Isosurface Modelling of *Soft* Objects in Computer Graphics

Craig William McPheeters, B.Sc.

Thesis submitted to the Council for National Academic Awards
in partial fulfilment of the requirements for the degree of
DOCTOR of PHILOSOPHY

March 1990

National Centre for Computer Animation
Department of Communication and Media
Dorset Institute

Isosurface Modelling of *Soft* Objects in Computer Graphics

Craig William McPheeters

ABSTRACT

There are many different modelling techniques used in computer graphics to describe a wide range of objects and phenomena. In this thesis, details of research into the isosurface modelling technique are presented. The isosurface technique is used in conjunction with more traditional modelling techniques to describe the objects needed in the different scenes of an animation. The isosurface modelling technique allows the description and animation of objects that would be extremely difficult, or impossible to describe using other methods. The objects suitable for description using isosurface modelling are *soft* objects. *Soft* objects merge elegantly with each other, pull apart, bubble, ripple and exhibit a variety of other effects.

The representation was studied in three phases of a computer animation project: modelling of the objects; animation of the objects; and the production of the images. The research clarifies and presents many algorithms needed to implement the isosurface representation in an animation system.

The creation of a hierarchical computer graphics animation system implementing the isosurface representation is described. The scalar fields defining the isosurfaces are represented using a scalar field description language, created as part of this research, which is automatically generated from the hierarchical description of the scene. This language has many techniques for combining and building the scalar field from a variety of components. Surface attributes of the objects are specified within the graphics system. Techniques are described which allow the handling of these attributes along with the scalar field calculation. Many animation techniques specific to the isosurface representation are presented.

By the conclusion of the research, a graphics system was created which elegantly handles the isosurface representation in a wide variety of animation situations. This thesis establishes that isosurface modelling of *soft* objects is a powerful and useful technique which has wide application in the computer graphics community.

For my family,

Mom, Dad, Gordon, Daniel and Laura.

Acknowledgements

Many thanks are owed to the people who have helped me throughout this work. Among them are:

The people in the Computer Services Unit (past and present!) who have helped me in a number of ways during this project. In particular Chris Spice and Ron Burns for their efforts in loaning me a much needed Mac during the writing of this thesis. Also Kevin Harvey for his help with PostScript and John Stovold for general Mac help.

All of the research assistants at the Dorset Institute (also past and present!) including: Robert, Anne, Antony, Mark, John and Rebecca.

The graphics group have encouraged me during this project in a number of ways: Alan Rudge, Alex King and Peter Hardie. Also thanks to Mike Stapleton for his continued support.

A special thank you to Prof. Brian 'Blob' Wyvill for supporting and encouraging me to become involved in computer graphics during and after I completed my B.Sc at the university of Calgary. Without his initial and continued support I would not have considered embarking on what has become six years of working as a researcher in the computer graphics field.

Finally I would like to thank my supervisors Prof. Tim Wheeler and Prof. Peter Comninos. Their efforts in enabling this project, supporting and encouraging me throughout its length as well as their valuable and timely criticism of the chapters of this thesis has all been greatly appreciated.

Contents

Abstract i

Acknowledgements iii

List of tables viii

List of figures ix

1 Introduction 1

1.1 General introduction 1

1.2 Aims of the research 2

1.3 Contribution of the research 4

1.4 Overview of the thesis 5

2 Literature Review 6

2.1 Introduction 6

2.2 Modelling techniques in computer graphics 7

2.3 Isosurface modelling 12

2.4 Visualisation 16

2.4.1 Boundary representation 18

2.4.2 Volumetric rendering 21

2.4.3 Special purpose visualisation hardware 22

2.5 Animation 24

2.6 Conclusions 25

3 Specification of the scalar field 26
3.1 Introduction 26
3.2 Methods of describing scalar fields 28
3.3 Description of the scalar field components 32
3.3.1 Algebraic components 35
3.3.2 Procedural components 45
3.3.3 Empirical components 48
3.4 Methods of combining scalar field components 50
3.4.1 Avoiding discontinuities 50
3.4.2 Composition operators 54
3.5 A Scalar field description language 62
3.6 Optimisation techniques for scalar field evaluation 66
3.7 Conclusions 68

4 Visualisation of the isosurface 71
4.1 Introduction 71
4.2 Factors involved in finding an isosurface 73
4.2.1 Search volumes 74
4.2.2 Minimum detail size 76
4.2.3 Binary or range classification 81
4.3 Boundary representation 82
4.3.1 Polygonal representation 84
4.3.2 Polygonisation method 88
4.3.3 Alternative boundary representations 98
4.4 Volumetric representation 101
4.5 Hybrid representation 102
4.6 Ray-tracing 103
4.7 Conclusions 104

5 Isosurface appearance 106
5.1 Introduction 106
5.2 Colour 107
5.3 Translucency 122
5.4 The calculation of a surface normal 125
5.5 Texture mapping 129
5.6 Boundary representation related problems 134
5.7 Conclusions 136

6 Isosurface animation 138
6.1 Introduction 138
6.2 Representation of animation 139
6.3 Geometric transformation 140
6.4 Scalar component parameter animation 148
6.4.1 Surface attributes 149
6.4.2 Velocity 156
6.4.3 Bias 161
6.5 Metamorphosis 166
6.6 Interaction with traditional modelling techniques 170
6.7 Conclusions 173

7 Graphics system description 174
7.1 Introduction 174
7.2 Structure of the graphical models 175
7.3 Parametric animation 180
7.4 Isosurface models 184
7.4.1 Creation of an SFDL program 187
7.4.2 Visualising the isosurface 193
7.5 Implementation details 195
7.6 Conclusions 201

8 Further research 203
8.1 Introduction 203
8.2 Specification 204
8.2.1 Scalar components 205
8.2.2 Scalar operators 207
8.2.3 User interface 208
8.3 Calculation 209
8.4 Visualisation 211
8.5 Attributes 213
8.5.1 Generalised interaction 213
8.5.2 Disassociate from surface 214
8.5.3 Retentive attributes 215
8.6 Interaction 216
8.6.1 Interaction between components 217
8.6.2 Traditional modelling 217
8.7 Animation 218
8.8 Conclusions 219

9 Conclusions **220**

Appendix 1 Example software **225**

Appendix 2 Bibliography **257**

List of tables

3.1 Characteristics required of each scalar component 34
3.2 Initial quadratic coefficients for quadratic components 41
3.3 Table of quadratic formulae for cube 47
3.4 Scalar field composition operators 55
3.5 Associative and commutative classification of binary operators 61
3.6 Scalar field description language summary 65

7.1 Geometric transformation instructions available in JED 180
7.2 Attribute instructions available in JED 180
7.3 Operators available in an expression 182

List of figures

2.1 Translational sweep of a polygon 8
2.2 Example of two constructive solid geometry operations 11

3.1 Summation of two quadratic formulae 30
3.2 Minimum of two quadratic formulae 30
3.3 Varying values for *influence* on a sphere 39
3.4 Demonstration of effect of discontinuity when combining components 51
3.5 Addition of two spheres using first approximation to smoothing 53
3.6 Graph of two spheres using the final smoothing formula 54
3.7 Addition of two spheres 57
3.8 Subtraction of the left sphere, addition of the right 57
3.9 Minimum of two spheres 58
3.10 Maximum of two spheres 58
3.11 Mixture of two spheres 59
3.12 Mirror of a sphere around scalar value one 59
3.13 An expression tree and its three traversals 63
3.14 Grammar to describe SFDL 66
3.15 Pseudo code for sphere component 70

4.1 Pseudo code to find isosurface intersections along a line 77
4.2 Aliasing problem detecting isosurface intersections 78
4.3 Orthogonal lattice subdivided into cuboids 83
4.4 The sixteen possible topologies of a square 85
4.5 Ambiguous surface topology and possible heuristic 85

4.6 Unique isosurface topologies of a cuboid....86
4.7 Example of surface cracks caused by differing resolutions....88
4.8 Illustration of polygonisation method....89
4.9 Data structures used in searching through a volume....90
4.10 Pseudo code for searching a volume for isosurfaces....92
4.11 Vertex numbering of a cuboid....94
4.12 Data structure used to store surface topology....95
4.13 Intersection calculation algorithm....97
4.14 Data structure to store polygonal mesh....98

5.1 Implementation of colour in the addition operator....113
5.2 Implementation of colour in the subtraction operator....115
5.3 Implementation of colour in the maximum operator....116
5.4 Implementation of colour for the minimum operator....117
5.5 Implementation of colour in the mixture operator....118
5.6 Final adjustment of colour....119
5.7 A red, a green and a blue sphere merging....120
5.8 Red noise being added to a figure....120
5.9 A red and a green noise field added to a half space....121
5.10 A red and a green sine component being added to a half space....121
5.11 Demonstration of the problem of transparency and boundary representation....124
5.12 Implementation of translucency for the addition operator....125
5.13 Numerical method for calculating surface normals....129
5.14 Example implementations of texture....134

6.1 Merging and pulling apart of two spheres....141
6.2 Merging and pulling apart of the union of two spheres....142
6.3 Merging cylinders....143
6.4 Sphere passing through a cylinder....144
6.5 Small ellipsoid passing through a larger ellipsoid....145
6.6 Sphere passing through a plane....146
6.7 Noise being moved through an ellipsoid....147
6.8 Cyan sphere passing through a stationary red sphere....150
6.9 Positive negative interaction....151
6.10 Six colourful spheres merging into a black sphere....152
6.11 Increasing noise amongst spheres....153
6.12 Sine wave passing across a frame....154
6.13 Three textured spheres merging and interacting....155

6.14 Different values for *velocity* on a sphere....158
6.15 *Velocity* implemented on one moving sphere....159
6.16 *Velocity* implemented on two moving spheres....160
6.17 Example of an incorrect implementation of *bias*....163
6.18 Several examples of *bias* defined on a torus....164
6.19 A moving sphere with *velocity* and *bias* implemented....165
6.20 Metamorphosis between a frame and a torus....169
6.21 Interaction of soft slime and a traditionally modelled grating....172

7.1 Partial list of JED instructions to produce a car....178
7.2 Partial structure of a car....179
7.3 General algorithm of basic *plot* routine....188
7.4 Partial listing of the routines called by *plot* to display the car in figure 7.1 189
7.5 JED interface displaying the automatic approximations of the scalar components....197
7.6 JED interface displaying a line drawing of an isosurface model....197
7.7 JED interface displaying a full rendering of an isosurface model....198
7.8 JED interface displaying the real time animation playback facility....198

8.1 The standard, and an extended method of merging colour....215

Chapter 1

Introduction

1.1 General introduction

As the computer graphics field has matured over the 25 years since its inception, there has been an increasing number of techniques available to model objects and phenomena. Initially, man made objects were the focus of a majority of the research in the computer graphics field. This was done in order to expand and enhance the industrial uses of computer graphics through applications such as computer aided design (CAD) and product visualisation. Techniques were quickly developed to allow the description of a variety of man made objects. Some researchers in the field were excited by the possibilities of describing a wider range of objects, and from this interest many new modelling techniques have been developed.

There is at present ongoing research advancing the modelling of objects and phenomena in practically all situations. The industrial uses of computer graphics continues to expand as the techniques which are used to describe objects become more general, and as the awareness of the applicability of computer graphics to many fields increases.

Each of the different techniques for modelling objects in computer graphics may have several advantages and disadvantages. Among the advantages

may be: ease of the design process; flexibility of design; efficiency of evaluation; simplicity of algorithms; quality of representation and others. Disadvantages may be: difficulty of designing general shapes; large data sets; lack of interaction; lack of intuitiveness; difficulty of editing and many others. There is not one representation in computer graphics that is best in all situations. A representation used for designing cars may be inappropriate for describing a tree.

Until recently, the majority of research in computer graphics was concerned with objects that did not move, or moved using simple motions. Objects which changed shape as they moved and phenomena for which movement is as important as the shape have been ignored until a few years ago. Two of the reasons for this are the wide range of research topics and effort in industrial applications, as well as the relatively low computing power that has been available for solving these problems. Some of the more recent techniques would not have been considered using the low computing power available from the computers used in the middle 1970's and early 1980's.

As computer graphics matures and is generally being used in a wider range of applications, there is an opportunity to invest research effort to new applications which may have future benefit. One such modelling technique is the use of isosurfaces defined within scalar fields to describe objects and their motion. This modelling technique has many features which make it an attractive representation for certain objects. For example, the isosurface modelling technique is able to easily create objects with many curved surfaces as well as objects which merge and pull apart. The isosurface modelling technique is able to describe many types of objects, examples of which are presented in the remainder of the thesis.

1.2 Aims of the research

The research described in the thesis has been undertaken with the aim of developing the isosurface modelling techniques into a robust and useful method for modelling objects in computer graphics, as well as determining the methods strengths and weakness'. There are a wide variety of objects able to be described using this modelling technique. Some of them are:

water; mud; curved objects; objects which metamorphose; objects which distort; characters in animations; bubbling objects; rippling objects; and many more. It is expected that isosurface modelling, once it is developed into an intuitive and easily used modelling technique, will be applied to the description and animation of a diverse range of objects. The types of objects for which the method are unsuitable will be determined.

Along with the advances made by this research to the isosurface modelling technique, the process of incorporating it into a computer animation system have also been studied. This was undertaken in order to understand the constraints and demands that may be imposed at that stage of implementation.

'Isosurface modelling' is a phrase applied to a collection of algorithms and data structures which combine to operate upon isosurfaces defined within scalar fields to allow the description and animation of a range of objects. These algorithms, when incorporated into a computer graphics animation system allow the description of a variety of objects, possibly in conjunction with some of the more traditional modelling approaches.

A wide range of research is available to be undertaken on various aspects of the isosurface modelling technique. The range of topics available and amount of research that is possible within the modelling technique required that the scope of this research be carefully defined. Although there are many interesting topics within the area, there is enough potential research to consume a lifetimes of work. As this research is concerned mainly with developing isosurface modelling into a robust and useful modelling technique, a suitable subset of the available research has been selected to achieve this aim. The selection of the topics to be studied in detail was accomplished during the first phase of this research. This selection was more or less adhered to during the project. Many avenues for further research were discovered, and are presented in chapter eight as topics for further research.

1.3 Contribution of the research

The first phase of this research was to review and detail the existing methods of approaching the isosurface representation of objects in a computer animation system. Through this review it became apparent that in some cases generality had been sacrificed in order to solve specific problems, or meet certain criteria of computer animation systems or applications. These cases will be discussed in the thesis. This loss of generality was restricting the range of objects and phenomena that could be represented using this technique. While these restrictions may be useful in certain circumstances, one of the goals of this research was to keep the techniques as general and flexible as possible. This required that some of the previous assumptions could not be made, new solutions to old problems had to be found in some cases. Within this new context, the circumstances when assumptions can be made and optimisation used become apparent.

The research has been broken into four major areas: the specification of the scalar field; visualising of the isosurface; the isosurface appearance; and animation of the isosurface. Each of these areas were studied and are presented in detail in this thesis. In each of the areas it was found that some specialised support was needed in the computer animation system, this is presented along with more general information in a section discussing the computer graphics system developed during this research.

In each of the four major areas there are contributions from this research to the isosurface modelling field, due in part to the goal of keeping components of the isosurface technique general and flexible. Images and animations have been produced, which to the knowledge of the author, have not been produced before and would be almost impossible to generate using more traditional techniques. Examples of these pictures are presented throughout the thesis.

The algorithms required in the implementation of isosurface modelling are discussed in this thesis. These algorithms are presented in sufficient detail to allow the implementation of most of the results of this research elsewhere. During this research a decision was made to use the 'C' programming language, although the algorithms will be presented in a

pseudo-code that should be easily translated to any block structured language.

1.4 Overview of the thesis

The thesis is divided into nine chapters and two appendices. A review of the literature pertinent to the research is presented in chapter two. The five major areas of this research are presented in chapters three to seven. Chapter three contains the topics related to the specification of the scalar field. Chapter four discusses visualising the isosurface. That chapter is mainly concerned with finding the isosurface in the scalar field, and determining its shape. Chapter five discusses the appearance of the isosurface. The appearance can be affected by factors such as colour and translucency, among others. Chapter six discusses issues related to isosurface animation. There are many possible types of isosurface animation, along with many techniques for accomplishing them. The demands imposed upon the animation system to support isosurface modelling, as well as details of the animation system developed during the course of this research are presented in chapter seven. During the research an effort was made to keep the work focused, thus many avenues of further research were discovered, but of necessity not explored. Chapter eight presents some of the possibilities for further research. The last chapter, nine, presents the overall conclusions found by the research.

In order to encourage the spread of the isosurface modelling technique, the first appendix presents 'C' code which solves a variety of visualisation tasks for both two and three dimensions. These pieces of code can be used in conjunction with the contents of this thesis to explore the isosurface modelling technique. The second appendix contains the bibliography.

Chapter 2

Literature review

2.1 Introduction

This literature review is presented in four main sections. In each of the sections previous related research is reviewed in order to establish the general context in which this research project is set.

The four sections are:

1) A general review of the modelling techniques available in computer graphics. This general review is undertaken in order to establish the need for a technique with the capabilities of isosurface modelling in the description of many models. Existing modelling techniques are unsuitable for the description of the objects presented in this thesis.

2) A review of the specific area of isosurface modelling. This review builds on the previous general review in order to establish the context in which the results of this research into isosurface modelling are set. The earlier work in isosurface modelling is outlined.

3) Visualisation. A wide variety of techniques have been developed within computer graphics for the visualisation of computer graphic

models. The focus of this section is on the visualisation of isosurface models.

4) Animation. Many complementary methods for describing an animation sequence have been developed. These animation techniques are briefly presented. Extensions to the capabilities of computer generated animation are available in conjunction with isosurface modelling. This potential is highlighted.

The overall purpose of this literature review is to describe the area of computer graphics in which the isosurface modelling technique is located, in order to highlight the contributions of this research project to the earlier research in the field.

2.2 Modelling techniques in computer graphics

Modelling techniques in computer graphics are used to represent a wide range of objects and phenomena. There are many modelling techniques available, due to the various strengths and weakness' of each technique in representing different types of objects. A modelling technique which is able to easily describe one type of object may be unsuitable for a different type.

Modelling techniques in computer graphics may be broadly classified into two major groups, general modelling techniques and specific modelling techniques. General modelling techniques are useful when applied to a wide variety of situations. For example, a general technique which is used to describe a car may also be applied to create a description of a house. A specific modelling technique is a technique that is created in order to solve a particular modelling problem. For example a modelling technique designed to create a variety of trees may not be easily applied to alternative modelling situations.

One of the most widely used modelling primitives in computer graphics is a polygon. A majority of the alternative modelling techniques in computer graphics can ultimately be defined in terms of a number of polygons. Polygons are a general and flexible modelling primitive in computer graphics. A major weakness associated with the polygon as a modelling primitive is the amount of time needed to create any but the most simplistic

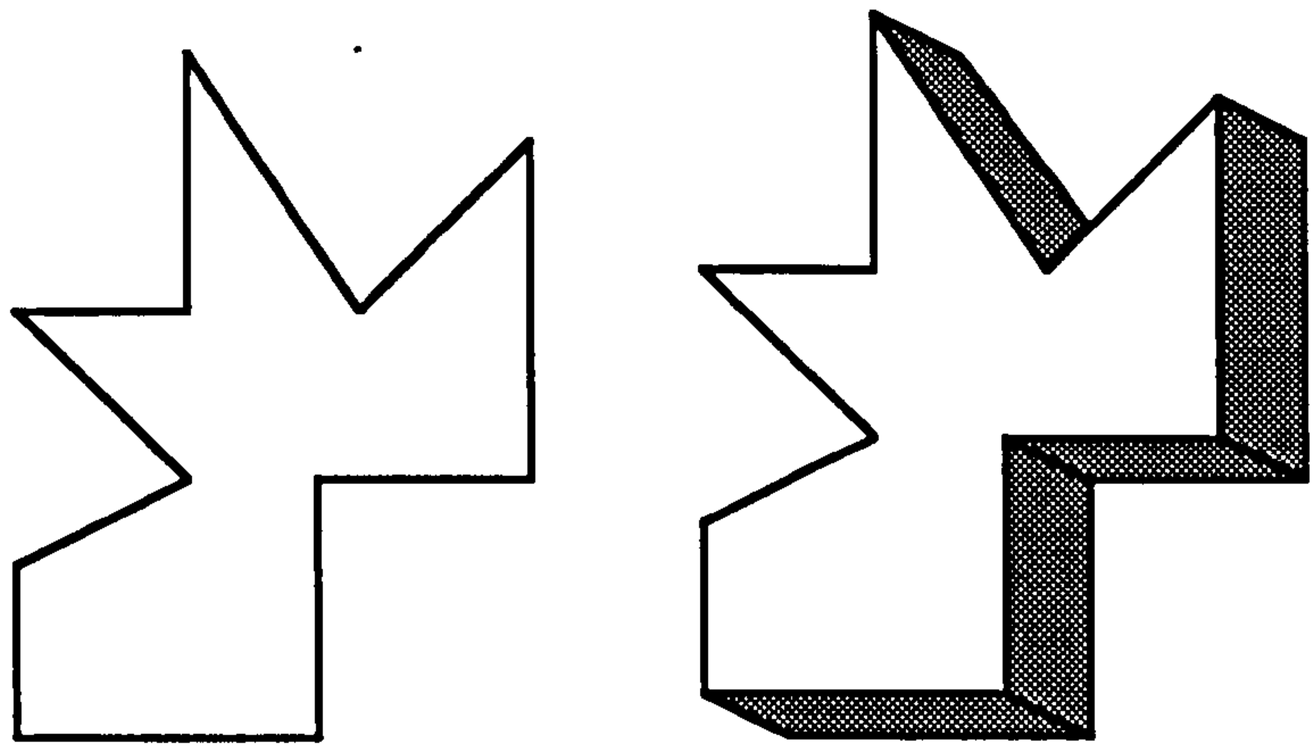


Fig 2.1 Translational sweep of a polygon.

of models when using it. A polygon is not sufficiently 'high level' to allow for easy specification of complex models. For example in order to create even a simple cylinder, each of twenty or so polygons must be specified accurately. This is a time consuming process.

A method which is used to increase the utility of the polygon as a modelling representation is the polygonal sweep. Polygonal sweeps are created by sweeping a polygon through space and creating the boundary of the swept volume. A polygon representing a two dimensional shape can be swept through space to create a three dimensional version of such a shape. There are typically three types of polygonal sweeps available: translational sweeps; rotational sweeps; and generalised sweeps. A translational sweep is used to translate a polygon along a vector, creating polygons along the boundary of the volume swept. This is the technique used in creating figure 2.1. A rotational sweep of a polygon is used to rotate a polygon around an axis by a specified angle, again creating the boundary of the volume swept. The third type of polygonal sweep, a generalised sweep, can be implemented in a number of ways. In a paper by Shani and Ballard in 1984, a form of generalised sweep called a generalised cylinder is discussed. These generalised cylinders are specified by defining a number of polygonal templates along a space curve. The outlines of the polygonal templates are then connected through a process of polygonal interpolation along the space curve specified.

The polygonal techniques described above are not without their weakness', two of these are the silhouette edge problem and the difficulty of producing a complex curved shape. The silhouette edge problem is caused by the fact that a number of straight lines are used to represent a smooth curve when

using the polygonal representation. This is evident in the polygonal representation of a sphere. If an insufficient number of polygons are created in representing the sphere, the silhouette will contain visible straight lines. In order to solve this problem more polygons must be created, however these extra polygons are only necessary for a close-up view of the sphere, which allows the missing edge detail to be seen. Creating a large number of polygons to be used in all views of a sphere will be inefficient in some circumstances. A solution to this problem has been suggested which involves the use of multiple model templates [Clark 1976; Crow 1982]. The second problem mentioned above with the polygonal representation is related to curved surfaces, such as those present in a model of a car. Curved surfaces are difficult to produce solely using polygonal techniques. Alternative modelling methods have been found which solve these problems.

A modelling technique which solves the silhouette edge problem and allows the specification of curved surfaces is the parametric mesh [Newman and Sproull 1979 pp.309-330; Foley and Van Dam 1982 pp.523-536]. A parametric mesh is composed of many parametric patches, which combine to create the surface of the model desired. A parametric patch is able to accurately represent a curved surface. A parametric patch may be either viewed directly, as it is often in ray tracing, or converted to a polygonal representation. As a parametric patch represents a curved surface, a sufficient number of polygons can be created to avoid the silhouette edge problem. Problems that exist with the parametric mesh representation include the difficulty of specifying non-rectangular patches and the problem with guaranteeing that patches meet correctly when translated into polygons at different levels of detail.

Parametric meshes allow for the creation of a large number of models that would have been extremely difficult using only the polygonal and polygonal sweep representation. Both parametric meshes and the polygonal based representations are useful in a graphics system. The parametric mesh does not supersede the polygonal methods.

The modelling techniques discussed above only describe the surfaces of models. This leads to an additional classification of modelling techniques into boundary representations and volumetric representations. A boundary representation of a model only specifies the surface. For instance a polygonal representation of a cube requires six square polygons which are

arranged to resemble the cubes outside surface. A volumetric representation of a cube represents the volume occupied by the cube. Each of the different techniques has separate strengths and weakness'. Boundary representations are generally visualised more quickly than volumetric representations. Volumetric representations generally have more information available to solve problems such as finding a centre of gravity and determining if two models interpenetrate.

A technique which does not use the boundary representation and is useful for describing a number of shapes is the quadratic formulation [Barr 1981]. Unlike the previous boundary representations, the quadratic representation defines a shape using a mathematical expression. A number of shapes can be described using the quadratic formulation, for instance: sphere; ellipsoid; cylinder; plane; paraboloid; and hyperboloids. More shapes are available. As shapes are described using the quadratic formulation mathematically, there is no difficulty in avoiding the silhouette edge problem.

The techniques discussed above are useful in creating a wide variety of models. Many applications require, however, that a representation of an object closely resemble reality, acting as an aid in the design and manufacturing processes for the object or product. A number of computer aided design (CAD) and computer aided manufacturing (CAM) packages have been developed. These systems are often based on a constructive solid geometry (CSG) approach for representing the objects. A review of many CSG and CAD/CAM systems was written by Requicha in 1983.

A CSG system differs from the previous polygonal and parametric mesh representation in that a number of primitives are supplied by the system which are combined using set theoretic operations to create the objects desired. Set theoretic operators such as union, intersection and difference are typically available. An example of a CSG operation is shown in figure 2.2. The CSG approach to modelling is a powerful method of describing specific objects in computer graphics. The objects most suitable for representation using CSG techniques are man made objects, such as those found in many engineering disciplines for example. While many CSG systems represent objects internally in volumetric representations, often they are converted into a boundary representation for visualisation.

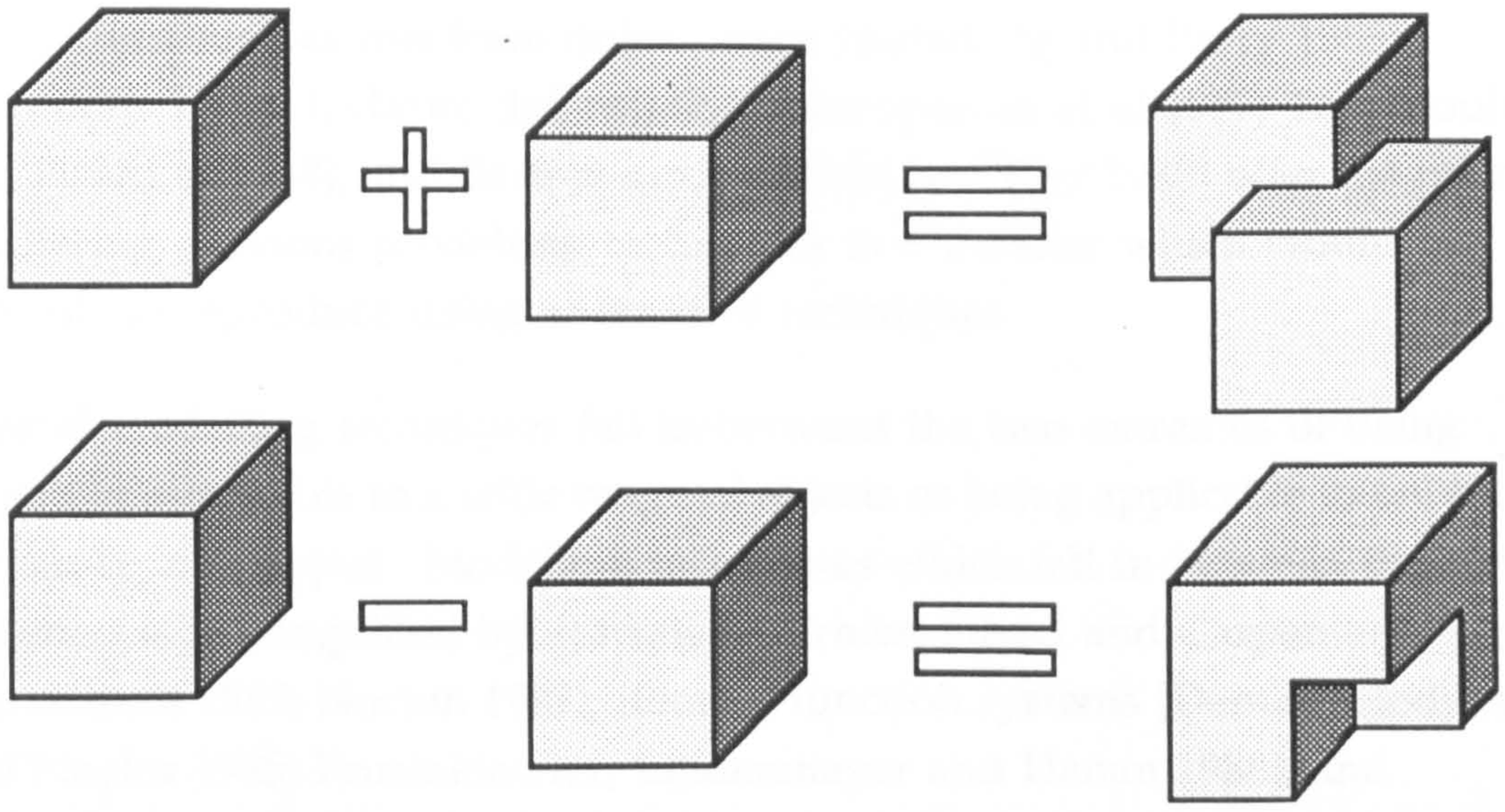


Fig 2.2 Example of two constructive solid geometry operations

The modelling techniques described above (polygonal, parametric mesh, algebraic and CSG) are used to create a majority of the objects in computer graphics. The techniques mentioned can be generally applied to a wide range of modelling problems in computer graphics.

There are a growing number of modelling techniques which sacrifice the generality of being easily applied to a wide range of different objects in order to gain the ability to easily describe specific types of models with great accuracy and flexibility.

An example of a modelling technique applied to a specific situation is the work on the modelling of trees [Aono and Kunni 1984; de Reffye et al 1988; Viennot et al 1989]. If more general modelling technique were used, a great deal of time would be required to produce the models presented in, for instance, Aono and Kunni's paper. The tree modelling techniques have a number of parameters that can be changed to easily create a large number of differently shaped trees.

There have been several research projects which study computer models of faces [Parke 1982; Waters 1987]. The models incorporate the ability to alter the expression of a face, such as happiness or shock, as well as altering the actual facial structure, for instance to change the size of the nose. Achieving similar results using traditional modelling techniques would be difficult.

Techniques such as free form deformation [Sederberg and Parry 1986; Comninos 1989], inelastic deformation [Terzopoulos et al 1987; Terzopoulos and Fleischer 1988], and flexible models [Platt and Barr 1988] offer the means of altering previous modelling techniques in a manner which would be difficult to reproduce using alternative techniques.

Several modelling techniques fall in-between the two extremes of being generally applicable to a wide range of objects or being applicable to only a certain type of object. Modelling techniques which fall in-between these two extremes are exemplified by: fractals [Fournier, Fussel and Carpenter 1982; Mandelbrot 1983; Norton 1982]; iterated function systems [Demko, Hodges and Naylor 1985; Prusinkiewicz, Lindenmayer and Hanan 1988]; and particles [Reeves 1983].

The isosurface modelling techniques fall into this last category. They have not been created to solve any one particular modelling problem and are yet more specific than the polygonal or parametric mesh representation. Objects easily represented using the isosurface technique include: mud; water; jelly; scientific phenomena, including some medical applications; objects which can bend or are pliable; and objects with many curved surfaces. Examples of some of these objects are presented in the following chapters.

The isosurface modelling techniques are not intended to replace an existing modelling technique. It is intended that the development of the isosurface modelling technique will allow the description of models that would be difficult without it.

2.3 Isosurface modelling

The phrase 'isosurface modelling' is used throughout this thesis to represent the modelling technique studied during this research project. Alternative phrases have been used in the past to describe similar and related research, phrases such as: 'equipotential surface' [Quarendon and Woodwark 1987; Nishimura et al 1985]; 'iso-valued surface' [Wright and Humbrecht 1979]; 'iso-valued contour surface' [Levoy 1988]; 'isosurface' [Bloomenthal 1987; Mandelbrot 1975; Pasko and Pilyugin 1988; Wyvill, McPheeters and Wyvill

1986]; 'implicit surface' [Blinn 1982; Bloomenthal 1987; Woodwark 1988; Wyvill and Wyvill 1989]; 'contour surface' [Zyda and Walker 1988]; '3D contouring' [Rosenfield et al 1984]; 'soft objects' [Wyvill, McPheeters and Wyvill 1986]; and meta-surface [Nishimura et al 1985]. The phrase 'isosurface modelling' indicates the modelling technique based upon iso-valued surfaces defined within scalar valued fields. The fact that it is a modelling technique indicates that it is possible to construct models using the representation, this is in contrast to many applications of iso-valued surfaces defined within scalar valued fields that are solely a means of visualising the contents of the scalar field [Levoy 1988; Trousset and Schmitt 1987].

The phrase 'isosurface' is an abbreviation for 'iso-valued surface.' This means literally 'equal valued surface.' The type of value being measured is dependent upon the application. Many two dimensional examples are available, a few of them are: isobars, these are used to display lines of equal barometric pressure, as shown daily in weather reports; isotherms, these are lines on maps connecting places of equal temperature; and isobaths, these are lines on maps connecting points of equal underwater depth. In each of these techniques, first a particular value of interest is chosen and then the appropriate contour lines are found. A different value being chosen would result in a different contour line being found. An isosurface is the three dimensional equivalent of isocontours. This is explained in more detail in chapter three.

The phrase 'isosurface modelling representation' is also used throughout this thesis. This phrase is intended to refer to the actual data representation and the representation of models within the isosurface modelling technique. It is possible for a particular isosurface modelling technique to be unsuitable for a particular application while the more general isosurface modelling representation is suitable.

There have been several previous research efforts directed to the area that is loosely collected under the phrase 'isosurface modelling.' One of the first to publish results of research in the area was Blinn in 1982. Blinn created a technique that was designed specifically to model electron density maps using an isosurface representation of the surfaces. Although the research was directed at a specific task, Blinn recognised that the technique may be more widely applied:

> 'The algorithm was created to model electron density maps of molecular structures, but it can be used for other artistically interesting shapes.'
>
> [Blinn 1982 p235]

Although the technique developed by Blinn was intended to be used only with spherical primitives, he explains that more general primitives are possible, and an example of this was shown. The paper describes a method of efficiently handling a large number of spherical primitives.

Three projects which were engaged in research in the isosurface modelling field without knowledge of each other were: the group working on meta-balls in Japan [Nishimura et al 1985]; the group working on *soft objects* [Wyvill, McPheeters and Wyvill 1986]; and Bloomenthal [Bloomenthal 1987].

Of the three groups, the work on meta-balls has provided the most spectacular images. The images produced by that group demonstrate the potential of the isosurface modelling technique when applied to complex, realistic models. The group involved in the research on meta-balls have said of their work:

> 'In this work, we have aimed to develop a method to create the shape and color of objects interactively in computer graphics, as in clay work and airbrushing.'
>
> [Nishimura et al 1985 p11]

In the meta-balls technique, a file is created which contains a list of meta-primitives. Associated with each meta-primitive is a list of attributes such as the location and orientation of the primitive, the strength of the primitive, its colour as well as other values. This description forms the basis of the scalar field from which the isosurface is extracted. A ray-casting approach is used for visualisation. The only meta-primitive described in the paper is the ellipsoid, although the possibility of including more is discussed.

The research on *soft objects* by the group based at the university of Calgary has resulted in the production of two research films to date, 'Soft' in 1986 and 'The Great Train Rubbery' in 1988. Both of these films have been

shown at numerous conferences, including SIGGRAPH. A number of papers have been written presenting the work on *soft objects* [Wyvill, McPheeters and Wyvill 1986,1987; Jevans and Wyvill 1988; Wyvill and Wyvill 1989].

In the work on *soft objects*, the isosurface modelling technique has been incorporated into a hierarchical graphics system which has been used to support a variety of research projects, called Graphicsland [Wyvill, McPheeters and Garbutt 1986]. A sphere was used as the initial primitive from which isosurface models are created, this has since been extended to include ellipsoids and superellipsoids [Wyvill and Wyvill 1989]. Each of the primitives are either added or subtracted to the scalar field. A polygonisation technique is described for extracting an isosurface from the scalar field.

The work by Bloomenthal in the isosurface modelling area is presented in two papers [Bloomenthal 1987; Bloomenthal 1988]. A technique for extracting an isosurface from a scalar field is described in detail in these papers. Several types of primitives for the modelling process are proposed and demonstrated, including one called a horned melon. A general method is not proposed for combining many components in the modelling of isosurfaces.

The above three research projects (meta-balls, *soft objects* and Bloomenthal) have concentrated mainly on the application of isosurface modelling to entertainment applications within computer graphics. At the time the above groups were publishing the entertainment applications of isosurface modelling, concurrent work was ongoing in other areas with some overlapping results. The work by Tindle in 1986 on fermi surface display discovered a technique which is related to that used in *soft objects* and by Bloomenthal for the visualisation of isosurfaces.. The work by Tindle was not on a modelling technique, but was rather purely a technique for visualising one type of isosurface, namely fermi surfaces.

There has been a lot of research conducted lately in the visualisation of scientific data. A majority of these techniques are used purely as methods of visualisation, not as modelling techniques. These will be discussed in more detail later.

Techniques which incorporate some of the principles of isosurface modelling in the description of texture volumes have been presented recently [Kajiya and Kay 1989; Perlin and Hoffert 1989]. These techniques are not surface modelling techniques but rather are used to add texture volumes to objects which are created using a variety of techniques. The techniques presented in this research would enhance the methods of creating texture volumes, just as texture volumes would increase the realism of selected isosurface models.

Amid this background of related research, there have been four main projects that are directly related to the area studied in this research: the work by Blinn; the work on meta-balls; the work on *soft objects*; and the work by Bloomenthal.

2.4 Visualisation

Remarkable progress has been made on the visualisation of objects within computer graphics. Advances have been made in both the hardware to display images and the software to produce images. Early images in computer graphics were displayed using printers or pen plotters. Advances such as the direct view storage tube enhanced the interactivity of computer graphics without significantly enhancing the image quality over that offered by pen plotters. With the technological advances offered by raster based display systems it became feasible to advance the realism of objects portrayed in computer graphic images.

Early attempts of modelling the effects of lighting on objects in computer graphics used a crude model for the interaction of light on a surface. Among other advances, improvements in lighting calculations for surfaces increased the quality of the images available. A review of lighting models and shading techniques was written by Hall in 1986.

There has been a steady increase of the realism of images produced on computers with the result that presently it is possible, in some instances, to produce images which are indistinguishable from a photograph of an actual scene [Cohen and Greenberg 1985].

A survey of the different techniques used in processing a model to produce an image was presented by Sutherland, Sproull and Schumacker in 1974. A considerable number of advances have been made since that time. As this literature review is primarily concerned with the general context of the computer graphics field, all of the advances will not be mentioned. Among some of the notable advances to realism in computer graphics are: the calculation of soft shadows [Amanatides 1984; Lee, Redner and Uselton 1985; Ward, Rubinstein and Clear 1988]; radiosity calculations [Nishita and Nakamae 1985; Cohen and Greenberg 1985]; solid texturing [Peachey 1985; Perlin 1985]; and texture volumes [Kajiya and Kay 1989; Perlin and Hoffert 1989] for example. Some of these advances have been incorporated into a visualisation standard proposed by Pixar, called the RenderMan[1] interface. The RenderMan interface uses a sophisticated format for the specification of photo realistic images which may be produced by many different applications.

The ability of obtaining photo realistic renderings of models is dependent upon many factors, one of which is that the behaviour and specification of the modelling techniques used in the model are well researched. The isosurface modelling techniques are relatively young and under developed in comparison to some of the other modelling techniques, polygons, parametric patches and quadratic surfaces for example. Additional research need be done in several areas in order to achieve the same degree of success with rendering isosurface models as has been achieved with polygonally based and parametric mesh models. As will be shown, the research presented in this thesis makes several contributions to the isosurface modelling area.

An area that has recently been defined, 'visualisation in scientific computing', or more simply 'scientific visualisation' [McCormick, Defanti and Brown 1987] incorporates a diverse set of visualisation techniques and proposes that standards be established to aid in the visualisation of scientific data. Methods of visualising scientific data can aid in the process of understanding the complex relationships which may exist within the data.

One of the fundamental problems in visualising an isosurface is in choosing which representation the surface shall take. Two main techniques are

---

1 RenderMan is a trademark of Pixar.

available for visualising isosurface models: processing of the scalar valued field to obtain a boundary representation of the isosurface; and direct viewing of a scalar field using volumetric techniques.

2.4.1 Boundary representation

With the introduction of X-ray computed tomography (CT) scans in 1970 [Fitzgerald 1989 p.26] two dimensional planar data sets were generated which represent a three dimensional volume. The display of this three dimensional volume was initially accomplished through the display of each of the two dimensional planar slices. This display technique demanded that the viewer imagine the relationship between the images and mentally construct a three dimensional representation of any features in the images.

Techniques were developed to improve this process, allowing three dimensional shapes to be displayed. The techniques proposed by Christiansen and Sederberg in 1978 suggested that contours be drawn on each two dimensional slice and be connected in a semi-automatic fashion by the algorithm described in their paper. The algorithm operates in conjunction with an attendant who is required to solve ambiguous circumstances which the algorithm can not handle. Several references to contouring techniques of this sort are available [Cook 1981; Ganapathy and Dennehy 1982; Boissonnat 1985]. Using such techniques, the features of interest in each planar slice are extracted, connected together and displayed as a three dimensional model. The boundary representation depicts a subset of the information available in a medical (or other) data set. More sophisticated techniques are now available for extracting three dimensional boundary representations from medical datasets [Lorensen and Cline 1987].

In 1979, Wright and Humbrecht developed a method of displaying isosurfaces using three dimensional contours. Although their method is well suited to a hidden line removal display, it does not contain enough topological information to allow a hidden surface display. A method of creating the three dimensional display is described:

> '... This is done by contouring two-dimensional subsets of the three dimensional array and suppressing the invisible

> parts of the contours. The union of all the contour line parts approximates the desired surface. ...'
>
> [Wright and Humbrecht 1979 p.182]

An alternative boundary representation of an isosurface is the dot display, first developed in 1981 by Connolly. A dot display has been used in the molecular modelling field for the display of models incorporating isosurfaces [Connolly 1983; Rosenfield et al 1984; Max and Getzoff 1988]. A dot display is a method of representing an isosurface in which a number of dots are located on an isosurface. The shape of the entire surface is then mentally completed by imagining the surface which the dots represent. Dot displays of complex surfaces may become ambiguous and difficult to interpret.

In 1982 Blinn proposed a method of visualising electron density maps which involved the display of implicit surfaces. He described a ray casting approach which handles a large number of spherical density clouds from which the isosurface is obtained. One of the problems Blinn solved was in the calculation of an intersection between a ray and an isosurface. The spherical density clouds are modelled using Gaussian bumps, which involve the use of the exponent operation. Although the intersection calculation is elegantly solved for the particular circumstances proposed in his paper, the method is not extensible to more general scalar fields such as those proposed in this research.

Work done on the meta-balls technique by Nishimura in 1985 used a slightly different approach than that of Blinn. In the method proposed in the meta-balls research, a method of finding the intersection of a ray with an isosurface was proposed which is quicker than that found by Blinn. Similarly to the work by Blinn, the meta-balls technique is not easily extended to scalar field components other than those developed in the research, which was mainly ellipsoids.

In 1986 two separate bodies of research reported findings with some overlap. Tindle published a short report (1 page) describing a technique used to display fermi surfaces. The technique used a method of extracting an isosurface from a scalar field resulting in a polygonal representation. The technique was similar to, although not as comprehensive as that published by Wyvill, McPheeters and Wyvill.

Two articles were published by Wyvill, McPheeters and Wyvill in 1986, together in a single issue of 'The Visual Computer.' The articles describe a modelling technique named *soft objects*. The *soft object* modelling technique is based on isosurfaces defined in scalar fields. The data structure involved in the visualisation process as well as the techniques and capabilities of the animation of *soft objects* were discussed. The technique produces a polygonal representation of an isosurface through various isosurface locating and tracking methods. The scalar fields from which the isosurfaces are extracted are composed mainly of quadratic representations of ellipsoids which are added together to obtain a field value. Negatively valued ellipsoids are also available, which create dents or depressions in the isosurface.

In the medical field, research published by Lorensen and Cline in 1987 proposed a technique which was similar in a number of ways to that proposed by Wyvill, McPheeters and Wyvill as well as Tindle for the creation of a boundary representation of an isosurface. A polygonal representation of an isosurface is extracted from data obtained from medical scanning equipment to create representations of the structure of internal organs, bones and tissue. Startling images are presented in this research of a number of surfaces extracted from a single scalar field data set of a human head. Depending upon which isosurface is extracted, a surface of skin, bone or brain tissue can be produced.

Two papers by Bloomenthal, one in 1987 and one in 1988, discuss various aspects of the implicit modelling technique. An elegant method of searching for, tracking and polygonising an isosurface is proposed. He uses adaptive space division techniques incorporating a set of heuristic rules to find the correct isosurface. An initial search volume is specified which must contain the isosurface, any portion of the isosurface not enclosed within this volume will not be found by his algorithm. A similar constraint is placed upon the visualisation technique proposed in this research. This will be explained in more detail in chapter four.

Research by Quarendon in 1987 on displaying scalar field data in WINSOM [Quarendon 1984] resulted in five different techniques being used for the display of scalar field data within their system:

'1. as a density cloud
2. by generating equipotential surfaces in the field
3. by colour mapping the field values onto a surface
4. by showing the field direction on a surface
5. by showing the field lines'

[Quarendon 1987 pp.1-2]

Scalar field data is specified in WINSOM as a discrete array of values, up to a maximum size of 64x64x64. The first method of display is equivalent to volumetric rendering, described in the next section. The second method displays a boundary representation of the isosurface, while the remaining methods are specific techniques for visualising scientific data.

A technique described by Gallagher and Nagtegaal in 1989 finds an isosurface in a scalar field producing a number of bi-cubic parametric patches.

2.4.2 Volumetric rendering

A method distinct from finding a boundary representation of an isosurface for visualising scalar field data is the volumetric rendering technique [Drebin, Carpenter and Hanrahan 1988; Sabella 1988; Upson and Keeler 1988]. Volumetric rendering makes no attempt to extract isosurfaces from the scalar field, but rather treats the scalar field as a volume which has to be visualised. Specific scalar field values can be assigned colour and translucency parameters, while the unspecified scalar values assume interpolated results for the parameters. The general procedure in calculating an image based on volumetric rendering is to calculate the colour of each pixel by determining which volume elements in the scalar field affect the pixel. The relevant volume elements and their associated colour and translucency values are then used in the calculation of the proper colour for each pixel. A volume element is also known as a voxel. The size of the voxels in volumetric rendering are either determined by the resolution of the scanning device or set by the user.

Volumetric rendering is a popular technique among groups wishing to view empirical data sets, whether obtained through: fluid flow simulation [Helman and Hesselink 1989; Long, Lyons and Lam 1989]; medical imaging

[Fuchs, Levoy and Pizer 1989; Mosher and Johnson 1989]; seismic data [Wolfe and Liu 1988]; or simulations of meteorology [Wilhelmson and Upson 1989] to name only a few of the applications.

The unique abilities offered by volumetric rendering can be widely applied to many applications of visualising three dimensional scalar data. The visualisation technique is likely to become increasingly popular, Craig Upson has said:

> ' ... volume rendering is doing for computer graphics in the late 1980s what the introduction of perspective did for drawing and painting during the Renaissance. '
>
> [Frenkel 1989 pp.426-427]

Volumetric rendering is useful mainly for applications which study scalar fields which contain complex interactions, relationships or multiple iso-valued surface of interest. Volumetric rendering is more expensive in terms of computation than the boundary representations. A decision must be made as to whether or not the capabilities offered by volumetric rendering are required for a particular application.

The isosurface modelling techniques discussed in this research are geared to the modelling of objects using isosurfaces. In this research the boundary representation has been used exclusively for the display of the isosurface models created. The decision to solely use the boundary representation in this research is justified in chapter four.

2.4.3 Special purpose visualisation hardware

Several research projects have studied the possibilities of speeding the display of isosurfaces and volumetric rendering. The display of isosurfaces is easily handled by several graphics workstations, for instance those by Silicon Graphics, as the surface can be represented polygonally. The process of extracting an isosurface from a scalar field has received little attention from the specialist hardware research groups, one exception is the work by Zyda and Walker in 1988. In their work they describe an architecture which is tuned to extract a contour surface from a 30x30x30 grid of scalar values

thirty times a second. In the paper the contour surface which is produced is not equivalent to a polygonisation of an isosurface. A contour surface is composed solely of lines, a polygonisation is composed of polygons. A polygonisation is suitable for use in conjunction with hidden surface routines, while a contour surface is not. Techniques such as distributed processing may also be applied to the problem of extracting an isosurface from a scalar field.

The process of volumetric rendering may involve the processing of in excess of ten Mbytes of raw data which represents millions of voxels. For instance data obtained from an magnetic resonance imaging (MRI) scan is typically produced at a resolution of 256 by 256, with several hundred scans composing one dataset. This results in several tens of millions of voxels.

In an article by Frenkel in 1989, Neurobiologist Vincent Argiro of Maharishi International University in Fairfield, Iowa, discussed a technique he had developed for the Silicon Graphics GT series of graphics computers. He described a method allowing 240,000 voxels per second to be displayed in the production of certain images. Argiro predicted maximums of 400,000 to 1,000,000 voxels per second possible on the currently available hardware [Frenkel 1989 p.430].

Research is also ongoing on specialist hardware for the display of volumetric data, such as that obtained from CT scans. In a paper by Goldwasser and Reynolds in 1987 an architecture is described which would allow the real time manipulation and display of 3D medical objects.

In the ongoing research into the Pixel-Planes architecture [Fuchs et al 1985] a method for displaying volumetric data has been produced [Fuchs et al 1989]. High quality images of an 256x256x256 voxel data set are estimated to take one second to produce. A degraded image is expected to be able to be produced ten times a second.

The real time display of voxel data is currently largely in the research domain, although this will change as the results of current research projects are commercialised.

2.5 Animation

Animation in computer graphics has advanced from specifying transformations over time (such as rotation, scaling and translation) to the present array of techniques which includes: behavioural animation [Reynolds 1987]; dynamics [Wilhelms 1987]; procedural animation [Comninos 1985]; parametric animation [Leffler, Ostby and Reeves 1988]; and scripting [McPheeters and Wyvill 1984]. A review of computer animation techniques was written by Pueyo and Tost in 1988, it covers several classifications of animation systems and their probable future development. Also, a review by Wyvill in 1988 discusses the impact of different modelling techniques to animation as well as different animation approaches, for instance kinematic versus dynamic.

There are two basic methods of extending the capabilities of computer animation:

1) Create new methods of transforming existing modelling techniques. The techniques based on dynamics is an example of this. Dynamics can be applied to a wide range of modelling techniques and can be used to create animation sequences which would have been extremely difficult using alternative techniques. Behavioural animation is also in this category.

2) Modelling techniques can be created which allow new forms of animation to be produced. Elastically deformable models are an example of this [Terzopoulos et al 1987].

The isosurface modelling techniques falls into the second category, it allows the creation of forms of animation which would otherwise be extremely expensive to produce. Merging, distorting and metamorphosing models are examples of this. In a paper by Wyvill, McPheeters and Wyvill in 1986 describing *soft object* animation, examples are given of the potential of the animation of isosurface models. Many extensions and additional examples are presented in this thesis.

2.6 Conclusions

The research presented in this thesis makes a number of contributions to computer graphics modelling and animation, these are detailed in the remaining chapters and summarised in chapter nine. This research project is preferably viewed in the context of the related research, although this thesis does stand on its own.

The isosurface modelling technique is a technique which allows the modelling of objects which are not easily described using alternative existing techniques. The animation of the isosurface models results in animation sequences which would be prohibitive to produce without it.

Of the literature related to isosurface modelling which exists in the computer graphics area, most seems to be directed towards visualisation rather than modelling or animation. This may change in the future as the potential of the isosurface modelling technique is realised. The indications are that the potential of the isosurface modelling technique is becoming accepted. A recent book on interactive computer graphics discusses the *soft object* modelling techniques [Burger and Gillies 1989 pp.434-435]. Also, a recent article in Byte magazine discussing parallel ray tracing entitled 'The art of ray tracing' displays models which were produced using the *soft object* modelling technique [Ransen 1990].

As the isosurface modelling techniques become more widely understood and developed through efforts such as this current research project, the modelling technique will become more widely known and accepted. The isosurface modelling techniques are able to be applied to a wide variety of applications.

Chapter 3

Specification of the scalar field

3.1 Introduction

A variety of modelling representations have been introduced to the computer graphics field, many techniques are used in their implementation. The surfaces that are generated by the different modelling representations can be broadly divided into two main types: parametric and implicit.

Parametric surfaces are characterised by the expression:

$$P = \{x(u,v), y(u,v), z(u,v)\}$$

Where u and v are the parameters of the equation. As u and v are varied through their domains, the set of points P is generated by evaluating functions x, y, and z. Polygons, patches and generalised cylinders are examples of parametric modelling techniques.

Implicit surfaces are characterised by the formulation:

$$f(P) = 0$$

The implicit formulation is used in the implementation of many modelling representations, including: some of the constructive solid geometry (CSG)

systems; in molecular modelling; meteorology; and medical imaging. The implicit formulation is the basis of the surfaces on which this research has focused.

The fundamental difference between the parametric and implicit formulations is apparent in the treatment of points. Using the implicit formulation, a point can be tested for membership of the set of surface points, but the set of surface points can not easily be generated without *a priori* information of the implicit formula. As a result of the evaluation of the implicit formula a point can be classified to be inside, on the surface or outside the volume of the model. This inside/outside classification of points is not easily accomplished using a parametric formulation. Using the parametric formulation the set of surface points can be quickly generated, however a given point can not be easily classified as being a member of the surface or not.

The evaluation of an implicit formula at any given point yields a scalar value. A scalar value specifies only magnitude, not direction. The implicit formula is often referred to as a scalar function, a function which evaluates to a scalar value. The value is referred to as a scalar value in order to avoid confusion between functions which evaluate to a vector, tensor, or other possible quantities.

A scalar field is a field which has, for any given point, an associated scalar value. In solving the implicit formulation, all points in the scalar field are found which match a particular scalar value. As scalar fields are defined using the real number system, there exists an infinite number of points which satisfy the implicit formulation of a surface. All implementations of the implicit formulation use a subset of the possible points as an approximation to the surface.

The surface represented by the implicit formulation is known by many names. The surface can be called: an implicit surface; an isosurface; an equipotential surface; a level surface; or some similar name as discussed in section 2.3. It should be noted that an isosurface may be composed of many disjoint parts, each of which encloses a separate volume of space.

There are many isosurfaces contained within most implicit formulations. The isosurface with a scalar value of one can be easily found by modifying the implicit formulation to:

$$f(P) = 1$$

or creating a new function, g defined as:

$$g(P) = f(P) - 1$$

and then solving for:

$$g(P) = 0$$

Similar techniques can be used to create a variety of scalar valued functions.

An implicit formulation of a scalar valued sphere can be expressed as:

$$x^2 + y^2 + z^2 - 1 = 0$$

The explicit, or parametric formulation of the same sphere is:

$$P = (\sin\phi \cos\Phi, \sin\phi \sin\Phi, \cos\Phi)$$
$$0 \leq \phi \leq 2\pi$$
$$0 \leq \Phi \leq 2\pi$$

Using isosurfaces as a modelling technique in computer graphics requires that two major components of a graphics system be present. There must be a means of specifying the scalar fields which are a basis of the technique, and there must be at least one method of extracting the isosurface or equivalent information from the scalar field for visualisation purposes. This chapter describes different techniques and problems associated with specifying the scalar field.

3.2 Methods of describing scalar fields

For a modelling technique to be useful in a wide variety of applications, the users of the system must not be required to interact with the system in a mathematical or algorithmic manner. Mathematical and algorithmic modes of interaction may be useful in specific situations, such as scientific visualisation. However an approach which is more intuitive is required for the method to be useful to designers or artists.

The methods used to implement scalar fields may bear no direct relation to those which are utilised by the user interface for their design. The implementation of the scalar field may require intricate mathematics, however the user interface may hide these complexities and provide an intuitive interface to the specification of scalar fields.

The techniques most suited for the specification of scalar fields may vary widely. Certainly the techniques will differ between an application dealing with sub-atomic physics and one dealing with the modelling of soft sculptural forms. In this chapter, techniques for specifying a scalar field to be used in the modelling of objects is discussed. The problem of designing a suitable user interface is left as an area for future research.

A mathematical approach to specifying the scalar field can be accomplished using algebraic expressions, such as the quadratic formula. The quadratic formula is suitable for many common surfaces, such as: spheres; ellipsoids; cylinders; half spaces; planes; paraboloids; and many others. A more complex formula can be used to represent a torus or other surfaces. One has only to leaf through a book on geometry and calculus to see examples of formulas which are used to describe relatively complex surfaces.

In general, the surfaces required by designers are not easily described as a single algebraic expression. Many separate components may be required in order to achieve the effect required by the designer. Representing a single quadratic surface using a purely algebraic approach is relatively straight forward. However, combining two or more quadratic surfaces into one scalar valued function is not as easily accomplished. The result of simply adding the two quadratic formulae together gives the wrong result, as is demonstrated in figure 3.1 Each of the formulae represents a sphere with an isosurface value of zero. The graph depicts the scalar field as sampled along the x axis, at y and z equal to zero. The desired result of the addition is that two spheres would be present. Instead, both of the spheres disappear due to the fact that the scalar value is always greater than zero and therefore never crosses the isosurface value. One alternative to the purely algebraic approach is procedural. A simple procedure which incorporates many quadratic formulae into one scalar field is:

Let Q_i be quadratic formula i

$$\text{return} \quad \min_{i=1}^{n} \left(Q_i (P) \right)$$

Using the above procedure, the objects will not blend smoothly. However this remains one of the simplest useful methods of combining many separate scalar components. The technique of finding the minimum value is applied to the previous example, and is demonstrated in figure 3.2

An alternative to the procedural approach for combining many algebraic formulae, is to find a higher order algebraic expression which has similar characteristics. This is difficult to generalise due to the fact that a majority of higher order functions have frequencies which need to be tightly

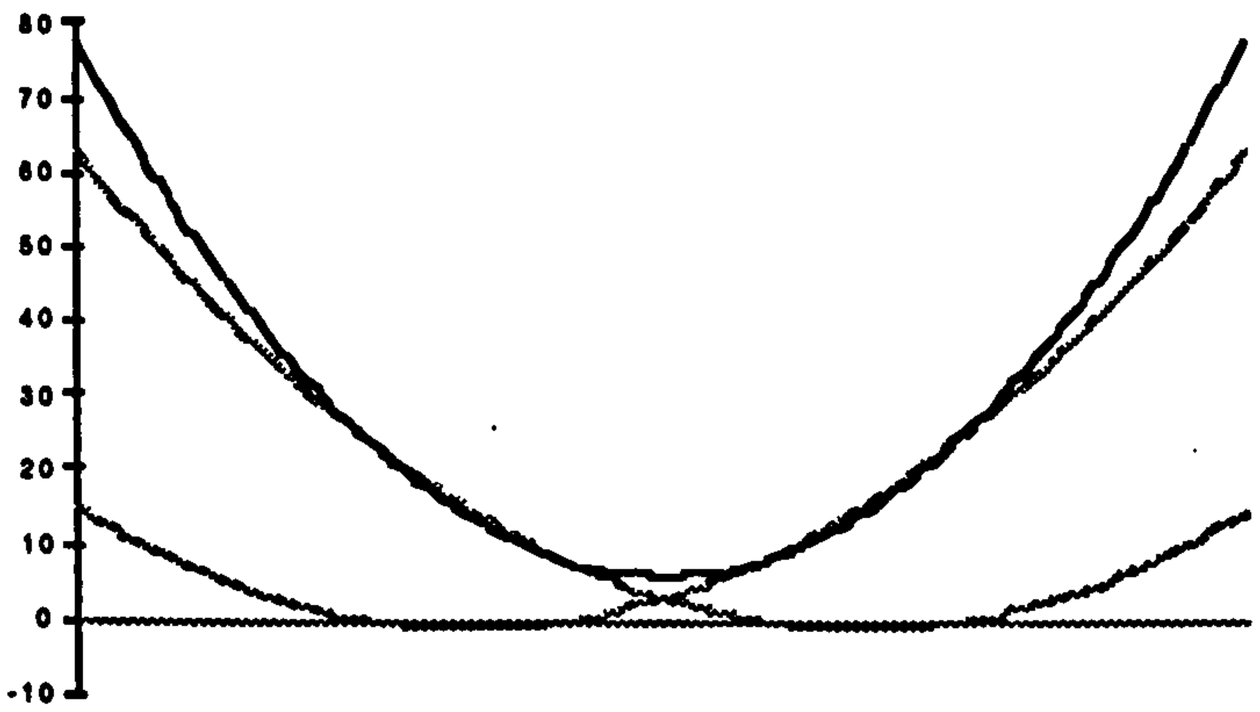


Fig 3.1 Summation of two quadratic formulae.

Each individual formulae is given as a grey curve. Notice that each grey curve crosses the zero line. The dark curve is the summation of both grey curves and does not cross the zero line.

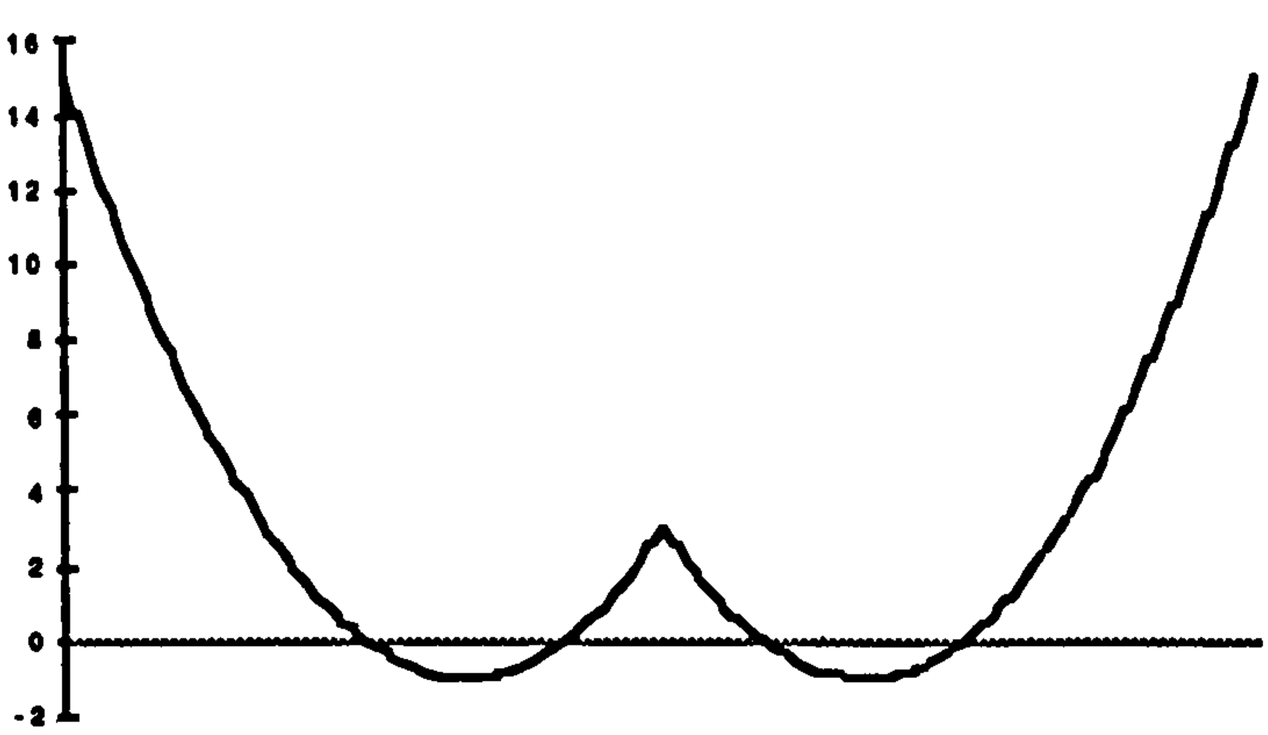


Fig 3.2 Minimum of two quadratic formulae

constrained in order to avoid artifacts.

A similar situation exists in the specification of the splines which are used in computer graphics. A spline with *n* points can be expressed as a polynomial of degree *n - 1*, or alternatively as a piece wise series of cubic polynomials. The piece wise approach is generally more efficient, easier to handle, and does not demonstrate the unwanted frequencies, seen as ripples, that characterise functions of high degree. The procedural, rather than the purely algebraic approach is most commonly used for the specification of splines in computer graphics.

A second disadvantage to using a purely algebraic approach for combining scalar valued functions is that it is not possible to combine all possible types of scalar function using purely algebraic means. The procedural techniques for combining scalar functions make it possible to create arbitrarily complex scalar fields in an easily managed fashion. Using a purely algebraic means of specifying scalar fields severely restricts the generality of the scalar field description. This is the main reason why the purely algebraic approach was not used as the sole basis of scalar field description in this research.

One advantage of the procedural approach is that each scalar valued component remains independent throughout the evaluation of the entire scalar field. This frees the creator of the graphics system from many restrictions which would otherwise be present using alternative means of combining the scalar functions. One further advantage is that each component can be as complex as required and be implemented in the manner for which it is naturally suited.

This research focuses on procedural means of constructing a scalar field from many diverse scalar valued components. A complex object is broken into a variety of simple parts. Each of these parts is represented directly as a scalar valued component which is included in the graphics system, or as a combination of these components.

There are many scalar valued components implemented in this research, along with many techniques used to combine the scalar components. Section 3.3 contains a description of the components from which the scalar fields are constructed. In section 3.4 the techniques which are used to combine each of the individual components into a scalar field are discussed, as are problems found during the combination process.

3.3 Description of scalar field components.

There are a variety of components from which a scalar field can be composed. These components are implemented using a number of techniques. It is necessary to establish the characteristics which are to be shared between all of the components as well as establishing the information that is required of the components during the modelling process. This is in order to ensure that the scalar field components interact in a consistent manner.

In the visualisation process, the operation of extracting an isosurface from a scalar field is accomplished by finding points which match a particular scalar value. The same scalar value is used in the search for points on the isosurface throughout the entire scalar field. This leads to the requirement that a particular scalar field value be chosen as the most common isosurface value. This value has been arbitrarily chosen as one.

Each scalar valued component will have a natural configuration which can be transformed and adjusted to achieve a wide variety of surfaces in the design process. For example, the sphere component has a natural configuration with a radius of one, centred about the origin. The implication of the decision above is that this natural configuration will occur at a scalar field value of one. Different scalar values for the isosurface will yield different radii of the sphere.

