of our discussion, this will be the operational definition of the intersection test. This definition aids at the unification of our approach since as the next paragraphs show, for the rest of the model classes all we need to do is change the extremes of the defining interval appropriately.

In the second model class (*II*), for example, where the defining parameter is a function, the extremes of the defining interval will be the minimum and maximum values the defining parameter may take inside a particular subcube. This presupposes that we know the behaviour of the defining function. For example, the function $f_1(x,y) = \sin(x+y)$ of the two coordinates x , y is a periodic function that can take any value between
$-1.0 \leq f_1(x,y) \leq +1.0$.

With regard to the third model class (*III*), we again estimate the minimum and maximum values of the defining parameter δ. An interesting case emerges here when δ is a fractal process. The estimation of the defining interval's extremes may be a very complex and time consuming process. For example, if we use the Mandelbrot set, as illustrated (plate 24) in

the previous chapter, a rough estimation about the defining interval would be **[0,max]**, where *max* denotes the maximum number of iterations needed before terminating the calculation of the fractal process. But the use of such a wide defining interval will result into more frequent intersections with the constraint interval, and therefore, more wasteful octant subdivisions. Since the Mandelbrot fractal set is time consuming to calculate this rough estimation of the defining interval is not recommended.

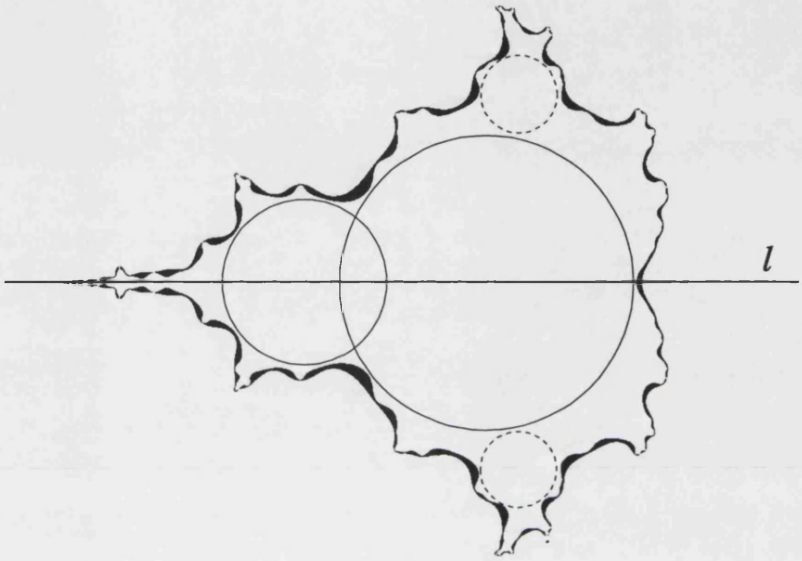


Figure 6.2 Speeding up the visualisation of the Mandelbrot set

As a solution, we recommend the use of bounding volume information. There are many choices available to us in this stage. For example, to surround the fractal set with other simpler to calculate geometric objects. However, in our implementation we achieve a better visualisation speed by excluding from the octree all points that lie in the 'inside' of our model. In this way, we avoid costly calculations of the defining process. Specifically, we define a set of two spheres and a torus that: if a subcube is found inside any of these three objects, then the corresponding defining interval is empty (i.e. it concerns points far away from the set's border) and therefore the intersection test is negative. In Figure 6.2 we see a cross-section of the model we have just described, on a plane that passes through the model's axis of symmetry (denoted by *l*). Here, the intersection of the torus with this plane is two circles (denoted by dotted perimeter) since its defining axis is made to coincide with axis *l*.

Eventually, in the fourth model class (*IV*), where the defining parameter is another implicit definition, we treat the constraint functions as follows. The constraint that defines the surface is an equation, with both parts being implicit definitions. The left hand part of the equation is treated as the constraint function and used for the determination of the constraint interval. The right hand part of the equation is again treated as a constraint function but now its corresponding constraint interval will become the model's defining interval. Once both intervals are determined, the intersection test can be answered in a way analogous to that of the other model classes; we have to determine whether the constraint interval intersects with the defining interval.

So far, we saw how a surface described by an implicit definition, as this was determined in the previous chapter, can be converted into an octree encoded form for its subsequent visualisation. By rephrasing the question of the intersection test we saw how we can exploit the implicit definition and achieve a model conversion effectively and efficiently. Additionally despite the differences in the descriptions of the implicit surfaces we aim to visualise, we showed how this model conversion can be achieved with a unified approach. The usefulness of such a unified approach is appreciated during the implementation of the techniques described, and relates to issues of simpler coding, code re-usability, increased expandability, and code maintainability. Furthermore, with the adoption of two assumptions, we made this model conversion process attainable even for complex definitions and more rapid in its implementation. Because the choice of the assumptions is critical to the rest of the implementation, in the next subsection we will present some alternatives to the above assumptions, compare them, and finally draw some conclusions about their usefulness.

6.3.2 Criticisms about the assumptions

In the first assumption, we use the subcube's circumscribing sphere instead of the subcube (cubical shaped) itself for the calculation of the constraint interval and therefore for the evaluation of the intersection test. This, compared to original approach of using cubical shaped subcubes has some advantages and disadvantages. The main advantage of using spherical subcubes is the simplification of the calculations involved. This, consequently, results in simpler algorithms and quicker implementations. Specifically, with spherical

subcubes we do not need to determine the extremes of the constraint interval geometrically by drawing the AC line, computing the intersection points, and evaluating the constraint function, but we do it analytically by simply adding and subtracting the radius of the circumscribing sphere. In this way, the orientation of the subcube relative to the surfaces of the model is not important, but only its location and size.

The disadvantages of this assumption however, are not insignificant. They stem from the fact that the volume covered by the cubical subcube is considerably less than the one covered by its circumscribing sphere. Specifically, for a cube of size s its volume is s^3, the radius r of its circumscribing sphere is

$$r = s\frac{\sqrt{3}}{2}$$

and its corresponding volume,

$$\frac{4}{3}\pi r^3 = \frac{4}{3}\pi s^3 \left(\frac{\sqrt{3}}{2}\right)^3 \approx 2.72 s^3$$

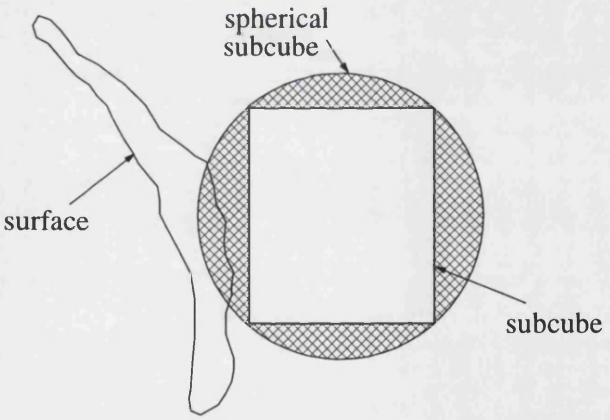


Figure 6.3 Cubical and spherical subcubes

Consequently, the intersection test that is answered for a large spherical volume, is used for the octant subdivision of less bulky subcubes. Therefore, it is likely that a subcube that does not intersect a surface of the model, may be incorrectly characterised as heterogenous if that surface is close enough to the subcube so that it intersects its circumscribing sphere (Figure 6.3).

Shape of octant	Cubical	Spherical
Constraint interval	$[\, d(A, B),\ d(A, D)\,]$	$[\, d(C, A) - r,\ d(C, A) + r\,]$
Size of interval	Variable, generally smaller	Fixed, maximum of equivalent cubical
Ease of calculation	Complex, geometrical, dependent on orientation	Straightforward, analytical, independent of orientation
Speed of calculation	Variable, dependent on subcube orientation	Fixed, very high, independent of orientation
Accuracy	Exact	Approximate, variable error

Table 6.1 Comparison between cubical and spherical octants

Analytically, this means that the width of the constraint interval, although fixed (since it is equal to the diameter of the circumscribing sphere), is always larger than the variable-sized interval (depending on subcube's orientation), that results from the cubical subcube. This erroneous classification of octants results into wasteful processing. Nevertheless, the error introduced at one level of the octant subdivision is eliminated at a next level where the size of the octants becomes smaller. By using the notation of the previous example, Table 6.1 summarises the differences between the cubical and spherical subcubes.

In the second assumption, we equated the cubelet (that corresponds to a voxel) with a point in space. Effectively, this allows us to determine the 'size of a point' according to the size of the surface and the size of our viewing window, i.e. image plane. The choice of the coordinates of the voxel's equivalent point, namely the *representative point*, may not always correspond to the voxel's centre. This mainly depends on the implementation of the visualisation algorithm used, and the possibly additional needs for anti-aliasing.

Let us consider the following example. Assume that we have to visualise a model that results in a very 'large' but also 'thin' surface. In such a case, it is very likely (depending on the relative size of the image plane) that the size of the voxel (and therefore the cubelet) is too large to be equated with a point and the choice of the coordinates of the representative point is critical for the accurate visualisation of that surface. In plates 47 and 48 we can see the effects of this observation. There (plates 47, 48, on the blue-coloured surface, the

highlights reveal Moiré patterns due to the 'thickness' of the surface. In such a case, we adjust the surface's thickness by introducing a degree of accuracy ε. This is achieved by checking whether the representative point is less than ε units far away from the surface. A good estimate about ε is $\frac{s}{2} \leq \varepsilon \leq \sqrt{3}\frac{s}{2}$, where s is the size of the cubelet. With this method we have achieved a very 'thin' surface on the model of plate 45.

Another way to trace accurately a 'thin' surface is to use more than one representative point for the intersection test. For example, one could use all eight vertices of the cubelet and if they are all found to be on one side of the surface, ignore the corresponding voxel, else proceed with rendering the corresponding pixel. The use of more than one point representative is a very time-consuming process. Furthermore, the determination of what side of the surface a point is in, is not always simple to implement. For these reasons, this second alternative is not recommended.

However, the use of more than one point representative may prove useful when the visualisation algorithm incorporates constructive solid geometry or anti-aliasing techniques. In such cases, the problem becomes obvious when the cubelet intersects the surface at its border therefore, some portion of the cubelet intersects with the model's surface and the rest either with the surrounding background or with another surface.

Although some of this discussion has been presented in the third chapter, we believe it important to outline here three of the most popular alternatives. The first is to use several points that are inside the cubelet but are randomly chosen. After calculating for each of them the colour that results from their intersection test, we can statistically combine them (e.g. average) in order to determine the colour of the corresponding pixel. The second alternative uses points that are chosen with a specific order. The most significant points that are selected first; these are the ones nearer to the observer. In this way the colour of the corresponding pixel depicts the material properties of the surface nearest to the observer. This alternative is therefore, more accurate compared to techniques that use a single representative point. The reason is that what we see is the surface that is nearest to us (i.e. observer). In this way we also avoid the problem of surfaces that overlap each other. The third alternative again uses representative points that are chosen with a certain order; with

regard to their distance from the observer, the nearest is considered first. The difference of this last alternative to the second alternative is that now the voxel is treated as an octant. Therefore *sub-voxels* are generated through voxel subdivisions, up to a predetermined level.

We found the third approach simpler to apply because it makes use of the octree algorithm that we have already implemented. When all possible intersections have been calculated, the colour of the pixel corresponding to the voxel is determined through a statistical combination of the sub-voxel colours.

The determination of the best alternative is not a simple task since it depends on the needs of the application. There are also cases where more than one alternative is used simultaneously for the same voxel or for different voxels of the same image. However, it has to be stated here that it is not the high degree of accuracy than makes an image look more realistic, but the way noise is blended into it [Williams & Collier 1983; Yellot 1983; Cook 1989].

6.3.3 Visualisation issues

Once the description of a surface is encoded into the octree structure, the visualisation of the surface is the final process we need to discuss. Again here the basis of our approach is the traditional octree visualisation as described in previous chapters. Nonetheless, there are two issues that need to be discussed. They originate from the implicit nature of the surfaces we visualise, and relate to the issue of rendering the pixels of the viewport (i.e. shading). During visualisation, in order to be able to apply a shading model, we need to know for a given point on a surface, the direction of the normal to that surface, as chapter three discusses. This normal is going to be a vector, denoted by a triplet in the three dimensional space, starting from that given point, and by convention pointing to the 'outside' of the surface.

Specifically, because there is no analytical definition of the surfaces involved, the determination of the normal vector of such a surface at any point on that surface, both in terms of measure and direction, cannot be computed by any precalculated formula and

therefore it has to be approximated. The problems of determining the normal vector and resolving the ambiguities of the 'inside' and 'outside' of a surface are discussed in the next paragraphs.

The implicit surface as it has been defined in the previous chapter, is described by a constraint. This constraint is an equation where the left hand part is called the *constraint function* and will be denoted by ***cf***, and the right hand part is called the defining parameter and is denoted by ***df***. Their difference, we assume that will form the *model's density function* and will be denoted by $MDF = cf - df$. In this way, a point p belongs to the surface if and only if the model's density function evaluates to zero: $MDF(p) = 0$. Additionally, the normal to that surface at any given point (on the surface) will be defined as the direction that the ***MDF*** function shows the largest change, i.e. is perpendicular to the surface. Therefore, for any given point, the normal N to the surface will be determined by the partial derivatives of the ***MDF*** function along the x-, y- and z- axes.

$$N = \left(\frac{\partial MDF}{\partial x}, \frac{\partial MDF}{\partial y}, \frac{\partial MDF}{\partial z}\right)$$

In our implementation, the partial derivatives will be approximated by assuming that $\partial x = \partial y = \partial z = 2\varepsilon$, where ε is a small number, approximately equal to half the size of a cubelet. Consequently, once a voxel has been identified as a candidate pixel, information about the location of its corresponding cubelet is passed into the shading algorithm. The centre of this cubelet is assumed to be a point on the surface we aim to render.[4] If we denote this point with C, and its displacements of ε units along the positive and negative direction of each coordinate axis (x , y , z) as C_{X^+}, C_{X^-}, C_{Y^+}, C_{Y^-}, C_{Z^+} and C_{Z^-} respectively, the approximation to the normal N at that point C will be:

$$N \approx \left((MDF(C_{X^+}) - MDF(C_{X^-})) , (MDF(C_{Y^+}) - MDF(C_{Y^-})) , (MDF(C_{Z^+}) - MDF(C_{Z^-})) \right)$$

The choice of the magnitude of the displacement ε will have an effect on the smoothness of the image. If, for example, ε is larger than the cubelet's size, the evaluation of the model's density function ***MDF*** for a given point will be affected considerably by the normal to the surface at neighbouring points. This observation is exploited when we need to

[4] If this is not true, we can use an interpolation technique like the successive binary approximation, in order to improve the accuracy of the coordinates of that particular point, thus ensuring that it belongs to the surface.

visualise models that their defining parameter is not continuous. One such example is the stepwise results that emerge from the process of the Mandelbrot set (model class *III*). There, we take ε to be equal to half the size of the cubelet, and a smooth image will be achieved.

If our ε proves to be too small, thus resulting in a zero-length normal vector, then we increase the size of ε and try to estimate the normal again. This rule for temporarily increasing the value of ε will definitely terminate after a small finite number of iterations because of the implicit assumption that the 'potential field' that the model generates should change density values within space. If the field does not change its density values then the modelled iso-surface is either the complete space or the empty set.

Once the vector of the normal has been estimated, we need to know whether its direction is towards the 'inside' or the 'outside' of the surface. But as we have already mentioned, not all the implicit surfaces are closed. Therefore, we need to discriminate our approach accordingly. We have observed that most of the surfaces that belong to model class *IV* are non-closed surfaces, unlike the rest of the classes. This observation is only indicative and from the way we treat the sense of 'inside', one can see that it does not incur any errors on the resulting images.

For the closed surfaces, once the normal vector is approximated, we displace the representative point *C*, by a distance of ε′ units[5] along the direction of the normal. Then, for that displaced point, we evaluate the model's density[6] function and depending on its sign we may invert the normal. It should also be re-iterated here that this is a convention we impose and as long as we are consistent with our conventions we get correct results.

For the non-closed surfaces, there is no meaning of 'inside' or 'outside'. In such a case, once the vector of the normal is approximated, we adjust its direction, so that it will always point towards the image plane. Specifically, in our implementation, where the scene is on the positive *z*- axis and the observer on the negative, we invert the vector of the normal if

[5] We usually assume that ε′ corresponds to the size of a cubelet, and the normal vector is normalized.

[6] Recall that the density function is defined (section 6.3.3) as the difference *constraint_function-defining_function*.

its z- coordinate is positive. In this way we make sure that the parts of the surface that are visible by the observer are properly rendered by the shading algorithm, thus producing all the shadows that are necessary for conveying information about the surface's curvature as plates 25 - 31 show. In the special case of the Voronoi tessellations, the colour of each visible surface is painted according to the identity of the nucleus that is responsible for its creation (i.e. the nearest). We can observe this all the Voronoi plates (34 - 48).

6.3.4 A comparison with the traditional octree

Although the stages of model conversion and surface visualisation are presented in different subsections of this chapter, in our implementation they are combined together. In this way, once a voxel is determined during the model conversion process, information about the location of its corresponding cubelet is passed over to the visualisation algorithm, its normal to the surface is approximated, and finally corresponding pixel is rendered. Then the model conversion process resumes searching for other eligible voxels.

Another deviation from the traditional octree method is that the visibility of a voxel by a particular observer is now tested during the model conversion process. Moreover, this visibility test does not examine voxels only, but octants of any size.

The reason for imposing these changes is twofold. First we reduce the necessary memory requirements since we do not need to store the complete octree structure in the computer's main memory. Second, we accelerate the visualisation of a model since we do not waste time for tracing invisible octants in relation to the current observer.

6.4 Predicting the values of the defining parameter

This section is concerned with the exploitation of knowledge that we can infer about the behaviour of the defining parameter for the purpose of accelerating the visualisation process. We will mainly use the findings of this study in visualising objects of the class *II* category of models (i.e. implicit models where their defining parameter is a function). What we aim

to gain is knowledge about the boundaries of the defining interval, as it is defined within a given volume of space. In this way we anticipate to achieve accurate representations of the models at a reduced calculation overhead, hence rapid response times.

During the octree visualisation, subcubes of different sizes and shapes (according to our first assumption) are used to determine the defining interval for a given model. Additionally, In models of class *II*, the defining parameter is assumed to be a function. Therefore, in order to estimate the defining interval we will need to know the behaviour of this function within the volume of space that specifies the given subcube. One recommendation we made during the relevant sections, was to prefer continuous functions. But continuity is not always sufficient to ensure the accurate computation of the defining interval.

What we propose here, is the estimation of the limits of the defining interval as they can be calculated from the *Lipschitz Condition*. This is a mathematical theorem that can be found in all textbooks of calculus. In the context of computer graphics, this condition has been used by Kay and Kajiya [1986] and Henzen and Barr [1987] for the purposes of speed and accuracy of particular visualisation algorithms. For reasons of clarity we will present the Lipschitz Condition for a simple case of a univariate function $f(x)$ defined along the interval [0 , 1], and we will extrapolate its effect for bivariate functions.

For a continuous function $f(x)$ over the interval [0 , 1], the Lipschitz Condition assumes that there exist two real numbers x_1 and x_2 in the interval [0 , 1], and a real non negative number k, which is called the *Lipschitz constant* that make the following condition true

$$|f(x_1) - f(x_2)| \leq k\,|x_1 - x_2| \quad , \quad x_1, x_2 \in [0, 1] \qquad \text{(Eq. 6.2)}$$

The inequality (Eq. 6.2) uses the Lipschitz constant k to bound the derivative (if it exists) of the function,

$$\left|\frac{df}{dx}\right| \leq k \qquad \text{(Eq. 6.3)}$$

By choosing $x_1 = 0$, $x_2 = 1$, and $x_0 \equiv [0 , 1]$ we can write the inequality (Eq. 6.2) for the pairs of values (x_0 , x_1) and (x_0 , x_2)

$$|f(x_0) - f(x_1 = 0)| \leq k\,|x_0 - 0| \qquad \text{(Eq. 6.4)}$$

$$|f(x_2 = 1) - f(x_0)| \leq k\,|1 - x_0| \qquad \text{(Eq. 6.5)}$$

The addition of (Eq. 6.4) to (Eq. 6.5) produces

$$|f(x_0) - f(x_1 = 0)| + |f(x_2 = 1) - f(x_0)| \leq k\,x_0 + k\,(1 - x_0) = k \qquad \text{(Eq. 6.6)}$$

by substituting $a = |f(x_0) - f(x_1 = 0)|$ and $b = |f(x_2 = 1) - f(x_0)|$ we can bound the value of the function f within the perimeter of an ellipse that has its focal points at $f(0)$ and $f(1)$ as the revised (Eq. 6.6) shows:

$$a + b \leq k = \textit{constant}$$

Assuming that we can map the interval [0,1] to any interval of real numbers, say $[x_{min}, x_{max}]$, in order to exploit the Lipschitz Condition we need to calculate the constant k. This is determined from inequality (Eq. 6.3) by calculating the *global maximum* of the derivative of the function f.

For the bivariate function $g(u, v)$, the Lipschitz constant will be calculated from the partial derivatives of g

$$k \geq \max_{0 \leq u,v \leq 1} \left[\left| \frac{\partial g(u,v)}{\partial u} \right| + \left| \frac{\partial g(u,v)}{\partial v} \right| \right]$$

If we cannot determine the value of the bound k, we may estimate it by sampling the slope of the defining function and using the maximum of the sampled derivatives.

To demonstrate the potential of this method and illustrate its use on the visualisation approach we proposed, we will present the following example. Assume that we have a model of class *II* in three-dimensional space where, the defining parameter is a function. We assume also that this function is univariate and uses as input parameter the x- coordinate[7] of the point we use to assess the point membership classification test that this model defines. Such a function may be the:

$$\delta(x) = x^2 - 7x + 6 \qquad \text{(Eq. 6.7)}$$

[7] It does not damage the generalisation of this technique to assume the use of the point's co-ordinate instead of any other intermediate mapping as the input parameter for the defining function.

During visualisation of this model, we have to test whether an octant intersects with any parts of the model's surface. As we have already explained, this test is possible by intersecting the relevant constraint and defining intervals. For the calculation of the constraint interval we resorted to spherical octants. However, the calculation of the defining interval depends on the exact function that makes the defining parameter δ (Eq. 6.7).

Let us assume that the centre of the spherical voxel is $p = (x_p , y_p , z_p) = (4 , 5 , 8)$ and its radius is $r = 2$. The limits of the defining interval will be determined by the use of the Lipschitz Condition. In this way we will calculate the boundaries of the defining function δ when its input parameter is allowed to take any value in the interval $[x_l, x_r]$. The interval of allowable values for the input parameter of the defining function depends on the shape of the octant and in the case of spherical octants is assumed to be $[x_p - r , x_p + r]$. Therefore in our example the input parameter for the defining function takes values in $[x_l, x_r] \equiv [2, 6]$.

