THE VISUALISATION OF REGULAR THREE DIMENSIONAL DATA

by

Mark W. Jones BSc. (Wales)

A thesis submitted to the University of Wales
in candidature for the degree of Philosophiæ Doctor

Department of Computer Science
University of Wales, Swansea

10 July 1995

Declaration

I declare that this work has not already been accepted in substance for any degree, and is not being concurrently submitted in candidature for any degree.

.....

(Candidate)

Statement 1

The work submitted herein for the degree of Philosophiæ Doctor has been carried out by Mark W. Jones under the supervision of Dr. Min Chen, Department of Computer Science, University of Wales, Swansea.

.....

(Candidate)

Statement 2

I hereby give consent for my thesis to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

.....

(Candidate)

.....

Date

Summary

This work is a thorough investigation of the area of visualisation of regular three dimensional data. The main contributions are new methods for:

- reconstructing surfaces from contour data;
- constructing voxel data from triangular meshes;
- real-time manipulation through the use of cut planes;
- ultra high quality and accurate rendering.

Various other work is presented which reduces the amount of calculations required during volume rendering, reduces the number of cubes that need to be considered during surface tiling and the combined application of particle systems and blobby models with high quality, computationally efficient rendering.

All these methods have offered new solutions and improved existing methods for the construction, manipulation and visualisation of volume data.

In addition to these new methods this work acts as a review and guide of current state of the art research, and gives in depth details of implementations and results of well known methods. Comparisons are made using these results of both computational expense and image quality, and these serve as a basis for the consideration of what visualisation technique to use for the resources available and the presentation of the data required. Reviews of each main visualisation topic are presented, in particular the review of volume rendering methods covers much of the recent research. Complementing this is the comparison of many alternate viewing models and efficiency tricks in the most thorough investigation to this researcher's knowledge. During the course of this research many existing methods have been implemented efficiently, in particular the surface tiling technique, and a method for measuring the distance between a point and a 3D triangle.

Parts of this research have been presented by the candidate at Eurographics 1994 (Oslo), the fifth Eurographics Workshop for Visualisation in Scientific Computing 1994 (Rostock), and the 13th UK Eurographics Conference 1995 (Loughborough).

Contents

Chapter 1

Introduction

The title of this work is *The Visualisation of Regular Three Dimensional Data*, and to understand the aims and scope of this research it is necessary to understand the exact meaning and context of the title.

The definition of visualisation from the Oxford English dictionary is *"to form a mental picture of"*. In the context of this research it means *"to form a picture of"*. The word mental is dropped because the person is removed from the process. The picture, henceforth image, is formed as the result of some process, and is available regardless of any human involvement. The image, when displayed on some device, is available to all people present, and allows no misinterpretation. The image is exactly the same to all that see it, and is not dependent upon the skills of the observer to understand the underlying object, and to be able to effectively project an image in their own brain. With the computer to perform this projection, the viewer has become passive rather than active, and by definition less effort is required by the user to visualise the object. In essence visualisation in this context means the *"automation of image projection of objects in order to achieve an effortless representation of the objects which is constant under all circumstances, and is independent of the viewers skills"*. This research is concerned with the processes that can be used to project the object which in this case is the regular three dimensional data of the title.

Regular has several definitions, but the most relevant is *"recurring constantly at a fixed interval"*. The data which this research is concerned with is that in which the values occur at fixed intervals, with constant spacing between the values in a dimension. The data can in fact be any measured or computed value and examples include object density, temperature, probability and pressure. This research is concerned with data that is three dimensional by nature. This simply means that the data is a collection of values over a finite space in each of three dimensions at a fixed point in time. The data is best understood as a regular lattice and is referred to as *volume data*.

The current research can be broadly divided into four categories. The first category is the aggregation of different methods to produce one system that allows volume data to be fully explored at varying levels of interactivity and quality. This involves the algorithmic issues of combining differing methods, and the user interface issues of how to make the system available to non expert users and to give them full intuitive control over their data. The second area is

the continual development of current methods to increase efficiency and image quality. This is achieved by exploiting coherence in either the data or the image, or by reducing algorithm costs by simplifying the algorithm at the expense of accuracy. The third area is that of creating artificial data from three dimensional objects (otherwise known as voxelisation) that will enable the use of common volume manipulation and visualisation techniques to take place on the data. The methods seek to retain the accuracy of the original representation. The final category is the search for an all encompassing method for the production of surfaces from arbitrary contours. This thesis concentrates on the latter three categories, and with the application of appropriate user interfaces could be combined with work being carried out within the first category.

The problems faced by researchers are those of image quality and speed of production. For the best overall visualisation of volume data, the light transmittance of every point in the volume should be defined, and an image of the light absorption of the data can be created. The discrete approximation of this is known as volume rendering, and due to the complex calculations of light absorption and shading, images produced using this method are very expensive to compute. When the volume data describes a surface, it is best if just that surface is visible during the visualisation of the data. Methods are available that project the data with varying degrees of quality and efficiency. Alternatively an intermediate surface representation can be computed which can be used to display the surface. In both cases the image quality depends upon the data, and even for large data sets displayed as small images, the underlying structure of the data is still visible. In the category of creating surfaces from contours, there is no general method for stitching together arbitrary contours in order to produce automatic visualisation of the data quickly. The speed of computation of images using any of the above techniques means that without the aid of sophisticated hardware, real-time manipulation is out of the question.

In summary the field of volume visualisation is flawed in several ways. Firstly there is no ultra high quality visualisation technique. Secondly there is no high quality, efficient method for voxelisation of general objects, and thirdly there is no general method for reconstruction of surfaces from contours.

The aim of this work is to be a thorough investigation of the methods available for the visualisation of regular three dimensional data and the contribution of methods that improve the image quality and computational aspects of visualisation. The course of this work should include implementations, comparisons and evaluations of current methods. Where possible these methods should be extended, either to improve the quality of the visualisation or the speed of computation. New solutions to the above problems would be most desirable, and if discovered, they should be fully investigated and presented appropriately. In short this work should offer a complete view of the current state of the art in volume data visualisation.

In Chapter 2, the processes known as surface tilers are examined. In the surface tiling algorithm the volume data is displayed as a 3D surface. The second chapter can be regarded as an in-depth review and has a comparison of surface tiling techniques, with implementation details. Some new work – *indexing of volume data*, which reduces the amount of computation required to create the surface, is presented in Section 2.5, and a comparison with the other methods show it to be roughly computationally equivalent with the most efficient existing method.

Chapter 3 reviews the broad category of volume rendering, which can be regarded as the true 3D visualisation technique for 3D data, since there is the potential for each data value along the ray passing through each point on the image to contribute to the colour and intensity of that point. The process of volume rendering is described in detail, and several *viewing models* are implemented, compared and evaluated. The volume rendering work is extended in Section 3.4, and this extension is compared with the existing techniques, and shown to be significantly faster with very little image degradation. To this researcher's knowledge there has not been such a thorough comparison, of both the image quality and computational costs of these methods.

The final category of volume data visualisation methods which is presented in Chapter 4 is that of displaying those data values themselves that conform to some criterion. The criterion is usually those data values that are greater than some value (threshold value) in which we are interested. A review of existing methods for the visualisation of data is made, and the more popular methods are implemented and compared. A method called the direct surface rendering method is presented, and shown to produce images of the data that are of a much higher quality. A new method for real-time cutting operations on the data using this method [1] is also presented. This work was presented at the Eurographics workshop for Visualisation in Scientific Computing 1994 (Rostock, Germany). It also appeared in "Visualisation in Scientific Computing", pp 1–8, edited by M. Göbel, H. Müller and B. Urban.

Together these chapters show the available methods for the visualisation of 3D volume data, and throughout well known example data sets are used such as the AVS Hydrogen data set, and the University of North Carolina data sets of CThead, MRbrain and superoxide dismutase. These data sets have either been computed or scanned using some device. Chapter 5 looks at some alternative ways of constructing and visualising volume data. Firstly the production of volume data which represents soft objects is investigated with the application of the direct surface rendering technique in order to visualise the objects. The use of particle systems to create the data is briefly touched upon. Secondly data is constructed directly from triangular meshes [2, 3] in a process known as voxelisation. This method is shown to produce more accurate images when used in combination with the direct surface rendering technique. Preliminary results of the voxelisation work [2] were presented at the 13th Annual Eurographics Conference (UK Chapter), and appeared in the conference proceedings. The voxelisation work [3] of this chapter has also been submitted to Computer Graphics Forum.

Chapter 6 looks at the use of contours for 3D data visualisation. The contours are in the form of a stack where each slice in the stack has a number of contours which represent the intersection of the object with that slice. Contour methods attempt to create a triangular mesh which best represents the original object, from these slices. This chapter reviews the current methods available and indicates their failings and shortcomings. It presents a new method for the creation of these triangular meshes [4] and shows how the method copes with all the cases that are problematic to the other methods. This work was presented at Eurographics 1994 (Oslo, Norway), and appeared in Computer Graphics Forum 13:3, pp C-75–C-84, under the title "A New Approach to the Construction of Surfaces from Contour Data".

Chapter 7 presents a review of the preceding chapters and gives a few concluding remarks

about the work. A glossary of graphics terms [5] complements this work.

This thesis can be considered to have been written at a high level. Although there are some implementation details, these are only provided where necessary and in all cases low level information such as data structures, discussions about why one function is more efficient than another, and so on, have been omitted for clarity. This deliberate action has been taken in order to make the thesis more readable, less tedious and not prohibitively long.