The accepted method of adjusting the size, location and orientation of the scalar components is through the application of a modelling transformation. This modelling transformation will be specified as a 4 by 4 transformation matrix. Although the size of an isosurface can be changed within certain limits by choosing different isosurface values, neither the location or orientation can be changed in this manner.

Another characteristic defined for each component is that the scalar field surrounding an object should fall to zero once it is sufficiently far away from the surface it is modelling. This allows for a natural interaction between separate objects. An object is not affected by changes to other objects if they are sufficiently far away. This characteristic also allows the possibility of a

more efficient evaluation technique for the scalar field. This technique will be described later in the thesis.

A direct implication of the above requirements is that the interior of the isosurface must consist of scalar values greater than one. The scalar value of a component will reach a maximum somewhere interior to the isosurface. The scalar value will fall to zero at some distance from the surface in a manner which is not yet specified.

In the absence of criteria which would make this impossible, the maximum scalar field value is around two. This is specified so that all possible composition operators will cooperate correctly, as will be shown.

The creation of a wide variety of scalar valued components is encouraged in this research. To encourage the implementation of new components, the implementation of each should be as straightforward as possible. Therefore each component is only required to supply a scalar value for each of the points demanded, no other information is needed. Consideration was given to the requirement that each component be able to supply a range of information. This was rejected because of the difficulty in implementation for all of the components which were foreseen in this research. This point is discussed further in subsequent chapters.

Some of the additional information that is desirable but not required, is a gradient at each point and a set of initial isosurface intersections which may have aided the visualisation process. As this information can be obtained from the scalar field numerically, no flexibility is lost by the simplification of the requirements for each scalar component. A consequence of this decision is that each component can be expressed as simply as possible, thus allowing the design of a wide variety of scalar valued components.

The final characteristic of the scalar components is that they should behave in a consistent fashion regardless of scale, orientation or position.

A summary of the common characteristics shared between all scalar field components is given in table 3.1.

- Each component is able to be evaluated for any point.
- Each component must be able to be transformed by a matrix.
- A default isosurface value of one is assumed.
- The interior of the isosurface surface is greater than one.
- The maximum scalar value is roughly two, unless otherwise specified.
- Each component will, outside of some range, evaluate to zero.
- The components should behave consistently, regardless of scale, orientation or position.

Table 3.1 Characteristics required of each scalar component

An alternative set of characteristics for the scalar components could have been produced. In the research by Middelditch and Sears in 1985, a distance function is found for each component, which is used in calculating blends. This distance function evaluates to the distance from a given point to the surface, becoming zero for a point on the surface. The distance function is difficult to create for complex components, such as those including random contributions. In the research published by Perlin and Hoffert in 1989, the functions return a value of one for any point at or inside the surface. The value of the function falls to zero in a unspecified manner as a point moves further away from the surface. This formulation has the disadvantage of creating a discontinuity at the surface value. As Perlin and Hoffert were not interested in creating surfaces, but were rather interested in texture volumes, this did not cause a problem in their work.

In this research, the underlying structure of the scalar components implemented is one of: algebraic; procedural; empirical; or possibly a combination of all three. Through this underlying structure, the components have been classified and are discussed in the following sections.

Each of the scalar valued components goes through three stages: initialisation, execution and completion. Initialisation and completion occur once, respectively at the start and end of the scalar field evaluation. Each component is usually executed at least once for every point examined by the isosurface visualisation process.

3.3.1 Algebraic components

A majority of the scalar valued components implemented as part of this research are based on algebraic expressions. Of these, six are implemented using the quadratic formula, and two are implemented using other techniques. Each of the components will be described, as will the methods used to satisfy the characteristics required of each.

Sphere

The sphere is the least complex of the components implemented. The sphere cannot be rotated, and is not distorted by non-uniform scaling. The sphere is implemented through the use of a simple expression. The relevant modelling information is extracted from the transformation matrix.

The origin of the sphere is calculated by transforming the origin of the coordinate system by the transformation matrix, and using the resulting point as the centre of the sphere. The radius of the sphere is calculated as the average of the scale for each of the three major axis. Non-uniform scaling could alternatively have been handled by finding the maximum, minimum or other relationship between each of the three axis. The average was chosen arbitrarily as it seemed a good compromise. If non-uniform scaling is desired, the ellipsoid component should be used instead of the sphere, as it will distort properly.

$$\text{Origin} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Transform-} \\ \text{ation} \\ \text{Matrix} \end{bmatrix}$$

$$X_{scale} = \left| \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \text{Transform-} \\ \text{ation} \\ \text{Matrix} \end{bmatrix} \right|$$

$$Y_{scale} = \left| \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \text{Transform-} \\ \text{ation} \\ \text{Matrix} \end{bmatrix} \right|$$

$$Z_{scale} = \left| \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \text{Transform-} \\ \text{ation} \\ \text{Matrix} \end{bmatrix} \right|$$

$$\text{radius} = \frac{X_{scale} + Y_{scale} + Z_{scale}}{3}$$

The radius and origin are found during the initialisation stage of the sphere component.

An implicit formulation of a sphere centred on point C with radius *r* is:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - \text{radius}^2 = 0$$

This formulation does not conform to the required characteristics. A modified expression which has the radius specified at scalar value one, with the interior of the surface greater than one is:

$$1 - \left((x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - \text{radius}^2\right) = 0$$

The maximum value of this function is $(1 + \text{radius}^2)$, there is no limit on the minimum value. The shape of the function will change according to the radius of the sphere. This will affect the manner in which the sphere interacts with objects around it, depending upon the scale of the spheres involved. This is a violation of one of the characteristics that are demanded of the scalar valued component.

An alternative implicit formulation of the sphere is:

$$\frac{(x-C_x)^2+(y-C_y)^2+(z-C_z)^2}{\text{radius}^2} \quad \text{, for } r > 0$$

This has a value of zero at the centre, a value of one at the radius and increases as a point moves away from the centre. This can be transformed into what is required by inverting the function, which puts the surface value at negative one, and adding two. This formulation of the sphere is given as:

$$\text{sphere}(x,y,z) = 2 - \left(\frac{(x-C_x)^2+(y-C_y)^2+(z-C_z)^2}{\text{radius}^2}\right) \quad \text{, for radius} > 0$$

This function has a maximum value of two regardless of the scale of the sphere. A minimum value is imposed on the function by clamping the minimum of the function to zero. This clamping satisfies a further characteristic required of scalar valued components, that the scalar value fall to zero at some distance away from the isosurface.

$$\text{result} = \text{maximum}\big(\text{sphere}(x,y,z), 0\big)$$

Clamping functions introduces discontinuities. These discontinuities can cause problems in a variety of ways if the clamped values are taken into consideration during the evaluation of the isosurface. The scalar valued components have been tuned to work best around a scalar value of one. Since the clamped value is at zero, this may be sufficiently removed from the surface that the discontinuity will not be visible. This point will be discussed again later.

The expression for the sphere proposed above does not constrain the shape of the quadratic function. The distance from the isosurface at which the function falls to zero can be specified, and is referred to as the *influence* of the sphere. The *influence* of the sphere controls the distance at which the sphere may influence other scalar components. As the *influence* parameter is increased, the sphere will influence components further and further away. *Influence* is specified in terms of a multiple of the radius. For a sphere with a radius of two, an *influence* of one implies that the sphere will evaluate to zero at two units away from the isosurface. Similarly an *influence* of one half implies the sphere will evaluate to zero at a distance of one away from the surface.

Small values of *influence* will bring the discontinuity which was introduced by clamping the function to zero, closer to the surface. For this reason small values of the *influence* parameter are to be avoided.

The *influence* parameter is implemented as a function which modifies the scalar value returned by the basic sphere function. The criteria required of the scalar components such as a default isosurface value of one and having the interior of the surface rise to a scalar value of two are satisfied by the new formulation which includes the calculation of *influence*. The basic sphere function used in the formulation of *influence* is:

$$\frac{(x-C_x)^2+(y-C_y)^2+(z-C_z)^2}{radius^2} \quad \text{, for radius} > 0$$

The operation of the *influence* function is defined as:

$$\text{Final scalar value} = F_{influence}\left(\frac{(x-C_x)^2+(y-C_y)^2+(z-C_z)^2}{radius^2}\right)$$

The criteria imposed upon the behaviour of the function $F_{influence}$ are:

$$F_{influence}(1) = 1$$
$$F_{influence}(I) = 0$$

To find the value of *I*, the value of the basic sphere function must be known for the point which is the correct distance away. This distance is defined in terms of *influence*. *Influence* is a multiple of the radius away from the surface, this makes the distance $(influence * \text{radius} + \text{radius})$. The value of the resulting sphere function at this point must be zero. Substituting the distance into the equation for the sphere, *I* becomes:

$$I = \frac{(influence * \text{radius} + \text{radius})^2}{radius^2} \quad \text{, radius} > 0$$

As there are only two constraints, they can be expressed in a polynomial of the form:

$$F_{influence}(v) = a * v + b$$

Substituting the constraints into the expression of the influence function, we obtain the following set of linear equations:

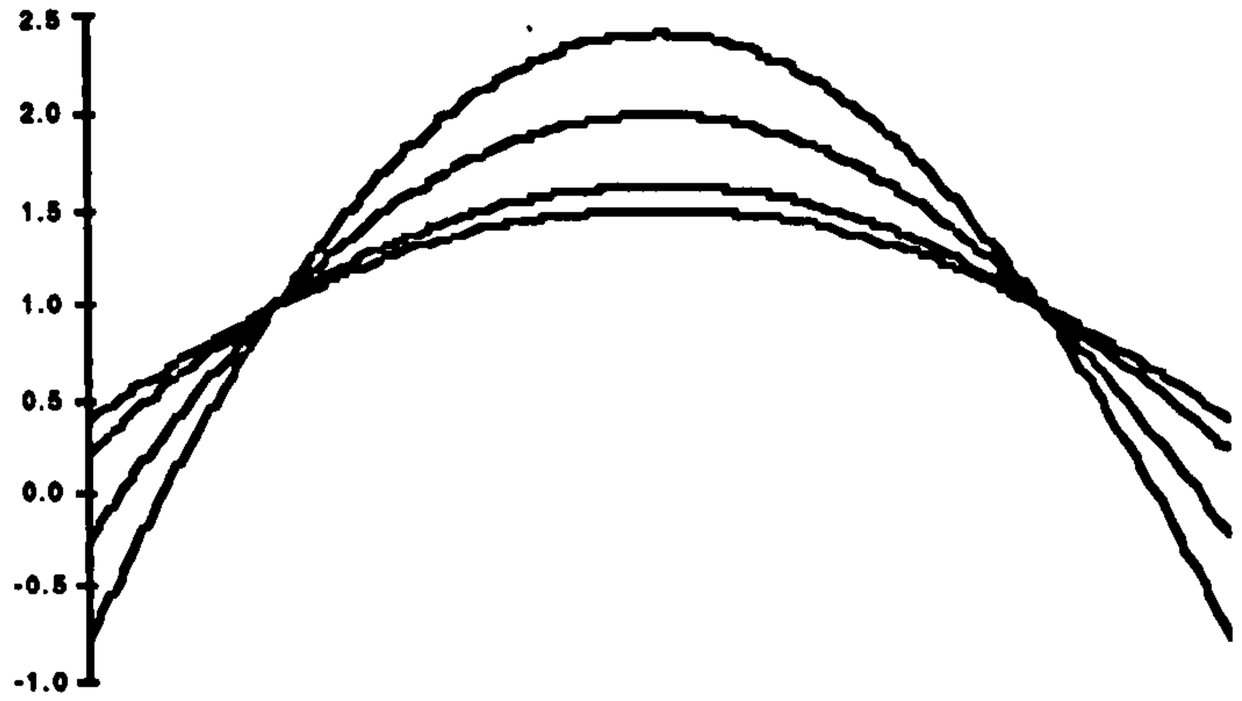


Fig 3.3 Varying values for *influence* on a sphere. The top curve has an influence of 0.5, below it are curves representing 1.0, 2.0 and 2.5

$$a + b = 1$$
$$Ia + b = 0$$

Solving for *a* and *b* and substituting back into the polynomial, we get:

$$F_{influence}(v) = \left(1 + \frac{I}{1 - I}\right) v - \frac{I}{1 - I}$$

A set of linear equations can be solved in a variety of ways, such as Gauss-Jordan elimination [Wilde 1988]. The expression of *influence* above guarantees that the interior of the isosurface is greater than one.

Applying the above formulation of *influence* back to the basic sphere function, we get:

$$sphere(x, y, z) = \left(1 + \frac{I}{1 - I}\right)\left(\frac{(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2}{radius^2}\right) - \frac{I}{1 - I}$$

$$I = \frac{(influence * radius + radius)^2}{radius^2}, \quad radius > 0$$

A family of curves defined on a sphere with a radius of one, for varying values of *influence* is shown in figure 3.3.

Quadratic components

The quadratic polynomial is represented by an equation of the form:

$$Ax^2 + By^2 + Cz^2 + Exy + Fxz + Gyz + Hx + Iy + Jz + D = 0$$

This equation can be represented in matrix form as:

$$Q = [x\ y\ z\ 1] \begin{bmatrix} A & \frac{E}{2} & \frac{F}{2} & \frac{H}{2} \\ \frac{E}{2} & B & \frac{G}{2} & \frac{I}{2} \\ \frac{F}{2} & \frac{G}{2} & C & \frac{J}{2} \\ \frac{H}{2} & \frac{I}{2} & \frac{J}{2} & D \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

All quadratic equations in the above form can be transformed using matrix algebra. In this application, a quadratic formula will be transformed by the modelling transformation matrix to yield a new quadratic formula which is used in the evaluation of the function. The technique for transforming the matrix form of a quadratic formula is described in detail elsewhere [Wilde 1988], and is given as:

Let Q_m be the matrix of quadratic coefficients

Let T^{-1} be the inverse of the transformation matrix

Transformed quadratic matrix = $(T^{-1}) \bullet Q_m \bullet (T^{-1})^{transpose}$

Using the above technique, the quadratic equation which is used in the representation of a scalar valued component is transformed once at the initialisation stage. This equation is then evaluated directly for each point that is required of the scalar valued component.

There are six scalar components implemented using the quadratic formula. They are: ellipsoid; cylinder; half space; hyperboloid; paraboloid; and a saddle function. Only the half space does not have any of the squared terms of the polynomial, and is thus a linear polynomial. The initial quadratic

Component	Non zero quadratic coefficients
ellipsoid	A = 1, B = 1, C = 1
cylinder	A = 1, C = 1
half space	H = 0, I = 1, J = 0
hyperboloid	A = -1, B = 1, C = -1, D = -distance
paraboloid	A = -1, C = -1, I = -1
saddle	A = -1, C = 1, I = -1

Table 3.2 Initial quadratic coefficients for quadratic components

formula used for each of the components is given in table 3.2. For the hyperboloid, the *distance* parameter is the distance between the parabolic sheets. If this distance is less than or equal to zero, the sheets merge to become a parabolic cylinder, otherwise a pair of parabolic cones are produced.

The quadratic formula must be augmented to account for the characteristics required of each of the components. The constraints that the isosurface must be at scalar value one, and the interior must be greater than one are accomplished differently for the half space as for the other components.

Influence is added to the half space in a similar manner as for the sphere, but with different constraints. They are:

$$F_{influence}(0) = 1$$
$$F_{influence}(I) = 0$$

$$I = 1 + influence, \quad influence > 1$$

These constraints can be expressed in an equation of the form:

$$F_{influence}(v) = a * v + b$$

A set of linear equations are produced as for the sphere component, they are:

$$b = 1$$
$$Ia + b = 0$$

Solving for *a* and *b* and substituting back into the polynomial gives:

$$F_{influence}(v) \doteq \left(\frac{-1}{I}\right) v + 1$$

The final expression for the half space is then:

$$result = \left(\frac{-1}{I}\right) * quadratic(x, y, z) + 1$$

Defining the other quadratic components without *influence* gives:

$$result = maximum\ \left(2 - quadratic(x,y,z), 0\right)$$

When defining the other quadratic components with *influence* the conditions on the function are similar to those for the sphere. They are:

$$F_{influence}(1) = 1$$
$$F_{influence}(I) = 0$$

$$I = (1 + influence)^2,\ influence > 0$$

Again, a set of linear equations are constructed and solved. The modified expression, taking into account the calculation of *influence*, with the interior of the isosurface greater than one and the default isosurface equal to one, is:

$$result = \left(1 + \frac{I}{1 - I}\right) quadratic(x, y, z) - \frac{I}{1 - I}$$

$$I = (1 + influence)^2,\ influence > 0$$

Torus

Of the possible expressions that could be used to describe a torus, the one that is used in this research is:

$$torus(x, y, z) = \left(\sqrt{x^2 + z^2} - R_{major}\right)^2 + y^2 - \left(R_{minor}\right)^2$$

The above expression can not be easily transformed by a transformation matrix into a new equation which takes into account the modelling

transformation. The approach that was taken in the implementation of the sphere could have been used for the torus, but this was considered unsatisfactory. The sphere component does not respond to non-uniform scaling, the torus must. The solution used is for the evaluation of the torus to proceed in its normal coordinate system. This involves finding the inverse of the modelling transformation, and using this inverse for every evaluation that is required of the torus.

$$\text{Result} = \text{torus}\left(P * T^{-1}\right)$$

This increases the complexity of the evaluation of the torus component by adding the multiplication of a point by a matrix. The advantage is purely in terms of the ease of implementation. As stated earlier, one of the advantages of the procedural approach for specifying the scalar field is that each component may be represented in the form to which it is naturally suited. For the torus the most suitable form is for the evaluation to progress in the torus' coordinate system.

The expression of the torus is modified slightly to take into account the fact that the isosurface should have a scalar value of one, and that the interior is greater than one:

$$1 - \text{torus}\,(x, y, z)$$

The specification of *influence* is in the number of minor radii the scalar field should extend before falling to zero. Similarly to previous implementations of *influence* a set of constraints is constructed with respect to the basic formation of a torus:

$$F_{influence}(0) = 1$$
$$F_{influence}(I) = 0$$

$$I = \left(influence * R_{minor}\right)^2, influence > 0$$

After creating a set of linear equations and solving them, the final expression of torus becomes:

$$\text{torus}_{\text{final}}(P) = \frac{-1}{I}\,\text{torus}\left(P * T^{-1}\right) + 1$$

$$I = \left(influence * R_{minor}\right)^2 , influence > 0$$

Sine

There are four parameters which may be specified for each sine wave component: maximum scalar value; minimum scalar value; frequency; and phase. The sine wave component is included primarily to provide a means of modifying the behaviour of other scalar components, as will be shown later in this chapter.

The sine wave component is symmetric about its origin, and thus can not be rotated. The origin of the sine component is extracted from its transformation matrix in a similar manner as for the sphere. The frequency of the sine component is adjusted inversely according to the scales of the three major axes. For example, if a particular frequency is scaled down by one half, the resulting frequency is twice the original. The scale in each of the axis is calculated similarly as for the spheres. The expression for frequency is given as:

$$\text{frequency}_{new} = \left(\frac{X_{scale} + Y_{scale} + Z_{scale}}{3}\right) * \text{frequency}_{old}$$

The expression for sine is then:

$$\sin(A + \text{phase}) * \text{amplitude} + \text{middle}$$

$$A = 2\pi \bullet \text{frequency} * \left| (x - C_x) + (y - C_y) + (z - C_z) \right|$$

$$\text{amplitude} = \text{max} - \text{min}$$

$$\text{middle} = \frac{\text{max} - \text{min}}{2} + \text{min}$$

The sine wave repeats continuously throughout three dimensional space, therefore there is no need to take into account an *influence* parameter. Also, as the maximum and minimum values of the sine wave are set by the user, there need be no modification to make sure the surface value is one. It

is often desirable, as will be seen, to specify a sine wave which does not cross the isosurface scalar value.

3.3.2 Procedural components

As all of the scalar valued components are implemented in a procedural form, the components which are actually classified as being procedural need to be clarified.

The basic implicit formulation of procedural components is more than can be expressed using algebraic expressions. For each of the algebraic components an algebraic expression was first introduced which solved the basic requirements of the implicit formulation. The components in this section have no such simple algebraic expressions, they are best expressed as a series of instructions. These instructions may range in complexity from complex procedures, to a series of algebraic expressions combined in a procedural manner.

Despite the fact that there were only three procedural scalar valued components defined in this research, the procedural components remain one of the most exciting possibilities for describing complex scalar fields. Bloomenthal also advocates the use of procedural scalar components in his paper [Bloomenthal 1987].

Methods used to produce three dimensional texture maps have been introduced recently [Peachey 1985; Perlin 1985]. The techniques used in the implementation of three dimensional texture maps, with slight modification, could also be used to produce complex scalar fields. A three dimensional texture map which projects fire onto the surface of an object can be modified to produce a scalar field which represents the fire. This could then be used as the basis of a scalar field with the result that the surface of the flames could be produced. Other three dimensional texture maps which seem suitable for the specification of scalar fields are: marble, ripples, turbulence, and clouds to name but a few.

Extensions to three dimensional texturing to define textures in volumes have been made recently [Perlin and Hoffert 1989; Kajiya and Kay 1989]. These extensions are not suitable for the description of surfaces, as is the aim

in this research, but are rather used for adding texture volumes to surfaces in order to enhance realism.

The three procedural scalar valued components implemented in this research are: the plane; cube; and noise. The first two involve repeated uses of the quadratic formula, the last is an application of a three dimensional texture mapping technique.

Plane

The plane scalar component is constructed from two half spaces, each 'back to back' and separated by a small space. This defines a sheet which can be viewed from either side, which is useful in the construction of many objects.

The two half spaces which form the basis of the planes are initialised so that both are defined in the x-z plane, one pointing towards the positive y axis at a y value of one half, and the other pointing towards the negative y axis at a y value of negative one half. Both of these quadratic equations are transformed in the same manner as were the quadratic polynomials in the previous section. The two quadratic polynomials are then used as follows, in the evaluation of the plane:

$$\text{Plane} = \text{minimum}\ (\text{half space}_1, \text{half space}_2)$$

Influence is specified exactly as it is done for the half space.

Cube

The cube scalar valued procedural component is implemented in a similar fashion as for the plane. The plane was constructed from two half spaces, the cube is constructed from six.

Quadratic formula	Normal	Origin
Q_1	1, 0, 0	0.5, 0, 0
Q_2	-1, 0, 0	-0.5, 0, 0
Q_3	0, 1, 0	0, 0.5, 0
Q_4	0, -1, 0	0, -0.5, 0
Q_5	0, 0, 1	0, 0, 0.5
Q_6	0, 0, -1	0, 0, -0.5

Table 3.3 Table of quadratic formulae for cube

The default cube is defined as being centred on the origin, with each side one unit long. This is done by defining six half spaces, as defined in table 3.3.

The evaluation is then defined as:

$$\text{cube} = \text{minimum}\ (Q_1, Q_2, Q_3, Q_4, Q_5, Q_6)$$

Each of the quadratic components of the cube are transformed in the standard manner for quadratic polynomials. The handling of *influence*, the default isosurface value of one, and the interior of the cube having scalar values greater than one, are handled exactly as for half spaces.

Noise

Noise is an example of a technique that was first implemented for use with three dimensional texture maps. The success of the implementation of noise as a scalar component may be an indication of the suitability of the three dimensional texture mapping field as a source of exciting scalar valued components.

Noise is defined as a function that adds or subtracts a semi-random value to the scalar field. The value is semi-random for two reasons: no truly random numbers exist on digital computers, they are usually defined as being pseudo random; and secondly the numbers are defined on a three dimensional lattice. Tri-cubic interpolation is used to yield a scalar value at any point in the lattice. Details of the implementation of noise functions can be found elsewhere [Perlin 1985; Lewis 1989].

The specification of noise requires that the largest and smallest random values be specified. These values are used as a range from which values are assigned to the key points of the integer lattice.

The evaluation of the noise scalar valued component proceeds in its original coordinate system. The transformation of points into this system is accomplished using the inverse transformation as was done for the torus and sine components.

There is no need for the *influence* parameter in the definition of noise, due to the fact that noise exists throughout the complete three dimensional coordinate system.

3.3.3 Empirical components

The general principles, applications and problems of implementing empirical components are outlined in this section. Empirical components are useful in situations where scalar fields are defined by an external process. The processes used to create the scalar fields are varied, but include: X-ray computed tomography (CT); magnetic resonance imaging (MRI); positron emission tomography; ultrasound imaging; X-ray crystallography; and various types of simulations. A good description of the various medical scanning technologies was written by Goldwasser and Reynolds in 1987.

The techniques implemented in this research are not entirely suitable for the scalar fields produced using medical imaging equipment. Such scalar fields often need to be pre-processed in order to accomplish filtering procedures, data enhancement and various other processes. A dedicated medical imaging system is better suited to this task. Three dimensional medical data sets are quite large, a data set larger than ten megabytes is not unusual. Handling such data sets requires specialised techniques which have not been investigated as part of this research. However, given these considerations, there is no fundamental reason for which medical or other empirical data sets can not be handled elegantly in this body of work.

One fundamental difference between empirical data sets, and the procedural or algebraic functions that the previous components have been based on, is

that an empirical data set contains pre-sampled scalar field information at particular locations. The spacing of these samples determines the frequency of the data set. In CT scans, the sampling frequency is on the order of a millimetre. The frequency of the sampling in a simulation data set will vary greatly depending upon whether or not the simulation is of sub-atomic physics, or of geophysical processes in rock formations. A more useful measurement of the sampling frequency, rather than a particular spatial measurement, is its relation to the frequencies that exist in the data before sampling. In order that aliasing be avoided, the data set must be sampled at the highest frequency of interest that exists in the object, or greater. If this is not done then information may be lost, or incorrect results may be obtained.

As an empirical data set consists of sampled data at discrete points, a technique must be found to associate a scalar value to every point in space. Recall that one of the characteristics demanded of these scalar valued components implemented in this research is that they are able to be evaluated at any point.

The mapping between an arbitrary point and a scalar value related to the data set can be accomplished in several ways. The most straightforward mapping would be to find the nearest discrete sample in the data set and use this. As the data set will most probably be composed of a regular mesh of scalar values, this search can be accomplished efficiently. If the empirical data set is composed of random points, the look up can be efficiently accomplished using an octree approach.

Assuming the data set is composed of a regular three dimensional lattice of scalar values, a similar technique to that which is used in the implementation of noise can be applied: tri-linear or tri-cubic interpolation.

The selection of a technique to be used for the extraction of a scalar field from an empirical data set has to be made in view of the particular set of circumstances at the time. There may be a speed/quality trade off which will influence this decision. Through the availability of several techniques for empirical scalar field extraction, most situations are adequately satisfied.

Empirical data sets represent somewhat of a specialisation in the scalar field arena. Raw data sets obtained through the measurement of real world data will seldom be used in the state in which it is received. Processing of the data is often required, as image processing proceeds on two dimensional

empirical data sets. Due to the nature of this research these processing techniques have not been explored. However, given an empirical data set which needs no further enhancement, many techniques exist for the extraction of a scalar field from it.

3.4 Methods of combining scalar field components

The approach taken for the description of the scalar field has been to break the scalar field into a variety of scalar valued components and combine them in some fashion. An alternative to this is to describe the scalar field as one component. This would be difficult for any moderately complex scalar field. The reasons for the decision to use many components were discussed in section 3.2. The components which are available, and components which could be made available are described in section 3.3. The methods of combining the scalar valued components, and methods of solving any problems when this combination takes place are considered in this section.

3.4.1 Avoiding discontinuities

One of the simplest methods of combining many scalar valued components into one scalar field is simply to add the contribution of each component. A variety of techniques will be described in the next section for combining the scalar fields from many components.

The method of using addition as a means of combining scalar components is described as:

Let C_i be scalar valued component *i*

Let P be the point for which a scalar field value is required

$$\text{scalar value} = \sum_{i=1}^{N} C_i\,(P)$$

The above formulation of a scalar field can be used to highlight a problem with the definition of many of the scalar field components described so far.

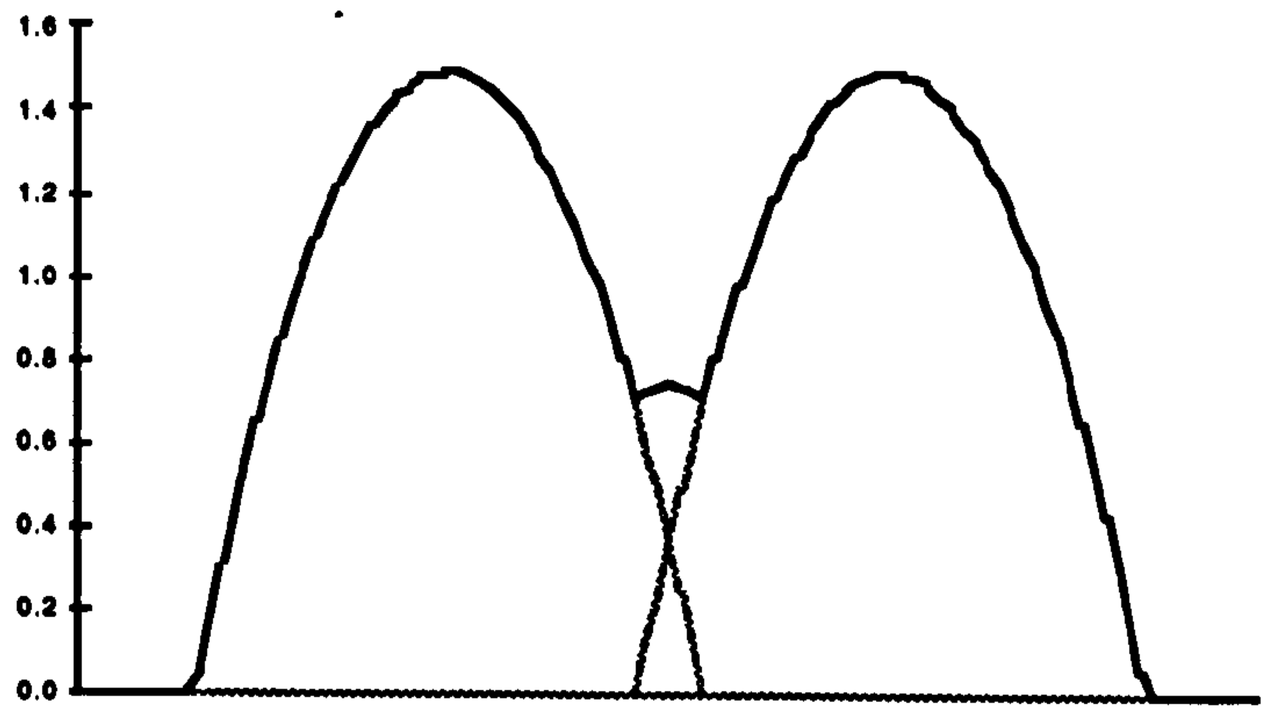


Fig 3.4 Demonstration of the effect of discontinuity when combining components

Assume the scalar field is composed of two spheres with radius of one and an *influence* of two, located at (1.3,0,0) and (-1.3,0,0). The graph of the scalar field values sampled along the x axis is given in figure 3.4.

It can be seen clearly from figure 3.4 that there is a discontinuity in the first derivative of the resulting scalar field. This discontinuity will be apparent in the interaction of any two components in which this simple clamping operation has been used to limit the functions to be greater than or equal to zero. Removing the clamping operation does not solve the problem, and in fact makes it worse. Removing the clamping operation causes the surfaces that were present to disappear completely. Clearly an alternative method of solving this problem is needed.

The discontinuity of the curvature of the scalar field is caused by a discontinuity in the curvature of the components upon which they are based. Each of the components which clamp the quadratic function to always be greater than zero creates a point in the scalar field for which a gradient is undefined. In order for the curvature of the scalar field to be continuous, the curvature at the clamp point needs to be specified.

The gradients in the scalar field of a sphere beyond the point at which the values are clamped are zero. In order that there be no visible bumps in an isosurface, the curvature of the sphere at the point of clamping must also be zero. To avoid discontinuities, the curvature of the function describing the sphere must smoothly approach zero as the function approaches a scalar value of zero.

This modification to the sphere component can be accomplished by modifying the values returned from the sphere function so that they conform to the characteristics needed to avoid discontinuities. The values must be smoothed.

This smoothing function must conform to several constraints in order to interact with the component in a logical manner. A scalar value of one can not be changed, otherwise the smoothing modification would have the effect of changing the isosurface of each of the components. This would be undesirable and is avoided. A scalar value of zero must remain zero after smoothing. This is required in order that the *influence* parameter continue to be specified correctly. The gradient at the scalar value zero must remain zero. These three constraints can be formulated into a polynomial of degree two. A further constraint is imposed at this point in order to solve a problem that will be considered later. This constraint requires that the second derivative of the scalar valued function must also be continuous. This latter constraint is imposed in order that the appearance of the generated isosurface does not exhibit ripples in shading as the components interact.

The constraints placed upon the smoothing function are then:

$$\begin{aligned} F_{smooth}(0) &= 0 \\ F_{smooth}'(0) &= 0 \\ F_{smooth}''(0) &= 0 \\ F_{smooth}(1) &= 1 \end{aligned}$$

The above constraints can be expressed in a polynomial of degree three:

$$F_{smooth}(v) = a * v^3 + b * v^2 + c * v + d$$

A set of linear equations can be constructed from the above constraints and are given as:

$$\begin{aligned} d &= 0 \\ c &= 0 \\ 2b &= 0 \\ a + b + c + d &= 1 \end{aligned}$$

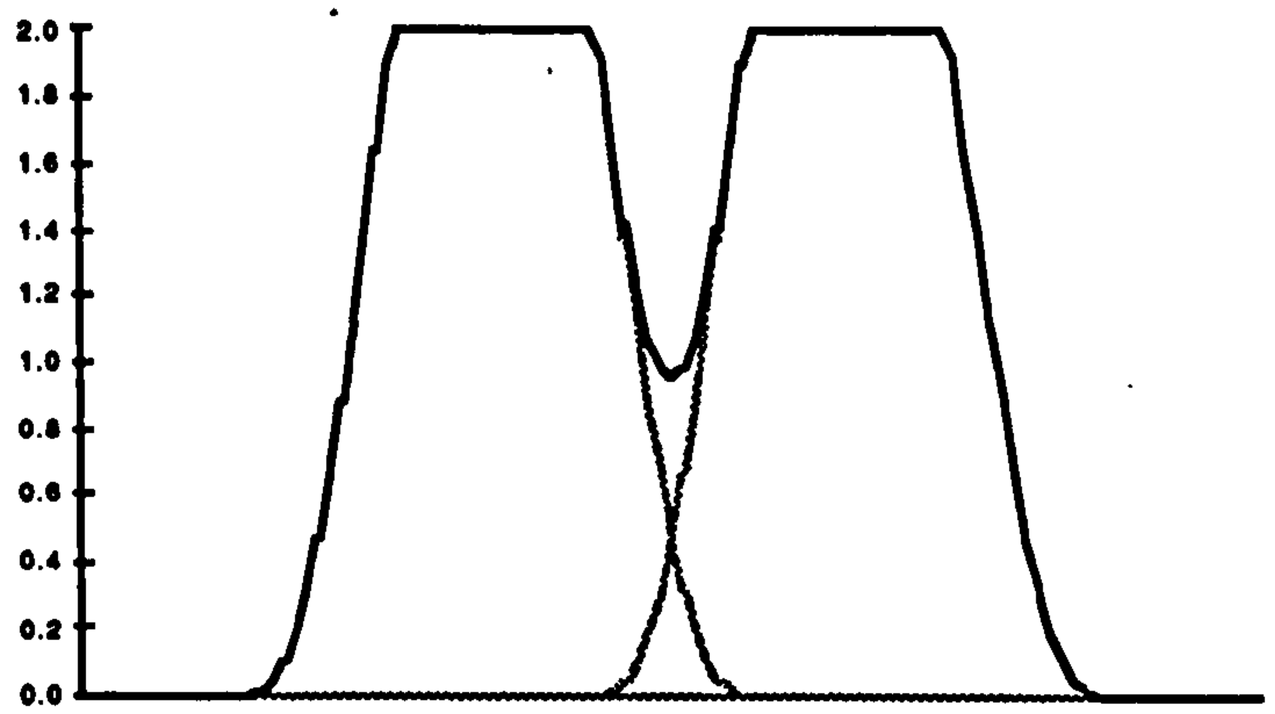


Fig 3.5 Addition of two spheres using first approximation to smoothing

The linear equations can be solved, and yield a solution which satisfies the above constraints. It is given as:

$$F_{smooth}(v) = v^3$$

A graph of the sphere before and after smoothing is given in figure 3.5. Unfortunately the above constraints were found to be insufficient, and one further constraint was added:

$$F_{smooth}(2) = 2$$

Solving the new set of linear equations yields a polynomial of degree four which satisfies the constraints:

$$F_{smooth}(v) = \frac{7}{4} v^3 - \frac{3}{4} v^4$$

This smoothing function is applied to each component which uses clamping. This is done by applying the smoothing polynomial to the scalar value after clamping. Clamping is applied to the scalar value before the smoothing operation takes place, as the smooth function has undesirable characteristics outside of the range of interest. This smoothing operation is applied to: the sphere; ellipsoid; cylinder; cube; plane; half space; torus; hyperboloid; paraboloid; and the saddle function.

$$\text{scalar value} = F_{smooth}\Big(\text{clamp}\ \big(\text{scalar function}\ (P)\big)\Big)$$

A graph of the two spheres used to produce figure 3.5 with the smoothing modification is given in figure 3.6

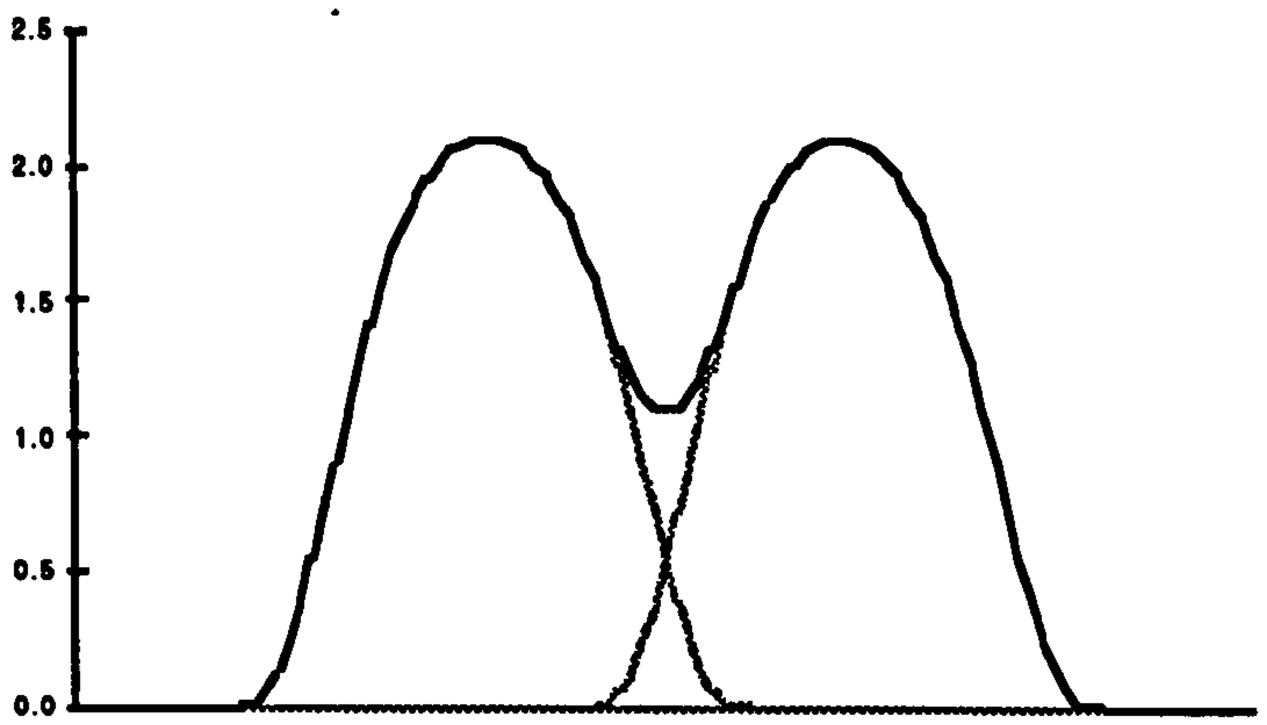


Fig 3.6 Graph of two spheres using the final smoothing formula

In order to avoid discontinuities in the scalar field, attention must be directed to each scalar valued component. Any discontinuity in the scalar value, or its first and second derivatives may be reflected in some fashion on the isosurface.

The implementation of *noise* guarantees continuity of the first and second derivative, as does the implementation of *sine*, therefore there is no need to smooth the values returned by these components.

In the interests of maintaining the ease of implementation of varied scalar valued components, the characteristics demanded of the component in terms of smoothness can be left until after the basic scalar field evaluation. In this manner implementation is simplified while continuity of the scalar field is maintained.

3.4.2 Composition operators

Previously in this chapter, there were two methods proposed for combining scalar valued components into a complete scalar field. The two methods were: finding the minimum of all components to yield a scalar value; and adding the values of all components to achieve the summation which yields a scalar field value.

There are six main types of composition operators proposed in this research to accommodate the creation of a scalar field from various scalar valued components. These operators are: addition; subtraction; minimum;

Operator	Representation	Result
Addition	A [+] B	A + B
Subtraction	A [-] B	A - B
Minimum (Intersection)	A [min] B	minimum of A or B
Maximum (Union)	A [max] B	maximum of A or B
Mix	A [mix]$_m$ B	$(m * A) + (1 - m) * B$
Mirror	A [mirror]$_m$	$2 * m - A$

Table 3.4 Scalar field composition operators

maximum; mix; and mirror. The definition of each of the operators is given in table 3.4.

Each of the scalar valued components represents an individual modelling component. The components can be transformed to any size, orientation and position through the use of the modelling transformation. It is through the use of the composition operators that the components can be combined in a variety of ways.

The addition operation is used to create the effect of smoothly merging objects. This is the most useful and commonly used of the operations. Two spheres which approach each other, if they are being added, will smoothly blend to eventually produce one surface. If the second sphere is being subtracted from the first, then it acts as a negative volume which is subtracted from the first. The minimum of two spheres will produce the surface of the intersection of the volumes of the two spheres. The maximum of two spheres will produce the union of the two spheres volume. The union of two components differs from the addition of two components in that they will not smoothly blend if the maximum (union), is being found. The mixture of two components presents an elegant method of metamorphosing between two scalar valued objects. The mirror operation is a method of turning a scalar component inside out. The scalar value most commonly used as a value about which to mirror is one. This would act as a complement operation, inverting the scalar field about the default isosurface value. For instance to view the inside of a sphere, it can be mirrored about a scalar value of one.

In the research published by Perlin and Hoffert in 1989, an alternative set of composition operators is proposed. Four operators are described: intersection; union; complement; and difference. An operator to accomplish smoothly merging objects is not proposed.

The composition operators as implemented in this research are demonstrated in the following series of figures, based upon two spheres which are spaced slightly apart.

In the following series of figures there are three diagrams presented for each composition operator. The contour diagram shows isovalues (from the middle) of: 1.5; 1; 0.5; and 0. Notice that the mirror operator reverses this order. A second diagram represents the magnitude of the scalar function as elevations, the function is sampled along the x - z plane at y equal to zero. The third diagram shows the graph of a sampling of the function along the x axis at y and z equal to zero.

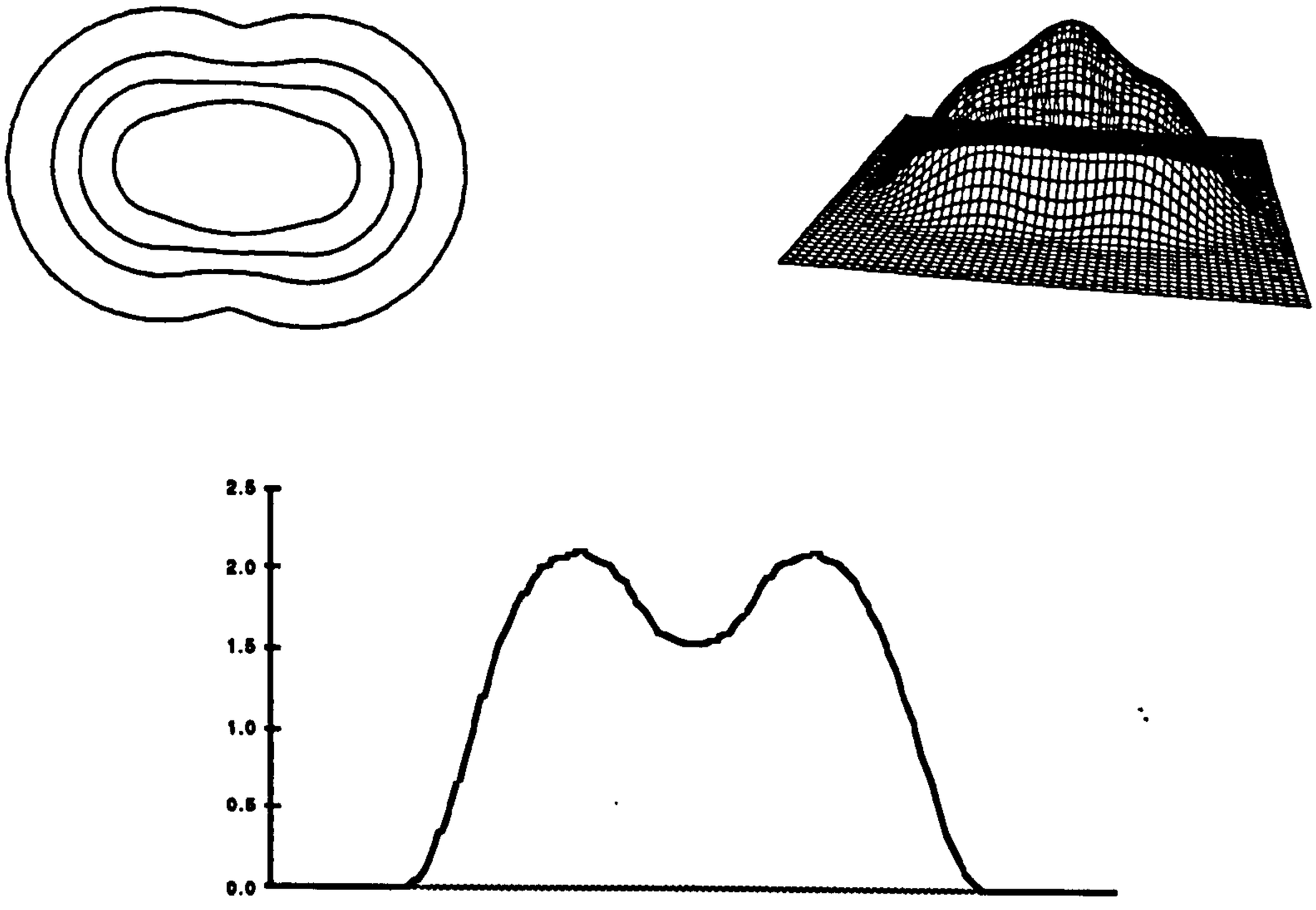


Fig 3.7 Addition of two spheres

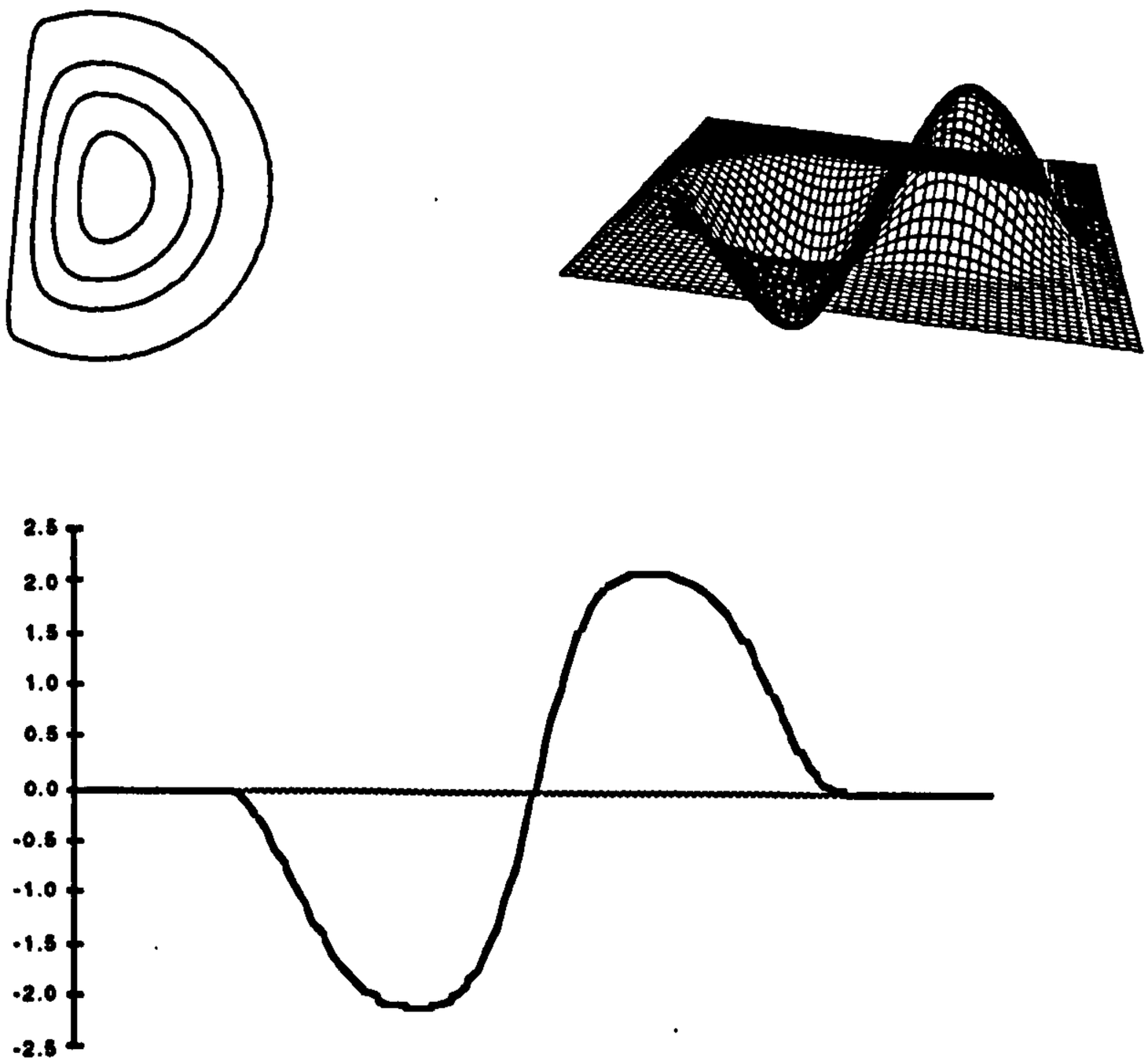


Fig 3.8 Subtraction of the left sphere, addition of the right

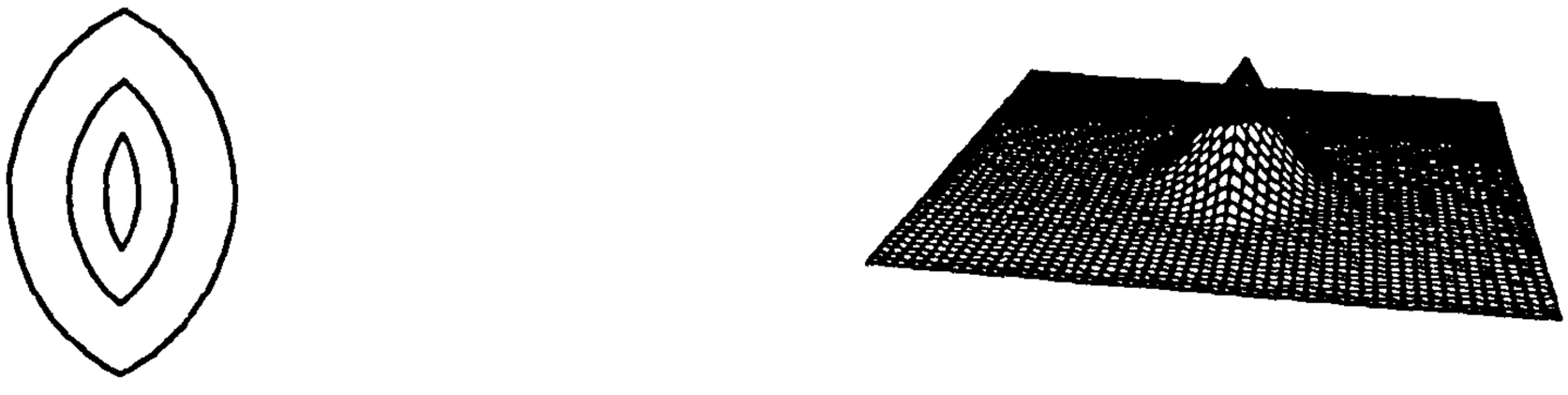


Fig 3.9 Minimum of two spheres

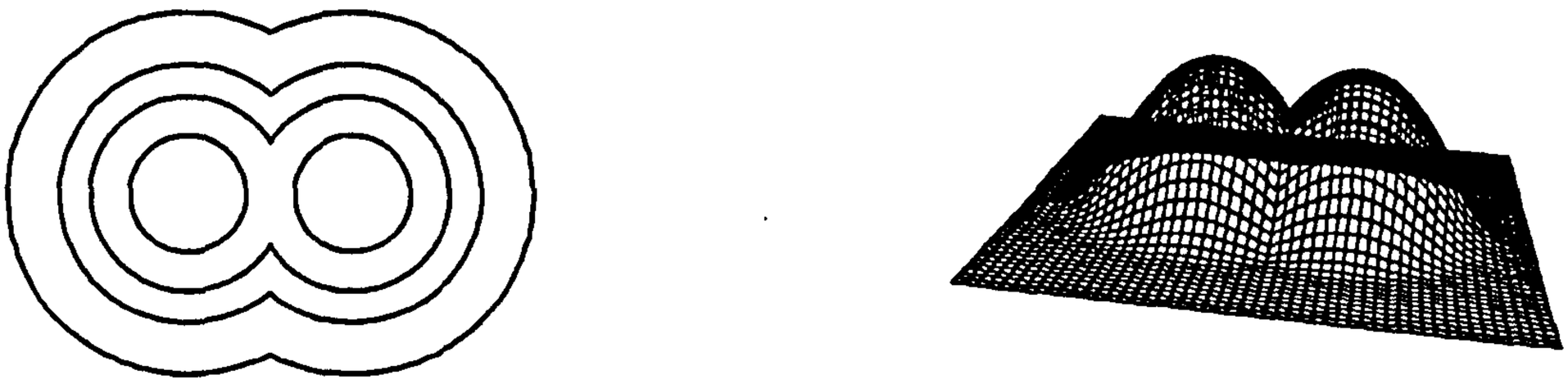


Fig 3.10 Maximum of two spheres

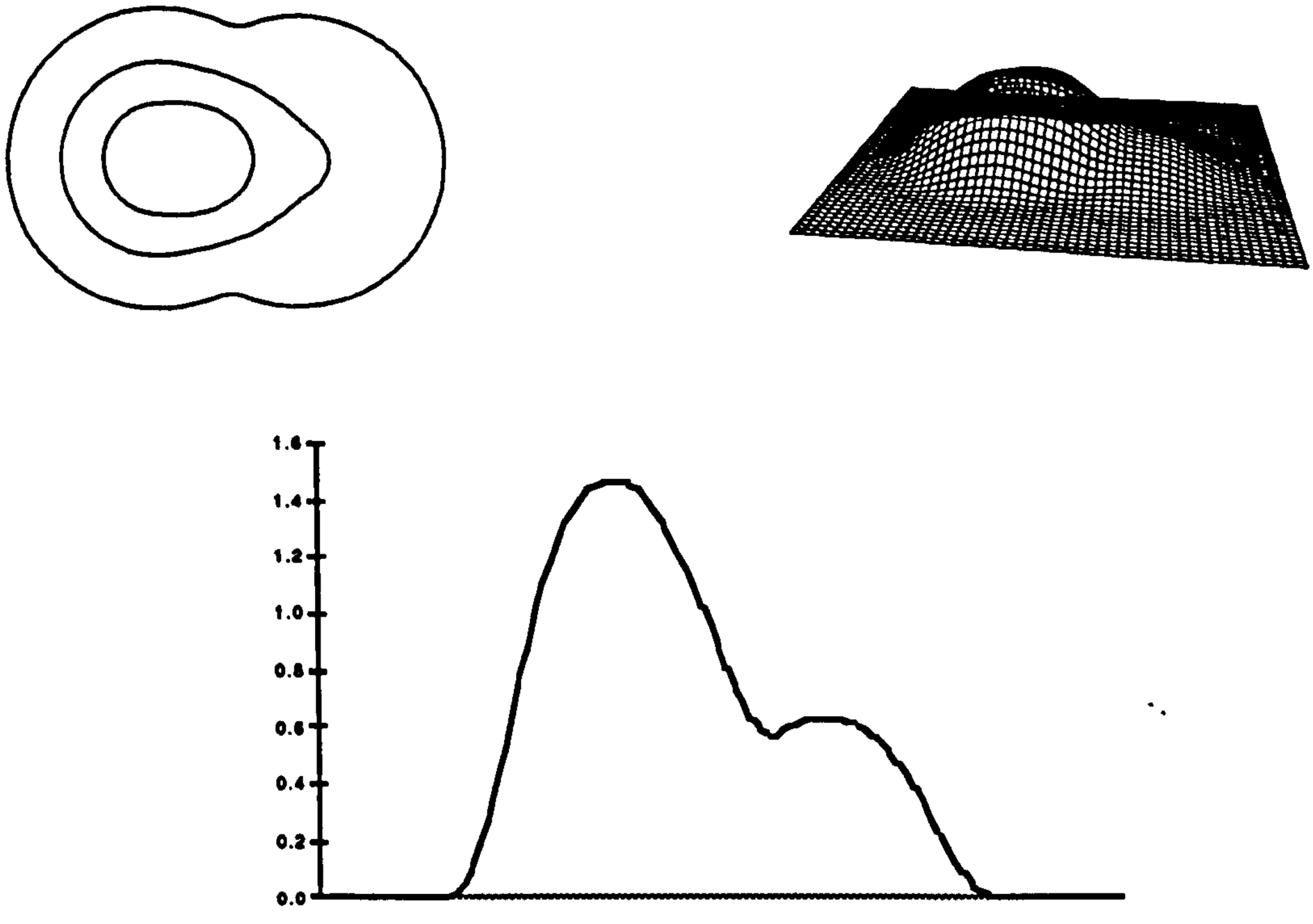


Fig 3.11 Mixture of two spheres.
(Seven tenths of the left, and three tenths of the right)

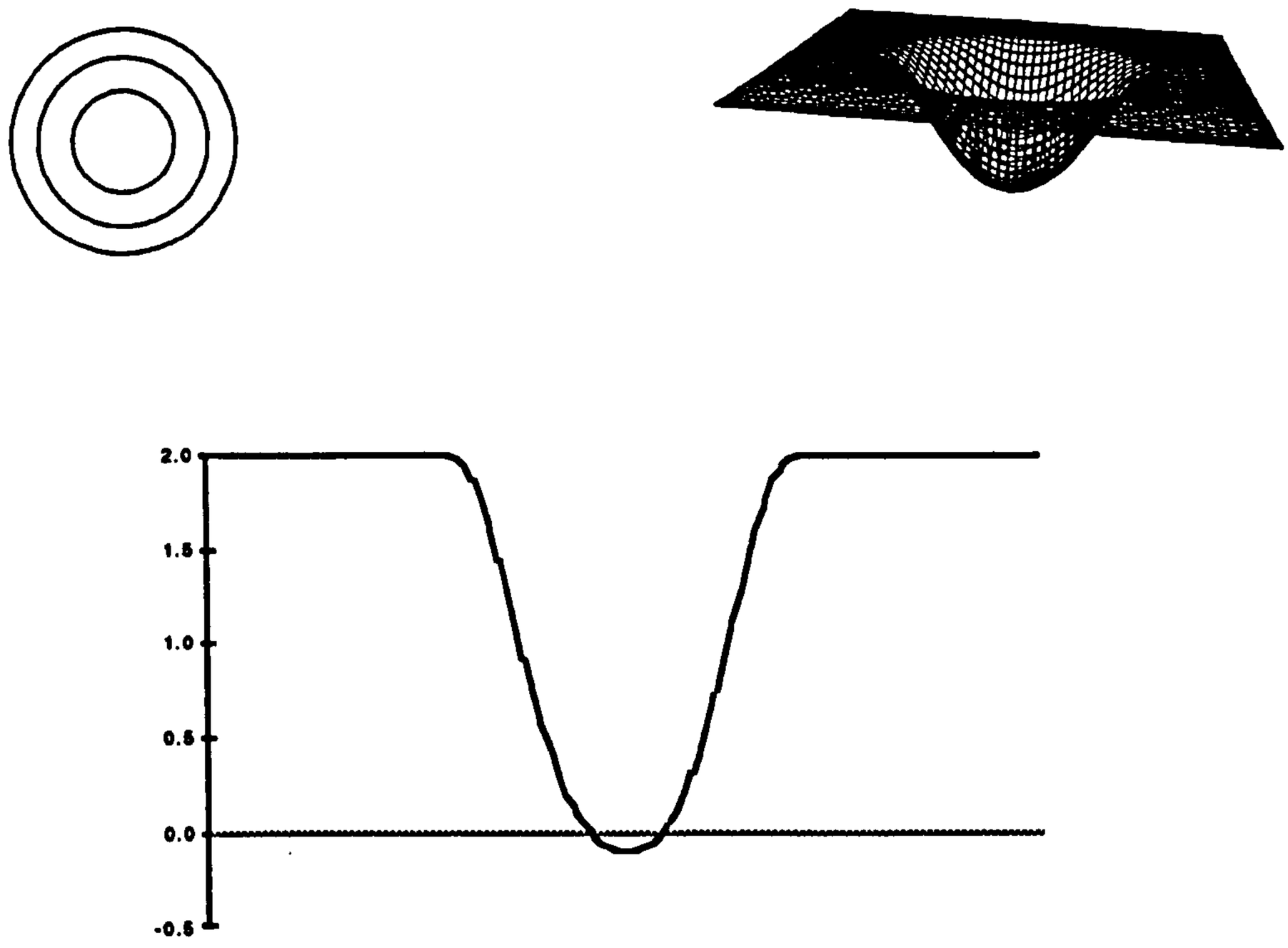


Fig 3.12 Mirror of a sphere around scalar value one

Complex objects can be modelled using the range of scalar components available and creating complex expressions involving the scalar component operators. An object can be created in parts through the use of the grouping operators. The order of evaluation can be specified by grouping components together. Notice that:

$$A \boxed{\min} (B \boxed{+} C)$$

is not equivalent to:

$$(A \boxed{\min} B) \boxed{+} C$$

Grouping of the above expression is essential in order that the correct result be obtained. Grouping also allows the possibility of the creation of an object library. A standard set of objects can be modelled in advance. These models can be incorporated as groups in the scalar field expression.

An operator is associative if the components being combined with it are able to be regrouped. For example:

$$(A + B) + C \quad = \quad A + (B + C)$$

An operator is commutative if components which use it can be rearranged. For example:

$$(A + B) \quad = \quad (B + A)$$

A list of the properties of the operators appears as table 3.5. The mirror operator is not considered, as it is a unary operator. For an operator to have either associative or commutative properties, the operator must be a binary operator.

A successful effort has been made to avoid discontinuities in the description of the scalar components, around the region of scalar value one. For the scalar field to remain free of discontinuities the operators that are used to combine the scalar components must not introduce any discontinuities. Four of the six operators do not introduce discontinuities. However the maximum and minimum operators do introduce a discontinuity.

The maximum and minimum operators are implemented as a binary decision, either one value or the other is used as a result of this operation.

Operator	Associative	Commutative
Addition	yes	yes
Subtraction	yes	no
Maximum	yes	yes
Minimum	yes	yes
Mix	no	no

Table 3.5 Associative and commutative classification of binary operators

This in effect creates a surface which divides the two possibilities. Points on this surface are undifferentiable.

The implementation of intersection and union as presented by Perlin and Hoffert would avoid the problem of an undifferentiable surface if used in this research. For example, their implementation of intersection is:

$$A \text{ intersection } B = A(P) * B(P)$$

The interior and surface of their objects have a scalar value of one. Their research is concerned with applying volume texture, not with isosurfaces. An implication of this is demonstrated by the fact that at an isosurface value of 0.5, the intersection of two spheres would not be calculated correctly. Although in this research project a default isosurface value of one is proposed, it is desirable for the operators to operate correctly regardless of the surface value. The implementation of intersection and union has been arbitrarily chosen to be that presented.

Depending upon the type of scalar field being modelled, it may be desirable to allow additional or fewer composition operators. Operators such as division and exponentials were considered but rejected as they do not have an intuitive result. Dividing one sphere by another results in a valid scalar field, however there is no intuitive sense of what the resulting isosurface should look like. Dividing a sphere by a constant scalar value, intuitively, should decrease the radius by the constant factor used. This is not what would happen in general.

All of the composition operators made available have predictable results. Since this research is primarily concerned with the modelling of the surface of objects, it is considered important to restrict the operators to those that make this task easier. The operators selected can be thought of in terms of an equivalent sculptural operation. The operators rejected have no such

equivalent sculptural operation. If the scalar field description were to be generalised to the modelling of scientific phenomena, it might prove useful to expand the composition operators that are available.

3.5 A scalar field description language

There are many scalar valued components from which a scalar field can be constructed. Six scalar operators are supplied to combine these components. The scalar operators allow the combination of scalar components in order to achieve many effects. In order to increase the control over the creation of the scalar field, grouping of the components and operators is allowed. The scalar field description language (SFDL) implemented during this research incorporates these ideas and is presented.

There were several points taken into consideration during the design and implementation of the description language, these were:

1) To allow the incorporation of all the scalar components, with the facility of adding new ones with ease.

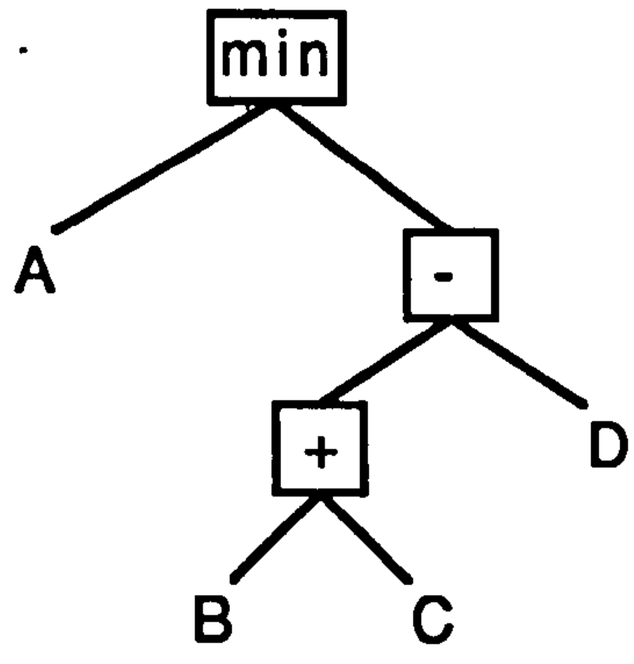
2) To allow the incorporation of all the scalar operators, with the facility of adding new ones with ease.

3) To allow for a future implementation of flow control and a decision making capability.

4) Not to attempt any optimisation at any early stage of the design process, as this often results in a loss of generality. (Optimisations will be considered once the basic design has been proven.)

5) Each SFDL 'program' will describe an instance in time. Animation facilities are not built into the language.