To use the Lipschitz Condition, however, we must use the input interval $[0 , 1]$. Therefore, the first step we have to make, is to transform the defining function of equation (Eq. 6.7) to map the interval $[2, 6]$ to the required interval $[0 , 1]$. This mapping will be computed by using the new variable x' of the transformed defining function δ' as:

$$x' = \frac{x-(x_p-r)}{(x_p+r)-(x_p-r)} = \frac{x-2}{6-2}$$

Consequently, $$x = 4x' + 2$$

and the defining function will be transformed accordingly:

$$\delta'(x') = 16x'^2 - 12x' - 4$$

In this way when for the defining function $\delta(x)$ the variable $x \in [2, 4]$, the transformed defining function $\delta'(x')$ will imply $x' \in [0, 1]$. Similarly, the middle-point x_p' will be mapped to the middle-point 0.5 of the [0 , 1] interval (linear transformation).

We can therefore apply the Lipschitz Condition, provided that we can estimate the constant k which is the maximum absolute value of the derivative of the transformed defining

function:

$$\left|\frac{d\delta'}{dx'}\right| \leq k$$

The derivative of δ' is $32x' - 12$ and it belongs to [-12 , +20] when $x' \in [0 , 1]$. Therefore we can assign $k = 20$.

The Lipschitz Condition therefore will suggest that:

$$|\delta'(x_p') - \delta'(x')| \leq k|x_p' - x'|, \; x' \in [0, 1] \equiv [x_l', x_r']$$

By replacing the appropriate values, taking into account the mapping of the defining function,

$$|\delta'(x_p') - \delta'(x')| = |\delta(x_p) - \delta(x)| \leq k|x_p' - x'| = k|0.5 - x'| = 20 \times 0.5 = 10, \; x' \in [0, 1]$$

We can safely assume therefore that the defining interval for this example situation is

$$[\delta(4) - 10, \; \delta(4) + 10] \equiv [-6 - 10, \; -6 + 10] \equiv [-16, +6]$$

A better refinement to this method is achieved if we split the interval $[x_l , x_r]$ in two intervals; the $[x_l , x_p]$ and the $[x_p , x_r]$. For each of them, we should transform the defining function to map the [0 , 1] and determine its boundaries. The required defining interval will then be the union of these two boundaries.

6.5 Conclusion

In this chapter we discussed how we can amend the octree method in order to create a visualisation approach that is suitable for the models we have constructed. Developing therefore an appropriate visualisation approach was necessary to ensure that the modelling approach we proposed in the previous chapter (five) can be used to its full potential. Moreover, because of the implicit nature of the models we use, we found it very important to introduce some techniques that were necessary for the acceleration of this visualisation approach. Of them, some were aiming at altering the specifics of the original octree approach, while others were aiming at studying the behaviour of the model, thus exploiting the spatial coherence of the scene.

Our intention in the next chapter will be to provide some criticisms regarding both the modelling and the visualisation approach that we proposed. We discuss two complementary issues. The first is concerned with the actual application of the modelling and visualisation approaches developed in the research described in this dissertation. The second is concerned with further research directions aiming at enhancing the methods we developed.

In this way, our objective is to avoid confining our modelling approach to the particular visualisation approach presented here. For this reason, in the next chapter we also outline the necessary algorithms for using our proposed models with a number of other visualisation techniques that are currently being used extensively in the literature. Such effort will allow us to direct to further research in order to exploit the capabilities of the modelling approach we have developed.

Chapter 7 Research considerations and directions

7.1 Introduction

In this chapter we will adopt a critical perspective from which we will assess the usefulness of the modelling approach that we propose in this dissertation. Issues that have arisen while using this approach will be discussed in section 7.2 where we will be concerned with the ease of modelling particular scenes, the accuracy of the visualised surfaces, speed of our visualisation approach, and precision problems encountered.

Our criticisms will then be constructive and we will focus our interest in presenting several methods for improving the usability of the modelling approach we propose. These methods reflect our considerations regarding the future of this research. We wish to continue our research in three different directions: towards further enhancements to the modelling approach we propose, towards alternative means for describing our models and finally towards alternative visualisation approaches. Every method we consider in this chapter is outlined, and where applicable references are given to particular techniques of significant relevance. Moreover, when appropriate we demonstrate the principles of the outlined method with some simple examples.

Specifically, we begin our presentation with criticisms regarding the utilisation of the modelling and visualisation approaches that we propose (section 7.2). Then we review the issue of enhancing the modelling approach that we proposed by assessing the challenges involved in two aspects: the visualisation of implicit models in the space of four dimensions (section 7.3) and the introduction of non-linear combinations of the measure of distance (section 7.4). Next, in section 7.5, we focus on alternative model descriptions where we outline two methods for determining a polygonal mesh that approximates to the surfaces we constructed. We present both a method for calculating a triangulation as well as a mesh of tetrahedra that approximate to the modelled surface.

Finally, we present two distinctly different visualisation approaches; ray tracing (section 7.6) and what we call *stochastic visualisation* (section 7.7). With regard to ray tracing we discuss a global illumination model as well as the 'Heidelberg model'. The concluding paragraphs (section 7.8) are concerned with the provision of a brief evaluation and summary of the whole of this dissertation with the anticipation of having fulfilled the role we set out in the beginning.

7.2 Criticisms

There are several issues against which a modelling approach may be contrasted. In the second chapter we presented the most significant ones. We have already commented on some of these issues in chapters five and six, when discussing our modelling approach. However, we believe that a few outstanding issues still need special treatment, as they seem to be the most frequently discussed in the literature. These are *ease of modelling*, *accuracy and numerical precision* of the necessary calculations, and *speed of visualisation.*

7.2.1 Ease of modelling

Ease of modelling is concerned with the usability of the modelling approach in question. Regarding our approach, it is understood that it is not possible to have feedback from users (designers). Therefore, our criticisms have been gathered solely from our own experience in using this approach.

Apart from our particular approach, however, we have also used several other modelling approaches thus enabling us to provide an accurate and comparative judgement. In particular, in our installation we also manipulate models based on the polygonal mesh and the analytical approach. We also visualise models that are based on a variety of splines (B, 6, NURBS) and some procedural methods such as extrusion, revolution, and other pseudo-random and fractal processes. Table 7.1 summarises the most important differences of the major modelling approaches.

	Polygonal mesh	Analytical	Our modelling
Number of primitives	1 (polygon)	Few (formuli)	Any object
Basic building mechanism	Patchwork	Set union	Model dependent
Other construction mechanisms	None inherent, need a pre-processing stage	If not in a formula, use blends, CSG	Inherent in the model (rotation, extrusion...), CSG
File sizes of model descriptions	Huge, depending on required precision	Small, depending on formuli parameters	Small, depending on participating primitives
Ease of model creation	If not hardware assisted, very tedious	Presupposes knowledge of the primitives	Simple, intuitive, generic
Alternative coding forms	None	Symbolic	Symbolic

Table 7.1 Evaluating ease of modelling

Each of the approaches has its own merits and is particularly good for a limited variety of objects that it can describe with minimal effort. But in spite of this, our approach has proved capable of describing a vast variety of objects, as we have already shown in chapter five. Moreover, it provides a direct and explicit way for specifying a model, and it allows a high degree of uniformity for the necessary model descriptions. Take, for example, the modelling of a body of revolution. With the polygonal mesh, we would first need to specify the contour which will have to be rotated and the axis of rotation. Then we will have to apply an intermediate process for the generation of the appropriate model description. With our approach, however, we simply describe the axis of rotation as a primitive member of one collection, and the contour that needs to be rotated as the defining function. In this respect therefore, we would attribute merit to our own approach.

Another aspect for comparing modelling approaches (regarding ease of applying them) is the size of the files that contain the model descriptions. Here again we believe that the modelling approach we propose ranks very high. Consider the same example of the model description of a body of revolution. With our approach, the resulting file consists of a few bytes (depending on the description of the defining function contour). With the polygonal mesh, the model description would initially be the same as ours (the axis and the contour),

but this description will become the input of the intermediate algorithm which would generate a large file of several kilobytes (1Kb = 1024 bytes), depending on the number of angular slices, of the final model description.

This difference in the model's file size, of at least two orders of magnitude, proves critical on models of computer graphics scenes with several such defined objects. Using the analytical approach, model descriptions may have necessitated a much smaller file size, but the functions that describe bodies of revolution are usually too complex to determine and depend on the specifics of the required contour and axis of rotation of the particular object.

One challenge that our modelling approach poses is the description of analytic functions in the model description. In class *II* models, for example, where the defining parameter is a function, we need to describe it in a computer readable manner. This is actually a difficulty that is shared with the analytic modelling approach. We can see two general methods to tackling this issue. The first allows the designer to use a pre-determined set of functions by simply denoting them with their appropriate identifier. The second method, which provides an 'intelligent' alternative to mapping pre-specified functions, is the use of a symbol-parser, that would allow the designer to describe any function using a pre-determined set of symbols and a syntax. One such example is the parsing facility of the Mathematica [Wolfram 1991] software, where the user denotes virtually any function using a pre-determined language (symbols and syntax).

The first method, which is the one that we adopted, is limited to the type of functions that have already been incorporated in the visualisation algorithm. In this respect, it does not allow for any flexibility but optimizations for manipulating the pre-defined functions may speed up the whole process of visualisation considerably. The second method is very flexible, so in most implementations it does not provide any optimizations because of the variety of functions it allows. Despite the enormous flexibility that the second method provides, it is very rarely implemented because of the programming effort it necessitates.

An approach that provides some flexibility but also permits certain optimizations, is a combination of the above two methods, where generic parameterized categories of functions are only permitted in the model description. Such an approach would allow for, say, polynomials of up to the sixth degree, and the model description should only specify the values of all seven parameters which are the coefficients of the polynomial's variable.

For piece-wise defined functions, the task of model description is very complex and we are inclined to impose certain restrictions on the total number of pieces that a particular function may consist of. Nevertheless, these limits are chosen arbitrarily and serve the purpose of coding simplicity. Additional tests regarding the geometrical continuity of the piece-wise functions are also imposed in order to enable the application of bounding volume information by the Lipschitz Condition [Herzen & Barr 1987].

7.2.2 Precision and accuracy

The next issue for criticisms is the accuracy and numerical precision of the calculations. This is mainly an issue of the visualisation approach, but certain aspects of it are inherited from the nature of the models we use. For example, in models of the *IV* class certain surface areas may be totally ignored by the visualisation algorithm, because the representative points of the relevant cubelets (or even subcubes) may fail the intersection test.

Specifically, precision errors are introduced when visualising models of any class. These errors are introduced in the calculation of both the constraint and the defining interval. But models of class *I* have a constant defining interval, and for models of class *II* and *III* the defining interval is usually straightforward and simple to compute, thence, a small scale of accuracy problems are encountered in the intersection test. However, models of the *IV* class predicate the computation of a defining interval which has to be determined with an amount of calculation that is comparable to those of the constraint interval. In this way, precision errors are introduced in both the constraint and the defining interval, thus making the intersection test inaccurate.

To eliminate these accuracy problems of the intersection test, we distinguish between the application of this test on subcubes and its application on cubelets. The first case (subcubes) is solved by the extension of the constraint and defining interval. Usually both ends of the intervals (minimum and maximum) are extended a small proportion, say 1% of the total length of the interval. Therefore the likelihood of missing out areas of the surface due to incorrect intersection tests is minimized. It follows, however, that we may have to characterise incorrectly some subcubes as heterogenous, thus wasting computer resources.

The second case, where the intersection test is applied on cubelets, is more difficult to solve since we have reached the final level of subcube subdivision. We can see two different techniques here for eliminating accuracy errors. The first extends the constraint and the defining interval and permits one more level of subdivision, thus providing a unified approach to eliminating the accuracy problem. The second technique is more strict and through additional verification tests, where applicable, assumes that the resulting intervals are accurate. These supplementary verification tests include geometrical confirmations, as implied from the computed intervals, as well as approximation methods for adjusting the detected discrepancies.

7.2.3 Speed of visualisation

The last issue that we will criticise in this section, is the speed of visualisation of the proposed modelling approach. From results in the literature regarding other modelling and visualisation approaches, and those we have collected from our own experiments, we can observe that the visualisation speed of our models is not very promising. Although it is not desirable, such poor visualisation speeds should have been expected mainly because of the high priority that we have placed on the accuracy of the represented objects.

An additional factor for obtaining such results is the use of code that has not been optimized to any great extent. This is a serious and time-consuming task, demanding the use of supplementary software tools such as profilers, which we feel is not within the scope of our experiments. Furthermore, we use common purpose computer hardware which places us in a disadvantage when compared with customised, special purpose, expensive installations.

The main reason for our choice is to achieve portable code that can be executed on virtually any hardware platform, as the first chapter clarifies.

Despite these criticisms, as we shall see in the next sections, we also suggest a number of alternative approaches for visualisation. The most interesting simulates a global illumination model and enables the generation of 'photo-realistic' images (section 7.6). At the same section we also outline a simple version of ray tracing, called ray casting, as it is implemented with the 'Heidelberg model'. Moreover, we also present (section 7.7) a very simple visualisation approach that is based on the nature of the point membership classification test. Nonetheless, we also suggest a way (section 7.5) for constructing a polygonal mesh out of the models we propose, thus indirectly enabling the use of the simple but extremely efficient polygonal mesh visualisation approach.

7.3 Four-dimensional space

In this section we will discuss the means for visualising surfaces that have been defined in four-dimensional space. This is not a new issue in computer graphics since such approaches have already appeared in the relevant literature [Banchoff 1990; Hanson & Heng 1992]. The challenge in the fourth, or higher, dimensions is the difficulty in conceptualising these spaces. An excellent aid to this challenge is Banchoff's [1990] book that uses artistic as well as computer graphics images to examine cross-sections (projections) of objects of the four-dimensional space. He also provides image sequences of the same four-dimensional object, in order to explore the choice of the viewpoint and the projection used for its visualisation.

In order to understand better the issue of projection, let us consider the following scene where a hyper-cube has edge size of two units and the coordinates of its vertices are determined by the permutations of the (± 1 , ± 1 , ± 1 , ± 1), along a four-dimensional orthogonal Cartesian coordinate system. Our aim is to establish a projection function that would map this hyper-cube onto the window of the two-dimensional viewplane.

In order to achieve this mapping we have a number of different choices. The first and simplest one is to ignore two out of the four coordinates of every four-dimensional vertex. In this way we achieve an orthographic projection which in the case of the hyper-cube would generate a square of size two, since it is determined by the vertices (± 1 , ± 1). For more complex four-dimensional objects, the choice of which of the two coordinates we ignore is crucial, because it will expose or hide several details of the object's hyper-surface.

Another projection we may also use is perspective, which, as we discussed in the third chapter, generates 'intuitive' results. From the four-dimensional space to the three-dimensional, assuming applicability of the Pythagorean theorem, the perspective function is a simple extension to the one used in chapter three. For a four-dimensional vertex p with coordinates $p = (x,y,z,\omega)$, its projection on the *perspective cube* which is at a distance $-d$ away from the observer along the fourth dimension $\omega = -d$, will become the vertex $p' = (x \times d/\omega , y \times d/\omega , z \times d/\omega , -d)$. This vertex is the three-dimensional point $(x \times d/\omega , y \times d/\omega , z \times d/\omega)$ in the perspective cube and represents the projection of the four-dimensional vertex p onto this perspective cube.

The perspective projection presented here does not solve our mapping problem completely because we eventually need to reach the two-dimensional space of the viewplane, and not the three-dimensional perspective cube. Moreover, we are not familiar with the conceptualization of the four dimensions, therefore we can experiment with different projection sequences. For example, we may use the perspective to get from four dimensions to three (of the perspective cube) and then use any depth sorting method such as the z-buffer to (orthographically) project the surfaces in the perspective cube on the two-dimensional window on the viewplane.

Another issue that we have to address is the shading model that we may use. This, as Hanson and Heng [1992] suggest, may be derived by extending the three-dimensional shading models. Our modelling approach gives us another way to achieve rendering because we manipulate objects as iso-hyper-surfaces. This is the approximation of the normal to the hyper-surfaces with the partial derivatives of the constraint and defining functions. In this way the fourth dimension is introduced with an amazingly straightforward way. Another

alternative would be to project the surfaces in three dimensions first, and then illuminate their projection using available rendering technology.

Work in visualizing the fourth dimension has been restricted because of our inability to conceptualize the four-dimensional space. Therefore, we would recommend that initially such four-dimensional visualisation approaches are implemented to visualize simple geometrical objects in order to familiarize the user with the effects of projections and the four-dimensional shading. An object very simple to conceptualise and well discussed in the literature is the hyper-cube. It is produced by the extrusion of a cube (the envelope produced by shifting) along a direction that is perpendicular to all the cube's edges. This direction will form the fourth dimension. The hyper-cube has therefore 16 vertices and 32 edges.

7.4 Non-linear propagation

So far we have assumed a linear propagation of the defining functions for all the surfaces we have modelled. In this section we will discuss the challenges of using functions of the measure of distance that propagate on a non-linear relationship. For example, we will use the inverse of the measure of distance, any exponential power of it, or even trigonometric functions of appropriately transformed distance measures.

In this section we will identify the assumptions that we have made in chapter six that are no longer true, or need additional adjustments. Then, we will distinguish the segments of the octree visualisation algorithm that need to be altered, as a consequence of the change in our basic assumptions. Finally, we will provide some general directions towards achieving these changes.

The most significant change is on the first of the two assumptions of chapter six. We calculated the maximum and minimum values of the constraint and the defining interval by adding or subtracting the radius of the subcube's circumscribing sphere to the value of the constraint and defining function evaluated at the sphere's centre.

In this way we used an implicitly made assumption that both the constraint and the defining functions are linear combinations of the measure of distance. For the defining function, we then made this assumption explicit and we used different combinations of the distance function as well as other pseudo-random processes.

By admitting that the measure of distance is included in non-linear relationships for the calculation of the constraint function, we have to use the same precautions that we identified in chapter five for the effective manipulation of the defining function. Consequently, we can no longer use our first assumption, but we have to approach the constraint function according to the particular combinations (of the measure of distance).

For example, when we use the *sin( )* function for determining the measure of distance, in the models shown in plates 18 - 20, we can conclude that we will only get values between [-1,1]. By shifting this interval one unit to the positive direction of real numbers, in order to avoid negative values, we can eventually have distance values between [0, 2]. As a result, we can use this observation to bound our constraint interval between [0,2].

Unfortunately, although such examples may prove very effective, we cannot generalise their applicability. There is no generic technique that would enable the accurate estimation of the constraint interval within the bounds of a subcube or its circumscribing sphere. The most suitable technique we can suggest is the exploitation of the Lipschitz Condition out of which we can derive the means to bound any specific constraint function. The Lipschitz Condition has already been presented in chapter six, where we discussed its proof but also the difficulties of its exploitation.

A more interesting case that has great applicability in physics is the propagation of fields that are described with the function of the inverse square of the measure of distance. The law of Newton about attraction forces between any two real objects, and a similar law between any two electrostatic loads that explain several phenomena in nature, are described with the function of the inverse square.

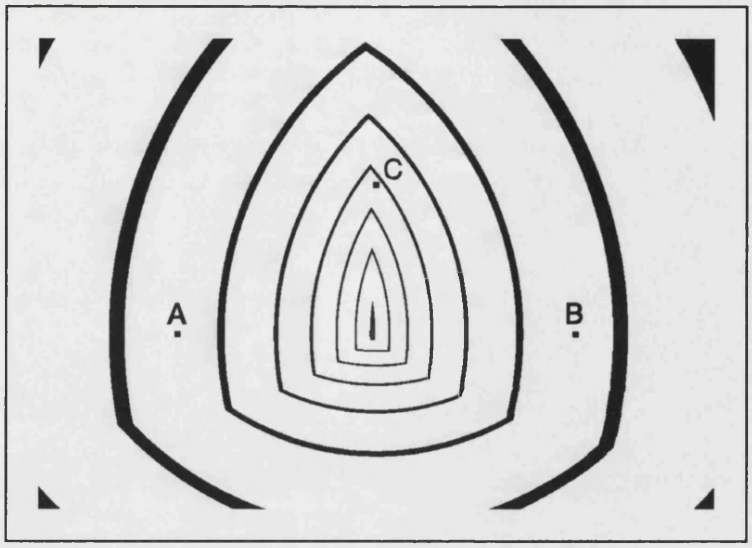


Figure 7.1 Using non-linear distance measures

In Figure 7.1 we see iso-density contours of the field generated by the following model

$$\{\, p \mid p \in \mathbb{R}^2 \,,\, d(p,C) = \delta \,\}$$

where C is a collection of three points and $d(p,C)$ denotes the weighted inverse square Euclidean distance from point p, to the collection C of three points, for a given weight vector $w(\ )$:

$$d(p,C) = \frac{w_i}{(d'(p,C))^2} \,, \quad d'(p,C) = \min_{i=1,3}(\|p - x_i\| \,, x_i \in C)$$

The next figure (Figure 7.2), shows the Voronoi tessellation when we treat these three points, x_1, x_2, x_3, as nuclei. This image was produced by the following model:

$$\{\, p \mid p \in \mathbb{R}^2 \,,\, d_1(p,C) = d_2(p,C) \,\}$$

where C is a collection of three points and d_1, d_2 denote the weighted inverse square Euclidean distance from the nearest and second nearest point in C from p for a given weight vector $w(\ )$:

$$d_1(p,C) = \frac{w_i}{(d_1'(p,C))^2} \,, \quad d_1'(p,C) = \min_{i=1,3}(\|p - x_i\| \,, x_i \in C)$$

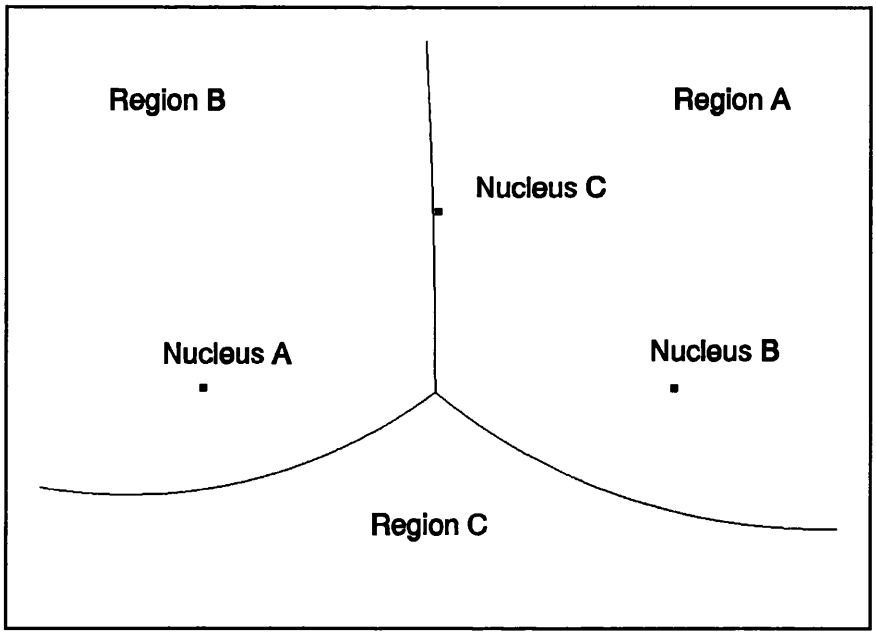


Figure 7.2 A Voronoi tessellation using non-linear distances

Observe here, that since we are inverting the measure of distance, the regions of minimal distance are the ones far away from the corresponding nuclei. Therefore, the classical definition of the Voronoi tessellation, as we presented it in chapter four, is no longer directly applicable. Consequently, we have to resort to our definition of (Eq. 5.9).