Chapter 2

Surface Tiling

2.1 Introduction

In almost all circumstances, it is preferable to view 3D scientific data in 3D, rather than as a series of 2D slices. A typical example is that of medical imaging. The data is obtained as a stack of either X-ray or magnetic resonance images, each one representing a cross section of a person's body. When diagnosing a patient's ailment, the doctor will examine the 2D slices for any anomalies, trying to build up a mental image of the 3D structure of the volume of interest. Surface tilers remove the need for this mental agility by extracting the 3D surfaces that are contained within the stack of 3D data. In Section 2.2 the general operation of a surface tiler is described along with the necessary conditions that should be met. Determination of the case tables is also described along with the idea of using a look up table to tile the surface. Section 2.2.3 introduces a surface tiling algorithm - marching cubes [6]. Some cases of the basic tiling algorithm produce ambiguous surfaces, which are investigated, and a process to disambiguate them is presented.

A review of other methods to produce, disambiguate and optimise the isosurface is given in Section 2.3.

While the basic marching cubes algorithm is effective, the results are achieved by a brute force traversal of the volume. Wilhelms and Van Gelder [7] state that between 30% and 70% of the time spent in surface reconstruction is spent examining empty cells. Thus a great increase in performance can be achieved if the set of cells considered to produce the isosurface is as close as possible to the subset of useful cells, but still containing all the useful cells. This is the impetus for the development of methods that seek to reduce the set of cells to be considered.

These methods are investigated in Section 2.4. Both the use of an octree to organise the volume data is described and the use of a surface tracking algorithm. A new acceleration method which creates an index to the volume data is presented in Section 2.5. A comparison of these methods is made in Section 2.6, where the results of using each method for various well known data sets are presented and the conclusions that can be drawn from these results are discussed. Finally the chapter is concluded in Section 2.7.

2.2 Surface Tilers

2.2.1 Basic properties

Surface tiling is more generally referred to as the process of isosurfacing. The 3D stack of data is searched to find all those points with a value equal to some predefined threshold value that defines the isosurface of interest. In general the data is defined by a function f, where:

$$f : \mathbb{R}^3 \Rightarrow \mathbb{R} \tag{2.1}$$

and in most applications we would assume that f is at least piecewise differentiable. If the data comes from some scanned source (for example computational tomography (CT) data) rather than some computed source (for example computational fluid dynamics (CFD) data) then f will only be defined at discrete points, and will not be continuous. Linear interpolation is used to find values between these discrete points at which the function values are known. The isosurface, S of value τ, is given by all those points x, where $f(x) = \tau$:

$$S = \left\{ x \in \mathbb{R}^3 \mid f(x) = \tau \right\} \tag{2.2}$$

and the volume of space I, which is inside the isosurface of threshold τ, is defined as:

$$I = \left\{ x \in \mathbb{R}^3 \mid f(x) > \tau \right\} \tag{2.3}$$

Such an ideal representation is not achievable on the computer since infinitely many points are required to model the surface. Instead the process of isosurfacing produces a mesh of geometric primitives, such as triangles, and planar or non-planar polygons which provide a good approximation to the surface. It is this mesh that is displayed, in order to present the 3D surface contained within the data. The problem is that of how to produce a mesh which best represents the isosurface. This is done by subdividing the volume into simple polyhedra - tetrahedra or cubes, and determining the intersection of the surface locally with these polyhedra. Since most volumes are parallelpiped in shape or can be extended to be parallelpiped in shape without loss of generality, subdivision can be achieved by dividing the volume into cubes, and if subdivision by tetrahedra is required, the cubes can be further subdivided into tetrahedra (Figure 2.1). Note that in the case of subdivision into five tetrahedra, two differently oriented divisions are required to ensure coherency in the tetrahedral mesh.

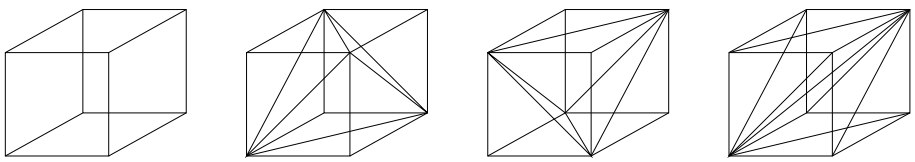


Figure 2.1: Subdivision into tetrahedra

Determining the surface is then reduced to the task of determining the intersection of the surface with each transverse polyhedron. A polyhedron is said to be transverse if at least one vertex is inside the surface and at least one vertex is outside the surface. That is, if the values at the vertices of the polyhedron are $v_0, \ldots, v_n$, then for the polyhedra to be transverse $\exists v_i v_j : v_i \leq \tau < v_j$. A transverse face or edge of a polyhedron is defined similarly. Since there are two possibilities for each vertex of the polyhedron - either inside the surface or not, then in the case of a subdivision using cubes there are $2^8 = 256$ possible cases, and for subdivision by tetrahedra there are $2^4 = 16$ possible cases. Polygon vertices are interpolated along each transverse polyhedron's edges and the resulting polygon indicates how the surface passes through the subvolume represented by the polyhedron.

The result is a mesh of polygons which may be displayed using standard rendering algorithms. Vertex normals used by rendering algorithms can either be calculated geometrically from the mesh, or they can be obtained directly from the data. Since the data models a function f, the normal to the surface at a point (x, y, z) is given by $f'(x, y, z)$, which is the first derivative of the function. Since the function is known only at discrete points, the derivative cannot be determined analytically, rather some approximation must be used. The gradient G, of the function f, can be calculated as a vector of gradients in each of the coordinate axes. These can be calculated using a central difference approximation (Equation 2.4), where h_x, h_y and h_z are the spacings of the data in each axis. Standard rendering algorithms such as Gouraud and Phong shading can display the mesh using the normals calculated at the polygon vertices to obtain high quality images.

$$\begin{aligned} G &= (g_x, g_y, g_z)\,, \\ g_x &= (f(x + h_x, y, z) - f(x - h_x, y, z))/h_x, \\ g_y &= (f(x, y + h_y, z) - f(x, y - h_y, z))/h_y, \\ g_z &= (f(x, y, z + h_z) - f(x, y, z - h_z))/h_z. \end{aligned} \tag{2.4}$$

It is desirable for the interpolated polygon points to appear only once in the mesh, that is to say that polygons share points where possible. This leads to a faster rendering of the mesh since less points are involved in the view transformation. It also leads to faster isosurfacing if the algorithm that establishes a correspondence between an existing point in the mesh data and the point to be interpolated is faster than calculating the floating point position and normal for that point.

It is also desirable that common edges between polygons should be used in both directions - a property known as coherency. That is, if for every adjoining polygons P_i and P_j, for the shared edge $V_{P_{in}}$, $V_{P_{in+1}}$, in P_i, P_j contains the edge $V_{P_{jk}}$, $V_{P_{jk+1}}$ where $V_{P_{jk}} = V_{P_{in+1}}$ and $V_{P_{jk+1}} = V_{P_{in}}$. This is because most renderers (both hardware and software) treat anti-clockwise polygons (to the viewing normal) as external faces and clockwise as internal faces. This is known as the right hand rule because if the right fist is placed in the interior of the object and the thumb is pointed outwards, normal to the surface, then the direction of the fingers (anti-clockwise) gives the order of the vertex list representing the polygon. From Figure 2.2 it is not difficult to see that complementary cases (vertices inside the volume for a particular polyhedron are no longer inside, and vice versa) must be treated differently to ensure a coherent surface is produced. Marked vertices are interior to the surface. It is

important to understand it is not necessary to produce coherent surfaces as a global post-processing step as in the case described by Wallin [8], but that they can be produced locally during the determination of surface intersections with each polyhedron.

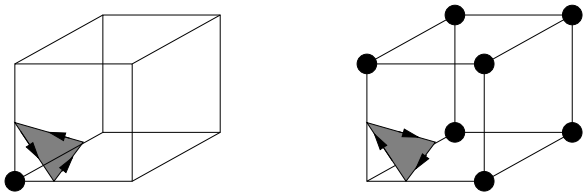


Figure 2.2: Complementary cases produce differently oriented surfaces

A surface is said to be connected if all edges occur in pairs and no polygon touches another polygon except at a common edge. Since polygon edges only occur on a face of a polyhedron and each face is common to just two (neighbouring) polyhedra the polygon edges will be shared by the polygonal surfaces passing through each of the polyhedra, therefore no two polygons can touch apart from at that edge.

The only doubtful case is when voxel values are actually equal to the threshold. Each of the edge intersections will be snapped to the face when all vertices are equal. But by observation it can be seen that if all surrounding voxel values are inside the surface then just these two polygons are generated and that they do satisfy the coherency, shared points and shared edges criteria. On the other hand if any of the surrounding voxel values are outside the surface then in those neighbouring polyhedra the surfaces will split, and will later connect at another edge in the volume.

In order to ensure the surface is closed and connected it must be assumed that no voxel value on the boundary of the volume is inside the surface, since this would mean the surface may enter or leave the volume without the edges involved being used in pairs in complementary directions. This can be assumed without loss of generality since a *shell* of values less than the threshold can be placed around the volume, thus ensuring the surface is totally contained within the volume.

In summary:

- A volume can be subdivided into simple polyhedra - cubes or tetrahedra.
- An approximation to the surface can be produced by determining how the surface intersects each polyhedron.
- The points of intersection with each polyhedron's edge are calculated using linear interpolation.
- The properties of the mesh that are desirable are:
  - each point in the mesh is unique;
  - edges should appear in pairs and in complementary directions to ensure connectivity and coherency;
  - a polygon should not appear twice unless it is to satisfy the above edge property.

- To ensure connectivity a shell of values outside the surface can be placed around the volume.

2.2.2 Determining the case table

For cubes there are 256 possible cases for surface tiling, each of which can be represented by an 8-bit vector $b_i (0 \leq i \leq 7)$. The vector is calculated as:

$$b_i = \begin{cases} 1 & \text{if vertex } i \text{ is inside or on the surface;} \\ 0 & \text{if vertex } i \text{ is outside the surface.} \end{cases} \tag{2.5}$$

where the vertices are numbered as in Figure 2.3(a).