The reason for setting most of these goals is self evident, with the possible exception of the fifth. It was imagined that SFDL would be produced from a separate 'parent' data structure. This 'parent' data structure would be able to handle animation, with the result that a new scalar program can be created as it is needed automatically by the data structure or an associated process. Animation facilities were not incorporated into SFDL mainly as a



prefix: min A - + B C D

infix: A min ((B + C) - D)

postfix: A B C + D - min

Fig 3.13 An expression tree and its three traversals

simplification. This has been accomplished without any loss in capability as will be demonstrated in chapter six.

Each scalar expression can be represented in several ways. Previously in this chapter the expressions were presented in infix notation, as:

A min (B + C) - D

This expression can be represented as an expression tree. An expression tree can be traversed in three ways: inorder, preorder and postorder. These three traversal techniques yield three expression representations: infix, prefix and postfix. Figure 3.13 presents an example of each of these three representations for the above expression.

Postfix is a popular form of representing expressions on computers due to their straightforward interpretation. A postfix expression is easily implemented using a stack of values. As each component is encountered, its scalar value would be put on top of the stack. Each operator would replace two stack values, or one in the case of *mirror*, with a new value. At the completion of this process a single scalar field value will be left on the stack. This scalar field value is the result of the evaluation of the expression.

The implementation of SFDL loosely follows the postfix representation. A stack is used in the evaluation of an expression, however the instructions that operate upon this stack are not the same as those for the pure postfix representation. In the postfix format, each component pushes its value onto the top of the stack. In the representation implemented in this research, each component *adds* its value to the top of the stack. This is done to improve efficiency. This concession does not however result in the loss of any capability. Examples of postfix and SFDL are given below:

infix: (A + B + C + D + E)

postfix: (ABCDE + + + +)

SFDL: (ABCDE)

As SFDL is optimised for addition, it is not surprising that in the above example SFDL gives the most efficient representation. It is expected that addition will be the most commonly used operator.

infix: (A + B) - C

postfix: A B + C -

SFDL: A B push 0 C -

For SFDL in this example, a new level of the stack is required in order that the third component not be added to the result of the addition of the first two.

A summary of the instructions available in SFDL is given in table 3.6.

Instruction	Comment
Load n	Replace the top of the stack with n.
Push n	Put n onto the top of the stack.
Add	Pop the top two values, add them and push the result.
Subtract	Pop the top two values, subtract them and push the result.
Min	Pop the top two values, push the minimum of the two.
Max	Pop the top two values, push the maximum of the two.
Mix_m	Pop the top two values, push the result of mixing them.
$Mirror_m$	Pop the top entry on the stack, apply the mirror operation to it and push the result.
Component	Add the value returned by the component to the top of the stack. Examples of components are: sphere, ellipsoid, cylinder and so on.
Return	Return the top element of the stack as the final result.

Table 3.6 Scalar field description language summary

A computer language normally has several features that are lacking in SFDL. Programs can not be written directly in SFDL at the moment. This is due to the fact that the description of the components involves a lot of information, a stand alone interface was not created. Presently the components are specified in the graphics system, which creates an SFDL program automatically. A second feature lacking in SFDL usually found in a computer language is a form of flow control and decision making. SFDL proceeds with the evaluation of each instruction once, which results in the evaluation of the scalar field. Flow control and decision making are possible in SFDL, but are not implemented at present.

A discussion of whether SFDL is a proper language, or simply a data structure will not be entered. Such a discussion is considered to be peripheral to this research.

It should be noted that although there may be many references to the same type of component in a single SFDL program, these references do not necessarily refer to the same instance of the component. Each component is represented as a separate instance of the type specified. Each instance is initialised with the correct modelling transformation matrix.

```
component ::=   sphere | ellipsoid | cylinder | half space
              | hyperboloid | paraboloid | saddle | torus | sine
              | plane | cube | noise | empirical

unary-op ::=    mirror
binary-op ::=   subtract | maximum | minimum | mix

value ::=       component
              | component value
              | load
              | load value

single ::=      value
              | single unary-op
              | single push single binary-op

SFDL program ::= single return
```

Fig 3.14 Grammar to describe SFDL

The execution of a component instruction, for example 'sphere', entails that the proper sphere instance is called with the point at which a scalar value is required as its only parameter. Upon completion, the component returns the scalar value and SFDL adds it to the top element of the stack.

During this research, it has been found that SFDL allows great flexibility in the description of scalar fields. The addition of new scalar components and operators is easily accomplished. The efficiency of SFDL has proven adequate for this research. Less flexible approaches, for instance offering only addition and subtraction without grouping, may be more efficient, but were not considered as the loss of functionality was undesirable.

A short grammar describing SFDL is presented in figure 3.14.

3.6 Optimisation techniques for scalar field evaluation

Optimisation techniques for the scalar field evaluation incorporate one or more of the following points:

1) If a scalar field value is small enough, it may be ignored.

2) If a point can be classified as being outside a components radius of influence, the evaluation of that component can be ignored.

3) Set theoretic methods may be applied which prune components that fall in regions of space for which no isosurface can be found, due to the combination of the other components and operators.

The first point is a heuristic rule, which may or may not work depending upon the particular heuristic used. This rule assumes that there is an accurate method of estimating the rough magnitude of the scalar component for any point, without actually evaluating it. If the component is complex, a technique of this sort may prove useful in some limited circumstances, largely depending upon the ability of quickly determining whether or not to proceed with the full evaluation.

The second method is the type which is normally implemented. This requires that the spatial extents of a component are known, or can be discovered. This is easily accomplished for some of the components, and not so easily for others.

A sphere has a precise, calculable field of influence. A point outside of a spheres field of influence will evaluate to zero. However, notice that if the mirror operation is applied to a sphere, or a group containing the sphere, the field of influence calculated will have to be modified. This applies equally to any component modified by the mirror operation.

The cylinder extends infinitely up and down the y axis, in its default position. In this default position, a partial bounding box can be found for the cylinder with regards to x and z, however once the cylinder is rotated slightly, the bounding box may extend infinitely in all directions. A bounding sphere would cope no better. At the rest position of the cylinder, a bounding sphere would need an infinite radius.

An alternative to bounding boxes or spheres is to apply the inverse of the modelling transformation to the test point, and test against the original cylinder. This method imposes a large overhead on the evaluation of each point, which may in some cases, exceed the cost of the evaluation it is intended to save.

Some components, such as noise, are by nature unbounded and can not be eliminated from portions of the scalar field evaluation by this second technique. Noise is defined over the entire three dimensional space.

The third method of optimising the scalar field evaluation would be to use methods which are currently being researched in the constructive solid geometry field. These techniques enable the pruning of some components, and other optimisations [Woodwark 1988]. However there remains research to be done to optimise the techniques and modify them so they are suitable for this application. An example of a pruning operation is apparent when half spaces are being used. If a positive sphere is completely on the positive side of a half space, its evaluation is not required in the scalar field evaluation, the same result is possible without it. This is not true for isosurface values in general but is true for a value of one. At other isosurface values, the pruning operations may or may not be used.

There are no scalar field optimisations presently being used in the evaluation of SFDL. The techniques raised in the second point seem the most promising in the near future. The components in this research that have finite bounds are: the sphere; ellipsoid; cube; and the torus. Optimisations in the evaluation of these components will prove useful in some circumstances.

The optimisation of scalar field evaluation remains a rich area for further research. Due to the general nature of the scalar component specification in this research, many of the techniques that were previously used are no longer completely appropriate. Optimisation techniques which handle the components in this research will presumably incorporate many different techniques.

3.7 Conclusions

For the visualisation of isosurfaces defined within scalar fields to be a viable method of modelling *soft objects* in computer graphics, there must, obviously, be sufficient control over the specification of the scalar field.

Several approaches to the specification of a scalar field are possible. One method would be to find an algebraic expression which is evaluated for every scalar field point required. It was quickly realised that a method involving the combination of many diverse components into one homogeneous scalar field is an attractive approach. A procedural approach was adopted.

There are many scalar valued components foreseen in this research. The components which have been implemented are: the sphere; ellipsoid; cylinder; half space; plane; paraboloid; hyperboloid; saddle function; cube; noise; and sine. There are six methods of combining these components: addition; subtraction; maximum (union); minimum (intersection); mix; and mirror. Through the use of these components and operators, many varied objects can be created.

A scalar field description language (SFDL) has been implemented which incorporates the ideas presented into a coherent structure for the specification of a scalar field. Methods to optimise the scalar field evaluation were briefly discussed. Previous methods of optimisation can not be used directly to solve the problem, they must be adapted to the special requirements imposed by the scalar field technique. This is primarily left as an area for future research.

During the design of the scalar components used in this research, a number of characteristics were formulated and a number of enhancements made to each of them. The sphere scalar component is presented as an example of how these differing features were finally implemented in the graphics system, and is shown in figure 3.15.

The design of a user interface used to generate SFDL programs has been largely ignored in this chapter, but will be briefly discussed in chapters seven and eight.

As implemented, the method of specifying the scalar field provides a rich environment for creating many varied and exciting objects. The ease with which SFDL was implemented makes the incorporation of isosurface modelling an attractive possibility for the modelling of many different objects in computer graphics.

```
type SPHERE is
    real    x, y, z                 Location
    real    influence               Value of influence set by user
    real    radius                  Determined by calculation on matrix
end

global boolean do_influence         Determine if the influence
                                    calculation is used  (set by system)
global boolean do_smooth            Determine if smooth calculation is
                                    used  (set by system)

function sphere (SPHERE s;  real x, y, z) returns (real)
begin
    real    result, r_2, dx, dy, dz, I          Temporary variables

    dx = s.x - x
    dy = s.y - y
    dz = s.z - z
    r_2 = dx * dx + dy * dy + dz * dz

    if (do_influence) begin
        I = (s.influence * s.radius + s.radius)^2 / radius^2
        result = (1 + I / (1 - I)) * (r_2 / s.radius^2) - (I / (1 - I))
    end
    else
        result = 2 - (r_2 / s.radius^2)
    endif

    if (result > 2) result = 2
    if (result < 0) result = 0

    if (do_smooth)
        result = (result * result * result) * ((7/4) - (result * (3/4)))

    return (result)
endfunction
```

Fig 3.15 Pseudo code for sphere component

Chapter 4

Visualisation of the isosurface

4.1 Introduction

An isosurface consists of the set of points which are satisfied by an implicit equation of the form:

$$f(P) = s$$

where s is the scalar value of the isosurface and P is a point in three dimensional space. An infinite number of points lay on an isosurface. There are a number of techniques of varying complexity and efficiency that may be used to visualise an isosurface. Five of the more important techniques are briefly introduced:

1) A selection of points on the isosurface can be displayed [Connolly 1986]. For the display of uncomplicated isosurfaces this may be sufficient, however more complicated isosurfaces quickly become confused and ambiguous when displayed using the dot surface technique. This is the quickest visualisation technique available without the use of specialised hardware as well as the easiest technique to implement.

2) A number of parallel planes can be used to intersect the scalar field. Contours can be generated where the isosurface penetrates these planes using a variety of available techniques. For instance, one published technique incorporates the contouring operation with a visible line algorithm, resulting in a hidden line image being produced [Wright and Humbrecht 1979].

3) A three dimensional boundary representation of the isosurface can be produced in a number of ways. This representation is suitable for display as a line drawing, or a hidden surface removal display using a number of established visualisation techniques. Several examples of this technique are available [Wyvill, McPheeters and Wyvill 1986; Tindle 1986; Bloomenthal 1987; Lorensen and Cline 1987].

4) The isosurface may be ray-traced directly. Several examples of this technique are available [Blinn 1982; Nishimura et al 1985; Jevans and Wyvill 1988; Kalra and Barr 1989].

5) Volumetric rendering techniques have been developed, initially primarily for visualising medical data sets. The volumetric rendering technique visualises the scalar field rather than an isosurface contained within the scalar field [Sabella 1988; Upson and Keeler 1988; Drebin, Carpenter and Hanrahan 1988]. Volumetric approaches to visualisation are now finding popularity in a number of fields, including the calculation of texture volumes [Kajiya and Kay 1989; Perlin and Hoffert 1989].

It was acknowledged at the outset of this research project that there would not be one 'best' technique used for the visualisation of isosurfaces. The selection of the most appropriate technique will depend upon the criteria of the application. The main criteria are:

- The amount of time available to display a frame.
- Type of display hardware.
- Compatibility with traditional modelling techniques.
- Complexity of scalar field data.
- Physical basis of scalar field.
- Incorporation of external data.
- Any specialised display hardware available.

This chapter is concerned primarily with the search for and representation of an isosurface. A related visualisation problem is in determining the appearance of an isosurface. The appearance of an isosurface is influenced by factors such as colour, translucency and texture. The issues related to isosurface appearance are discussed in the next chapter.

4.2 Factors involved in finding an isosurface

There are many factors that must be taken into consideration in order to solve the visualisation problem for isosurfaces contained within scalar fields. If the characteristics of the isosurfaces contained within the scalar field are known then it may be possible to implement a number of optimisations. There are no restrictions on the isosurfaces that can be specified in this research. The isosurfaces may range from: a closed surface enclosing a central key point; a finite surface within the convex hull of all objects defined; or at worst a spatially unbounded surface that may or may not be present depending upon many interrelated factors. The optimum techniques to handle each of these cases will be different. Optimisations may be used to handle one case which are not appropriate in other areas.

Four of the fundamental problems associated with the visualisation process are:

1) What volume of space must be searched for the isosurface?
2) How finely should this space be searched?
3) What is or is not an isosurface?
4) How shall the isosurface be represented?

These four problems can be partitioned into two separate tasks. Firstly, the isosurface must be found, and secondly the isosurface must be followed and represented. These two tasks may be implemented as separate stages or they may be incorporated into a single overall algorithm. The problems related to finding the isosurface are discussed first.

4.2.1 Search volumes

The scalar fields used in this research are composed of scalar valued components combined using a variety of operators into a single scalar field description language (SFDL) program which is evaluated at any point in order to find the scalar field value. Of the twelve different scalar valued components which can make up a scalar field, only four have finite spatial extents: the sphere; ellipsoid; torus; and cube. Of the six scalar operators used to combine the components, five will maintain finite spatial extents for the components and one will not. The *mirror* operator may change a component with finite spatial extents into one with infinite extents. For instance if a cube is *mirrored* about a scalar value of one the volume in which it will evaluate to a non-zero scalar value is infinite.

Many of the visualisation techniques require that a volume of space be searched in order to find an isosurface. One exception to this is the direct ray-tracing of the isosurfaces. Direct ray-tracing will be considered later in this chapter.

Bounded and unbounded components

In trying to determine the volume of space to limit the search for an isosurface, the presence or absence of unbounded components is a major factor. Any scalar field containing only the four bounded components combined using any operator except *mirror* will have a finite calculable spatial extent. The volume of space enclosed by this extent will be the only volume which can possibly contain an isosurface.

Although this research project allows the use of unbounded components in the description of a scalar field, this is not the case for all research done in the isosurface modelling field. The use of only bounded components which are combined with a few simple operators simplifies the search for an isosurface. Many of the set theoretic constructive solid geometry systems which use implicitly defined surfaces are careful to avoid the use of

unbounded components [Kalra and Barr 1989]. The use of only bounded components was rejected for the reasons stated in chapter three.

If the scalar field description contains unbounded scalar components, the calculation of a finite extent may be impossible, and is at best more difficult. A variety of unbounded components may have a finite bound depending upon the techniques used to combine them. Recall that the cube component consists of six half space components, each of which is unbounded. Because of the intersection of these half spaces, the cube has a finite extent. Techniques have been developed in the constructive solid geometry field (CSG) that attempt to analyse situations of this type in order to establish extents as well as to cull components which are unnecessary in the evaluation of the scalar field [Woodwark 1988].

Unbounded components are available in the description of the scalar fields. The techniques that are used to combine the components are not guaranteed to yield a bounded scalar field. An alternative to that based upon examination of the scalar field description for finding a finite volume of space to search for an isosurface must be found.

View volume

A volume that can be used in some circumstances is to restrict the search for the isosurface to the view volume, that volume of space which is visible from the graphical camera. As with the viewing calculation for the display of traditional graphical primitives, limits on the 'near' and 'far' viewing planes are set in order to create a finite view volume. If a search volume was able to be calculated from inspection of the SFDL program, it can be intersected with the view volume to yield the final volume of space that must be searched for an isosurface.

Using the view volume as the basis of the search volume has several disadvantages. Every time the camera moves, the isosurface must either be recalculated or augmented in some fashion. This will cause a heavy computation load that will be difficult to meet in order to achieve real time visualisation of the isosurfaces.

A second disadvantage to the technique of limiting the search for an isosurface to the view volume is that some display techniques require that the entire isosurface model be present. Techniques such as ray tracing, radiosity and the calculation of shadows are examples of this. Ray tracing may include reflections of objects that are out of the direct view of the camera. Radiosity calculations may lead to colour bleeding from out of view, and shadows cast by objects which are out of view may be visible.

User specified search volume

It can be seen that in the worst case, for instance ray tracing a reflective scene with some unbounded components (for instance planes and cylinders), that no reliable limits to the search volume can be automatically set. It is necessary therefore to establish arbitrary limits on this search.

The amount of time required to find an isosurface is directly dependent upon the size of the search volume. Therefore, rather than establishing arbitrarily large default limits, the selection of an appropriate search volume is left to the user. This volume can be specified in a number of ways, such as x, y,and z maxima and minima, or as a centre and radius of a search sphere. The first technique is used in this research as it is more elegantly and efficiently implemented.

The need for the user to specify a search volume is unfortunate, however it cannot be avoided as the nature of the scalar field specification is general and may result in isosurfaces that are infinitely large in some circumstances. In the case of an infinitely large isosurface, the user will have to determine which portion to visualise.

4.2.2 Minimum detail size

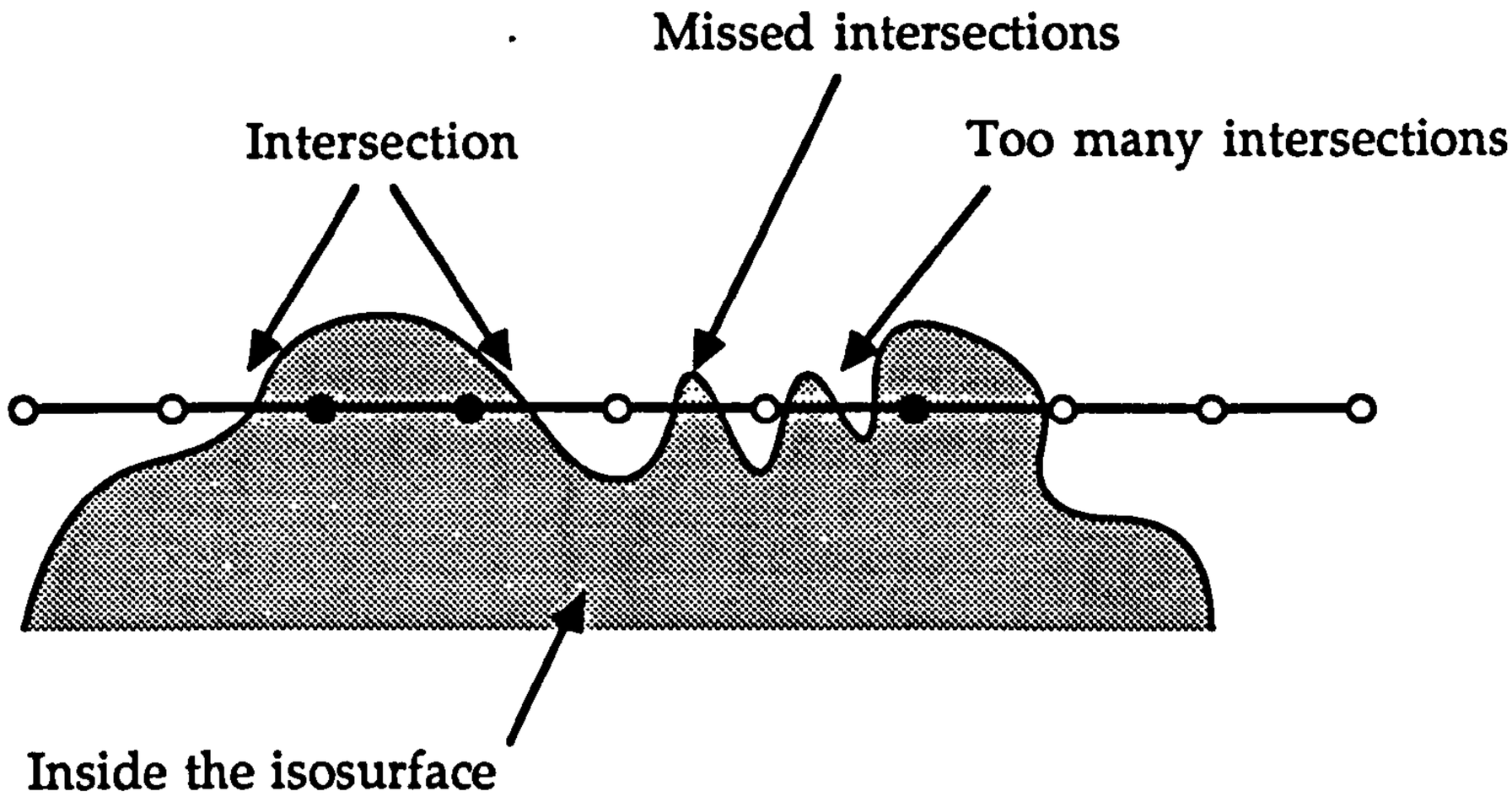
A second problem caused by the fact that the scalar field can be evaluated at any point is that a decision must be made regarding how finely to sample the search volume in the search for isosurfaces. The search for an isosurface

```
real      isovalue          The isosurface value
real      function field()  The scalar valued function
real      size              The increment size along the line
real      length            The length of the line to be sampled
real      n                 The number of samples along line
boolean   inside            Whether or not we're inside surface
vector    P                 The origin of the line
vector    D                 The direction of the line, unit length

n = length / size

if (field(P) ≥ isovalue)  then inside = true
                          else  inside = false

i = 0

while (i < n) begin
    P = P + (size * D)
    if (inside) begin
        Determine if we have crossed outside of the isosurface
        if (field (P) < isovalue) then inside = false
    end
    else
        Determine if we have crossed to inside the isosurface
        if (field (P) ≥ isovalue) then inside = true
    endif
    i = i + 1
endwhile
```

Fig 4.1 Pseudo-code to find isosurface intersections along a line

in a volume can be simplified for the purposes of illustration to the problem of finding the isosurface intersections along a line. A potential solution to this problem is given in figure 4.1.

The success of the algorithm is completely dependent upon whether or not the line is sampled often enough to detect the transitions from being outside the surface to inside the surface, and vice versa. If the line is under sampled features may be missed. If the line is over sampled it will unnecessarily slow the visualisation process. The question of determining the correct sampling frequency is discussed in this section.



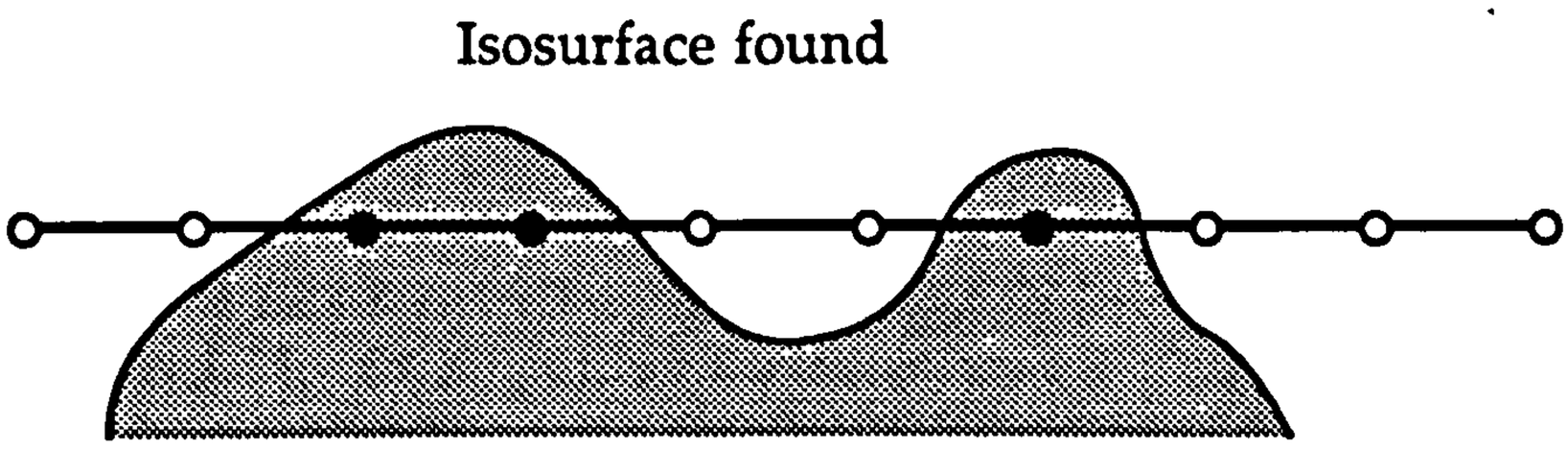


Fig 4.2 Aliasing problem detecting isosurface intersections

Detection of intersections

It is interesting to note that an isosurface intersection is detected by the transition from a scalar field value being greater than the isosurface value to one that is less than the surface value, or vice versa. This condition does guarantee that at least one intersection exists between the two points sampled, and in fact indicates that there are an odd number of intersections between the two points sampled. The lack of a transition between two points does not guarantee the absence of an intersection, but rather indicates the presence of zero or an even number of intersections. The lack of a transition indicates precisely, that at the sampling frequency no intersection was detected. There may in fact be a multitude of intersections that would have been found if the sampling rate were increased. This is an aliasing problem. An example is shown in figure 4.2

Calculation of minimum detail size

If it is possible to analytically establish the minimum detail size present in a scalar field, then this minimum size can be used in determining the spatial increment at which the search for the isosurface is carried out. If the smallest detail present in a scalar field is one inch, then a search for an isosurface carried out at increments of one tenth of an inch would be wasteful. However a search carried out at two inch increments may miss the smallest isosurface detail. In this case, an optimum search would be carried out at one half of an inch, which would be guaranteed to find all of the isosurface details in the search volume. It should be noted that different volumes of space may have different minimum detail sizes, depending upon the contents of the volume.

Limits on the smallest details that are present in a scalar field can be found for simple cases. An example of this is apparent when the description of a scalar field is limited only to spheres being added to a scalar field. In this case the smallest sphere can easily be found by inspection of the SFDL program. This then becomes the smallest detail, and a search conducted at half of this size will find all isosurfaces.

Consider an SFDL program composed of only sphere components with the addition and subtraction operators available. Through the use of the subtraction operation it is possible to generate detail sizes that are much smaller than the smallest sphere present in the description. Consider a sphere subtracting all but a small sliver from a second sphere. The previous technique of calculating the minimum detail size will fail. There is one last technique available in this example to determine the minimum detail size present, that is through the inspection of the relation of the spheres in the SFDL program. An analysis may result in an analytic calculation of the smallest detail that may be present.

Alternative techniques for finding minimum detail size

SFDL as implemented in this research contains twelve different scalar valued components and six ways of combining them. Some of the components may contain random factors, as in the noise component. When using SFDL it is not always possible to analytically determine the size of the smallest isosurface that may be present. Alternative techniques must be found in order to determine the sample size to use in the search for an isosurface.

For raster based displays of isosurfaces, it may be possible to relate the pixel size on the screen to the objects contained in the volume. This is possible for ray-tracing as well as techniques dependent upon the view volume in the calculation of the search volume. This relation between pixel and object size makes it possible to establish a limit on the searching size, as searching at a sub-pixel level for isosurfaces is unnecessary, with a possible exception being made to accommodate anti-aliasing.

Visualisation techniques which operate in world coordinates and produce isosurfaces in world coordinates are unable to determine the size of objects in terms of pixels on the display screen. The graphics camera can be arbitrarily close or far away from an object. A different technique must be found to limit the minimum search size.

A technique proposed by Kalra and Barr in 1989 involves the use of Lipschitz constants. Lipschitz constants are used in relating the rate of change of a field at a point to the possibility of an isosurface intersection within a particular volume. This method guarantees that all isosurface intersections are found, but at the loss of requiring that the Lipschitz constant be defined for each scalar component. Lipschitz constants have been defined for ellipsoids, superquadrics and deformations. Research is ongoing into extending the range of surfaces that are available when using this technique. Although this method holds promise for the future, its use was not considered during this research.

The remaining obvious technique available is for the user to specify the minimum search size. The smaller this limit the longer it will take to calculate the isosurface.

The specification of the minimum detail size for the search of an isosurface is by necessity defined by the user in this research. The smaller the minimum detail size the longer it will take to calculate the isosurface. A small minimum detail size may also produce a better result than a larger one.

4.2.3 Binary or range classification

An SFDL program can be evaluated at any point to obtain a scalar field value. This value can be classified as being either below, above or at the isosurface value. This classification is generally reduced to a binary decision by combining 'at the surface value' with either 'below' or 'above'. The binary classification of points simplifies many of the algorithms with little or no effect on the isosurface produced. The binary classification leads to an isosurface with no real thickness, it is a partitioning of the scalar data into two groups, 'inside' and 'outside'.

An alternative to treating the isosurface as an infinitely thin sheet is to give it a thickness specified in terms of a range of scalar values. For example the surface may be defined as all scalar values between 0.9 and 1.0. This treatment of the surface is useful when dealing with empirical data. Consider for example the representation of a surface of tissue derived from an X-ray computed tomography data set. Because of errors in the sampling process and the varying density of tissue, a binary classification of points may lead to holes being produced in the isosurface. In this case, a more useful classification of the surface is the range of scalar values that constitute a surface.

Techniques for finding isosurfaces using range instead of binary classification have not been investigated in this research. The range classification of points is most useful for empirical data sets, which are not the focus of this research. Visualisation techniques such as volumetric

rendering seem most suited for the display of range classifications of isosurfaces.

A method of simulating the range approach using binary classification that may work in some cases is to generate two isosurfaces, one at the high value and one at the low value of the range. The two surfaces may be combined in the same display to produce a continuous set of surfaces. However, the volumetric approach to visualisation is more suitable than this *ad hoc* solution.

The isosurfaces in this research use a binary classification of points. A scalar value is either:

1) Outside, which is below the isosurface value (also known as 'cold'), or
2) Inside, which is at or above the isosurface value (also known as 'hot').

4.3 Boundary representation

The objects that are the focus of this research are represented in their implicit formulation as an SFDL program. This SFDL program is difficult to visualise directly. Conversion of the SFDL implicit formulation into a more traditional representation is useful for a variety of reasons.

A wide range of visualisation techniques have been developed for the traditional representations in computer graphics. Shadow casting and radiosity are examples of two visualisation techniques, examples of representations are: polygons; cubic parametric meshes; and constructive solid geometry representations. These visualisation techniques are not easily applied to isosurface models in their implicit representation. If implicit models are converted to a traditional representation their incorporation into a more traditional modelled scene is facilitated. The incorporation of isosurface models into traditionally modelled scenes is important in order to quickly realise the potential of the isosurface modelling technique.

In order to generate a boundary representation, the boundary of an isosurface must be found. This search can be conducted using a variety of techniques. The information needed as the basis of the search is the volume

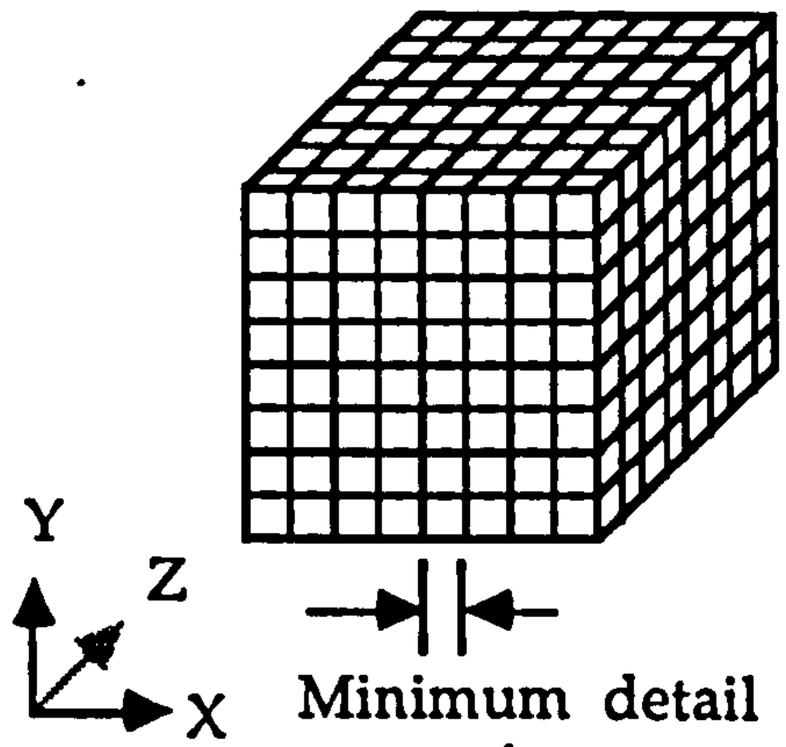


Fig 4.3 Orthogonal lattice subdivided into cuboids

of space which limits this search, the smallest detail size to be detected, and a method of determining the isosurface, as discussed in section 4.2.

The method most commonly used to search this volume is to use a three dimensional orthogonal lattice subdivided into cubes with sides of the minimum detail size. Refer to figure 4.3. There are different methods for the traversal of this lattice.

Often the search for an isosurface and the generation of an isosurface will occur at different resolutions. For example, in order to find a number of spheres in a volume the search must progress at half of their radius' size. The generation of the spheres isosurface, once found, will be at a much smaller size. If a minimum detail size can not be determined, it is safest to conduct the search for and the generation of the isosurface at the same resolution, otherwise small details may be missed.

If the search of the volume is not conducted at the minimum detail size in complex areas of the scalar field, a risk is present that an isosurface may be missed. Complex areas of the scalar field can be considered to be those areas that contain one or more components. An example of a complex area is that left when one sphere is subtracted from a second sphere. The resulting isosurface may be very small.

Through the decomposition of the search volume into small cuboids, the problem of finding the isosurface is greatly simplified. Each cuboid can be quickly classified to be outside the surface, inside the surface or spanning the surface. When some of a cuboids vertices are above and some below the isosurface value the cuboid spans the isosurface.

4.3.1 Polygonal representation

Through various 'divide and conquer' techniques, the search for an isosurface is reduced to the problem of determining the surface topology in a cuboid that straddles the surface. A similar approach to reducing the search for an isosurface to finding the surface topology of the surface within a cube has been made by several researchers [Wyvill, McPheeters and Wyvill 1986; Tindle 1986; Bloomenthal 1987; Lorensen and Cline 1987].

The eight vertices of a cube can each be in one of two states, below the surface value or at and above it. This yields 256 (2^8) possible combinations of the cuboid classification. If the more complex trinary classification were used, this would yield 6561 (3^8) possible combinations. As there are no obvious advantages to trinary classification it will not be considered further in this research.

The sixteen possible classifications of a square, along with the appropriate isosurface topologies are given in figure 4.4. Notice that there is an arbitrary selection of topologies present for cases five and ten. The corners of the square which are 'inside' the isosurface value have been connected for this example. One heuristic rule that is available to resolve this situation is to sample the middle of the square to determine which of the two different topologies to use. However, as with all heuristics, this will fail in some circumstances to select the proper topology. This is shown in figure 4.5.

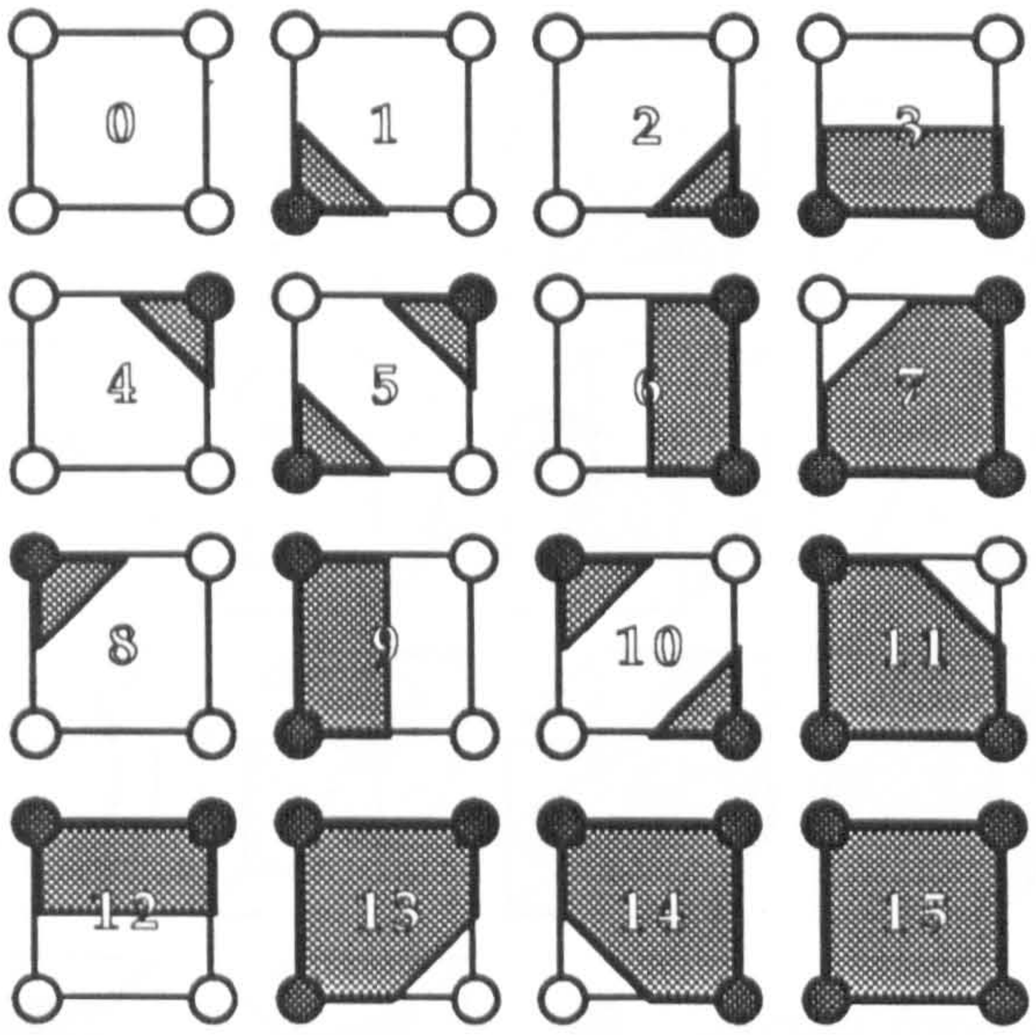


Fig 4.4 The sixteen possible topologies of a square

It was decided throughout this research that it does not matter which solution is used in cases of ambiguity, so long as the solution is used consistently. As there is no guarantee of the accuracy of an isosurface at the minimum detail size selected in this research, the use of a heuristic which sometimes fails did not seem an attractive proposition. The heuristic complicates the algorithm and increases the surface generation time. In cases where the incorrect connection is made, it is clear that the minimum detail size is too large. The reduction of the detail size would elegantly solve the problem.

Of the 256 possible classifications of a cuboid, many are similar. Each of the 256 possibilities can be rotated, reflected or had the classification of its points inverted to obtain one of fourteen different classifications. These are given

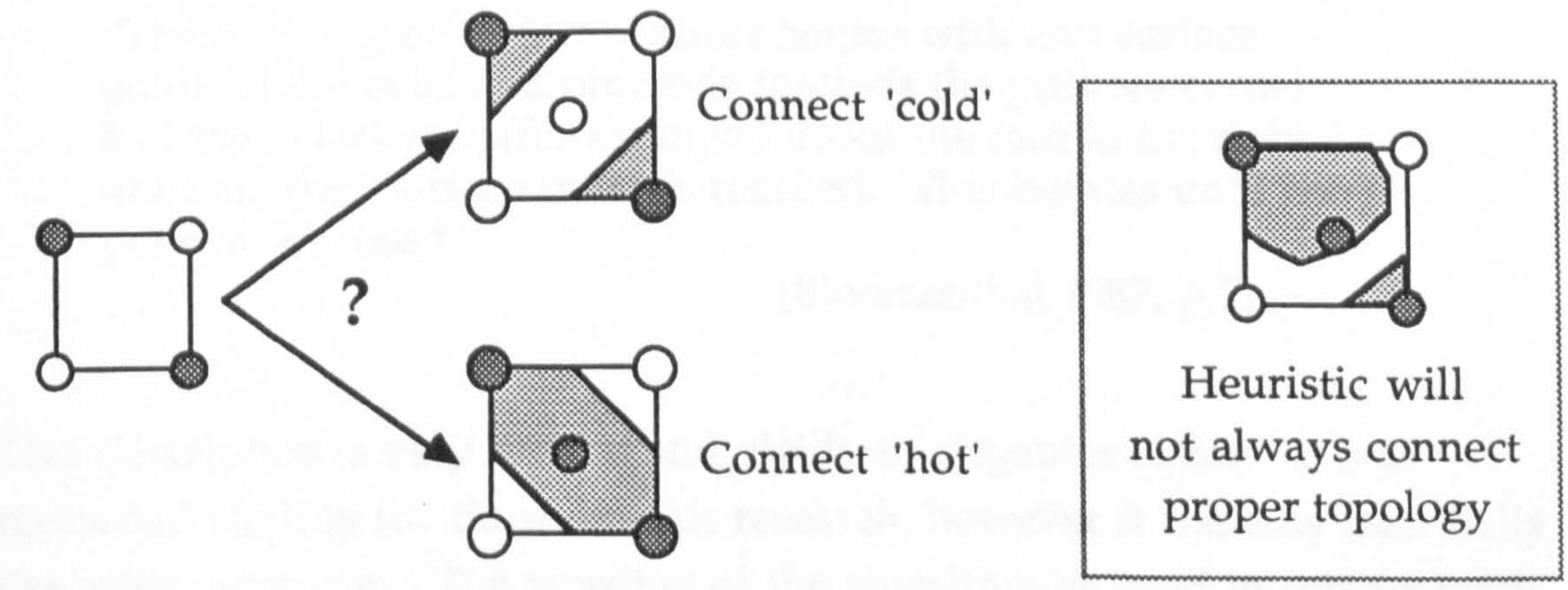


Fig 4.5 Ambiguous surface topology and possible heuristic

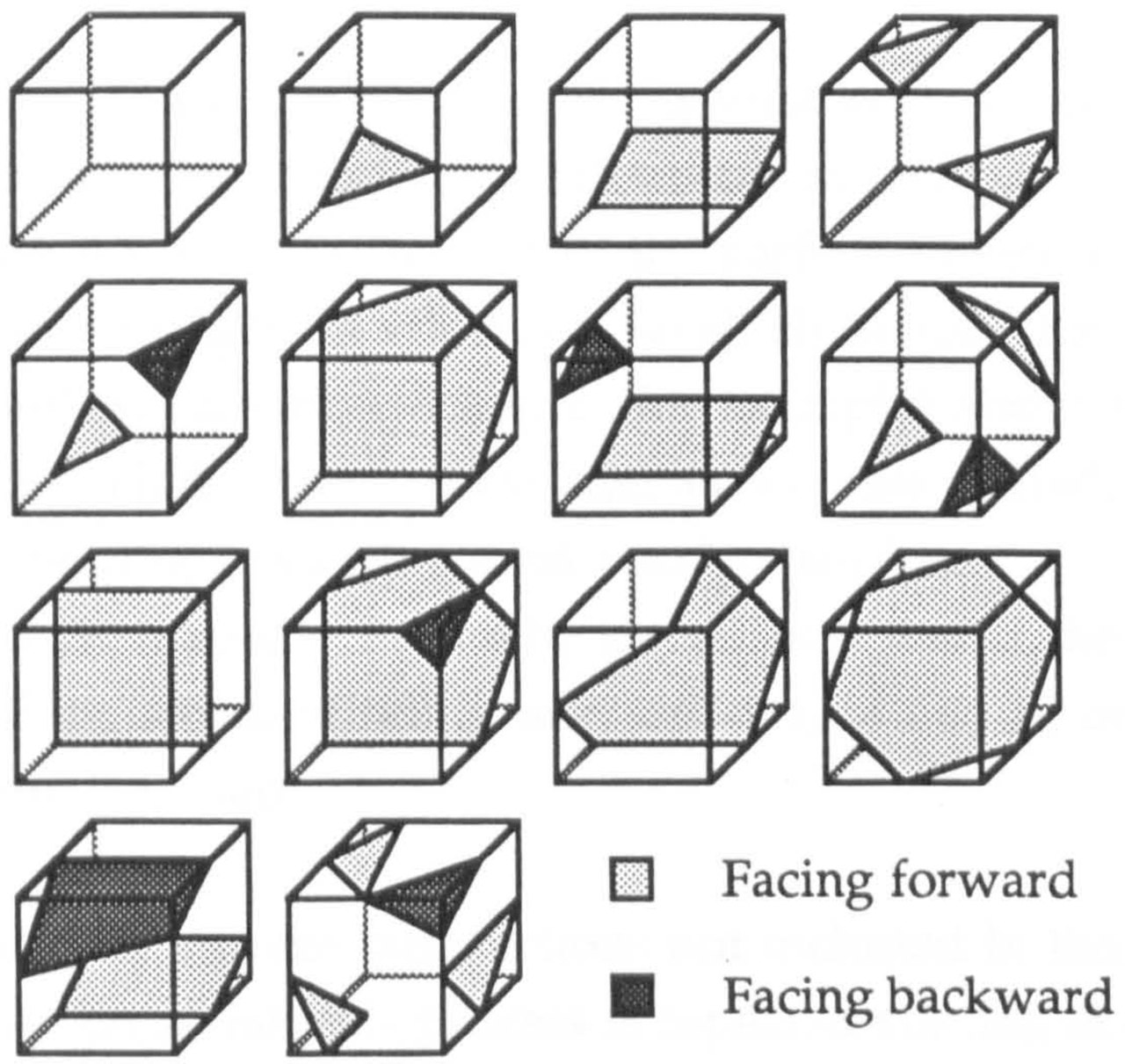


Fig 4.6 Unique isosurface topologies of a cuboid

in figure 4.6.

The most efficient technique found to relate the classification of a cuboid to a surface topology is through the use of a table with 256 entries. Each entry contains a list of the polygons and surface intersections needed to represent the surface topology of the particular cuboid.

There are several algorithmic techniques available to determine the topology of the isosurface within a cuboid. The most elegant found was presented by Bloomenthal in 1987, and is given in its original form as:

> "The ordering of surface vertices begins with any surface point on the cube and proceeds towards the positive corner and then clockwise (for example) about the face to the right until another surface point is reached. This iterates until the polygon is closed."
>
> [Bloomenthal 1987, p.7]

This description is very concise and yields an elegant solution. It was reworded slightly for clarity in this research, however it remains essentially the same algorithm. The wording of the algorithm as used in this research is:

The ordering of surface vertices begins with any surface intersection on the cube. Travel from the surface intersection towards the interior of the surface. There are two remaining edges on which to travel, chose the one which involves moving clockwise (for example) about the face which includes the surface intersection just visited, continue moving clockwise until another surface intersection is reached. Iterate by travelling towards the interior of the surface finding intersections, as above, until the polygon is closed.

If there are any surface intersections not included in the polygonisation so far, this process is repeated starting at one of the surface intersections not included.

The selection of 'clockwise' or 'anticlockwise' in the algorithm will yield the same surface topology from the algorithm. If 'clockwise' is used then the polygons are generated with the vertices in clockwise order when viewed from outside of the surface. 'Anticlockwise' produces the same polygons with their vertices reversed.

One great advantage of the algorithm presented by Bloomenthal is that it is independent of the space filling primitive used. It works equally well for pyramids, cuboids, dodecahedrons or more complex shapes.

The polygonal representation of the isosurface boundary proceeds by first decomposing the scalar field into a group of cuboids. The cuboids that span the surface are found and a surface topology produced. As each interior edge of the lattice is shared by four cuboids, an intersection of an isosurface along an edge may be shared by up to four polygons. For the proper surface to be produced, the same intersection location must be used in each of the polygons which shares an edge. This may be difficult if the lattice contains differently sized cuboids, perhaps produced by different search sizes being used in different portions of the lattice. If neighbouring cuboids are different sizes, then care must be taken in order that no cracks appear in the isosurface. These cracks are highlighted in figure 4.7. Bloomenthal has presented an algorithm to solve this problem for cases when the search volume is decomposed using an octree approach.

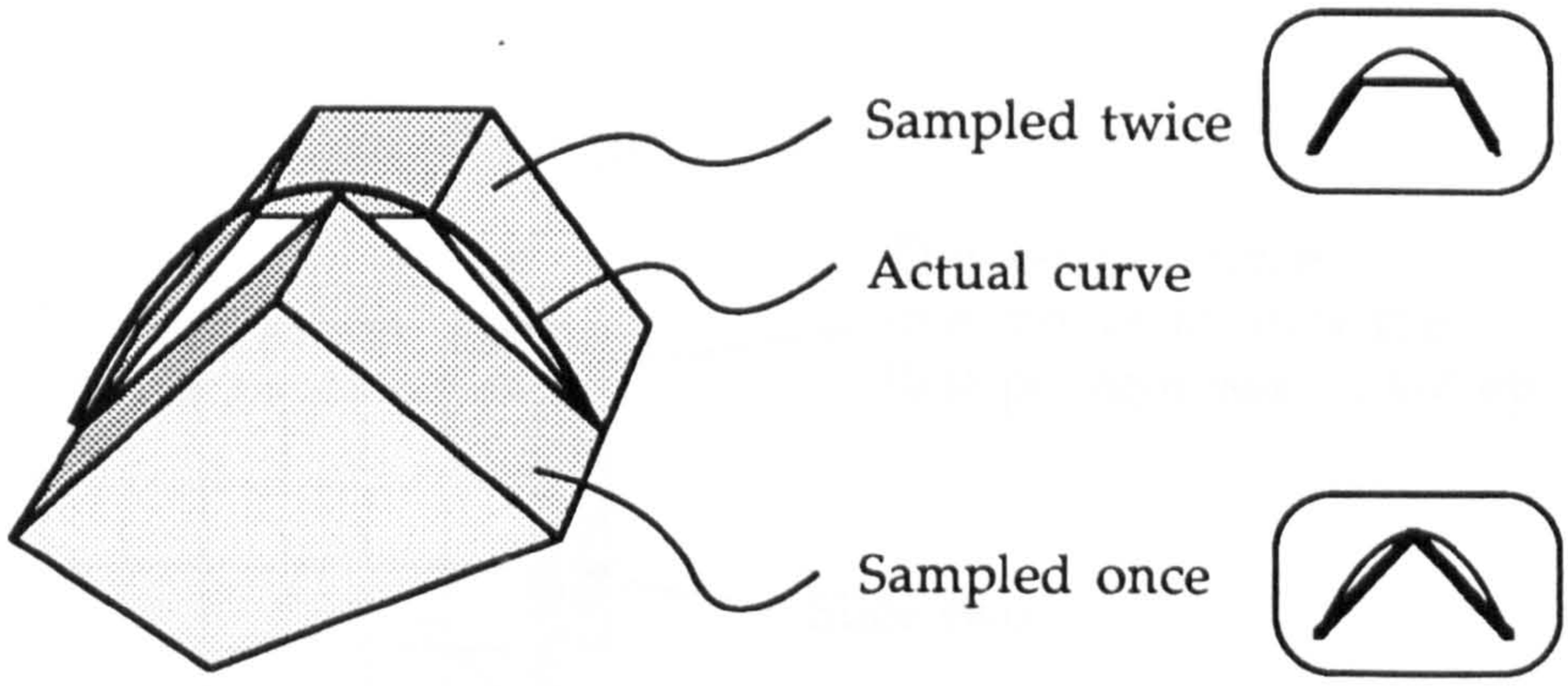


Fig 4.7 Example of surface cracks caused by differing resolutions

4.3.2 Polygonisation method

There are five main steps in the polygonisation technique used in this research for the visualisation of an isosurface. The algorithm gives reliable results at any specified detail size. One fault is that because of the exhaustive searching used, the algorithm tends to take between several seconds and a few minutes to complete. However the algorithm does not re-compute the scalar field for any point more than once and calculates the intersection of the isosurface with an edge only once. In these two senses the algorithm is optimal.

The five areas of the algorithm that will be described are:

1) Method of searching the volume for an isosurface.
2) Classification of the cuboids into a particular surface topology.
3) Representation of the surface.
4) Calculation of the surface intersections.
5) Storage of the polygons generated.

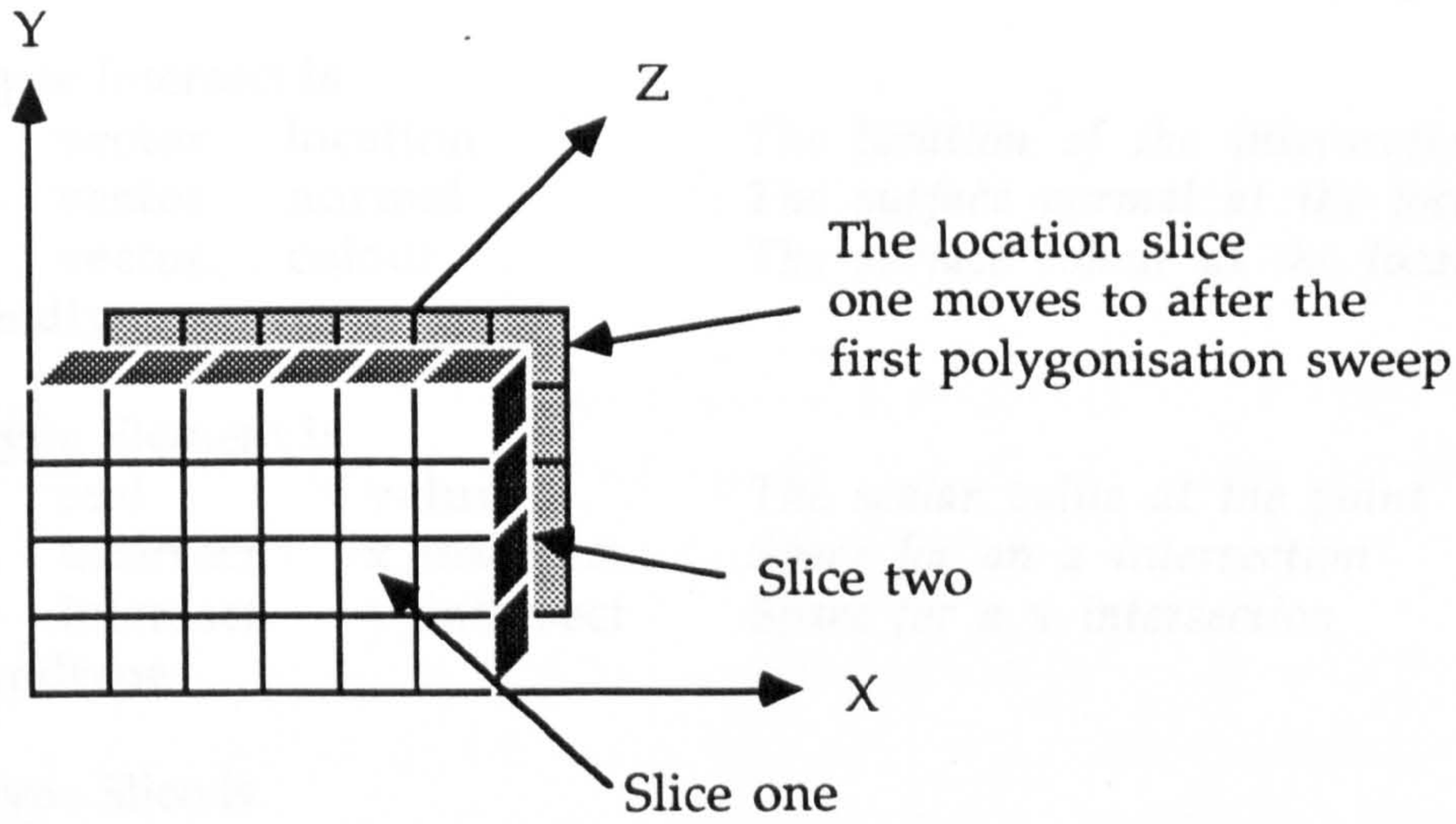


Fig 4.8 Illustration of polygonisation method.

Search technique

The volume of space that must be considered as possibly containing an isosurface is specified by the user of the system, as is the minimum detail size. In order to achieve the best results, the search for isosurfaces is conducted at the minimum detail size. In this way all isosurfaces will be found within the given tolerances.

The value of the scalar field can be determined at any point by the evaluation of the SFDL program. As the number of evaluations of the scalar field presents one of the bounds on the execution time of the algorithm, the number of SFDL evaluations is kept to a minimum. Coherence is utilised at appropriate opportunities.

The search volume is divided into cuboids on a three dimensional lattice at the minimum detail size. These cuboids are traversed from one side of the volume to the other. A two dimensional plane can be thought of as moving through the search volume. The two dimensional plane is aligned with the lattice in order to simplify the calculation of coordinates for the vertices of the lattice on the plane. As cuboids are required, two adjacent planes are needed to traverse the volume each supplying four of the eight vertices in a cuboid. This is shown in figure 4.8. After the polygonisation of

```
type Intersect is
    vector    location        The location of the intersection
    vector    normal          The surface normal at the location
    vector    colour          The surface colour at the location
endtype

type Element is
    real          value           The scalar value at the point
    Intersect     x_intersect     Space for an x intersection
    Intersect     y_intersect     Space for a y intersection
endtype

type Slice is
                                  Two dimensional array of elements
    Element       vertices [x_resolution] [y_resolution]
    vector        origin          Origin of lower left corner
endtype

Slice             s1              Allocate first slice
Slice             s2              Allocate second slice
Slice             slice1          First pointer to a slice
Slice             slice2          Second pointer to a slice
                                  Intra-slice intersection (z)
Intersect         intra[x_resolution][y_resolution]
```

Fig 4.9 Data structures used in searching through a volume

the cuboids between two adjacent planes, the next set of cuboids needs to be polygonised. This new set of cuboids shares one plane with the previous set of cuboids and requires the calculation of the scalar values on a second plane. This process is accomplished by leaving the middle plane where it is and 'hopping' the other over it and recalculating its scalar values. This process continues until the entire volume has been traversed.

The data structure consists of two two-dimensional arrays. The swapping of the arrays in the traversal process is accomplished by swapping two pointers to these arrays. Each element of the arrays can store a scalar value and two intersections. Intra plane intersections are stored in a third array. A summary of the data structure is presented in figure 4.9.

Consider a volume ten by ten by ten cuboids in size. The number of SFDL evaluations required in the search for the surface is 1331[1]. Without the advantage of the coherency in the algorithm and data structure above, 8000 evaluations would be required[2].

With the exception of the boundary cuboids, each edge of a cuboid is shared by three other cuboids. Storage of surface intersections along each edge is also accommodated in the searching data structure.

Another advantage of the data structure is that large volumes of space can be searched as the whole array does not need to be stored in memory at the same time. For instance a 500 by 500 by 500 volume would require the storage of 125 million elements. Using the data structure proposed, two arrays of 250,000 would be required, a considerable savings.

The algorithm used in this research to search the volume is summarised in figure 4.10.

---

[1] Since 11 vertices are needed to represent a volume 10 cuboids across, the calculation is 11 * 11 * 11.

[2] There are 10 * 10 * 10 = 1000 cuboids, with 8 evaluations for each cuboid.

```
Include all types and variables from figure 4.9

global    real      volume_x_size    X size of search volume
global    real      volume_y_size    Y size of search volume
global    real      volume_z_size    Z size of search volume
global    real      volume_z_min     Z minimum of volume
global    real      volume_z_max     Z maximum of volume
global    real      detail_size      Minimum detail size

global    integer   x_resolution     Set for use elsewhere
global    integer   y_resolution     Set for use elsewhere
global    integer   z_resolution     Set for use elsewhere
global    real      z                Current value of z in traversal

procedure evaluate (Slice slice)
    for every (element in slice) do
        element.value = SFDL_evaluate (location of element)
endprocedure

procedure polygonise (Slice slice1, slice2)
    for every (cuboid between the two slices) do
        polygonise_cube (cube)
endprocedure

procedure isosurface
begin
    x_resolution = volume_x_size / detail_size
    y_resolution = volume_y_size / detail_size
    z_resolution = volume_z_size / detail_size

    slice1 = pointer_to (s1)
    slice2 = pointer_to (s2)

    z = volume_z_min

    evaluate (slice1)
    while (z ≤ volume_z_max) begin
        evaluate (slice2)
        polygonise (slice1, slice2)
        swap slice1 and slice2
        z = z + z_resolution
    endwhile
endprocedure
```

Fig 4.10 Pseudo code for searching a volume for isosurfaces

A similar method of searching for an isosurface was presented in 1987 by Lorensen and Cline. In their method, data obtained from a medical scanner is read in a slice at a time and polygonised at the scanning resolution. Due to their method of computing gradients at the vertices of the cuboids, four slices are required in memory simultaneously.

An alternative method of searching for an isosurface was proposed by Wyvill, McPheeters and Wyvill in 1986. In the technique presented a set of initial surface intersections are found through heuristic methods. This was easily accomplished for some circumstances as the scalar fields were constructed of simple components (ellipsoids). The heuristic chosen occasionally fails in circumstances involving negative scalar components. The initial surface intersections are used as seeds and the isosurface is tracked. A hash table is used to store common vertices and to determine if a particular cuboid has previously been processed.

A method by Bloomenthal [Bloomenthal 1987] also requires a set of initial surface intersections. These intersections are used to construct a converging octree. The octree is polygonised in a manner which takes account of unequal sized neighbours.

Both of the techniques above require that initial isosurface intersections be supplied to the visualisation process. These are found through heuristic methods, which may fail in any but the most simplistic of cases. When using the noise or sine components for instance, many isosurfaces may be generated which are disjoint from an original surface. Finding these disjoint isosurfaces is difficult. The visualisation process implemented in this research handles disjoint surfaces elegantly.

Classification of cuboids

Each cuboid is composed of eight vertices, four from each of the two slices. A comparison is made between each vertex and the isosurface value. Using a binary classification, either a one or a zero is assigned to each vertex. This yields an index into a look up table with 256 entries. An index of zero indicates no surface intersection, the cuboid is outside of the surface. An index of 255 indicates the cuboid is completely within the isosurface.

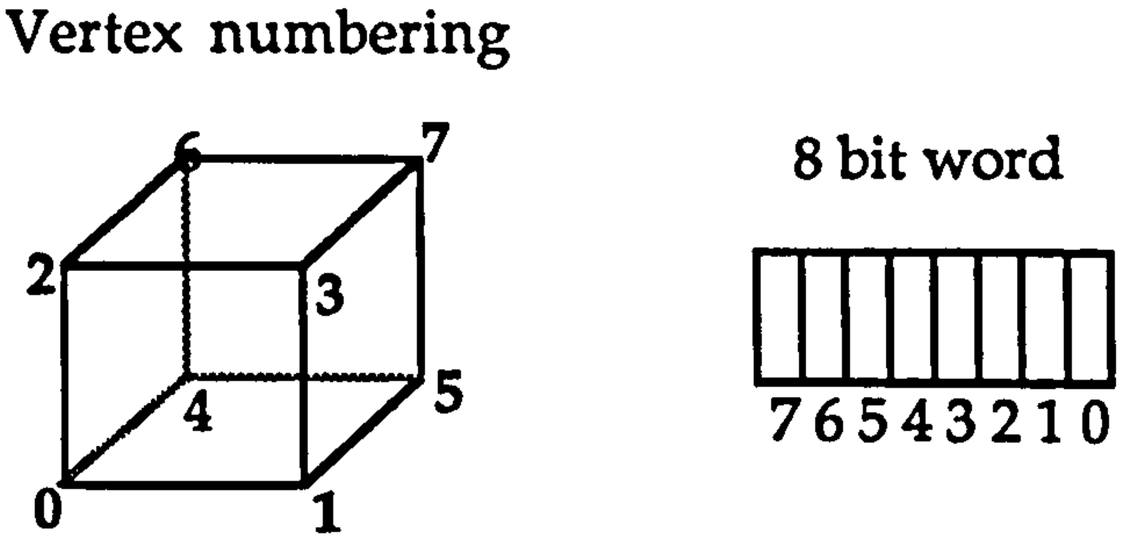


Fig 4.11 Vertex numbering of a cuboid

The calculation of the index can be accomplished using bit operators if the implementation language supports them, otherwise addition of the appropriate values is used. Each of the vertices of a cuboid are labelled with a number from zero to seven. This label corresponds to a bit in an eight bit word. A bit is 'on' if its vertex is inside the isosurface. This is shown in figure 4.11.

Representation of surface topology

Each entry in the table contains a representation of the surface topology for the appropriate cuboid. This topology may consist of a number of separate polygons, each with a variable number of edges. The data structure used to represent this topology is described in figure 4.12. A linked list of edge markers and new polygon markers is sufficient. An edge marker represents the intersection of the isosurface along the edge indicated. A new polygon marker indicates the start of a new polygon.

For the cuboids on the boundary of the search volume, it may be necessary to cap a face of the volume when an isosurface intersects the edge of the volume. This capping operation is carried out at the users discretion. The capping operation is carried out using a similar method as for the main isosurface. The algorithm presented by Bloomenthal is modified and used to initialise the table of topologies for a face. This is used to produce a look up table with sixteen (2^4) entries. Each entry in the look up table consists of a linked list with one of three possible markers: an edge; a corner; or a new polygon marker.

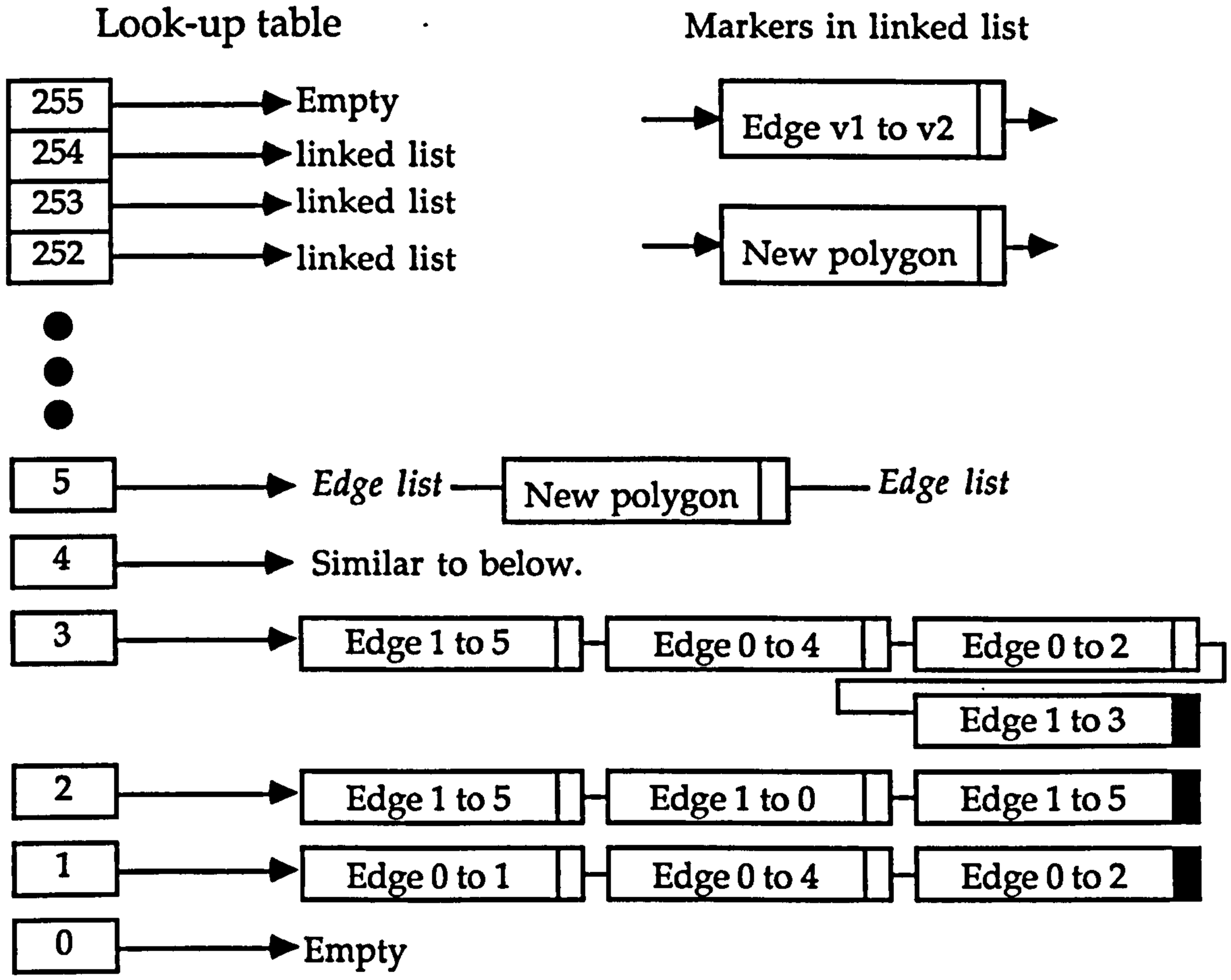


Fig 4.12 Data structure used to store surface topology

Calculation of isosurface intersections

For each edge that requires an intersection we know that there are an odd number of intersections between the two points. The algorithm used to calculate the isosurface intersection assumes there is only one intersection along the edge. If there are more then the algorithm will not fail, but will give an inaccurate result. Only one intersection is returned by this algorithm. In the case where more than one intersection exists along an edge the surface should be resolved using a smaller detail level.