7.5 Polygonisation of surfaces

In general, we are against the approximation of surfaces. However, one utility they offer, which we would advocate, is the real time manipulation of models, that can only be achieved on polygonal meshes in conjunction with special purpose hardware.

The subject of polygonisation, and the special case of *triangulation*, has appeared in the literature on numerous occasions in various computer graphics applications. This subject is very popular since it provides methods that generate a polygonal mesh to approximate to any given surface. More specifically, these methods convert a model description of almost any modelling approach in a polygonal mesh model. Under this perspective, polygonisation methods can be seen as an intermediate stage between modelling and visualisation.

Following the classification schema of modelling approaches, as we presented in chapter two, we have to consider all these methods as part of the modelling process and not as part of the visualisation. We will be concerned with two polygonisation methods here. The first, produces a mesh of triangles and the second a mesh of tetrahedra out of which we may also build a triangulation.

7.5.1 Surface triangulation

Triangles are the simplest form of polygons that can be used to approximate to any surface. They are by definition planar since they are determined by three vertices only, they demand data structures of minimal size for storing them in computer memory, and there exist several hardware implementations of shading models especially developed to handle triangles (Gouraud, Phong shading).

The triangulation process we prefer to present is outlined in Wallin [1991] and is especially developed for the manipulation of volumetric data in medical computer graphics applications. The method is divided in two stages. The first stage identifies and gathers surface points that would become vertices, and the second stage assembles these vertices into polygons that construct the required mesh.

In the first stage, we assume a lattice of cubes to enclose the whole scene. The faces of all these cubes are tested for intersection with the required iso-surface. The aim is to determine points on the cubes' faces that belong to the iso-surface. In order to detect possible intersections, we apply the following test. For all the faces of every cube, in the three-dimensional lattice, we evaluate the constraint function at the corners and the centre point. If the results of this evaluation include values larger as well as smaller than the values of the defining function at the same points, then we can assume that the iso-surface intersects with this particular facet. This conclusion is a consequence of the application of a theorem from Calculus, the Bolzano theorem, which we applied on the iso-surface.

The Bolzano theorem assumes a real-valued function $f(x)$ which is continuous on the interval $[a, b]$ and states that if $f(a) \times f(b) < 0$ then there exists a point $c \in [a, b]$ that $f(c) = 0$.

To ensure applicability of this theorem, we must assume that the iso-surface is continuous, which is not always correct. When such discontinuities exist, we apply the theorem on the constituent surface patches, and we apply additional intersection tests at the borders of these patches. Once intersection has been detected, we then determine the intersection points with an interpolation method such as the Regula-Falsi, or the successive binary approximation.

The second stage then connects 'intelligently' all the vertices together, in order to produce the required mesh of polygons. The polygons produced with this method may have from 3 to 12 vertices according to Wallin's [1991] study. All these polygons have their vertices (and edges) on the faces of the cubes, therefore the resulting polygons may not be planar. Consequently, a subsequent triangulation stage is necessary to ensure planarity of the generated facets.

Triangulation occurs by decomposing the polygons into triangles, and there are several techniques to achieve such a decomposition. Wallin chose to connect each pair of consecutive vertices with the centre of gravity of the initial polygon. In this way he ensures that the resulting polygons do not coincide with the faces of the cubical lattice thus avoiding degeneracies.

Although this method generates accurate mesh representations of iso-surfaces, we must stress that it lacks a unified mathematical background and is best suited in the domain of medical imaging. For this reason we will present a second polygonisation approach, that is based on the Delaunay triangulation.

7.5.2 Solid tetrahedra-isation

This method has been outlined by Baker [1989] and is especially suitable for polygonising closed solid objects. The generated model is a mesh of tetrahedra that determine the volume of space occupied by a closed surface. With certain assumptions, we can modify this method to generate meshes of not necessarily closed surfaces.

This method is based on the problem of the Delaunay triangulation. This problem is best expressed in the two–dimensional space where it is concerned with the determination of a mesh of triangles out of a set of given points, the nuclei. The required triangles must be formed in such a way that they have their vertices on the nuclei, and when we draw their circumscribing circles they do not contain any other nuclei but only the ones used to determine these circles.

Bowyer [1981] describes an algorithm to determine these triangulations in the two-dimensional space. Moreover, Baker [1989], and Vassberg and Dailey [1990] describe an extension to the Delaunay triangulation problem in three dimensions. The re-defined Delaunay triangulation problem states that for a given set of points (nuclei) we must determine a set of tetrahedra that have their vertices on the nuclei, and their circumscribing spheres do not contain any other nuclei but the ones used to determine these spheres.

For our three-dimensional surfaces, we may use this re-defined Delaunay 'tetrahedra-isation', the algorithm of which is outlined here. We start with six appropriately positioned tetrahedra that are assumed to cover completely the volume of space that surrounds the surfaces we wish to polygonise. Then, we start generating points using the octree visualisation approach, as defined in chapter six. Every point that we determine is used as a nucleus. In this way, the previously determined 'tetrahedra-isation' no longer complies with Delaunay's constraint (that circumscribing spheres do not cover any nuclei) and therefore we solve again the Delaunay problem by re-arranging the vertices of neighbouring tetrahedra.

The process is repeated for every point that we find to belong to the modelled surface. We are always certain that the introduction of a new nucleus would destroy the previously determined 'tetrahedra-isation' imposed by the Delaunay constraint, because every new nucleus is enclosed by the initial set of six tetrahedra which were assumed completely to surround the required surfaces. Therefore it must belong to one of the circumscribing spheres.

The process ends when we no longer wish to introduce new nuclei. Better defined termination criteria are what Baker calls *quality criteria* on tetrahedral elements. These introduce a metric based on the size and shape of the tetrahedra used. If some tetrahedra do not conform with the pre-defined measures, the process of 'tetrahedra-isation' must continue.

The generated mesh of tetrahedra, from the process we have outlined here, covers the volume of space that is enclosed by the surface of a closed object. Moreover, the mesh consists of tetrahedra the vertices of which, may nor belong to the same 'side' of an object. This problem was identified by Baker and he suggested the classification of the generated tetrahedra in three categories depending whether: a tetrahedron has one or more points inside the object, belongs totally to one piece of the object, or belonging to the interface between two discontinued surface patches covering the object. In this way, we attain better control over the generated mesh.

The 'tetrahedra-isation' algorithm presented here generates meshes that describe closed surfaces only. However, we can adjust this process slightly in order to describe any surface. What we have to do is to assign a certain 'thickness' on the surfaces that we want to polygonise. In this way, the algorithm would generate a mesh of tetrahedra that would construct a 'thick' structure between the 'sides' of the surface. The sides of the tetrahedra that do not belong to the 'inside' of the thick surface may then be identified and collected in order to provide a polygonised description of the surface.

The main negative concern for this algorithm is that it is extremely time consuming. Vassberg and Dailey [1990] used information from Baker [1986] to benchmark different versions of this algorithm which originally demanded 8 hours of CPU time on a CRAY X-MP supercomputer. The best optimised implementation of this algorithm was presented in their paper [1990] where they used 24 minutes on the same supercomputer. The model they used for their benchmarks was the description of a BOEING 747-200 with 12,038 nuclei and generated almost 58,000 tetrahedra. This amazing improvement in performance was achieved by appropriately sorting the generated tetrahedra. The sorting was based on information about the relative location of the tetrahedra and was implemented with an octree data structure, similar to the one we use in our visualisation approach of chapter six.

With regard to polygonising surfaces we must reiterate that there exist many techniques. We presented only two, Wallin's and Baker's, because we believe they are suitable for the modelling but they also exploit the particulars of the visualisation approach we adopted.

Another approach to visualising our implicit models would stem from the introduction of a global illumination model and the principles of ray tracing. Therefore, in the next section we will consider the problem of intersecting a ray with an implicit surface in the context of ray tracing. The determination of appropriate intersection points will become the main concern of the next section.

7.6 Ray tracing implicit models

In this section we will explain how we may use the ray tracing visualisation approach on the models we propose. We can see two distinctly different approaches here that are of interest. The first uses the 'Heidelberg' ray tracing model [Meinzer *et al.* 1991], and the second uses the classical ray tracing approach as presented in chapter three.

7.6.1 The 'Heidelberg' ray tracing model

The visualisation approach is an example of faking ray tracing but is straight-forward to implement and produces noteworthy results. The method uses a shading model that defines light absorption as a measure that is proportional to the *density* of the visualised field. In this way, the value of the light intensity is diminished proportionally to the density of the field it passes through. Similarly, light reflection is modelled as intensity value which is inversely proportional to the density of the model's field. In other words, light reflection is more likely to occur when crossing space that exhibits a high gradient of density values. Light transmission is also modelled as a reduction in the light intensity which is proportional to the value of the density of field that the transmitted light passes through. Another optical phenomenon, *scattering*, which is the amount of incident light scattered towards the observer is also simulated.

For the purposes of this shading model the effects of the optical phenomena (transmission, reflection, scatter) are not treated as inter-related but as totally independent. In this way, light transmission will only calculate light that passes directly towards the observer and will not take into account any other light that has been possibly reflected off any other surface. The reason is to ensure quick visualisation of required model.

This shading model also assumes that there exist only two light sources, the intensity and location of which cannot be altered by the user. One source is assumed to coincide with the observer, and the other is located at exactly 45° to the left of the observer at the same horizontal plane as the observer. For each light source, a subset of the optical phenomena is implemented. In particular, for the first light source only diffuse reflection is used, whereas for the second, scatter, specular and diffuse reflection are all simulated. These simplifications have been introduced in order to speed up the visualisation process. Nonetheless, the image is well illuminated, since the 45° positioned source introduces highlights and shadows, and the source coincidental to the observer illuminates the dark areas that the first missed out.

Because of this separation of the effects of the two light sources, the image of a model is generated by the weighted sum (superposition) of the images produced by the independent application of the shading model as it is determined by each of the light sources. By adjusting the weights during the image addition, the quality (e.g. contrast) of the final image may be controlled by the user.

The complete visualisation algorithm starts off with the assumption that we can impose a canonical, orthogonal, three-dimensional grid of cubes that covers the required surfaces. Each such cube may be the equivalent of the cubelets we used in the octree visualisation approach of chapter six.

For each such cubelet, we have to determine the value of the field's density function, which is a combination of the values of the constraint and defining functions, evaluated at the centre of the cubelet. This mapping from the constraint and defining functions, to the density function, will be used to effect the shading model in two complementary ways. Firstly, it

will be used to determine the attenuation of the light as it passes through the cubelets. Secondly, it will be used to detect the boundaries and the normal of the required surface. Therefore, the combination of the defining and the constraint functions into a density function need some consideration. Usually, the difference between the constraint and the defining functions will make the density function on the required surface to evaluate as zero. This was actually our initial definition of the surface:

$$constraint\_function = defining\_function$$

Knowing where the surface is, in terms of values of the density function, enables us to define the vicinity of the surface as an interval of density values around zero[1]. In this way, density values away from this interval are of no interest to the algorithm and may be safely considered as noise and completely ignored from the calculations of the shading model. The density value on the centre of every cubelet is then used as a representative density value for the shading model.

Rays emanate from the observer towards the scene. Both the orthographic and the perspective type of projection may be used. The orthographic projection is easier to implement since it presupposes rays that are aligned with the orientation of the cubelets. With either type of projection, a ray is assumed to penetrate the cubelets and the intensity of light that it carries through is diminished according to the density of the cubelets. When this ray encounters the surface (density value zero) reflection is simulated. In a similar way, the light that illuminates the sources is also attenuated as it passes through the cubelets.

This visualisation approach is simple to implement but it does not simulate reflections of surfaces onto other surfaces. The difficulty one may encounter is the appropriate calculations of the density function and the subsequent calculations of the shading model. The authors of this approach [Meinzer *et al.* 1991] suggest the use of linear transformations because of their simplicity.

[1] Recall (section 6.3.3) that the density function is the difference *constraint_function-defining_function.*

Another issue is the resolution of the grid of cubelets. We suggest that we use a high resolution and evaluate their density function as needed. An alternative will be to use the octree visualisation approach to characterise all cubelets that belong to the vicinity of the surface and totally ignore the rest of the cubelets since they will have a density value beyond the limits we imposed in the previous paragraphs.

Compared to the octree visualisation approach, this approach is equally time consuming. However when several views of the same surface are required, it may prove to be faster since we only need to characterise the cubelets once, and then we simply traverse the cubelets according to the direction of a ray.

The Heidelberg ray tracing model does not offer the optical effects of a global illumination model, but gives us a quick visualisation approach of a quality similar to the octree. The next approach that we propose, however, will be addressing the issue of global illumination disregarding, to a reasonable extent, the speed of the visualisation.

7.6.2 A global illumination model

This approach introduces ray tracing of implicit surfaces but unlike the Heidelberg approach, uses a global illumination model. As we have already presented in chapter three, the classical ray tracing assumes the *eye ray* which emanates from the observer and is directed towards the scene. When this eye ray intersects with a surface of the scene, we apply the chosen shading model and then follow the generated child rays, accounting for reflection and refraction. Additionally, for the determination of the intensity of the incident light, we also try to establish, with the *shadow feelers*, whether this intersection point is in the shadow of some other surfaces, or it can be directly 'seen' by the light sources of the scene. Once the colour contribution of this intersection point is determined we follow all the generated child rays. Then, when these new rays intersect with surfaces in the scene, we apply the same shading model and follow the newly generated rays recursively until we reach some termination criteria.

This process, that describes the fundamental concept of ray tracing, is based on our ability to determine the intersection between a ray[2] and surfaces of objects in the scene. Using analytical models, the intersection problem may be well defined and its solution is either exact or approximated depending on the nature of the equations involved. The reader is referred to chapter three for a more detailed discussion. However, with the models that we propose, the intersection problem cannot be described analytically. Consequently we will have to establish another method for approximating to it.

Approximation techniques like the Newton Raphson and the Regula-Falsi cannot be used. The main reason is that we do not know the behaviour of the constraint and the defining functions. This lack of knowledge makes us unaware of the local maxima and minima that the constraint and defining functions may exhibit as they are evaluated along the locus of a point that defines a line (i.e. the ray). Therefore, in the case of a ray intersecting several times with the implicit surface, any such approximation algorithm may converge to an unwanted intersection point, or oscillate between two intersection points without converging to any of them. The term 'unwanted intersections' means that the intersection point found is not the nearest to the origin of the ray toward the positive direction of the ray, but any other intersection point. Therefore we can never be certain about the solutions offered by such approximation algorithms.

The method of volume ray tracing by Kaufman, Cohen and Yagel [1993] is not suitable for our purposes because it explicitly imposes a three-dimensional canonical orthogonal lattice of cubes to cover the whole scene. Therefore, for a high resolution image of say, 3000 × 3000 pixels, they demand enormous computer storage capacity that would store information about the contents of $3000^3 \approx 27 \times 10^9$ cubes. The method of volume ray tracing that we propose, is distinctly different because it is based on sampling the values of the constraint and defining functions at regularly spaced specific points along the ray that we want to intersect. Firstly, for reasons of clarity, we re-introduce (from section 6.3.3) the concept of the *density function* which is also used in the Heidelberg ray tracing approach. In this way, the value of the density function at any given point is defined to be the difference of the

[2] Such rays, as we have discussed in chapter three, include the eye ray, its children, and any shadow feeler.

value of the constraint function minus the value of the defining function at this particular point. Consequently, the surface we have modelled will have a density value of zero.

$$density\_value = constraint\_value - defining\_value$$

In order to determine the nearest intersection of a ray with an implicit surface, along the positive direction of the ray, we start sampling the density function at regular intervals and observe the sign of the sampled density value. When we encounter a difference in the sign (say, from positive to negative) we can assume that an intersection point exists within the last sampling interval. Our assumption is correct in the case of analytically continuous surfaces (Bolzano theorem), but may not be correct for the rest of the cases. Let us assume for the moment that we have analytically continuous surfaces and we will see how to include discontinuous surfaces later.

Once we have encountered a change in the sign of the density value, we stop the sampling process and try to find a better approximation to the suspected intersection point. Here we can use the successive binary approximation method. Assuming analytical continuity of the sampled surfaces, we may encounter two different cases with regard to the number of intersection points. There may be exactly one such point, or more than one.

If there is only one point, the approximation method we use will converge to it. If there are more than one, however, our method may converge to any of them, not necessarily the nearest. In this latter case, we must consider the magnitude of the error that we may introduce by incorrectly choosing any other intersection point but the nearest. The results may be disastrous and greatly depend on the length of the sampling interval.

If, for example, the length of this interval is comparable with the dimensions of the window on the viewplane, then it is very likely that portions of the surface may be missed out, resulting to a highly inaccurate image on the viewport. If, however, the length of the interval is smaller than the dimensions of the viewplane's sub-windows, then the introduced error is insignificant and may not even become apparent on the generated image because it would affect the colour of at most one pixel on the viewport. Let us recall from chapter three, that

the sub-window is the rectangular area on the viewplane's window that maps exactly to one pixel on the viewport.

These observations, will be used to define the length of the sampling interval to be small enough when compared with the dimensions of the viewplane's sub-window. We estimate that a value of half the smaller of the sub-window's dimensions is sufficient for the purposes of our visualisation approach.

Coming back to the non-continuous case, we will introduce some 'intuitive' amendments to the intersection finding algorithm in order to enable the detection and correct determination of intersection points. We can observe that the binary successive approximation method may never converge to any specific value since there may not exist any intersection point but instead there may be a discontinuity in the surface.

In such cases the approximation method is likely to converge to one end-point of the discontinued surface or other sort of degeneracies may occur while evaluating the density function. These signs of non-convergence should be used for the proper identification of such problems. Once such a case has been detected, we can abandon the binary successive approximation method, reset our sampling algorithm, and continue sampling along the ray until another change of sign of the density value is detected.

When we find an intersection point, we apply a shading model that simulates global illumination, as we have already discussed in chapter three. With regard to the normal to the surface at this intersection point, we can use the same technique that we used during the octree visualisation approach.

Although we do not impose an orthogonal lattice of cubelets on the scene, the regular sampling that we propose implicitly introduces such a lattice. However, every time we use this lattice we have ensured that it is aligned with the direction of the ray we want to intersect. Furthermore, the length of the sampling interval is small enough to ensure that whatever the orientation of the ray the sampling interval will not cause trouble in areas that are covered with more than one sub-window.

This visualisation approach is unavoidably time consuming. The intersection of a ray with the scene may take several samples until an intersection point is found. There are also rays that they never hit any surface, thus wasting considerable time during sampling. It is therefore essential that we bound the scene with a simple geometrical object beyond which we never sample the density function but assume that no intersection points can be found.

Another adjustment we can make is in the number of iterations that we allow during the binary successive approximation method. During the octree visualisation approach we stopped the refining of the location of a surface point when the level of tolerance was smaller than the size of the cubelet. Recalling that the size of a cubelet is such that it is exactly mapped onto one sub-window on the viewplane, and that the sampling interval is also smaller than the dimensions of a sub-window, we can assume that a few iterations of the successive binary approximation method are enough to ensure an accurate definition of the intersection point.

Finally, another issue that we must consider in order to gain significant acceleration of this ray tracing approach is space coherence information regarding the distribution of the objects in the scene. Consider, for example, that we know that at particular volume of space the scene is empty from objects. Accordingly, whenever a ray passes through this region of space, we do not need to sample the density function at all. This space coherence information can be extracted effortlessly from the octree visualisation approach presented in the previous chapter.

7.7 A stochastic visualisation process

This section is concerned with the use of a pseudo-random number generator for suggesting point coordinates in order to evaluate the point membership classification test of the definition of the implicit models that we proposed.

This approach starts with a randomly chosen point. This point is specified by randomly choosing its coordinates. The coordinates of this point are used to evaluate the constraint and

the defining function. If these two function evaluate to *similar* values then we can assume that this point is in the vicinity of the surface we try to visualise. If this point evaluates the constraint and the defining function in distinctly different values, then we can safely ignore this point as it does not belong to the surface.

This is actually the concept of the point membership classification test upon which we based the construction of the models we proposed. Therefore, what we actually do is to test whether a randomly chosen point validates the point membership classification test of the model we intend to visualise.

Once we identify a candidate point as a member of the model's surface, we need to calculate its colour, map it on the viewplane, and paint the corresponding pixel accordingly. At this point we can distinguish between two different cases depending whether the corresponding pixel has already been painted or not. If it has not been painted, then we proceed by applying the chosen shading model, determine the colour of the corresponding pixel, and paint it. However, if it has already been painted, it means that we have already found a point that maps to this pixel.

Therefore, we have to check whether the newly found point is behind or in front of the previously found point, in relation to the observer. If the new point is in front, we over-paint the pixel else, we ignore it. This process implements a hidden surface removal mechanism, but it implies that we need to know the depth of every point that has been already painted.

After painting the corresponding pixel, or deciding to ignore the newly found point, we restart the algorithm by choosing randomly another point (by randomly assigning its coordinates) and proceed likewise. In this way this visualisation algorithm would continue to examine points and would never terminate. What we should do is to devise some termination criteria that would ensure an acceptable image on the viewport. One criterion may be to terminate the algorithm after a predefined number of points (say, 1600) has been found to belong to the scene. Another termination criterion would be to allow the user to stop the algorithm whenever the user finds it appropriate.

From this visualisation approach there is an issue that we wish to discuss. This is the trade-off between the accuracy of the generated images against the speed of the visualisation algorithm. We suggested that a candidate point belongs to the scene when the constraint and defining functions evaluate to similar values. Instead, for generating accurate representations, our attitude should have been to accept only the points that make the values between the constraint and defining functions equal. However, such an attitude would be very inappropriate to adopt because we would reject the vast majority of candidate points and waste CPU time. For this reason, we introduce the concept of *similarity*. A precise description of similarity would be that two numbers are similar if the absolute value of their difference is less than ε, for an arbitrary real positive value of ε.

In this way, points that are near to the surfaces of the scene will be treated as if they belong to the scene. But with this compromise we introduce an error to the image. This is a trade-off between speed of visualisation and accuracy of image and it depends on the value of ε. The smaller the value of ε, the more accurate the generated image but the slower the speed of visualisation, and vice versa.