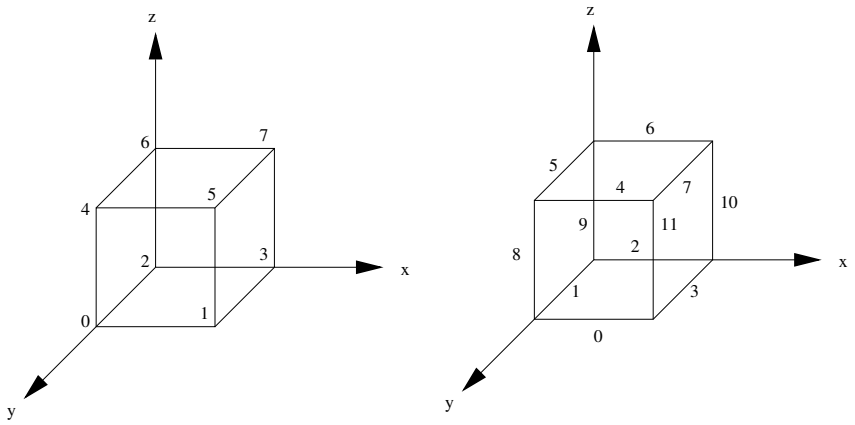


Figure 2.3: (a) Vertex enumeration. (b) Edge enumeration

A tiling for each of the possible cases is devised, and placed into a case table. The tiling of the surface is achieved by calculating the 8-bit vector b for each cube comprising the data, and looking up that cube in the table. The tiling for that cube is then the tiling as indicated for that case in the table. In calculating the case table it is only necessary to calculate the tiling for a few cases, and then by using rotation (Table 2.1) and transposition (Table 2.2), all other cases can be derived. It is recognised that rotation can take place about each axis, and transposition about each plane.

Rotation ($+90^o$) about	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
x	b_5	b_4	b_1	b_0	b_7	b_6	b_3	b_2
y	b_6	b_2	b_4	b_0	b_7	b_3	b_5	b_1
z	b_6	b_4	b_7	b_5	b_2	b_0	b_3	b_1

Table 2.1: Table showing bit manipulation for rotation

Taking each case in turn, all cases that can be derived from it can be eliminated from that table leaving the 20 cases of Figure 2.4. The marked vertices indicate the vertex is inside the surface.

Transposition about	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
xy	b_3	b_2	b_1	b_0	b_7	b_6	b_5	b_4
yz	b_6	b_7	b_4	b_5	b_2	b_3	b_0	b_1
zx	b_5	b_4	b_7	b_6	b_1	b_0	b_3	b_2

Table 2.2: Table showing bit manipulation for transposition

To derive the case table is a simple matter of devising tilings for each of the different base cases and applying the rotations and transpositions to obtain all possible cases.

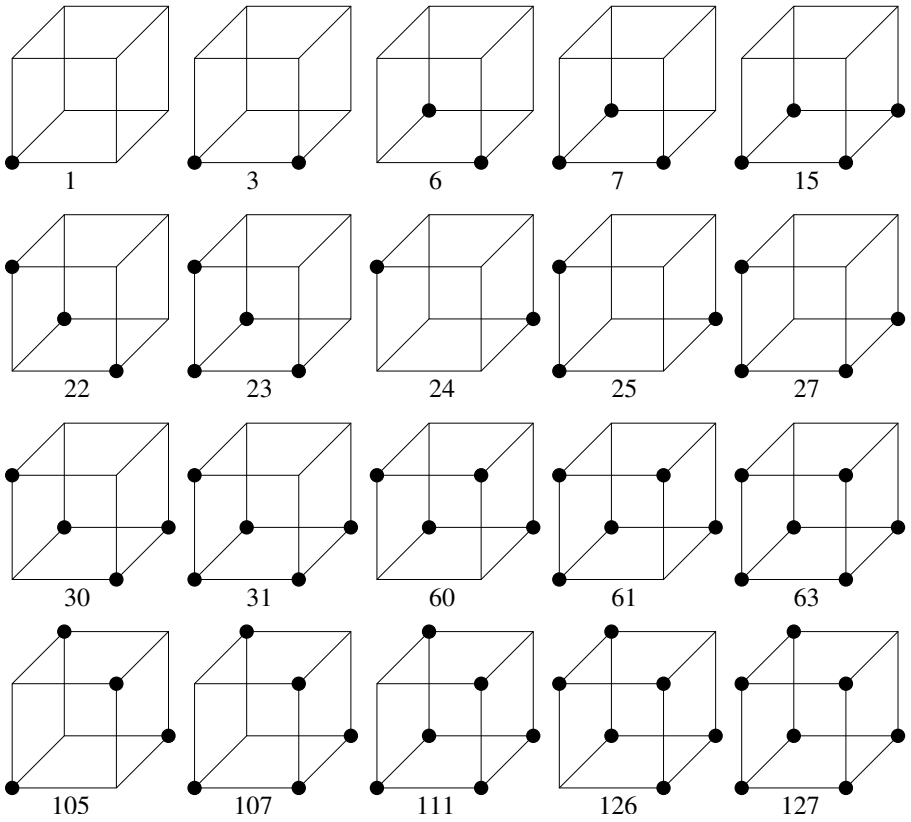


Figure 2.4: The 20 base cases for the surface tiling of a cube.

2.2.3 Marching Cubes

In Section 2.2.1 the general requirements for a surface tiler were described and Section 2.2.2 showed how the case table could be derived. In this section the process by which the surface is generated for each polyhedron, and one such surface tiling algorithm, namely marching cubes [6] are described.

For marching cubes, each of the 20 cases is given a specific tiling (Figure 2.5). In the figure marked vertices are those that are inside the surface. Using the enumeration convention for edges as shown in Figure 2.3(b), the tiling cases are given in Table 2.3.

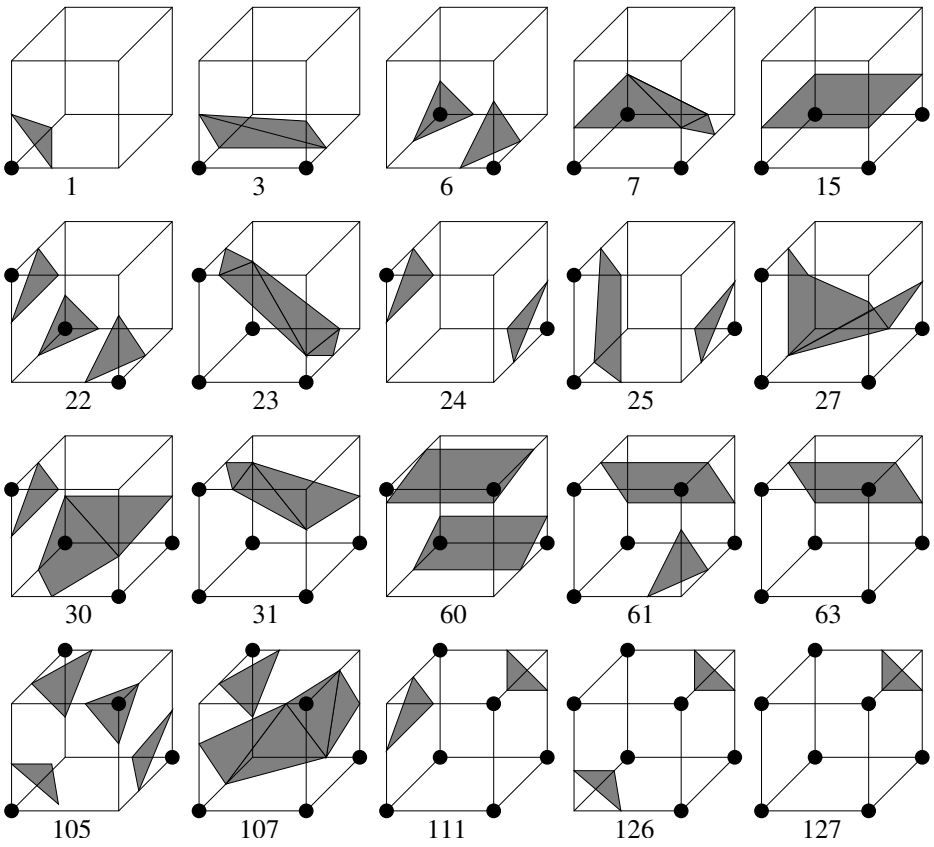


Figure 2.5: Tiling cases for marching cubes

	Triangles				
Case	1	2	3	4	5
1	0, 1, 8				
3	1, 8, 3	3, 8, 11			
6	1, 2, 9	0, 11, 3			
7	8, 11, 9	9, 11, 2	2, 11 3		
15	8, 11, 9	9, 11, 10			
22	1, 2, 9	0, 11, 3	8, 5, 4		
23	3, 2, 11	11, 2, 9	9, 4, 11	9, 5, 4	
24	8, 5, 4	3, 10, 2			
25	5, 4, 0	0, 1, 5	2, 3, 10		
27	10, 2, 11	11, 2, 1	11, 1, 4	4, 1, 5	
30	8, 5, 4	9, 1, 0	9, 0, 11	9, 11, 10	
31	11, 10, 9	9, 4, 11	9, 5, 4		
60	5, 7, 11	5, 11, 8	10, 9, 3	3, 9, 1	
61	0, 3, 11	5, 7, 10	5, 10, 9		
63	5, 7, 10	10, 9, 5			
105	0, 1, 8	3, 10, 2	6, 5, 9	4, 7, 11	
107	6, 5, 9	8, 4, 1	4, 2, 1	4, 7, 2	7, 10, 2
111	8, 4, 5	6, 7, 10			
126	8, 1, 0	6, 7, 10			
127	6, 7, 10				

Table 2.3: Table showing cases for cube tiling

The marching cubes algorithm proceeds by reading the data, and considering cubes made up of 8 data values, 4 values from one slice, and 4 from the next. Each of these data values is compared to the threshold, and a bit is set in an 8 bit flag if the vertex is inside the surface. The result is an integer between 0 and 255 which determines how the cube is tiled. If the flag is 0, then all data values are outside the surface, and the surface does not intersect the cube, similarly if the flag is 255, then all data values are inside the surface. Only cubes that produce values between 0 and 255 are transverse and therefore produce an intersection with the surface. This value is used to look up the tiling case in the look up table which covers each of the tiling possibilities mentioned above. For efficiency purposes this table should contain all the data that is needed in order to calculate a tiling for each case - namely the edges that need to have intersections to be calculated for, and the connectivity of the vertices produced by the intersections.