A step wise linear interpolation is used in the intersection calculation. In order to avoid degenerate cases, a limit is imposed upon the number of iterations in the calculation of the intersection, rather than using a predefined tolerance which the intersection must reach. The pseudo-code

for the intersection algorithm is shown in figure 4.13. Notice that intersections are only calculated once for each edge. This intersection is used for each cuboid that shares an edge.

Storage of polygons

The polygons generated by the polygonisation algorithm are stored in a polygon mesh. The polygon mesh consists of two parts, a list of vertices and a list of polygons with pointers into this list. This structure is more compact than alternatives as each vertex is usually used more than once. The edges of each polygon are not stored in the structure, although they could be with a little additional effort. The data structure used is given in figure 4.14.

Efficiency

The algorithm presented is the most efficient technique of exhaustively searching the volume for isosurfaces. Each point on the lattice is only evaluated once. Surface intersections can be approximated to varying accuracies, and each intersection is only calculated once.

While the algorithm may be the most efficient, there are a large number of ways of implementing it. Each implementation may vary in overall efficiency.

The major bound on the algorithm is the amount of time it takes to evaluate the SFDL for every point in the lattice. Complex SFDL programs will increase the time needed to find the isosurface.

The execution time of the algorithm is also affected by the complexity of the surface found. If every cuboid contains a surface, then a considerable amount of time will be spent in the polygonisation and intersection routines.

```
                                        Maximum number of interpolations
global    integer   max_interps
global    function linear_interpolate () returns (vector)

function intersect (Slice slice1, slice2; integer edge)
    returns (vector)
begin
    vector    v1, v2, pt
    real      value1, value2, value
    integer   interps

    if (intersection already calculated)
        return (previous intersection)

    v1 = location of first vertex of edge          Get info from slices
    v2 = location of second vertex of edge

    value1 = scalar value at v1                    Get info from slices
    value2 = scalar value at v2

    pt = linear_interpolate (v1, v2, value1, value2)
    interps = 1

    while (interps < max_interps) begin
        value = SFDL_evaluate (pt)
        if (value ≥ surface) begin
            if (value1 < surface) begin
                swap pt and v2; swap value and value2
            end
            else begin
                swap v1 and value1; swap value and value1
            endif
        end
        else begin
            if (value1 < surface) begin
                swap pt and v1; swap value and value1
            end
            else begin
                swap pt and v2; swap value and value2
            endif
            pt = linear_interpolate (v1, v2, value1, value2)
            interps = interps + 1
        endwhile

        store (pt into appropriate edge intersection)
        return (pt)
    endfunction
```

Fig 4.13 Intersection calculation algorithm

```
type Vertex is
    vector    location          Location of vertex
    vector    normal            Surface normal of vertex
    vector    colour            Colour of vertex
endtype

type Polygon is
    integer       number_of_vertices
    integer       vertex_number[*]      Allocate dynamically
endtype

type Polygon_mesh is
    Vertex        vertices[*]           Allocate dynamically
    Polygon       polygons[*]           Allocate dynamically
endtype
```

Fig 4.14 Data structure to store polygonal mesh

4.3.3 Alternative boundary representations

The polygonal representation is one example of a boundary representation for isosurfaces. It is useful as: it is easily calculated; has a sufficiently pleasing appearance in many cases; is easily incorporated into many graphics systems; and it is often supported by specialised display hardware. Alternative boundary representations are presented briefly.

Parametric mesh

With some difficulty, a parametric cubic mesh can be used as the boundary primitive instead of a polygon. Parametric meshes are composed entirely of parametric patches. Parametric cubic patches are usually four sided, so slight modification will have to be made to either generate four sided meshes in

the representation of the isosurface or to create a patch representation that works for parametric patches with between three and six sides.

Parametric patches can have a specified curvature at each vertex, so the silhouette of this representation is more likely to be smooth. Polygonal mesh silhouettes may be angular even though they represent a smooth surface. This is due to the fact that polygons can only approximate a curved surface.

There is an equivalent amount of work needed in the creation of polygonal or parametric mesh representations. A similar algorithm can be used to generate both representations. The isosurface must be located, and intersections found. Additional information required of the parametric mesh is the curvature of the isosurface at each surface intersection, which can be calculated numerically.

A parametric mesh is produced by the research presented by Gallagher and Nagtegaal in 1989.

Cuboid representation

An alternative to generating a polygonal or parametric mesh from the cuboids in the lattice is to display the actual cuboids which straddle the surface. This can give an approximation to the surface, although possibly at the expense of more polygons. Without any optimisations, each cuboid requires six polygons to fully describe. Adjacent faces of neighbouring cuboids could be generated only once in the same manner as the intersections are only calculated once. The intersection and polygonisation calculations are thus avoided, speeding the surface generation process.

Dot representation

A dot surface can be produced in a variety of ways. A dot approximation to the surface can be generated by displaying a dot at the corner of each cuboid

that spans the isosurface. Dots can be output only once for each location in a variety of ways.

A closer approximation to the isosurface using dots can be created by following a similar algorithm as for polygons, except not producing the polygons, only the vertices.

Dot representations may be easier and faster to display as the techniques required to display a dot on a screen are simpler than those to display a three dimensional polygon. This may not be a consideration if there is specialised hardware to display polygons on the computer displaying the isosurface.

This representation can be used in conjunction with the polygonisation method chosen in this research. The data structure produced for the polygonal representation can be used to either display complete polygons or just the polygonal vertices (dots). A dot display can be used to quickly display a surface, while the polygonal display can be used as a higher quality representation.

Contours

A series of planar contours can be generated following the same basic algorithm as for polygons. Instead of polygonising cuboids using two adjacent slices through the search space, contours can be generated by polygonising the squares present on a slice as it is moved through the search volume. This technique may be useful on monochrome displays in conjunction with a hidden line algorithm. This technique was described by Wright and Humbrecht in 1979.

The algorithm used to polygonise squares is similar to that used in the capping of isosurfaces that are found to extrude out of the search volume, as described in section 4.3.2.

4.4 Volumetric representation

Unlike the boundary representation which seeks to partition the scalar field into two groups, volumetric rendering treats the scalar field as one complete entity. The scalar field data contained within the volume is assigned colour, translucency and other attributes. A visualisation is created directly from this information. No intermediate representation, for example polygons, is used between the scalar field and the image produced.

There are a number of algorithms used to generate volumetric renderings of scalar field data as described in chapter two. The basis of the algorithms are that ranges of scalar values are assigned surface properties. Using a variety of techniques, the scalar data that may effect a pixel is found. An appropriate calculation is then made to find the colour of the pixels, taking into account translucency.

Using volumetric rendering, startling images can be produced which show both internal structure and exterior detail. Images presented in journals and advertising literature of, for instance, the Pixar Image Computer[3] would have been extremely difficult to produce using alternate means.

The use of volumetric rendering in medical applications, once solely a research curiosity, is now being used in diagnosis and in the planning stages of surgical procedures. Medical experts have praised the benefits of the new technology in increasing their ability to accomplish tricky procedures [Frenkel 1989].

While volumetric rendering is becoming increasingly popular in the medical and scientific visualisation fields, it has not been considered as the main visualisation technique in this research for a number of reasons.

Firstly, the objects represented in this research do not have the internal structure which would make volumetric rendering desirable. The boundary of the objects, the manner they interact, and the appearance of their boundaries are the main concerns of this research.

---

[3] Pixar Image Computer is a trademark of Pixar.

Secondly, volumetric rendering can be used to display just a boundary of an isosurface, although at a much higher cost than pure boundary techniques.

Volumetric rendering should not be discounted as a 'scientific' approach which has no other applications. As discussed in chapter two, volumetric rendering is increasing in popularity and importance. Volumetric rendering has possible applications in the entertainment and simulation areas in the display of clouds, fire, storms and similar phenomena.

Recent research into adding texture detail to objects has explored the possibilities of texture volumes, *texels* [Kajiya and Kay 1989; Perlin and Hoffert 1989]. In the research by Kajiya and Kay, volumetric rendering techniques are used to visualise texture volumes which contain fur textures. The research by Perlin and Hoffert incorporates a volumetric rendering technique, called *ray marching*, to visualise complex three dimensional texture volumes.

4.5 Hybrid representation

Research into methods of displaying isosurface models is continuing in several directions. One possible solution is the combination of octrees and voxels. Through the use of octrees, the boundary of an isosurface can be found, and represented as voxels. This may give some of the benefits of volumetric rendering at a lower computational cost. Large portions of the volume may be represented as 'empty' or 'full' voxels. Processing of completely empty or full voxels can be optimised. Boundary voxels, those voxels that contain part of an isosurface, may or may not be used depending upon the particular implementation of a voxel. Boundary voxels can be displayed using various algorithms. A boundary representation can be extracted from voxel data using similar algorithms as those presented earlier in this chapter.

Specialised hardware is being developed to handle the real time display of and interaction with voxel data. This hardware is being applied towards specialised problems, such as the display of medical imaging data [Goldwasser and Reynolds 1987].

4.6 Ray-tracing

Ray-tracing can be accomplished using an intermediate representation, such as a polygonal mesh, as a means of visualising the isosurface. Many references to techniques of ray-tracing polygonal models are available. Ray-tracing of the isosurface directly from the scalar field is possible using a number of techniques. Previous research projects have accomplished this in a number of ways.

The fundamental problem in ray-tracing an implicit surface is the intersection calculation between a ray and an implicit formula. A similar problem was discussed in section 4.2.2 of finding the intersections of an isosurface along a line. Several analytic techniques exist for limited scalar field descriptions [Blinn 1982; Nishimura et al 1985; Kalra and Barr 1989].

The scalar fields used in this research are more general than those used in previous research efforts. The incorporation of random components precludes the use of purely analytic techniques for finding the intersection of an isosurface with a ray. The calculation of intersections of an isosurface along a line suffers from the same problems as the visualisation technique in this research, the specification of the search volume and the minimum detail size.

Boundary representations are often sampled less frequently than in ray-tracing to produce primitives larger than a pixel. This is an optimisation that utilises the coherence of the primitives. The ray-tracing of an image is generally carried out at a much higher frequency. The calculation of every pixel is typical when ray-tracing images.

Ray-tracing is unsuitable for the rapid display of images due to the high computation load demanded by the technique. An alternative technique must be available for visualisation during the initial phases of the design process.

Optimisations of the ray-tracing technique, as well as applying it to more general isosurfaces, was accomplished by Jevans and Wyvill in 1988. In their implementation, a lattice of voxels is first produced which

approximates the boundary of the isosurface. Each voxel is then labelled empty, full or boundary. Boundary voxels contain variations of octrees, called SEV trees, which contain additional boundary information. A majority of the voxels a ray encounters are likely to be completely 'empty' or 'full'. These voxels are quickly skipped and processing continued with the next voxel along the ray. Voxels which contain SEV trees contain sufficient information to calculate an accurate intersection.

A recent technique by Kalra and Barr incorporates the use of Lipschitz constants in order to guarantee the detection of ray intersections with implicit surfaces. In their method, a search volume is recursively decomposed into volumes which do or do not contain an isosurface. This calculation involves the use of the Lipschitz constant in a region as well as the scalar value at the centre and the size of the region. Using the Lipschitz constant it is possible to determine if the isosurface is able to intersect the volume. If it can, then the region is recursively decomposed and each region checked again until a specified tolerance is reached or a region does not contain an isosurface. The intersection calculation between a ray and a region also involves the use of the Lipschitz constant. As techniques for calculating Lipschitz constants for a variety of surfaces become available, techniques based on this method will become more popular.

4.7 Conclusions

There exist various methods for the visualisation of isosurfaces within scalar fields, five methods were mentioned in this chapter. Each of the different methods has distinct advantages and disadvantages. The volumetric rendering technique, while giving the most accurate displays of the internal structure of models requires considerable time to compute. The dot display of an isosurface is the quickest method, but may give a confusing display for relatively complex surfaces.

There are no fundamental problems inhibiting the display of isosurfaces. Several of the solutions proposed result in adequate images. The main area for improvement is in the amount of time required to display complex scalar fields. This problem can be approached in several ways:

1) Computers are generally increasing in computational speed. One approach to decrease display time is to use increasingly powerful computers. Although this is a trivial solution, it may also be realistic. For example the computers used during this research execute approximately two million instructions per second. The latest version of the processor used in the computer is rated at ten times that performance. A similar increase in performance is available for floating point calculation [Thompson 1990]. More powerful processors are currently available.

2) Implementation of the display algorithms is suitable for a parallel environment. Distribution of the task across many processors would decrease the display time.

3) Several optimisations are available in the display process, depending upon the content of the SFDL program. Inspection of the SFDL program could lead to several heuristic rules being used. One such heuristic is the calculation of the search volume in limited circumstances for boundary representations.

Volumetric rendering is a visualisation technique that will be used with increasing frequency in the future. Present applications of medical and scientific visualisation are adequately served by a variety of techniques, volumetric rendering being an important one. The incorporation of volumetric rendering along with rendering of traditional models in the same scene will lead to the display of complex phenomena in a realistically rendered image. Fire, clouds and storms are examples of phenomena that can be added to scenes using a combination of volumetric rendering and traditional rendering. The same phenomena could be added to scenes using boundary representations, although with a different effect.

The method proposed in this research to solve the visualisation task is robust and easily implemented. Using this technique, isosurface modelling could easily be incorporated into traditional computer graphic modelling and animation systems.

Chapter 5

Isosurface appearance

5.1 Introduction

The basis of the isosurface modelling technique is the specification and visualisation of isosurfaces in scalar fields. Using the techniques developed in chapter three it is possible to describe a wide range of isosurfaces. In chapter four several techniques were proposed for visualising the isosurfaces within the created scalar fields. The treatment of attributes that affect the appearance of an isosurface will be discussed in this chapter.

The appearance of an isosurface is influenced by factors such as colour, translucency, lighting calculations and various texture mapping techniques. These, and other factors, are referred to as the surface attributes of an object. Through the specification of the surface attributes an isosurface will appear more realistic and visually interesting. Established techniques exist for handling surface attributes in conjunction with the traditional modelling representations in computer graphics. The isosurface modelling technique presents a unique potential for extending the ways in which surface attributes are handled.

One unique aspect of the isosurface modelling technique is that the shape and the number of isosurfaces present in a model is dependent upon the

position of each of the scalar components. As the scalar components change so will the manner in which the isosurfaces interact. As the number or nature of the isosurfaces in a scene change, the surface attributes must also change in a consistent manner.

If all of the scalar components in a scene have the same surface attributes then it follows that the resulting isosurfaces will have those attributes also, regardless of the position of the components. For example, if all of the scalar components are red then it follows that the isosurfaces will also be red. Techniques for combining the attributes of scalar components for which this relation does not hold are also discussed.

The main topic discussed in this chapter is the determination of the surface attributes for any point on an isosurface. The point may be influenced by several scalar components with different attributes. There is often no particular 'best' approach to the problem, in such a case several approaches for the solution of the problem are presented.

5.2 Colour

The problem of determining the colour of a point on an isosurface influenced by two dissimilar components can best be presented with a simple example. Consider that:

> Two spheres are spaced slightly apart and combined using the addition operator to yield one isosurface. One of the spheres is red and the other is green. What colour is the resulting isosurface?

There is not one correct answer to this question, the answer depends largely upon which phenomena the isosurfaces are meant to represent. Three answers are immediately apparent:

1) The entire isosurface is yellow. This assumes that the two spheres simulate drops of coloured water. The water would mix completely

when the drops come into contact, therefore the entire isosurface will change colour and become yellow[1].

2) One side of the isosurface is red, the other is green. This loosely simulates the situation if the two spheres were made of a malleable solid. There would be little mixing of the solids and a division would be visible between the red and green portions.

3) One side of the isosurface is red, the other side is green and in-between is a smooth transition to yellow. This loosely resembles two semi-solid balls which exhibit limited mixing. This technique tends to give the most aesthetically pleasing appearance, although this is of course a value judgement.

It is obvious that the metaphor used to represent the interaction of the two spheres is fundamental for the correct interaction of surface attributes. Each of the three metaphors discussed above leads to a different result. Each of the metaphors is easily implemented, posing no problem to their incorporation as a solution to this problem. More complex metaphors are easily found. One example is:

4) The merging of the two spheres simulates spheres merged by a sculptor pushing material around into the depicted shape. As the material is not thoroughly mixed, just rearranged, the surface appearance is mainly red and green, with streaks of red, green and yellow throughout the region where the two spheres interact.

This last metaphor would be difficult to specify and implement, which is however no reason to ignore it. The metaphor represents a process that would be useful if it were implemented. However, the last metaphor and even more complex metaphors are not considered further in this research.

The implementation of the first three metaphors are outlined in the order they were presented. Assuming N represents the number of components, the solutions are:

---

[1] The additive colour system, the system representing light, is used in this research for purposes of efficiency. The subtractive primaries, as used in print, present an alternative colour system not explored, although the addition of this system would be trivial. One may also use the HSV, CIE or alternative colour system.

1) The average of all components.

$$\text{Colour of isosurface} = \frac{\sum_{i=1}^{N} \text{Colour of component}_i}{N}$$

2) The colour at the point P is that of the strongest component:

let Comp_i be the component with the largest contribution at point P.

Colour of point P is the colour of Comp_i

3) The colour at point P is the weighted average of the component colours:

let $F_i\,(P)$ be the contribution of component i at point P
let C_i be the colour of component i

$$\text{Colour of point P} = \frac{\sum_{i=1}^{N} F_i\,(P) * C_i}{\text{iso-surface value}}$$

The scalar fields in this research are generated using SFDL. SFDL allows a wide range of scalar fields to be produced using a number of scalar components and combination operators. If only the addition operator is used with positive components the above solutions may be sufficient. However extensions must be made to handle situations that are possible when describing scalar fields in SFDL.

It is possible that a negative scalar component value will be taken into account when calculating the scalar field values. One example of this is when the noise component is being used to produce ripples in the surface of a sphere. In this case, the noise component will vary between negative and positive values, their magnitude depending upon the size of the desired ripples.

A potential problem when dealing with negative scalar component contributions can be demonstrated by the presentation of a specific situation:

> A green noise component is being added to a red sphere. The values of the noise component range between -0.5 and 0.5. What colour is the resulting sphere?

There are a number of potential solutions to this problem, some of them are:

1) The entire sphere is red. As the noise component does not cross the isosurface threshold, it is not directly visible. It is only visible as an effect on the other components. Therefore the colour of the noise component is not taken into consideration.

2) The bumps in the sphere are green, the troughs and the rest of the sphere is red. As the bumps are caused by the noise they must be the same colour as the noise. When troughs are cut into the red sphere they reveal more of the red sphere.

3) The bumps and troughs of the sphere are yellowish, the rest of the sphere is red. The absolute magnitude of the contribution of the noise component to the sphere is used in an averaging calculation for the colour. This reflects the amount of change introduced by the noise component.

4) The bumps are yellowish (red and positive green), the troughs contain red and some negative green. Negative colour contributions can be resolved to actual screen colours in a variety of ways. The clamping of the colours to zero is the easiest method of solving the problem of negative colour. It is also possible to find the range of colours present in an image and map this range to the colours available.

The first solution proposed avoids the problem entirely. If the range of the noise component is changed to 1.2 and 0.8, the problem in this example remains unsolved. The proposed solution also fails to find a colour for a surface produced when using only noise components that are below the surface value, for instance the addition of two noise components ranging

between -0.8 and 0.8. In this case the two noise components will occasionally combine to create a value greater than the isosurface value.

The second solution roughly follows a sculptural approach, with material being added (green) and material being taken away to reveal the surface underneath.

The third solution is roughly similar to the semi-solids metaphor presented as the third solution in the previous example, with the absolute magnitude of the contributions being taken into account.

The implementation of the fourth solution will introduce undesirable behaviour into the system. Clamping of the negative colour values to zero will introduce discontinuities, while the remapping of the colour range will cause strobing problems. Remapping of the colours may cause the colour of an entire surface to lighten and darken throughout an animation or between separate animations.

In order to simplify the design of isosurface models a choice is made between the alternative methods of handling surface attributes where a selection is present. This selection is made in order to reduce the complexity of the modelling system implemented, and is arbitrary. Alternative implementations may allow the full range of techniques for handling surface attributes discussed in this research, at the discretion of the implementer.

The colour of an isosurface in SFDL is closely related to the description of the shape of the isosurface. It is possible to disassociate the specification of colour from the surface description using techniques that will be described in section 5.5. The remainder of this section assumes that colour is directly related to the shape of an isosurface, and is specified by applying a colour to each scalar component and following the rules outlined below to calculate a final colour at any given point.

The treatment of colour by each of the scalar operators in SFDL is discussed.

Addition

A weighted average of all of the components that have a positive contribution to the scalar field is taken. This has the effect of using the colour of the positive component in positive - negative interactions, and mixing the colours in positive - positive interactions. A first attempt at solving this problem, ignoring negative contributions, is given as:

$$\text{The colour of point P} = \frac{\sum_{i=1}^{N} F_i(P) * \text{Color}_i(P)}{\text{scalar field value}}$$

The above example is sufficient if only positive scalar field contributions are present, otherwise it will may lead to negative values being calculated for colour.

To handle negative scalar field contributions, the following can be used as a first attempt:

$$\text{The colour of point P} = \frac{\sum_{i=1}^{N} \begin{cases} F_i(P) * \text{Color}_i(P) & ; \text{if } F_i(P) > 0 \\ 0 & ; \text{otherwise} \end{cases}}{\text{scalar field value}}$$

The above example will not always properly calculate colour. For instance, assume the isosurface value is one, and there are three contributions to the scalar field: one, one and negative one. Adding these three contributions gives a scalar field value of one, indicating that the point is on the isosurface. Following the above calculation for colour however may yield a colour that is over saturated by a factor of two. What is needed is to divide by the summation of the positive scalar field values used in the addition of the colours. This can be expressed as:

```
type COLOUR is
    vector    colour      The colour in RGB components
    real      value       The scalar value used in colour calculation
endtype

add col1 and col2, returning the result.
    col1 is the value calculated previously,
    col2 is the new value

function addition (COLOUR col1, col2) returns (COLOUR)
begin
    COLOUR     result

    if (col2.value ≤ 0) return (col1)

    result.colour = col1.colour + (col2.colour * col2.value)
    result.value = col1.value + col2.value

    return (result)
endfunction
```

Fig 5.1 Implementation of colour in the addition operator

$$\text{The colour of point P} = \frac{\sum_{i=1}^{N} \begin{cases} F_i(P) * Color_i(P) & ; \text{if } F_i(P) > 0 \\ 0 & ; \text{otherwise} \end{cases}}{\sum_{i=1}^{N} \begin{cases} F_i(P) & ; \text{if } F_i(P) > 0 \\ 0 & ; \text{otherwise} \end{cases}}$$

This can be partially implemented as described in the pseudo-code in figure 5.1. The colour will be adjusted at the completion of an SFDL execution by dividing by the scalar value calculated. This is demonstrated later in this section, as figure 5.6.

Subtraction

In order to remain consistent with the implementation of addition, the subtraction operator will not affect colour when dealing with positive values. Subtracting a positive value becomes the same as adding a negative value. There is however the possibility of subtracting a negative scalar field value. Arithmetically this double negative yields a positive value. A decision must be taken whether or not the allow the 'double negative' effect of the subtraction operator.

The subtraction operator is used mainly as a means of changing the shape of the isosurfaces. Scalar components will be subtracted from other scalar components to yield new surfaces. The only components that are able to directly yield negative scalar values are presently the noise and sine components. Using these components it is possible to have a situation where a negative value is subtracted from a positive value, 'double negative.' The subtraction operator could be implemented to handle 'double negative' cases in a manner which makes it behave the same as the addition operator does. Alternatively, the parameters of the components could be rearranged and addition substituted to achieve a similar effect. For instance, instead of a noise component which varies between 0 and 0.5 being subtracted, a noise component ranging between -0.5 and 0 could be added.

The implementation of the 'double negative' effect of subtraction can be made to be consistent with addition, but this adds no additional functionality to the system. By contrast, if the subtraction operator always disregards colour, then this feature can be used to gain flexibility. For example, a noise component with values between -0.5 and 0.5 can be subtracted to alter the shape of an isosurface, but not change its colour. If the noise component were added, the shape would undergo a similar change but the colour would also be changed at each positive contribution.

Again, the choice of one of the possible methods of implementing the subtraction operation is arbitrary. The implementation of subtract in SFDL does not alter the colour component. This is described in figure 5.2.

```
subtract col2 from col1

function subtract (COLOUR col1, col2) returns (COLOUR)
begin
    return (col1)
endfunction
```

Fig 5.2 Implementation of colour in the subtraction operator

Maximum (union)

The maximum operator has the effect of finding the union of scalar components, it is a method of building up an isosurface. In this respect it is similar to the addition operator. In order to be consistent, the maximum operator therefore alters the colour of the point according to the scalar field contributions.

The largest of two scalar field values is used as the scalar result of the maximum operator. The resulting colour of two components when combined using maximum can be handled in at least two ways:

1) The colour of the component with the largest scalar field value is used as the resulting colour. This will yield a sharp boundary between the colour of the two components.

2) A weighted average of the colour of the two components is used, weighted upon the scalar field values.

Again, the selection of one of the above techniques is arbitrary. The second approach has the advantage that it loosely resembles the manner in which addition would behave, mixing the two colours to create a smooth transition. However, the effect created by the first approach is not achievable in any other manner within SFDL, if it is not chosen then functionality will have been lost.

Using the most popular technique of visualisation in this research, the polygonal representation, the first approach will introduce details that will

```
apply the maximum operator to two colours

function maximum (COLOUR col1, col2) returns (COLOUR)
begin
    COLOUR          result
    real            total, t

    if (col2.value < 0) return (col1)

    total = col1.value + col2.value

    if (total is approximately equal to zero) return (col1)

    Calculate proportion of each colours contribution to result
    Maintain the relation between colour and value

    t = col1.value / total
    result.colour = (col1.colour * t) + (col2.colour * (1 - t))
    result.value = (col1.value * t) + (col2.value * (1 - t))

    return (result)
endfunction
```

Fig 5.3 Implementation of colour in the maximum operator

be difficult to represent. This problem with the polygonal representation will be discussed in section 5.6.

In SFDL the second approach, a weighted average, is used in the implementation of the maximum operator. For consistency with addition negative scalar components do not make a colour contribution. The implementation is presented in figure 5.3.

Minimum (intersection)

The minimum operator is similar to subtract in that it offers a means of reducing the area of an isosurface.

The manner in which minimum is normally used is to trim a scalar field, leaving only the portions of interest. For instance the spokes of a wheel can be created quickly by adding several rotated cylinders and intersecting them

```
Apply minimum to col1 and col2

function minimum (COLOUR col1, col2) returns (COLOUR)
begin
    return (col1)
endfunction
```

Fig 5.4 Implementation of colour for the minimum operator

with a sphere of a suitable radius. In this case the minimum operator is used purely as a method of trimming the isosurface, the colour should not be affected.

It is expected that trimming will be the most common use of the minimum operator. It has thus been implemented so that the first component encountered is used for the colour calculation, similarly to the subtract operator. This implies that the operator is not commutative in its treatment of colour. For example, when dealing with colour:

$$A \boxed{\min} B \neq B \boxed{\min} A$$

The implementation of minimum is given in figure 5.4.

Mirror

As the mirror operator is a unary operator, it does not deal with two colours, only one. Therefore no change of colour is needed by the mirror operator. It is possible that a positive field value will become negative after this operation, however since colour is unaffected by this operation, it will remain positive. This operation can not lead to negative colour being produced.

```
Mix the two colours corresponding to parameter m.
   The value of m is between 0 and 1.

function mix (COLOUR col1, col2; real m) returns(COLOUR)
begin
   COLOUR     result

   if m = 0 use col1, if m = 1 use col2, otherwise use a mixture

   result.colour = (col2.colour - col1.colour) * m + col1.colour
   result.value = (col2.value - col1.value) * m + col1.value

   return (result)
endfunction
```

Fig 5.5 Implementation of colour in the mixture operator

Mix

The mix operator offers a method of mixing between two scalar field values. The treatment of colour will proceed along the same lines. Although it is possible to mix between a negative and a positive scalar field value, this will not create a negative colour value as both of the colour values are guaranteed to be positive. The implementation of mix is described in figure 5.5.

Final calculation of colour

The final calculation for colour must make an adjustment by dividing by the scalar value that was calculated. The only operator that could possibly yield a negative scalar value is the mirror operator. However, the mirror operator will not affect the scalar value stored in the colour structure. The mirror operator only affects the scalar value returned by an SFDL program, which is stored in a separate location. There will be no negative values for

```
function fix_colour (COLOUR col) returns (COLOUR)
begin

    Avoid division by zero

    if (col.value is approximately equal to zero) return (col)

    col.colour = col.colour / col.value

endfunction
```

Fig 5.6 Final adjustment of colour

colour, as each of the operators result in only a positive value for colour. The final calculation of colour is presented in figure 5.6.

Several examples of the implementation of colour are given in figures 5.7 through 5.10.

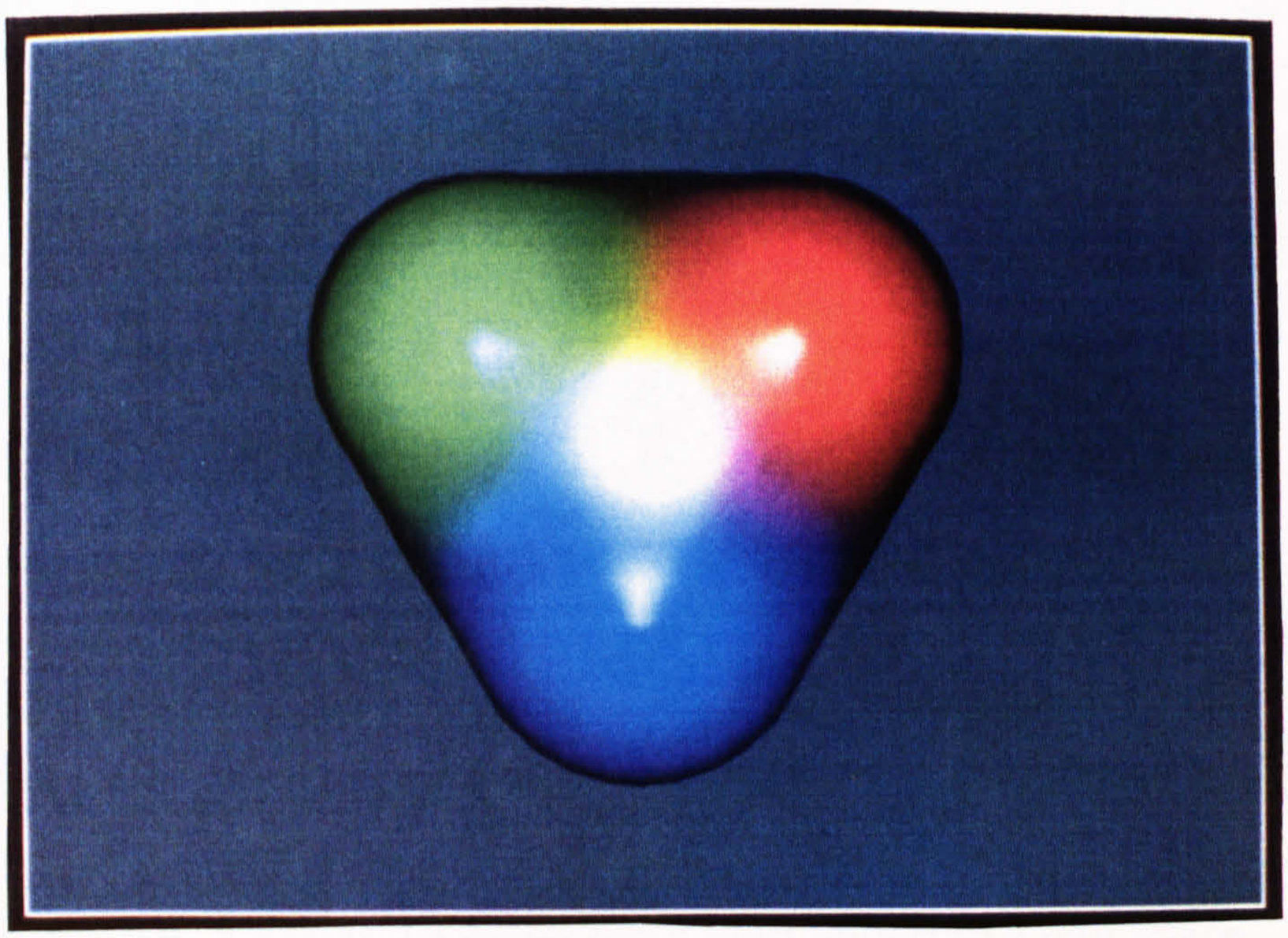


Fig 5.7 A red, a green and a blue sphere merging

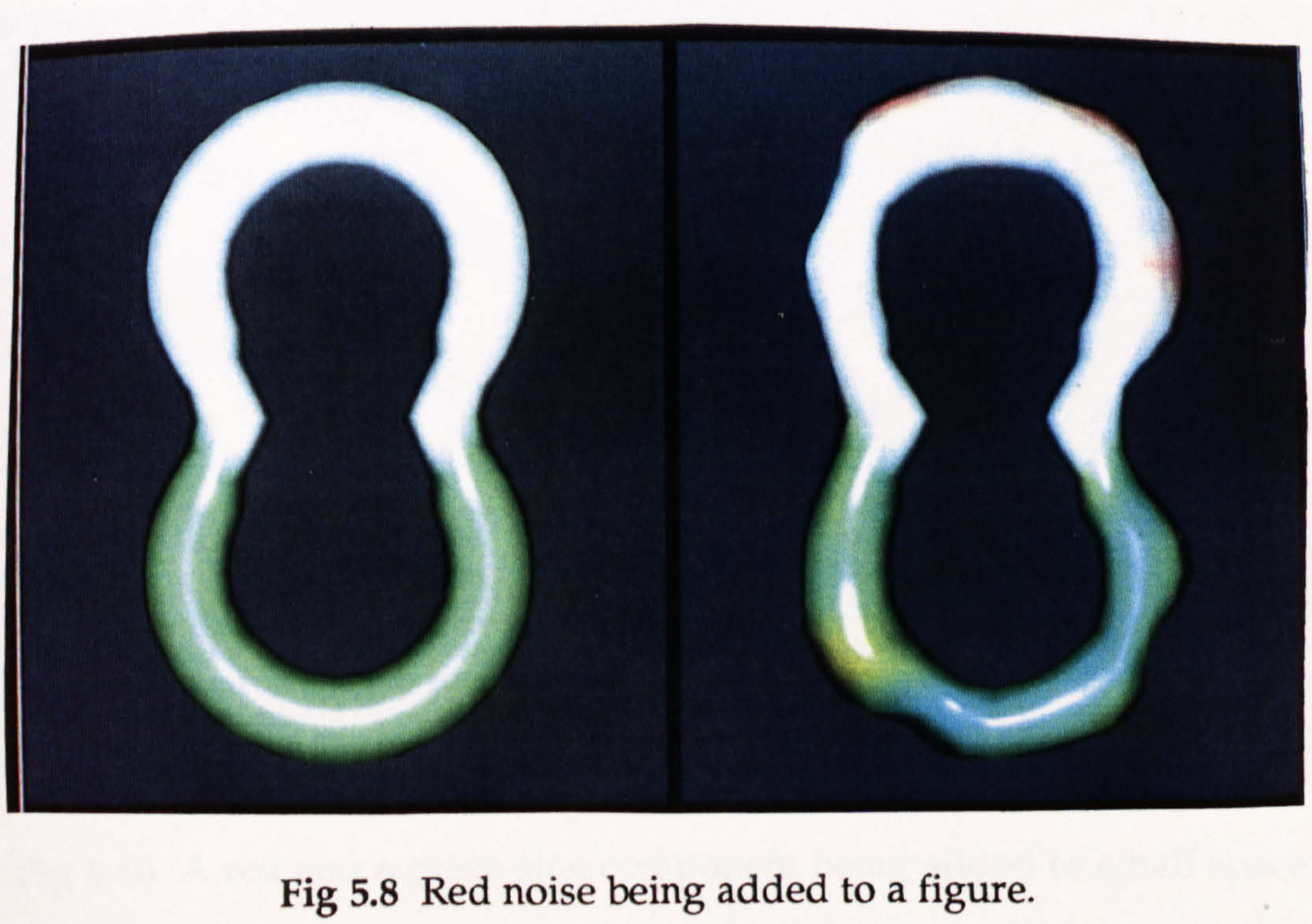


Fig 5.8 Red noise being added to a figure.

The figure was created using two torii and subtracting a sphere from the middle. Notice that the subtraction of the sphere does not affect the colour. The noise varies between positive and negative values, the only contribution to colour are at the positive values

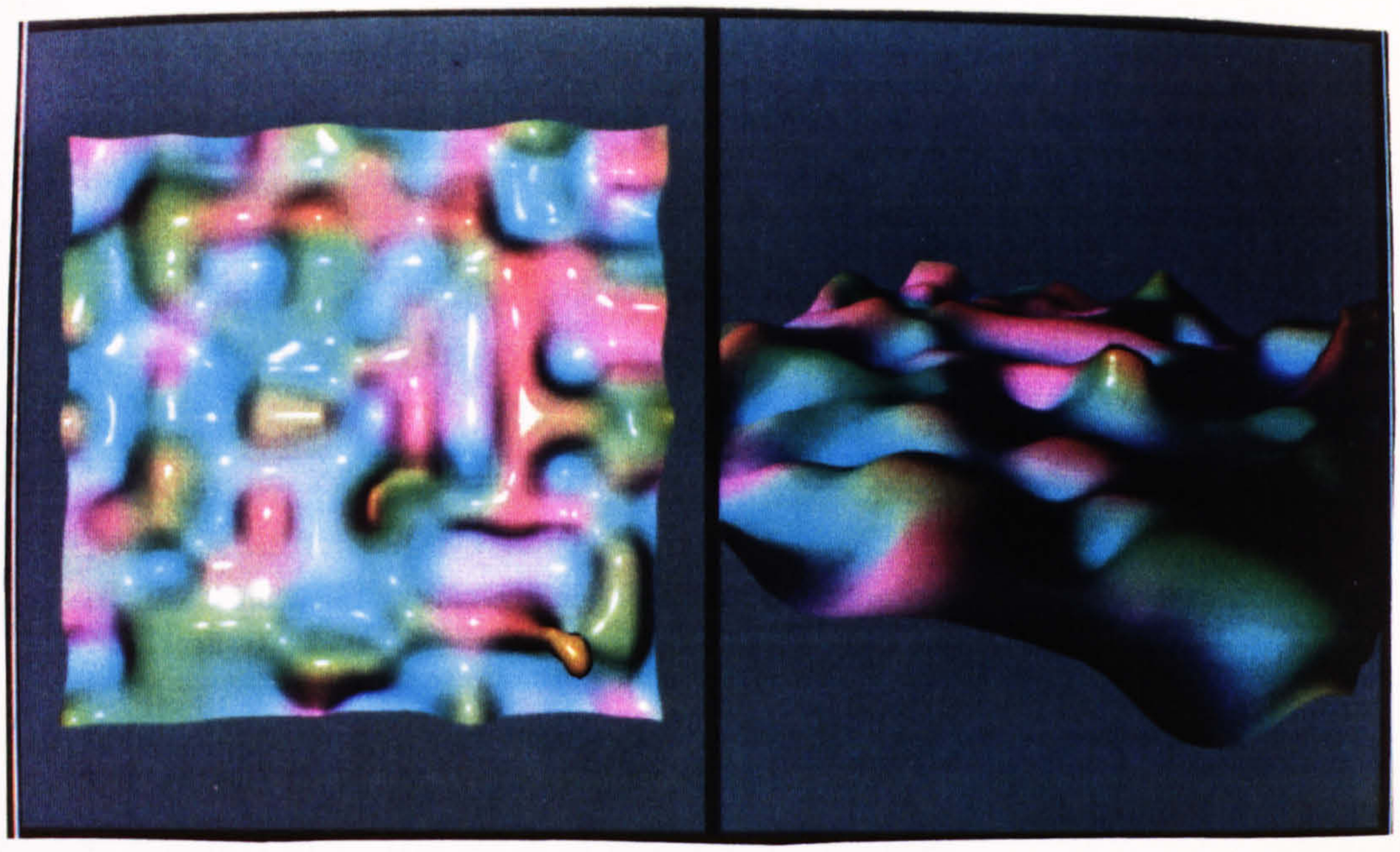


Fig 5.9 A red and a green noise field added to a half space

Notice that the search volume limits the half space to resemble a square

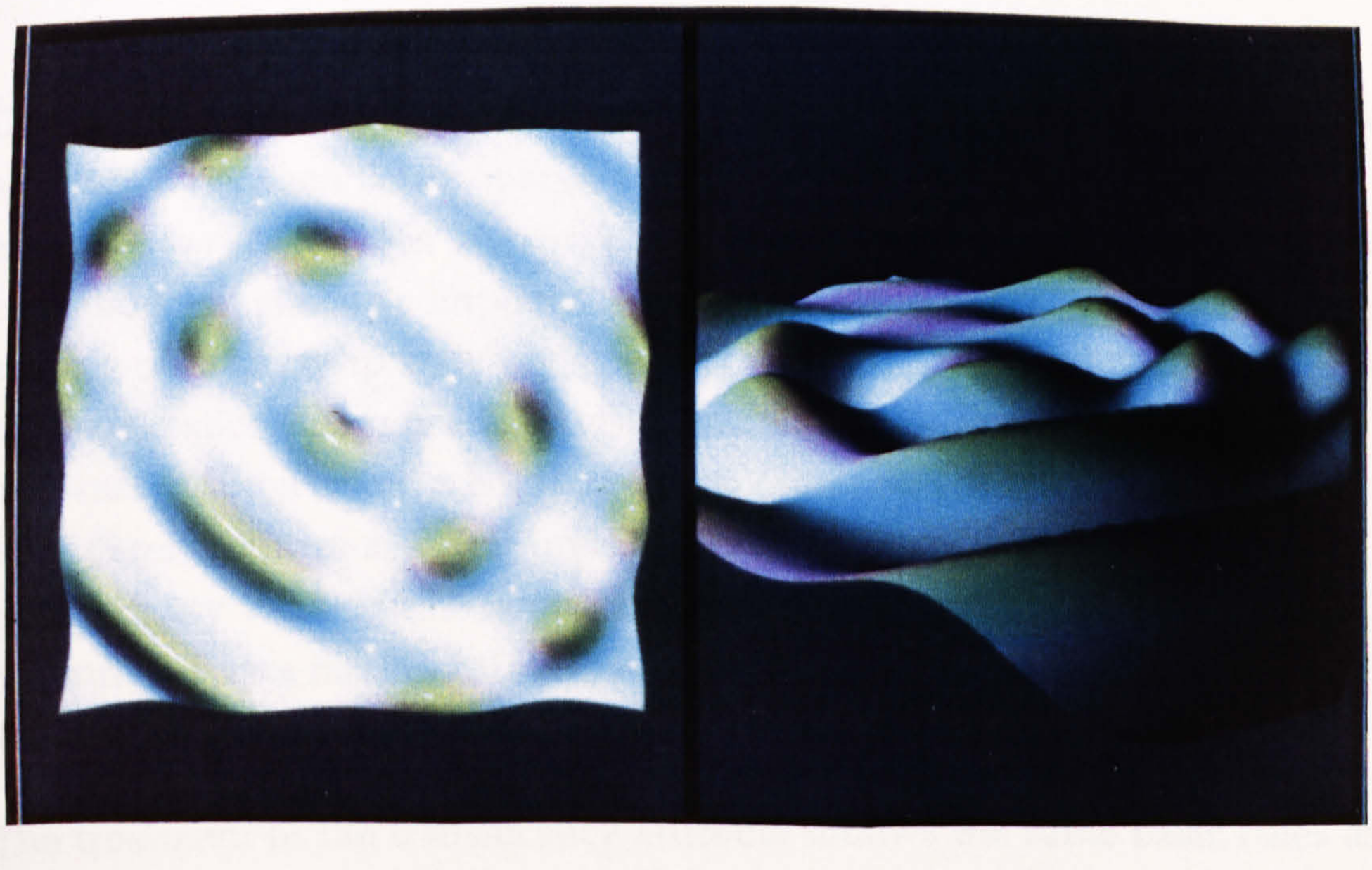


Fig 5.10 A red and a green sine component being added to a half space

The red sine component is located centrally, the green sine component is located to the upper right of the image on the left

5.3 Translucency

Several different words may be used to represent the surface property which allows varying amounts of light to pass through a surface. 'Transparency,' 'translucency' and 'opacity' are all possibilities. In this research, 'translucency' is used to represent a surface which is neither transparent or opaque.

> '**Translucency**: allowing light to pass through partially or diffusely; semitransparent.
> **Transparency**: the state of being transparent.
> **Transparent**: permitting the uninterrupted passage of light; clear; a *window* is transparent.
> **Opaque**: not transmitting light; not transparent or translucent.'
>
> [CED]

Depending upon the specification of the translucency attribute, an isosurface will range between being opaque and transparent. The translucency attribute varies between zero and one. A value of zero implies the surface is opaque, a value of one implies the surface is transparent. A transparent surface may be completely invisible in a rendering, depending upon the type of visualisation being used. A visualisation based upon a high quality software z-buffer using a boundary representation will accurately portray a transparent surface as being invisible. There are circumstances when a translucency value of one is useful. The colour of a translucent surface is determined by a mixture of the basic surface colour and what is behind it. The full range of translucency is valid.

The treatment of the translucency attribute follows the same basic rules as for the colour attribute. The same criteria apply, the value of the attribute must not become negative or exceed some maximum value, in this case one. The implementation of translucency is similar to that for colour, described in the previous section.

The specification of a translucency attribute may cause difficulties with some visualisation techniques.

It is generally difficult to portray translucency when using a visualisation based on a line drawing. Different types of dotted lines could be used to some effect. The visualisations based on line drawings in this research completely ignore the effects of the translucency attribute. This is in order to speed the processing of these renderings, as they are normally used to quickly displaying an isosurface, rather than as a final visualisation.

There is a general problem when dealing with translucency in conjunction with a boundary representation of an isosurface. Although the surface attributes can be found for any point in space, they are generally restricted to the isosurface boundary when dealing with a boundary representation. If a portion of the isosurface becomes transparent, the problem is in determining what is shown 'inside' the isosurface. The following example highlights this problem.

> Consider two spheres, one is white and the other is red. The white sphere is opaque, the red sphere is transparent. The two spheres are spaced slightly apart and are combined using addition to yield one isosurface. As the attributes mix, a pinkish surface is created in-between the two spheres. This pink surface is less translucent towards the white end eventually becoming opaque, and is more translucent towards the red end, becoming transparent. If the camera is placed such that it is looking into the white sphere from the side of the red sphere, what should be seen?

As this is a boundary representation, the camera will look straight through the region of interaction between the spheres to see the interior of the white sphere. The white sphere appears to be hollow, and the illusion usually portrayed by the boundary representation of enclosing a solid volume breaks down. This is demonstrated in figure 5.11.

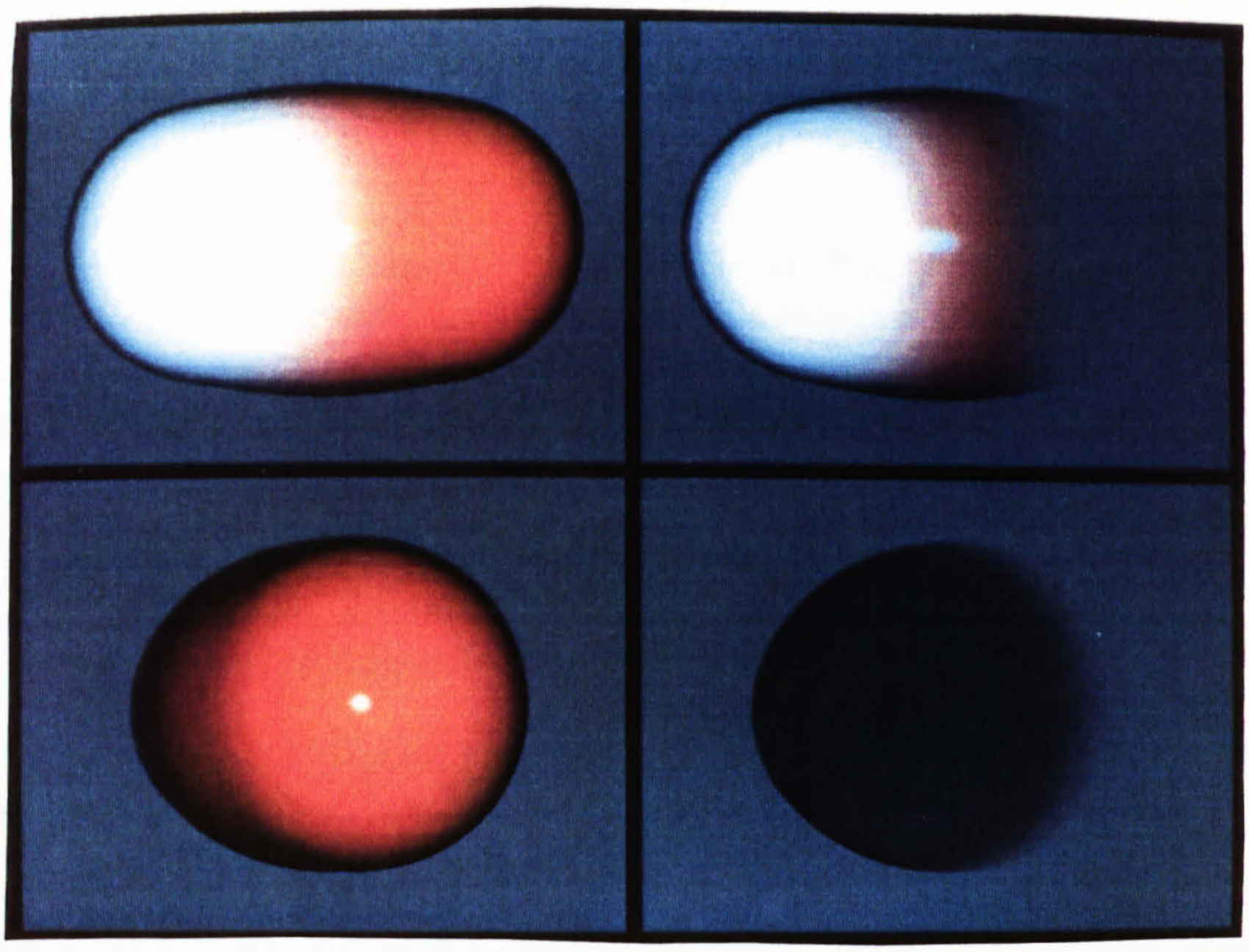


Fig 5.11 Demonstration of the problem of transparency and boundary representation

In order to correctly visualise the translucency attribute a different visualisation technique must be used. Translucency is handled elegantly when using volumetric rendering. This is in fact the basis of the success of the volumetric rendering approach. Also, ray-tracing could be implemented so as to correctly handle translucency.

Volumetric rendering and ray-tracing are more expensive in terms of computation than using a boundary representation with a traditional renderer, such as a z-buffer. Excellent results can be obtained using translucency with a z-buffer depending upon the circumstances. If translucency values are not too close to one, the technique gives excellent results.

Translucency has been implemented in this research using the full range of values between opaque and transparent. This is in spite of the limitations of any particular visualisation technique in handling this range, and the subsequent implications this has for the visualisation of the interior of an isosurface. As there may be wide application for the isosurface modelling technique, no assumptions about the best approach to solve this problem

```
The general definition of a surface attribute below may contain
    additional values.

type ATTRIBUTE is
    vector    colour      The colour at this point
    real      trans       The translucency value
    ...                   Any other attributes
    real      value       The scalar value
endtype

add attr1 and attr2, returning the result.
    attr1 is the attribute calculated previously,
    attr2 is the new attribute to be added

function addition (ATTRIBUTE attr1, attr2) returns (ATTRIBUTE)
begin
    ATTRIBUTE    result

    if (attr2.value ≤ 0) return (attr1)

    result.colour = attr1.colour + (attr2.colour * attr2.value)
    result.trans = attr1.trans + (attr2.trans * attr2.value)
    result.value = attr1.value + attr2.value

    return (result)
endfunction
```

Fig 5.12 Implementation of translucency for the addition operator

have been made. The implementation of the surface attributes is sufficiently general to support any of the visualisation techniques.

The implementation of translucency follows the same techniques as for colour. The implementation of translucency can be incorporated into that for colour by generalising the treatment of the attributes, as demonstrated in the re-implementation of the 'addition' operator shown in figure 5.12.

5.4 The calculation of a surface normal

One of the most important methods for improving the visual appearance of an isosurface is through a lighting calculation to produce a properly shaded

surface. In order to calculate the shading for a given point on a surface, a surface normal is required for the point. A surface normal is used in determining the direction a surface is facing. The lighting calculations used in this research are the same as those used in traditional modelling representations, the methods used to find a surface normal are not.

There are several methods of finding the surface normal at a point on an isosurface. The three methods considered in this research were:

1) Require that each scalar component calculate a gradient at the point required. Methods for combining these gradients can then be found which would allow SFDL to calculate the combined surface normal. The surface normal for a point is directly related to the gradient of the scalar field at the point. Given the gradient, a surface normal is easily found.

2) A surface normal can be determined directly from a boundary representation in a variety of ways. For instance, when using a polygonal mesh, surface normals can be found by first averaging polygon normals to find vertex normals and then interpolating to find the normal at any point on a polygon.

3) A gradient can be found for any point in a scalar field directly using numerical techniques.

The first method would involve changing each of the scalar components in SFDL so that they are able to supply a gradient. Some of the scalar components can easily supply gradients in an efficient manner. It is possible to easily calculate a surface normal for any point in a spheres radius of influence. For example, using a sphere component represented by the implicit formula:

$$2 - (x^2 + y^2 + z^2 - r^2) = 0$$

The surface normal with a length of one is calculated as:

$$\frac{\{x, y, z\}}{|\{x, y, z\}|}$$

The above expression can also be expressed as:

normalise (x, y, z)

The direction is normalised to have a length of one in order that it be handled by SFDL correctly. SFDL could use vector algebra to mix the normals in a similar fashion as for the other attributes. Before mixing, the normals would be changed so that their length corresponds to the scalar value of the component. A final normalisation of the calculated surface normal would be required before it is used in the lighting calculation.

The calculation of a surface normal for all of the components is not as easily accomplished as it was for the sphere. Although the gradient for a torus can be found directly, it is not as straightforward as for the sphere. Alternatively, a gradient for each of the components could be found numerically. Numerical techniques can be used to calculate gradients for components which can not be determined analytically. The final surface normal can be calculated from this gradient.

The requirement demanding the calculation of a normal for each of the scalar components complicates the implementation of the components. Also, the calculation of a normal in each component requires that one normalisation operation be performed for each scalar component evaluation. Each normalisation requires a square root operation.

The second solution proposed demands that surface normals at any point be found from information stored in the polygonal mesh. This can be accomplished in a variety of ways. The easiest technique is to have the polygonal normal represent the surface normal for all of the points on the polygon. This will give a crude shading effect. The polygons in the boundary representation would be visible as flat facets. A vertex normal can be found by averaging the polygonal normals of all of the polygons that share a vertex. The vertex normals can then be interpolated to find an approximated normal for any point on a polygon. This will produce a better result than using a single normal for the entire polygon. Surfaces displayed using this technique tend to exhibit small blemishes where the calculation of an interpolated surface normal deviates too widely from the actual surface normal at a point.

The third solution proposed is the one used in this research. It is easily implemented and gives a satisfactory result. The gradient of the scalar field is approximated using finite differences. This is accomplished by sampling

the scalar field six additional times and calculating the change in the values for each of three directions. The surface normal is then obtained from this gradient. The operation is given as:

$$\begin{aligned} \text{grad\_x} &= f(P + \Delta x) - f(P - \Delta x) \\ \text{grad\_y} &= f(P + \Delta y) - f(P - \Delta y) \\ \text{grad\_z} &= f(P + \Delta z) - f(P - \Delta z) \end{aligned}$$

$$\text{surface normal} = \text{normalise}(\{-\text{grad\_x}, -\text{grad\_y}, -\text{grad\_z}\})$$

The selection of appropriate values of Δx, Δy, and Δz determine the success of the technique. One selection of the Δ size is to have it match the increment size of the lattice when using the polygonal representation outlined in chapter four. This has the advantage of reducing SFDL evaluations, as the lattice is previously evaluated at each vertex in the search for the surface. However, this may result in a poor image as the increment is large in relation to the surface detail size. In this research, Δ is set to one tenth the lattice increment in each of the directions. The method of finding a surface normal used in this research is summarised in figure 5.13.

There is a problem with the numerical approach when dealing with isosurfaces which contain small details, such as those that can be generated using the minimum (intersection) scalar operator. If the size of Δ is too large this will lead to artefacts being visible on the surface.

In general, the proposed technique for numerically finding the surface normal works well, and is easily implemented. Bloomenthal uses a similar technique for calculating the surface normals [Bloomenthal 1987], although in his paper he restricts the calculation of a gradient to three additional scalar field calculations. Six additional samples are used in this research in order to gain a closer approximation to the actual gradient.

```
global  real      detail_size .          The minimum detail size

global  function  SFDL_evaluate ()       returns (real)
global  function  normalise()            returns (vector)

function normal (vector P) returns (vector)
begin
    vector    normal
    real      delta

    delta = detail_size / 10

    normal.x = SFDL_evaluate (P – {delta, 0, 0})
                    – SFDL_evaluate (P + {delta, 0, 0})

    normal.y = SFDL_evaluate (P – {0, delta, 0})
                    – SFDL_evaluate (P + {0, delta, 0})

    normal.z = SFDL_evaluate (P – {0, 0, delta})
                    – SFDL_evaluate (P + {0, 0, delta})

    return (normalise(gradient))
endfunction
```

Fig 5.13 Numerical method for calculating surface normals

5.5 Texture mapping

Texture mapping is widely used in computer graphics to efficiently add detail to a scene. This detail would otherwise require a prohibitive amount of effort to describe and time to create an image. There are two established techniques used for accomplishing texture mapping: two dimensional texture mapping and three dimensional texture mapping. Three dimensional texture mapping is also referred to as solid texturing.

A recent technique involving texture volumes is useful for displaying objects which include fur and fire for instance [Kajiya and Kay 1989; Perlin and Hoffert 1989]. The texture volume techniques will not be considered in this research as they are not *surface* properties, but rather *volume*

properties. The incorporation of texture volumes into the isosurface modelling technique is left as a topic for further research.

One of the difficult problems when applying texture mapping techniques to an isosurface is in finding a method of associating the points on an isosurface to the coordinate system of a texture map. As an isosurface proceeds through an animation, new surfaces may be created and shapes may change. Ideally the texture mapping technique will give the impression of a texture being attached to an isosurface. A texture map is perceived as being the surface, it is not just an attribute that is loosely attached. As the surface moves or changes, the texture should also change appropriately. An isosurface with a flag mapped onto it should retain a similar appearance as the isosurface moves or as small changes to the surface are made.

Traditionally in computer graphics the application of a texture to a surface poses no fundamental problem. This is due to the fact that a majority of modelling techniques are parametric in nature. As texture mapping is a parametric technique a mapping between the two parametric spaces is all that is required. Isosurface modelling is an implicit technique. There are two major aspects to the isosurface modelling technique which cause problems when texture mapping is applied.

The first problem is that the isosurfaces created using this modelling representation have shapes that are likely to change dramatically during an animation sequence. A single surface may split into several, later being recombined. The material with which the isosurface is perceived as being composed should remain consistent.

The second problem is that traditionally there is a direct one to one relationship between a graphical primitive and a surface which is displayed. Any translation needed in a coordinate system between these two representations can proceed naturally. When using a boundary representation of an isosurface there are intermediate stages between the graphical primitives and the final surface produced. The intermediate stages in this research are the polygonal representation and the SFDL representation. This makes the mapping from the starting representation (for example a sphere) to the final representation (a number of polygons) difficult, as the mapping must proceed through the intermediate stages.

A two dimensional texture mapping can in general be reduced to the problem of mapping an image onto a three dimensional surface. The image can be represented by a pair of parameters {u, v} each of which vary between zero and one. The problem of texture mapping a two dimensional image then becomes equivalent to finding the {u, v} parameters for any point on a surface:

$$\{u, v\} = \{f_u(x, y, z), f_v(x, y, z)\}$$

Recall that the isosurface modelling representation is an implicit technique. A parametric technique may be defined directly in terms of a {u, v} parameter. In the case of a parametric representation, a relation can easily be found between the modelling parameters and the texturing parameters.

An example which finds the {u, v} parameters for a sphere centred on the origin is shown. The solution maps an image onto the sphere twice, reflected around the x - y plane.

$$\text{let vect} = \text{normalise}(x, y, z)$$

$$u = \frac{(\{1, 0, 0\} \text{ dot vect}) + 1}{2}$$

$$v = \frac{(\{0, 1, 0\} \text{ dot vect}) + 1}{2}$$

An equivalent set of functions could be found for many of the scalar components. This would normally be sufficient, as a single primitive would map to a single surface when using traditional parametric modelling techniques. However, methods of combining the {u, v} parameters consistent with the techniques in SFDL need to be found.

Three dimensional texture mapping can be expressed similarly to two dimensional texture mapping, but is generally much different in implementation. Three dimensional texture maps require three parameters, each of which is not restricted to any particular range of values. Each of these values is usually directly or indirectly related to the {x, y, z} coordinates. Three dimensional texture mapping parameters can be expressed as:

$$\{u, v, w\} = \{f_u(x, y, z), f_v(x, y, z), f_w(x, y, z)\}$$

An example mapping from $\{x, y, z\}$ to $\{u, v, w\}$ is:

$$u = 2 * x$$
$$v = y$$
$$w = z$$

This has the effect of scaling the three dimensional texture map by one half in the x axis, squeezing the texture.

A second example is presented for discussion.

> Two spheres are spaced slightly apart yielding one surface.
> How are textures applied to these spheres?

There are three possible methods of applying texture to this example:

1) Each sphere can have individual instances of the same texture.
2) Both spheres can have completely different textures.
3) A texture can be applied to the resulting surface rather than an individual component.

The first two cases can be handled in a similar fashion. At a point in the region where the two spheres interact, two sets of $\{u, v, w\}$ parameters are needed, one for each sphere, as well as a parameter indicating the strength of each texture. Each sphere in isolation would have a strength parameter which creates only one texture map, as they come together a blending occurs.

The last case requires that the method of finding a $\{u, v, w\}$ parameter be changed to operate on the surface produced by the two spheres. It requires that methods of calculating the parameters can be assigned for each scalar operator as well as each scalar component in SFDL. A method for setting the initial parameters could either derive information from the SFDL program directly, or be set by the user. The solution of this case requires that instead of a parameter being calculated for a point which is used in all subsequent texturing processes, a hierarchy of texturing operations and associated methods of combining them must be used.

The presence of increasing numbers of components and operators in a texture description demands that an increasing amount of information be generated at each point to describe the texture produced. Consider for example three spheres. One sphere could have its own texture, the other two a texture applied to their combination, and the texture of these two groups mixed for the final result. This process can not be represented using a single parameter for each point, or even one parameter for each component at a point. More complicated examples are possible.

Texture mapping is a technique that can be applied to colour, the surface normals, translucency and any other available surface attributes. An example implementation of texture mapping in SFDL is shown in figure 5.14. This implementation does not allow bump mapping and is demonstrated only for the sphere component and addition operator. To incorporate bump mapping, an alternative technique for finding the surface normal, as discussed in section 5.4, must be used. In this research, numerical rather than procedural methods are used in the determination of a surface normal. In order to allow texture functions to modify surface normals, the procedural method for finding surface normals must be used.

The implementation of texture mapping in figure 5.14 allows the production of textures that move with each isosurface and mix gracefully as differing textures come into contact.

Previous research on texture mapping isosurface models has been presented in a paper by Wyvill, McPheeters and Wyvill in 1987. In that research an abstract texture space is constructed and labelled $\{F, H, C\}$. The parameters are calculated from the scalar components using a weighted average. Examples are presented in the paper of two smoothly merging spheres with the same texture being applied. As only a single parameter is used in the texture calculation, complex interactions involving many components and operators are not possible. It is also not possible to demonstrate the interaction of two spheres with different textures interacting using the techniques presented in the paper. The technique presented in this research handles these situations.

```
                                        The parameter attr is called by reference
function sphere (SPHERE s; vector P; real value; ATTRIBUTE attr)
        returns (real)
begin
    ...
    As in chapter three.
    ...
    value = that calculated above

    attr.colour = s.colour
    attr.trans = s.trans
    attr.value = value

                        Again attr is called by reference, and may be changed
    if (s.texture exists) call (s.texture (P, attr))

    return (value)
endfunction

global    vector    point          Point for SFDL evaluation

function addition (ATTRIBUTE attr1, attr2; procedure texture)
        returns (ATTRIBUTE)
begin
    ATTRIBUTE    result
    ...
    as in section 5.3
    ...
                                        Result is called by reference
    if (texture exists at this operation) call texture (point, result)

    return (result)
endfunction
```

Fig 5.14 Example implementations of texture

5.6 Boundary representation related problems

An advantage to using a boundary representation for an isosurface is its efficiency in a number of respects. A boundary representation can be quickly displayed using a number of techniques in computer graphics. A single boundary representation can also be displayed from several viewpoints,

eliminating the requirement of recomputing the isosurface. The calculation of a boundary representation requires fewer SFDL evaluations than would be needed, for instance, with a volumetric rendering approach. Only the information at the vertices of the representation are calculated. The information at the vertices is interpolated to obtain interior values.

A boundary representation is sufficient as long as it does not introduce unwanted aliasing artefacts into an image. One of the more obvious artefacts, discussed in chapter four, is the presence of visible facets in the silhouette edge of a boundary representation. This can be avoided by generating additional polygons in the visualisation process.