What we propose is to avoid having a predefined universal value for ε but to change it according to the *success rate* of the visualisation algorithm. We define success rate to be the proportion of the number of points we used to paint pixels on the viewport over the total number of candidate points that we have examined so far. In order to increase the efficiency of the algorithm, indicated by the success rate, we will have to increase the value of ε. Similarly, if the success rate is higher than a desired level, we may have to decrease the value of ε and make the visualisation algorithm more accurate. This adaptive technique for assigning the value of ε again relies on the actual values ε takes, but in this way the range of permissible values is adjusted to the specifics of the scene. The desired level of the measure of the algorithms's success rate may be determined from a series of trial-and-error experiments.

Another technique that we may introduce to generate acceptable images at quicker speeds of visualisation, is to replace, during the stages of shading and mapping (Figure 3.1), the successful candidate points with small and simple geometric objects such as spheres or

cubes. Ranjan and Fournier [1994] suggested the use of spheres as a means to cover the skeleton of implicit models. But they converted the description of their implicit models into sets of spheres at a stage prior to visualisation. What we propose instead, is to make this conversion during visualisation. In this way we do not need a pre-processing stage, but we treat the successful candidate points as centre points of small spheres or cubes.

In this way we can use the exact point membership classification test, abandon the utilisation of ε and use spheres to replace the successful candidates. The radius of the spheres may be chosen such that, when mapped to the viewport, they would paint more than one pixel.

A problem arising with this technique is how we can establish whether a sphere is in front of another (which has been already painted), in relation to a given observer. We recommend to avoid determining the intersection between two spheres with analytical means, but simply record, for every pixel that we paint, the depth (distance form the observer) of its corresponding point. In this way we use the same process as for the 'un-accelerated' algorithm, as presented in the beginning of this section.

Spatial coherence may also be exploited by detecting volumes of space that the scene is empty. Using the octree visualisation approach, we can determine cubical regions of space where we are certain that the point membership classification test fails. However, directing the random generator to avoid these regions of space counteracts the savings offered by this information.

This visualisation approach offers a number of benefits. It is simple to conceptualise, and straightforward to implement. If we are not interested in the quality of the generated images, we can produce a rough approximation to the scene very quickly, compared to other visualisation alternatives (i.e. octree, ray tracing, polygonisation). This feature is very useful for the designer during the phase of modelling, since the model can be previewed and adjusted before we commit greater computer resources for a more accurately generated image.

Finally, the way we have structured the algorithm, makes this approach easily implemented for the exploitation the specialized hardware architecture of the *Z-buffer*: for every pixel that we paint, we also update the Z-buffer with the depth of the coordinates of the point we used for this colouring.

7.8 Concluding remarks

The aim of this dissertation was to explore the potential of computer graphics in its ability to visualise and manipulate geometric surfaces that cannot be handled with analytical tools. To achieve our aim, we felt it essential to study the domain of computer graphics both in terms of modelling and visualisation. In this way we identified computer graphics research that was relevant for our purpose. Additionally, we also recognized gaps in the relevant literature that we had to bridge in order to attain our aim.

Once we established the context within which we had to relate our research, we proceeded with describing formally (mathematically) the problem we aimed to resolve. We called this the initial problem definition and it formed the basis for the development of a new modelling approach. To achieve this development we went through two phases of extending the initial problem definition. The first, aimed at determining a more 'radical' (unconventional) definition for the measure of distance, while the second phase aimed at introducing a more 'intuitive' perspective for calculating the distance between objects.

After deriving this enhanced definition for surface description, we devoted our efforts in the understanding of its potential. We achieved this by dividing our surface definition in four different categories depending on the nature of a parameter that participates in the surface definition which we named the defining parameter.

For each category of this classification we conducted an elaborate analysis through a selection of examples. Plates 11 - 16 are associated with the *Class I* category. Plates 17 - 20 demonstrate the ability of our modelling approach to encompass several surface construction mechanisms such as bodies of revolution and extrusions (*Class II* models). Using *Class III*

models we built surfaces (plates 23, 24) that have been modulated with random and fractal processes. This class of models (*III*) reveals yet another aspect of the modelling approach that we have constructed; its ability to cope with implicitly defined data that cannot be approximated otherwise. With models of *Class IV* we demonstrated two more extremely significant capabilities of our modelling approach; its intuitive nature and its ability to generalize surface construction procedures. For this class we performed two series of experiments. The first, was to define intuitively paraboloid-like surfaces as plates 25 - 31 demonstrate. The second series of experiments extended the definition of the Voronoi tessellation in order to accommodate several geometrical objects as nuclei which were also assigned a weight factor. Plates 34 - 44 illustrate the effect of weights in a weighted Voronoi tessellation. Additionally, plates 32, 33 and 45 - 48 demonstrate some examples of weighted tessellations using points and lines as nuclei.

However, in order to complete our research and yield a useful modelling approach we also had to provide a visualisation approach. We chose to generate accurate representations of the surfaces we could describe and therefore we had to avoid approximations (to the modelled surfaces) as much as possible. The visualisation approach we proposed was adopted from the literature but was significantly adjusted in order to fulfil our specifications. However, because of our quest for accuracy of the surface representations, the visualisation approach we adopted was not particularly efficient. For this reason, we investigated (e.g. bounding volume information, spherical subcubes etc.) and proposed (e.g. polygonisation, stochastic sampling etc.) a number of different techniques that we may use to achieve significantly better visualisation speeds.

Finally, we have provided constructive criticisms over the complete combination of modelling and visualisation approaches that we used. These criticisms aimed at placing our research within the wider framework of computer graphics. This was attained by discussing our experience using the proposed modelling - visualisation combination, but also by briefly establishing a set of new directions for further enhancing our research.

In the research described in this dissertation we demonstrated the significance of implicit modelling. Moreover, the modelling approach that we developed illustrates how we can employ modern technology in order to conceptualise and understand the principles of implicit modelling. This was achieved through a computer graphics visualisation approach that we assembled - an adaptation of the octree visualisation - which enabled us to visualise the implicit definition of several families of geometrical objects. This piece of research, embodied in the construction of a modelling as well as a visualisation approach, has contributed to the conceptualisation, definition, manipulation and visualisation of implicitly defined geometrical objects (i.e. surfaces). We believe that the colour plates demonstrate the principal capabilities of the our (modelling/visualisation) approach, namely, its intuitive nature, the ability to generalise, and the refining of conceptualisation of implicitly defined objects.

Appendix A

Colour Plates

Plate 1	Mechanical parts using Constructive Solid Geometry
Plate 2	The method of polyspheres
Plate 3	Constant shading
Plate 4	Gouraud shading
Plate 5	Phong shading
Plates 6 - 10	Contour maps using the sum of distance
Plates 11, 12	The sum of distance from three points being constant
Plates 13 - 16	Varying the value of the defining parameter
Plate 17	The value of the defining parameter being a line
Plates 18 - 20	The defining parameter being the sin( ) function
Plates 21, 22	Using pseudo-random number generators
Plate 23	Surface modulation using a pseudo-random number generator
Plate 24	The Mandelbrot set (inset) being rotated
Plates 25, 26	A paraboloid defined by a point and a planar disk
Plates 27, 28	An extruded parabola defined by a point and a line segment
Plates 29, 30	A paraboloid defined by a line segment and a planar disk
Plate 31	A paraboloid defined by two line segments
Plate 32	Simple three dimensional Voronoi tessellation
Plate 33	Extended Voronoi tessellation determined by three line segments
Plate 34 - 44	Varying the weight of a nucleus
Plates 45, 46	Weighted tessellations using points
Plate 47	Weighted tessellation using line segments
Plate 48	Weighted Voronoi tessellation determined by line segments and points