By the use of the table, the edges to be intersected and connectivity information can be found with a minimum of computation, and therefore most of the computational time is spent determining the actual points and normals. Checking which points already exist is achieved most efficiently by using a linked list of cubes about which information is known, with a hash table providing access to the list. By observation it can be seen that the marching cubes algorithm progresses through the volume in a row by row, slice by slice order so that information need only be passed to the three neighbouring cubes in front of the current cube. Once isosurfaced, which includes passing known points to neighbouring cubes, the cube can be removed from the list safely because it will never be visited again, and no other cube will need to store information in it.

Using this method of removing cubes no longer required, the number of cubes stored is controlled, and kept to a minimum, thus reducing the amount of space that methods which store all the points require, and increasing the efficiency of the search. Furthermore by storing a code indicating which faces are transverse, neighbouring cubes which do not contain any common information with the current cube need not be created or updated in the list. For example in the first case of the second column of Figure 2.5 (case 3), the neighbour in the increasing x direction needs to be either created or updated, whereas the neighbours in the increasing y and z directions do not share any information with the current cube and need not be considered at this stage.

The algorithm for marching cubes on a volume consisting of x data points in the x direction, y in the y direction and z in the z direction, can be stated simply as:

```
read_data(x,y,z)
  for k=0 to z-2
    for j=0 to y-2
      for i=0 to x-2
        flag = calculate_flag(i,j,k)
        isosurface_cube(i,j,k,flag)
      endfor
    endfor
  endfor
```

where the read_data procedure obtains the dimensions of the data, and reads in the data. The loop progresses through the volume in a row by row, slice by slice order, calculating the case flag as described above, isosurfacing each cube using the case table and taking care of all the information that needs to be held for cubes remaining to be isosurfaced. The algorithm finds the complete surface contained in the volume as a whole by considering each cube within the volume and determining the surface local to each transverse cube. The result is a mesh of points, normals and connectivity information which may then be displayed using a standard rendering technique such as flat, Gouraud or Phong shading.

2.2.4 Ambiguous tiling cases

It was first shown by Dürst [9] that the cube cases as used in marching cubes are incomplete since it is possible to have an ambiguous tiling, for example see Figure 2.6. The two cases are 130 and 235 and using the case table we obtain the tiling as shown in Figure 2.6(a). In this case a hole has appeared in the surface where there should not be one, which can be confirmed by drawing a line from the bottom front right vertex within the object to the external top left back vertex without passing through the surface.

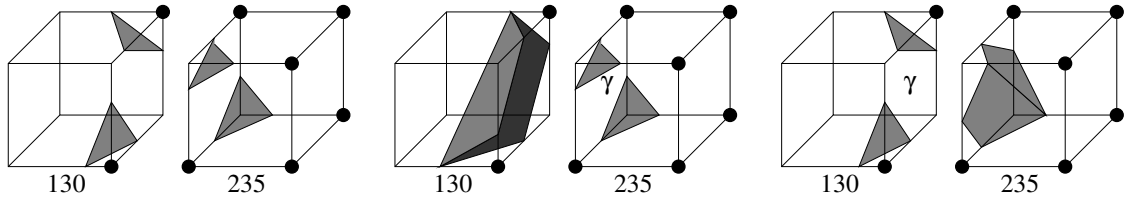


Figure 2.6: (a) Ambiguous situation. Solution if γ is (b) inside, (c) outside

Such ambiguous situations occur when two vertices belonging within a surface occur diagonally opposite within a face, along with two vertices that are external. Figure 2.7 shows how the face could be divided. In case a) $\gamma < \tau$, that is the centre of the face is outside the surface, and in case b) $\gamma > \tau$, that is the centre of the face is inside the surface.

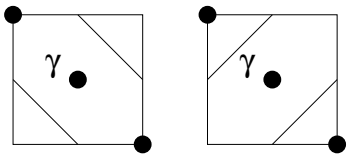


Figure 2.7: Vertices are (a) Separated or (b) Connected.

Once an ambiguous case has been identified the centre of each ambiguous face can be checked to determine whether or not it is inside the surface. This can give up to 64 different possibilities (6 faces in case 105 (Figure 2.4), therefore $2^6 = 64$ possible cases).

In order to deal with these additional ambiguous cases an extension to the case table was developed, which indicates which cases are ambiguous, and provides an index to a secondary table which contains the tiling possibilities depending upon the value at the centre of each ambiguous face.

For the ambiguous cases (6, 22, 25, 30, 60, 61, 105, 107, and 111) a pointer is stored in the main case table indicating where in the second table information on how to disambiguate the case is stored. Firstly the value at the centre of each face is bilinearly interpolated and compared with the threshold operator. By setting a flag corresponding to the face to 1 (if it is inside the surface) or 0 (otherwise), an additional pointer to a new tiling case can be constructed. It is this tiling that is used to resolve the ambiguity.

2.3 Review of Surface Tiling Methods and Variations

2.3.1 Introduction

The main surface tiling method was described in Section 2.2 along with a simple cube by cube traversal. In this section other ways of generating isosurfaces are described and reviewed.

2.3.2 Topological ambiguities

As described in Section 2.2.4 it was first pointed out by Dürst [9] that the marching cubes algorithm [6] resulted in ambiguous surfaces that contained holes where there should not be any. Further methods [10, 11] have been suggested which resolve these ambiguities, and are discussed in this section.

Wilhelms and Gelder [10] made an investigation into the disambiguation of topological cases. They identify the distinct cases of tiling a cube, and indicate which of these cases are ambiguous – namely those with two opposing vertices that are diagonally opposite within a face.

They include a brief suggestion by Lorensen and Cline which attempts to correctly tile the surface, but show that it can create false surfaces in other cases. The methods they describe for disambiguation of the tiled surface are called facial average values, gradient consistency heuristics and quadratic fit.

The facial average values method is similar to that of the extension made to marching cubes in Section 2.2.4 and is also used by Wyvill et al. [12]. For this method the value at the centre of the ambiguous face is bilinearly interpolated within the face, and used to determine if the centre of the face is inside or outside the surface.

The gradient consistency heuristic methods use information from beyond the cell to disambiguate. The centre pointing gradient method calculates the component of the gradient at the cell vertices that points towards the centre of the face. This gradient is added to the vertices to obtain an improved estimate of the function at the centre of the face.

The quadratic fit method assumes that the underlying function within the data can be represented locally by a quadratic function. This function is obtained using the least squares method, and again the value at the centre of the face is used to determine whether the centre of the face is inside the surface or not.

The above methods produce several alternate cell tilings for ambiguous cases which are dependent upon the value at the centre of the face. As described in Section 2.2.4, the major case table can be supplemented by a subcase table which determines the tiling of an ambiguous cell.

Nielson and Hamann [11] investigate a method by which the problematical cases for marching cubes can be handled. They describe the operation of marching cubes and identify the different cases. The ambiguous cases from these are identified and a suggestion to correct the matter is made. For their method they treat the ambiguous face as a unit square, and by using the function for bilinear interpolation they show that contour curves of the face are hyperbolas.

The ambiguous cases occur when both asymptotes of this hyperbola intersect the domain. For this case they find the bilinear interpolant γ at the intersection point of the asymptotes, and compare it to the isosurface value τ. If $\gamma \leq \tau$ the vertices are separated.

The tiling of the cube is dependent upon how many faces are ambiguous, and which have vertices to be separated, or not.

All of these methods use a similar subsidiary case table to code the possible tilings to the one of Section 2.2.4. The main difference between the methods is how each method determines the value within the face that is used to refer to the case table. All the methods produce topologically correct surfaces which are subtly different according to which method is chosen. It is impossible to say whether one method is correct and the others are not.

These ambiguities were also recognised by Payne and Toga [13]. Their solution was to simplify the tiling process by dividing the cube into five tetrahedra. The tetrahedra can then be simply tiled using zero, one or two triangles depending upon the case. Since there are 4 vertices, there are $2^4 = 16$ possible cases, which can be simply reduced to 3 cases (Figure 2.8).

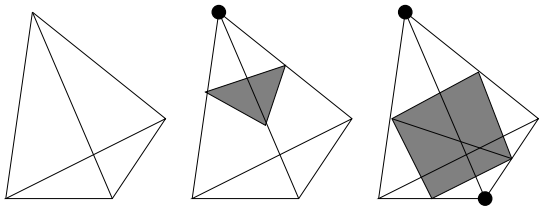


Figure 2.8: The three cases for tiling tetrahedra.

By subdividing the cube into tetrahedra, ambiguous cases are avoided, and the tiling can go ahead successfully. The surface produced is topologically correct, and closed except for possible holes at the boundary. In order to preserve continuity along cube boundaries, adjacent cubes had to be subdivided into mirror-image patterns (Figure 2.1).

The advantage of this method is its simplicity but this is achieved at the expense of producing many more triangles per cube than marching cubes.

2.3.3 Producing isosurfaces with fewer polygons

One of the main problems associated with surface tiling methods is the fact that the resulting mesh generated by the isosurfacing procedure is often quite large. For a typical CT medical

data set of 113 slices of 256 × 256 pixels, the resulting mesh can be in excess of 500,000 triangles. This size of mesh is caused by the fact that each cell can produce many triangles – up to 12, and there are many cells on the surface boundary. Several methods attempt to reduce the size of the mesh using one of two approaches. The first approach is to reduce the size of the mesh during isosurface production, and the second is to reduce the size of the mesh as a post-processing step.