Aliasing artefacts may be apparent in the representation of the surface attributes of an isosurface. If the colour of an isosurface changes at a frequency greater than that at which the surface is sampled then there will be aliasing artefacts introduced into the colour of the isosurface. Presently, most of the attribute detail is specified at the same frequency as is the surface topological detail. The specification of the surface attributes is done at the same time as surface shape. With the exception of texture mapping, which can introduce many small details, if the isosurface shape does not contain any topological aliasing artefacts then the surface attributes will not.

Recall that the maximum (union) operator could have been implemented in two ways. The manner which mixed the attributes was selected instead of choosing one or the other of the attributes. This in effect changes the join between the attributes from a line to a region. A smoothly changing region is easier to represent; it can be approximated by sampling, and reconstructed using interpolation.

The texture attribute is a method of introducing fine detail into an image. Texture maps may contain many small details which are not accurately approximated by the process of first sampling and then later using interpolation to find a value. Unlike the other attributes, texture mapping information can not be stored at the vertices of a boundary representation. The SFDL evaluation must be made for every point on an isosurface to accurately represent texture.

There is an analogy between this situation and different surface shading techniques. Consider flat shading, Gouraud shading and Phong shading. Flat shading is similar to assigning an attribute to the entire isosurface.

There is no change of that attribute on the surface. Gouraud shading is similar to calculating the attributes at the vertices of a boundary representation. Interpolation is used to find interior points. Phong shading is similar to what is required to accurately represent texture. A SFDL evaluation must be done for every point on the isosurface to calculate texture.

If the complete texture mapping model presented in the previous section is used then there is no easy way of storing information at each vertex to accommodate texture mapping after the surface has been found. The texture mapping model can be simplified in order to easily accommodate the boundary representation at the expense of the generality of texture treatment. This constraint influenced the implementation of texture by Wyvill, McPheeters and Wyvill in 1987, in which a single parameter is used. This single parameter is easily stored in a polygonal mesh representation and was easily handled by the z-buffer used for visualisation.

There is also the problem of accurately visualising translucency when using a boundary representation. This was discussed in section 5.3.

Regardless of the problems associated with the boundary representation, it is a useful technique in the study and representation of isosurfaces. As this body of research is not restricted to a single method of visualisation, an appropriate representation and visualisation technique can be chosen depending upon the circumstances of a particular situation.

5.7 Conclusions

For isosurfaces defined in a scalar field to be a successful modelling technique in computer graphics, sufficient control over the appearance of the isosurface must be present. The incorporation of colour, transparency and texturing information into the isosurface modelling technique has been discussed in this chapter. Several possible methods for finding a surface normal were also proposed. With the exception of texture mapping, each of the surface attributes can be elegantly and efficiently implemented. Methods of implementing texture mapping were proposed, however they are more complex and expensive than the implementation of the other attributes.

Rather than mimicking the functionality of the implementation of surface attributes in the traditional modelling techniques, the treatment of surface attributes in isosurface modelling offers several extensions. Mixing of the attributes between separate scalar components is handled elegantly using the techniques proposed. Using this modelling technique it is possible to easily create shapes that would be extremely difficult to create traditionally.

The implementation of surface attributes is accomplished by the extension of SFDL and the scalar components. These extensions are straightforward and pose no problem to the incorporation of surface attributes into the isosurface modelling technique.

Chapter 6

Isosurface animation

6.1 Introduction

In chapters three, four and five techniques have been discussed for the specification and visualisation of a wide variety of objects using isosurface modelling techniques. Animation sequences can be created using these modelling techniques when they are incorporated into an appropriate animation system. Several extensions to the isosurface modelling techniques are made inorder to support various animation techniques. These extensions are discussed.

Techniques for producing isosurface animation are discussed and demonstrated in this chapter. The effects which can be accomplished with isosurface animation would be extremely difficult to replicate using traditional computer animation and modelling techniques.

Three types of isosurface animation are discussed: geometric animation, component parameter animation and metamorphosis. Geometric animation deals with techniques for changing the geometric transformations upon which each of the scalar components are based. As the scalar components move, rotate and change in scale, the isosurfaces merge and split apart elegantly. Component parameter animation involves

the modification of the parameters which describe a scalar component, such as colour, during an animation sequence. Two parameters have been added to the scalar components in order to enhance the realism of the interaction of components during an animation sequence, *velocity* and *bias*. The effect, implications and implementation of these parameters will be discussed. The last animation technique discussed in this chapter is metamorphosis, a technique which is difficult to implement using traditional modelling techniques in computer graphics. Metamorphosis can be easily and efficiently implemented using isosurface animation techniques.

6.2 Representation of animation

Many techniques have been developed in computer graphics for specifying sequences of computer animation. A partial list of the different types of animation include: character animation; dynamics; kinematics; reverse kinematics; scripting; procedural; parametric; and behavioural. It is desirable that many forms of animation are able to be used in conjunction with the isosurface modelling representation. This will allow the description of a diverse range of animation sequences.

The approach used in this research for creating an animation sequence is to separate the specification of the animation sequence from the specification of the isosurface. SFDL has not been extended to incorporate the specification of animation sequences. An animation sequence is specified within an animation system. This animation system will then create the correct SFDL programs for each instance of time required.

The animation system used in this research is called JED, which will be described in chapter seven. JED is a parametric animation system, all of the parameters in a graphical model are able to be changed during an animation sequence. Examples of parameters in JED are: geometric transformations; attributes; scalar operator parameters; and the scalar component parameters such as *influence*. *Influence* and other component parameters were discussed in chapter three.

Separating the specification of an animation sequence from the specification of the isosurface allows the implementation of many varied animation

techniques without altering the basic structure of SFDL. The animation system can be extended to allow the incorporation of many different types of animation, each of which is subsequently automatically translated into a series of SFDL programs as they are required.

6.3 Geometric transformation

Geometric transformations such as scale, rotation and translation can be applied to each object, or collection of objects, in a model. These transformations can be changed over time in an animation system to generate an animation sequence. Animation sequences based on changes in geometric information are one of the more common forms of animation in the computer graphics industry.

Animation sequences based upon changes in the geometry of an isosurface model have a major advantage over their traditionally modelled counterparts. As the objects are transformed in an animation the surfaces will deform, merge, disappear and reappear according to the rules governing isosurfaces and scalar fields. These changes in surface topology are extremely difficult to achieve when using traditional techniques. The ability to easily create changes in surface topology is one of the main strengths of the isosurface modelling representation.

Care must be exercised when creating a visualisation of the isosurfaces in an animation sequence. Recall that the visualisation process depends upon two important parameters: the volume of space in which to search for an isosurface; and the minimum size of an isosurface detail. These parameters may need to be included in the specification of an animation sequence, changing as the animation progresses. For instance if the scalar components are moving through space while the search volume remains static, the isosurfaces may overlap, or leave the search volume producing partial results. Alternatively if an isosurface reduces in size while the minimum detail size remains fixed, the isosurface may disappear, slipping through the grid used to search the volume of space.

Examples of animation sequences follow. The animation sequences are presented as a series of images which should be read in 'reading order,' from the left to the right and from the top to the bottom.

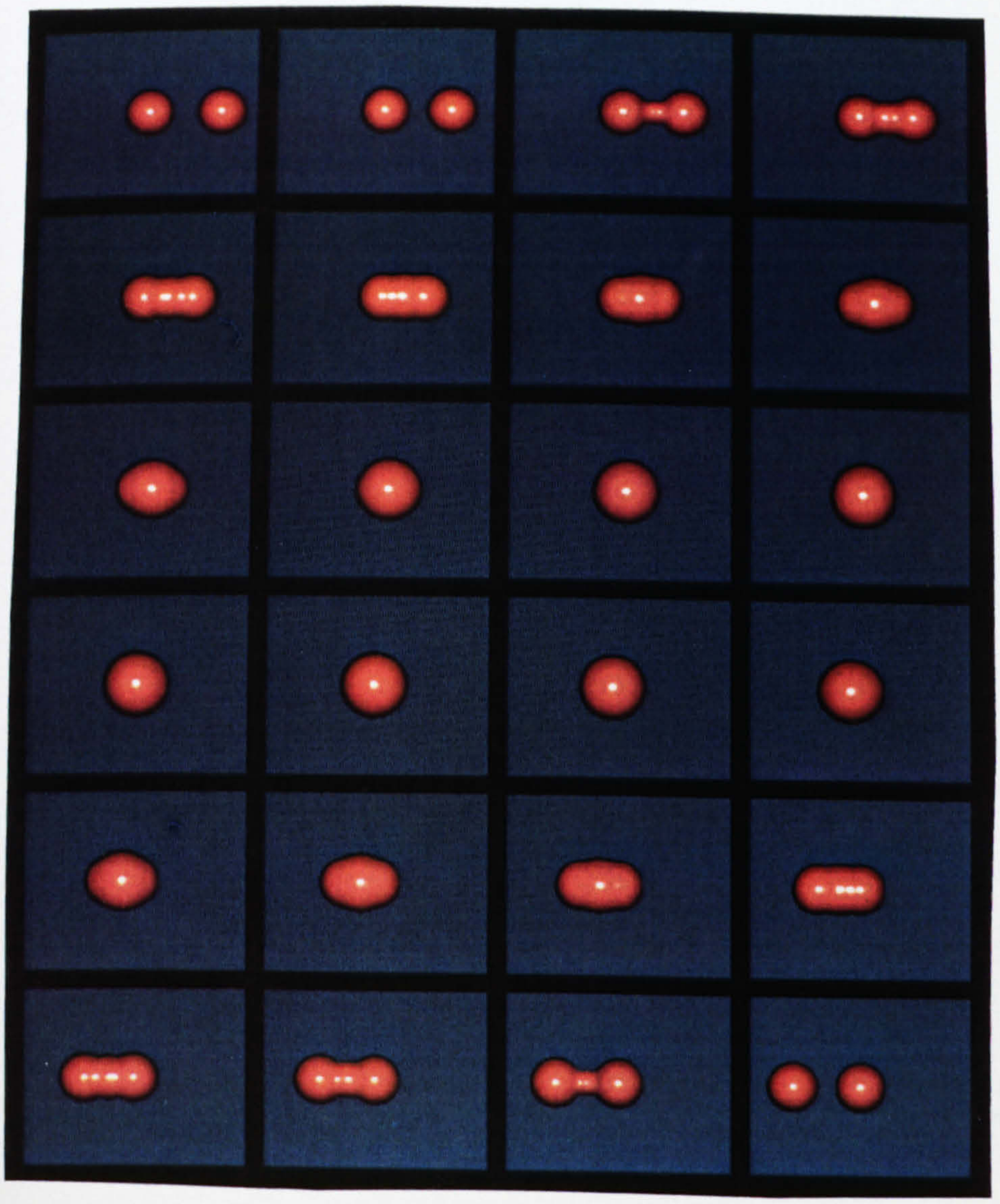


Fig 6.1 Merging and pulling apart of two spheres.

The spheres are combined using the addition operator.

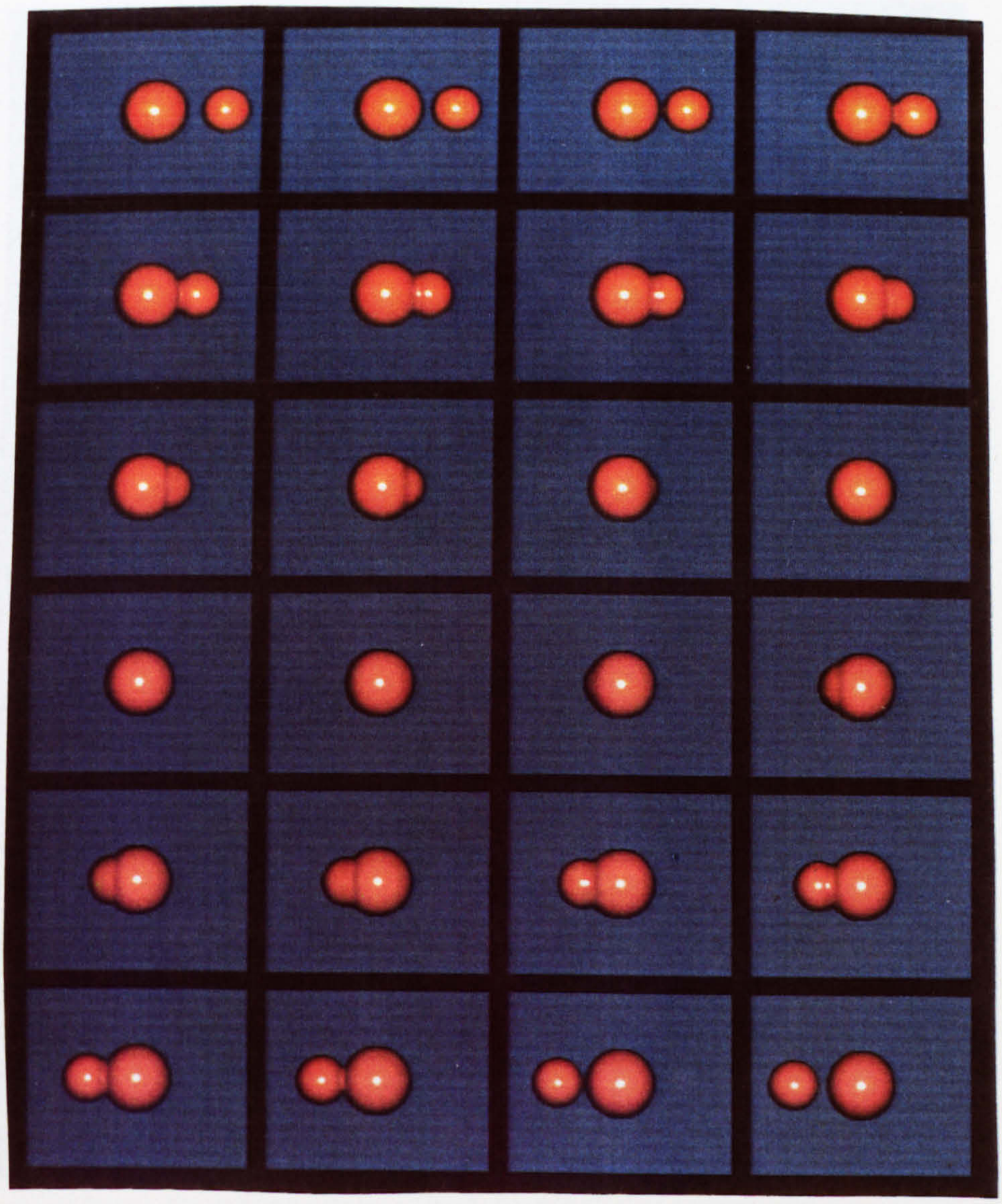


Fig 6.2 Merging and pulling apart of the union of two spheres

The two spheres are combined using the union (maximum) operator. Notice that the spheres do not blend as they do in figure 6.1.

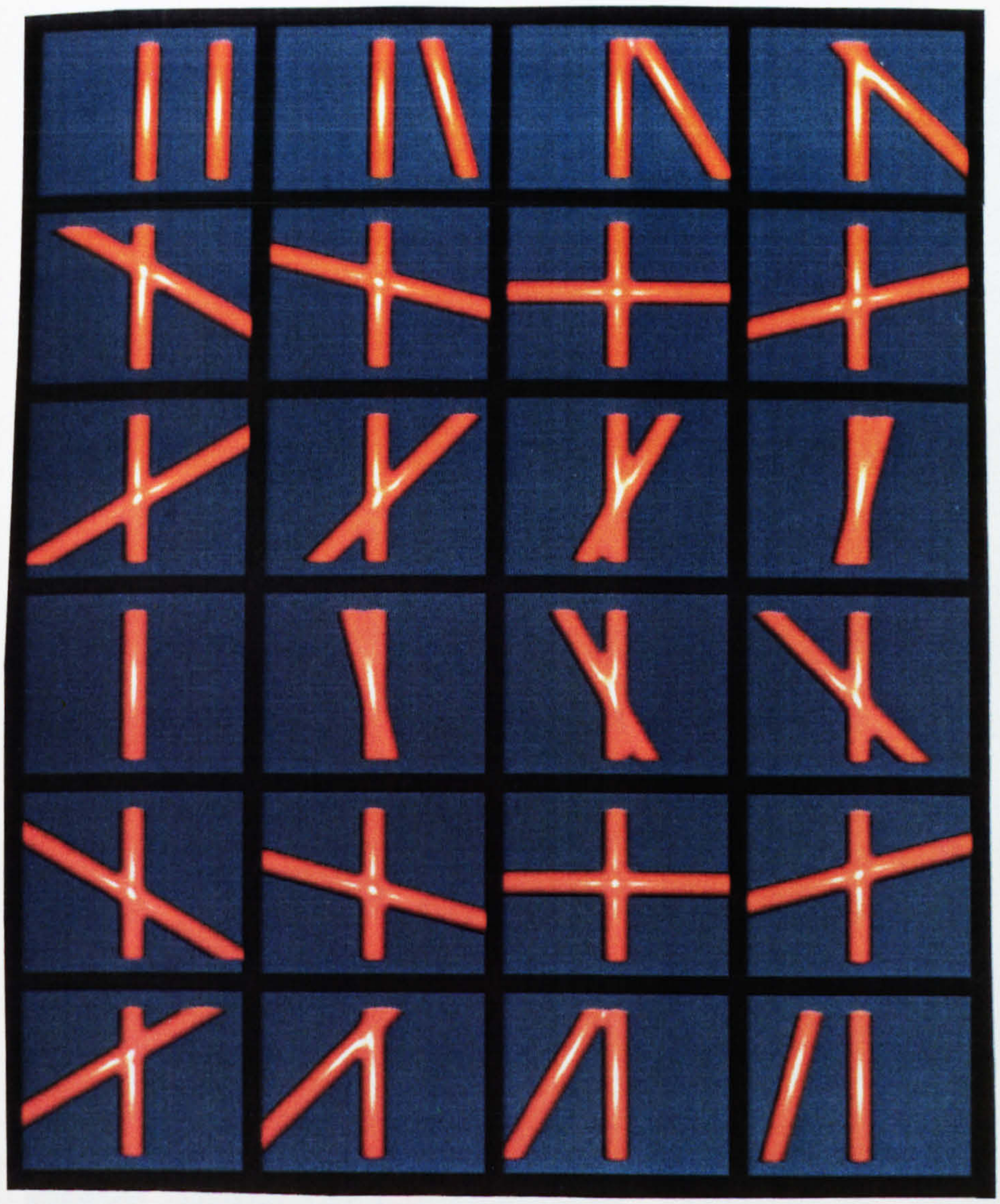


Fig 6.3 Merging cylinders

One cylinder remains stationary as the second rotates and moves across it. Notice the smooth joins as the two cylinders interact.

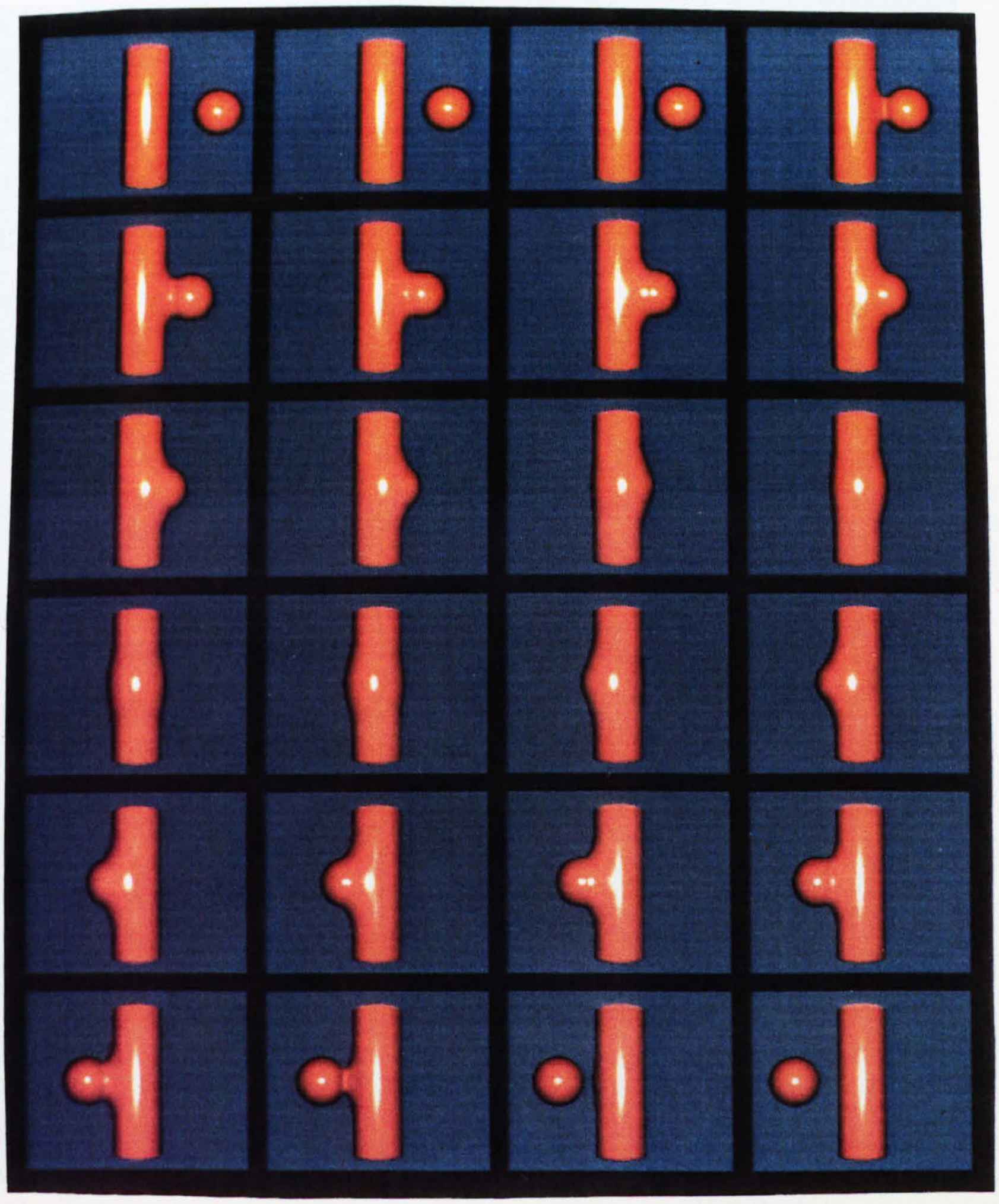


Fig 6.4 Sphere passing through a cylinder

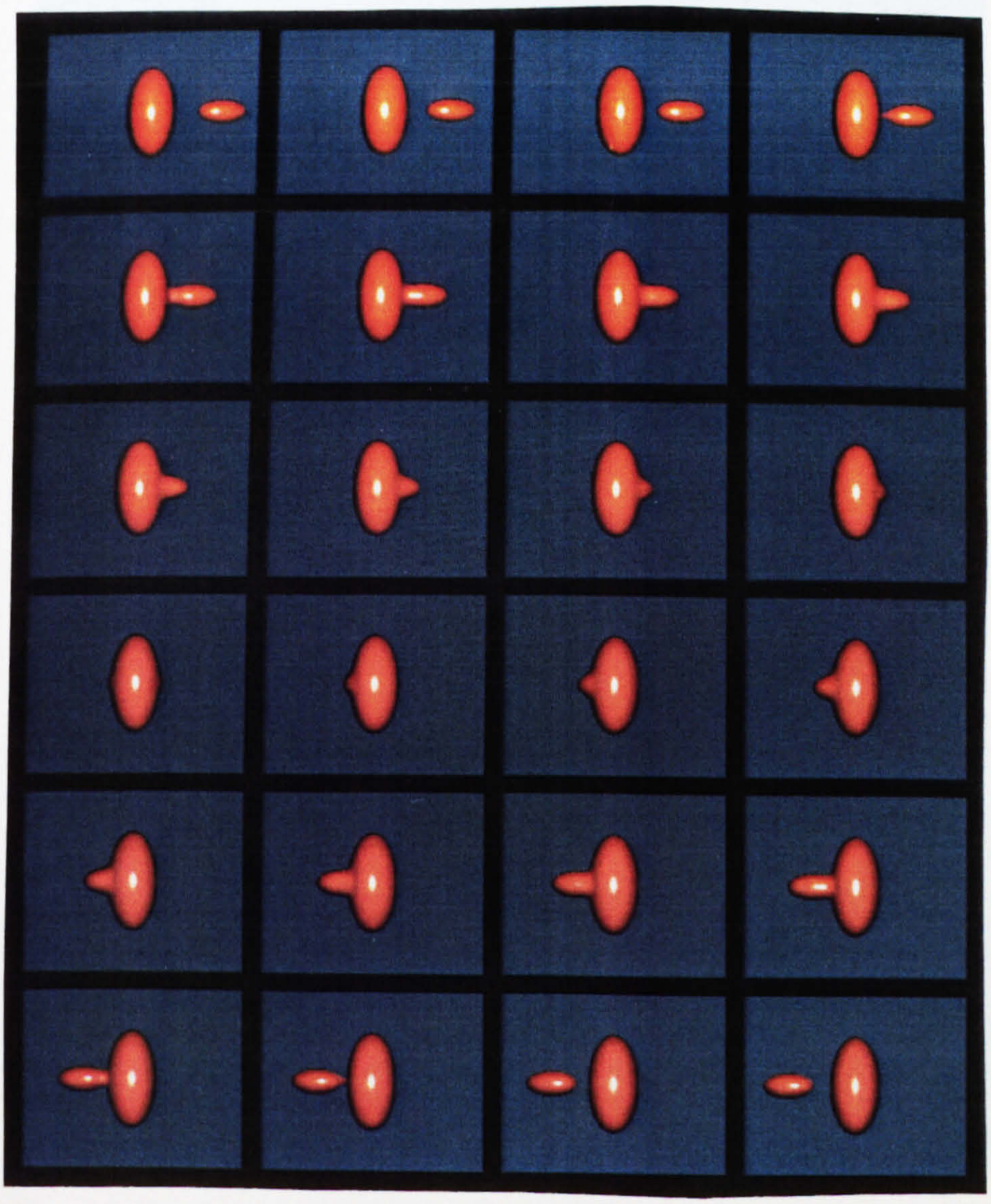


Fig 6.5 Small ellipsoid passing through a larger ellipsoid

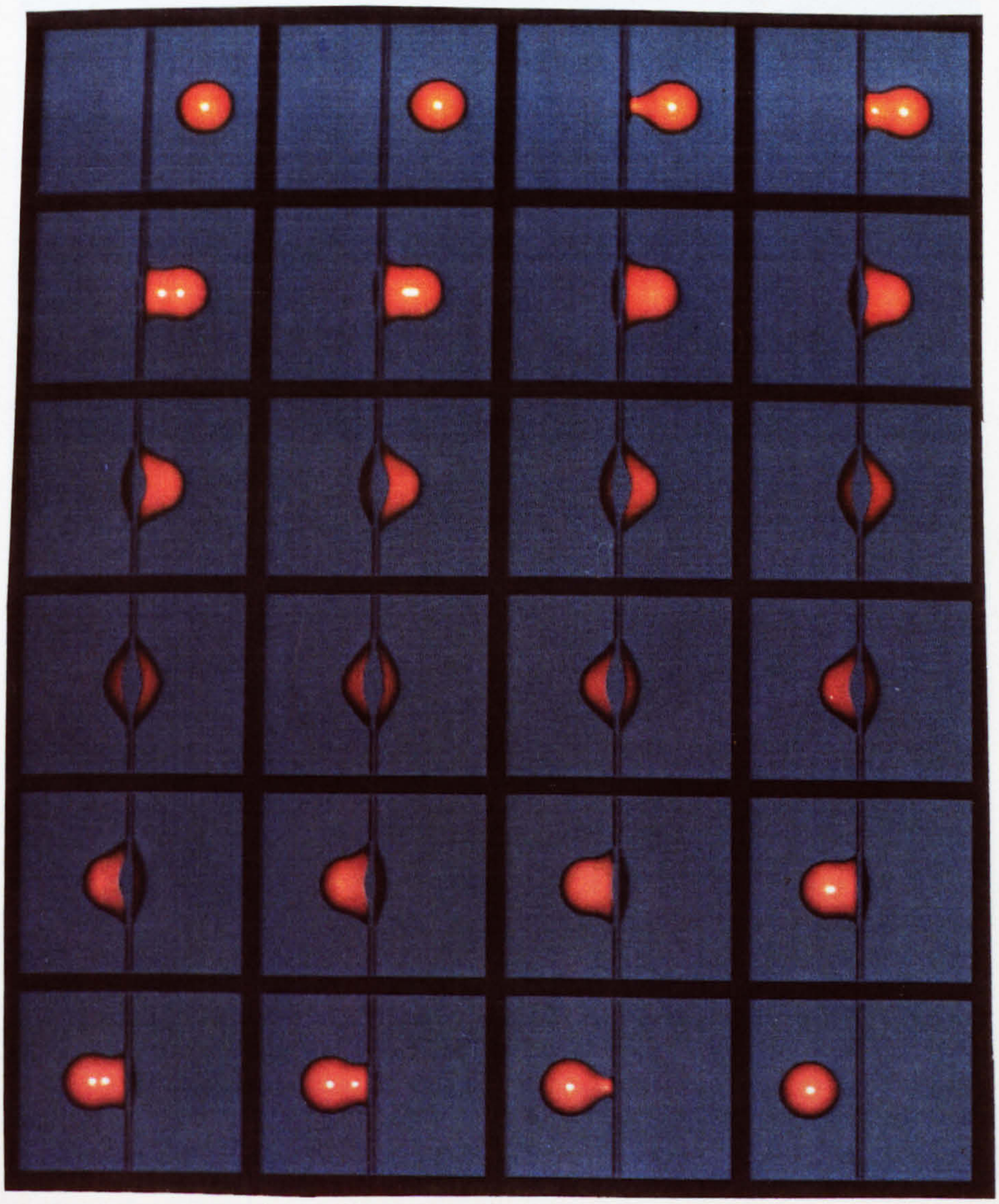


Fig 6.6 Sphere passing through a plane

The plane is being observed edge on, and is thus only visible as two vertical lines. Each line represents the plane as it would be seen from the appropriate side.

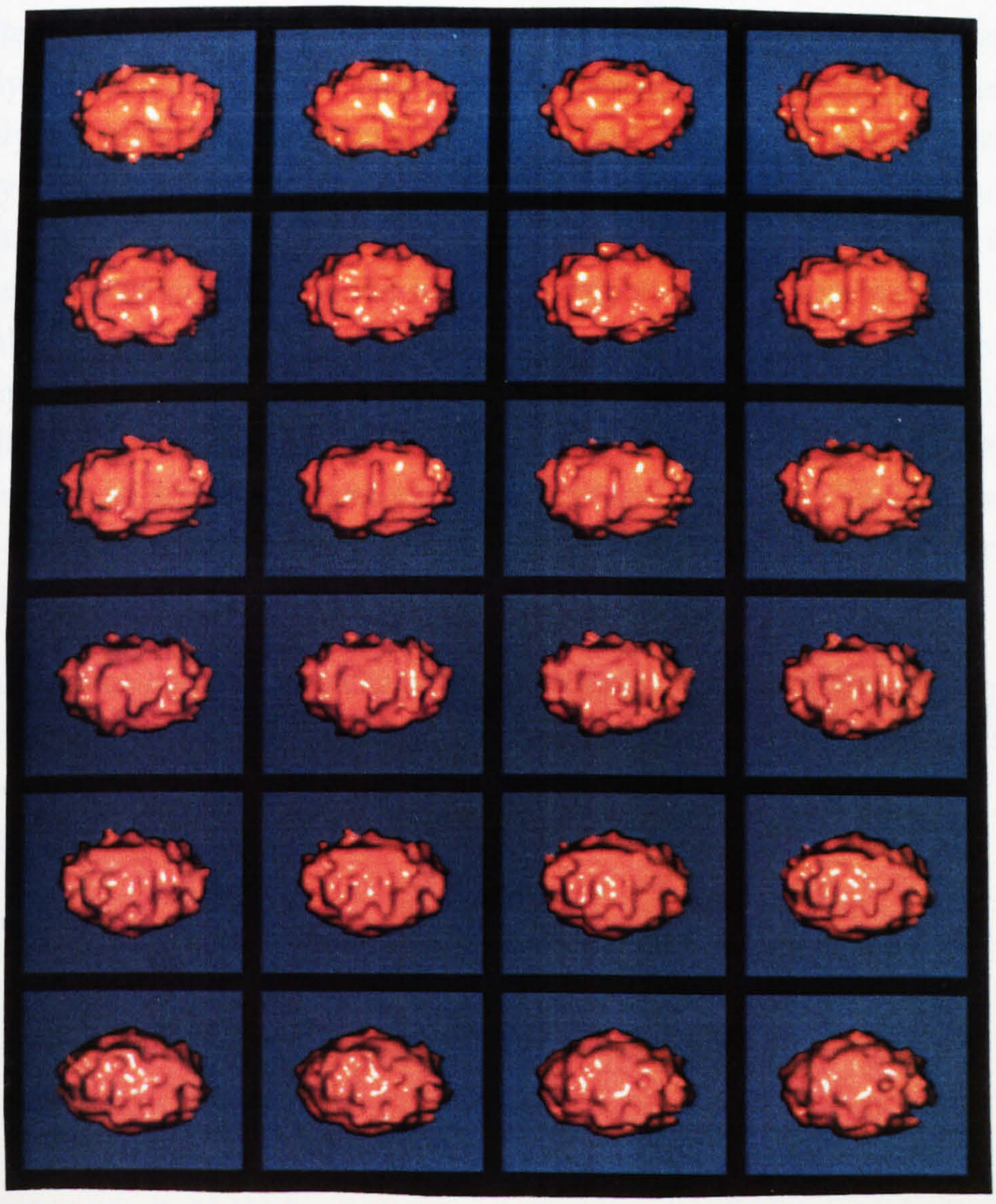


Fig 6.7 Noise being moved through an ellipsoid

The noise field is moving from the back to the front of the ellipsoid. Notice that the frequency of the noise is visible. This effect can be reduced through adding additional noise values at different frequencies.

Notice that in the previous animation sequences there are no characteristics which determine absolutely that the sequences were produced using isosurface modelling techniques. The same sequences could have been produced using a variety of traditional techniques with an extreme amount of patience. However, each of the previous animations were easily and quickly described using the isosurface modelling representation, needing roughly ten minutes per animation sequence to specify, and less than twenty minutes to generate. Producing equivalent images using traditional techniques would require several orders of magnitude more effort. For instance the specification of the images polygon-by-polygon may take several days. Any required changes in the animation sequences are easily accomplished when using the isosurface representation. Traditionally modelled imitations may need to be completely re-specified to accommodate a change in an animation sequence. In practical terms the images shown are never likely to be produced using any technique other than isosurface modelling.

The isosurface animation techniques are ideally suited for the description of animation sequences of the types demonstrated. Traditional modelling techniques are wholly inadequate to efficiently represent the same animations.

6.4 Scalar component parameter animation

Recall that an SFDL program consists of two integral parts: the scalar components and the scalar operators. A type of animation involving changing a scalar operator parameter over time is discussed in section 6.5, metamorphosis.

A scalar component may contain several parameters as well as the geometric information specified in a transformation matrix. Geometric animation involves changing the transformation matrix at each instance in an animation sequence. Geometric animation is discussed in section 6.3. Component parameter animation involves changing the parameters in a scalar component as part of an animation sequence. An example of a scalar component parameter is colour.

The distinction between component parameter animation and geometric animation is made purely for purposes of discussion. Both forms of animation are handled similarly; an animation sequence is specified in an animation system which automatically generates the required SFDL programs.

Three forms of component parameter animation will be discussed: surface attributes; *velocity*; and *bias*. The latter two types of component parameter animation involve changes being made to SFDL.

6.4.1 Surface attributes

There are three main surface attributes implemented in this research: colour; translucency; and texture mapping. This list could be extended to suit the needs of a particular application. For example the addition of reflectance, refraction and a specular coefficient may be useful to support a ray-tracing visualisation.

Each of the parameters in a scalar component may be restricted to a particular range of values. For example, translucency must be between zero and one, inclusive. Values outside of this range will produce undefined results. If the value of each of the parameters in a scalar component is valid then a continuous animation will result. The animation of surface attribute parameters is implemented in a straight forward manner. An animation sequence is specified in an animation system and the appropriate SFDL programs are produced.

Additional animation sequences which involve changes in the colour of the resulting isosurfaces as well as component parameter animation are shown. A demonstration of texture mapping is shown in figure 6.13.

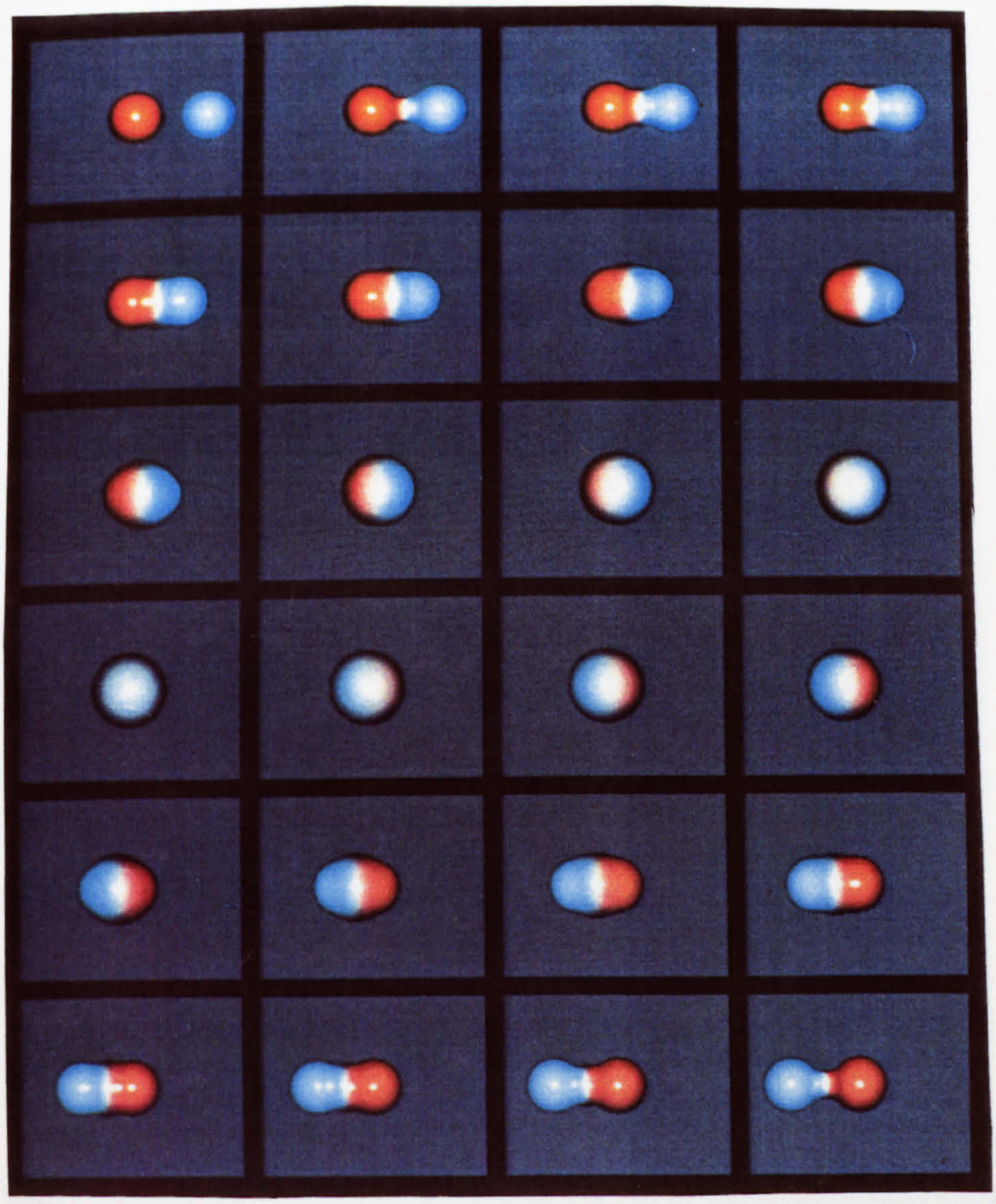


Fig 6.8 Cyan sphere passing through a stationary red sphere

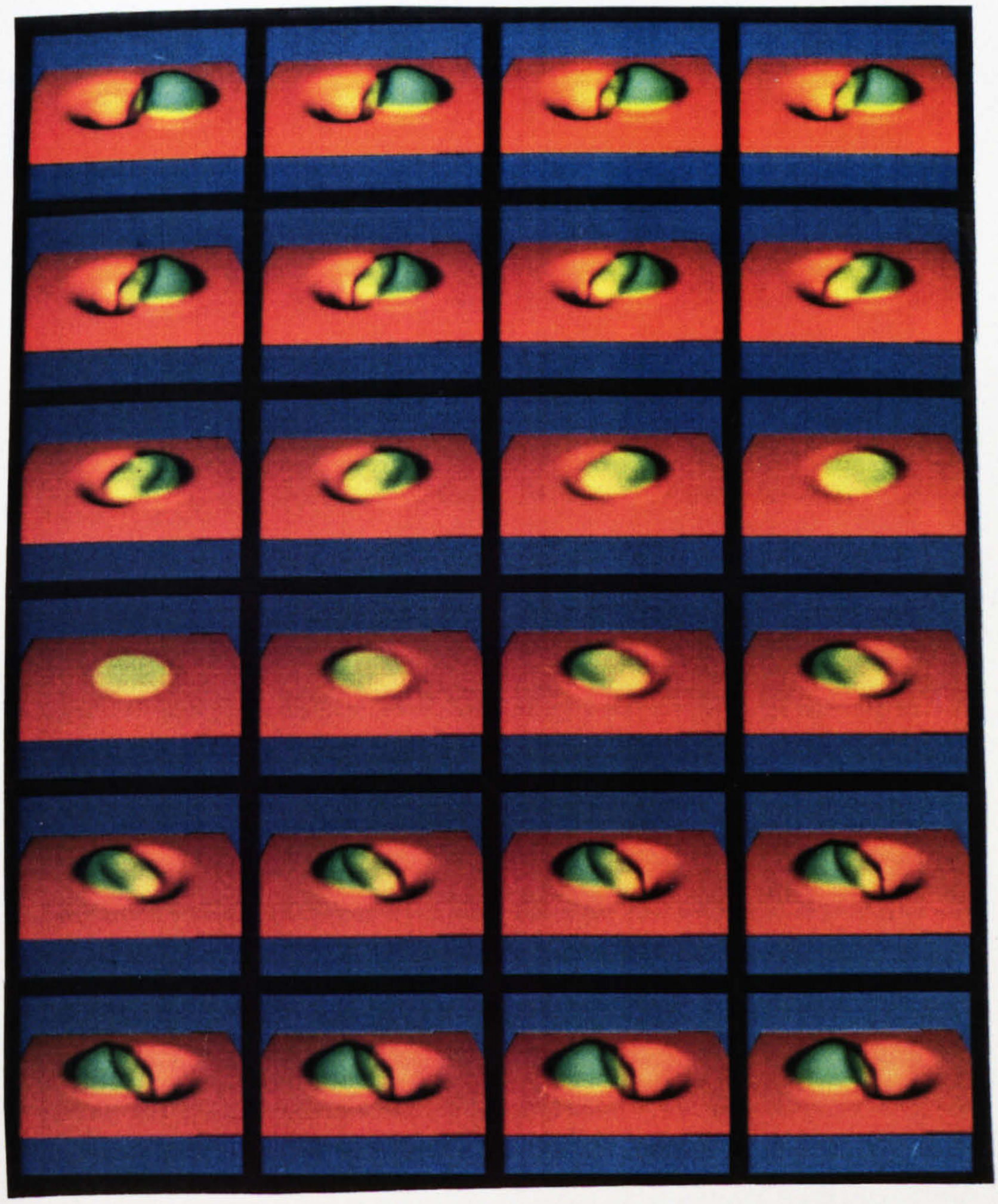


Fig 6.9 Positive negative interaction

A green sphere exchanges places with a sphere being subtracted from the scalar field. As the subtraction operator ignores colour, the hole produced takes on the colour of the half space both components are interacting with.

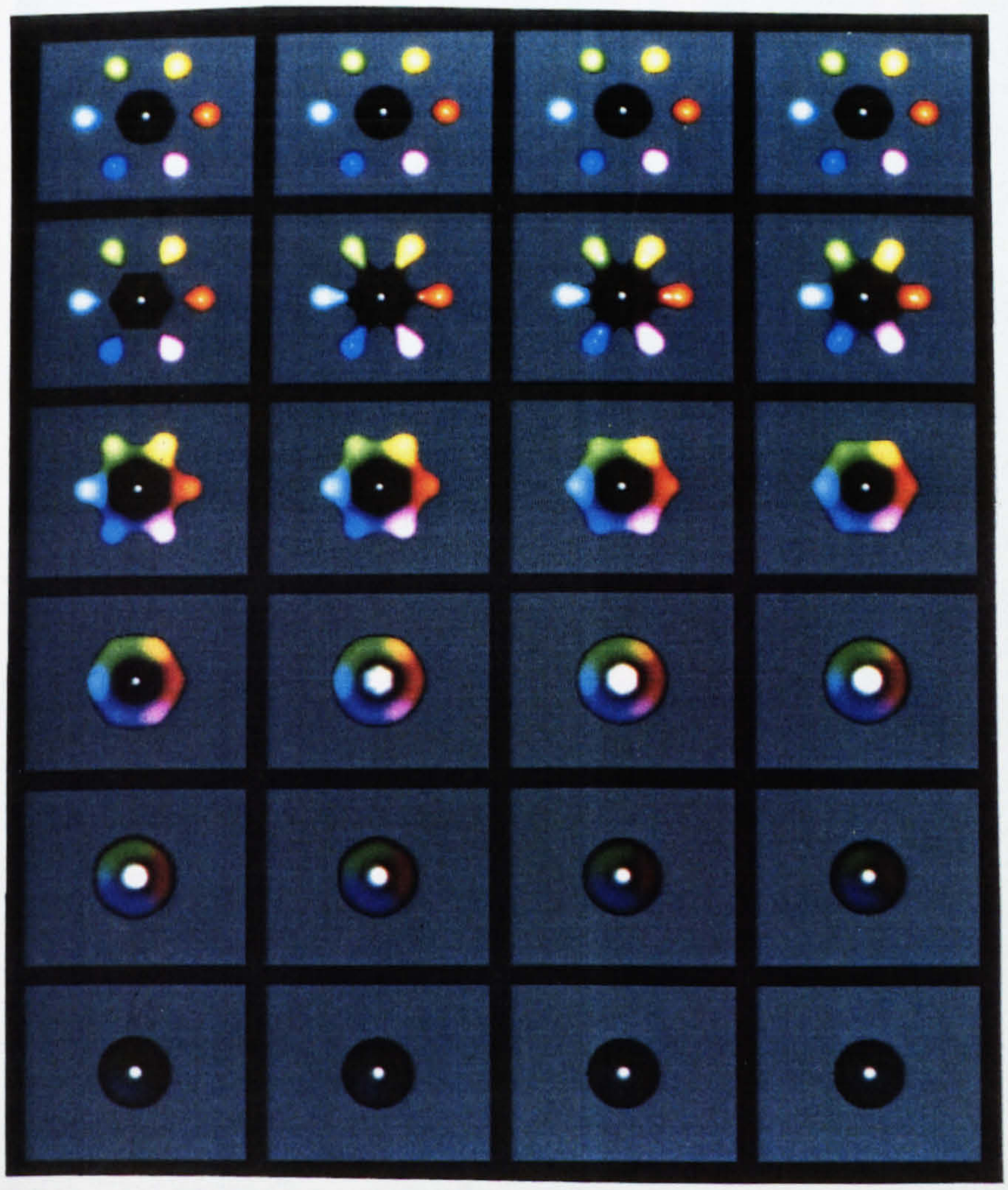


Fig 6.10 Six colourful spheres merging into a black sphere

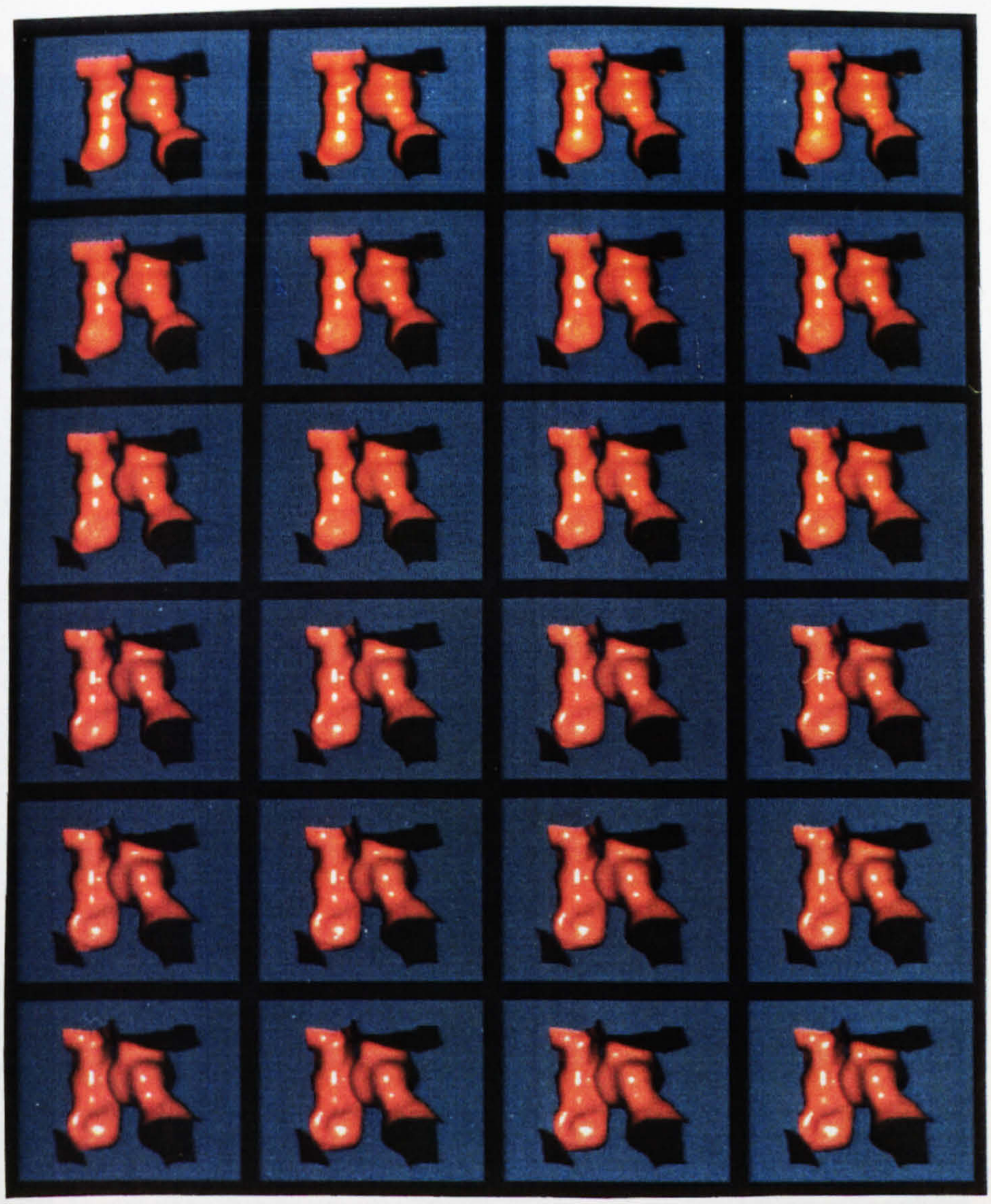


Fig 6.11 Increasing noise amongst spheres

Notice that the search volume is specified so that the spheres on the edge are cut off. The maximum and minimum noise values are increasing in magnitude as the animation progresses. This is an example of component parameter animation.

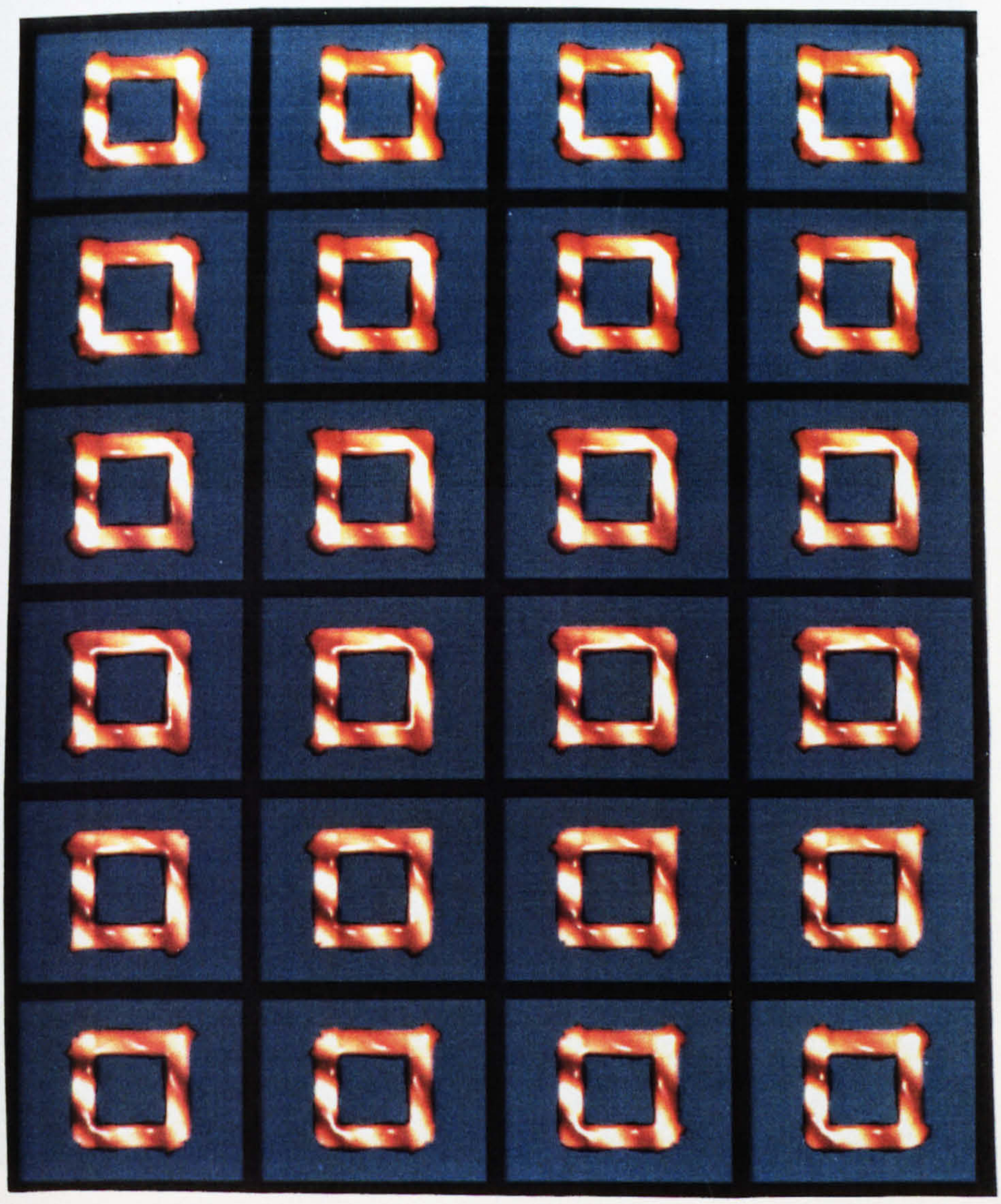


Fig 6.12 Sine wave passing across a frame

The frame is constructed from the union of four scaled cubes. The phase parameter of the sine wave component changes over time.

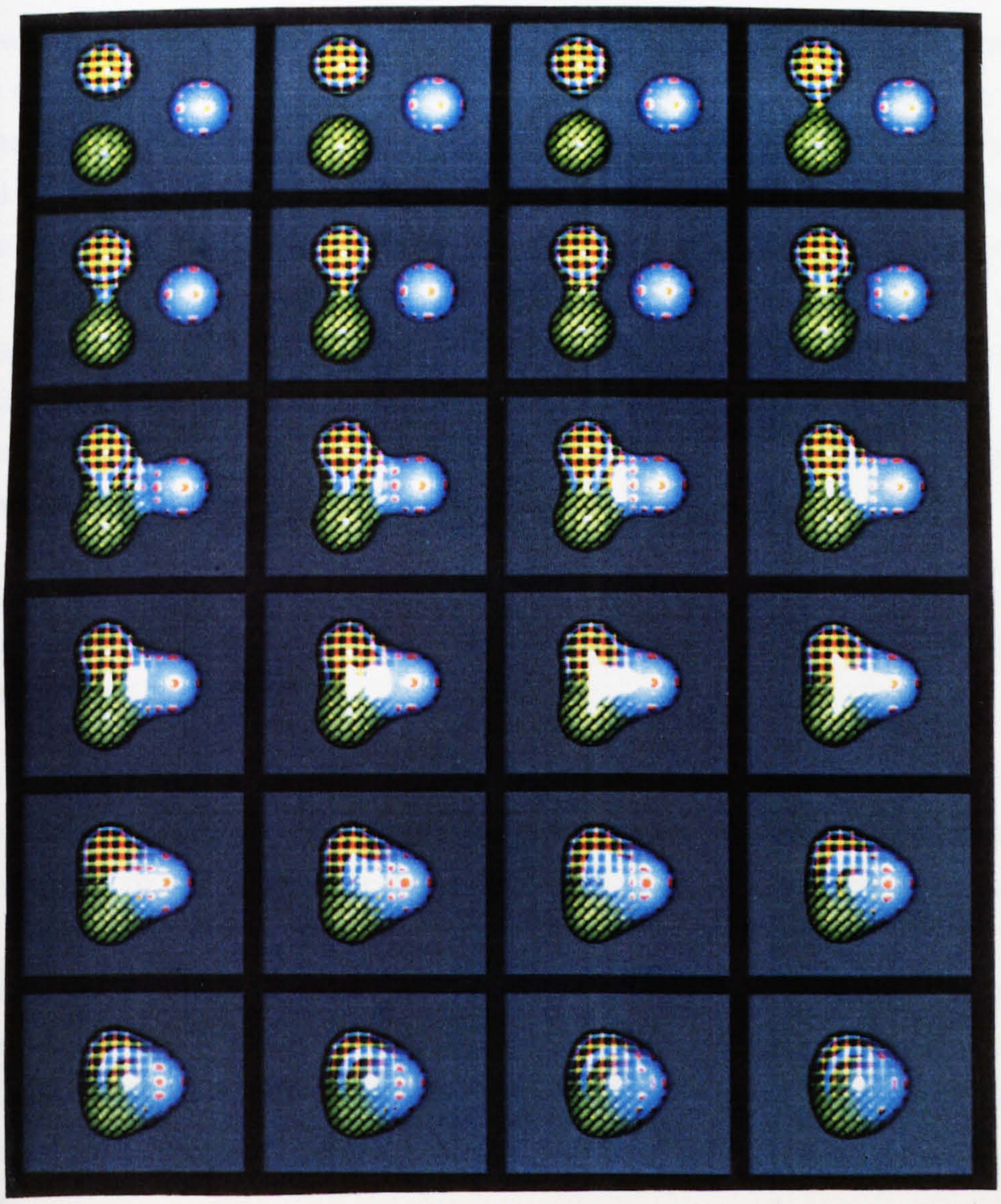


Fig 6.13 Three textured spheres merging and interacting

This is a demonstration of the texture mapping technique proposed in chapter four.

6.4.2 Velocity

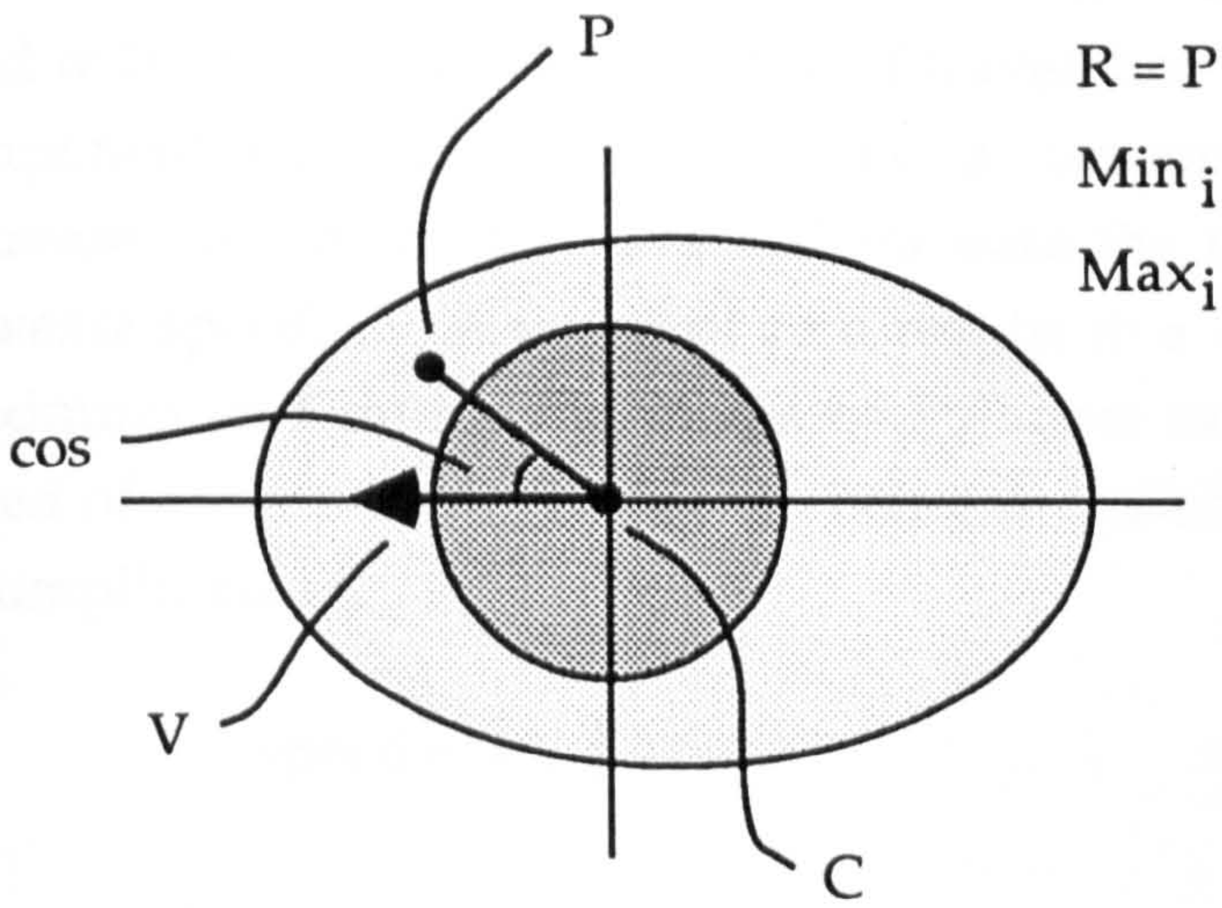
Recall that normally a scalar component in isolation has a scalar value of one at its surface, is greater than one in its interior, and will fall to zero at some distance away from its surface. The distance at which the scalar value falls to zero is defined in terms of the *radius of influence*, or simply the *influence* of a component. Increasing the value of the *influence* parameter will increase the distance at which a component will contribute a value greater than zero to the scalar field. The default value for *influence* is one, which indicates the scalar field falls to zero at one radii away from the isosurface. For scalar components such as *plane* and *cube* a similar formulation for *influence* is defined.

The *influence* parameter controls the size of the non-zero portion of the scalar field. A scalar field is symmetric when controlled solely by the *influence* parameter.

The fact that the scalar field of a component is symmetric, extending in all directions equally, causes scalar components to exhibit an anticipation when interacting in an animation sequence. Examples of this can be seen in figure 6.1. Before the two spheres in the figure touch, they are both being 'attracted' to each other. This is caused by the fact that as the two spheres approach they enter the others *radius of influence*. This creates the change in the surfaces demonstrated in the figures. A physical analogy for a material which is simulated by this behaviour would be something akin to magnetic jelly.

A more realistic interaction can be accomplished by taking the *velocity* of each component into account when evaluating the scalar component. The *influence* of a component should not be as great in the direction of travel as in the opposite direction. The implementation of *velocity* will allow the interaction of spheres which are approaching and pulling apart to be different. Two spheres which approach will not interact until they are almost touching. As the spheres pull apart they will continue to interact at a greater distance than normal.

Velocity can be implemented to allow the user to specify the value for *influence* in the direction of travel, as well as the opposite direction. The *influence* in-between these two extremes is interpolated according to the formulae and figure below.



$R = P - C$

Min_i = minimum influence

Max_i = maximum influence

By rearranging the equation of the vector dot product, the value of *cos* can be expressed as:

$$\cos = \frac{V \bullet R}{|V| * |R|}$$

The value of *cos* will vary between one and negative one. It will have a value of one when *P* is along *V* on the left of the sphere and negative one in the opposite direction. This is used in a formula for interpolating between the maximum and minimum of the *influence* parameters specified, depending upon the direction of the vector *R*. The maximum value for *influence* is obtained when *R* is along *V*. This is the *influence* in the direction of travel. The minimum is obtained when *R* points in the opposite direction. When *R* is in-between the two extremes, an interpolation between the two *influence* parameters is used. The formula to accomplish this calculation, using a linear interpolation, is:

Min_i = minimum *influence* for component

Max_i = maximum *influence* for component

$$\textit{influence} \text{ at point } P = Max_i + \left(\frac{\cos + 1}{2}\right) * (Min_i - Max_i)$$

The value of *influence* obtained from the above formulae is used as indicated in chapter three, section 3.2.1, in the calculation of a scalar value from a component at any given point.

In the above implementation, the specification of the *velocity* parameter is used only to indicate the direction of travel, not the speed at which the component is travelling. The speed of a component could be used to decrease the minimum value and increase the maximum value of *influence* specified. A speed of zero results in a symmetric scalar field. The maximum and minimum values of *influence* can be derived from the speed of a component rather than being specified by the user. This can be accomplished as:

$$\text{speed} = |\,V\,|$$

$$\text{Min}_i = \frac{\text{influence}}{\text{speed} + 1}$$

$$\text{Max}_i = \text{influence} * (\text{speed} + 1)$$

Examples of different *velocities* of a single sphere are shown in figure 6.14. Two examples of animation involving *velocity* are shown in figure 6.15 and figure 6.16. In figure 6.15 only one sphere is moving. In figure 6.16 both spheres are moving.