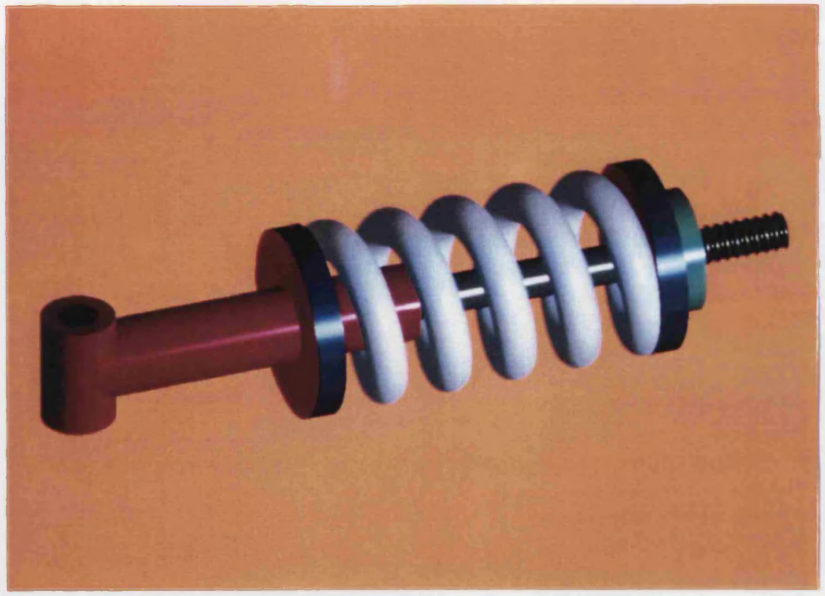


Plate 1

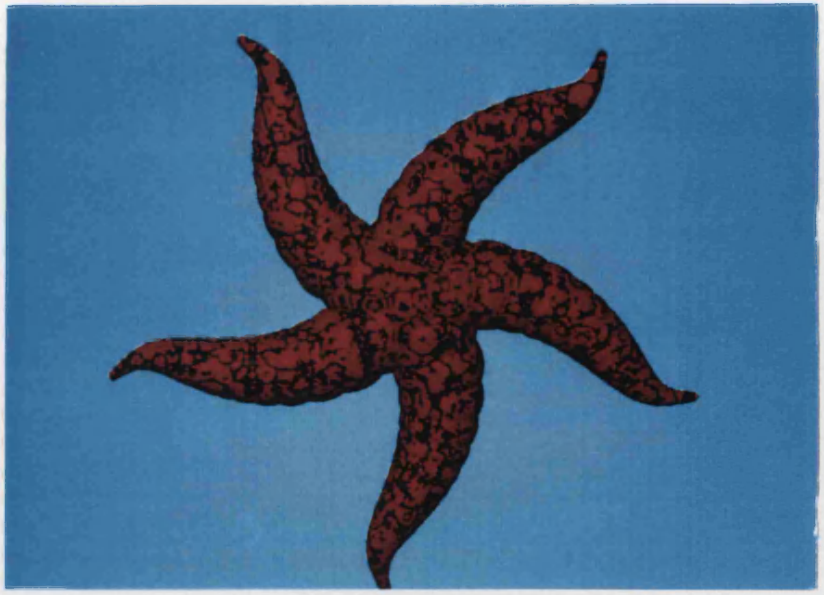


Plate 2

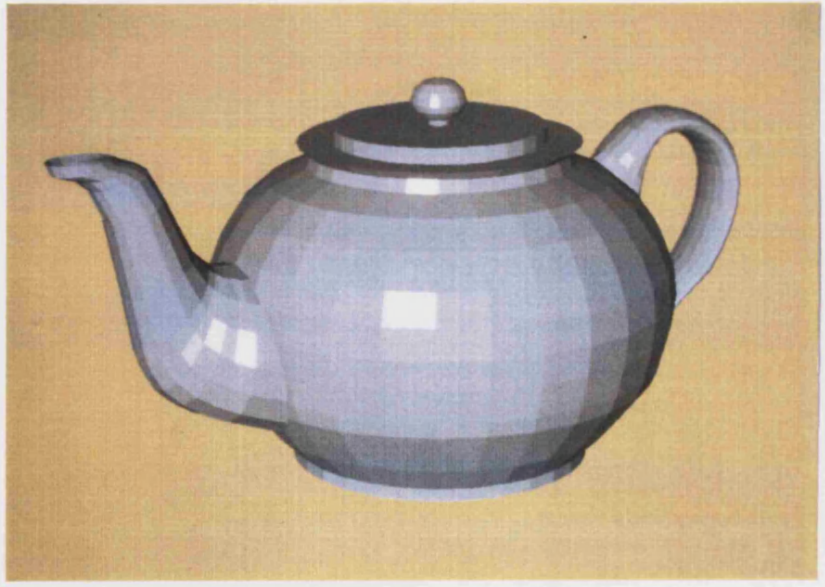


Plate 3

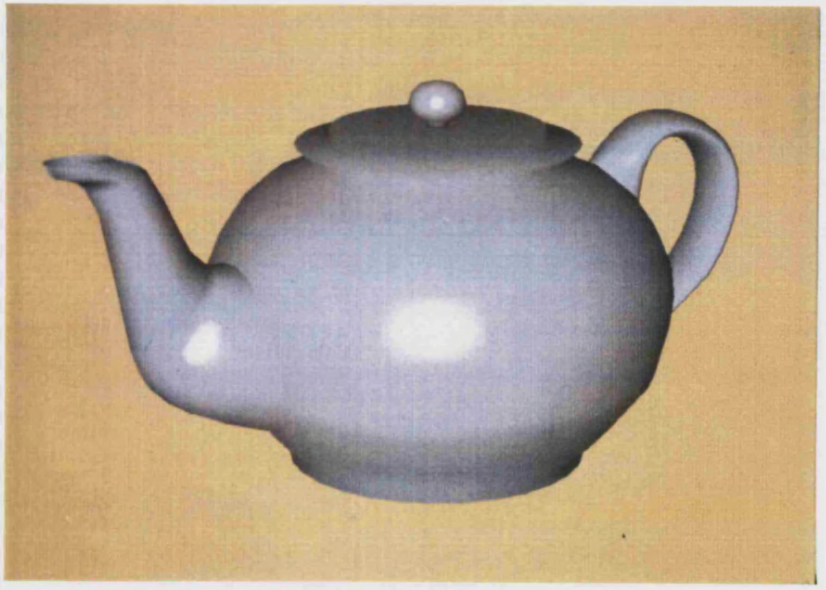


Plate 4

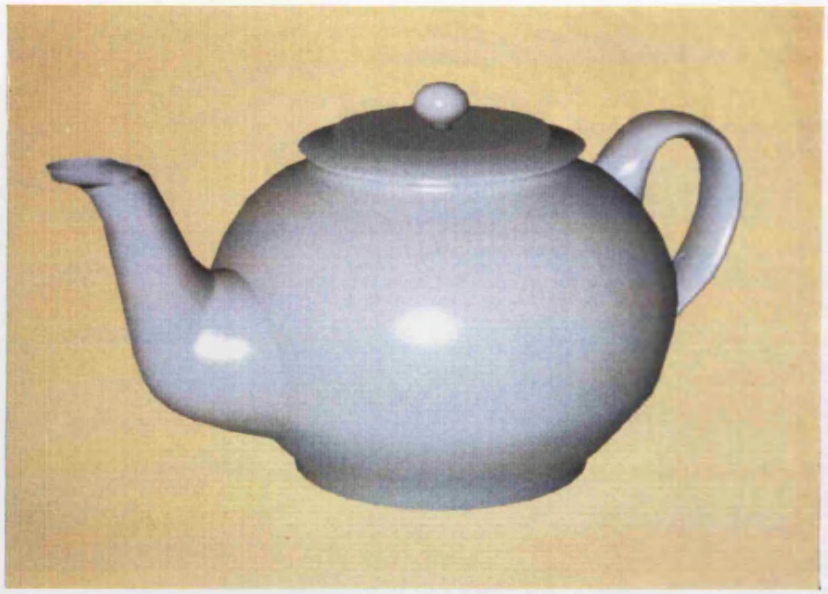


Plate 5

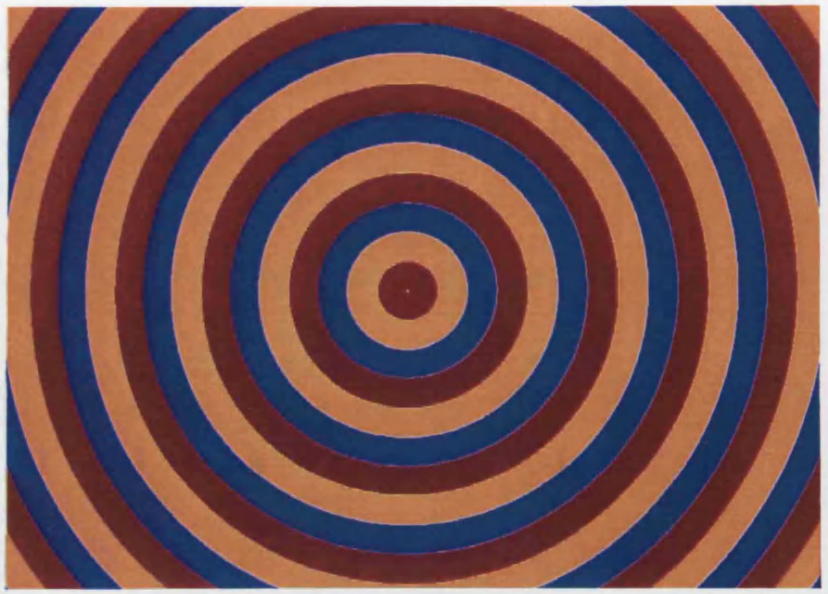


Plate 6

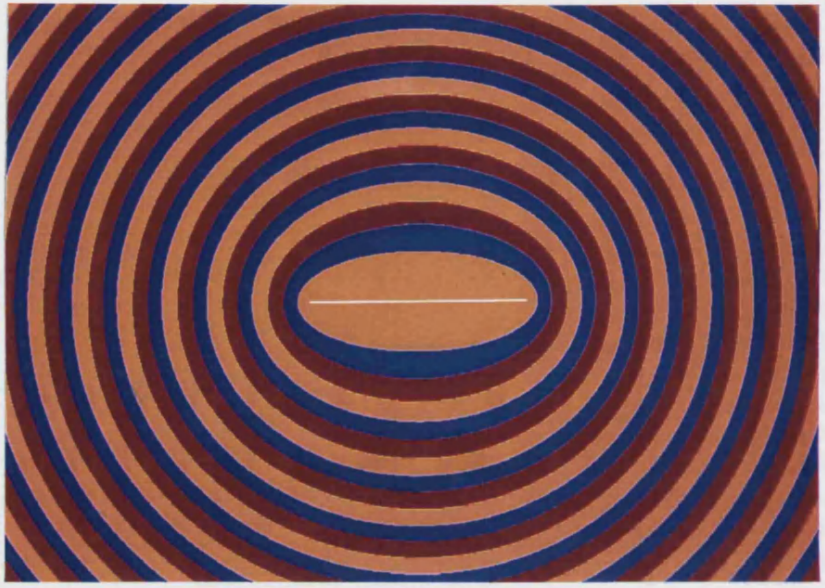


Plate 7

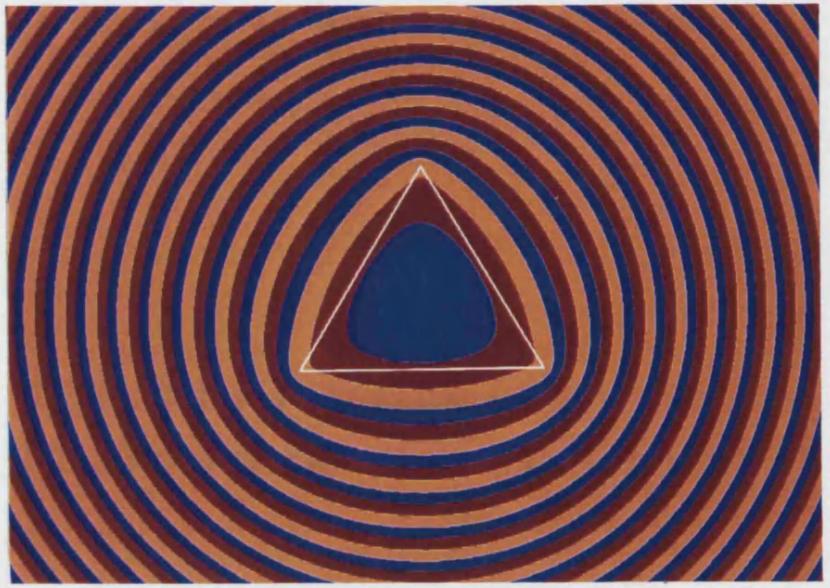


Plate 8

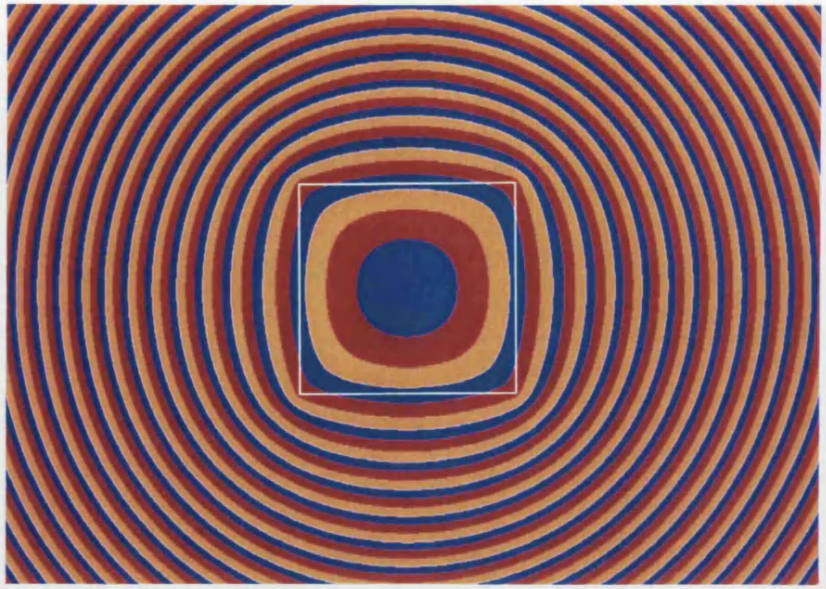


Plate 9

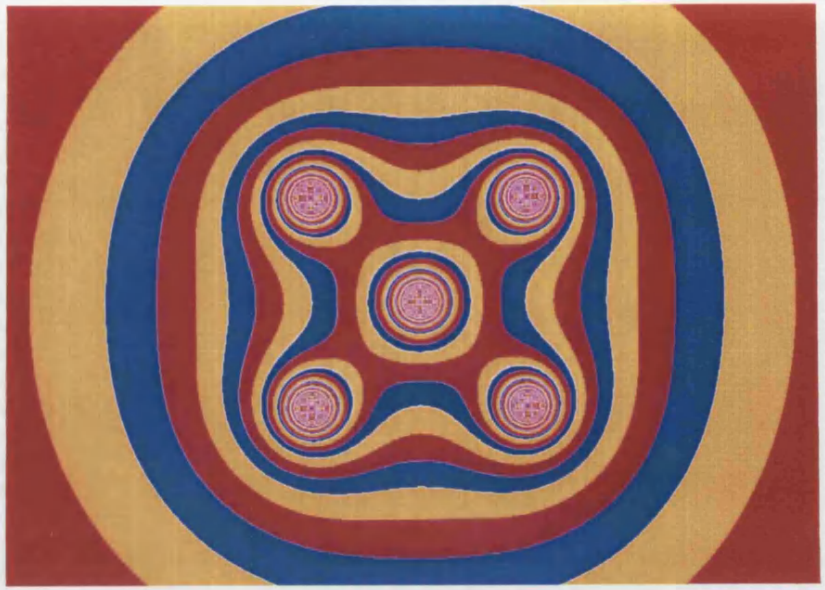


Plate 10

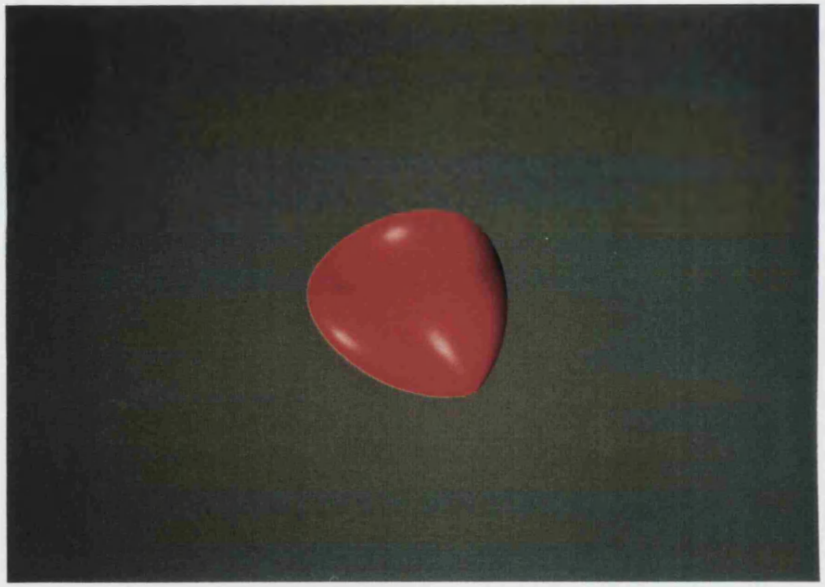


Plate 11

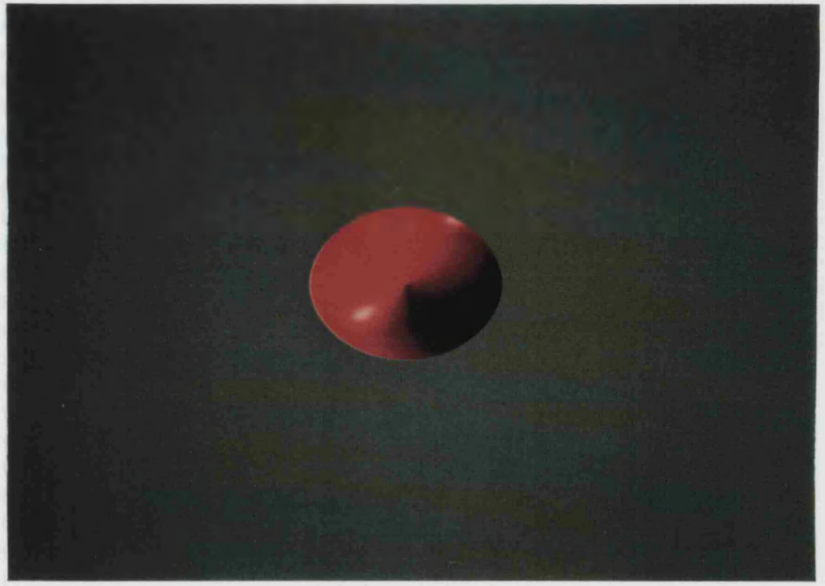


Plate 12

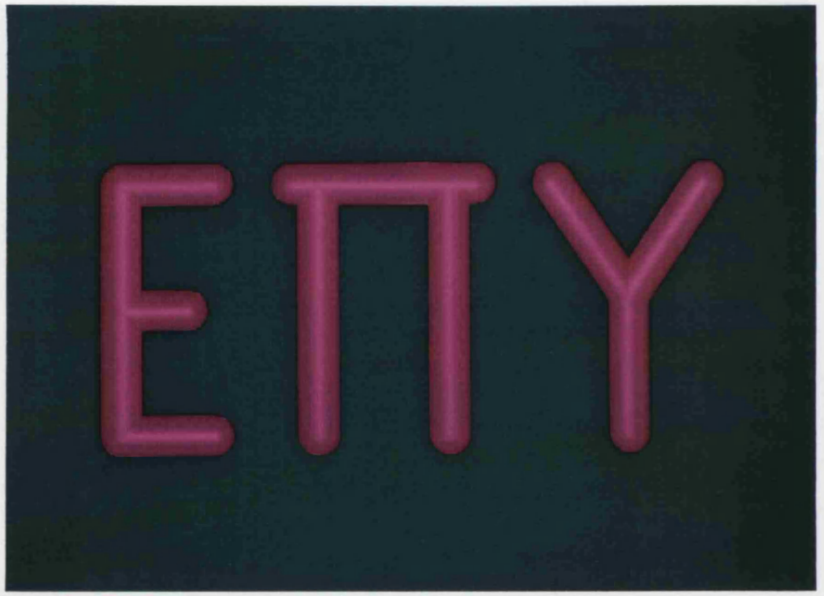


Plate 13

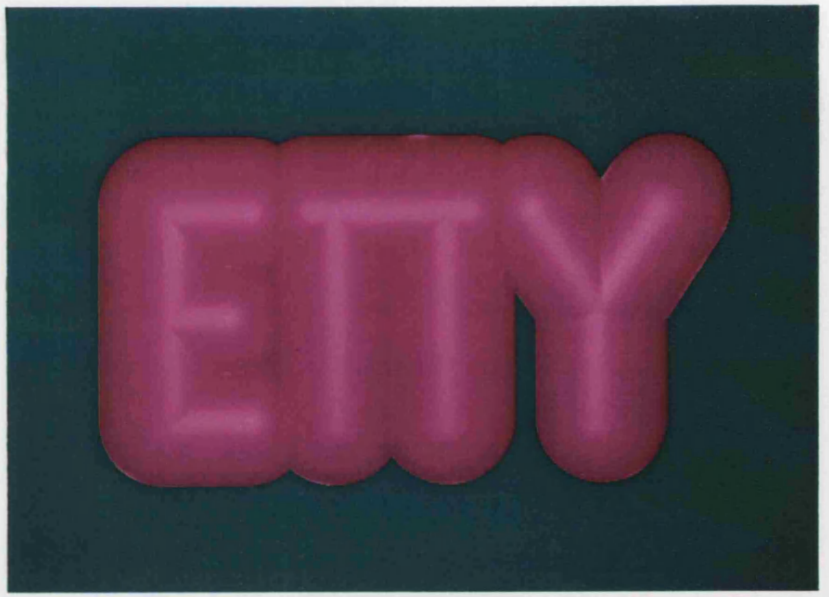


Plate 14

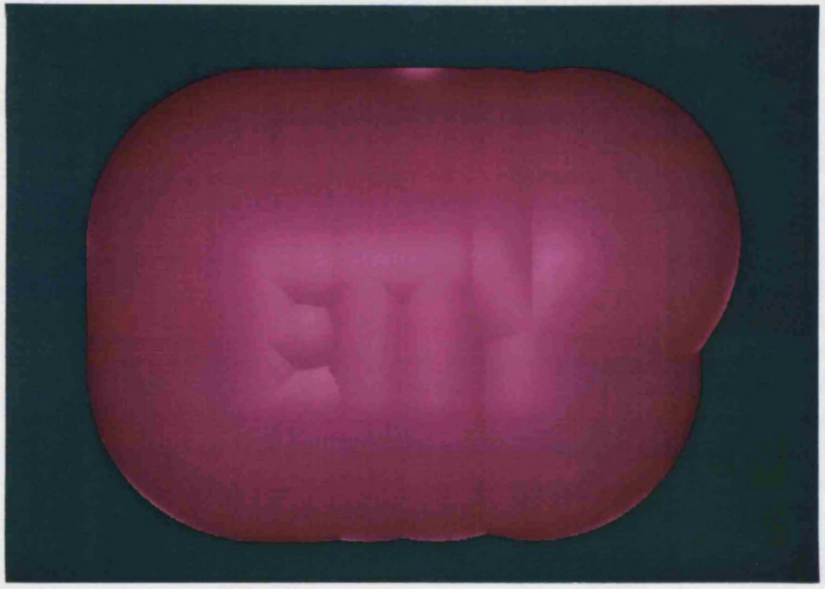


Plate 15

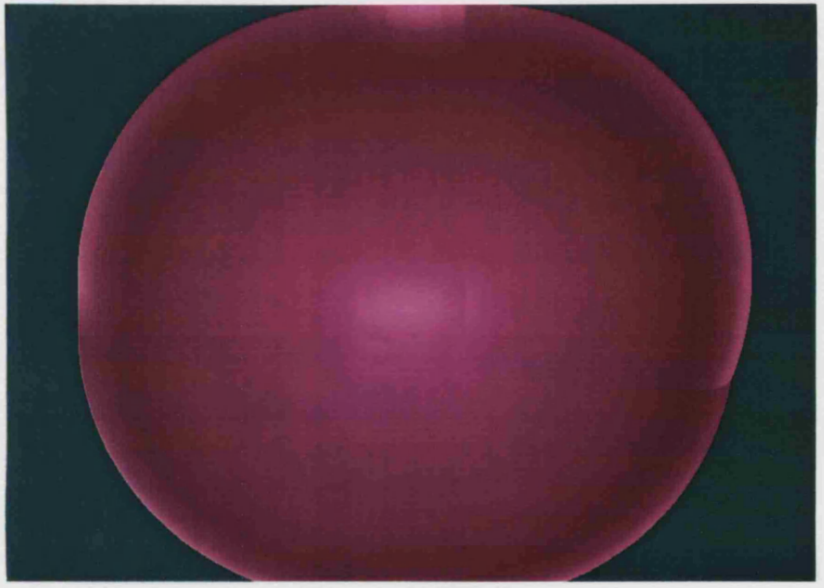


Plate 16

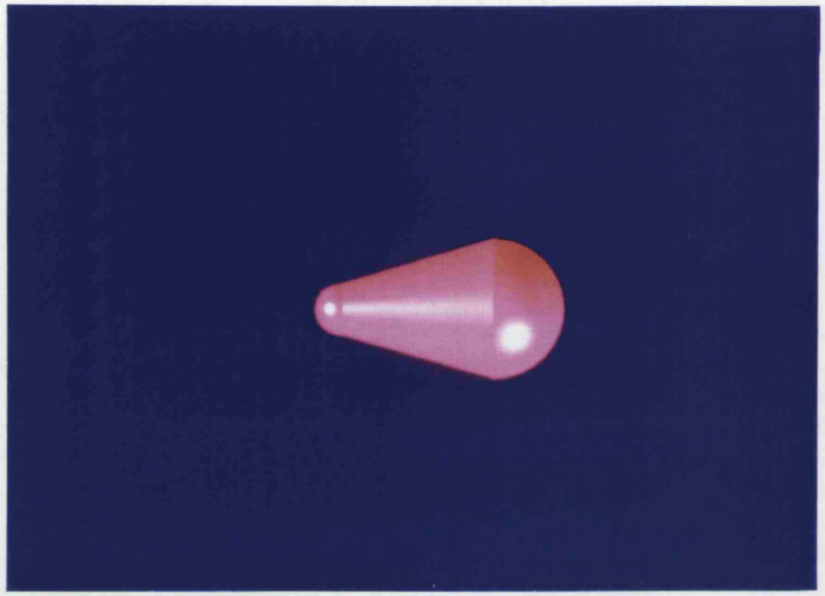


Plate 17

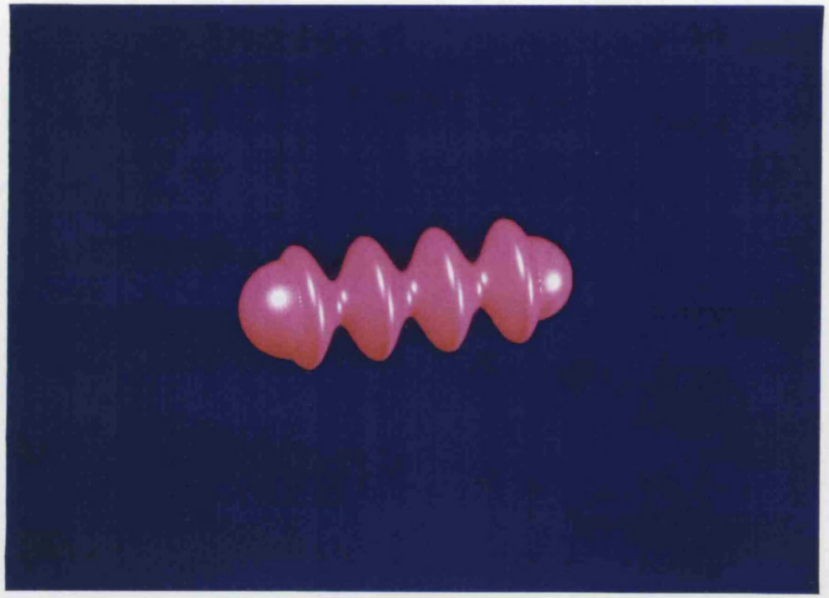


Plate 18

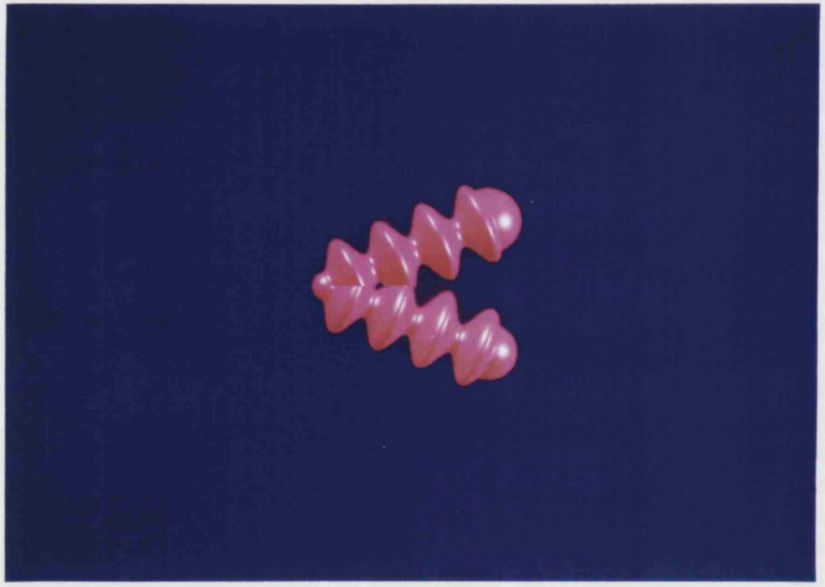


Plate 19

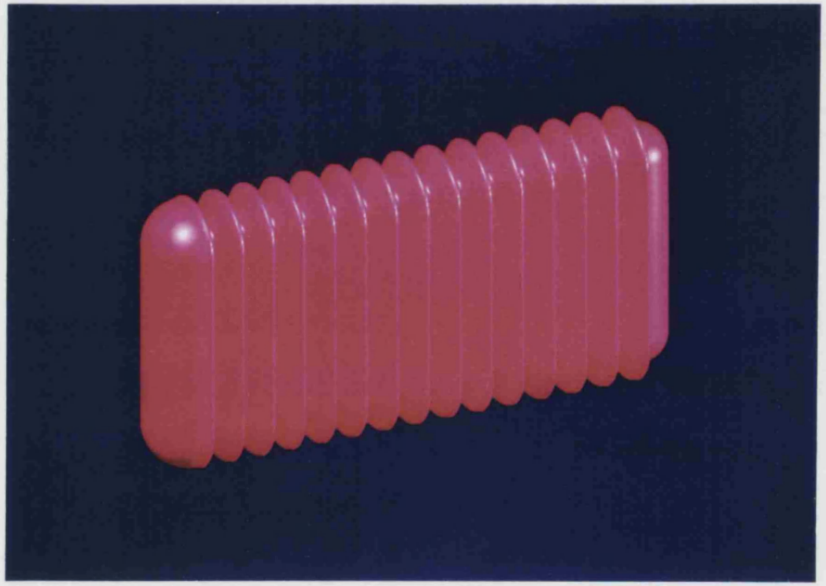


Plate 20

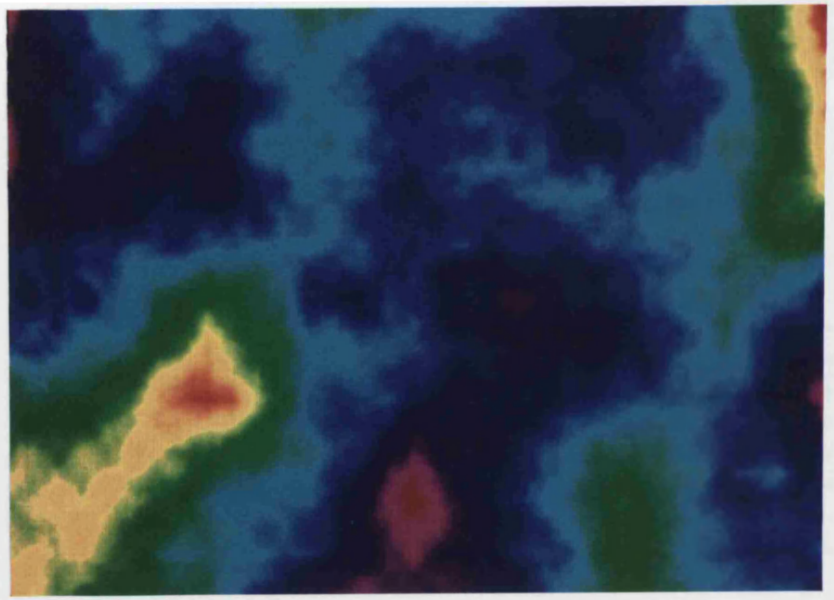


Plate 21

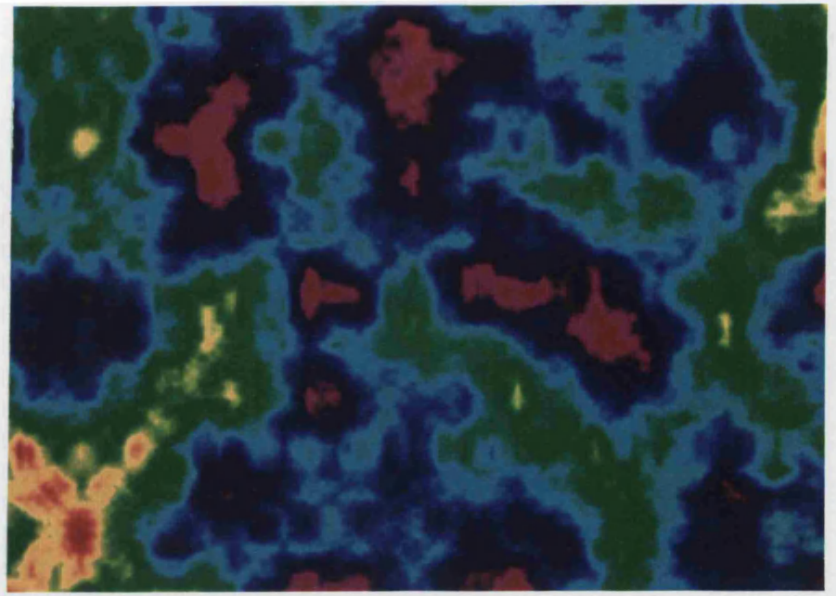


Plate 22

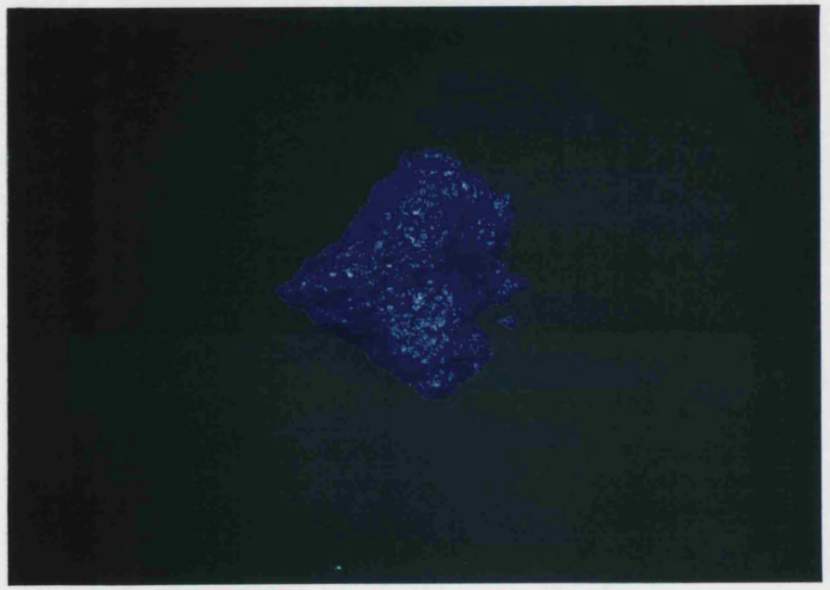


Plate 23

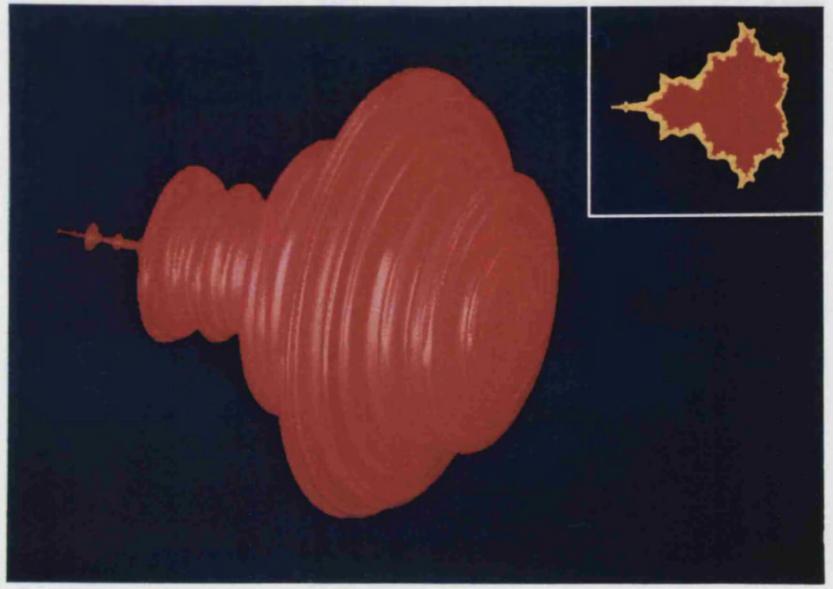


Plate 24

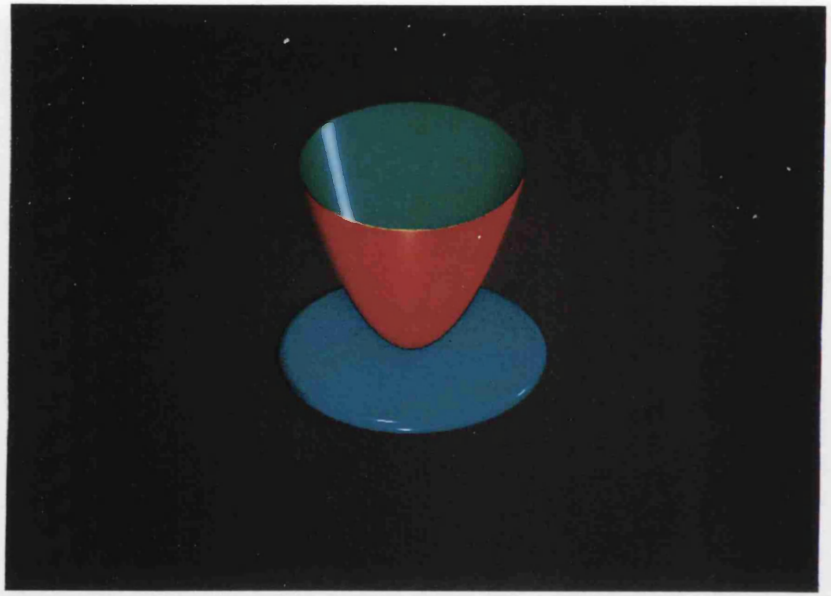


Plate 25

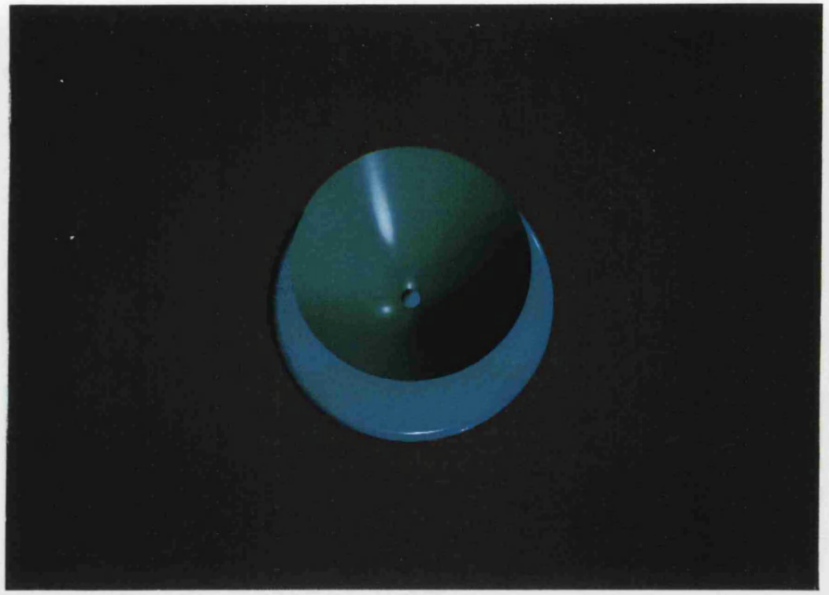


Plate 26

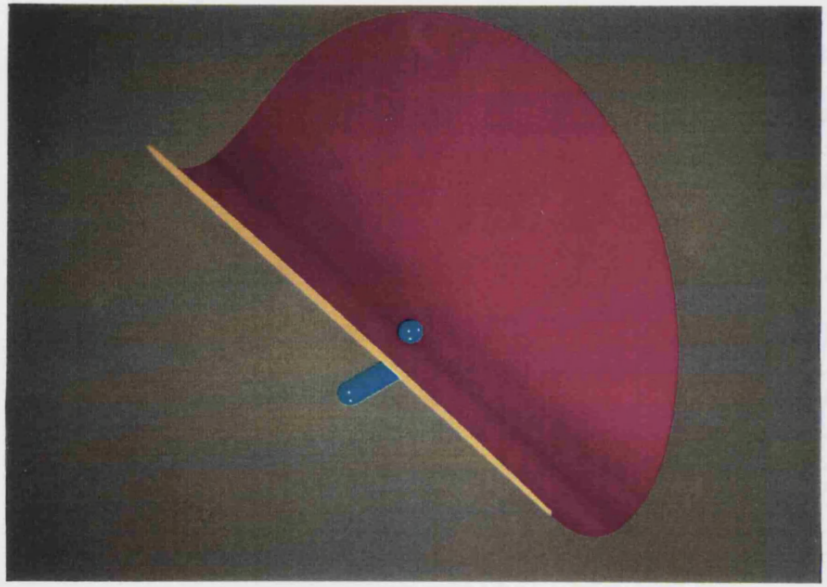


Plate 27

Plate 28

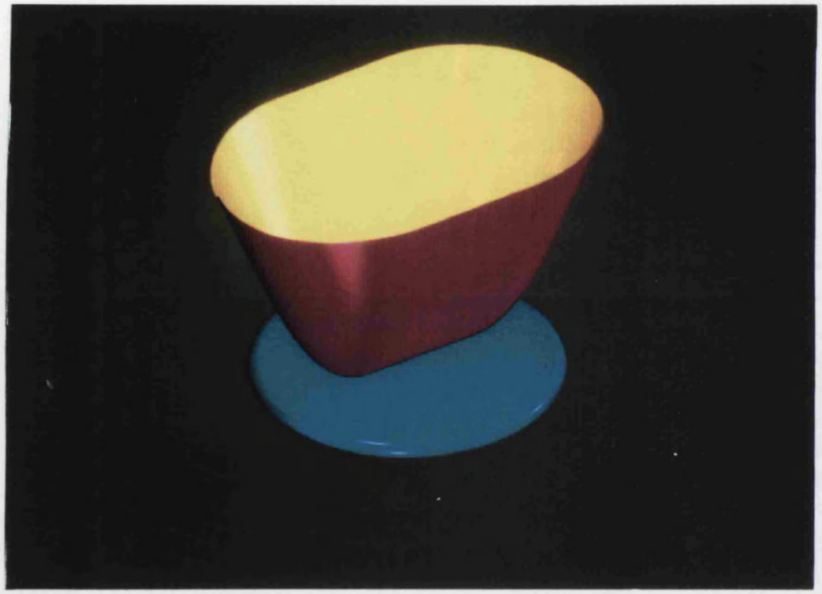


Plate 29

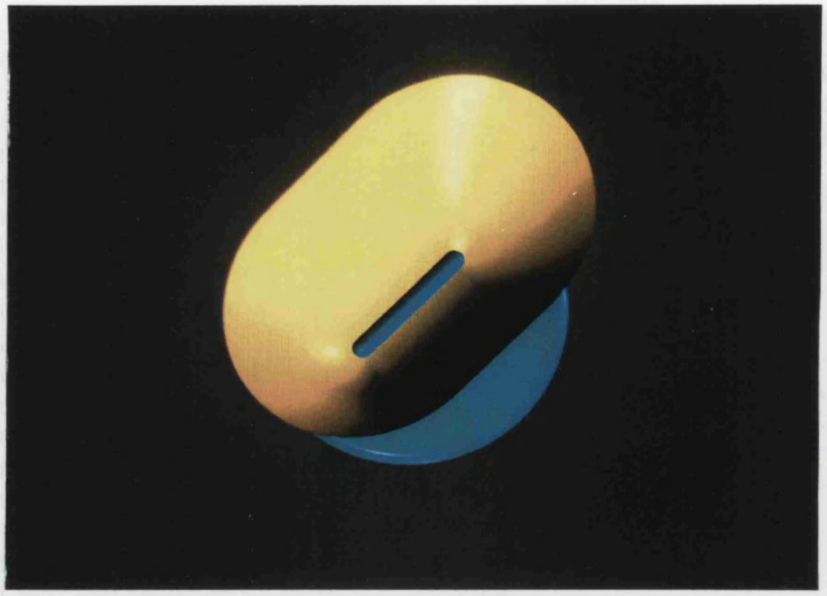


Plate 30

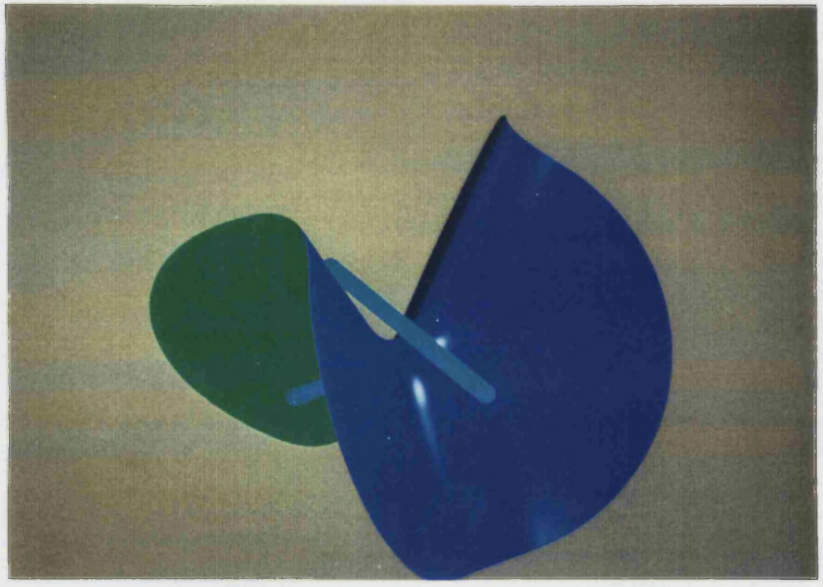


Plate 31

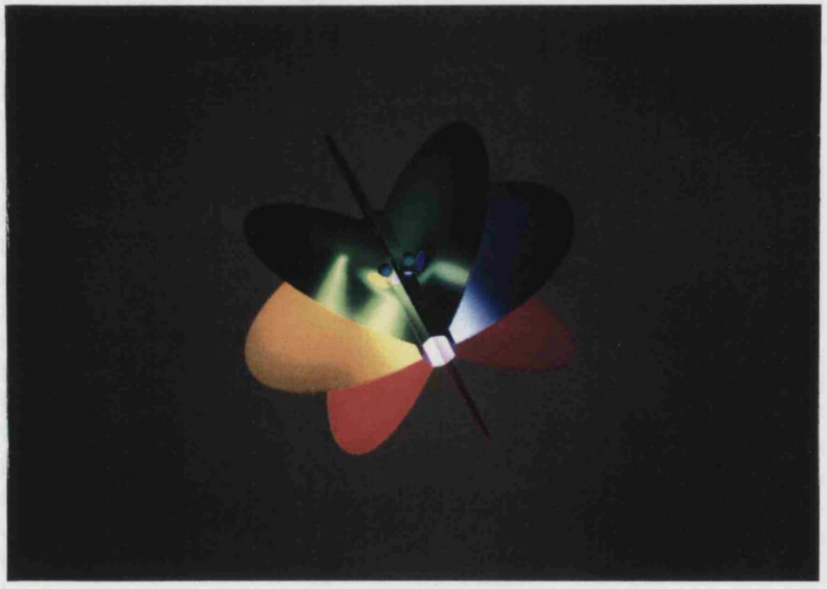


Plate 32

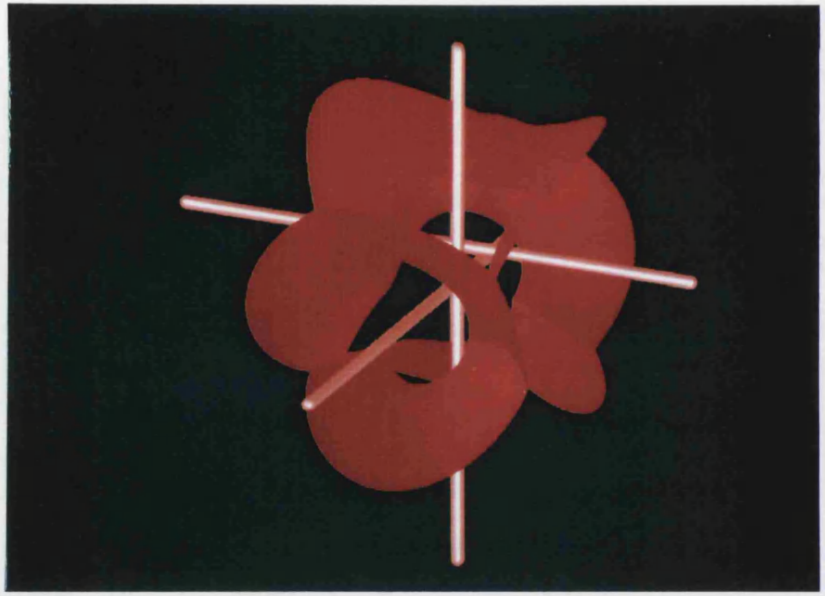


Plate 33

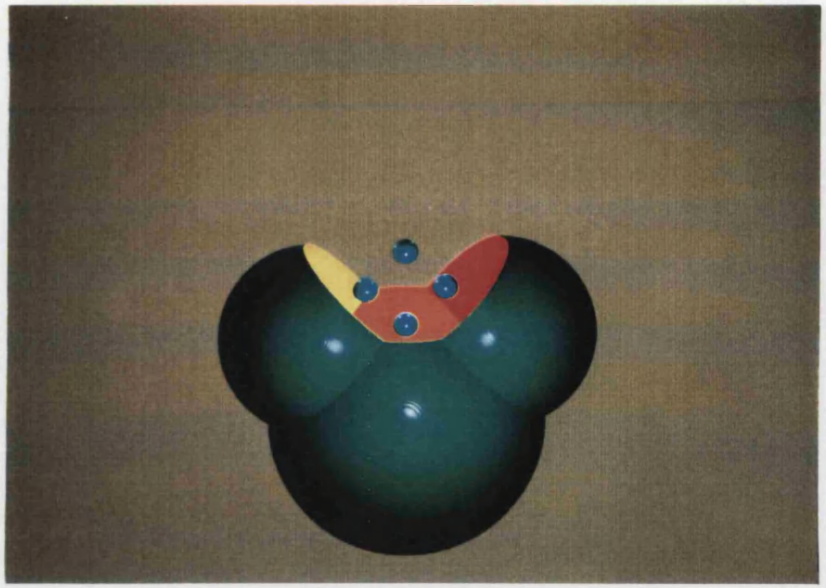


Plate 34

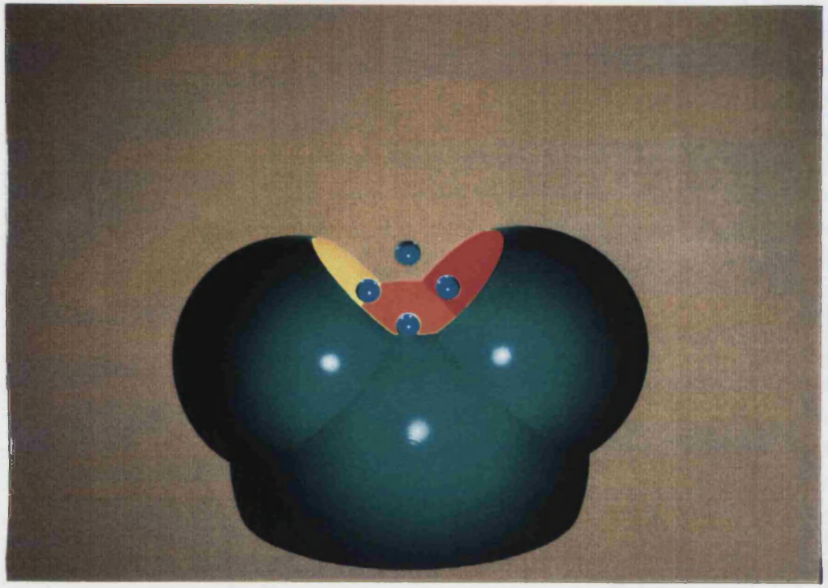


Plate 35

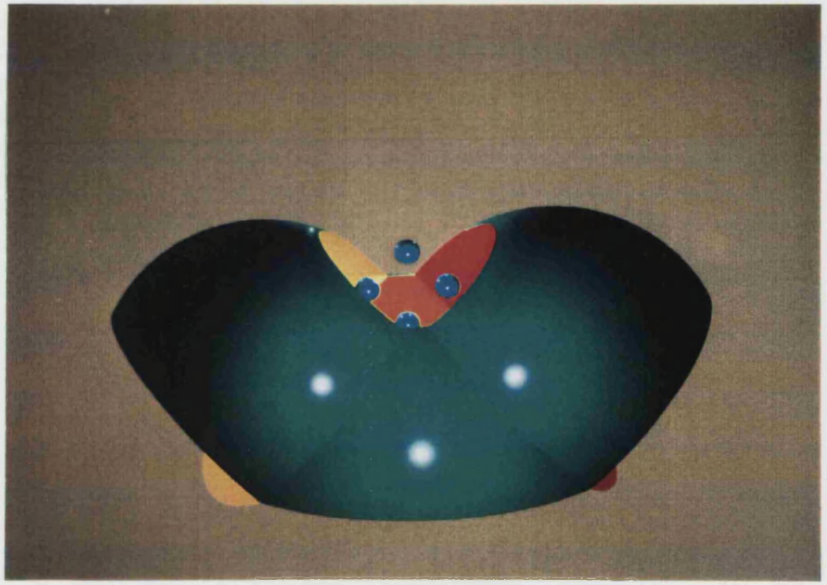


Plate 36

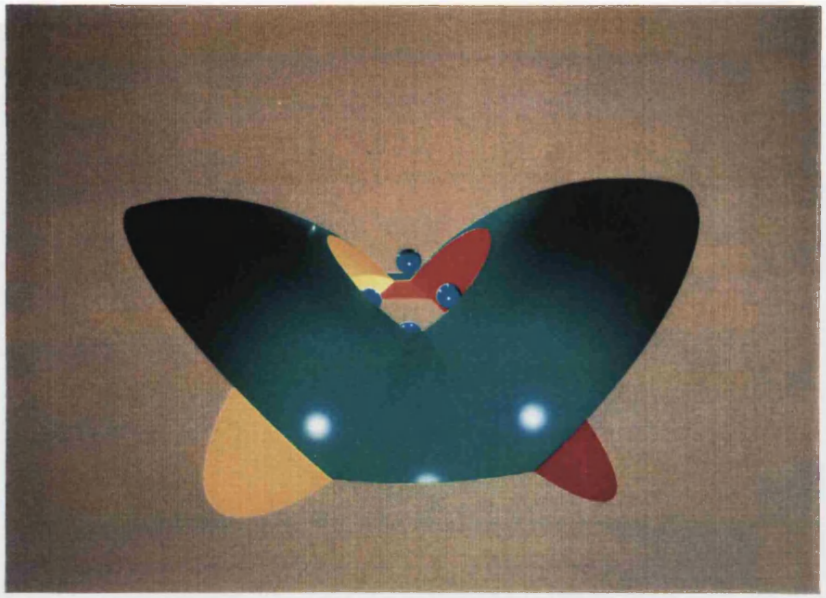


Plate 37

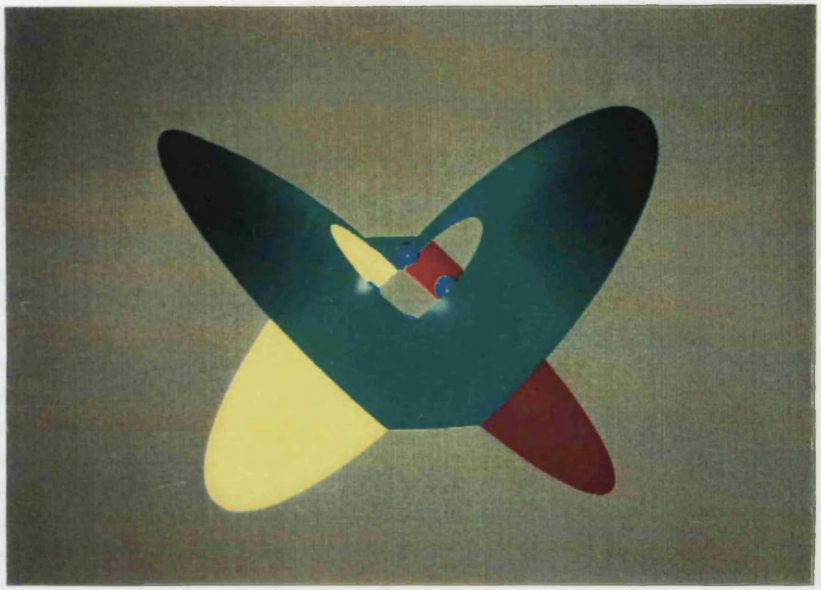


Plate 38

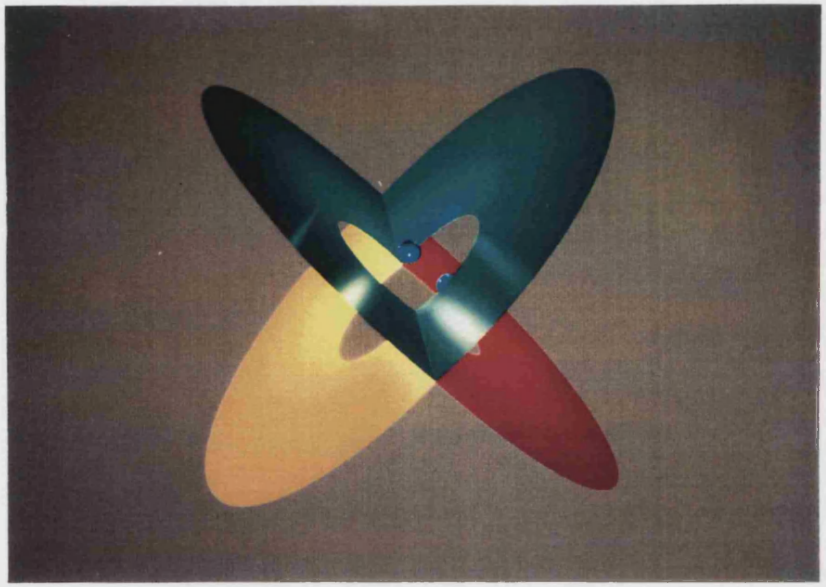


Plate 39

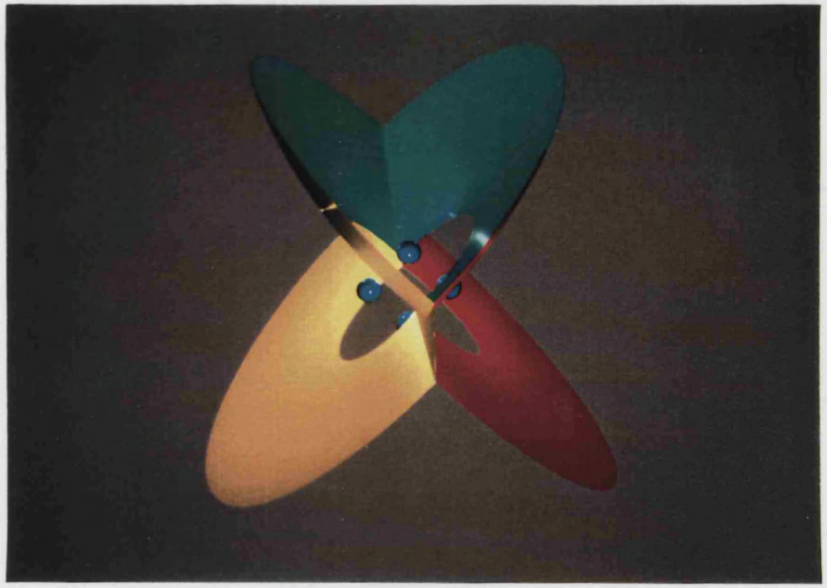


Plate 40

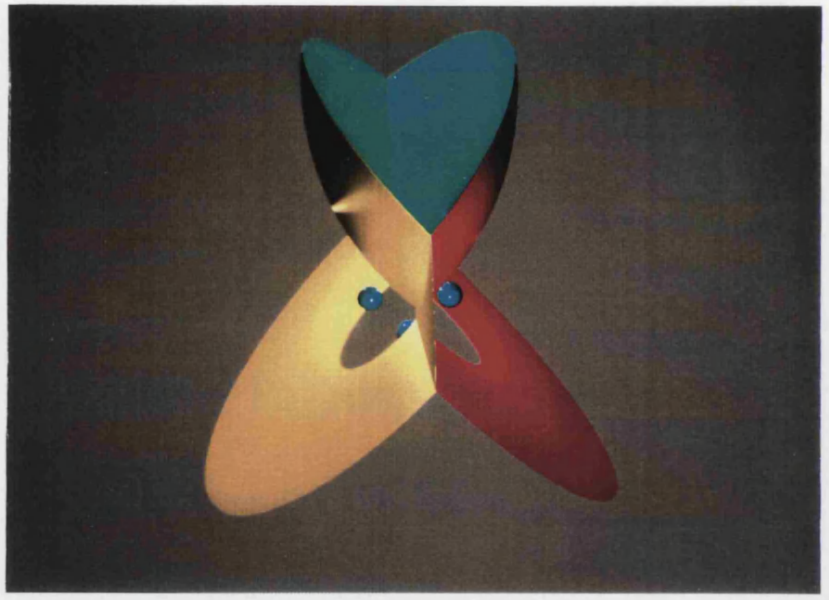


Plate 41

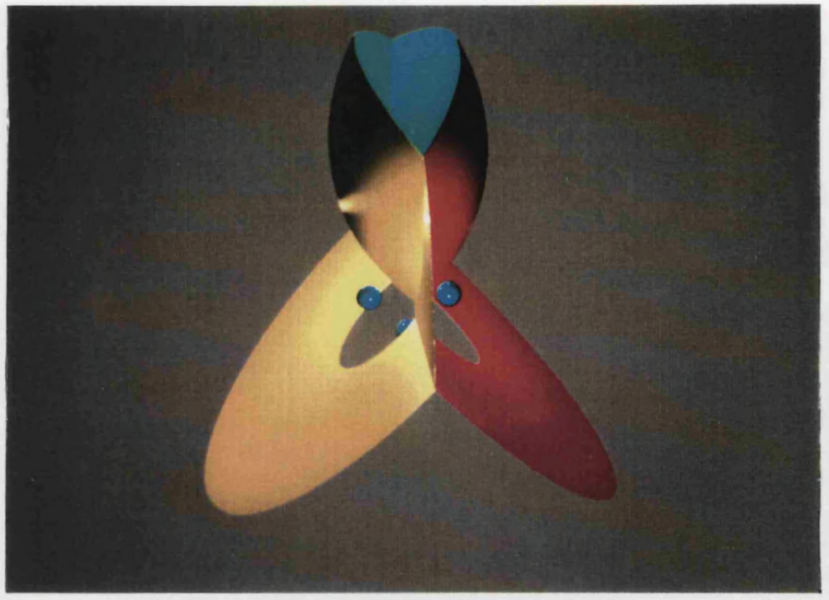


Plate 42

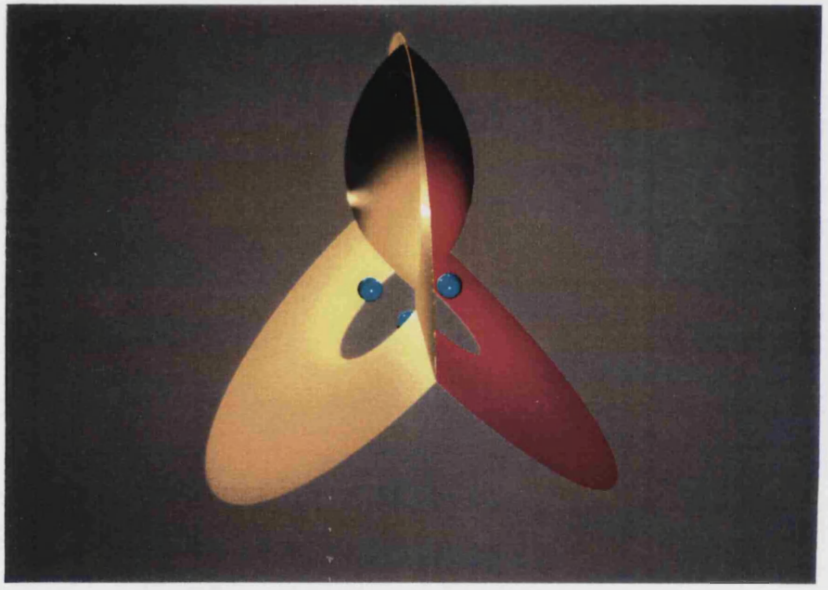


Plate 43

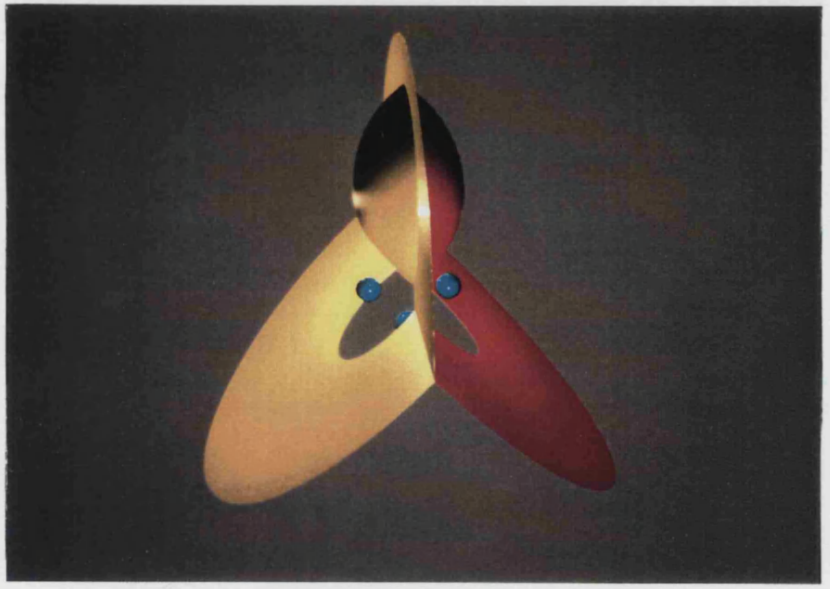


Plate 44

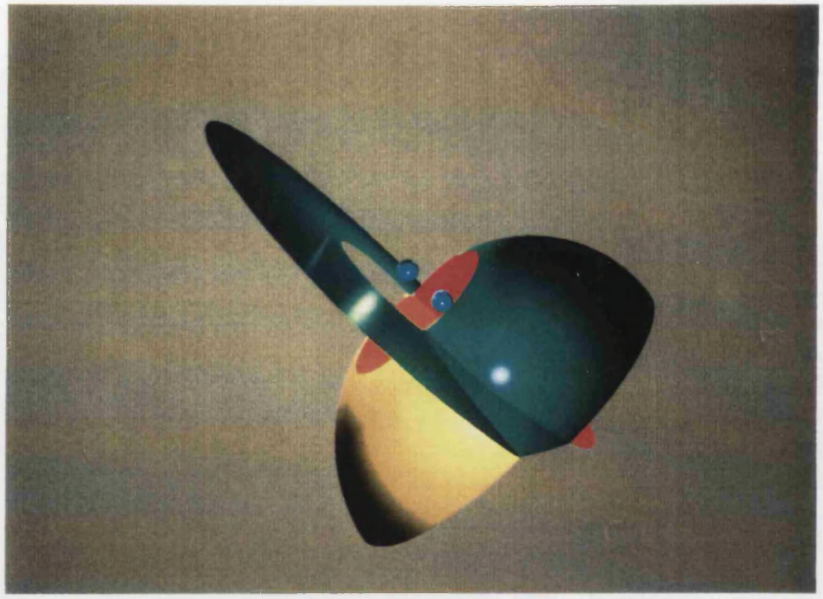


Plate 45

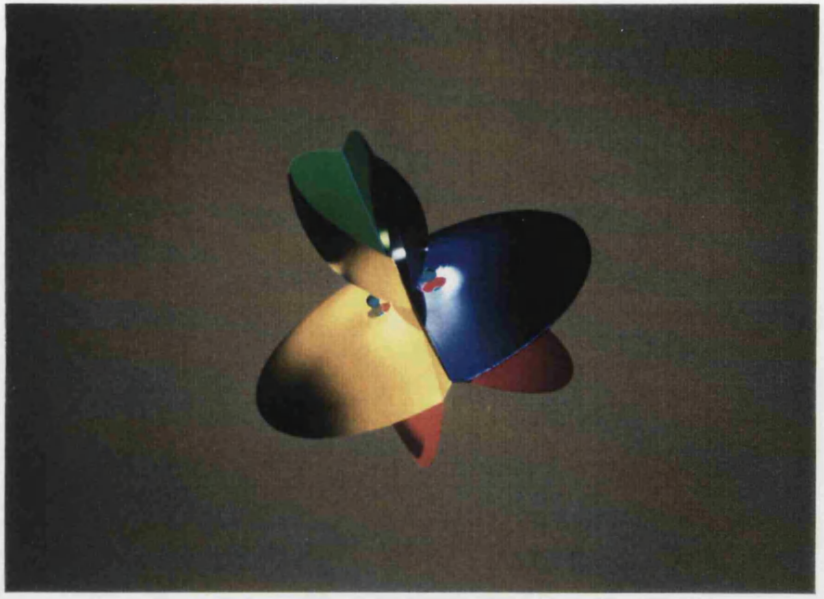


Plate 46

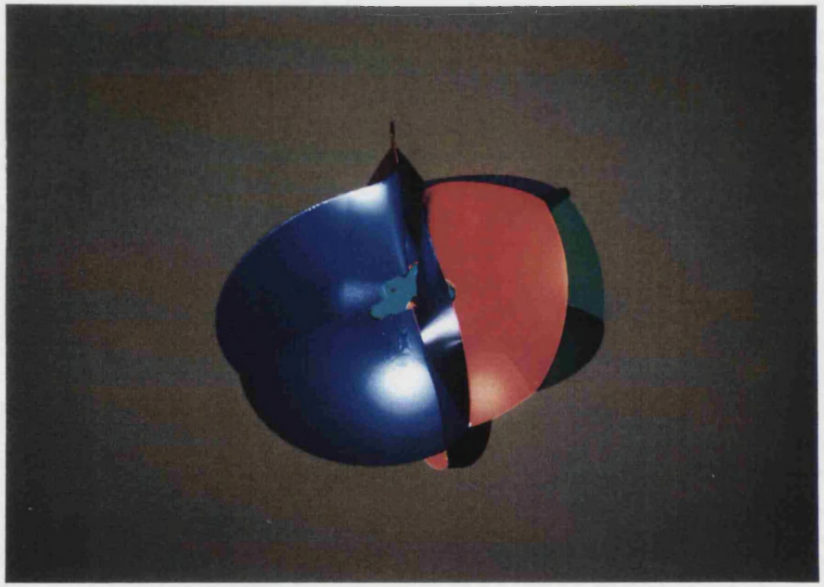


Plate 47

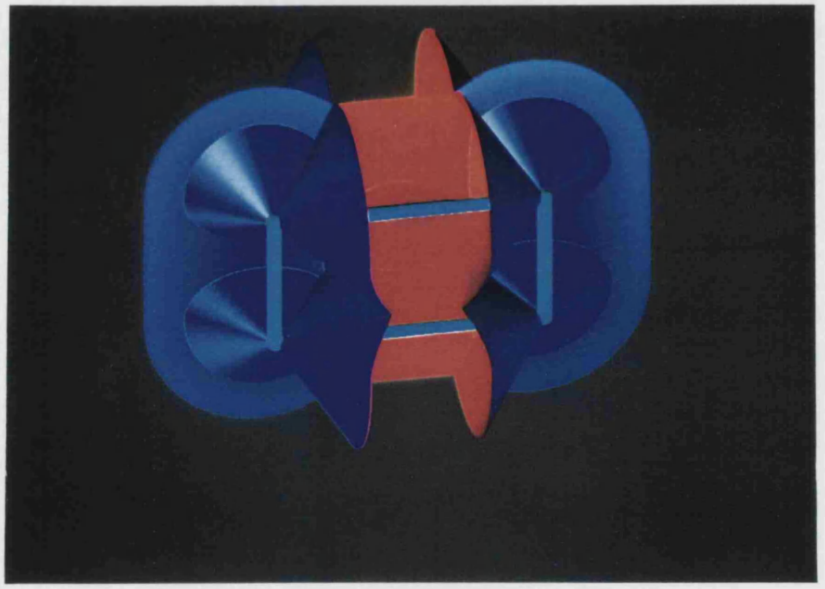


Plate 48

Appendix B

The purpose of this appendix is to highlight some important features of the software that we generated using our modelling and visualisation approach. First we present a list of the most important C++ classes that we constructed. Then, we discuss the constituent modules of the software produced and finally, we discuss the main stages of the software execution. Some examples of time used for achieving some of the colour plates will conclude this appendix.

C++ Objects

Analytic_object:	The description of a scene and methods for transforming it for different coordinate systems.
Cluster:	Particular subclass of the Analytic_object holding information about polygonal meshes.
CSGlists:	The implementation of the Constructive Solid Geometry approach, using lists.
CSGtree:	The implementation of the Constructive Solid Geometry approach, using trees.
Fractal_Texture:	The implementation of a process that generates smooth but random texture or surface variations using a pseudo random number generator
Material:	The materials data base.
Matrix:	A generic 4x4 matrix.
Palette:	information about the colour capabilities of a particular computer graphics viewport.
pixelvector:	A generic two-dimensional vector for the VIEWPORT coordinate system.
ranges:	Information about limiting the growth of surfaces