Müller and Stark [14] present the splitting box algorithm which produces surfaces adaptively. It retains the properties of the marching cubes surface in that vertices inside the surface, and vertices outside the surface are still separated by the algorithm. In addition it preserves the topology of marching cubes, which is to say that surfaces are closed (except possibly at the boundary of the grid) and that all connected components are preserved. The algorithm works by performing a marching cubes tiling at a higher level, and then calculating how accurate that tiling is to the precise tiling. If it is not accurate to within a certain tolerance, the box is split, and the process continues until the box is the size of one cell, in which case the tiling will be the marching cubes tiling for that cell.

Their results indicate the adaptive mesh to be about 45%–65% the size of the mesh produced by marching cubes. They also provide results that their algorithm is faster than marching cubes, although they indicate that they have not made any optimisations as described in Section 2.2. With these optimisations it is estimated that their algorithm would be 50%–100% slower than marching cubes. The quality of the mesh produced by the method is acceptable, although there are obvious artifacts in the images they present.

Yun and Park [15] attempt to fit a new set of polygonal volume primitives (PVPs) to the basic cube cases of marching cubes. They suggest that just five PVPs are all that are required to cater for all cubes cases. For each cube case that contains a surface primitive, the closest candidate PVPs are found, along with candidate orientations. The surface generated at that cube is calculated and used to choose the most likely candidate. The advantage of this method is the fact that less triangles are produced for the surface – only 1 or 2 per cube. This results in less storage requirements, and faster rendering. The PVPs also ensure that none of the ambiguities of surface tiling methods occur.

The main problem with this method is the fact that the authors exclude some of the marching cubes cases since "*they have too few points and are not expected to make much contribution to establishing the surface*". The cases they reject constitute over 70% of cases [11] and therefore must constitute an important part of the representation of the surface. These cases along with grey level gradients contain much of the fine surface details. This opinion is borne out by the fact the images in their paper are lacking in fine surface detail, even though the original data has a fine grid. Details of testing are provided, but no comparison is made with the standard method. The run times seem a little excessive and could be due to the number of surface and PVP normals that need to be calculated to decide the closest candidate PVP.

Avila et al [16]. use the marching cubes algorithm to produce an isosurface, but manage to store it compactly and accelerate the projection. They observe that it is sufficient to allow only four possible vertex positions along each cell edge. The surface information can be stored compactly just using an indication of which edge position the triangle's vertex is positioned closest to. The rendering can also be accelerated by using precomputed tables for

vertex positions and normals.

Some of the various methods and techniques outlined above are compared by Ning and Bloomenthal in [17] where they give tables which compare case occurrence and the number of triangles produced for each method.

2.3.4 Reduction of triangular mesh size

There are various methods [18, 19, 20] to reduce the size of a triangular mesh as a post-processing step. Hoppe et al. [18] state the problem as "*Given a collection of data points X in $\mathbb{R}^3$ and an initial triangular mesh M_0 near the data, find a mesh M of the same topological type as M_0 that fits the data well and has a small number of vertices*". Their main emphasis is towards the production of a simplified surface from sample points which have been determined by scanning some surface. They identify three factors of the surface which must be minimised – measure of fit, number of vertices, and a regularising term. This final term is a measure of vertex lengths and ensures that no erroneous spikes occur in the data. The above factors are minimised to produce the surface representation by optimising existing vertex positions, and adding or removing vertices. Their method can be used to perform mesh simplification by sampling a set of points randomly on the original surface, adding the original vertices of the mesh and additional points from boundary edges, and then applying the optimisation. The resulting simplified mesh has the advantage that vertices and edges are concentrated in regions of sharp features and high curvature with a few large facets covering flat regions.

Turk [20] describes a method for retiling polygonal meshes using a number of vertices specified by the user. The method works best for meshes that represent smoothly curved surfaces, and is not particularly useful for meshes with corners and sharp edges. The only restriction on the mesh is the fact that edges are shared by at most two polygons. In such cases the method will be guaranteed to have the same topology as the original mesh.

The first step is to evenly distribute the new vertices over the surface by randomly placing each vertex in an area of the surface with least density of new vertices, and then repelling each vertex away from its neighbours along the surface. The repelling force can simply be constant, in which case a mesh with an even distribution of vertices is created. It is better to concentrate vertices in the regions of high curvature, and to ensure this, a repelling force inversely proportional to local curvature is used.

In order to triangulate these vertices, and retain the topology of the original mesh, each polygon in the mesh, along with the new vertices within it, is triangulated. Each of the original vertices is then removed from the mesh and the local area is retiled. Once all of the original vertices have been removed, the resulting mesh contains the specified number of vertices, and still has the correct topology. Some cases do occur which result in a change in the topology of the mesh, if a vertex is removed. In these cases the vertex is necessary to describe the surface correctly, and is thus retained.

The main application of the work is for the reduction in size of triangular meshes, and to that end the method is successful. Turk also shows how models can be interpolated between the original and retiled models which could be used in application areas where an object

moves from a low resolution area to a high resolution area (for example flight simulations and virtual reality). The images provided show meshes with well behaved triangles as obtained from the original mesh. They also show meshes with various numbers of vertices derived from the same original mesh, that represent the original mesh very well.

Schroeder et al. [19] describe the process of decimating a triangular mesh. For their method they make multiple passes through the mesh, removing vertices that fall within their decimation criteria. The triangles that use that vertex are also all removed and the resulting hole is patched by triangulating the area local to that vertex.

They seek to retain the original topology and also provide a good geometric approximation to the original mesh. They also show that the resulting mesh can use just a subset of the original vertices by removing, but never adding vertices.

The decimation criteria depends upon how a vertex is classified. A simple vertex is removed if it is within a certain distance to a plane which is averaged from neighbouring triangle centres. A boundary vertex is removed if it is within a certain distance from an edge which can be constructed from local triangles.

The decimation process is applied to volume modelling. In both cases meshes from the data sets are shown along with meshes that have been decimated by up to 90%. The progression from original mesh to decimated mesh is intuitive, with the topology of the mesh retained, and surface details slowly flattening out. The method is quite successful for large reductions in the number of triangles of meshes.

2.3.5 Production of higher order surfaces

Miller et al. [21] suggest a new method for producing surfaces from volume data which has the advantages of producing models of varying resolution and when applied to noisy data is stable. The method is analogous to blowing up an elastic balloon within an object and allowing the balloon to fill concavities within the object. First a candidate model (they suggest an icosahedron) is placed within the object, and the vertices of the model are moved outwards whilst remaining connected. Movement is determined by the gradient in the data local to each vertex, and the distance of a vertex from its neighbours. Each vertex can be represented by a function of these factors which is then minimised over the whole model to produce the final object. The term governing the position of neighbours ensures that no vertex is *held up* by erroneous data, or escapes from an *opening* in the object. Both factors would serve to produce a surface which was not closed if a different surface extraction procedure was used. At any time vertices may be added to the model if it is not complex enough to sufficiently define the object. Such introductions are also governed by terms in the function representing the model, and are hence affected by the minimisation step. The result is an approximation to the surface by a closed geometric object using a resolution specified by the user. The new model which has been derived from the candidate model is called a Geometrically Deformed Model (GDM).

The problems with this method are that the quality of the resulting model does not approach that of a surface produced by conventional techniques and is therefore not useful as a visualisation tool. The model itself is dependent upon local data values and as such has a

tendency to intersect itself. More importantly it has not been successfully applied to any data set without some manual intervention, and the computing time alone is several hours as compared to several seconds or minutes as in the case of other visualisation methods. They mainly use GDMs to produce models which are more sparse than the large triangular meshes other tiling methods have produced.

Gallagher and Nagtegall [22] are concerned with the production of surfaces from finite element meshes and coarse grids. The problem they identify is the fact that results are only available at coarsely spaced points which leads to surfaces with low resolutions. One solution is to resample the mesh at a higher resolution, interpolating new data values, but this leads to more computation and higher storage costs. The method they suggest is that of fitting bi-cubic result surface segment patches (RSS), using the vertices and normals of the surface intersection with the edges of the element. They show how to calculate the RSS patches with special attention given to the production of surfaces that have high geometric order, and ease of definition based on the available data. This results in a surface which is smooth in appearance and allows the result values to be understood more readily.

2.4 Acceleration Methods

2.4.1 Introduction

In this section several methods are compared which attempt to reduce the number of cubes considered during isosurfacing. In Section 2.4.2 an octree is used to spatially organise the data and the traversal method takes advantage of the octree data structure to eliminate many of the cubes that do not enter into the tiling process. Section 2.4.3 shows how tracking of the surface through the volume is achieved in order to consider those cubes, and only those cubes that enter into the isosurfacing process.

2.4.2 Octree method

The octree method [7] seeks to reduce the number of cubes considered as candidates for isosurfacing by pre-processing the data in order to gain as much information as possible about the volume. The pre-processing step involves building an octree where the leaf nodes point to the origin of the 8 candidate cubes in the volume to be isosurfaced, and each node in the octree contains the maximum and minimum data values of the subtree with that node as root.

The isosurfacing of the volume is then reduced to the task of traversing the octree for a particular threshold value, calculating the flag and isosurfacing each group of 8 cubes selected as candidates by the octree, using the marching method. By observation it can be seen that if the octree is traversed in-order, each cube need only pass information to its three neighbours in the increasing x,y and z directions. This can be seen clearly in two dimensions (Figure 2.9), where passing information in the increasing x and y directions allows all interpolated points to be shared. This not only allows the use of the same procedures for isosurfacing the cubes, but also allows the use of the forward information passing procedures

as described in Section 2.2.3, thus allowing a fairer comparison of the performance of the two methods, since only the traversal methods differ.