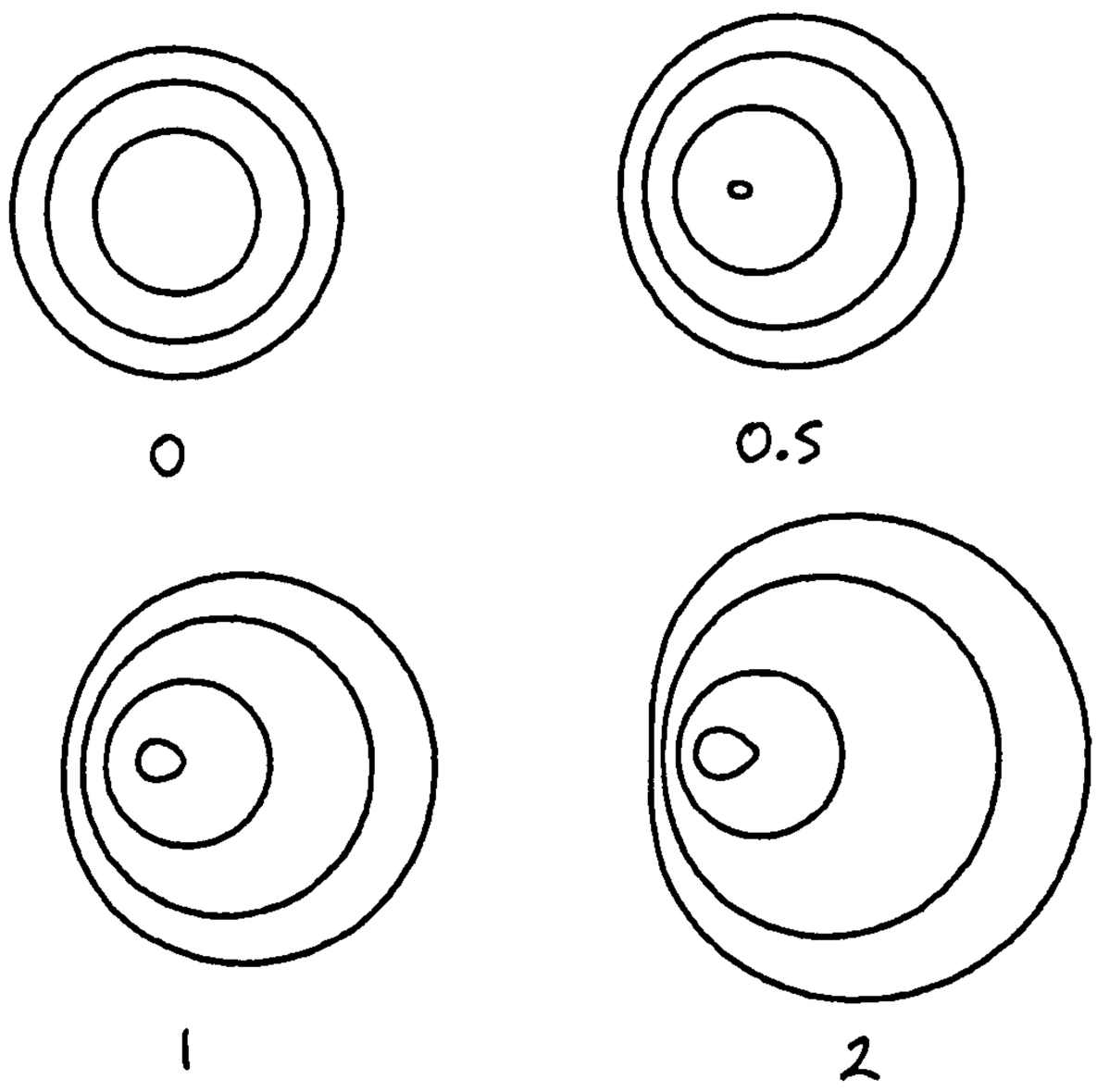


Fig 6.14 Different values for *velocity* on a sphere. Isosurface values between zero and one are shown

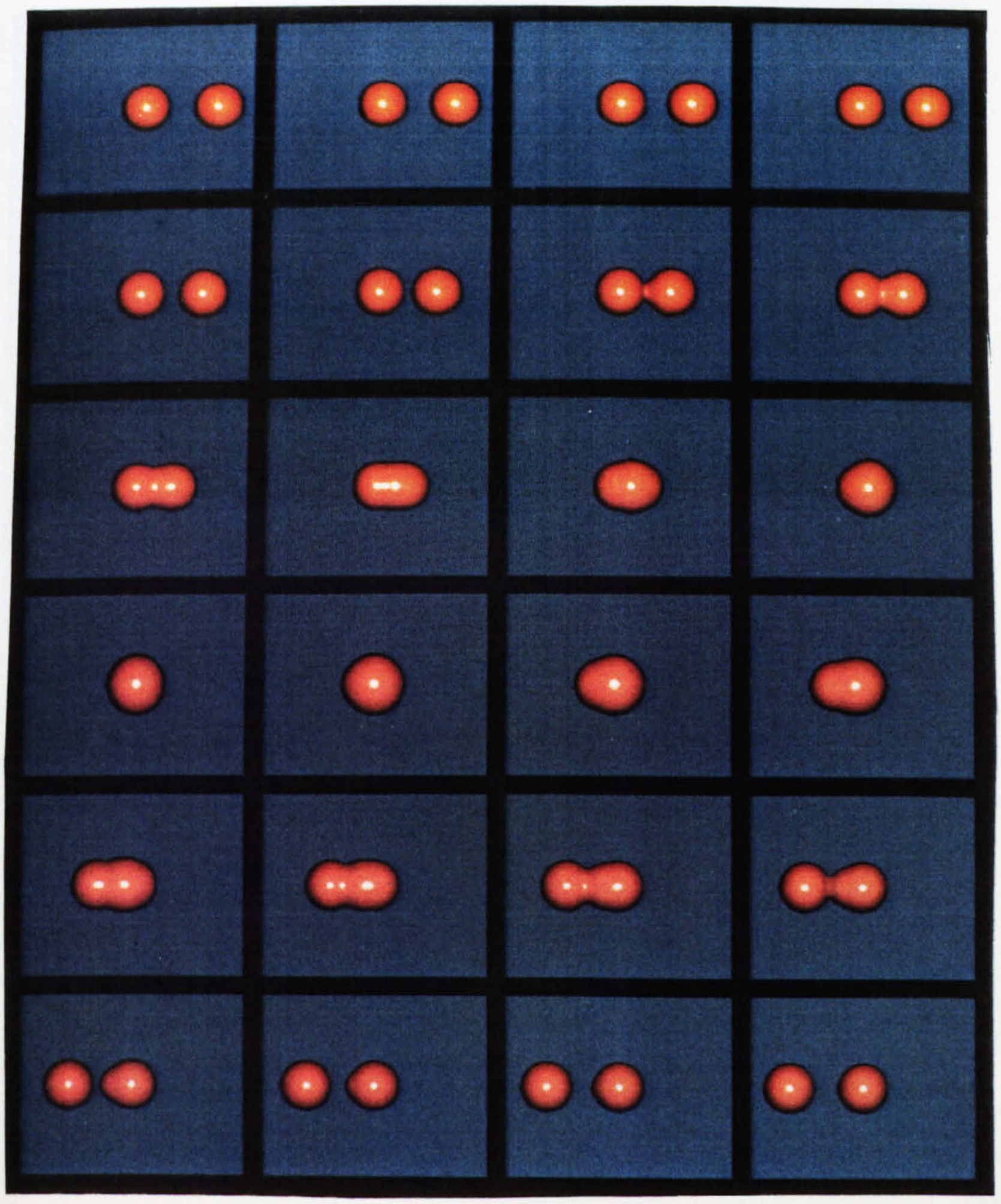


Fig 6.15 *Velocity* implemented on one moving sphere

Notice that initially as the moving sphere on the right approaches the stationary sphere on the left, the *influence* of the moving sphere is clearly less than the stationary sphere. As the moving sphere emerges from the stationary sphere, the *influence* has increased, the stationary sphere is now 'behind' the moving sphere.

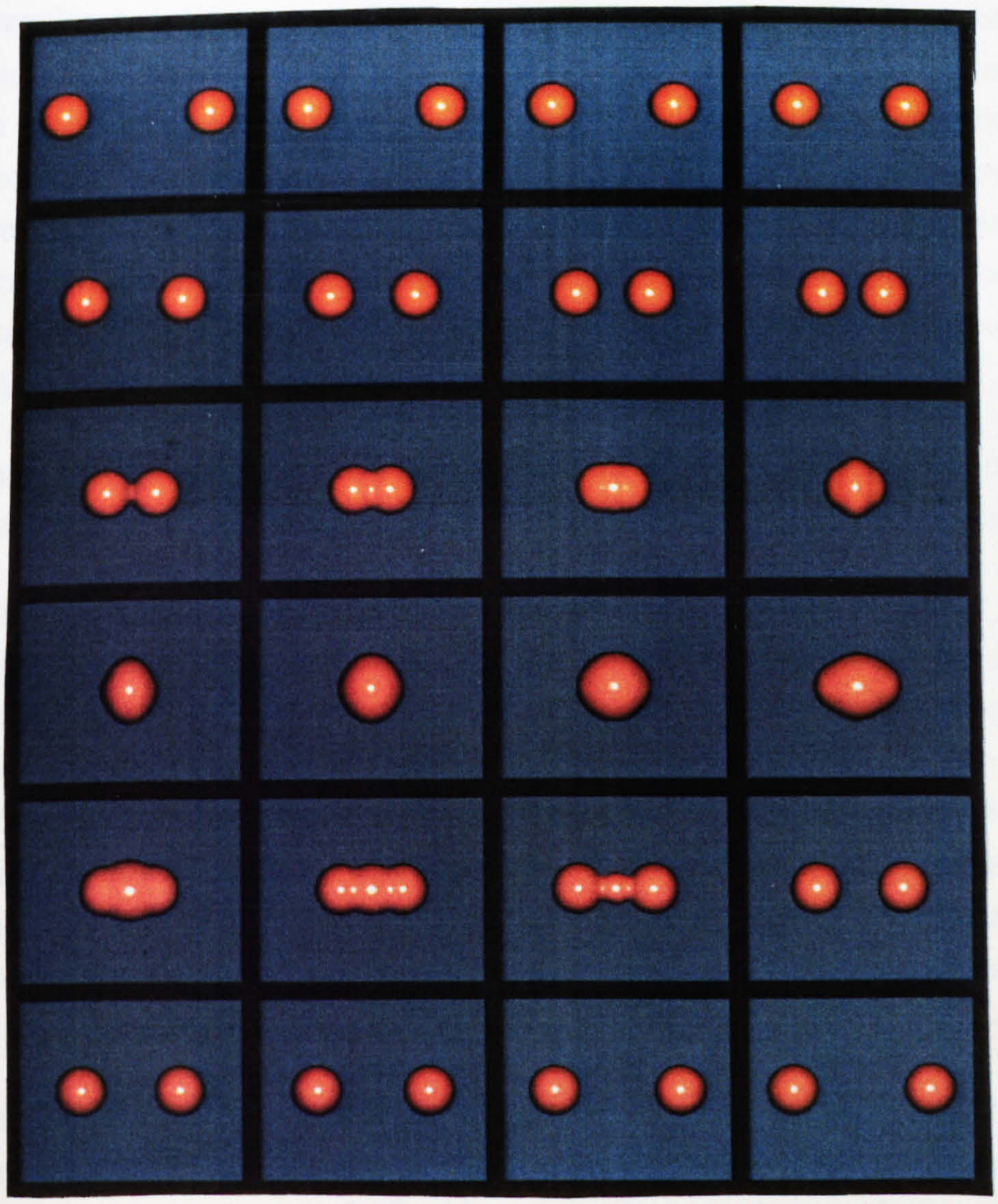


Fig 6.16 *Velocity* implemented on two moving spheres

As the spheres approach they do not interact until they are almost touching, they both have a small *influence* 'ahead' of them. As the spheres move apart they interact at a greater distance than before.

6.4.3 Bias

The implementation of *velocity* as indicated in section 6.4.2 improves the interaction of moving scalar components. However, as the components move about in isolation their scalar fields do not change shape. The components should squash as they decelerate and stretch as they accelerate. This effect can be accomplished using an animation system. The animation system could vary the scaling of a component in the direction of travel depending upon the rate of acceleration. No additional changes are needed in SFDL to implement 'squash and stretch'.

'Squash and stretch' are used mainly at the beginning and the end of movements to accentuate the motion and give the appearance of flexibility.

To complete the implementation of realistic movements for scalar components a non-symmetric scaling is needed. Notice in figure 6.15 that as the moving sphere emerges from the stationary sphere, the moving sphere quickly assumes its normal shape, even while the stationary sphere is biased towards the moving sphere. This is a result of the implementation of *velocity*. *Velocity* is based upon changes in the *influence* parameter. *Influence* is implemented so as not to change the shape of the isosurface of the component for which it is calculated. A scalar value of one remains one for a component regardless of the value of *influence*.

A method of biasing the moving sphere in the opposite direction to that of the travel is needed. If a geometric scaling technique were used, the moving sphere would elongate on both sides, away from and towards the stationary sphere. This is not the effect desired. A method of biasing the moving sphere towards the stationary sphere is needed. *Bias* is a non-symmetric operation, giving a different result than scaling would.

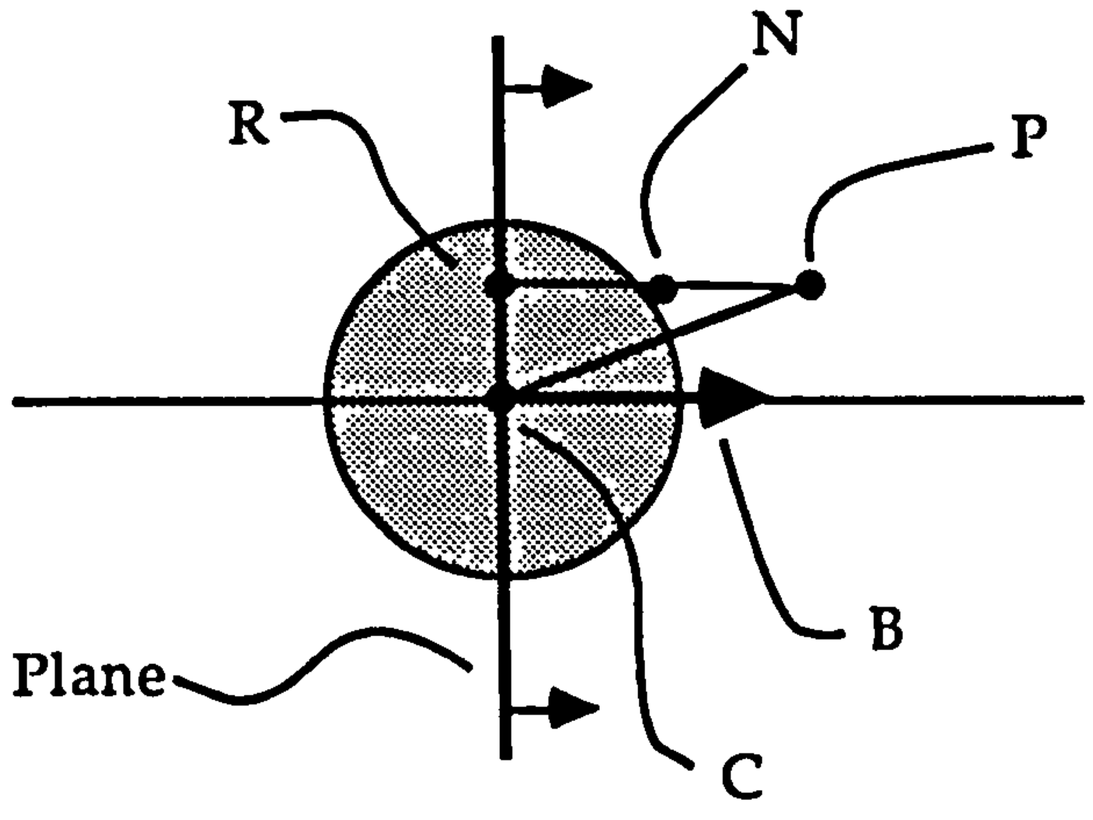
Using a *bias* control the example in figure 6.15 could appear more realistic. Unlike *influence*, *bias* will actually change the shape of an isosurface. *Bias* is specified using a direction in which the bias occurs, and a positive value, called the *bias magnitude*. A *bias magnitude* greater than one will stretch half of the component in the direction indicated, a magnitude of less than one will squash the component.

The component is divided in half by a plane running through its centre, perpendicular to the *bias* direction.

Bias is implemented by changing the location of the point for which a scalar value is required. The *bias* calculation is performed for each scalar component defining a scalar field. The *bias* calculation finds a new point, which is then used in the evaluation of the scalar component. A value for *bias magnitude* which is greater than one implies that the new point moves closer to the interior of a component, generally increasing its scalar value. A *bias magnitude* of less that one moves the point further away from the centre, causing a smaller value to be returned for the scalar field evaluation. Moving points towards the centre of a scalar component will increase the isosurface size. Moving a point further away will cause the isosurface to decrease in size.

The calculation of *bias* will be performed before the calculation of the scalar value. The *bias* calculation will result in a new location for the point at which a scalar value is required. The point calculated using *bias* is used in the remainder of the scalar component evaluation.

The first implementation of *bias* is described:



sign = the result of the plane equation being evaluated at P

The value of *sign* will be positive if *P* is on the correct side of the plane, in the example above, the right hand side. If *sign* is negative the point *P* is not changed. In the first implementation, the point *P* is scaled towards the centre of the component, along the line between *P* and C according to the *bias magnitude* value. The larger the *bias magnitude*, the closer the point

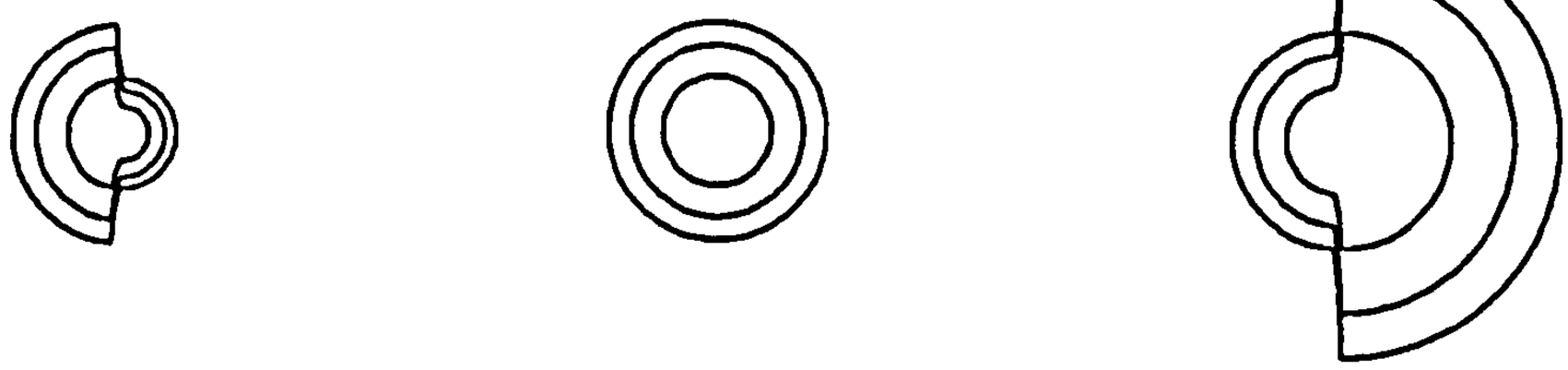


Fig 6.17 Example of an incorrect implementation of *bias*

will be moved to the centre of the component. *Bias magnitude* is inversely proportional to the distance the point is moved away from the centre of the component. This will have the desired effect of elongating the component for larger value of *bias magnitude*. A value of one for *bias magnitude* will leave the point unchanged. The first attempt at *bias* is given as:

if $(\text{sign} > 0)$ **then**

$$P = \frac{P - C}{\text{Bias}_{\text{magnitude}}} + C$$

endif

This first implementation does not give the desired result however, as shown in figure 6.17. The correct implementation will have the displacement of a point grow gradually less and less as the point approaches the plane. This is accomplished through a calculation involving the point *R*.

Recall that:

$$\text{Projection of CP along B} = \frac{CP \bullet B}{|B|^2} \; B$$

This is used to find point *R*, by subtracting the projection defined above from *P*:

$$R = P - \frac{CP \bullet B}{|B|^2} B$$

Point *R* is the point on the plane closest to point *P*. The vector *RP* is in the direction the point will be moved to accomplish the *bias* calculation. The calculation of the new point, *N*, is then:

$$N = R + \left(\frac{CP \bullet B}{|B|^2} B \right) * \frac{1}{Bias_{magnitude}}$$

Using simple algebra, this is changed into the following final expression of the new point:

$$N = P + \left(\frac{CP \bullet B}{|B|^2} B \right) * \left(\frac{1}{Bias_{magnitude}} - 1 \right)$$

Several examples of a torus with varying *bias magnitudes* are given in figure 6.18. An animation similar to that in figure 6.1, incorporating *bias* is given in figure 6.19.

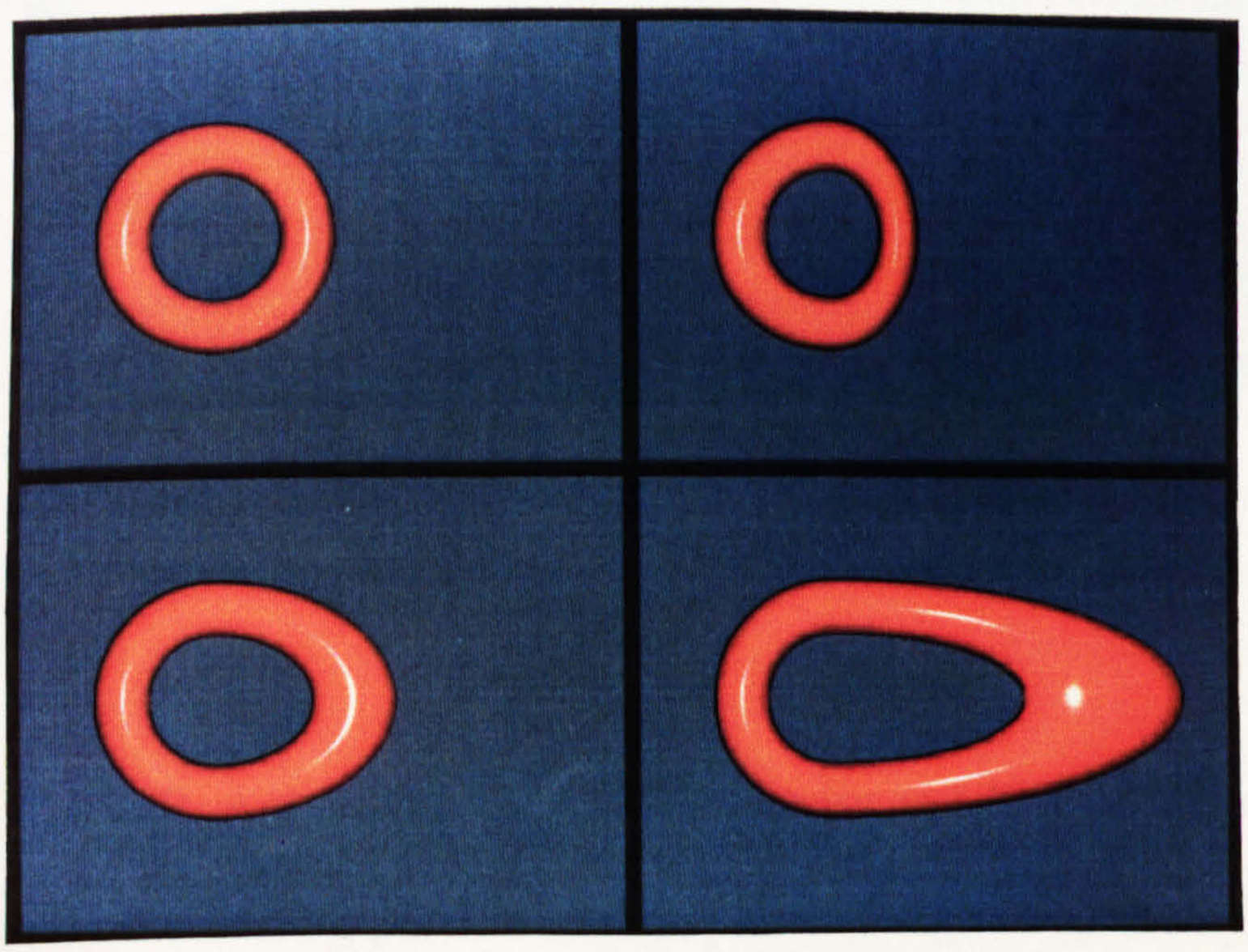


Fig 6.18 Several examples of *bias* defined on a torus

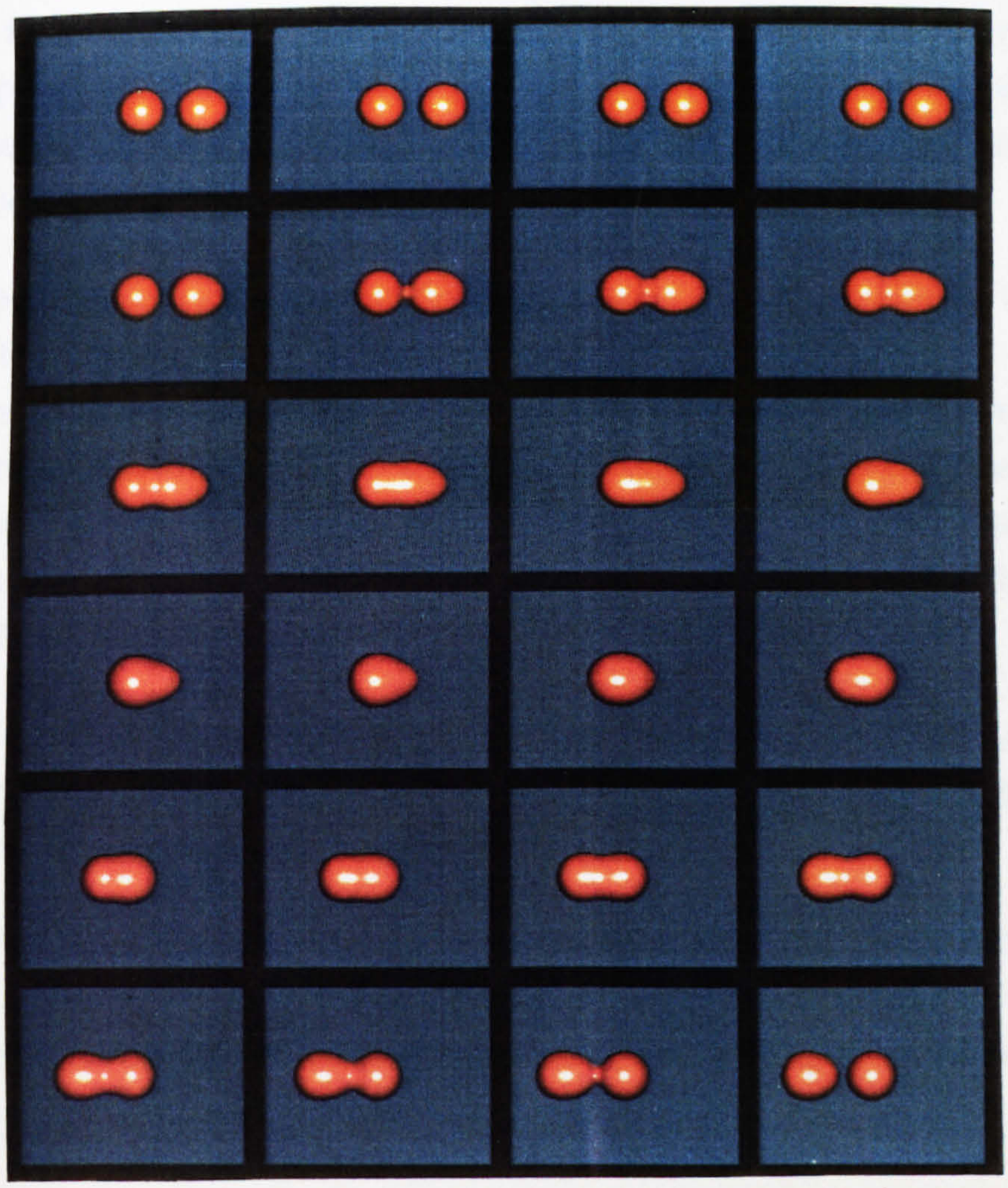


Fig 6.19 A moving sphere with *velocity* and *bias* implemented

Notice that the trailing portion of the sphere elongates as the sphere increases in speed. This enhances the animation effect.

6.5 Metamorphosis

Metamorphosis is a process in which an object undergoes a 'complete change in character or appearance' [CED]. Metamorphosis is typically used in computer graphics to transform one object into another. A simple example would be a cube which metamorphoses into a sphere. More complex examples are easily found.

The specification of a metamorphosis process typically indicates the starting shape, the ending shape, and the amount of time in which the metamorphosis occurs. The final result for all possible implementations of metamorphosis are the same, the desired shape is achieved. The methods of achieving a metamorphosis are varied. Consider two isosurface models which are each composed of many scalar component spheres. Three methods of implementing metamorphosis in this situation are:

1) The starting and ending models are polygonised producing two polygonal models. These polygonal models are used in a polygonal metamorphosis technique.

2) A mapping between the sphere components in the two models is found. This mapping is used to produce intermediate models which consist of the rearranged spheres. This technique involves a rearrangement of the scalar components.

3) The mix scalar operator is used to create the intermediate models. These models initially consist entirely of the first and then have increasingly large proportions of the second model until the final result is achieved. This calculation is based upon the actual scalar field values.

Although the net result of these three methods of implementing metamorphosis will be the same, entirely different intermediate stages will be produced by each of them.

The first implementation of metamorphosis will most likely result in a flurry of polygons in the intermediate stages which does not resemble either of the models. This flurry of polygons will gradually coalesce, to resemble

and then become the second model. Occasionally a continuous surface may be maintained by the polygonal metamorphosis technique. This will depend largely upon the similarity of the starting and ending models. If the two models are similar in size and shape a continuous surface may possibly be maintained. If the two models are dissimilar in size and shape or have fundamentally different surface topologies it is extremely difficult to create a continuous surface in the intermediate stages of a metamorphosis. The effect created when using this technique of metamorphosis may be useful in limited circumstances. It is envisaged that a continuous surface will be desired in most circumstances.

The second method of implementing metamorphosis produces continuous surfaces. In the intermediate stages the movement of the spheres will be apparent as they move to their new locations. Again, this is a useful effect which will satisfy some circumstances. The implementation of this technique is hampered by the diverse range of scalar components and scalar operators with which it is possible to describe a scalar field. The example above, which contains many spheres, is easily handled using a simple implementation of metamorphosis. In an example which contains many different scalar components and scalar operators, it may be difficult to produce a mapping from one model into another. Heuristic rules can be created to handle individual circumstances, but are difficult to generalise. For instance, it is not clear how a mapping from a sphere into a torus is accomplished. This method of implementing metamorphosis may be useful in limited circumstances, but is difficult in general.

The third and last method proposed for implementing metamorphosis is based upon the actual scalar field values of the two models. An interpolation at any point between two scalar field values is found. This interpolation is easily implemented by finding the scalar field value at a point in each model and then mixing between the two. This is accomplished using the *mix* scalar operator in SFDL. This interpolation is expressed as an algebraic expression:

$$F_{ab}(P) = F_a(P) + m * \left(F_b(P) - F_a(P)\right), \quad 0 \le m \le 1$$

An example of the results of this method are given in figure 6.20.

The method of interpolating scalar field values can be enhanced by using a mapping between the spatial extents of the two models. To illustrate a

situation which will be improved with this additional mapping, consider two models which are identical in shape and located so that they do not overlap. A mixture in-between these two models will produce one model which fades away and one model which appears out of nowhere. A more realistic solution for this metamorphosis is to map the first model spatially into the second and then perform the interpolation.

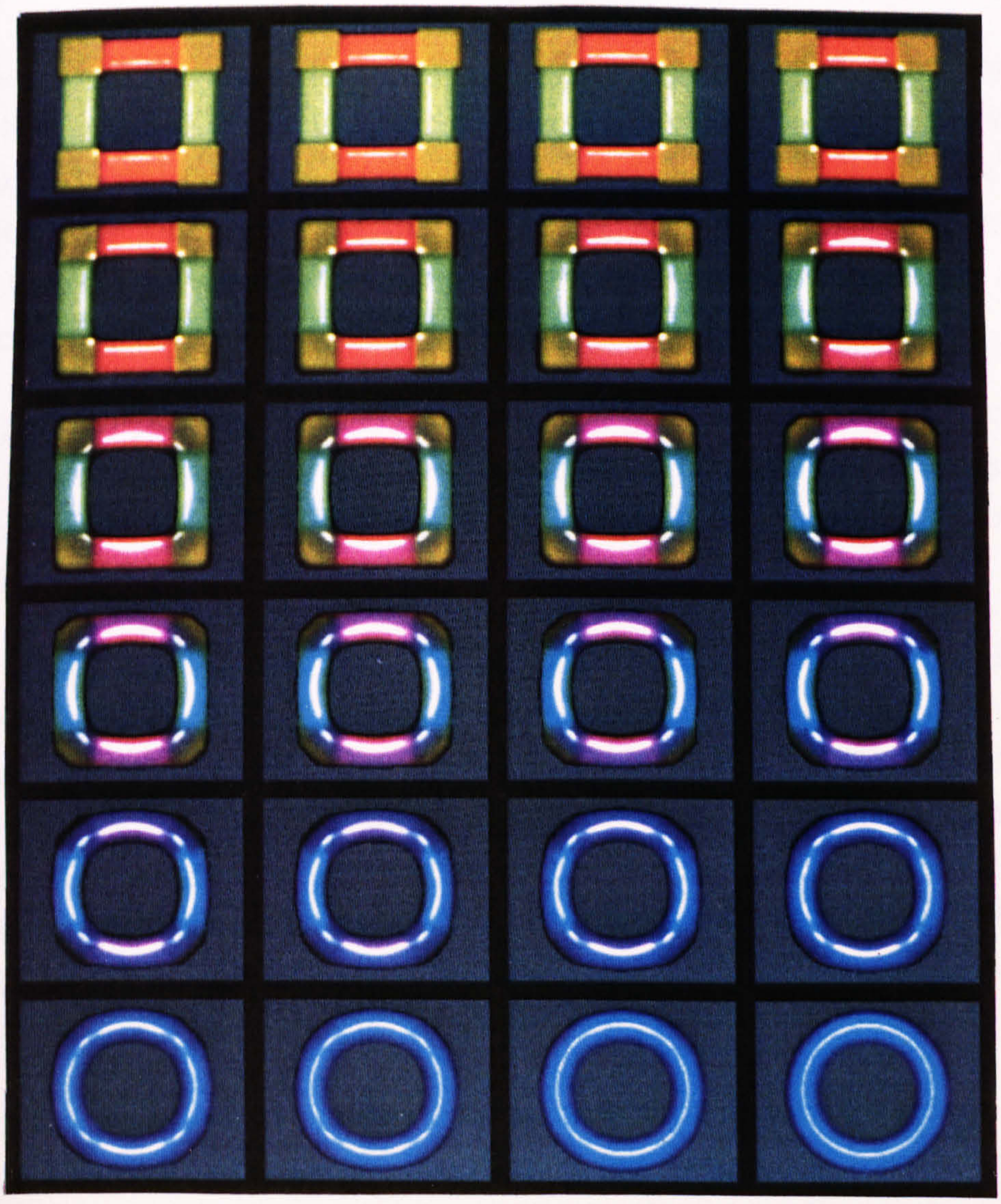


Fig 6.20 Metamorphosis between a frame and a torus

The frame is produced by adding cube components scaled and oriented properly. Notice that the colour of the resulting surface is also affected by the metamorphosis operation.

6.6 Interaction with traditional modelling techniques.

In specifying the interaction between an isosurface model and a traditional model it is desirable for the two models to interact in a realistic manner. For instance, if an isosurface model were to approach and then touch a traditionally modelled vase, one would not expect the isosurface model to pass through the model of the vase. This would be the case if care was not exercised in designing an animation sequence.

The general problem of collision detection and deformation are not discussed in this thesis. However, a subset of the problem is discussed. A general method of handling the interaction between isosurface models and traditional models is left as future research.

The partial solution proposed in this research is to manually roughly translate the traditional model into an implicit representation. This is accomplished by a designer constructing a replica of the traditional model using the isosurface modelling techniques. This process may be unwieldy for traditional models which undergo animation. This would require that the isosurface replica changes as the traditional model changes. This may not be easily accomplished, as the isosurface replica is created manually from the traditional model.

Once the traditional models are converted into isosurface models, they are used to restrict the regions in which an isosurface can be generated. This is accomplished by subtracting the replica implicit model from the scalar field value of the normal isosurface model. This has the effect of creating a field around a traditional model which an isosurface can not enter.

This technique can be expressed using algebra as follows. M is the isosurface model, and R is the isosurface replica of the traditional model.

$$\text{Value}(P) = M(P) - R(P)$$

To illustrate the use of this technique in a particular circumstance, an example is presented. Consider a sewer grating and a road, both of which are traditionally modelled. An approximation to this can be easily generated

using the isosurface modelling techniques. An isosurface 'slime' can then travel along the road and pass into the sewer without moving directly through any element of the sewer grate or below the road surface. Figure 6.21 illustrates an isosurface which moves through a grating.

The accuracy of the replica determines the visual success of this approach. Complex traditional models are difficult to reproduce using the isosurface modelling techniques, demanding that the replica be only an approximation to the actual model.

The technique discussed in this chapter provides a powerful means for combining isosurface and traditional models, in limited circumstances.

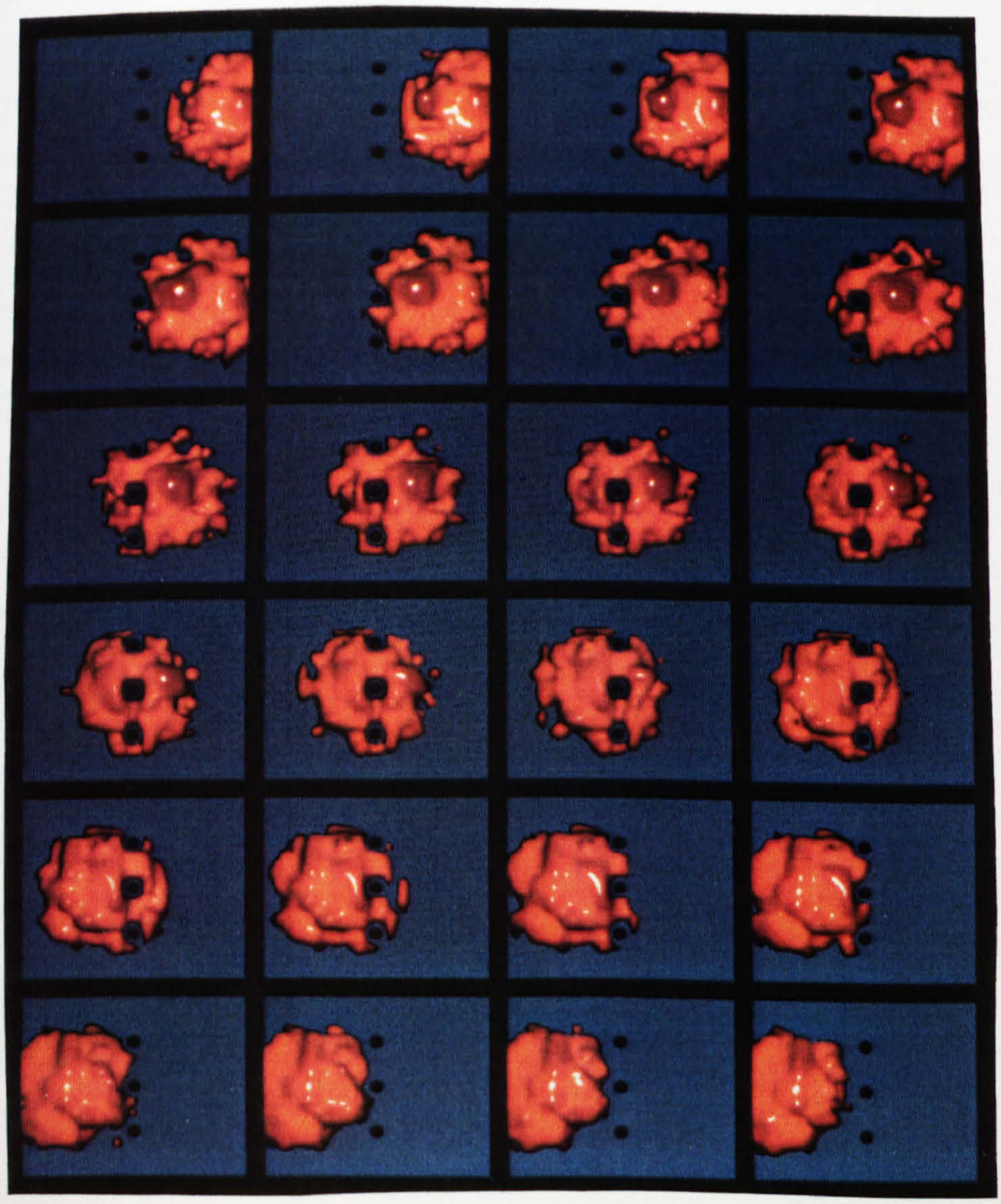


Fig 6.21 Interaction of soft slime and a traditionally modelled grating

The grating is represented by the three dark circles. The slime is represented by noise being added to an ellipsoid. The noise remains stationary, causing the ellipsoid to distort as it moves.

6.7 Conclusions

Using the isosurface animation techniques described in this chapter it is possible to easily describe animation sequences which would be extremely difficult to reproduce using traditional modelling techniques. The ease and efficiency with which these animation sequences are specified and generated is in itself a powerful argument in favour of implementing isosurface modelling and animation techniques. If images similar or related to those presented in this thesis are needed in a project, isosurface techniques are ideally suited for their production.

One of the disadvantages of isosurface animation is the lack of real time interaction which is available using traditional techniques in conjunction with appropriate computer hardware and software. The isosurfaces in a model need to be recalculated for each frame of an animation in which they change. This creates a large compute load. This compute load can not be met in real time using generally available computer hardware. Proposals for the solution to this problem are presented in the next chapter.

The isosurface animation techniques have been implemented with two small changes to each scalar component to enhance their animation capabilities. SFDL remains largely as it was specified in chapter three. Animation is accomplished through an animation system which produces SFDL programs as needed.

The isosurface animation techniques present unique opportunities to the designer which would be difficult to reproduce in non-traditional animation systems. The ease with which *velocity* and *bias* were implemented indicate the flexibility of the isosurface approach to modelling. It is considered that isosurface animation techniques offer a valuable addition to the many methods of producing animation sequences. No major problems have been found in this research to preclude the use of isosurface animation. The animation sequences presented offer only a small taste of what is potentially possible using the techniques.

Chapter 7

Graphics system description

7.1 Introduction

Techniques have been discussed in previous chapters for creating, altering the appearance, visualisation and animation of isosurfaces in scalar fields. Isosurface models are created through the combination of a variety of scalar components and scalar operators. It is possible to create complex isosurfaces with relative ease using the techniques discussed in this thesis.

The description of isosurface models takes place in a graphics system which automatically creates scalar field description language (SFDL) programs. Animation of the isosurface models takes place in an animation system which may or may not be part of the graphics system. Facilities for the description of an animation sequence are not included in SFDL.

A variety of graphics and animation systems are able to incorporate the isosurface modelling techniques. During the course of this research a graphics system with integrated support for animation has been created, called JED. JED supports the isosurface modelling techniques as well as a number of the traditional modelling representations.

Animation in JED is accomplished using parametric techniques. Any value that needs to be changed during an animation sequence is replaced by an expression. The value of an expression may change during an animation sequence. An example of a value that can be replaced with an expression is the number of degrees in a rotation. By varying the value of an expression an animation sequence can be created.

The isosurface modelling techniques are not restricted to any particular animation system or any particular type of animation system. JED was created in order to have complete freedom in all aspects of isosurface modelling, including the specification of the interaction and integration of isosurface modelling with traditional modelling. This freedom may not have been possible if the isosurface modelling techniques had been incorporated into an existing animation system during this research.

The structure used to represent graphical models in JED is discussed in this chapter, as well as: parametric animation; incorporating the isosurface modelling techniques into the graphics system; implementation details of JED; and finally, the conclusions reached.

Throughout this chapter short sequences of the JED language are presented in examples. While an effort has been made to present the examples and their context in a clear manner, no effort has been made to describe the entire JED language in this thesis. The portions of JED dealing with isosurface modelling are discussed in this chapter. Although the capabilities available in the traditional modelling aspects of JED are utilised in this research, a discussion of their implementation and methods for their description is considered to be peripheral to the focus of this research.

7.2 Structure of the graphical models

The structure of the graphical models in JED, as well as aspects of their implementation have been influenced by research at the university of Calgary in the Graphicsland project [Wyvill, McPheeters and Garbutt 1986]. The hierarchical data structure used in JED, a directed cyclic graph, is similar to the data structure used in the Graphicsland project.

The hierarchical models created in JED have a tree structure. At the leaves of the tree are the graphics primitives from which models are created. The interior branches of the tree represent groupings of both graphical primitives and other groups.

There are three traditional modelling representations in JED: individual polygons; translational sweeps of a polygon; and rotational sweeps of a polygon. Using the facilities within JED each of these three techniques can be combined and transformed to form complex models.

In order to create models within JED, a variety of objects are created, initialised and organised in order to represent the desired structure. Each object in JED is of a particular type, which must be explicitly stated. The attributes that can be specified within each object depends upon its type. For instance the focal length of a lens can be specified in a camera object but not in a polygon. A short list of the object types available in JED are: groups; polygons; translational sweeps; rotational sweeps; cameras; point light sources; infinite light sources; spot lights; filters; values; points; tables; and all of the scalar components (soft sphere, soft ellipse, soft torus and so on).

Groups are used as the internal nodes of the hierarchical data structure. A *group* contains references to a number of separate objects, each of which can be either other *groups* or one of the graphics primitives. Each of the references contained in a *group* can be transformed using a variety of geometric transformations and surface attributes.

Using the hierarchical modelling techniques, models are built from a collection of simpler objects. Simple objects are collected in *groups*, each transformed appropriately. These *groups* are then collected into more complex *groups* until eventually the desired structure is achieved.

A short example of an object created in JED is a simplistic version of a car. This car is composed of four wheels and a body. The body is composed of two doors and a chassis. The wheels should be red, the doors blue and the rest of the body green. This car can be created in JED in a variety of ways. A partial listing of the JED instructions used to create a car is given in figure 7.1. The hierarchical data structure which represents the car is given in figure 7.2.

The geometric transformations that are available are listed in table 7.1. The attributes available in JED are listed in table 7.2.

In a hierarchical data structure, there is a possibility that an object may contain a reference to itself, or to an object which contains a reference back to the first. These situations are respectively called recursion and mutual recursion. JED has been implemented to allow both recursion and mutual recursion. The benefits of using recursion in the description of graphical models was discussed in a paper by Wyvill, Liblong and Hutchinson in 1984.

A few points regarding the syntax of the JED language may be necessary. In figure 7.1, all instructions in-between the *open* and *close* instruction apply to the object named in the *open* instruction. A *plot* instruction is the method of referencing a different object. Each *plot* reference can be geometrically transformed or have attributes set. These transformations and attributes immediately follow a *plot* command and are specified using the *instance* instruction. These *instance* instructions apply to only the single *plot* reference which precedes them.

```
open car
    type group
    plot body
    plot wheel
        instance origin <10, 0, 10>
        instance rgbcolour <1, 0, 0>
    plot wheel
        instance origin <-10, 0, 10>
        instance rgbcolour <1, 0, 0>
    plot wheel
        instance origin <10, 0, -10>
        instance rgbcolour <1, 0, 0>
    plot wheel
        instance origin <-10, 0, -10>
        instance rgbcolour <1, 0, 0>
close

open body
    type group
    plot chassis
        instance rgbcolour <0, 1, 0>
    plot door
        instance origin <10, 0, 0>
        instance rgbcolour <0, 0, 1>
    plot door
        instance roty 180
        instance origin <-10, 0, 3>
        instance rgbcolour <0, 0, 1>
close
```

Fig 7.1 Partial list of JED instructions to produce a car

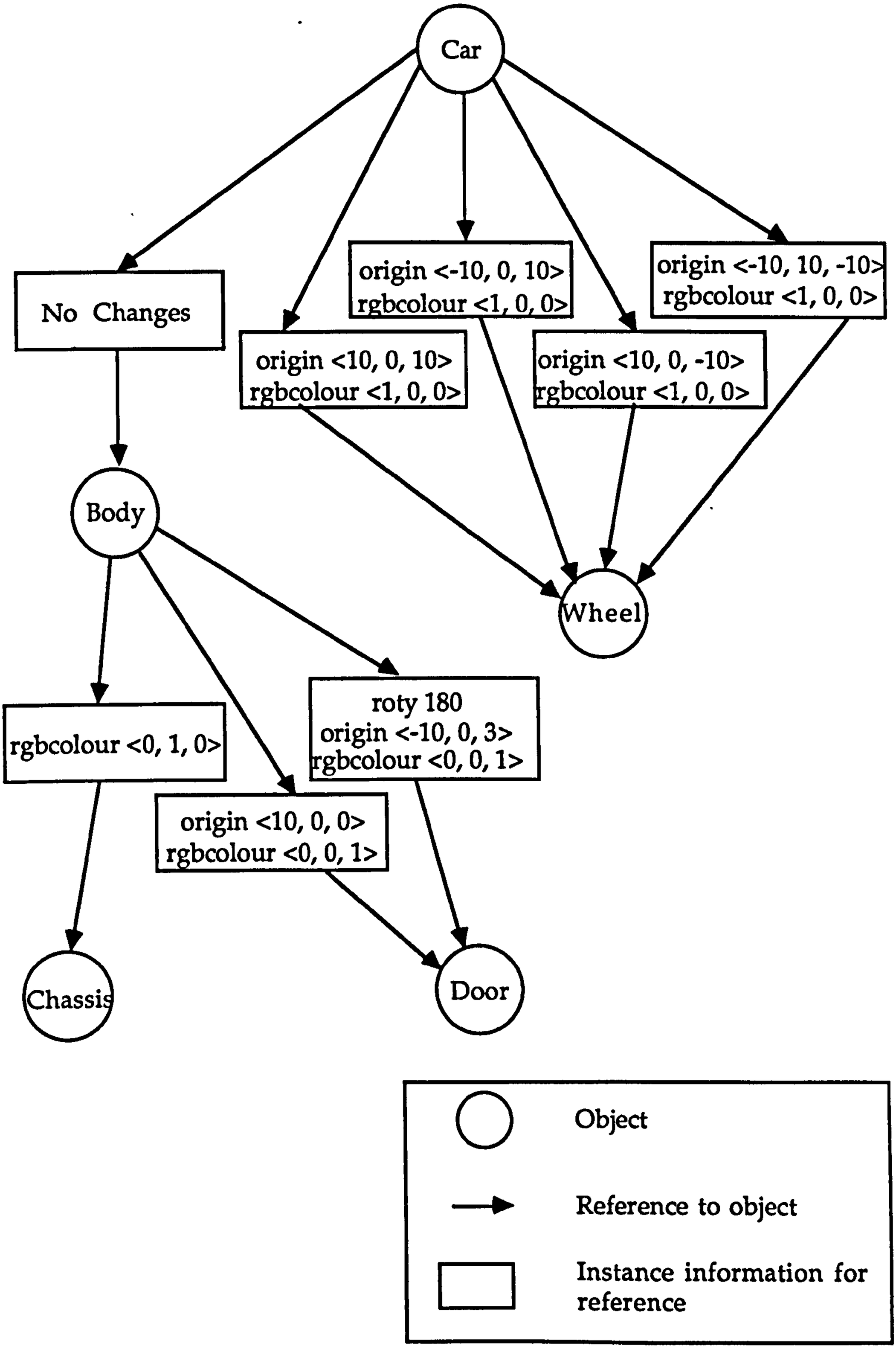


Fig 7.2 Partial structure of a car

origin <num, num, num>	relative change of origin (translation)
scale <num, num, num>	relative scaling
rotx num	rotation along x axis
roty num	rotation along y axis
rotz num	rotation along z axis
align <num, num, num> to <num, num, num>	align the first vector into the second
rotaxis <num, num, num> by num	rotation about the vector by the indicated angle

Table 7.1 Geometric transformation instructions available in JED

rgbcolour <num, num, num>	specify colour in red, green and blue components
hsvcolour <num, num, num>	specify colour in hue, saturation and value components
translucency num	specify the translucency of an object
shading flat | gouraud | phong	specify the shading model to apply to an object
ambient num	specify the ambient light coefficient of an object
specular num	specify the specular coefficient of an object
texture name	specify a texture map to be applied to an object

Table 7.2 Attribute instructions available in JED.

7.3 Parametric animation

Any one of a number of animation techniques could have been implemented in JED instead of parametric animation. Some of these alternative animation techniques are introduced in chapter two. Parametric animation was chosen due to its suitability in describing the animation sequences required during this research project. The animation sequences

in this research are largely of a technical nature. The ability to easily animate any value in JED was considered more important than being able to create a sequence based on dynamics or the behaviour of the models. Animation sequences incorporating character animation can be produced within JED, although no special support has been supplied.

The parameters specified in JED instructions are composed of either one or three values. Parameters with three values are called triplets, and are always enclosed in pointed brackets ('<' and '>'). A parameter composed of a single value is not enclosed in brackets unless it involves an expression, in which case it is enclosed in round brackets ('(' and ')').

The specification of an animation sequence is accomplished by replacing the values which must change during an animation with expressions involving a *time* variable. There are three types of variables in JED: *values*, *points* and *tables*.

Time is a *value* type variable, it evaluates to a single numeric value. A *point* type variable evaluates to a triplet. A *table* variable will evaluate to a single number or a triplet, depending upon the context in which it is used.

Although the *time* variable is normally used to control an animation in JED, any alternative *value* type variable would be sufficient. The parameter which is used in the definition of an animation sequence is the one used to control the animation. In the subsequent examples of animation presented in this thesis, *time* is used as the parameter controlling the progress of an animation sequence.

In creating an animation sequence, the units and limits for *time* will have to be defined. *Time* is a value defined in the real number system. Typically, values of *time* indicate frame numbers or seconds, any alternative meaning can be applied to the value of time. Negative values for time are valid. Examples which specify animation in JED are given below.

A simple rotation about the *x* axis by 360 degrees is specified as:

```
rotx  360
```

An animation of the above rotation between *time* values of zero and ten is specified as:

```
rotx  ( time * 36 )
```

Operator	Meaning
n + m	addition of single numbers or triplets. The triplets are added element by element.
n - m	subtraction of single numbers or triplets. The triplets are subtracted element by element.
n * m	multiplication of single numbers or triplets. The triplets are multiplied element by element.
n / m	division of single numbers or triplets. The triplets are divided element by element.
$<N_x, N_y, N_z>$ distance $<M_x, M_y, M_z>$	distance between two points
n mod m	modulus
n div m	integer division
n floor	the integer ≤ n
n ceil	the integer ≥ n
n abs	absolute value of number
$<N_x, N_y, N_z>$ first	first component of triplet
$<N_x, N_y, N_z>$ second	second component of triplet
$<N_x, N_y, N_z>$ third	third component of triplet

Table 7.3 Operators available in an expression

In the above instruction, the expression enclosed in brackets replaces a single value. The type of brackets indicate that the expression evaluates to a single numeric value. If *time* is zero, the above expression evaluates to zero. If *time* if ten the expression evaluates to the desired value, 360. As the *time* value changes, so will the value of the expression. An expression involving a triplet is given as:

origin < (time * 10), ((time - 3) * 20), 0 >

Note that an expression evaluating to a triplet is enclosed in pointed brackets. Expressions in JED can be arbitrarily complex. A list of the possible operations available in an expression are listed in table 7.3.

Expressions in JED can also involve *table* elements. A *table* is a series of values which are interpolated in order to obtain a result. Complex animations are implemented using *tables*. An example of a *table* named *fred* is given as:

```
open fred
    type table
    1.0   value 0.5
    2.0   value 10
    3.0   value 0.5
close
```

The numbers preceding the *value* instructions are the indices of the *table* elements. The value of the *table* at the indices is given. The value of the *table* in-between the indices is calculated using interpolation. An expression involving the *table* above is given as:

```
influence  ( fred [ time ] )
```

The expression is surrounded by round brackets, indicating that it evaluates to a single value. The value of the above expression will change according to the value of *time*. If *time* is less than or equal to 1.0 (the first element of *table fred*) the *table* will return the value of the first entry, 0.5. As *time* varies between any two elements in the *table* a cubic interpolation is used to determine the value which is returned. Values of *time* beyond the index of the last entry, in this case three, will yield the value of the last entry. *Tables* can also involve triplets in a similar manner.

It is often desirable to have a model follow an arbitrary path in an animation sequence. This can be implemented in JED in two stages. Firstly, a table is created which lists the locations of the model at each key point in time required. The locations in-between these key points will be determined using interpolation. For example:

```
open  newpath
    type  table
    1.0  value  <0, 0, 0>
    2.5  value  <10, 3, 0>
    5.3  value  <-5, 6, 0>
```

```
            8.0  value  <0, 0, 0>
        close
```

In order to move an object along the above path, the following instruction is used:

```
origin  < newpath [time] >
```

This will result in a relative change of origin according to the table specified. One more instruction is required in order for the model to be correctly oriented along the path as it moves. Assuming the vector <1, 0, 0> (relative to the model) should always point along the path, the instruction is:

```
align  <1, 0, 0> to <<newpath [time + 0.1]> – <newpath [time]>>
```

This uses a numerical technique to gain an approximation to the gradient of the path at any point. In this example, the *rot* instruction should be issued before the *origin* instruction to obtain the desired result. Banking of the model as it moves along the path is not handled. Banking can be handled through the addition of an additional rotation along with a table of appropriate banking values.

Using parametric animation, animation sequences were quickly created to test the isosurface techniques as they were being implemented. The parametric animation techniques implemented in JED have proven flexible and capable of generating all of the animations required during this research.

7.4 Isosurface models

Models which incorporate the isosurface modelling techniques are easily created in JED. Recall from chapter three that the description of a scalar field in SFDL consists of scalar components combined using scalar operators into a structure representing the desired model. The scalar components are available in JED as object types, the scalar operators are available to transform references to objects within *groups*.

A short example demonstrating the combination of two scalar sphere components is presented:

```
open  twospheres
      type group
      plot  softsphere
      plot  softsphere
            instance origin <1.5, 0, 0>
close
open  softsphere
      type ssphere
close
```

The *softsphere* object is created and its type is set to a scalar sphere component. Additional parameters can be specified in the *softsphere* object, including: *influence, velocity* and *bias*. The *twospheres* object is defined to be of type *group*, and contains two references to the scalar sphere, the second of which is translated in the x axis by 1.5. The default scalar operator is used in this example for combining the two spheres, addition.

The two scalar spheres could be animated using the techniques discussed in the previous section; the triplet in the *origin* instruction can be replaced by an expression.

A second example is presented in which the minimum scalar value (intersection) of two spheres is found:

```
open  twospheres
      type group
      plot  softsphere
      plot  softsphere
            instance origin <1.5, 0, 0>
            instance  sintersection
close
```

A more complex example involving the *noise* and *torus* components is given as:

```
open  softmodel
      type group
      plot  softtorus
            instance rotz (time * 30)
      plot  softnoise
```

```
close
open  softtorus
      type  storus
      major  4
      minor  1
close
open  softnoise
      type  snoise
      minimum  -0.7
      maximum  0.7
close
```

The above example will produce an animation as the value of time changes. The maximum and minimum values of *noise* are sufficient to allow pieces of the isosurface produced to be disjoint from the main *torus*.

Isosurface models can be easily incorporated into traditional models, as shown in the following example. Assume that the object named *apparatus* has been defined in JED and has been modelled traditionally.

```
open  fred
      type  group
      plot  apparatus
      plot  softmodel
            instance  scale  <0.5, 0.5, 0.5>
            instance  origin  <0, 10, 0>
close
```

In this example, *softmodel* is scaled down by one half and moved 10 units up in the *y* axis, it is then displayed along with the *apparatus* model. A new object named *fred* is created which contains both a traditional model and an isosurface model.

There are no limits imposed on the complexity of the models produced, other than those implied by the finite amount of computer memory and time available for their calculation. No specific implementation defined limits are imposed.

7.4.1 Creation of an SFDL program

In order that the isosurface model created within JED be displayed, an SFDL program must be created which represents the model. It is also necessary to select a method of converting an SFDL program into a representation suitable for display purposes. At the moment only one method is available in JED, a three dimensional polygonisation of an isosurface.

In order to understand the technique used to convert the hierarchical isosurface model into an SFDL program, it is necessary to understand the method used to convert a traditional model in JED into a form easily displayed.

The routine used to traverse the hierarchical data structure producing the appropriate graphical primitives is called *plot*. *Plot* recursively reduces *group* objects into appropriate low level graphics routines. The *plot* routine implemented in this research is an extensive modification of the *plot* routine used in Graphicsland for the traversal of their hierarchical data structure. Both implementations of *plot* handle traditional models in a similar fashion. Extensive modifications have been made to the *plot* routine in this research to support the creation of SFDL programs and to allow real time interaction. The implementation of *plot* used in this research is discussed.

A *group* object typically contains many *plot* instructions. Each *plot* instruction may be geometrically transformed, have attributes changed or allow a scalar operator to be specified. The attributes and transformations that immediately follow a *plot* instruction are handled and then *plot* is called recursively to display the object referenced. This recursion stops when a graphics primitive, such as a polygon, is reached. When plotting a graphics primitive, the appropriate transformation matrix and attributes are recovered and applied. The attributes and transformation matrix are maintained by *plot* in a number of stacks. The general algorithm of the *plot* routine is listed in figure 7.3. The result of applying *plot* to the car example presented in figure 7.1 is given in figure 7.4.

```
procedure plot (object)

    if (object is a graphics primitive) then
        call the appropriate plot routine for this primitive
    end
    else begin

        The object must be of a group type if not a primitive

        for each plot instruction in the object begin
            push geometric transformations applied to this plot
                instruction onto a stack
            push attributes applied to this plot instruction onto a stack
            plot the named object
            pop the attributes
            pop the matrices
        end
    end
end
```

Fig 7.3 General algorithm of basic plot routine

```
plot (car)
    plot (body)
        push colour <0, 1, 0>
        plot chassis
            ... etc
        pop colour
        push origin <10, 0, 0>
        push colour <0, 0, 1>
        plot door
            ... etc
        pop colour
        pop matrix
        push roty 180
        push origin <-10, 0, 3>
        push colour <0, 0, 1>
        plot door
            ... etc
        pop colour
        pop matrix
        pop matrix
push origin <10, 0, 10>
push colour <1, 0, 0>
plot wheel
    ... etc
pop colour
pop matrix
push origin <-10, 0, 10>
push colour <1, 0, 0>
plot wheel
    ... etc
pop colour
pop matrix
push origin <10, 0, -10>
push colour <1, 0, 0>
plot wheel
    ... etc
pop colour
pop matrix
push origin <-10, 0, -10>
    push colour <1, 0, 0>
    plot wheel
        ... etc
    pop colour
    pop matrix
done
```

Fig 7.4 Partial listing of the routines called by *plot* to display the car in figure 7.1

There are a number of extensions required to the *plot* procedure in order to handle isosurface modelling. The first extension is the creation of a new object type, a *filter*. A *filter* is similar to a *group* in that it contains many *plot* instructions, each transformed by different attributes and geometric transformations. A *filter* has the additional ability of converting particular object types from one format to another, or to otherwise change the object types in some manner. A *filter* is able to intercept the calls to any of the graphics primitives that plot makes. When a call to a graphics primitive is intercepted, one of several possible actions can occur:

1) The graphics primitive is ignored. This can be used to filter out all graphics primitives of a certain type.

2) The graphics primitive is recorded in some form and then not processed any further. This may be used in creating an SFDL program. A scalar component can be recorded in an SFDL program in the proper format, after which processing is halted on the primitive.

3) The graphics primitive is recorded in some form and then passed along to the standard routine for its regular treatment. This may be used to gather statistics.

4) The graphics primitive is modified in some manner and then passed along to the standard routine for its regular treatment. This may be used to generate several effects, such as a polygonal mesh exploding; each polygon can be intercepted, transformed by an appropriate amount, and then handled normally to create the effect desired.

5) The graphics primitive is converted into a different format. This is useful for creating approximations to complex graphics primitives. For example a scalar sphere component can be converted into a number of polygons that approximate it.

The action that occurs when a graphics primitive is intercepted depends upon the *filter* specified. A *filter* to convert an isosurface model into an SFDL program will intercept all of the *plot* calls dealing with scalar components and produce the appropriate SFDL instructions. A *filter* is a generalisation of what is required to implement isosurface modelling.

Isosurface modelling is implemented differently in Graphicsland. In Graphicsland all of the scalar components in a group are collected in a list.

This list is later processed to obtain a scalar field value at any point. The list in Graphicsland loses the hierarchical structure information inherent in a model. This implies that operations such as intersection can not be completely implemented in Graphicsland. Also, in Graphicsland it is not possible to produce more than one isosurface polygonisation in a single model.

Using *filter* objects in JED it is possible to specify a polygonisation at any point in a model. This is useful in a number of ways, for example in creating isosurface models which do not interact. It is also useful in terms of efficiency when using the polygonisation routine implemented in this research. For example, if two isosurface models are far removed from one another and do not interact, they can be polygonised separately. An alternative in this case to separate polygonisation is to specify a large polygonisation search space and polygonise the models together. The advantage implied by the last point will depend upon the polygonisation technique used. A further advantage to specifying a polygonisation in a hierarchical model is the ability to reference the polygonisation multiple times, saving the need to calculate the isosurface for each reference. This is useful in character animation for example. A character can be built using the isosurface modelling techniques, polygonised at a particular point in the hierarchy and then referenced a number of times. As each reference can be transformed, a character can appear more than once in a scene, each time at a different scale, rotation or location.

JED has been implemented so that many different filters can be used in a model concurrently. It is necessary to specify which of the available *filters* is to be used in each of the *filter* objects created. Each of the *filters* available in JED contains information on which graphics primitives it needs to intercept. Additional parameters can be specified in each *filter* object. The meaning of the additional parameters will depend upon which *filter* is selected. A *filter* specifying a polygonisation is given as:

```
open  twospheres
      type filter
      location  internal  "soft1"
      valuearg  0  0.25
      pointarg  0  <-3, -3, -3>
      pointarg  1  <6, 6, 6>
      plot  softsphere
```

```
        plot  softsphere
              instance  origin  <1.5, 0, 0>
close
open  softsphere
      type ssphere
close
```

In this example the *twospheres* object will produce a polygonisation of the two scalar sphere components. If the *twospheres* object were subsequently plotted in another object, the same polygonisation would be used each time *twospheres* was plotted. There is no need to re-compute the polygonisation each time it is referenced. Re-using the polygonal mesh created in the polygonisation is more efficient than calling the polygonisation routine each time the filter object is referenced.

The *location* instruction in a *filter* object selects which *filter* to use. The *pointarg* and *valuearg* instructions are used to set parameters of the particular *filter* chosen. The example above selects: isosurface three dimensional polygonisation; a search size of 0.25; specifies the search origin to be <-3, -3, -3>; and specifies the search space size to be <6, 6, 6>.

The *plot* routine discussed previously must be modified to allow the production of SFDL programs. Scalar operators are specified in *group* or *filter* objects, therefore these operators must be handled by the *plot* routine. Wherever in a model scalar operator instructions are encountered, appropriate SFDL instructions must be produced. Consider for example the following extract of JED code which is in a *group* type object:

```
...
plot  softies
      instance ssubtract
...
```

This should produce the following SFDL instructions in the appropriate SFDL program:

```
...
sPUSH  0.0
...  SFDL instructions to plot 'softies'
sSUBTRACT
...
```

Each instance of a scalar operator, other than *addition* and *mirror*, is bracketed at the start by an *sPUSH* instruction and at the end by the appropriate operator. In-between the two instructions will be the appropriate SFDL instructions to create the indicated object, which will range from a single component to a number of components and operators.

The *mirror* scalar operator does not need to be preceded with the *sPUSH* instruction as it is a unary operator. It is simply placed into the SFDL program as it is encountered. Addition is the default scalar operator used in SFDL and therefore does not need to be handled specially.

An alternative method of incorporating isosurface modelling into the graphics system is to allow SFDL programs to be written by the user of the system directly. These programs could then be incorporated into the graphics model. This would force the user of the system to learn another separate method of describing objects, as well as learning a graphics language (SFDL) for their description. SFDL was never intended to be written by the user. The method implemented allows a friendly interface to SFDL as well as full benefit being made of the parametric and hierarchical aspects of JED.