Screen_map:	A memory based map of the complete viewport to test for visibility of subcubes of the octree visualisation approach.
Sigma_object:	A subclass for the implicit models proposed in this dissertation.
Stack:	A generic stack structure for the octree.
Stackinfo:	Information about the octants of the octree visualisation approach.
vector3:	A generic three-dimensional vector
Viewport:	information about the specification of the geometry of a particular computer graphics viewport (e.g. VGA, XGA, HPGLplotter, ...).

Virtual_Buffer: A memory based immediate buffer for interfacing with the virtual viewport implementation.

Window: Controls the mapping from the continuous space of the computer graphics scene to the discrete space of a viewport and its associated palette.

Modules of the software used

The software implemented consists of eight modules namely, *octree*, *model*, *display*, *sigmadis*, *sigmacls*, *window*, *smap*, and *vrtbuf*. The first two (i.e. octree and model) import or, construct a scene and locate it according to the observer's viewing parameters. The next module (display) implements the octree visualisation algorithm, as well as the constructive solid geometry mechanism for a variety of geometrical objects. Then, the next two modules (sigmadis and sigmacls) attach to the octree visualisation algorithm in order to provide the necessary C++ classes for the manipulation of the implicitly defined surfaces that we study. Finally, the last three modules (display, smap and vrtbuf) are used to provide us with a real or virtual (or both) implementations of the viewport.

These modules are compiled separately but linked together in order to make an executable file under the DOS operating system. Information between the modules is achieved via common header files (.h) which include C++ standard libraries as well as the simple generic C++ objects and methods that we used. For example, methods that permit the multiplication of a matrix with a vector as well as the scaling and normalisation of a vector are available via the header files to all modules.

Procedural decomposition of the software

Upon invocation of the executable file, the following processes take place in order.

1. Verify existence of viewport and negotiate parameters (Window - Palette - Viewport)
2. Construct scene (Octree - Model)
3. Project scene according to viewing parameters (Model)
4. Use the octree algorithm until all pixels are painted (Display - Screen_Map)
5. Save results and exit (Display - Window)

Timing examples

The following table illustrates the amount of real time needed to visualise some of the high resolution (2048 x 1536), full colour (24bit) plates.

Plate	Time
Plate 11	13 mins
Plate 18	16 mins
Plate 19	20 mins
Plate 23	98 mins
Plate 24	134 mins
Plate 25	21 mins
Plate 28	17 mins
Plate 31	19 mins
Plate 44	230 mins
Plate 46	89 mins
Plate 47	160 mins
Plate 48	269 mins

Table B.1 Example execution times

References

Agin G. and Binford T. (1976).'Representation and description of curved objects'. *IEEE Transactions on Computers.* C-25. pp. 439 - 449.

Amanatides J. (1984). 'Ray tracing with cones'. *Computer Graphics, ACM SIGGRAPH '84.* Vol. 18, No. 3, pp. 129 - 135. July 1984.

Amanatides J. (1987a). 'Realism in computer graphics'. *Proceedings of the conference held at Computer Graphics 87.* Online Publications, London, October 1987. pp. 1 - 26.

Amanatides J. and Woo A. (1987b). 'A fast voxel traversal algorithm for ray tracing'. *Proceedings of the conference held at Computer Graphics 87.* Online Publications, London, October 1987. pp. 149 - 156.

Angell I. and Brownrigg D. (1987). 'Fractals for terrain maps & texturing'. *Proceedings of the conference Computer Graphics 87 in Computer Animation.* London, October 1987, Online Publications, pp. 203 - 216.

Angell I. and Moore R. (1986). 'A quad-tree algorithm for displaying a two-dimensional slice of an n-dimensional weighted Voronoi tessellation'. *Eurographics 1986,* pp. 19 - 27.