11	12	15	16
9	10	13	14
3	4	7	8
1	2	5	6

Figure 2.9: In 2D information is passed in the increasing x and y directions

The main problem with this is the amount of storage required for the octree - when the volume data is of dimensions $2^k \times 2^k \times 2^k$ the octree gives the ideal ratio of octree nodes to data approaching $\frac{1}{7}$ for large k. Whereas for volumes other than this size it can become inefficient. Wilhelms and Gelder [7] present an elegant solution to this problem called the Branch On Need Octree (BONO), and prove that in the worst case the ratio is $\leq \frac{1}{6}$ for volumes with all dimensions larger than 32. Once created the BONO can be traversed in the same way an octree is traversed, in order to produce a list of candidate cubes to isosurface.

To build an octree for data of dimensions a_i $(i = 1, 2, 3)$ where a_i is the number of points (1 based, that is, in direction d_i, points are labelled $1...a_i$) in each dimension d_i, the range is considered, where r_i is the number of cubes (0 based) in that direction. Therefore $r_i = a_i - 2$.

Let $k = max\,(|r_i|)$ $i = 1, 2, 3$ where $|x|$ is the number of the highest bit in the binary representation of x. Then the number of nodes, *nodes* required to store the BONO, and the position of the first node in each level, *depth* of the BONO, *place* [*depth*] is given by the following code:

```
depth=1; nodes=1; place[0]=0; /* this is the root node */
while (k>0)
  t = ((r1 >> k)+1)*((r2 >> k)+1)*((r3 >> k)+1);
  place[depth] = nodes;
  nodes=nodes+t;
  depth = depth+1;
  k=k-1;
endwhile;
```

where >> is the bitwise right shift operator.

The BONO is then created recursively by determining the directions in which the BONO branches at each node by examining the k^{th} bit of the range r_i, where $k = max\,(|r_i|)$ and creating the node with the maximum and minimum values of its subtree and a pointer to the first of its 1 to 8 children which are stored contiguously, along with the 3 bit branch code p_i denoting the direction in which to branch calculated by $p_i = r_{ik}$ where r_{ik} is the k^{th} bit of r_i and $k = max\,(|r_i|)$ $i = 1, 2, 3$. The leaves of the tree contain a pointer to the origin of the 8-27 voxel values comprising the 1-8 cubes the leaf's maximum and minimum represent.

For example the Chapel Hill CT head data set has dimensions $256 \times 256 \times 113$, with a shell of values this becomes 258*258*115. The root will have range

$$r_1\ 100000000 = 256$$

$$r_2\ 100000000 = 256$$

$$r_3\ 01110001 = 113 \text{ and } k = 8$$

and so will branch in the d_1 and d_2 directions, but not the d_3 direction, and therefore $p = 011$. The new range of a child n_i, of the node is calculated by removing the k^{th} bit of r_i if the child branches upwards in direction d_i, or the range is equal to $2^k - 1$ if the child branches downwards in direction d_i.

$$n_i = \begin{cases} r_i \text{ xor } 2^k, & \text{if } d_i = 1 \text{ or } p_i = 0, \\ 2^k - 1, & \text{otherwise.} \end{cases} \tag{2.6}$$

In the above example one of the children will be 011, that is, it will branch upwards in the d_1 and d_2 directions. Its range will be 0,0,113, that is a row of 114 cubes, and the origin of the block in the data is $2^k \times 2^k \times 2^0$ =256,256,0. The new origin o_i is calculated from the old origin b_i as

$$o_i = \begin{cases} b_i, & \text{if } x_i = 0, \\ b_i + 2^k, & \text{if } x_i = 1. \end{cases} \tag{2.7}$$

By creating the BONO using the above method and traversing it as a normal octree is traversed, a list of candidate cubes to isosurface is produced.

2.4.3 Surface tracking

Surface tracking methods attempt to follow the surface to be extracted through the 3D volume. The methods do not rely on any pre-processing step, but do require that one point on the surface be known. The algorithm investigated here is known as the *chain of cubes* algorithm [23].

The algorithm requires a seed cube which is known to be transverse. This cube is isosurfaced using look up tables as described earlier and then in order to track the surface, all neighbouring transverse cubes are placed on a stack of cubes to be isosurfaced. These cubes are easily identified since they must share a transverse face with the current cube. By following the surface into a neighbouring transverse cube and maintaining a stack of cubes left to be processed, the whole surface can be extracted by following each track around the surface until a previously isosurfaced cube is found. At this point the algorithm uses the first cube on the stack as a starting point for the next track. In this way a *chain of cubes* tracks across the surface.

Once the stack is empty the whole surface has been tracked, and therefore the isosurface contained within the data has been found by considering only those cubes that are transverse, and no other *empty* parts of the volume. There are two main problems with this method - the first is the fact that a seed cube must be known, and only the surface containing that seed cube is found. This leads to problems where more than one surface exists in the data

set, since each surface requires a seed cube to be known, and for arbitrary data sets, such as CT data, this would necessitate a scan of the data to find such seed cubes, obviating any benefit over marching cubes.

The second problem is the fact that the order of the cubes to be isosurfaced is arbitrary and cubes now have to pass information in all directions. It now becomes the case that a cube cannot be removed from the information passing list until all of its neighbours have been considered (rather than when only it had been considered as for the previous methods), which gives rise to the situation where the cubes in the list can no longer be controlled as in previous methods. For each cube to be isosurfaced up to 6 faces and 12 edge neighbours must be considered, thus increasing the overhead of keeping track of all known points beyond that of calculating all the points even if some are repeated.

The first problem cannot be overcome, the method was not intended to be used on arbitrary sampled discrete volume data sets, but rather to be used to visualise continuous trivariate mathematical functions. Using conventional methods, the function would have to be evaluated at each grid point within the volume before a surface tiling algorithm could be used. This method just requires a starting point which can then be used to track the surface, only calculating the function at grid points that *straddle* the surface. Therefore not only does this method save time by not considering the whole volume, but also by only evaluating the function where necessary.

The second problem can be overcome by using the method to first of all track the whole surface, without producing the surface, but rather a volume of bits indicating the transverse cubes. This data structure can then be traversed using three simple nested loops giving those cubes to be isosurfaced in the same order as marching cubes. This allows the book keeping and isosurfacing functions to be used for this method as well, thus allowing a fair comparison to be carried out.

2.5 Indexing – A New Acceleration Method

2.5.1 Indexing data

In this section a new method is presented that reduces the number of cubes to be considered through the use of a two dimensional index of the data. In certain commonly occurring circumstances this method leads to those cubes, and only those cubes, that contribute to the surface being considered during the tiling process.

For this method an index is created for the 3D data set and that portion of the index appropriate for the current threshold value τ can be traversed to produce a list of candidate cubes for isosurfacing. For most data sets the volume data has been pre-processed such that it is a regular grid of data where each element of the volume is one byte, if this is assumed the index can be given its most simple form as in Figure 2.10.

Any real threshold $\tau, 0 \leq \tau \leq 255$, would give the head of the linked list of the set of cells containing a surface intersection for that threshold. To build the lists, each cube in the volume is considered, and the maximum and minimum vertex values are found. Part of the

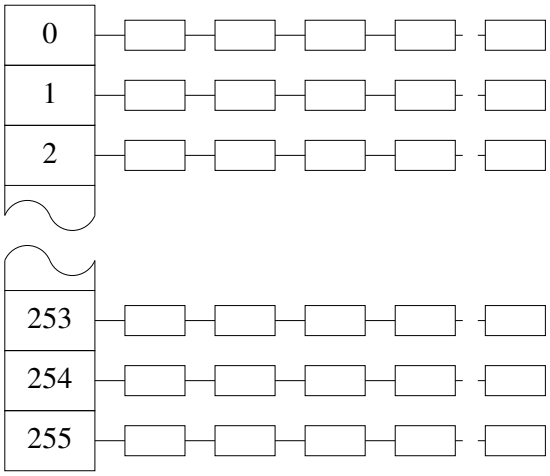


Figure 2.10: Index in simplest form

surface intersects with the cube if the threshold falls between these values, e.g. in Figure 2.11 any threshold τ with $2 \leq \tau < 8$ will produce a surface intersection with the cube and so the cube must be added to the lists in the index at positions 2 to 7 inclusive.

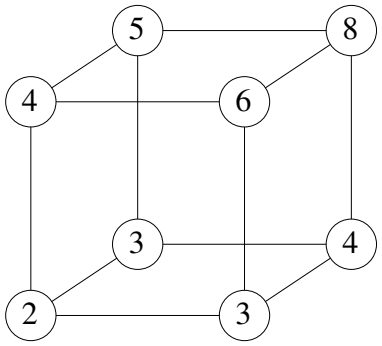


Figure 2.11: Example cube

It is obvious from the above example that many cubes will appear repeatedly in the lists and will therefore increase the amount of storage required, in fact experiments have shown that each cube will appear an average of 20 times in the index which results in a lot of redundancy. The storage requirements can be improved by deciding to use a better index. The index could be chosen from one among the following:

- A predefined number of lists could be chosen, for example 256. Then for a data set the range is calculated and each list represents a fraction of that range. For example a data set with a range of 0 to 255, with 256 sets, will result in each set representing 1 isosurface threshold and therefore the number of cubes to look at is minimal, whereas for a data set with a range of 4096, with 256 sets, each set will represent 16 isosurface thresholds, and therefore the number of cubes is not minimal. This has been tested on several data sets and results have shown that the set is about 4% bigger than the minimal set.
- The number of sets could depend upon the range of data. For example 256 sets would be

needed for the first case above, and 4096 for the second case to achieve minimal surface sets. This would increase the storage - by how much still needs to be investigated, but the sets would be minimal. The storage increase may outweigh the benefits of minimal set.