7.4.2 Visualising the isosurface

There are currently two methods of visualising isosurface models in JED. The first is to create a *filter* object and select the isosurface polygonisation routine. The second is the default action taken by JED; if a scalar component is plotted and not intercepted by a *filter* an approximation to the scalar component is used in its place. The second possibility is discussed first.

Consider the first examples produced in section 7.4. Two scalar spheres were plotted but not incorporated into a *filter* object. As a polygonisation routine was not specified, the sphere can either be ignored or handled specially by the graphics system in some manner. The second alternative is used in JED. The default action in JED is to replace the scalar component with a polygonal approximation. The approximation of the spheres would appear in the proper location and be of the proper scale, but would not exhibit any of the blending aspects included in the isosurface modelling techniques. Approximations exists for: *sphere; ellipsoid; cube; torus; cylinder; plane;* and

the *half space* scalar components. The other scalar components are not approximated, they are ignored if not converted by a *filter* object into a polygonal mesh.

The replacement of the scalar components with approximations greatly speeds the display of models. This is useful for establishing rough positions for each of the scalar components. When each of the components is positioned more or less correctly, the isosurface polygonisation method of visualisation is used. Polygonisation is much slower than the approximation method.

Polygonisations are created by producing SFDL programs as discussed in section 7.4.1. The implementation of the *filter* accomplishing isosurface polygonisation is outlined, the filter is implemented as seven steps:

1) Initialise the appropriate routines according to the parameters set in the *filter* object. Arrange for the graphics primitives needed by this *filter* to be intercepted, in the case of isosurface polygonisation this will be all of the scalar components.

2) Create and initialise a structure which will contain the SFDL program. Push this onto a global stack to be used by all routines which create SFDL instructions.

3) Plot the remainder of the object using the *plot* routine as discussed. This will cause the SFDL program to be created with the correct instructions.

4) Add an *sRETURN* instruction to the SFDL program. This instruction halts the evaluation of an SFDL program and returns a value.

5) Call the polygonisation routine supplying the SFDL program created.

6) Store the created polygonal mesh in the objects structure for re-use until a new polygonisation is required.

7) Pop the SFDL program from the global stack and free it. Restore the previous actions to the graphics primitives that were intercepted by this *filter*.

In order to easily allow both the approximation and full polygonisation method of visualisation to be used in JED, it is possible to change a *filter* type

object into a *group*, and vice versa. This will usually have the effect of turning polygonisation on and off. The exception to this rule is when the polygonisations are hierarchically nested. In this case is it necessary to change all *filter* type objects between the main object being displayed and the *filter* in question into *group* objects.

It is possible for JED to produce real time animations immediately using the approximations of the scalar components available. Real time animation of an isosurface model which is polygonised must first be processed and then replayed. A facility for storing complex animation sequences for subsequent real time play back is available in JED.

7.5 Implementation details

JED was written mainly in the 'C' programming language. Parts of the user interface are written in LEX and YACC. LEX is a lexical analyser which is used to break a stream of characters into symbols according to the specified rules. YACC stands for 'yet another compiler compiler' and is used for creating program code from BNF grammars. Both LEX and YACC are part of the UNIX[1] operating system. JED consists of roughly 40,000 lines of code, which is distributed throughout 140 files. JED was developed on a Silicon Graphics IRIS[2] 3130 work station running the UNIX operating system. Silicon Graphics IRIS 3130 work stations have specialised graphics hardware to facilitate real time animation of images composed mainly of line segments.

A window manager, MEX, is distributed with the IRIS work stations which can be used at the user's discretion. MEX was considered unsuitable for this project due to the fact that it limits the graphical modes the screen was able to enter, specifically it is not possible to display 24 bit images when using MEX. A set of routines were created, called JOY, to substitute for MEX. The routines in JOY have different strengths and weakness' than those in MEX, and are by no means a direct substitute. JOY is not able to automatically re-size or move windows. Windows in JOY are of a fixed size and location. In

---

1 UNIX is a trademark of AT&T.
2 IRIS is a trademark of Silicon Graphics Inc.

this sense, JOY is more of a panel manager then a window manager. JOY supports all graphic modes possible on the IRIS, with the inclusion of 24 bit mode. Windows can be drawn automatically regardless of the display mode.

A variety of tools were created within JOY to facilitate the creation of graphical interfaces, the most important of which are: multiple windows; pull down menus; buttons; sliders; and text ports with full editing capabilities. Using these facilities it is possible to quickly create a graphical interface to a variety of programs. Figures 7.5, 7.6, 7.7 and 7.8 present images of the graphical interface to JED. It is possible to: run animation sequences; display models; change camera settings; interactively edit polygons; and control a wide variety of other tasks using this interface to JED.

One of the useful aspects of this interface is the ability to have JED automatically re-display a model when it has been modified. This greatly reduces the time to execute display-modify-display cycles. When a modification is made to the 'active' model it is immediately and automatically re-displayed. This facility is available at the users discretion.

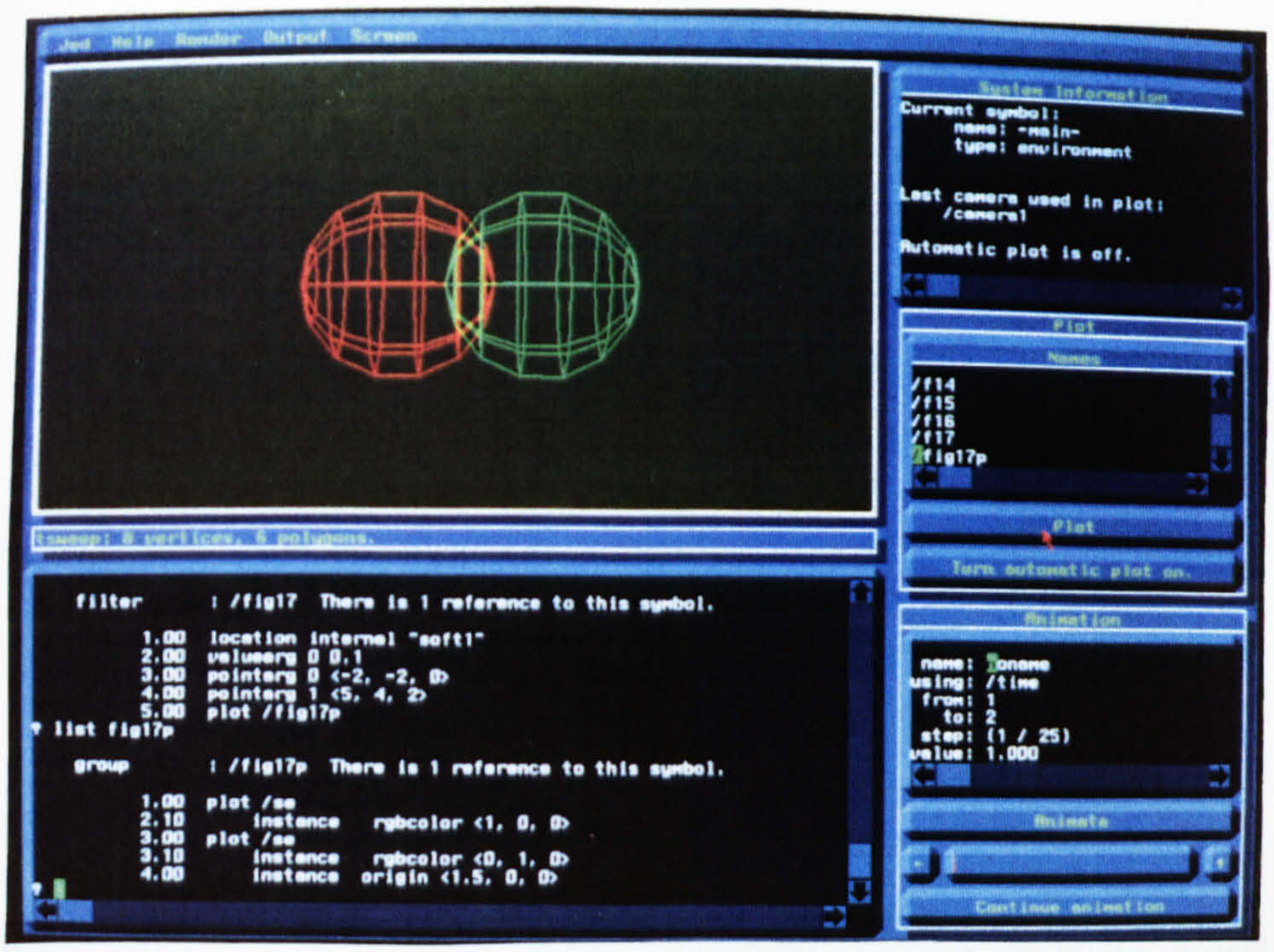


Fig 7.5 JED interface displaying the automatic approximation of the scalar components. In this example, the scalar components are not intercepted by a polygonisation routine.

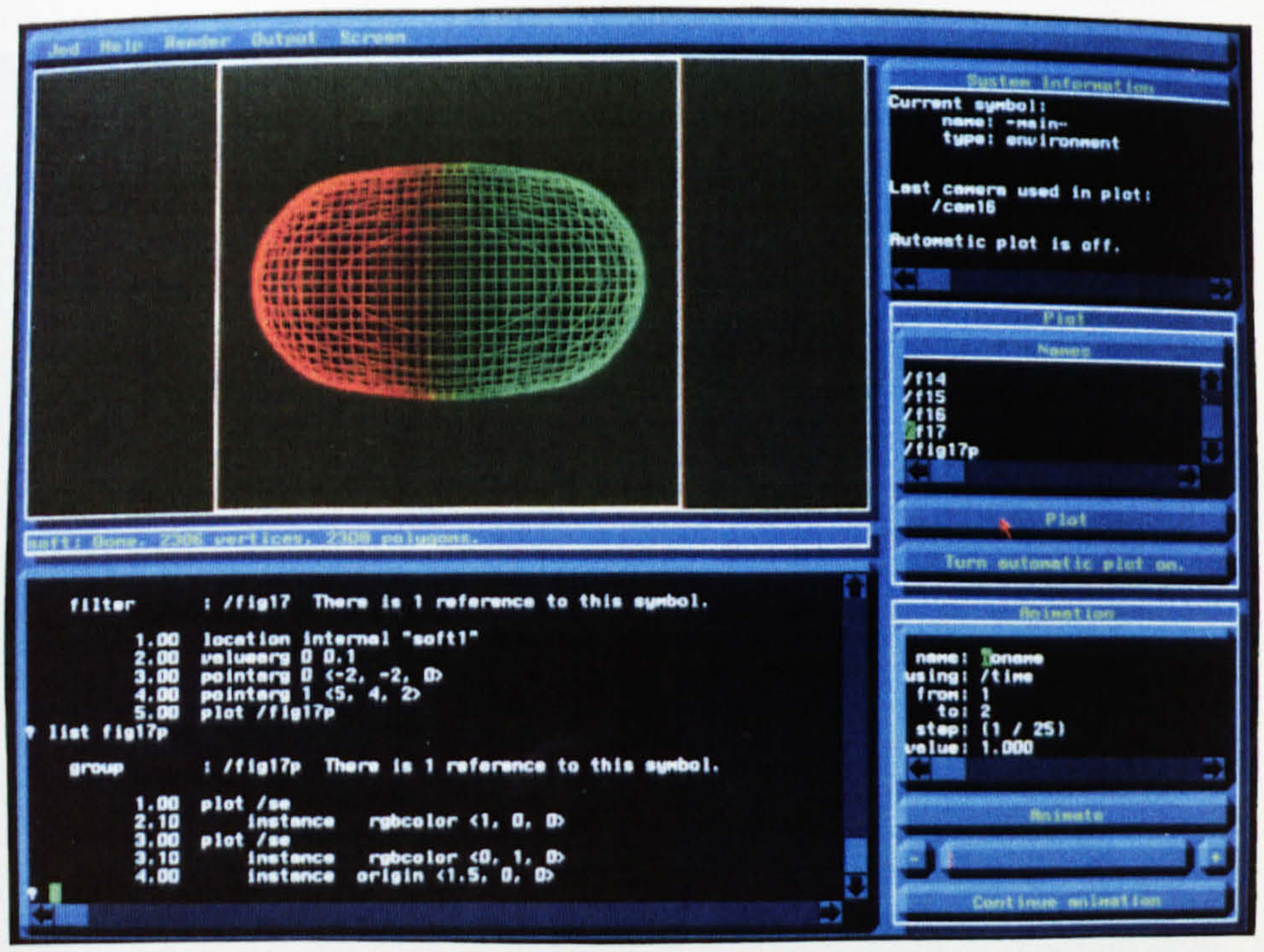


Fig 7.6 JED interface displaying a line drawing of an isosurface model

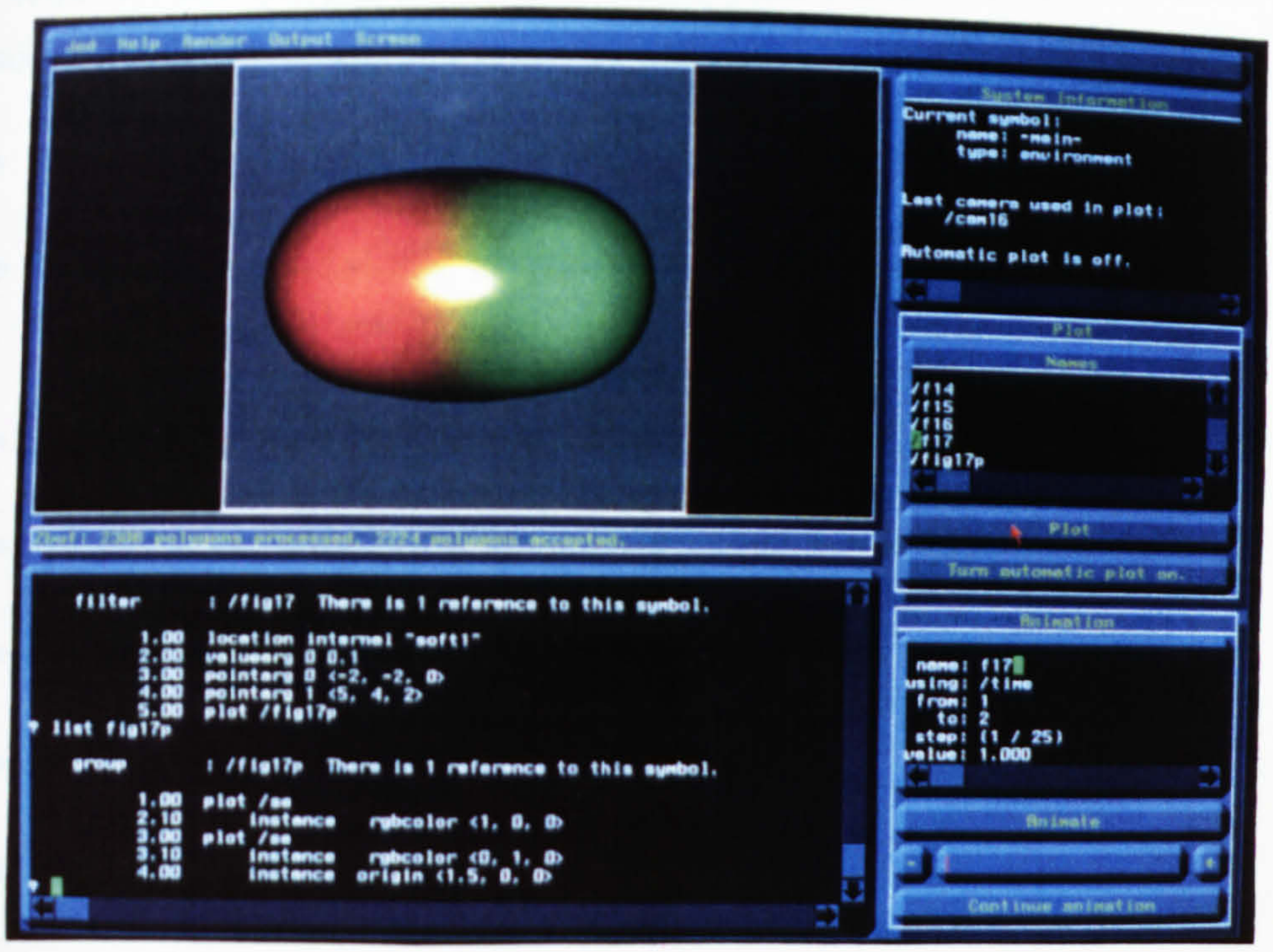


Fig 7.7 JED interface displaying a full rendering of an isosurface model

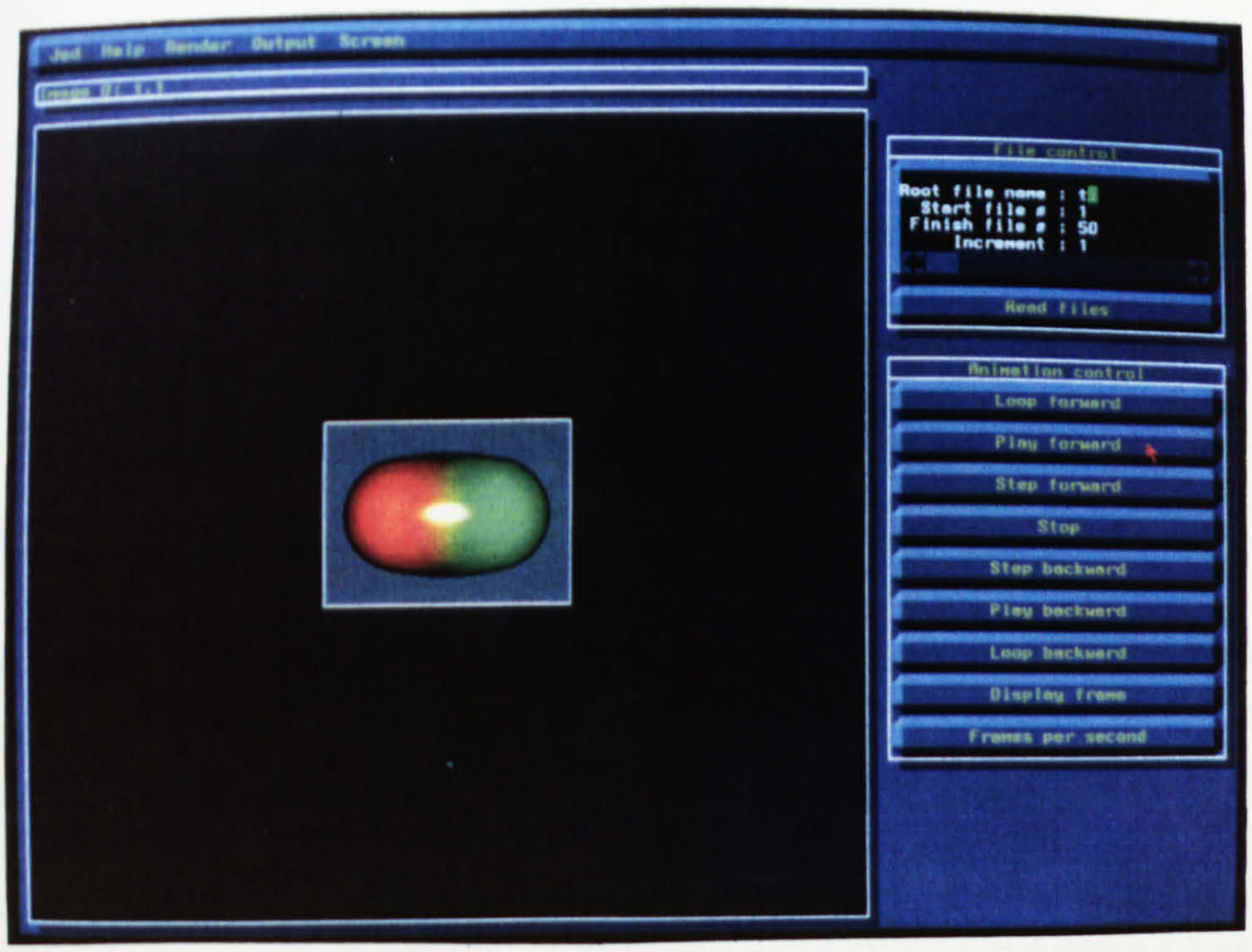


Fig 7.8 JED interface displaying the real time animation playback facility

JED was written during a two and a half year period at The Dorset Institute. Some of the features of JED are:

- Traditional polygonal and polygonal sweep object types.
- Isosurface modelling, allowing a variety of scalar components, scalar operators and attributes.
- Hierarchical recursive data structure.
- Parametric animation.
- Multiple cameras, one active at a time.
- Interactive control of camera.
- Pull down menus.
- Interactive polygon editing.
- On-line help facility.
- Real time animation created on the fly for suitable models, or real time play back of stored sequences of animation (raster or line drawing) for more complex models.
- Multiple light sources. Three types of lights are available: directional; point; or spot light.
- Three dimensional textures. Raster images from paint system or JED are available as environment maps or two dimensional projections. Existence mapping is also available.
- Direct or indirect (through stored image files) output to video.

JED is composed of two major parts, the user interface and the graphics kernel. A variety of alternative user interfaces could have been used to control the graphics kernel.

The graphics kernel consists of a set of data structures and routines for their manipulation or conversion. There are a variety of renderers available for converting representations of three dimensional models into raster images. A conversion is also available for changing the description of an isosurface

model into an SFDL program. A polygonal representation of an isosurface can be obtained from an SFDL program using routines available in JED. The polygonisation routine has been discussed in chapter four.

The graphics kernel is implemented as three separate modules:

1) The *basic* module contains routines to handle: linked lists; stacks; binary trees; polygons; polygonal meshes; a software z-buffer; raster images; a window manager; and an event manager among others.

2) The *soft* module contains routines for the creation of SFDL programs, the specification of surface attributes and polygonisation. This module is dependent upon aspects of the *basic* module.

3) The *kernel* module contains routines for the creation, modification and various conversions of the hierarchical parametric models used in the graphics kernel. This module is dependent upon the previous two modules.

JED is implemented as a single program. The division between each of the separate modules in JED is not visible to the user. JED presents a consistent interface that is suitable for the description of polygonal models, models based on polygonal sweeps and isosurface models. Additional modelling techniques could be added to JED.

JED can be controlled using a separate program in two ways:

1) A program can create JED files which are subsequently interpreted by JED which will perform the appropriate actions.

2) A program can control JED directly using the inter-process communication facilities available in UNIX. A similar technique is used heavily in Graphicsland. In Graphicsland, inter-process communication is used to control independent programs such as the graphics system and the renderers [Wyvill, McPheeters and Garbutt 1986].

Controlling JED through a separate program either directly or through intermediate files, presents a straight forward technique for extending the motion control available in JED. A program to control object dynamics can create an animation sequence using a variety of techniques. This animation

sequence can be translated into appropriate instructions and subsequently interpreted by JED.

7.6 Conclusions

The isosurface modelling techniques have been successfully implemented in a computer graphics modelling and animation system, JED. During the implementation of JED no major problems were encountered with the integration of isosurface modelling and traditional modelling techniques. The major difference between isosurface modelling and traditional modelling is the need to store the scalar field graphics primitives in an intermediate data structure for processing at the appropriate time. In comparison, traditional modelling techniques are composed of independent elements, which can be completely processed as they are encountered. With isosurface modelling, the isosurface is created from the SFDL program, which is in turn created from the hierarchical data structure. The isosurface modelling techniques introduce an extra layer between the graphics system data structures and the resulting surface.

Several techniques were found for the implementation of isosurface modelling, the most general technique was chosen. The possible techniques are:

1) Create only one SFDL program for each model displayed. This is the easiest approach, which however has several disadvantages. One disadvantage is the need to recalculate isosurface models which are replicated, without change, several times in a model.

2) Multiple SFDL programs can be created using the hierarchical data structure and a stack of SFDL programs, the top of which receives SFDL instructions. As a new isosurface model is encountered, an SFDL program structure is pushed onto the stack and is subsequently used to store all SFDL instructions created. As an SFDL program is popped from this stack, a polygonisation is calculated and stored in the appropriate data structure.

3) A generalisation of the above, called *filters*. *Filters* can be used to intercept isosurface primitives, or any other graphic primitives

defined in JED. *Filters* can be used to implement isosurface modelling, as well as a variety of other effects.

In JED, once a polygonisation is calculated it is re-used until an element of the model becomes modified. This has proven useful. For instance multiple views of an isosurface are possible without having to recalculate the isosurface polygonisation for each image.

Isosurface modelling has been incorporated into a graphics system along with traditional modelling techniques. No barrier to the implementation of isosurface modelling has been encountered during the implementation of this graphics system. Isosurface modelling remains a viable solution to a variety of modelling tasks within the computer graphics field.

Although the implementation of isosurface modelling may be more difficult than traditional techniques, the effort is worth it if images of the type presented in this thesis are required. Reproducing the images using traditional techniques would transfer an enormous burden of work onto the designers using the system. Although the implementation of this modelling technique is not as straightforward as some of the traditional techniques, the effort should be repaid when the modelling technique is easily available to the users of the graphics system.

Chapter 8

Further research

8.1 Introduction

Many avenues for future research were discovered during the course of this research project, but of necessity not explored for reasons which will be discussed. The body of research presented in this thesis pertains mainly to the fundamental problems of, and solutions to isosurface modelling. It is possible to augment the techniques presented in this thesis through further research in the areas that are presented in this chapter.

During this research project a decision was made to limit the areas of research to those initially planned, to avoid being side tracked by the lure of the many possibilities and extensions to the techniques that were discovered. While many of the topics for further research present exciting possibilities for the isosurface modelling techniques, their pursuit was considered to be peripheral to the main focus of this research. The main aim of this research is to develop isosurface modelling into a useful and robust modelling technique. In order to achieve this aim, research was needed in five main areas, each of which has been discussed in this thesis. While research into the topics discussed in this chapter will be beneficial to

the isosurface modelling technique, their development was not essential in order to achieve the aims of this research project.

The research presented in this thesis contributes to the development of the isosurface modelling techniques. Contributions range from methods of scalar field description, evaluation and isosurface visualisation through to their incorporation into a complex animation system, controlling both traditional and isosurface modelling techniques. Alternative, related research themes could have been followed with equal legitimacy. For example, in this thesis the treatment of the visualisation of isosurfaces has been brief. The visualisation of isosurfaces is worthy of an independent research effort at doctoral level. Many of the issues involved in visualising isosurface models have been clarified in this research and through this clarification process a method for visualising isosurfaces has been developed. While the method is robust and easily implemented, many useful extensions can be made.

It must be remembered that the areas for future research that are presented in this chapter are just that, areas for *future* research. The problems that require solving, as well as potential extensions and improvements are introduced in sufficient detail to identify an area considered worthy of future research. Potential methods of implementation are proposed in some cases. The suitability of the methods of implementation can not be evaluated until the research has been carried out. The contents of this chapter should be read with some caution, they are by nature somewhat speculative. However, each of the areas mentioned in the following sections was considered to be worthy of further research.

The research topics discussed in this chapter are by no means an exhaustive list. Areas of research not mentioned should not be precluded.

8.2 Specification

There are three main areas for improvement to the specification of scalar fields within the framework of the techniques implemented during this research: enhancement of existing as well as the definition of additional

scalar components; scalar operators; and suitable user interfaces. Each of these will be considered.

Completely different techniques for the specification of isosurface models to those implemented during this research are not considered, although suitable alternatives may be available.

8.2.1 Scalar components

There are currently twelve scalar components implemented in SFDL. The majority of the scalar components allow a number of parameters to be specified. The parameters that are available are: *influence; velocity;* and *bias*. Two extensions are possible: enhancement of the existing scalar components; and the addition of new scalar components.

A parameter which would be useful if implemented is *acceleration*. This would augment the *velocity* parameter already implemented. *Velocity* is used to change the shape of the field of *influence* for a scalar component, in order that it does not influence components in its direction of travel as much as it influences those away from its direction of travel. This is used to stop the 'magnetic jelly' effect discussed earlier. The addition of an *acceleration* parameter could increase the utility of the *velocity* parameter when applied to components which do not follow a straight path. Currently, artefacts may be created when the *velocity* parameter is used in conjunction with a component which does not follow a straight path.

In order to realistically simulate a moving flexible mass, it is necessary to specify the *velocity* and *bias* parameters. It would be desirable to be able to specify a higher level parameter which may be translated into *velocity, bias* and *acceleration* parameters through an appropriate mechanism. Such a parameter may be called *inertia*. Low values of *inertia* would resemble the situation as it presently exists. Higher values of *inertia* would cause the isosurface models to appear much more flexible. An example of an object that exhibits high *inertia* is a plastic bag partially full of water. The bag will not respond to changes in the direction of movement immediately or uniformly, rather, the water would move around inside the bag distorting its shape. A situation with lower *inertia* would be the same bag full of jelly.

Jelly is more rigid than water, which would cause the bag to respond more quickly, although not instantaneously, to changes in its direction.

The second area in which scalar components can be extended is through the inclusion of additional scalar components. Previous research efforts have produced super ellipsoids [Wyvill and Wyvill 1989] and the horned melon [Bloomenthal 1987]. Another useful component would be the generalised cylinder. A spline can be used to describe the centre of the cylinder. The radii of the cylinder can be specified at points along its length and interpolated. Generalised cylinders were described in relation to polygonal models in a paper by Shani and Ballard in 1984. Generalised cylinders implemented in an implicit fashion are demonstrated in a paper by Bloomenthal (1987), they are used to produce n-way branches which are useful in the description of trees.

A rich source of scalar field components exists in the three dimensional texture area. Many of the three dimensional texture maps are suitable for inclusion in the isosurface modelling techniques. A texture map describing fire [Comninos 1989] could be included, making it possible to produce an isosurface model of flames. In the model of fire by Comninos, the flames are easily and realistically animated. A marble texture map, as well as others, could also prove to be rich sources for interesting isosurfaces.

Methods could be found to develop parametric cubic meshes into a form suitable for inclusion in the isosurface modelling techniques. A method of finding the point on a cubic mesh closest to a point in space is required. This method would then be used to generate a scalar value which is dependent upon the distance from a point to the parametric surface. The inclusion of a technique similar to parametric cubic meshes would increase the range of models that could be described using the isosurface technique. A scalar field representation of cubic meshes would also ease the incorporation of some traditional models into the isosurface modelling techniques.

In view of the range and diversity of additional components that can be added to SFDL, it would be desirable to have a procedural interface which would allow the user to specify new components. The pixel stream editor (PSE) created by Perlin in 1985 demonstrates this principle as applied to three dimensional texture mapping.

8.2.2 Scalar operators

As with the scalar components, scalar operators can be extended by either enhancing the existing operators, or by creating new ones.

During the development of the scalar operators, arbitrary decisions were made regarding how the operators treat the attributes of the scalar components. The treatment of colour and other attributes by each scalar operator has been defined in chapter five. Several arbitrary choices were made in the implementation of the scalar operators, dealing with negative colour for example. It may be desirable to allow a flexible means of defining the behaviour of each of the scalar operators. A general method of describing the behaviour of the scalar operators with regards to colour, translucency and the other attributes may lead to interesting results.

The creation of additional scalar operators is possible, one example is an operator named *connect*. The *connect* operator offers a means of connecting two separate isosurface models. *Connect* requires the specification of two scalar models as well as a location and plane in each model. The plane will pass through the location given and can thus be specified with an additional vector, normal to the plane. A point which requires scalar field evaluation will be projected onto both planes. If the point is not in-between the planes, one or the other scalar model will be used in the evaluation of the point. If the point is in-between the planes then the projected point on each of the planes is evaluated and a mixture of these two scalar values is found. If the point is closer to one plane than the other, a higher proportion of the scalar value from the closer plane is used. This mixed value is the final result of the *connect* operator.

An example of the use of *connect* is in connecting a torus to a cube, this would create a model which resembles a torus on one side, a cube on the other and a smooth transition in-between. This effect would be difficult to reproduce using traditional modelling techniques. The *mix* operator is similar to *connect*. The *mix* operator operates in the temporal domain, while the *connect* operator operates in the spatial domain. The effects that are possible using *connect* can not be reproduced using alternative

techniques in SFDL at the moment. The principle of the *connect* operator was first proposed by Rudge 1987.

Alternative methods for implementing the scalar operators can also be accomplished found. The paper presented by Perlin and Hoffert in 1989 demonstrates alternative techniques for implementing: intersection; union; difference; and complement. The technique implemented in their research has the unfortunate effect of changing the shape of the scalar fields surrounding the isosurface. This will affect the manner in which scalar components interact. A possible method of incorporating the techniques proposed by Perlin and Hoffert is to adjust the scalar value after it is combined with another. For instance, intersection is formulated as:

$$A \text{ intersect } B = A(P) * B(P)$$

The adjustment needed can be specified with the following set of constraints:

$$\begin{aligned} \text{Adjust}(0) &= 0 \\ \text{Adjust}(0.25) &= 0.5 \\ \text{Adjust}(1) &= 1 \\ \text{Adjust}(4) &= 2 \end{aligned}$$

Similar adjustments can be found for the other scalar operators.

8.2.3 User interface

Although SFDL is not tied to a particular graphics system, it is currently only implemented in JED. The interface in JED for creating SFDL programs is largely text based, although implemented within a graphical user interface. Isosurface models are created by typing their description into JED using the graphics language supplied. An interactive method of creating isosurface models directly, rather than through an intermediate language, would be useful.

Due to the wide application of the isosurface modelling techniques that is possible, it is foreseen that a number of specialised user interfaces which are each able to describe a certain class of isosurface model will be preferred to

an attempt to create one all encompassing user interface. The user interfaces will be application oriented.

An interface into isosurface modelling was created at the university of Calgary by Jevans in 1988. This program allows the creation of isosurface models composed primarily of ellipses. The scalar operators are limited to addition and subtraction. The interface has been used successfully to create isosurface models which have been used in a number of films produced by the Graphicsland group at the university of Calgary, for example 'The Great Train Rubbery' [Wyvill et al 1988].

The user interface developed by Jevans was not designed to handle specialised tasks such as the visualisation of oil well simulations or medical imaging. Both of these are possible applications of the isosurface modelling technique.

There are many possibilities available for the implementation of user interfaces for the isosurface modelling techniques. The creation of a varied collection of user interfaces may speed the adoption of the isosurface modelling techniques in the graphics community.

8.3 Calculation

There are two main methods available to decrease the amount of time necessary to display an isosurface: decrease the amount of time taken in scalar field evaluation; and investigate alternative methods of visualising the isosurface. The second option will be discussed in section 8.5.

In order to decrease the amount of time taken in evaluating the scalar field, alternative methods of solving an SFDL program need to be found. A promising area for investigation is in the use of spatial extents in the evaluation of the SFDL program.

Currently, SFDL programs use a stack oriented approach for the evaluation of a point in a scalar field. This method is flexible and elegant, however more efficient methods may be found. One of the main areas of inefficiency in the evaluation of an SFDL program is the complete evaluation of the entire program for every point, regardless of its location. A distinction

should be made between the original SFDL program, which describes the complete scalar field, and the method of calculating the scalar value at any point. A method of calculation is constrained by one fundamental rule, it should result in the same scalar value that would be obtained if the original SFDL program were evaluated. A method for calculating a scalar value at any given point incorporating spatial extents is considered.

There are two main ways of determining spatial extents for components and groups of components. First, the spatial extents of each of the components is calculated. This may result in infinite extents for some of the components, such as *noise*. A technique that may be used to further refine the extents is to consider the scalar operators used in the SFDL program. For example, an SFDL program containing an *ellipse* and *noise* component at the first calculation has an infinite spatial extent, as *noise* has no spatial bound. If the intersection operator were being used to combine the two components, a finite extent could be found for the SFDL program. An examination of both the scalar component and scalar operators should be made in the calculation of the spatial extents.

Care must also be made to consider the *mirror* scalar operator, which may cause bounded scalar components to become unbounded.

These spatial extents can be used in determining which scalar components may influence the scalar value of a point. Spatial extents can be used as a basis of increasing the efficiency of scalar field evaluation. The repeated evaluation of an SFDL program with many hundreds of scalar components may be prohibitive without some means of optimising its evaluation.

Spatial extents can be incorporated into an SFDL program evaluation in two ways:

1) SFDL is modified to allow spatial extents to be calculated, these are checked as the program is being evaluated and appropriate measures taken.

2) Many SFDL programs are created from the original, each valid only in a portion of space. This would use a clipping technique with the original SFDL program being duplicated and then 'clipped' to a variety of different volumes.

The first method introduces an overhead to the evaluation of each SFDL evaluation, the comparison of the point to an extent. In the worst case this will increase the time to evaluate the scalar field at a point, the case when all extents are checked and all scalar components are evaluated. The second approach will pre-process the SFDL program into many different SFDL programs. Each newly created SFDL program will be completely evaluated for each point within its volume. The success of the second technique depends upon the method used to divide the volume and create new SFDL programs. A method of space division such as an octree would be suitable. The second method seems to hold more promise and has several advantages over the first. For instance, if for a particular volume of space, clipping the original SFDL program to the volume resulted in an empty SFDL program that entire volume of space could be ignored in subsequent evaluation and visualisation calculations.

The success of the above methods depend upon finding the spatial extents of combinations of scalar components and scalar operators. Research done previously in the constructive solids geometry field [Woodwark 1988] may be applied .

8.4 Visualisation

There were five techniques presented in chapter four that have been used in the past for visualising an isosurface: dot-surfaces; two-dimensional contours; boundary representation; ray tracing; and volumetric rendering. Although the five techniques are a representative sample, there are many variations available for each of their implementations resulting in a wide range of possible visualisation techniques. Depending upon the application, it may be useful to combine two or more techniques into one visualisation system, with the techniques available either separately or in conjunction.

The optimum methods for displaying different types of scalar field information are not obvious. Qualitative research may be valuable in discovering preferences for the display of different types of information, such as temperature, turbulence, stress, and pressure. With the gaining popularity of scientific visualisation, attempts at discovering preferences in the display of scalar field information may have wide application.

The scalar fields in this research are mainly used as a means of obtaining the isosurfaces. The content of the scalar field outside of the regions around the isosurface is of little interest. This suggests that a boundary representation is most suitable for the display of the surfaces created during this research project.

There have been several algorithms for the display of the boundaries of isosurfaces implemented in recent years. Not one of them is sufficiently robust or general to preclude further research in the field.

The research presented in 1989 by Kalra and Barr holds much promise, although additional research needs to be done in obtaining the Lipschitz constant for a wider range of surfaces. The Lipschitz constant is used to determine how finely to search a particular volume of space in order to find the isosurfaces in may contain.

The combination of many techniques can be used in the development of a method of visualising an isosurface. The five methods of visualisation mentioned above can be used in conjunction with techniques such as: adaptive space division; calculation of spatial extents; stochastic sampling; adaptive search for the surface; and a multitude of others.

The development of a more sophisticated method of visualisation than that implemented in this research project would be desirable to ease the further development of this modelling technique. The major short coming of the present technique is considered to be the need to specify the search volume for isosurface models. The search volume may be calculated directly from many of the isosurface models. The specification of a search volume is generally only necessarily for models which are truly unbounded, or for situations where a partial isosurface is desired, otherwise it should be automatically calculated.

The development of additional techniques for the visualisation of isosurfaces is an area with many possibilities for future research.

8.5 Attributes

The shape of an isosurface is controlled through the specification of the scalar field along with the selection of an appropriate isosurface value. The appearance of the isosurface will depend upon the specification of attributes for the scalar field. As implemented in this research, attributes are specified for each scalar component. Each scalar operator has a clearly defined method of handling the combination of these attributes during the evaluation of an SFDL program. There are two techniques for extending the treatment of attributes arising from this description: the techniques for handling the attributes when scalar components are combined can be extended; or the specification of attributes can be accomplished in a different manner than for each scalar component. A third possibility also exists for the extension of the treatment of attributes, and is described in section 8.5.3.

8.5.1 Generalised interaction

Many possibilities for handling attributes were proposed in chapter five, with one method being selected in the implementation of each scalar operator. Using these techniques many varied models can be created. However, occasionally there may be insufficient control over the appearance of an isosurface while the overall shape is acceptable. Alternative possibilities exist for handling surface attributes.

In chapter five, the treatment of two differently coloured spheres which were being added together was discussed. Four different possibilities were proposed for the colour of the resulting isosurface. One of the possibilities was selected and has been implemented. A general interface to the methods of the treatment of attributes by scalar operators or a wider selection of available methods would be valuable. The ability to specify user defined methods would be valuable and may lead to interesting and unexpected results.

8.5.2 Disassociate from surface

The specification of surface attributes is currently closely tied to the description of a scalar field. This precludes some effects which would otherwise be possible. For example, consider two spheres which merge to yield a peanut shell shape. It is possible to specify different colours for each sphere and have the colour of the isosurface blend in-between the two. If the desired effect is to have the colours blend in a different manner than that available, such as orthogonal to the original blend, this would not be possible within the isosurface modelling techniques at the moment. An example of this has been simulated and is presented in figure 8.1.

Tying the specification of surface attributes to the specification of surface shapes can lead to another problem. Consider that for a particular model the shape of a surface has been achieved and the attributes are being specified. If the colours of the model do not follow the same pattern as the distribution of scalar components, additional components will have to be added. This would in turn alter the shape of the surface specified. One possible solution for this problem would be to specify the attributes of an isosurface separately from the shape of an isosurface. A similar mechanism could be used for each specification, an SFDL program can be used to specify the surface shape while a separate SFDL program specifies the attributes, or each attribute. By default one SFDL program is used to specify all information. The technique discussed for extending the specification of attributes is easily implemented within the frame work of the existing research.

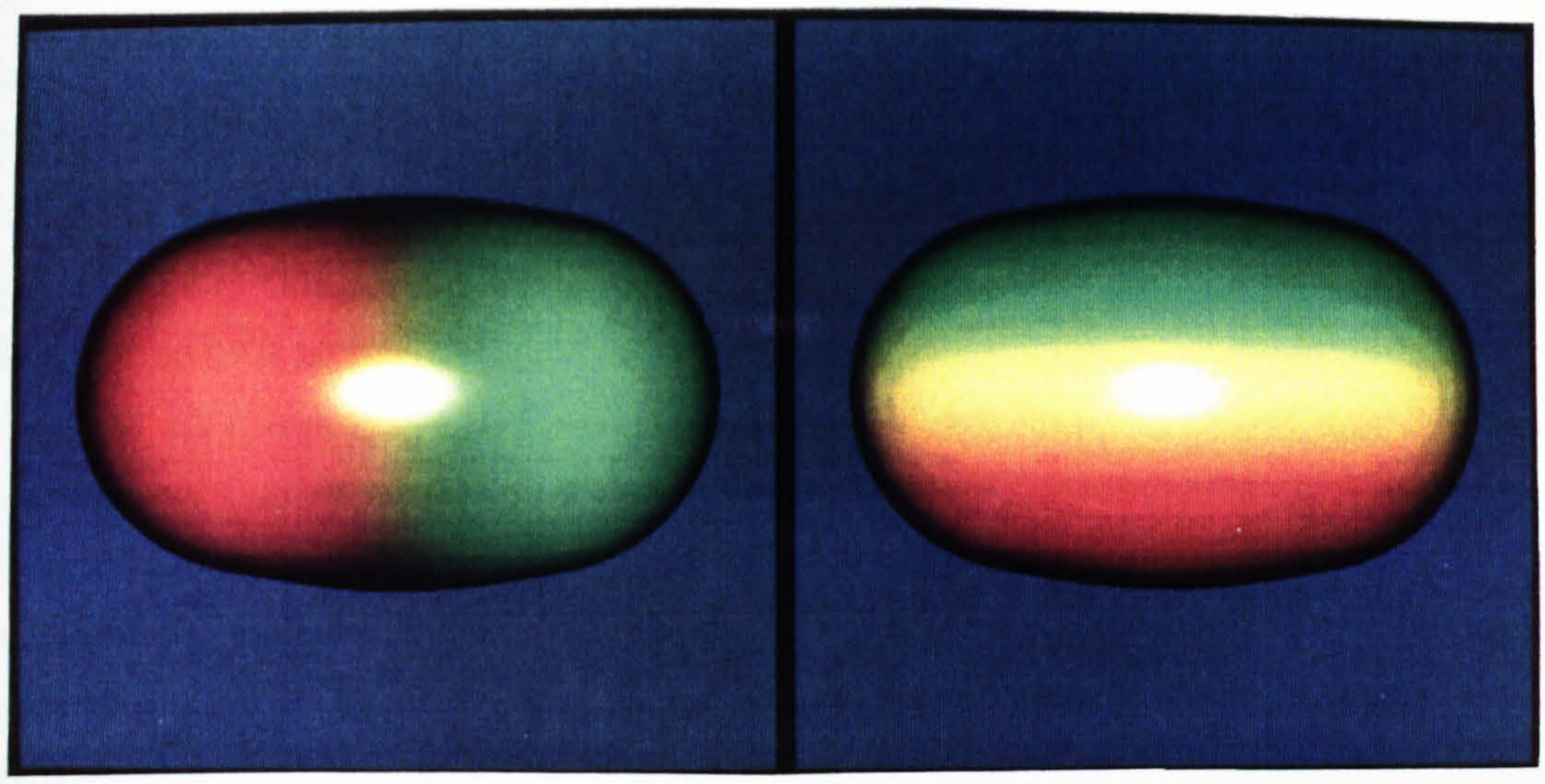


Fig 8.1 The standard, and an extended method of merging colour (simulated)

A second technique would be to allow a similar facility to that available in the pixel stream editor [Perlin 1985]. This second technique may be easily implemented outside of the framework of the isosurface modelling techniques, as a three dimensional texture map. The advantage of evaluating the texture map within the isosurface description is uncertain. When used with a boundary representation, a texture map is best left as a post process. Otherwise an aliasing problem exists as the attributes are ultimately defined only at the vertices of the mesh created. Higher frequencies may exist in the texture than are capable of being represented by specifying attributes at the vertices of a mesh.

8.5.3 Retentive attributes

Consider an example in which a red sphere and a green sphere are separate and then merge to create a yellow surface. The two spheres then move apart again. What colour should the two spheres be when they separate? Several

possibilities exist. As implemented in this research the spheres revert back to their original colours. The possibility proposed in this section is to have the spheres retain the colour of the mixture, yellow.

Retentive attributes can be implemented with varying degrees of accuracy. One possibility is to calculate the colour of the centre of each component during each instance of an animation. The colours in the first image are specified by the user. As an animation progresses the components will change colour according to their surroundings. One drawback to this method of implementation is that to find the colour of a component at animation frame *n*, frame *n - 1* must be known. This would slow the process of finding an arbitrary frame. For example, to find frame ten, the colours in the first nine frames must first be calculated. It is presently possible in JED to find an arbitrary frame in an animation sequence without processing the earlier frames. Due to the problem of finding an arbitrary frame in conjunction with retentive attributes, it is suggested that when implemented, it be possible to 'turn off' the retentive attribute ability. This will speed the design process as correct attributes are not always required. Attributes do not alter the shape of an isosurface.

A closer approximation to retentive attributes would take into consideration situations where an attribute on an isosurface changes, but the centre of the component does not. This may be caused by two components coming close together but not completely merging. Solving this problem will be considerably more difficult.

8.6 Interaction

The interaction of scalar components with other scalar components, as well as the interaction of models constructed from scalar components is currently controlled through the selection of a scalar operator and the specification of parameters such as *influence, velocity* and *bias*. More complex techniques for controlling the interaction of scalar components and isosurface models have been largely ignored previously in this thesis. In this section, a number of methods for extending the isosurface techniques to handle complex forms of interaction are discussed. The two topics chosen for

discussion are: the interaction between components; and the interaction with traditional modelling methods.

8.6.1 Interaction between components

As mentioned in the introduction to this section, the interaction between components is currently controlled through the selection of a scalar operator and specification of parameters. One method for extending this interaction has been discussed in section 8.2.2, extending the scalar operators as well as adding new ones. A scalar operator named *connect* was proposed as an example of a scalar operator that would add new functionality to isosurface modelling.

A problem with the interaction of isosurface models was presented by Wyvill and Wyvill in 1989, a solution is still required. The problem is the interaction of separate components of an isosurface model. The example presented in their paper is a model of a human figure. A body is constructed using the available scalar components. The problem is high lighted as the arms of the body change position. When the arms are extended from the body the correct isosurface is generated. As the arms are lowered to the side of the body there is a possibility that the arm may interact with the torso and merge. It is desirable to describe both the arms and torso in the same model as they are joined at the shoulder. The arm and torso should interact in the shoulder region but not lower down in the body.

Investigation into the solution of this problem should prove an interesting area of further research.

8.6.2 Traditional modelling

The problem of the interaction between isosurface models and traditional models was introduced and briefly discussed in chapter six, section 6.6. The method proposed to handle the interaction between an isosurface model and a traditional model is to manually translate the traditional model into an isosurface representation. This new representation is subtracted from the

original isosurface model. Although this method is straight forward and requires no additional changes to JED or SFDL, it is difficult and laborious to achieve for relatively complex models. A problem was also mentioned where the traditional model was being animated, the isosurface replica must change in the same manner as does the traditional model.

Clearly, an automatic means of translating traditional models into an isosurface representation would be useful. Accomplishing this task will likely require additions to the scalar components available, facilitating the translation into an isosurface representation.

A second problem in the interaction of isosurface models and traditional models is in placing traditional models accurately in close proximity to the isosurface. For example, imagine an isosurface model of a character with traditionally modelled glasses, as was accomplished by Pacific Data Images [Gould 1989]. In order to place the glasses accurately on the model, a location on the isosurface is required. If the correct location is not obtained, the glasses may 'float' away from, or interpenetrate the surface of the character. One method of solving this problem is through trial and error, adjusting the location of the glasses at several key frames until the proper effect is achieved. This was the method chosen by Pacific Data Images. The preferred solution would be to obtain the location of a particular spot on the isosurface automatically.

The location on the isosurface could be specified by a point in space and a vector which represents a ray which is extended to meet the isosurface. The point where the ray meets the surface is the point required in the remainder of the calculation. The situation when a ray does not intersect the isosurface or intersects the surface more than once must be correctly handled.

A method such as the one proposed above for labelling points on the isosurface would ease the integration of isosurface and traditional models, each being used to describe objects for which they are naturally suited.

8.7 Animation

Animation is accomplished in this research by changing the many parameters, scalar operators and scalar components using parametric

animation techniques. There are three parameters supplied to increase the realism of isosurface models as they move: *influence; velocity;* and *bias.* New forms of animation will be enabled as some of the areas discussed in this chapter are implemented. For example, *acceleration, connect* and *inertia.*

An animation technique such as dynamics [Wilhelms 1987] can be implemented outside of the isosurface techniques as suggested in chapter six. Free form deformation may usefully be implemented within the isosurface modelling techniques, as it may create different isosurface topologies than would be possible if implemented as a post process, operating on the polygonal mesh produced for instance.

JED can be extended in a number of ways: incorporating a scripting language; procedural definition of objects; more complex interpolation of values; and other extensions are possible.

Many extensions to the animation of isosurfaces are enabled through extensions to the specification, calculation, visualisation or addition of attributes for isosurfaces. For instance, the metamorphosis animation technique was enabled by the specification of the *mix* scalar operator. As the entire range of techniques for describing isosurface models is enhanced, the range of animation sequences that are possible will be increased.

8.8 Conclusions

It has been demonstrated in this chapter that the isosurface modelling techniques implemented during this research permit many possibilities for extension. Isosurface modelling offers a fertile area for further research, with extensions or improvements available in every aspect of the modelling representation.

Several areas for further research are introduced in this chapter. The study and subsequent implementation of these areas for further research will increase the desirability of the isosurface modelling technique in both general and specific applications.

Chapter 9

Conclusions

This thesis has presented the results of research into the isosurface modelling of objects in computer graphics. The research has been restricted largely to five main areas of isosurface modelling, those of: the scalar field description; isosurface visualisation; specification and calculation of surface attributes; extensions to the animation capabilities; and the incorporation of all of these into a graphics system. Some of the potential areas for further research were also discussed.

The use of isosurfaces defined within scalar fields as the basis of a modelling technique is growing increasingly popular in the computer graphics industry. The application of isosurface modelling to the entertainment field has been accomplished in several systems, for instance Graphicsland [Wyvill, McPheeters and Garbutt 1986] and by the group working on metaballs [Nishimura et al 1985]. Pacific Data Images has used isosurface modelling techniques in a television advertisement [Gould 1989]. A recent article in *Byte* discussing ray-tracing displayed images which contained models which were created using the isosurface modelling techniques [Ransen 1990]. The basic techniques are also being used in many areas of

science and engineering, for instance: medical imaging; molecular modelling; meteorology; and in the visualisation of simulations, such as those produced using computational fluid dynamics.

The isosurface modelling representation, as presented in this thesis, is a viable alternative for the modelling of many objects in computer graphics. Isosurface modelling is most useful when used in conjunction with alternative techniques, each being selected to model objects for which it is suitable.

The objects most suitable for description using the isosurface technique are *soft* objects. *Soft* objects are objects which in some manner change in response to their surroundings. While the isosurface modelling technique may not be suitable for creating a detailed description of a car or a tree, traditional techniques are not suitable for the description of the objects presented in this thesis. Objects suitable for description using the isosurface technique include: water; mud; fire; jelly; bubbles; models generated from a simulation, producing quantities such as pressure, temperature and saturation; empirical data such as that produced in medical imaging; and characters in animation sequences. Many more uses of the isosurface modelling technique are possible; although these are not easily quantifiable. As the modelling technique becomes more widely accepted, new applications will be discovered. The modelling technique clearly has capabilities that are not available in other modelling techniques.

The isosurface modelling technique is more expensive, in terms of computation, than many of the traditional techniques. This will be less of a problem in the future due to two main factors. Firstly, techniques for increasing the efficiency of the existing algorithms are proposed in the thesis. As these and other techniques are implemented the modelling representation will be processed more efficiently on any available computing hardware. Secondly, the computational power of the processors that are currently available is an order of magnitude greater than that of the processor used during this research. This increase in the computational power of processors is likely to continue in the future. The combination of faster processors and increased efficiency of the algorithms will benefit the isosurface modelling technique in three main ways: more detailed renderings of existing models will be possible; the creation of more complex models will be practicable; and the interactivity of the technique will increase as a result of a decrease in the time required to display a model.

One of the most compelling arguments in favour of the implementation of the isosurface modelling representation is the ease with which the images in this thesis were described. Graphical models are easily created using the isosurface modelling technique which would be impractical using traditional techniques. If images similar to those presented in this thesis are required, the isosurface modelling technique is the most suitable method available for their description and subsequent visualisation.

The research presented in this thesis extends the isosurface modelling technique in many areas. These areas are briefly summarised as:

1) The creation of a language (SFDL) for the description of scalar fields using a wide variety of scalar components. Twelve scalar components were discussed as well as six scalar operators for combining them. Techniques for handling the: addition; subtraction; intersection; union; complement; and mixture of scalar components are proposed and demonstrated in this thesis. A parameter which controls the interaction of components has been proposed and implemented in this research, called *influence*. Through the creation of SFDL, a coherent flexible method for describing scalar fields is available. SFDL incorporates the scalar components and operators in a manner which allows the creation of complex isosurface models.

2) The constraints and demands on the isosurface visualisation process are clarified in this research. Through this clarification process the limitations of some of the existing techniques for isosurface visualisation become apparent. A robust, easily implemented and efficient technique is proposed which completely searches a specified volume of space for isosurfaces. Specific isosurface descriptions may allow optimisations to be used to speed this visualisation process. These opportunities are discussed in the thesis.

3) Techniques for the specification and combination of surface attributes including: colour; translucency; texturing; and surface normals are discussed. It became apparent that many possibilities are available for the treatment of surface attributes within the isosurface modelling techniques. These possibilities are discussed and selections made to present a coherent treatment of surface attributes within the modelling technique.

4) Techniques are proposed which increase the realism of isosurface models as they are being animated. Two additional parameters which can be specified to modify the behaviour of components are discussed, *bias* and *velocity*. *Bias* and *Velocity* allow additional control over the interaction of scalar components in an animation sequence.

5) The incorporation of the isosurface modelling techniques into a parametric hierarchical computer animation system is discussed. Techniques for specifying isosurface models in a graphics system, as well as techniques for creating SFDL programs from a hierarchical data structure are presented. The issues involved in visualising isosurfaces as related to graphics system description are discussed and resolved. The incorporation of the isosurface modelling techniques into a graphics system allowing some of the alternative traditional modelling techniques has been accomplished.

Further extensions to the modelling technique are possible in every area discussed. Some of these extensions are presented as topics for further research.

The computer graphics field has grown to become a major industry, with indications that this growth will continue in the future. Part of the reason for the growth is the increased diversity and sophistication of the applications in the computer graphics field. Computer aided design tools have, for instance, grown from a research curiosity to become useful tools replacing in many circumstances the traditional methods of accomplishing a given task.

In many cases the results of research into the modelling and manipulation of objects in computer graphics have become widely accepted. Among the many examples, witness the current popularity of fractals and free form deformations. The computer graphics industry continues to absorb new ideas, increasing the range of objects and motions that can be easily described.

Isosurfaces and scalar fields are representations that will become more widely used for the visualisation of objects and phenomena in computer graphics. This is guaranteed by the results of current research into 'scientific visualisation' in the computer graphics field.

As demonstrated throughout this thesis, when using the isosurface modelling technique is is possible to quickly and easily describe models which would require an enormous effort to reproduce using alternative modelling techniques. Models with: moving fluid surfaces; smoothly curved joins between separate objects; and complex topography are examples of effects that are easily accomplished when using the modelling technique. As the isosurface modelling techniques become accepted throughout the computer graphics industry the creation of models which would have been impractical before isosurface modelling was available will become commonplace.

The research presented in this thesis improves and expands the isosurface modelling techniques in many areas. Isosurface modelling has been developed into a robust and useful modelling technique. This thesis establishes that isosurface modelling of *soft* objects is a powerful and useful technique which has wide application in the computer graphics community.

Appendix 1

Example software

A1.1 Introduction

Examples of 'C' code that can be used to visualise a scalar field in a variety of ways is presented in this appendix. This code has been extracted from the main body of this research and has been modified in order to clarify the main thrust of the process. Extraneous optimisations and complex structures have been replaced by simpler constructs.

While the code presented in this appendix is not identical to the code used in this research, is accomplishes several visualisations and can be modified and extended in a variety of ways. The code is inefficient in a number of respects. Efficiency has been sacrificed for clarity. The code presented is complete in that it is all that is required to accomplish a number of visualisations. However the code is not complete in that a number of routines are not supplied, including: clipping; shading; and allowing a variety of scalar fields to be specified by the user. Knowledge of the 'C' programming language is required. The programs are documented lightly.

The scalar field visualised by the programs presented in this appendix is that which is produced by two sphere components spaced slightly apart and

added together. Extensions can be made through incorporating the improvements discussed in the main body of the thesis.