Angell I. and Tsoubelis D. (1992). '*Advanced graphics on VGA and XGA cards using Borland C++*'. MacMillan, London.

Apostolatos N. (1981). '*Numerical analysis*'. Textbook for the School of Mathematic, University of Athens. Two volumes written in Greek. Athens University Press.

Appel A. (1968). 'Some techniques for Shading Machine - Renderings of Solids'. *SJCC 1968.* Thomson Books, Washington D.C., pp. 37 - 45.

Arnaldi B., Priol T., Bouatouch K. (1987). 'A new space subdivision method for ray tracing CSG modelled scenes'. *The Visual Computer.*, Springer-Verlag, Vol. 3, pp. 98 - 108.

Arvo James and Kirk D. (1989). 'A survey of Ray Tracing Acceleration Techniques'. *An Introduction to Ray Tracing*. Glassner A.S. (ed.). Academic Press. London. pp. 201 - 262.

Atkin P., Ghee S., Packer J. (1987). 'Transputer architectures for ray tracing'. *Proceedings of the conference held at Computer Graphics 87*. Online Publications, London, October 1987. pp. 157 - 172.

Baker T. (1989). 'Automatic Mesh Generation for Complex Three-Dimensional Regions Using a Constrained Delaunay Triangulation'. *Engineering with Computers*, Vol. 5, pp. 161 - 175.

Banchoff T. (1990). *Beyond the third dimension; geometry, computer graphics, and higher dimensions*. Scientific American Library, No. 33, 1990.

Barnsley M, Jacquin A., Malassenet F., Reuter l. and Sloan A. (1988). 'Harnessing Chaos for Image Synthesis'. *ACM SIGGRAPH' 1988*, pp. 131 - 140.

Barnsley M. (1989). 'Fractals and Chaos'. *BCS Conference Proceedings on Fractals and Chaos*, 6-7 December 1989, London.

Barr A. (1989). 'Physically-Based Modeling: Past, Present, and Future'. Panel session at *ACM SIGGRAPH' 89 Panel Proceedings*. Vol. 23, No. 8, pp. 191 - 209, December 1989.

Barry Phillip J. and Goldman R.N. (1988). 'A Recursive Evaluation Algorithm for a Class of Catmull-Rom Splines'. *ACM SIGGRAPH '88*. Vol. 22, N0. 4, August 1988, pp. 199 - 204.

Barsky Brian A. (1984). 'A Description and Evaluation of Various 3-D Models'. *IEEE Computer Graphics & Applications*. Vol. 4, No.1 pp. 38 -52. January 1984.

Barsky Brian A. (1981). 'The Beta-spline: A Local Representation Based on Shape parameters and Fundamental Geometric Measures'. *PhD thesis*. Univ. of Utah, Salt Lake City. December 1981.

Barsky Brian A. and Beatty J.C. (1982). 'Varying the Betas in Beta-splines'. *Technical report* no. UCB/CSD 82/112, Computer Science Division, Electrical Engineering and

Computer Sciences Dept., Univ. of California, Berkeley, December 1982. Also *technical report* no. CS-82-49, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada.

Barsky Brian A. and Beatty J.C. (1983). 'Local Control of Bias and Tension in Beta-splines' *Computer Graphics, SIGGRAPH '83.* Vol. 17, No. 3, July 1983, pp. 193 - 218. Also in *ACM Transactions in Computer Graphics.* Vol. 2, No. 2, April 1983, pp. 109 - 134.

Bézier P.E. (1972). 'Emploi des machines à commande numérique'. Masson et Cie. Paris, 1970. Also published in English as: 'Numerical Control' in *Mathematics and Applications*, A.Robin Forrest and Anne F. Punkhurst, trans., John Wiley and Sons, London, 1972.

Bézier P.E. (1974). 'Mathematical and Practical Possibilities of UNISURF'. *Computer Aided Geometric Design*, Robert E. Barnhill and Richard F. Riesenfeld eds., Academic Press, New York, pp. 127 - 152.

Bézier P.E. (1977). 'Essai de definition numérique des courbes et des surfaces expérimentales'. PhD thesis, l'Université Pierre et Marie Curie, Paris, February.

Bier A. (1983). 'Solidviews. An Interactive Three - Dimentional Illustrator'. *BS & MS Thesis.* Dept. of EE & CS, MIT, May 1983.

Blinn J. and Newell M. (1976). 'Texture and Reflection in Computer generated images', *Communications of the ACM*, Vol. 19, No. 10, pp. 542 - 547, October 1976.