- The number of cubes in each set could be limited, or a best fit algorithm could be used. For example we could state that there is to be no more than 20,000 cubes in each set, the data would then be divided up so that the first set represents, for example, thresholds of 0-7, the second 8-11, third 12-13 and so on. The sets again would not be minimal, but a far better storage factor would be achieved.
- The simplest way would be to assume the data set has a resolution of 8 bits and then use 256 sets to give minimal sets, and then concentrate on developing a good runlength encoding algorithm to reduce the storage needed. Such an assumption is valid since most data sets are reduced to 8 bit for volume rendering.
- It may also be possible to check sets for equivalences and common subsets so that those sets only need be stored once, and then referenced by all the sets. For example, if the minimum value of a number of cubes is 2, and the maximum 8, then all of those cubes will appear 6 times in sets 2-7. Instead of this, those cubes could be stored once in a set and then referenced by each of the sets 2-7. This could be very complicated to decide and may increase the time to create the index to unacceptable levels. The benefit in reduced storage may not be enough, although this method is worth considering.
- A 2D index (Figure 2.12) can be used where each cube is placed in the list representing its maximum and minimum threshold value. Since each cube appears once, and only once in any position of the index, there is no redundancy. The transverse cubes are then those given by lists pointed to by index positions (x,y) where $x \leq \tau < y$.

The 2D index seemed the most promising and was implemented. Details are given in Section 2.5.2. Unlike the octree method there is a situation where the set of candidate cubes is exactly the set of cubes containing the isosurface. This special case arises from the fact that whereas the octree method organises the data spatially, the indexing method organises it according to the threshold. The case occurs when the sizes of both dimensions of the index are equal to the range of the data, e.g. if the data has 8 bits precision and the index is 256×256 then the candidate list for a particular threshold are exactly the set of all transverse cubes for that threshold.

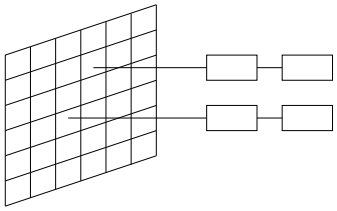


Figure 2.12: 2D Index

Once the index has been created, it can be used to extract lists of candidate cubes. Since these lists are not necessarily in order each appropriate list is traversed and a bit is set for

each cube in a 3D array of bits representing all the cubes in the volume. This 3D array can now be traversed bit by bit to give an identical traversal to the marching cubes algorithm, but only producing those cubes that need to be considered. Simple speed-ups can be achieved by treating the 3D array of bits as an array of integers or characters and skipping over 8, 16, 24 or 32 cubes at a time with one comparison (to zero).

2.5.2 Implementation details

In this section the implementation of a new method which indexes volume data is presented. The general idea behind this is to reduce the number of cubes considered during surface determination so that time is not wasted checking subvolumes which do not contain the surface.

Let $v_0, ..., v_7$ be the values at the vertices of a cube within volume data according to Figure 2.3(a). If $min(v_0, ..., v_7) \leq \tau < max(v_0, ..., v_7)$ $v_0, ..., v_7, \tau \in \mathbb{R}$, the isosurface of value τ passes through the cube. This assumes that if $v_i > \tau$ then v_i is inside the surface. The data is indexed by classifying it according to its maximum and minimum vertex values and placing it into a 2D index, unless $max = min$, in which case no surface passes through that cube. The reason that a 2D index is used is so that each cube is stored at most once in the position as calculated below.

Let each datum in the volume of dimensions x, y, z be referenced by V_{ijk}. $(0 \leq i \leq x)$, $(0 \leq j \leq y)$, $(0 \leq k \leq z)$, where $i, j, k \in \mathbb{N}$ and $V_{ijk} \in \mathbb{R}$.

For the case where the data set consists of **integer** numbers:

Let n be the size of the 2D index, $n \in \mathbb{N}$.

Let α be the minimum of all V_{ijk}.

Let β be the supremum of all V_{ijk}.

Let $\gamma = \frac{n}{\beta - \alpha}$

To distribute the cubes in the index, each cubes position (p, q) $p, q \in \{0, ..., n-1\}$ is found using,

$$p = \lfloor (min - \alpha) * \gamma \rfloor \quad q = \lfloor (max - \alpha) * \gamma - 1 \rfloor$$

where $min = min(v_0, ..., v_7)$, $max = max(v_0, ..., v_7)$.

Note since $p \geq q$ no cubes appear in the lower left of the index.

To recover the cubes from the index, all cubes $v_0, ..., v_7$ that satisfy $min(v_0, ..., v_7) \leq \tau < max(v_0, ..., v_7)$ will contain part of the isosurface for the given value τ where $\tau \in \mathbb{R}$. If a cube has a minimum value min and a maximum value max then for a value τ it must be recovered if $\tau \in [min, max)$.

The positions in the index that list such cubes are given by all (p, q) where

$$0 \leq p \leq \lfloor (\tau - \alpha) * \gamma \rfloor \text{ and } \lfloor (\tau - \alpha) * \gamma \rfloor \leq q \leq n$$

Special case : if $\gamma = 1$ and each $V_{ijk} \in \{0, ..., n-1\}$, the data is said to be fully indexed using the above method, and for any isosurface threshold value $\tau \in \mathbb{R}$ only those cubes that have a surface passing through them need to be examined.

More specifically most data is of a resolution of 8 bits, or is pre-processed to be of that resolution. Therefore by choosing n to be 256, $\gamma = 1$ and the data is fully indexed. Data is pre-processed using the following method.

For each $V_{ijk} \in [r, s]$, each V'_{ijk} in the new 8 bit data is given by $V'_{ijk} = g(V_{ijk})$ where

$$g : [r, s] \to \{0, ..., 255\} \text{ and } r, s \text{ are real.}$$

For the case where the data set consists of **real** numbers:

Let n be the size of the 2D index, $n \in \mathbb{N}$.

Let α be the infinum of all V_{ijk}.

Let β be the supremum of all V_{ijk}.

Let $\gamma = \frac{n}{\beta - \alpha}$ To distribute the cubes in the index, each cube's position (p, q) $p, q \in \{0, ..., n-1\}$ is found using,

$$p = \lfloor (min - \alpha) * \gamma \rfloor \quad q = \lfloor (max - \alpha) * \gamma \rfloor$$

where $min = min(v_0, ..., v_7)$, $max = max(v_0, ..., v_7)$.

To recover the cubes from the index, the positions are given by all (p, q) where

$$0 \leq p \leq \lfloor (\tau - \alpha) * \gamma \rfloor \text{ and } \lfloor (\tau - \alpha) * \gamma \rfloor \leq q \leq n$$

The reason why this distinction has to be made is because in the first case we are simply placing a finite set of integer numbers in a set of buckets, whereas in the second case we are placing the continuous real numbers in a finite number of buckets. We would have to decide whether max and $max + \delta$ would appear in seperate buckets, where $\delta \to 0$. Due to the machine representation of real numbers, and the fact that δ is some definite small number, this is hard to calculate. It is easier to place cubes according to the simpler equation, rather than finding the *correct* (more tightly bound) bucket.

2.6 Analysis

The above methods all offer alternative ways of traversing the data to the marching cubes algorithm's brute force traversal. By implementing each method as described, the efficient cube tiling and book keeping algorithms described in Section 2.2.3 can be used with just the way the list of candidate cubes is produced differing, thus allowing a fair comparison of the methods. Marching cubes and the octree method work for general arbitrary data sets and thus can be compared directly on all data sets. The indexing method operates in a similar

way to these methods but to be totally effective requires the data to be in an 8 bit format. The chain of cubes method is tested on data more appropriate to its action, data which contains one closed surface and an initial known transverse cube.

The surface tiling tables were coded and the various traversal methods implemented in C. The results in this section are for tests carried out on a DEC 3000 Alpha AXP model 400 workstation. The data used are the CThead (Figure 2.13), MRbrain[1] (Figure 2.14), lobster (Figure 2.15) and hydrogen molecule[2] (Figure 2.16).

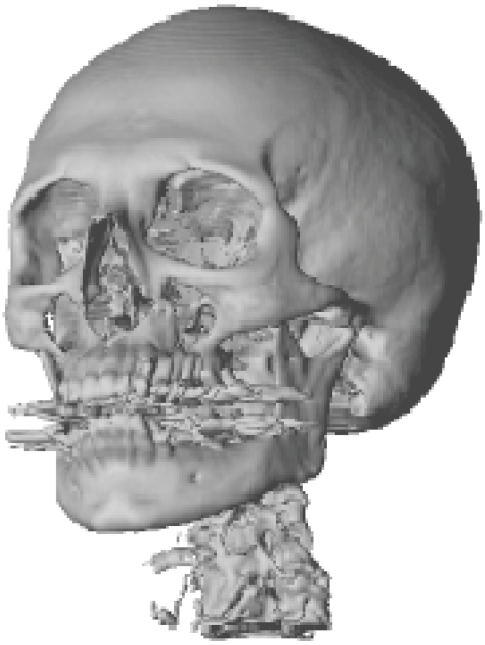


Figure 2.13: Surface produced for $\tau = 400$ from CThead data set

As expected marching cubes takes longer than all the other methods due to its brute force traversal of the volume. The advantage it does offer is the fact that the traversal is straight forward to implement, and it requires no pre-processing of the volume, and therefore no extra storage requirements for data organisation.

Marching cubes can be directly compared to the octree method, and as can be seen from testing on various data sets (Table 2.4) the octree method offers a substantial saving in time during isosurfacing, at the expense of a pre-processing step. In general the time taken for the octree method to pre-process the data, and perform isosurfacing is roughly equivalent to the time taken for marching cubes to extract an isosurface. Therefore for any more than one isosurface extraction it is beneficial to use the octree method. The space required by the octree method never exceeds the size of the data set.

Both methods are compared to the indexing method for data which is 8 bits in resolution. As can be seen from Table 2.4 the pre-processing step takes approximately double the time the octree method requires, and produces an index which is comparable in size to the octree (in

[1]Obtained from the University North Carolina, Chapel Hill
[2]From AVS^{TM} by Advanced Visual Systems Inc.