A1.2 Description of the contents

The contents of the program code is broken into four sections. A 'makefile' is presented firstly, in order to illustrate the method of assembling the associated files into an executable program. There is one 'include' file, eleven utility files and three program files. These are presented briefly:

Makefile This file is used in UNIX to compile the files into executable programs. As all of the files included are not required for each of the three programs, this makefile specifies which are needed for each program. At the moment, it is possible to selectively compile different portions of the 'output.c' file in order to produce PostScript[1] format files or the propriety line format used in this research. This is accomplished through the definition of a variable in this makefile.

appendix.h This file is included in each of the other 'C' programs. It contains common data type definitions, global variables and so on.

eval.c This file contains the code necessary to evaluate the scalar field at a point in space. This file is set up to evaluate to the addition of two sphere components, spaced slightly apart.

hotcode.c The functions contained in this file are used to associate a bit flag with either a cube or a square composed of scalar values. Each of the values is either below the isosurface value or above it. 'Above' includes isosurface values which are 'on' the surface. These functions translate from a cube or square to a bit flag (called a 'hotcode') or vice versa.

intersect.c The function contained in this file is used to calculate an intersection along a line according to the scalar values at the end points and the isosurface value being found. This can be extended by

---

1 PostScript is a registered trademark of Adobe Systems Inc.

incorporating the method discussed in the thesis, using multiple samples and progressively finer approximations.

isocontour.c The functions contained in this file are used to output the isocontour at a certain iso-value. At the moment only a line is output, an easy extension would be to output areas contained 'inside' the isocontour.

isosurface.c The functions contained in this file are used to output the isosurface at a certain iso-value. This file calls the 'poly_cube.c' file to produce the appropriate locations. This files main purpose is to divide the space into cubes and to call appropriate routines to polygonise and output them.

matrix.c The functions contained in this file operate on matrices and points in a variety of ways. Routines to calculate viewing matrices as well as the projection of points by a matrix are included.

output.c The functions contained in this file are used to control the output of a variety of graphical primitives. The format of the output is either in PostScript or a propriety format used during this research. This file can be easily changed to incorporate any alternative formats required.

painters.c The contents of this file are used to produce an elevation display of a two dimensional slice through the scalar field. The elevations on this slice are then drawn back to front, depending upon the current view point. At the moment the polygons output by the 'output.c' file are not shaded. This would be an easy extension.

poly_cube.c This file is used to produce the polygons required to represent the topology of the cube passed to the procedure. This file incorporates the 'Bloomenthal' algorithm discussed in the main body of the thesis.

poly_face.c This file is similar to the above, except for the fact that it deals with squares instead of cubes.

contour.c This file is the main driver used to generate a variety of isocontours of a scalar field. A demonstration of the output produced by this program is given later in the appendix.

elevation.c This file is the main driver used to generate the elevation diagrams of a scalar field. A demonstration of the output produced by this program is given later in the appendix.

surface.c This file is the main driver used to generate a three dimensional isosurface from a scalar field. A demonstration of the output produced by this program is given later in the appendix.

```
#
#       The following are used for outputing in the JOY line drawing format
#
#CFLAGS= -O -s -DOUTPUT_WIRE
#CC= joy_cc
#LIBS= -ljoy -lm

#
#       The following are used for outputing in postscript format
#
CFLAGS= -O -s -DOUTPUT_POSTSCRIPT
CC= cc
LIBS= -lm

surface:        surface.o eval.o output.o intersect.o hotcode.o matrix.o \
                        poly_cube.o isosurface.o
                $(CC) $(CFLAGS) -o surface surface.o eval.o output.o \
                        intersect.o hotcode.o matrix.o poly_cube.o \
                        isosurface.o $(LIBS)

elevation:      elevation.o eval.o output.o isocontour.o intersect.o hotcode.o \
                        poly_face.o painters.o matrix.o
                $(CC) $(CFLAGS) -o elevation elevation.o eval.o output.o \
                        isocontour.o intersect.o hotcode.o poly_face.o painters.o \
                        matrix.o $(LIBS)

contour:        contour.o eval.o output.o isocontour.o intersect.o hotcode.o \
                        poly_face.o
                $(CC) $(CFLAGS) -o contour contour.o eval.o output.o \
                        isocontour.o intersect.o hotcode.o poly_face.o $(LIBS)
```

```
/*
 *  Global definitions and variables
 *
 */

#include <stdio.h>

typedef float real;

typedef struct {
    real        m[16];
} MATRIX;

typedef struct {
    real        v[4];
} POINT4;

typedef struct {
    POINT4      vert[10];
    int         n_vert;
} POLYGON;

/* Define a 2D lattice of scalar field values */

real    **plane;        /* A 2d scalar plane (used in isocontour, painters) */
real    ***field;       /* A 3d scalar field (used in isosurface) */

real    minx, miny, minz,       /* Min X, Y and Z of field */
        maxx, maxy, maxz;       /* Max X, Y and Z of field */
int     divx, divy, divz,       /* Number of divisions in X, Y and Z */
        xres, yres;             /* Resolution of final image */

real    incx, incy, incz;       /* Calculated increment in X, Y and Z */

/* ------------------- */

/* Constants used in the polygonization process */

#define s_POLY          -1
#define s_STOP          -2
#define s_CORNER        0x100

/* ------------------- */

#define PI                          3.14159265358979323846
#define to_radians(d)               ((d) * PI / 180.0)
#define to_degrees(r)               ((r) * 180.0 / PI)
#define blk_copy(t,f,l)             blt ((t), (f), (l))

/* ------------------- */

void    intersect ();
int     *polygonise_face(),
        *polygonise_cube();
```

```
#include "appendix.h"

static real     influence (infl, radius, v)
real            infl, radius, v;
{
    real        a, b, I;

    I = (infl * radius + radius);
    I = (I * I) / (radius * radius);

    a = 1.0 + (I / (1 - I));
    b = -I / (1 - I);

    v = a * v + b;

    return (v);
}

static  real    smooth (v)
real            v;
{
    /* First clamp to range */

    if (v < 0)          v = 0;
    if (v > 2.0)        v = 2.0;

    v = (v*v*v*((14.0/8.0) + v * (-6.0/8.0)));

    return (v);
}

real    eval (x, y, z)
real    x, y, z;
{
     /* The routine has two spheres, added together
      *
      *         first:  radius = 1, influence = 1       loc (0,0,0).
      *         second: radius = 0.5, influence = 1     loc (1.25,0,0).
      */

     real       value1, value2;

     value1 = (x * x + y * y + z * z);
     value2 = ((x-1.25) * (x-1.25) + y * y + z * z) / (0.5 * 0.5);

     value1 = influence (1.0, 1.0, value1);
     value1 = smooth (value1);

     value2 = influence (1.0, 0.5, value2);
     value2 = smooth (value2);

     return (value1 + value2);
}

void    eval_plane ()
{
    int i, j;

    fprintf (stderr, "Evaluating plane...");

    /* Allocate a complete 2D lattice of values */

    plane = (real **) malloc (sizeof (real *) * (divx+1));
    for (i = 0; i <= divx; ++i) {
        plane[i] = (real *) malloc (sizeof (real) * (divy+1));
    }

    /* Evaluate the 2D lattice */

    for (i = 0; i <= divx; ++i) {
```

```
        for (j = 0; j <= divy; ++j) {
            plane[i][j] = eval (minx + (i*incx), miny + (j*incy), 0.0);
        }
    }
}

void    eval_field ()
{
    int i, j, k;

    fprintf (stderr, "Evaluating field...");

    /* Allocate a complete 3D lattice of values */

    field = (real ***) malloc (sizeof (real **) * (divx+1));
    for (i = 0; i <= divx; ++i) {
        field[i] = (real **) malloc (sizeof (real *) * (divy+1));
        for (j = 0; j <= divy; ++j) {
            field[i][j] = (real *) malloc (sizeof (real) * (divz+1));
        }
    }

    /* Evaluate the 3D lattice */

    for (i = 0; i <= divx; ++i) {
        for (j = 0; j <= divy; ++j) {
            for (k = 0; k <= divz; ++k) {
                field[i][j][k] = eval (minx + (i*incx),
                                       miny + (j*incy),
                                       minz + (k*incz));
            }
        }
    }
}
```

```
/*
 *
 *      Routines to set hotcode from topology or set topology
 *      from hotcode.
 *
 */

#include "appendix.h"

int             hotcode_from_cube (cube, isovalue)
real            cube[8], isovalue;
{
    int hotcode;

    hotcode = 0;
    if (cube[0] >= isovalue) hotcode |= 0x01;
    if (cube[1] >= isovalue) hotcode |= 0x02;
    if (cube[2] >= isovalue) hotcode |= 0x04;
    if (cube[3] >= isovalue) hotcode |= 0x08;
    if (cube[4] >= isovalue) hotcode |= 0x10;
    if (cube[5] >= isovalue) hotcode |= 0x20;
    if (cube[6] >= isovalue) hotcode |= 0x40;
    if (cube[7] >= isovalue) hotcode |= 0x80;

    return (hotcode);
}

void            hotcode_to_cube (cube, hotcode, isovalue)
real            cube[8], isovalue;
int             hotcode;
{
    real        plus, minus;

    plus = isovalue + 1;
    minus = isovalue - 1;

    if (hotcode & 0x01) cube[0] = plus; else cube[0] = minus;
    if (hotcode & 0x02) cube[1] = plus; else cube[1] = minus;
    if (hotcode & 0x04) cube[2] = plus; else cube[2] = minus;
    if (hotcode & 0x08) cube[3] = plus; else cube[3] = minus;
    if (hotcode & 0x10) cube[4] = plus; else cube[4] = minus;
    if (hotcode & 0x20) cube[5] = plus; else cube[5] = minus;
    if (hotcode & 0x40) cube[6] = plus; else cube[6] = minus;
    if (hotcode & 0x80) cube[7] = plus; else cube[7] = minus;
}

int             hotcode_from_face (face, isovalue)
real            face[4], isovalue;
{
    int hotcode;

    hotcode = 0;
    if (face[0] >= isovalue) hotcode |= 0x01;
    if (face[1] >= isovalue) hotcode |= 0x02;
    if (face[2] >= isovalue) hotcode |= 0x04;
    if (face[3] >= isovalue) hotcode |= 0x08;

    return (hotcode);
}

void            hotcode_to_face (face, hotcode, isovalue)
real            face[4], isovalue;
int             hotcode;
{
    real        plus, minus;

    plus = isovalue + 1;
    minus = isovalue - 1;

    if (hotcode & 0x01) face[0] = plus; else face[0] = minus;
```

```
    if (hotcode & 0x02) face[1] = plus; else face[1] = minus;
    if (hotcode & 0x04) face[2] = plus; else face[2] = minus;
    if (hotcode & 0x08) face[3] = plus; else face[3] = minus;
}
```

```
#include "appendix.h"

void    intersect (isovalue, x1, y1, z1, v1, x2, y2, z2, v2, nx, ny, nz)
real    isovalue, x1, y1, z1, v1, x2, y2, z2, v2, *nx, *ny, *nz;
{
    real        vx, vy, vz, t;

    /* Linear interpolation */

    vx = x2 - x1;
    vy = y2 - y1;
    vz = z2 - z1;

    t = (isovalue - v1) / (v2 - v1);

    vx = (vx * t) + x1;
    vy = (vy * t) + y1;
    vz = (vz * t) + z1;

    *nx = vx;
    *ny = vy;
    *nz = vz;
}
```

```
/*
 *
 *      The vertices of a square in these routines are numbered as:
 *
 *              3       2
 *                -----
 *                |   |
 *                -----
 *              0       1
 *
 *      An edge is represented as two, two bit values.  Each value is
 *      obtained by shifting or masking.  Each two bit value corresponds
 *      to one of the vertex numbers above.
 *
 */

#include "appendix.h"

/* ------------------------------------------------------------------- */

void    isocontour (isovalue)
real    isovalue;
{
    int         i, j, cur, v1, v2,
                *plist,                 /* Pointer into polygon list */
                drawing_poly;
    real        scalar_values[4],       /* Scalar values of vertices */
                v1x, v1y, v2x, v2y,     /* Coordinates of edges */
                newx, newy, newz;       /* Coordinate of intersection */

    /* For every cube in the 2D lattice, polygonise it */

    for (i = 0; i < divx; ++i) {
        for (j = 0; j < divy; ++j) {

            /* Set the appropriate values for this square */

            scalar_values[0] = plane[i][j];
            scalar_values[1] = plane[i+1][j];
            scalar_values[2] = plane[i+1][j+1];
            scalar_values[3] = plane[i][j+1];

            /* Polygonise it */

            plist = polygonise_face (scalar_values, isovalue);

            /* Interpret list returned from polygonise routine,
             *   the list will contain edges, corners, new polygon markers,
             *   and an end marker.
             *
             * As this program generates only the contours, the list must
             *   be processed to ignore the corners.
             */

            drawing_poly = 0;           /* Set if we're drawing a polygon */

            for (cur = 0; plist[cur] != s_STOP; ++cur) {
                if (plist[cur] == s_POLY) {
                    if (drawing_poly) output_doit ();
                    drawing_poly = 0;
                }
                else if (plist[cur] & s_CORNER) {
                    /* Ignore corners */
                }
                else {
                    /*  This is an edge, so there must be a intersection
                     *    along it.  Find this intersection
                     */

                    /* Set the first vertex locations */
```

```
                v1 = plist[cur] >> 2;
                v1x = minx + (i+((v1==1) || (v1==2)))*incx;
                v1y = miny + (j+(v1>1))*incy;

                /* Set the second vertex locations */
                v2 = plist[cur] & 3;
                v2x = minx + (i+((v2==1) || (v2==2)))*incx;
                v2y = miny + (j+(v2>1))*incy;

                /* Calculate intersection */
                intersect (isovalue,
                           v1x, v1y, 0.0, scalar_values[v1],
                           v2x, v2y, 0.0, scalar_values[v2],
                           &newx, &newy, &newz);

                /* Either move to, or draw to this intersection */
                    newx = xres * ((newx - minx) / (maxx - minx));
                    newy = yres * ((newy - miny) / (maxy - miny));

                    if (drawing_poly) output_draw ((int)newx, (int)newy);
                    else output_move ((int)newx, (int)newy);
                    drawing_poly = 1;
                }
            }
        }
    }
}
```

```
/*
 *      Brute force method of finding an isosurface.
 *        Solve the scalar field for entire region, no optimisations.
 *
 *      This shouldn't be used for fields bigger than about 50x50x50
 */

#include "appendix.h"

void    view_poly();

void    isosurface (vfx, vfy, vfz, vtx, vty, vtz, isovalue)
real    vfx, vfy, vfz, vtx, vty, vtz, isovalue;
{
    int         i, j, k, cur, v1, v2, n_vert,
                *plist;                 /* Pointer into polygon list */
real        cube[8],                    /* Scalar values of vertices */
            v1x, v1y, v1z,              /* Coordinates of edges */
            v2x, v2y, v2z,
            newx, newy, newz;           /* Coordinate of intersection */

MATRIX      look, lens, view;
POLYGON     poly;

fprintf (stderr, "Finding isosurface...");

/* Build the 3D view matrix */

j_mat_lookat (&look, vfx, vfy, vfz, vtx, vty, vtz, 0.0);
j_mat_lens   (&lens, 55.0, 0.5, 50.0, ((real)yres / (real)xres));
j_mat_mult (&look, &lens, &view);

/* For every cube in the 3D lattice, polygonise it */

for (i = 0; i < divx; ++i) {
    for (j = 0; j < divy; ++j) {
        for (k = 0; k < divz; ++k) {

            /* Set the appropriate values for this cube */

            cube[0] = field[i][j][k];
            cube[1] = field[i+1][j][k];
            cube[2] = field[i][j+1][k];
            cube[3] = field[i+1][j+1][k];
            cube[4] = field[i][j][k+1];
            cube[5] = field[i+1][j][k+1];
            cube[6] = field[i][j+1][k+1];
            cube[7] = field[i+1][j+1][k+1];

            /* Polygonise it */

            plist = polygonise_cube (cube, isovalue);

            /* Interpret list returned from polygonise routine,
             *   the list will contain edges, new polygon markers,
             *   and an end marker.
             *
             */

            n_vert = 0;                 /* First edge in poly */

            for (cur = 0; plist[cur] != s_STOP; ++cur) {
                if (plist[cur] == s_POLY) {
                    poly.n_vert = n_vert;

                    view_poly (view, poly);

                    n_vert = 0;
                }
                else {
```

```
/*  This is an edge, so there must be a intersection
 *    .     along it.  Find this intersection
 */

/* Set the first vertex locations */
v1 = plist[cur] >> 3;
v1x = minx + (i+((v1 & 1) != 0))*incx;
v1y = miny + (j+((v1 & 2) != 0))*incy;
v1z = minz + (k+((v1 & 4) != 0))*incz;

/* Set the second vertex locations */
v2 = plist[cur] & 7;
v2x = minx + (i+((v2 & 1) != 0))*incx;
v2y = miny + (j+((v2 & 2) != 0))*incy;
v2z = minz + (k+((v2 & 4) != 0))*incz;

                    /* Calculate intersection */
                    intersect (isovalue,
                        v1x, v1y, v1z, cube[v1],
                        v2x, v2y, v2z, cube[v2],
                        &newx, &newy, &newz);

                    /* Store in polygon list */

                    poly.vert[n_vert].v[0] = newx;
                    poly.vert[n_vert].v[1] = newy;
                    poly.vert[n_vert].v[2] = newz;
                    poly.vert[n_vert].v[3] = 1.0;
                    n_vert++;
                }
            }
        }
    }
}

void    view_poly (view, poly)
MATRIX  view;
POLYGON poly;
{
    int         i;
    POINT4      tpt;

    /* Project poly onto screen (without clipping!) */

    for (i = 0; i < poly.n_vert; ++i) {
        j_mat_mult_point4 (&view, &poly.vert[i], &tpt);

        tpt.v[0] /= tpt.v[3];   /* Divide by W */
        tpt.v[1] /= tpt.v[3];
        tpt.v[2] /= tpt.v[3];

        poly.vert[i].v[0] = (tpt.v[0] + 1.0) * (xres / 2);
        poly.vert[i].v[1] = (tpt.v[1] + 1.0) * (yres / 2);
    }

    output_colour ((int)(tpt.v[2] * 255),
                   (int)(tpt.v[2] * 255),
                         (int)(tpt.v[2] * 255));

    output_poly (poly);
}
```

```
/*
 * Description: Matrix manipulation routines.
 *
 * Notes: Long hand notation is used in the interest of the small increase
 *      in speed.  This, of course, produces larger code.
 *
 *      The matrices are stored in row major order - the same as the
 *      IRIS stores them.
 *
 *      Point multiplications are on the left, so matrix concatenations
 *      should progress from left to right.
 *
 *      Registers are used to get around constructs like:
 *         m->m[0] = a->m[0] * b->m[0];
 *      and replace them with:
 *         m[0] = a[0] * b[0]
 *      they aren't any faster, and in fact assemble (on the IRIS 3000's) to
 *      exactly the same thing.  However the second seems more clear and
 *      maybe on some machines will turn out faster?
 *
 *      Matrix:
 *                0  1  2  3
 *                4  5  6  7
 *                8  9 10 11
 *               12 13 14 15
 *
 */

#include <math.h>
#include "appendix.h"

void            j_mat_id (a_mat)                /* Load identity matrix */
MATRIX          *a_mat;
{
    real        *a;

    a = a_mat->m;

    a[0] = a[5] = a[10] = a[15] = 1.0;
    a[1] = a[2] = a[3] = a[4] = a[6] = a[7] = 0.0;
    a[8] = a[9] = a[11] = a[12] = a[13] = a[14] = 0.0;
}

void            j_mat_copy (a, b)       /* B = A */
MATRIX          *a, *b;
{
    blk_copy (b, a, sizeof(MATRIX));
}

void            j_mat_mult (a_mat, b_mat, c_mat)        /* a * b = c */
MATRIX          *a_mat, *b_mat, *c_mat;
{
    real        *a, *b, *c;

a = a_mat->m;
b = b_mat->m;
c = c_mat->m;

c[ 0] = a[ 0]*b[ 0]+a[ 1]*b[ 4]+a[ 2]*b[ 8]+a[ 3]*b[12];
c[ 1] = a[ 0]*b[ 1]+a[ 1]*b[ 5]+a[ 2]*b[ 9]+a[ 3]*b[13];
c[ 2] = a[ 0]*b[ 2]+a[ 1]*b[ 6]+a[ 2]*b[10]+a[ 3]*b[14];
c[ 3] = a[ 0]*b[ 3]+a[ 1]*b[ 7]+a[ 2]*b[11]+a[ 3]*b[15];

c[ 4] = a[ 4]*b[ 0]+a[ 5]*b[ 4]+a[ 6]*b[ 8]+a[ 7]*b[12];
c[ 5] = a[ 4]*b[ 1]+a[ 5]*b[ 5]+a[ 6]*b[ 9]+a[ 7]*b[13];
c[ 6] = a[ 4]*b[ 2]+a[ 5]*b[ 6]+a[ 6]*b[10]+a[ 7]*b[14];
c[ 7] = a[ 4]*b[ 3]+a[ 5]*b[ 7]+a[ 6]*b[11]+a[ 7]*b[15];

c[ 8] = a[ 8]*b[ 0]+a[ 9]*b[ 4]+a[10]*b[ 8]+a[11]*b[12];
c[ 9] = a[ 8]*b[ 1]+a[ 9]*b[ 5]+a[10]*b[ 9]+a[11]*b[13];
```

```
    c[10] = a[ 8]*b[ 2]+a[ 9]*b[ 6]+a[10]*b[10]+a[11]*b[14];
    c[11] = a[ 8]*b[ 3]+a[ 9]*b[ 7]+a[10]*b[11]+a[11]*b[15];

    c[12] = a[12]*b[ 0]+a[13]*b[ 4]+a[14]*b[ 8]+a[15]*b[12];
    c[13] = a[12]*b[ 1]+a[13]*b[ 5]+a[14]*b[ 9]+a[15]*b[13];
    c[14] = a[12]*b[ 2]+a[13]*b[ 6]+a[14]*b[10]+a[15]*b[14];
    c[15] = a[12]*b[ 3]+a[13]*b[ 7]+a[14]*b[11]+a[15]*b[15];
}

/* p_pt * m_mat = r_pt */

void            j_mat_mult_point4 (m_mat, p_pt, r_pt)
MATRIX          *m_mat;
POINT4          *p_pt, *r_pt;
{
    real        *m, *p, *r;

    m = m_mat->m;
    p = p_pt->v;
    r = r_pt->v;

    r[0] = p[0]*m[ 0] + p[1]*m[ 4] + p[2]*m[ 8] + p[3]*m[12];
    r[1] = p[0]*m[ 1] + p[1]*m[ 5] + p[2]*m[ 9] + p[3]*m[13];
    r[2] = p[0]*m[ 2] + p[1]*m[ 6] + p[2]*m[10] + p[3]*m[14];
    r[3] = p[0]*m[ 3] + p[1]*m[ 7] + p[2]*m[11] + p[3]*m[15];
}

void            j_mat_scale (m, x, y, z)
MATRIX          *m;
real            x, y, z;
{
    j_mat_id (m);
    m->m[ 0] = x;
    m->m[ 5] = y;
    m->m[10] = z;
}

void            j_mat_origin (m, x, y, z)
MATRIX          *m;
real            x, y, z;
{
    j_mat_id (m);
    m->m[12] = x;
    m->m[13] = y;
    m->m[14] = z;
}

void            j_mat_rotx (m, val)
MATRIX          *m;
real            val;
{
    j_mat_id (m);
    m->m[ 5] =  cos(to_radians(val));
    m->m[ 9] = -sin(to_radians(val));
    m->m[ 6] = -m->m[ 9];
    m->m[10] =  m->m[ 5];
}

void            j_mat_roty (m, val)
MATRIX          *m;
real            val;
{
    j_mat_id (m);
    m->m[ 0] =  cos(to_radians(val));
    m->m[ 8] =  sin(to_radians(val));
    m->m[ 2] = -m->m[ 8];
    m->m[10] =  m->m[ 0];
}

void            j_mat_rotz (m, val)
```

```
MATRIX          *m;
real            val;
{
    j_mat_id (m);
    m->m[ 0] =  cos(to_radians(val));
    m->m[ 1] =  sin(to_radians(val));
    m->m[ 4] = -m->m[ 1];
    m->m[ 5] =  m->m[ 0];
}

void    j_mat_lookat (m, vx, vy, vz, px, py, pz, twist)
MATRIX  *m;
real    vx, vy, vz, px, py, pz, twist;
{
    /* The following is taken from the:
    IRIS users guide, vol II, graphics reference,
    Appendix C, Transformation matrices.

    this was done so that there would be consistiency between the
    hardware matrices and the software matrices.
*/

MATRIX      trans, roty, rotx, rotz, tmat;
real        sin_y, cos_y, sin_x, cos_x, t1, t2, tx, ty, tz;

extern real sqrt ();

tx  = (px - vx);
ty  = (py - vy);
tz  = (pz - vz);
tx *= tx;
ty *= ty;
tz *= tz;
t1  = sqrt (tx + tz);
t2  = sqrt (tx + ty + tz);

if (t1 < 0.001) {
    sin_y = 0.0;
    cos_y = 1.0;
}
else {
    sin_y = (px - vx) / t1;
    cos_y = (vz - pz) / t1;
}

if (t2 < 0.001) {
    sin_x = 0.0;
    cos_x = 1.0;
}
else {
    sin_x = (vy - py) / t2;
    cos_x = t1 / t2;
}

j_mat_origin (&trans, -vx, -vy, -vz);

j_mat_id  (&roty);
roty.m[0]  = cos_y;
roty.m[2]  = -sin_y;
roty.m[8]  = sin_y;
roty.m[10] = cos_y;

j_mat_id (&rotx);
rotx.m[5]  = cos_x;
rotx.m[6]  = sin_x;
rotx.m[9]  = -sin_x;
rotx.m[10] = cos_x;

j_mat_rotz (&rotz, -twist);
```

```
    j_mat_mult (&trans, &roty, m);
    j_mat_mult (m, &rotx, &tmat);
    j_mat_mult (&tmat, &rotz, m);
}

void    j_mat_window (m, left, right, bottom, top, near, far)
MATRIX  *m;
real    left, right, bottom, top, near, far;
{
    /* No error checking is done */

    /* The following is taken from the:
        IRIS users guide, vol II, graphics reference,
        Appendix C, Transformation matrices.
        this was done so that there would be consistiency between the
        hardware matrices and the software matrices.
    */

    j_mat_id (m);

    m->m[0]  = (2 * near) / (right - left);
    m->m[5]  = (2 * near) / (top - bottom);
    m->m[8]  = (right + left) / (right - left);
    m->m[9]  = (top + bottom) / (top - bottom);
    m->m[10] = -(far + near) / (far - near);
    m->m[11] = -1.0;
    m->m[14] = -(2 * far * near) / (far - near);
    m->m[15] = 0.0;
}

void    j_mat_lens (m, mm, near, far, aspect)
MATRIX  *m;
real    mm, near, far, aspect;
{
    real        s;

    s = ((0.35 / 2.0) / (mm / 100.0)) * near;

    j_mat_window (m, -s, s, -s * aspect, s * aspect, near, far);
}
```

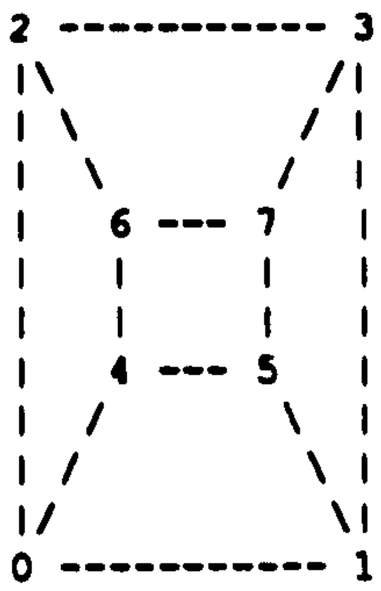
```
#include "appendix.h"

#ifdef OUTPUT_WIRE

typedef struct Wire_format_ {
    FILE        *fp;
    int         (*start)(),
                (*finish)(),
                (*color)(),
                (*move)(),
                (*draw)(),
                (*circle)(),
                (*point)();

    char        *user_data;
} WIRE;

extern  WIRE    *j_wire_write_new(),
                *j_wire_read_new();

static  WIRE    *w;

#endif OUTPUT_WIRE

#define         DPI     300             /* Laser is 300 dots per inch */

void    output_move(),
        output_draw(),
        output_poly(),
        output_doit(),
        output_colour();

void    open_output ()
{
#ifdef OUTPUT_WIRE
    w = j_wire_write_new (0, stdout);
    w->start (w, xres, yres, 0, 0, 50, 50, 50, 1);

#endif
#ifdef OUTPUT_POSTSCRIPT

    printf ("%%!PS-Adobe- \n");
    printf ("%%%%Creator: Isosurface modelling example \n");
    printf ("%%%%For: Fun \n");
    printf ("%%%%Pages: 1 \n");
    printf ("%%%%EndComments \n");
    printf ("%%%%!\n");

    printf ("gsave\n");
    printf ("initmatrix\n");
    printf ("72 %d div dup scale\n", DPI);
    printf ("225 450 translate\n");
    printf ("2 setlinewidth\n");
    printf ("/m { moveto } def\n");
    printf ("/l { lineto } def\n");
    printf ("/s { stroke } def\n");

#endif
}

void    close_output ()
{
#ifdef OUTPUT_WIRE

    w->finish (w);

#endif
#ifdef OUTPUT_POSTSCRIPT
```

```
    printf ("showpage\n");
    printf ("grestore\n");
    printf ("%%%%Trailer \n");

#endif
}

void    output_move (x, y)
int     x, y;
{
#ifdef OUTPUT_WIRE

    w->move (w, x, y);

#endif
#ifdef OUTPUT_POSTSCRIPT

    printf ("%d %d m\n", x, y);

#endif
}

void    output_draw (x, y)
int     x, y;
{
#ifdef OUTPUT_WIRE

    w->draw (w, x, y);

#endif
#ifdef OUTPUT_POSTSCRIPT

    printf ("%d %d l\n", x, y);
#endif
}

void    output_doit ()
{
#ifdef OUTPUT_POSTSCRIPT

    printf ("s\n");

#endif
}

void    output_colour (r, g, b)
int     r, g, b;
{
#ifdef OUTPUT_WIRE

    w->color (w, r, g, b);

#endif
}

void    output_poly (p)
POLYGON p;
{
    int i;

    output_move ((int)p.vert[p.n_vert-1].v[0], (int)p.vert[p.n_vert-1].v[1]);

    for (i = 0; i < p.n_vert; ++i) {
        output_draw ((int)p.vert[i].v[0], (int)p.vert[i].v[1]);
    }

    output_doit ();
}
```

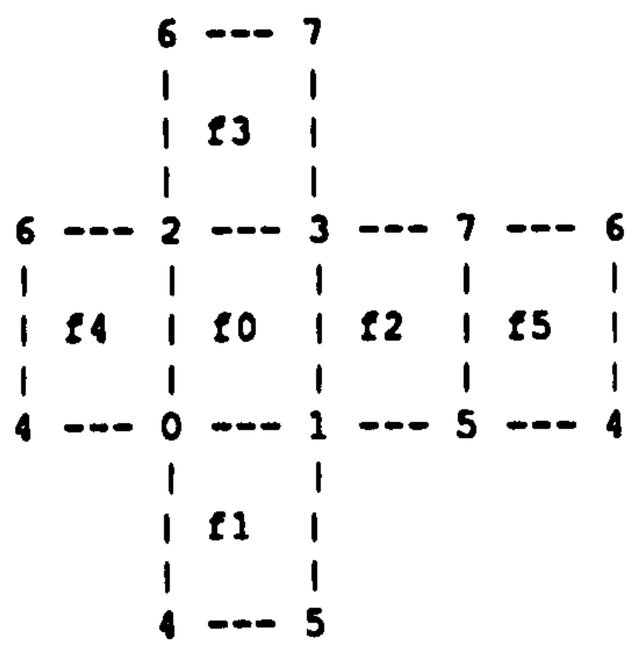
```
#include "appendix.h"

typedef struct {
    int         i, j;
    real        dist;
} ENTRY;

ENTRY entry[10000];

void    painters (vfx, vfy, vfz, vtx, vty, vtz, yscale)
real    vfx, vfy, vfz, vtx, vty, vtz, yscale;
{
    int         cur, i, j;
    MATRIX      look, lens, view;
    POINT4      pt1, pt2, tpt;
    ENTRY       tentry;
    POLYGON     poly;

    j_mat_lookat (&look, vfx, vfy, vfz, vtx, vty, vtz, 0.0);
    j_mat_lens   (&lens, 55.0, 0.5, 1000.0, ((real)xres / (real)yres));
    j_mat_mult (&look, &lens, &view);

    fprintf (stderr, "Sorting plane... ");

    cur = 0;

    for (i = 0; i < divx; ++i) {          /* Create list of average values */
        for (j = 0; j < divy; ++j) {

            pt1.v[0] = minx + (i * incx) + incx/2.0;
            pt1.v[1] = 0.0;
            pt1.v[2] = miny + (j * incy) + incy/2.0;
            pt1.v[3] = 1.0;
            j_mat_mult_point4 (&view, &pt1, &pt2);

            /* Should clip here */

            pt2.v[0] /= pt2.v[3];         /* Divide by W */
            pt2.v[1] /= pt2.v[3];
            pt2.v[2] /= pt2.v[3];

            entry[cur].i = i;             /* Enter this rectangle */
            entry[cur].j = j;
            entry[cur++].dist = pt2.v[2];
        }
    }

    /* Sort this list into descending order */

    for (i = 0; i < cur; ++i) {
        for (j = i+1; j < cur; ++j) {
            if (entry[i].dist < entry[j].dist) {
                tentry = entry[i];
                entry[i] = entry[j];
                entry[j] = tentry;
            }
        }
    }

    /* Draw squares from back to front */

    fprintf (stderr, "Drawing plane... ");

    for (i = 0; i < cur; ++i) {

        poly.vert[0].v[0] = minx + (entry[i].i * incx);
        poly.vert[0].v[1] = plane[entry[i].i][entry[i].j] * yscale;
        poly.vert[0].v[2] = miny + (entry[i].j * incy);
        poly.vert[0].v[3] = 1.0;
```

```
    poly.vert[1].v[0] = minx + ((entry[i].i+1) * incx);
    poly.vert[1].v[1] = plane[entry[i].i+1][entry[i].j] * yscale;
    poly.vert[1].v[2] = miny + (entry[i].j * incy);
    poly.vert[1].v[3] = 1.0;

    poly.vert[2].v[0] = minx + ((entry[i].i+1) * incx);
    poly.vert[2].v[1] = plane[entry[i].i+1][entry[i].j+1] * yscale;
    poly.vert[2].v[2] = miny + ((entry[i].j+1) * incy);
    poly.vert[2].v[3] = 1.0;

    poly.vert[3].v[0] = minx + (entry[i].i * incx);
    poly.vert[3].v[1] = plane[entry[i].i][entry[i].j+1] * yscale;
    poly.vert[3].v[2] = miny + ((entry[i].j+1) * incy);
    poly.vert[3].v[3] = 1.0;

    /* Project poly onto screen (without clipping) */

    for (j = 0; j < 4; ++j) {
        j_mat_mult_point4 (&view, &poly.vert[j], &tpt);

        tpt.v[0] /= tpt.v[3];       /* Divide by W */
        tpt.v[1] /= tpt.v[3];
        tpt.v[2] /= tpt.v[3];

        poly.vert[j].v[0] = (tpt.v[0] + 1.0) * (xres / 2);
        poly.vert[j].v[1] = (tpt.v[1] + 1.0) * (yres / 2);
    }

    poly.n_vert = 4;

    output_colour ((int)((((real)i)/(real)cur) * 255),
                   (int)((((real)i)/(real)cur) * 255),
                   (int)((((real)i)/(real)cur) * 255));

    output_poly (poly);
    }
}
```

```
/*
 *
 * Description: Implementation of Bloomenthal polygonization algorithm, this
 *      is used to make a hash table that is indexed with the 'hotcodes'
 *      to give a list of polygons and edges.
 *
 *      The algorithm is described elsewhere
 *
 */

/*
        In this method of polygonization a cube is used as the unit
        for space division.  The topology of the cube is in tables below.
        The faces and vertices are numbered, as follows.
```



Where the face (0,1,3,2) is in front of the face (4,5,7,6)



If the cube is folded out, the faces are numbered as given to the left

```
*/

#include "appendix.h"

#define SizeEdgeHash    077             /* Each edge: 2 vertices, 3 bits each */
#define SizeCubeHash    0xff            /* 8 vertices, one bit each */
#define NumEdges        (12*2)          /* 12 edges, one in each direction */

static int              intersected[SizeEdgeHash+1];
static int              edge[NumEdges][2];
static int              next_face[SizeEdgeHash+1];
static int              next_vert[SizeEdgeHash+1];
static int              poly_hash_table[SizeCubeHash+1][40];

static int              inited = 0;

static void             build_cube ();
int                     *polygonise_cube ();

/* This data is encoded so that it will come out in the orientation defined
   in relation to the hot side of the surface.  In this case, the hot side
   will 'see' the clockwise side of the polygons.  This is right, as the
```

```
   cold side (outside) is then anti-clockwise */

/* List of faces in clockwise order */

static int face[6][4] = {
    { 0, 2, 3, 1 },
    { 0, 1, 5, 4 },
    { 1, 3, 7, 5 },
    { 3, 2, 6, 7 },
    { 0, 4, 6, 2 },
    { 4, 5, 7, 6 }};

/* Table used to construct next_face, and next_vert.
   each (directed) edge is given in the first two columns, then
     the face to travel to from this edge is given, along with the
     index of the vertex to travel to.

 */

static int              cube_topology[][4] = {

/*   v1 v2  f  v        */

    { 0, 1, 1, 2 },
    { 2, 3, 0, 3 },
    { 4, 5, 5, 2 },
    { 6, 7, 3, 0 },

    { 0, 2, 0, 2 },
    { 1, 3, 2, 2 },
    { 4, 6, 4, 3 },
    { 5, 7, 5, 3 },

    { 0, 4, 4, 2 },
    { 1, 5, 1, 3 },
    { 2, 6, 3, 3 },
    { 3, 7, 2, 3 },

    /* And again, in the opposite directions */

    { 1, 0, 0, 1 },
    { 3, 2, 3, 2 },
    { 5, 4, 1, 0 },
    { 7, 6, 5, 0 },

    { 2, 0, 4, 1 },
    { 3, 1, 0, 0 },
    { 6, 4, 5, 1 },
    { 7, 5, 2, 0 },

    { 4, 0, 1, 1 },
    { 5, 1, 2, 1 },
    { 6, 2, 4, 0 },
    { 7, 3, 3, 1 }};

static          init ()
{
    real        cube[8];
    int         i, hot, cold, e;

    for (i = 0; i <= SizeEdgeHash; ++i) {
        next_face[i] = -1;
        next_vert[i] = -1;
        intersected[i] = 0;
    }

    for (i = 0; i < NumEdges; ++i) {
        hot = cube_topology[i][0];
        cold = cube_topology[i][1];
        e = (hot << 3) | cold;
```

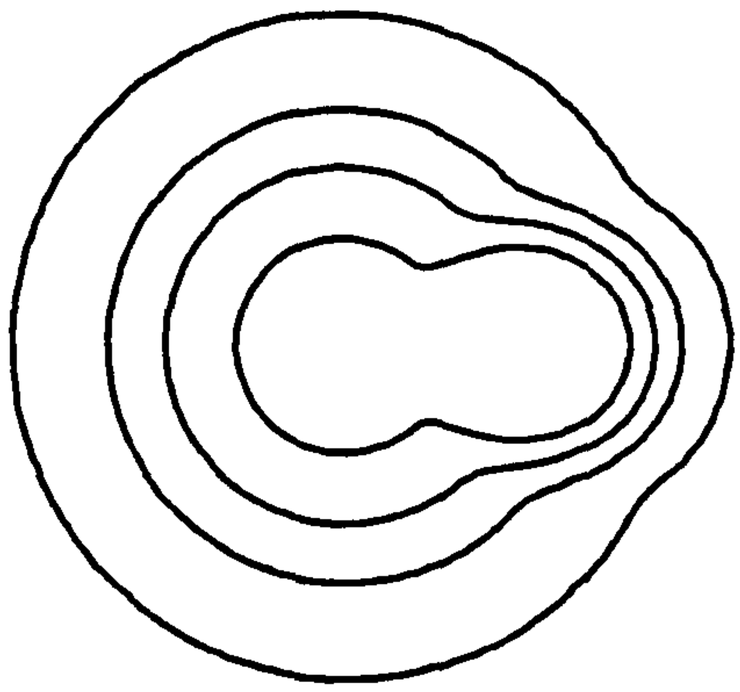
```
        next_face[e] = cube_topology[i][2];
        next_vert[e] = cube_topology[i][3];
        edge[i][0] = hot;
        edge[i][1] = cold;
    }

    /* Initialize polygon tables */

    for (i = 0; i <= SizeCubeHash; ++i) {
        hotcode_to_cube (cube, i, 1.0);
        build_cube (cube, 1.0, &poly_hash_table[i][0]);
    }

    inited = 1;
}

/* Bloomenthal algorithm */

static void     build_cube (cube, isovalue, poly_list)
real            cube[8], isovalue;
int             *poly_list;
{
    int         i, e, v1, v2, start_edge, cur_edge, next, cur, last,
                face_n, face_i, tedge, listlen;

    /* Clear intersection list, No intersections yet */

    for (i = 0; i <= SizeEdgeHash; ++i) intersected[i] = 0;

    /* Find all intersections in cube, using directional edges */

    for (i = 0; i < NumEdges; ++i) {
        v1 = edge[i][0];
        v2 = edge[i][1];
        if (cube[v1] >= isovalue && cube[v2] < isovalue) {
            e = (v1 << 3) | v2;
            intersected[e] = 1;
        }
    }

    listlen = 0;

    /* Apply bloomenthal algorithm */

    for (start_edge = 0; start_edge < SizeEdgeHash; ++start_edge) {
        if (intersected[start_edge]) {          /* Start at any intersection */

            cur_edge = -1;

            cur = start_edge >> 3;                      /* Change direction */
            last = start_edge & 07;

            tedge = (last << 3) | cur;                  /* Where to? */
            face_n = next_face[tedge];
            face_i = next_vert[tedge];

            while (cur_edge != start_edge) {            /* Until poly closed */

                next = face[face_n][face_i];
                cur_edge = (cur << 3) | next;

                if (intersected[cur_edge]) {            /* Intersection */
                    poly_list[listlen++] = cur_edge;        /* Record */
                    intersected[cur_edge] = 0;              /* Processed it */

                    tedge = (next << 3) | cur;              /* Reverse  */
                    face_n = next_face[tedge];              /*  direction */
                    face_i = next_vert[tedge];              /*   & continue */
                }
                else {
```

```
                    cur = next;                         /* Proceed around */
                    face_i = (face_i + 1) & 03;         /*   face         */
                }
            }
            poly_list[listlen++] = s_POLY;      /* End of polygon */
        }
    }
    poly_list[listlen++] = s_STOP;              /* End of list */
}

int             *polygonise_cube (cube, isovalue)
real            cube[8], isovalue;
{
    int         hotcode;

    if (!inited) init ();

    hotcode = hotcode_from_cube (cube, isovalue);

    return (&poly_hash_table[hotcode][0]);
}
```

```
/*
 *
 * Description: Implementation of a polygonisation routine to polygonise
 *      a rectangle.
 *
 *      This routine makes the same choices as the bloomenthal
 *      algorithm regarding connectivity of opposite hot corners.
 *
 */

#include "appendix.h"

#define local static

#define MaxHotCode      0x0F

static int              plist[MaxHotCode+1][10];
static int              inited = 0;

local void              build_face ();
int                     *polygonise_face ();

local           build_plists ()
{
    real        face[4];
    int         i;

    for (i = 0; i <= MaxHotCode; ++i) {
        hotcode_to_face (face, i, 1.0);
        build_face (face, 1.0, &plist[i][0]);
    }
    inited = 1;
}

local void      add_to_list (list, length, item, dir)
int             *list, length, item, dir;
{
    int i, min;

    if (dir == 1) {
        list[length] = item;
    }
    else {
        for (min = 0, i = 0; i < length; ++i) if (list[i] == s_POLY) min = i+1;
        for (i = length - 1; i >= min; --i) list[i+1] = list[i];
        list[min] = item;
    }
}

local void      build_face (face, isovalue, plist)
real            face[4], isovalue;
int             *plist;
{
    int         listlen, visited,
                start, cur, next,
                inc, done_poly;

for (visited = 0, start = 0;
     start < 4 && face[start] < isovalue;
     ++start)
     visited |= (1 << start);

listlen = 0;

while (visited != 0xF) {
    inc = 1;
    cur = start;
    next = (cur + inc) & 03;

    add_to_list (plist, listlen++, s_CORNER | cur, inc);
```

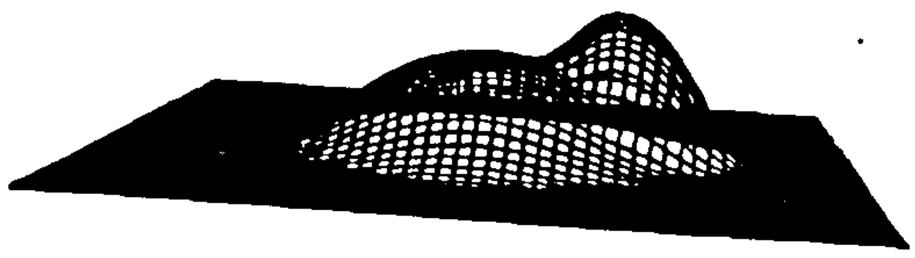
```
        visited |= (1 << cur);

        done_poly = 0;

        while (!done_poly) {
            visited |= (1 << next);
            if (face[next] < isovalue) {
                add_to_list (plist, listlen++, (cur<<2)|next, inc);
                if (inc == -1) done_poly = 1;
                else {
                    inc = -1;
                    cur = start;
                    next = (cur + inc) & 03;
                }
            }
                else {
                    add_to_list (plist, listlen++, s_CORNER | next, inc);
                    cur = next;
                    next = (next + inc) & 03;
                    if (next == start) done_poly = 1;
                }
            }
            add_to_list (plist, listlen++, s_POLY, 1);

            for (++start; start < 4
                        && ((visited & (1<<start))
                          || face[start] < isovalue);
                ++start)
                visited |= (1<<start);
        }
        add_to_list (plist, listlen++, s_STOP, 1);
    }

    int     *polygonise_face (face, isovalue)
    real    face[4], isovalue;
    {
        int         hotcode;

        if (!inited) build_plists ();

        hotcode = hotcode_from_face (face, isovalue);

        return (&plist[hotcode][0]);
    }
```

```
#include "appendix.h"

main(argc, argv)
int     argc;
char    **argv;
{
    if (argc != 9) {
        fprintf (stderr,
            "Usage: %s minx miny maxx maxy divx divy xres yres\n", argv[0]);
        exit (0);
    }

    sscanf (argv[1], "%f", &minx);
    sscanf (argv[2], "%f", &miny);
    sscanf (argv[3], "%f", &maxx);
    sscanf (argv[4], "%f", &maxy);
    sscanf (argv[5], "%d", &divx);
    sscanf (argv[6], "%d", &divy);
    sscanf (argv[7], "%d", &xres);
    sscanf (argv[8], "%d", &yres);

    incx = (maxx - minx) / divx;
    incy = (maxy - miny) / divy;

    eval_plane ();

    /* Polygonise the isocontours */

    open_output ();

    output_colour (0, 0, 0);
    isocontour (0.001);

    output_colour (0, 0, 255);
    isocontour (0.5);

    output_colour (255, 255, 255);
    isocontour (1.0);

    output_colour (255, 0, 0);
    isocontour (1.5);

    fprintf (stderr, "\n");

    close_output ();

    exit (0);
}
```



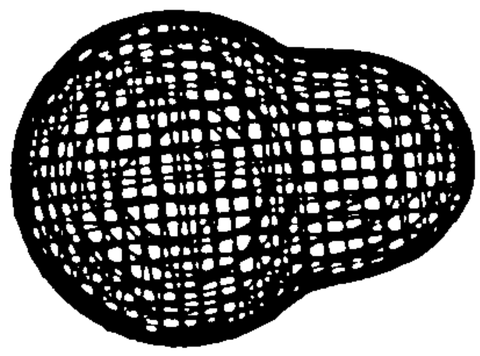
contour -3 -2 3 2 100 100 900 600 > contour.w

```
#include "appendix.h"

main(argc, argv)
int     argc;
char    **argv;
{
    real        vfx, vfy, vfz,
                vtx, vty, vtz,
                yscale;

    if (argc != 13) {
        fprintf (stderr,
            "Usage: %s minx miny maxx maxy divx divy xres yres yscale\n",
                argv[0]);
        fprintf (stderr, "     view-from-x view-from-y view-from-z\n");
        exit (0);
    }

    sscanf (argv[1], "%f", &minx);
    sscanf (argv[2], "%f", &miny);
    sscanf (argv[3], "%f", &maxx);
    sscanf (argv[4], "%f", &maxy);
    sscanf (argv[5], "%d", &divx);
    sscanf (argv[6], "%d", &divy);
    sscanf (argv[7], "%d", &xres);
    sscanf (argv[8], "%d", &yres);
    sscanf (argv[9], "%f", &yscale);
    sscanf (argv[10], "%f", &vfx);
    sscanf (argv[11], "%f", &vfy);
    sscanf (argv[12], "%f", &vfz);

    incx = (maxx - minx) / divx;
    incy = (maxy - miny) / divy;

    vtx = (maxx - minx) / 2 + minx;      /* Look at middle of plane */
    vty = 0.0;
    vtz = (maxy - miny) / 2 + miny;

    eval_plane ();

    /* Polygonise the elevations */

    open_output ();

    painters (vfx, vfy, vfz, vtx, vty, vtz, yscale);

    fprintf (stderr, "\n");

    close_output ();

    exit (0);
}
```



elevation -3 -3 3 3 50 50 900 600 1 1 5 13 > elevation.w

```
#include "appendix.h"

main(argc, argv)
int     argc;
char    **argv;
{
    real        vfx, vfy, vfz,
                vtx, vty, vtz;

    if (argc != 15) {
        fprintf (stderr,
            "Usage: %s minx miny minz maxx maxy maxz divx divy divz \n",
                argv[0]);
        fprintf (stderr,
            "    xres yres view-from-x view-from-y view-from-z\n");
        exit (0);
    }

    sscanf (argv[1], "%f", &minx);
    sscanf (argv[2], "%f", &miny);
    sscanf (argv[3], "%f", &minz);
    sscanf (argv[4], "%f", &maxx);
    sscanf (argv[5], "%f", &maxy);
    sscanf (argv[6], "%f", &maxz);
    sscanf (argv[7], "%d", &divx);
    sscanf (argv[8], "%d", &divy);
    sscanf (argv[9], "%d", &divz);
    sscanf (argv[10], "%d", &xres);
    sscanf (argv[11], "%d", &yres);
    sscanf (argv[12], "%f", &vfx);
    sscanf (argv[13], "%f", &vfy);
    sscanf (argv[14], "%f", &vfz);

    incx = (maxx - minx) / divx;
    incy = (maxy - miny) / divy;
    incz = (maxz - minz) / divz;

    vtx = (maxx - minx) / 2 + minx;      /* Look at middle of volume */
    vty = (maxy - miny) / 2 + miny;
    vtz = (maxz - minz) / 2 + minz;

    eval_field ();

    /* Polygonise the isosurface */

    open_output ();

    isosurface (vfx, vfy, vfz, vtx, vty, vtz, 1.0);

    fprintf (stderr, "\n");

    close_output ();
    exit (0);
}
```



surface -3 -3 -3 3 3 3 50 50 50 900 600 1 1 10 > surface.w

Appendix 2

Bibliography

Amanatides, J.,
'Ray Tracing with Cones,' *Computer Graphics*, Vol.18, No.3, July 1984, pp.129-135.

Aono, M., and T. L. Kunii,
'Botanical Tree Image Generation,' *IEEE Computer Graphics and Applications*, May 1984, pp.10-34.

Atherton, P. R.,
'A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry,' *Computer Graphics*, Vol.17, No.3, July 1983, pp.73-82.

Baker, H. H.,
'Building, Visualising, and Computing on Surfaces of Evolution,' *IEEE Computer Graphics and Applications*, July 1988, pp.31-41.

Baldwin, L.,
'Color Considerations,' *Byte*, September 1984, pp.227-246.

Barnhill, R. E.,
'Surfaces in computer aided geometric design: A survey with new results,' *Computer Aided Geometric Design*, No.2, 1985, pp.1-17.

Barr, A.H.,
'Superquadrics and Angle Preserving Transformations,' *IEEE Computer Graphics and Applications*, January 1981, pp.11-23.

Barsky, B. A.,
'A Description and Evaluation of Various 3-D Models,' *IEEE Computer Graphics and Applications*, January 1984, pp.38-52.

Barsky, B. A., and J. C. Beatty,
'Local Control of Bias and Tension in Beta-Splines,' *ACM Transactions on Graphics*, Vol.2, No.2, April 1983, pp.109-134.

Bergeron, P.,
'A General Version of Crow's Shadow Volumes,' *IEEE Computer Graphics and Applications*, September 1986, pp.17-28.

Blinn, J. F.,
'A Generalization of Algebraic Surface Drawing,' *ACM Transactions on Graphics*, Vol.1, No.3, July 1982, pp.235-256.

Blinn, J. F.,
'Me and My (Fake) Shadow,' *IEEE Computer Graphics and Applications*, January 1988, pp82-86.

Blinn, J. F., and M. E. Newell,
'Texture and Reflection in Computer Generated Images,' *Communications of the ACM*, Vol.19, No.10, October 1976, pp.542-547.

Blinn, J.,
'Simulation of Wrinkled Surfaces,' *Computer Graphics*, Vol.12, No.3, 1978, pp.286-292.

Bloomenthal, J.,
'Boundary Representation of Implicit Surfaces,' *Research Report CSL-87-2, Xerox PARC*, (Draft), October 1987.

Bloomenthal, J.,
'Designing with Implicit Surfaces,' *Research Report, Xerox Parc.*, (Draft).

Bloomenthal, J.,
'Polygonization of Implicit Surfaces,' *Computer Aided Geometric Design*, No.5, 1988, pp.341-355.

Boissonnat, J. D.,
'Surface Reconstruction from Planar Cross-Sections,' *IEEE Proceedings on Computer Vision and Pattern Recognition*, 1985, pp.393-397.

Burger, P., and D. Gillies, *Interactive Computer Graphics*, Addison-Wesley Publishing Company, 1989.

Christiansen, H. N., and T. W. Sederberg,
'Conversion of Complex Contour Line Definitions into Polygonal Element Mosaics,' *Computer Graphics*, August 1978, pp.187-192.

Clark, J.H.,
'Hierarchical Geometric Models for Visible Surface Algorithms,' *Communications of the ACM*, Vol.19, No.10, October 1976, pp.547-554.

Cohen, E.,
'A Method for Plotting Curves Defined by Implicit Equations,' *Computer Graphics*, July 1976, pp.263-265.

Cohen, M. F., and D. P. Greenberg,
'The Hemi-Cube: A Radiosity Solution for Complex Environments,' *Computer Graphics*, July 1985, Vol.19, No.3, pp.31-40.

Comninos, P. P.,
'Computer Animation in Interior and Industrial Design,' *Computers and Graphics*, Vol.9, No.4, 1985, pp.449-453.

Comninos, P. P.,
'Fast bends or fast free-form deformation of polyhedral data,' *Conference proceedings of Computer Graphics '89*, London, November 1989, pp.225-241.

Comninos, P. P.,
'The CGAL Animation Environment and its Application in the Entertainment Industry,' *International Electronic Image Week*, Nice, April 1986, pp.325-332.

Comninos, P.,
Private communication, Dorset Institute, 1989.

Connolly, M. L.,
'Plotting Protein Surfaces,' *Journal of Molecular Graphics*, Vol.4, No.2, June 1986, pp.93-96.

Connolly, M., L.,
'Solvent-Accessible Surfaces of Proteins and Nucleic Acids,' *Science*, August 1983, pp.709-713.

Cook, P. N.,
'A Study of Three-Dimensional Reconstruction Algorithms,' *Automedica*, Vol.4, 1981, pp.3-12.

Cook, R. L., T. Porter and L. Carpenter,
'Distributed Ray Tracing,' *Computer Graphics*, Vol.18, No.3, July 1984, pp.137-145.

Crow, F.C.,
'A More Flexible Image Generation Environment,' *Computer Graphics*, Vol.16, No.3, July 1982, pp.9-18.

de Reffye, P., C. Edelin, J. Françon, M. Jaeger and C. Puech,
'Plant Models Faithful to Botanical Structure and Development,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.151-158.

Demko, S., L. Hodges and B. Naylor,
'Construction of Fractal Objects with Iterated Function Systems,' *Computer Graphics*, Vol.19. No.3, July 1985, pp.271-278.

DeRose, T. D., and B. A. Barsky,
'Geometric Continuity, Shape Parameters, and Geometric Constructions for Catmull-Rom Splines,' *ACM Transactions on Graphics*, Vol.7, No.1, January 1988, pp.1-41.

Doctor, L. J., and J. G. Torborg,
'Display Techniques for Octree-Encoded Objects,' *IEEE Computer Graphics and Applications*, pp.29-38.

Donoho, A. W., D. L. Donoho and M. Gasko,
'MacSpin: Dynamic Graphics on a Desktop Computer,' *IEEE Computer Graphics and Applications*, July 1988, pp.51-58.

Drebin, R. A., L. Carpenter and P. Hanrahan,
'Volume Rendering,' *Computer Graphics*, Vol.22, No.4, July 1988, pp.65-74.

Earnshaw, R. A.,
'The mathematics of computer graphics,' *The Visual Computer*, Vol.3, 1987, pp.115-124.

Fitzgerald, K.,
'Technology in Medicine: Too Much Too Soon?' *IEEE Spectrum*, December 1989, pp.24-29.

Foley, J. D., and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison Wesley Publishing Company, 1982.

Fournier, A., D. Fussel, and L. Carpenter,
'Computer Rendering of Stochastic Models,' *Communications of the ACM*, Vol.25, No.6, June 1982, pp.371-384.

Frenkel, K. A.,
'Volume Rendering,' *Communications of the ACM*, Vol.32, No.4, April 1989, pp.426-435.

Fuchs, H., J. Goldfeather, J. P. Hultquist, S. Spach, J. Austin, F. P. Brooks, Jr., J. Eyles and J. Poulton,
'Fast Spheres, Textures, Transparencies, and Image Enhancements in Pixel-Planes,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.111-120.

Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs and L. Israel,
'Pixel-Planes 5: A Hetrogeneous Multiprocessor Graphis System Using Processor-Enhanced Memories,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.79-88.

Fuchs, H., M. Levoy, and S. M. Pizer,
'Interactive Visualisation of 3D Medical Data,' *IEEE Computer Graphics and Applications*, August 1989, pp.46-51.

Funt, B. V., and E. C. Bryant,
'A Computer Vision System that Analyses CT-Scans of Sawlogs,' *IEEE Proceedings on Computer Vision and Pattern Recognition*, 1985, pp.175-177.

Gallagher, R. S., and J. C. Nagtegaal,
'An Efficient 3-D Visualisation Technique for Finite Element Models and Other Coarse Volumes,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.185-194.

Ganapathy, S., and T. G. Dennehy,
'A New General Triangulation Method for Planar Contours,' *Computer Graphics*, Vol.16, No.3, July 1982, pp.69-75.

Gardner, G. Y.,
'Visual Simulation of Clouds,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.297-303.

Goldwasser, S. M., and R. A. Reynolds,
'Real-Time Display and Manipulation of 3-D Medical Objects: The Voxel Processor Architecture,' *Computer Vision Graphics and Image Procesing*, Vol.39, 1987, pp.1-27.

Gould, R. L.,
'Case Study: Crest Sparkle Singers,' *Character Animation Tutorial*, SIGGRAPH 1989, pp.39-49.

Grimes, J., L. Kohn and R. Bharadhwaj,
'The Intel i860 64-Bit Processor: A General-Purpose CPU with 3D Graphics Capabilities,' *IEEE Computer Graphics and Applications*, July 1989, pp.85-94.

Halbert, A. R., S. J. P. Todd, and J. R. Woodwark,
'Generalizing Active Zones for Set-Theoretic Solid Models,' *IBM UKSC Report 182*, January 1988.

Hall, R.,
'A characterization of illumination models and shading techniques,' *The Visual Computer*, Vol.2, 1986, pp.268-277.

Harada, K., and E. Nakamae,
'Sampling point settings on cubic splines for computer animation,' *The Visual Computer*, Vol.5, 1989, pp.14-21.

Helman, J., and L. Hesselink,
'Representation and Display of Vector Field Topology in Fluid Flow Data Sets,' IEE Computer Graphics and Applications, August 1989, pp.27-36.

Herrmann, L. R.,
'Laplacian-Isoparametric Grid Generation Scheme,' *American Society of Civil Engineers Journal of the Engineering Mechanics Division*, October 1976, pp.749-756.

Ito, H.,
'An Experimental Program for Meta-Balls,' *Pixel*, No.77, 1989, (In Japanese).

Ito, H.,
'Ray tracing for Meta-Balls,' *Pixel*, 1987, pp.76-80, 1989, (In Japanese).

Jevans, D.,
Creep, a program used for creating *soft objects* that is part of the Graphicsland project, Department of Computer Science, University of Calgary, 1988.

Jevans, D., and B. Wyvill,
'Ray Tracing Implicit Surfaces,' *Research Report No. 88/292/04*, The University of Calgary, Department of Computer Science, January 1988.

Kajiya, J. T., and B. P. Von Herzen,
'Ray Tracing Volume Densities,' *Computer Graphics*, Vol.18, No.3, July 1984, pp.165-174.

Kajiya, J. T., and T. L. Kay,
'Rendering Fur with Three Dimensional Textures,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.271-280.

Kalra, D., and A. H. Barr,
'Guaranteed Ray Intersections with Implicit Surfaces,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.297-306.

Kaufman, A., and R. Bakalash,
'Memory and Processing Architecture for 3D Voxel-Based Imagery,' *IEEE Computer Graphics and Applications*, November 1988, pp.10-23.

Keppel, E.,
'Approximating Complex Surfaces by Triangulation of Contour Lines,' *IBM Journal of Research and Development*, January 1975, pp.2-11.

Kernighan, B. W., and D. M. Ritchie,
The C Programming Language, Prentice-Hall Software Series, 1978.

Kesson, M. A.,
'An Investigation into the Modelling of Iso-Surfaces of Scalar Fields,' *Master of Arts thesis*, Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, September 1989.

Koide, A., A. Doi, and K. Kajioka,
'Polyhedral approximation approach to molecular orbital graphics,' *Journal of Molecular Graphics*, Vol.4, No.3, September 1986, pp.149-160.

Lee, M. E., R. A. Redner, and S. P. Uselton,
'Statistically Optimized Sampling for Distributed Ray Tracing,' Computer Graphics, Vol.19, No.3, July 1985, pp.61-78.

Leffler, S. J., E. F. Ostby and W. T. Reeves,
'A Tool-based 3-D Modelling and Animation Workstation,' *EUUG Spring 1988*, London 13-15, pp.29-40.

Levoy, M.,
'Display of Surfaces from Volumetric Data,' *IEEE Computer Graphics and Applications*, May 1988, pp.29-37.

Lewis, J. P.,
'Algorithms for Solid Noise Synthesis,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.263-270.

Long, M. B., K. Lyons, and J. K. Lam,
'Acquisition and Representation of 2D and 3D Data from Turbulent Flows and Flames,' *IEEE Computer Graphics and Applications*, August 1989, pp.39-45.

Lorensen, W. E., and H. E. Cline,
'Marching Cubes: A High Resolution 3D Surface Construction Algorithm,' *Computer Graphics*, Vol.21, No.4, July 1987, pp.163-169.

Mandelbrot, B. B.,
On the Geometry of Homogeneous Turbulence with Stress on the Fractal Dimension of Iso-surfaces of Scalars, *J. of Fluid Mechanics*, Vol.72, 1975, pp.401-416.

Mandelbrot, B. B.,
The Fractral Geometry of Nature, *W. H. Freeman and Company*, 1983.

Max, N. L., and E. D. Getzoff,
'Spherical Harmonic Molecular Surfaces,' *IEEE Computer Graphics and Applications*, July 1988, pp.42-50.

McCormick, B. H., T. A. DeFanti, and M. D. Brown, eds,
'Visualisation in Scientific Computing,' *Computer Graphics*, Vol.21, No.6, November 1987.

McPheeters, C., and B. Wyvill,
'A Tutorial Guide to the ANI Animation System,' *Research Report No. 84/187/45*, The University of Calgary, Department of Computer Science, December 1984.

Middleditch, A. E., and K. H. Sears,
'Blend Surfaces for Set Theoretic Volume Modelling Systems,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.161-170.

Miller, G. S. P.,
'The Motion Dynamics of Snakes and Worms,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.169-178.

Morgan, A. P.,
'A Method for Computing All Solutions to Systems of Polynomial Equations,' *ACM Transactions on Mathematical Software*, Vol.9, No.1, March 1983, pp.1-17.

Mosher, C., and R. Johnson, *IEEE Computer Graphics and Applications*, August 1989, p.17.

Newman, W. M., and R. F. Sproull, *Principles of Interactive Computer Graphics*, McGraw Hill Book Company, 1979.

Nishimura, H., M. Mirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura,
'Object Modeling by Distribution Function and a Method of Image Generation,' *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, Vol.J68-D, No.4, April 1985, pp.718-725 (In Japanese). (Translated into english by Takao Fujiwara while at Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989.)

Nishita, T., and E. Nakamae,
'Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.23-30.

Norton, A.,
'Generation and Display of Geometric Fractals in 3-D,' *Computer Graphics*, Vol.16, No.3, July 1982, pp.61-67.

O'Rourke, J.,
'Polyhedra of Minimal Area as 3D Object Models,' *Proceedings of the 1981 International Conference on Artificial Intelligence*, 1981, pp.664-666.

Papathomas, T. V., J. A. Schiavone and B. Julesz,
'Applications of Computer Graphics to the Visualisation of Metrological Data,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.327-334.

Parke, F. R.,
'A Parameterized Model for Facial Animation,' *IEEE Computer Graphics and Applications*, November 1982.

Pasko, A. A., V. V. Pilyugin, and V. N. Pokrovskiy,
'Geometric Modeling in the Analysis of Trivariate Functions,' *Computers and Graphics*, Vol.12, No.3, 1988, pp.457-465.

Peachey, D. R.,
'Solid Texturing of Complex Surfaces,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.279-286.

Perlin, K.,
'An Image Synthesizer,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.287-296.

Perlin, K., and E. M. Hoffert,
'Hypertexture', *Computer Graphics*, Vol.23, No.3, July 1989, pp.253-262.

Platt, J. C., and A. H. Barr,
'Constraint Methods for Flexible Models,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.279-288.

Prusinkiewicz, P., A. Linkenmayer, and J. Hanan,
'Developmental Models of Herbaceous Plants for Computer Imagery Purposes,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.141-150.

Pueyo, X., and D. Tost,
'A Survey of Computer Animation,' *Computer Graphics Forum*, 1988, pp.281-300.

Purvis, G. D., and C. Culberson,
'On the graphical display of molecular electrostatic force-fields and gradients of the electron density,' *Journal of Molecular Graphics*, Vol.4, No.2, June 1986, pp.88-92.

Quarendon, P.,
'A System for Displaying Three -Dimensional Fields,' *IBM UKSC Report 171*, November 1987.

Quarendon, P.,
'Winsom User's Guide,' *IBM UKSC Report 123*, August 1984.

Quarendon, P., and J. R. Woodwark,
'Three-Dimensional Models for Computer Graphics,' *IBM UKSC Report 158*, May 1987.

Ransen, O. F.,
'The Art of Ray Tracing,' *Byte*, February 1990, pp.238-242.

Reeves, W. T.,
'Particle Systems – A Technique for Modeling a Class of Fuzzy Objects,' *ACM Transactions on Graphics*, Vol.2, No.2, April 1983, pp.91-108.

Requicha, A. A. G., and H. B. Voelcker,
'Solid Modeling: Current Status and Research Directions,' *IEEE Computer Graphis and Applications*, October 1983, pp.25-37.

Reynolds, C. W.,
'Flocks, Herds, and Schools: A Distributed Behavioral Model,' *Computer Graphics*, Vol.21, No.4, July 1987, pp.25-34.

Rosenfield Jr., R. E., S. M. Swanson, E. F. Meyer Jr., H. L. Carrell, and P. Murray-Rust,
'Mapping the atomic environment of functional groups: turning 3D scatter plots into pseudo-density contours,' *Journal of Molecular Graphics*, Vol.2, No.2, June 1984, pp.43-46.

Rossignac, J. R., and A. A. G. Requicha,
'Constant-Radius Blending in Solid Modeling,' *Computers in Mechanical Engineering*, July 1984, pp.65-73.

Rossignac, J. R., and A. A. G. Requicha,
'Depth-Buffering Display Techniques for Constructive Solid Geometry,' *IEEE Computer Graphics and Applications*, September 1986, pp.29-39.

Rudge, A.,
Private communication, Dorset Institute, 1988.

Sabella, P.,
'A Rendering Algorithm for Visualising 3D Scalar Fields,' *Computer Graphics*, Vol.22, No.4, July 1988, pp.51-58.

Samet, H, and R. E. Webber,
'Hierarchical Data Structures and Algorithms for Computer Graphics,' *IEEE Computer Graphics and Applications*, July 1988, pp.59-75.

Samet, H, C. A. Shaffer, and R. E. Webber,
'The Segment Quadtree-Based Representation for Linear Features,' *IEEE Proceedings on Computer Vision and Pattern Recognition '85*, 1985, pp.385-389.

Sato, H., M. Ishii, K. Sato, M. Ikesaka, H. Ishihata, M. Kakimoto, K. Hirota and K. Inoue,
'Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor,' *Computer Graphics*, Vol.19, No.3, July 1985, pp.95-102.

Schwartz, E. L., B. Merker, E. Wolfon and A. Shaw,
'Applications of Computer Graphics and Image Processing to 2D and 3D modeling of the Functional Architecture of Visual Cortex,' *IEEE Computer Graphics and Applications*, July 1988, pp.13-23.

Sederberg, T. W. and S. R. Parry,
'Free-Form Deformation of Solid Geometric Models,' *Computer Graphics*, Vol.20, No.4, August 1986, pp.151-160.

Sederberg, T. W., and A. K. Zundel,
'Scan Line Display of Algebraic Surfaces,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.147-156.

Shani, U., and D. H. Ballard,
'Splines as Embeddings for Generalized Cylinders,' *Computer Vision, Graphics and Image Processing*, Vol.27, No.2, 1984, pp.129-156.

Smith, A. R.,
'Plants, Fractals, and Formal Languages,' *Computer Graphics*, Vol.18, No.3, July 1984, pp.1-10.

Sutherland, I. E., R. F. Sproull, and R. A. Schumacker,
'A Characterization of Ten Hidden-Surface Algorithms,' *Computing Surveys*, Vol.6, No.1, March 1974, pp.293-347.

Tanaka, T., S. Naito, and T. Takahashi,
'Generalized symetry and its application to 3D shape generation,' *The Visual Computer*, Vol.5, 1989, pp.83-94.

Terzopoulos, D., and K. Fleischer,
'Deformable models,' *The Visual Computer*, Vol.4, 1988, pp.306-331.

Terzopoulos, D., and K. Fleischer,
'Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.269-278.

Terzopoulos, D., J. Platt, A. Barr and K. Fleischer,
'Elastically Deformable Models,' *Computer Graphics*, Vol.21, No.4, July 1987, pp.205-214.

Thompson, T.,
'040, Motorola's 68040 Microprocessor,' *Byte*, February 1990, pp.96a-96c.

Tindle, G. L.,
'Fermi Surface Display,' *Computers and Graphics*, Vol.10, No.1, 1986, pp.77-79.

Trousset, Y., and F. Schmitt,
'Active-Ray Tracing for 3D Medical Imaging,' *Eurographics*, 1987, pp.139-150.

Tuy, H. K., and L. T. Tuy,
'Direct 2-D Display of 3-D Objects,' *IEEE Computer Graphics and Applications*, October 1984, pp.29-33.

Upson, C., and M. Keeler,
'VBUFFER: Visible Volume Rendering,' *Computer Graphics*, Vol.22, No.4, July 1988, pp.59-64.

Upson, C., T. Faulhaber Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam,
'The Application Visualisation System: A Computational Environment for Scientific Visualisation,' *IEEE Computer Graphics and Applications*, July 1989, pp.30-42.

Viennot, X.G., G. Eyrolles, N. Janey and D. Arques,
'Combinatorial Analysis of Ramified Patterns and Computer Imagery of Trees,' *Computer Graphics*, Vol.23, No.3, July 1989, pp.31-40.

Von Herzen, B., and A. H. Barr,
'Accurate Triangulations of Deformed, Intersecting Surfaces,' *Computer Graphics*, Vol.21, No.4, July 1987, pp.103-110.

Ward, G. J., F. M. Rubinstein, and R. D. Clear,
'A Ray Tracing Solution for Diffuse Interreflection,' *Computer Graphics*, Vol.22, No.4, August 1988, pp.85-92.

Waters, K.,
'A Muscle Model for Animating Three-Dimensional Facial Expression,' *Computer Graphics*, Vol.21, No.4, July 1987, pp.17-24.

Wilde, C.,
Linear Algebra, Addison-Wesley Publishing Company, 1988.

Wilhelms, J.,
'Toward Automatic Motion Control,' *IEEE Computer Graphics and Applications*, April 1987, pp.11-22.

Wilhelmson, R., and C. Upson, *IEEE Computer Graphics and Applications*, August 1989, pp.19.

Wolfe, R. H. Jr., and C. N. Liu,
'Interactive Visualisation of 3D Seismic Data: A Volumetric Method,' *IEEE Computer Graphics and Applications*, July 1988, pp.24-30

Woodwark, J. R.,
'Blends in Geometric Modelling,' *IBM UKSC Report 190*, April 1988.

Woodwark, J. R.,
'Eliminating Redundant Primitives from Set-Theoretic Solid Models by a Consideration of Constituents,' *IBM UKSC Report 188*, April 1988.

Wright, T., and J. Humbrecht,
'ISOSRF – An Algorithm for Plotting Iso-Valued Surfaces of a Function of Three Variables,' *Computer Graphics*, Vol.13, No.4, 1979, pp.182-189.

Wyvill, B., and G. Wyvill,
'Field Functions for Implicit Surfaces,' *The Visual Computer*, Vol.5, 1989, pp.75-82.

Wyvill, B., B. Liblong, and N. Hutchinson,
'Using Recursion to Describe Polygonal Surfaces,' *Proceedings of Graphics Interface '84*, Ottawa, Canada, June 1984.

Wyvill, B., C. McPheeters, and G. Wyvill,
'Animating *Soft Objects*,' *The Visual Computer*, Vol.2, 1986, pp.235-242.

Wyvill, B., C. McPheeters, and R. Garbutt,
'University of Calgary 3-D Computer Animation System,' *SMPTE Journal*, Vol.95, no.6, June 1986, pp.629-636.

Wyvill, B., et al,
'The Great Train Rubbery,' A short film, Department of Computer Science, University of Calgary, 1988.

Wyvill, B.,
Navigating the Animation Jungle, Draft, Department of Computer Science, University of Calgary, 1988.

Wyvill, G., B. Wyvill, and C. McPheeters,
'Solid Texturing of *Soft Objects*,' *IEEE Computer Graphics and Applications*, December 1987, pp.20-26.

Wyvill, G., C. McPheeters, and B. Wyvill,
'Data Structure for *Soft Objects*,' *The Visual Computer*, No.2, 1986, pp.227-234.

Xie, S., and T. W. Calvert,
'The CSG-EESI Scheme for Representing Solids with a Conversion Expert System,' *IEEE Proceedings on Computer Vision and Pattern Recognition,* 1985, pp.124-129.

Yerry, M. A., and M. S. Shephard,
'A Modified Quadtree Approach to Finite Element Mesh Generation,' *IEEE Computer Graphics and Applications,* Vol.3, No.1, 1983, pp.39-46.

Zucker, S. W.,
'Survey: Region Growing, Childhood and Adolescence,' *Computer Graphics and Image Processing,* Vol.5, 1976, pp.382-399.

Zyda, M. J., A. R. Jones, and P. G. Hogan,
'Surface Construction from Planar Contours,' *Computers and Graphics,* Vol.11, No.4, 1987, pp.393-408.

Zyda, M. J., and R. A. Walker,
'Design notes on a Single Board Multiprocessor for Real-Time Contour Surface Display Generation,' *Computers and Graphics,* Vol.12, No.1, 1988, pp.91-97.