Blinn F. (1982). 'A generalization of algebraic surface drawing'. *ACM Transactions in Computer Graphics.* Vol. 1, No. 3, July, pp. 235 - 256.

Bloomenthal J., Wyvill B. (1990). 'Interactive techniques for Implicit Modelling'. *Computer Graphics, Special issue on 1990 Symposium on interactive 3D graphics.* Vol. 24, No. 2, March 1990, pp. 109 - 114.

Bloomenthal J. and Shoemake K. (1991). 'Convolution surfaces'. *ACM SIGGRAPH '91, Computer Graphics*, Vol. 25, No 4, July 1991, pp. 251 - 256.

Borland (1992). *Programmer's guide.* Borland International reference manual for the C++ programming language, version 3.1.

Bowyer A. (1981). 'Computing Dirichlet tessellations'. *The Computer Journal*, Vol. 24, No. 2, pp. 162 - 166.

Bowyer A., Wallis A., Milne P. (1987). 'Symbolic ray tracing'. *Proceedings of the conference held at Computer Graphics 87.* Online Publications, London, October 1987. pp. 127 - 133.

Bowyer A., Davenport J.H., Milne P., Padget J., Wallis A. (1989). 'A geometric algebra system'. *Geometric Reasoning.* J. Woodwark (ed.) Oxford Science Publications, Clarendon Press, Oxford, 1989.

Burtnyk N. and Wein M. (1976). 'Interactive skeleton techniques for enhancing motion dynamics in key frame animation'. *Communications of the ACM*, Vol 19, No 10, October 1976, pp. 564 - 584.

Carry H.B. and Schoenberg I.J. (1947). 'On spline distributions and their limits: The Polya Distribution Functions, Abstract 380t'. *Bull. American Mathematical Society.* Vol. 53, p. 1114.

Carry B. and Schoenberg I.J. (1966). 'On Polya Frequency Functions IV: The Fundamental Spline Functions and their Limits'. *J. d'Analyse Mathematique.* Vol. 17, pp. 71 - 107.

Catmull Edwin and Rom R. (1974). 'A class of local interpolating splines'. *Computer Aided Geometric Design.* R. E. Barnhill and R. F. Riesenfeld eds. Academic Press, New York, pp. 317 - 326.

Catmull E., Clark J. (1978). 'Recursively generated B-spline surfaces on arbitrary topological meshes'. *Computer Aided Design.* Vol. 10, No. 6, pp. 350 - 355, November 1978.

Clark J. (1976). 'Hierarchical geometric models for visible surface algorithms'. *Communications of the ACM.* Vol. 19, No. 10, October 1976, pp ?? - ??.

Cleary J.G., Wyvill B.M., Birtwistle G.M., Vatti R. (1985). 'Multiprocessor ray tracing'. *Computer Graphics For.* Vol. 5, pp. 3 - 12.

Clifton III T. and Wefer F. (1993) 'Direct Volume Display Devices'. *IEEE Computer Graphics and Applications*, Vol. 13, No. 4, July 1993, pp. 57 - 65.

Cook R.L. (1989). 'Stochastic Sampling and Distributed Ray Tracing'. *An introduction to ray tracing*. Glassner A.S. (ed.). Academic Press. London pp. 161 - 199.

Coons S.A. (1964). 'Surfaces for Computer Aided Design'. *Technical report*, Design Division, Dept. of Mechanical Engineering, MIT. Cambridge, Mass.

Coons S.A. (1967). 'Surfaces for Computer-Aided Design of Space Forms'. *Technical report no. MAC-TR-41* . Project MAC, MIT, Mass., June 1967. Available as AD-663 504 from NTIS, Spriengfield.

Cox M. (1972). 'The numerical evaluation of B-splines'. *J. Inst. Maths. Applic.* Vol. 10, pp. 134 - 149.

deBoor C. (1972). 'On calculating with B-splines'. *J. Approximation Theory*. Vol. 6, pp. 50 - 62.

Delaunay B. (1933). 'Sur la sphère vide'. *Bull. Academia of Sciences URSS, Classe des Sciences mathématiques et naturelles*, Vol. 6, pp. 793 - 800.

Devaney R. (1989). '*An introduction to Chaotic Dynamical Systems*'. Second edition, Addison-Wesley.

Dias Maria (1994). 'Ray Tracing Interference Color: Visualizing Newton's Rings'. *IEEE Computer Graphics and Applications*, Vol. 14, No. 3, May 1994, pp. 17 - 20.

Dippé M. and Swenson J. (1984). 'An adaptive subdivision algorithm and parallel architecture for realistic image synthesis'. *Computer Graphics, ACM SIGGRAPH '84*. Vol. 18, No. 3, pp. 149 - 158. July 1984.

Dirichlet G. (1850). 'Über die Reduction der positiven quadratischen formen mit drei unbestimmten ganzen zahlen'. *J. reine angew. Math.*, Vol. 40, pp. 209 - 227.

Doctor, L. J. and Torborg, J. G. (1981). 'Display Techniques for Octree-Encoded Objects.' *Computer Graphics and Applications*, 1(3), pp. 29 - 38.

Ellis J. (1991). 'The Ray Casting Engine and Ray Representations: A technical summary'. *International Journal of Computational Geometry and Applications*. Vol. 1, No. 4, December 1991, pp. 347 - 380.

Faux I. and Pratt M. (1979). *'Computational Geometry for Design and Manufacture'*. Ellis Horwood, Chichester.

Feibush E., Elliot A., Levoy M. and Cook R. (1980). 'Synthetic Texturing Using Digital Filters'. *Computer Graphics, ACM SIGGRAPH '80*, Vol.14, No. 3, pp. 294 - 301, July 1980.

FiELD (1992). *'Pamphlet for the advrtisement of the FiELD software'*. Lighting Technologies, Boulder, Colorado.

Firby P. and Stobne D. (1987). 'Colour manipulation of Superposed Families of Curves'. *The Computer Journal*, Vol. 30, No. 4, August 1987.

Foley J. and van Dam A. (1983). *'Fundamentals of Interactive Computer Graphics'*. Addison Wesley, Reading, MA, USA.

Foley J., van Dam A., Feiner S. and Hughes J. (1990). *'Computer Graphics principles and practice'*. Addison Wesley, Reading, MA, USA, second edition 1990.

Foley T., Lane D., Nielson G. and Ramaraj R. (1990). 'Visualizing functions over a sphere'. *IEEE Computer Graphics and Applications*, Vol. 10, No. 1, January 1990, pp. 32 - 40.

Forrest R.A. (1972). 'On Coons and Other Methods for the Representation of Curved Surfaces'. *Computer Graphics and Image Processing*. Vol. 1, No. 4, December 1972, pp. 341 - 359.

Forsey David R. and Bartels R. H. (1988). 'Hierarchical B-Spline Refinement'. *SIGGRAPH '88*. Vol. 22, No. 4, pp. 205 - 212.

Fujimoto A., Iwata K. (1985). 'Accelerated ray tracing'. *Proceedings of Computer Graphics, Tokyo '85*. pp. 41 - 65.

Fujimoto A., Tanaka T., Iwata K. (1986). 'ARTS: Accelerated Ray tracing System'. *IEEE Computer Graphics and Applications.* Vol. 6, No. 4, pp. 16 - 26.

Glassner A. (1989). 'An overview of ray tracing'. *An introduction to ray tracing.* Glassner A.S. (ed.). Academic Press. London pp. 1 - 31.

Gleick J. (1988). *CHAOS.* Penguin Group, Richard Clay Ltd. Bungay, Suffolk.

Goldsmith J., Salmon J. (1985). 'A ray tracing system for the hypercube'. *Caltech Concurrent Computing Project Memorandum.* HM154, California Institute of Technology

Goldstein R.A., Nagel R. (1971). '3-D Visual Simulation'. *Simulation.* pp. 25 - 31. January 1971.

Gouraud, H. (1971). 'Continuous Shading of curved surfaces.' *IEEE Transactions on Computers*, Vol C-20(6),June, pp. 623 - 628.

Green P. and Sibson R. (1978). 'Computing Dirichlet tessellations in the plane'. *The Computer Journal*, Vol. 21, pp. 168 - 173.

Greene N. and Heckbert P. (1986). 'Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter'. *IEEE Computer Graphics and applications*, Vol. 6, No. 6, pp. 21 - 27, June 1986.

Haines E. (1989). 'Essential Ray Tracing Algorithms'. *An introduction to ray tracing.* Glassner A.S. (ed.). Academic Press. London pp. 33 - 77.

Hall R.A, Geenberg D. (1983). 'A testbed for realistic image synthesis'. *ACM Transactions in Computer Graphics.* Vol. 2, No. 3, pp. 10 - 20.

Hall R.A. (1989). '*Illumination and colour in computer generated imagery*'. Springer Verlang, New York.

Halton J.H. (1970). 'A retrospective and prospective survey of the Monte Carlo method'. *SIAM rev.*. Vol. 12, No. 1, January 1970.

Hanrahan P. (1989). 'A Survey of Ray-Surface Intersection Algorithms'. *An introduction to ray tracing*. Glassner A.S. (ed.). Academic Press. London pp. 79 - 119.

Hanson A. and Heng P. (1992). 'Illuminating the fourth dimension'. *IEEE Computer Graphics and Applications*, Vol.12, No. 4, July 1992, pp. 54 - 62.

Heath T. (1956). 'The thirteen books of Euclid's Elements'. *Dover reprints*, New York.

Heckbert P. (1982). 'Color Image Quantization for Frame Buffer Display'. *ACM SIGGRAPH' 1982*. pp. 297 - 307.

Heckbert P.S. (1986). 'Survey of texture mapping'. *IEEE Computer Graphics and Applications*. Vol. 6, No. 11, pp. 56 - 67.

Heckbert P.S., Hanrahan P. (1984). 'Beam tracing polygonal objects'. *Computer Graphics, ACM SIGGRAPH '84*. Vol. 18, No. 3, pp. 119 - 127. July 1984.

Heckbert P.S. (1989). 'Writing a Ray Tracer'. *An introduction to ray tracing*. Glassner A.S. (ed.). Academic Press. London pp. 263 - 293.

Henderson P. (1993). '*Object -oriented specification and design with C++*'. McGraw Hill, UK.

Herzen VB. and Barr A. (1987). 'Accurate triangulations of Deformed, Intersecting surfaces'. *ACM SIGGRAPH '87*. Vol. 21, No. 4, July 1987, pp. 103 - 110.

Hoffman M. and Hopcroft E. (1985). 'Automatic surface generation in computer-aided design'. *The Visual computer*. Vol. 1, pp. 92 - 100.

Holladay T. (1980). 'An Optimum Algorithm for Halftone Generation for Displays and Hard Copies'. *Proceedings of the Society for Information Display*, 21(2), pp. 185 - 192.

Horn A. (1989). 'IFS and the interactive design of tiling structures'. *BCS Conference Proceedings on Fractals and Chaos*, 6-7 December 1989, London.

Hunter G.M. and Steiglitz K. (1979). 'Operations on images using Quad-trees'. *IEEE Transactions of Pattern Analysis and Machine Intelligence.* PAMI-1 No. 2, April 1979, pp. 145 - 153.

Jameson A., Baker T. and Weatherill N. (1986). 'Calculation of Inviscid Transonic Flow over a Complete Aircraft'. *AIAA 24th Aerospace Sciences Meeting*, AIAA-86-0103, Reno, Nevada, January 6-9, 1986.

Kajiya T. (1983). 'New techniques for ray tracing procedurally defined objects'. *ACM Transactions in Computer Graphics.* Vol. 2, No. 3, July 1983, pp. 161 - 181.

Kaufman A., Cohen D. and Yagel R. (1993). 'Volume Graphics'. *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 51 - 64.

Kay D.S. (1979a) 'Transparency, refraction and ray tracing for computer synthesized images'. *Master thesis.* Cornell Univ., Ithaca, NY. January 1979.

Kay D.S., Greenberg D. (1979b) 'Transparency for computer synthesized images'. *ACM SIGGRAPH '79.* Chicago, Ill., pp. 158 - 164.

Kay T. L. and Kajiya J. (1986). 'Ray Tracing Complex Scenes'. *Computer Graphics ACM SIGGRAPH '86.* Vol. 20, No. 4, August, pp. 269 - 278.

Kernighan B. and Ritchie D. (1988). *The C Programming Language.* Second edition based on the Draft-Proposed ANSI C. Prentice Hall Software Series, Englewood Cliffs, New Jersey 07632, page 46.

Knuth D. (1987). 'Digital Halftones by Dot Diffusion'. *ACM Transactions on Graphics*, 6(4), October, pp. 245 - 273.

Kobayashi H., Nakamura T., Shigei Y. (1987). 'Parallel processing of an object space for image synthesis using ray tracing'. *The Visual Computer.*, Springer-Verlag, Vol. 3, No. 1, pp. 13 - 22.

Κολλιας Γ. (1984). 'Δομες Δεδομενων'. Βοηθημα του μαθηματος "Δομες Δεδομενων" στο Τμημα Ηλεκτρολογων του Ε.Μ.Πολυτεχνειου και στο Μαθηματικο Τμημα του Πανεπιστημιου Αθηνας. Αθηνα, Νοεμβριος 1984. ('Data Structures'.

Textbook for the dept. of Electrical Engineers and dept. of Mathematics of the University of Athens. Athens, November 1984, in Greek).

Mandelbrot B. (1977). *FRACTALS. Form, Chance and Dimension.* W. Freeman &Co. San Fransisco, 1977. It is the modified second version of the *:Les objects fractals: forme, hasard et dimension.* Paris & Montreal: Flammarion 1975.

Mandelbrot B. (1982). *The Fractal Geometry of Nature.* W. Freeman &Co. New York, 1982.

McMillan T. (1992). 'The promise of portable color'. *Computer Graphics World,* Vol. 15, No. 9, September 1992 pp. 30 - 40.

Meagher, D. (1982). 'Geometric Modelling using Octree encoding.' *Computer Graphics and Image Processing,* 19, pp. 129 - 147.

Meagher, D. (1980). 'Octree Encoding: A New Technique for the Representation, Manipulation, and Display of Arbitrary Three-Dimensional Objects by Computer'. *Technical Report* no. IPL-TR-80-111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, NY, October 1980.

Meagher, D. (1982). 'Geometric Modelling using Octree encoding.' *Computer Graphics and Image Processing,* 19, pp. 129 - 147.

Menon J., Marisa R. and Zagajac J. (1994). 'More Powerful Solid Modelling through Ray Representations'. *IEEE Computer Graphics and Applications,* Vol. 14, No 3, May 1994, pp. 22 - 35.

Meyer G. and Greenberg D. (1980). 'Perceptual Colour Spaces for Computer Graphics'. *ACM SIGGRAPH' 1980,* pp. 254 - 261.

Murray J. and vanRyper W. (1994).*'Graphics file formats'*. O' reilly Associates, USA.

Nemoto K., Omachi T. (1986). 'An adaptive subdivision by sliding boundary surfaces for fast ray tracing'.*Graphics Interface '86.* Vancouver, B.C., pp. 43 - 48. May 1986.

Nishimura H., Ohno H., Kawata T., Shirakawa I., Omura K. (1983). 'LINKS-1 a parallel pipelined multicomputer system for image creation'. *Proceedings of the 10th symposium on Computer Architecture, SIGARCH.* pp. 387 - 394.

Papanickolaou C. (1978). *'Eucleidean Geometry'.* National Textbooks Publishing Organisation, Athens.

Payne B., Toga A. (1992). 'Distance field manipulation of surface models'. *IEEE Computer Graphics and Applications,* Vol. 12, No 1, January 1992, pp. 65 - 71.

Peitgen H.-O. and Richter P.H. (1986). *The Beauty of Fractals.* Springer-Verlang, Berlin Heidelberg, page 191.

Phong B. T. (1975). 'Illumination for Computer Generated Pictures.' *Communications of the ACM,* 18(6), June, pp. 311 - 317.

Pickover C. (1989). 'A Short Recipe for Seashell Synthesis'. *IEEE Computer Graphics and Applications,* Vol. 9, No. 11, November, pp. 8 - 11.

Postcript® (1987) 'Language Reference Manual'. *Adobe Systems Incorporated.* 8th ed. Addison - Wesley, pp. 70 - 71.

Ranjan V. and Fournier A. (1994). 'Volume Models for Volumetric Data'. *IEEE Computer,* Vol. 27, No. 7, July 1994, pp. 28 - 36.

Riesenfeld R. (1973). 'Applications of B-spline approximation to geometric problems of computer aided design'. *PhD Thesis.* Syracuse University, Syracuse, New York.

Rockwood A. (1989). 'The displacement method for implicit blending surfaces in solid models'. *ACM Transactions on Graphics,* Vol. 8, No. 4, pp. 279 - 297.

Schoenberg Isaac J. (1946). 'Contributions to the Problem of Approximation of Equidistant Data by Analytic Functions'. *Quarterly Applied Math.* Vol. 4, No. 1, pp. 45 - 99, 112 - 141.

Shani U. and Ballard D. (1984). 'Splines as Embeddings for Generalized Cylinders'. *Computer Vision, Graphics, and Image Processing.* Vol. 27, pp. 129 - 156.

Shinya M., Takahashi T. and Naito S. (1987). 'Principles and applications of pencil tracing'. *Computer Graphics, ACM SIGGRAPH '87.* Vol. 21, No. 4, pp. 45 - 54. July 1987.

Sidhu G.S. and Boute R.T. (1972). 'Property encoding: applications in binary picture encoding and boundary following'. *IEEE Transactions on Computers.* Vol. C-21 No. 11, November 1972.

Smarte, G. and Baran, N. (1988). 'Display Technology; Face to Face'. *BYTE,* Vol. 13, No. 9, September 1988, pp. 243 - 252.

Stroustrup B. (1987). *The C++ programming language.* Addison Wesley, USA, 1987.

Terzopoulos D. (1989). 'Physically-Based Modeling: Past, Present, and Future'. Panel session at *SIGGRAPH' 89 Panel Proceedings.* Vol. 23, No. 8, pp. 191 - 209, December 1989.

Thiessen A. (1911). 'Precipitation averages for large areas' *Monthly Weather Review.* Vol 39, pp. 1082 - 1084.

Thomas S.W. (1986). 'Dispersive refraction in ray tracing'. *The Visual Computer.* Vol. 2 pp. 3 - 8.

Tiller W. (1983). 'Rational B-splines for curve and surface representation'. *IEEE Computer Graphics and Applications.* Vol. 3, No. 6, pp. 61 - 69.

Tsoubelis D. (1985). *'Remote sensing: Interpolation and contouring manipulations of images taken by satellites'* (in Greek). Dissertation in Astronautics. Athens School of Mathematics, University of Athens, June 1985.

Ullner M.K. (1983). 'Parallel machines for computer graphics'. *PhD. Dissertation.* California Institute of Technology, Pasadena, CA. Reference: 5112:TR:83.

Vassberg J. and Dailey K. (1990). 'AIRPLANE: Experiences, Benchmarks and Improvements. *American Institute of Aeronautics and Astronautics Aerospace Sciences Meeting,* Paper AIAA-90-2998, Portland, OR, August 20-23, 1990.

Versprille K. (1975). 'Computer Aided Design Applications of the Rational B-spline Approximation form'. *PhD thesis.* Syracuse University, Syracuse, New York.

Voronoi G. (1908) 'Nouvelles applications des paramétres continus à la theorie des formes quadratiques, Deuxième Mémoire, Recherches sur le parallélоèdres primitifs. *J. reine angew. Math.* 134, pp. 198 - 287 136, pp. 67 - 178.

Voronoi G. (1909) 'Nouvelles applications des paramétres continus à la theorie des formes quadratiques, Deuxième Mémoire, Recherches sur le parallélоèdres primitifs. *J. reine angew. Math.* 136, pp. 67 - 178.

Voss R. (1985). 'Random Fractal Forgeries'. *Fundamental Algorithms for Computer Graphics* R.A. Earnshaw (ed.). Springer-Verlang, Berlin, pp. 805 - 835.

Wallin Å (1991). 'Constructing Iso-surfaces from CT Data'. *IEEE Computer Graphics and Applications*, Vol. 11, No. 6, November 1991, pp. 28 - 33.

Ward (1994). 'The RADIANCE Lighting Simulation and Rendering System'. *ACM SIGGRAPH '1994.* Orlando, Florida, July 24 - 29, 1994. Computer Graphics proceedings, Annual Conference Series, pp. 459 - 472.

Watson D. (1981). 'Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes'. *The Computer Journal*, Vol. 24, No. 2, pp. 167 - 172.

Weghorst H., Hooper G. and Greenberg D. (1984). 'Improved Computational Methods for Ray Tracing'. *ACM Transactions in Computer Graphics.* Vol. 3, No. 1, January, pp. 52 - 69.

Whitted, T. (1980). 'An Improved Illumination Model for Shaded Display.' *Communications of the ACM*, Vol. 23, No. 6, June, pp. 343 - 349.

Wijk Van (1984). 'Ray tracing objects defined by sweeping a sphere'. *Eurographics '84.* pp. 73 - 82, Copenhagen. Also reprinted in *Computer Graphics*, Vol. 3, No. 9, pp. 283 - 290.

Williams D.R., Collier R. (1983). 'Consequences of spatial sampling by a human photoreceptor mosaic'. *Science.* Issue 221, 22 July 1983, pp. 385 - 387.

Wolberg G. (1994). '*Digital image Warping*'. IEEE Computer Society Press Monograph, 3rd edition, Los Alamitos, California.

Wolfram S. (1991). *Mathematica: a System for Doing Mathematics by Computer.* Second edition, Addisson-Wesley, 1991.

Woodwark J.R. (1984). 'Compressed quad-trees'. *The Computer Journal.* Vol. 27 No. 3, pp 193 - 288.

Woodwark J. (1986). 'Blends in Geometric modelling'. *Proceeding of the 2nd IMA conference on the Mathematics of surfaces.* Cardiff, September 1986.

Wyszecki G. and Stiles W. (1982). '*Color Science: Concepts and Methods, Quantitative Data and Formulae*', 2nd edition, Wiley, New York.

Wyvill G., McPheeters C., Wyvill B. (1986a). 'Soft Objects. Advanced Computer Graphics'. Proceedings of *Computer Graphics, Tokyo.* 1986, pp. 113 - 128.

Wyvill G., McPheeters C., Wyvill B. (1986b). 'Data structures for soft objects'. *The Visual Computer*, Vol 2, pp. 227 - 234. Springer Verlag, 1986.

Wyvill G., McPheeters C., Wyvill B. (1986c). 'Animating soft objects'. *The Visual Computer*, Vol 2, pp. 235 - 242. Springer Verlag, 1986.

Xiang Z. and Joy G. (1994). 'Color Image Quantization by Agglomerative clustering'. *IEEE Computer Graphics and Applications*, Vol. 14, No. 3, May 1994, pp. 44 - 48

Yellot J.I.Jr. (1983). 'Spectral consequences of photoreceptor sampling in the Phesus retina'. *Science.* Issue 221, 22 July 1983, pp. 382 - 385.