Data Set	Marching Cubes	Octree Method	Indexing Method	Chain of Cubes
CT head 12 bit				
Pre-processing (secs)		12.22		
Surface production (secs)	28.60	13.47		
Space required (MBytes)		6.3		
CT head 8 bit				
Pre-processing (secs)		11.87	19.31	
Surface production (secs)	25.45	12.12	11.68	
Space required (MBytes)		3.1	1.6	
MR brain 12 bit				
Pre-processing (secs)		11.85		
Surface production (secs)	38.32	23.53		
Space required (MBytes)		6.1		
MR brain 8 bit				
Pre-processing (secs)		11.32	22.18	
Surface production (secs)	35.98	23.98	22.85	
Space required (MBytes)		3.0	3.6	
Hydrogen				
Pre-processing (secs)		0.43	0.97	
Surface production (secs)	0.75	0.23	0.22	
Space required (MBytes)		0.11	0.84	
Lobster				
Pre-processing (secs)		0.81	1.91	
Surface production (secs)	2.53	1.79	1.67	
Space required (MBytes)		0.21	0.62	
Cube				
Pre-processing (secs)		1.60	3.94	
Surface production (secs)	3.43	1.65	1.53	2.18
Space required (MBytes)		0.43	3.17	
Sphere				
Pre-processing (secs)		1.65	4.50	
Surface production (secs)	3.27	1.47	1.37	1.9
Space required (MBytes)		0.43	3.12	

Table 2.4: Results of testing on various data sets

Figure 2.14: Surface produced for $\tau = 600$ from MRbrain data set

Figure 2.15: Surface produced for $\tau = 132$ from lobster data set

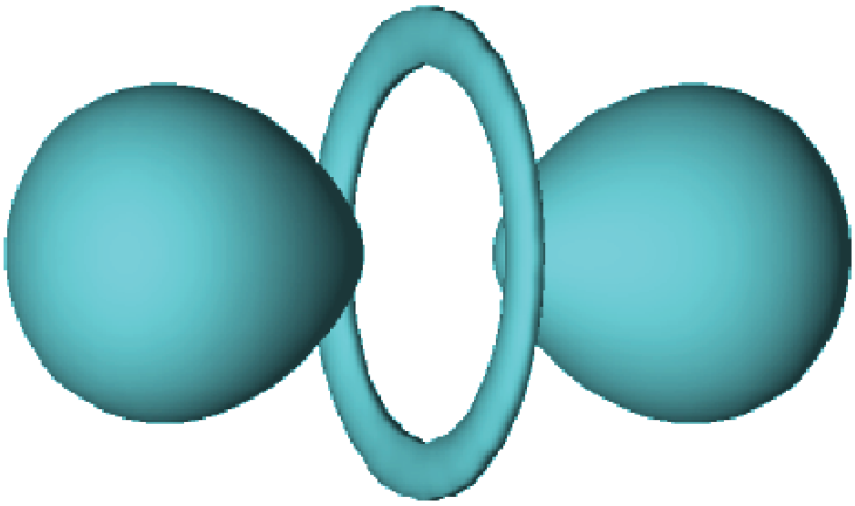


Figure 2.16: Surface produced for $\tau = 128$ from hydrogen data set

some cases smaller and others greater). The execution time for isosurface extraction is then less than the octree method. This is due to the fact that the octree method considers some cubes that do not contribute a surface tile during the isosurface process, whereas the indexing method considers only cubes that do. It is beneficial to use the indexing method when a simple surface shell is required which can be used for fast display or for certain operations such as cutting [24]. This benefit is a result of the fact that to produce such a shell, only the index needs to be considered, and not the volume data itself.

Finally the chain of cubes method was tested on two data sets suited to its operation, data that contains a cube in one case, and a sphere in the second. Given an initial start cube the chain of cubes method extracts the whole of the connected surface without the need for pre-processing the data, or extra storage requirements. Although faster than marching cubes, it is still not as fast as the octree or index methods since extra computation is required to track neighbours across the surface. It is suited to situations where data is computed from some function, since it would only need to compute the data at straddle points and not throughout the volume as the other methods require. This benefit was not tested, but results of testing can be found in the work of Zahlten and Jürgens [23].

2.7 Conclusion

In this chapter the process of surface tiling has been described. After defining the criteria that a tiling method should fulfil (Section 2.2), the method by which the tiling case table is defined, is given (Section 2.2.2). A straightforward, brute force algorithm has been introduced (Section 2.2.3) and the ambiguous cases have been identified and rectified (Section 2.2.4). Section 2.3 gave a review of alternative methods to disambiguate tiling cases, methods that reduce the number of triangles in the mesh, and methods that produce higher order surfaces. Several improved algorithms (Section 2.4) have been illustrated and one original method - the indexing method (Section 2.5) - has been presented. Their operation and implementation

have been discussed, and a comparison has been made using several well known data sets (Section 2.6). From this comparison it can be seen that for simplicity, marching cubes should be used. The chain of cubes method should be used where the data is computed as needed, and the octree method used if more than one isosurface is required from the same data set. The indexing method can also be used in that situation, and has the advantage that the list of cells for a certain threshold are exactly those cells that contain surface tiles for that threshold, and therefore other operations that require such a list can be carried out without the need for a costly referal to the data.

Chapter 3

Volume Rendering

3.1 Introduction

Volume rendering offers an alternative method to surface tiling as described in Chapter 2, for the investigation of three dimensional data. Surface tiling can be regarded as giving one particular view of the data set, one which just presents all instances of one value – the threshold value. All other values within the data are ignored and do not contribute to the final image. This is acceptable when the data being visualised contains a surface that is readily understandable, as is the case when viewing objects contained within the data produced by CT scans. In certain circumstances this view alone is not enough to reveal the subtle variations in the data, and for such data sets volume rendering was developed [25, 26, 27, 28]. In this chapter the underlying technique employed by volume rendering is given in Section 3.2 presented with the aid of a volume rendering model introduced by Levoy [25]. Section 3.3 examines various other volume rendering models and the differing representations they give of the same data. A more efficient method for sampling volume data is presented in Section 3.4 and acceleration techniques are covered in Section 3.5. In Section 3.6 a thorough comparison is made of many of the more popular techniques with the more efficient method of Section 3.4. A review of existing methods, systems and techniques is given in Section 3.7.

3.2 Volume Rendering

3.2.1 Introduction

Many of the three dimensional data sets that need to be visualised contain an interesting range of values throughout the volume. By interesting, it is meant those parts of the volume to which the viewer's attention must be drawn in order for the viewer to gain insight to the physical phenomena the data represents. If the range of values is small, as for example the visualisation of the human skull from CT scans, then a surface tiling method will suffice (Chapter 2). Most data sets do not fall into this category, but rather have a larger range of values or several different values which need to be represented in the visualisation. Such data sets need a method which can display the volume as a whole and visualise correctly those

data values in which the viewer is interested.

The method known as volume rendering or direct volume rendering allows just that by rendering (visualising) the whole of the volume according to some definable criteria which describes those data values that are interesting, and how they should be dealt with.

3.2.2 Classification of data

In a typical data set every voxel contains some material property such as object density, probability, pressure or temperature. These data values have been measured or calculated in the *volume generation* stage. Visualisation of the data set will be based upon an interpretation of these values, which is defined by those values that are of interest, and how they will influence the overall visualisation. The data is converted from the values originally contained within the sampled volume into data in which each value indicates the importance of the corresponding voxel in the volume. These values are usually scaled linearly between the maximum and minimum representable by the number of bits available for each voxel. For example for an 8 bit representation, points with a value of zero are uninteresting, and points with a value of 255 are most interesting, with all other values scaled linearly between these. These values are used to compute opacity for each voxel by normalising them to lie between 0 and 1. An opacity of zero indicates the point is transparent, and therefore will not be seen, and a value of 1 is opaque, and will not only be seen, but will *obscure* points behind it. All other values are semi-transparent and will be seen with varying degrees of clarity. Each voxel is also given a colour, again based upon its value and/or position, which is combined according to opacity with all voxels behind it during the visualisation.

This process of converting the original data values into values of opacity is known as the classification process. It can be carried out with the use of a lookup table (LUT) using the very simple algorithm below.

for $k = 1$ **to** z
  for $j = 1$ **to** y
    for $i = 1$ **to** x
      $voxel'[i, j, k] = LUT[voxel[i, j, k]]$

The LUT will either have been defined by some expert who can decide between those values that are interesting and those that are not, or defined as the result of interaction. The user will adjust the values in the look up table according to the images produced in order to direct the visualisation of the data based upon what information needs to be gathered from the data and what view is desired of the volume.

Usually a more advanced classification process has to be carried out. In the case of data which has been generated by a medical imaging device, several stages of preprocessing may be carried out before the final grey level volume is created. This preprocessing consists of some or all of the following steps:

- filtering of original images
  The original images can be spatially filtered to remove noise using techniques such as median, modal and k–closest averaging filters. Other filters, could be used to detect and enhance edges [29, 30, 31].

- interpolation of images to create the 3D volume of grey level values
  Often the data is of a much higher resolution in the x and y image axes compared to the number of images (z axis). In some cases the resolution can differ by a factor of 20 [4]. If a regular volume is required which has equal dimensions in each axis, new slices must be interpolated between those already known [32, 33].

- segmentation of volume
  In medical applications it is desirable to display the different objects contained within the data separately, or to attach a different attribute, such as colour, to each object. In order to display, for example, the spinal column, from a set of CT scans, the object in question has to be located and separated (segmented) from the surrounding objects. There is no automatic segmentation algorithm that works in all cases [34, 35] although interactive segmentation has proved to be successful [36, 37, 38, 39].

  A typical segmentation process involves the user locating within a slice a seed point in the object they are interested in, from which a region is grown using thresholds. The thresholds define a range of values which encompass the object. The result of the region growing can be viewed in 3D, and parameters can be adjusted to correct the region interactively. Commonly regions that are incorrectly connected can have the *bridges* eroded, and regions that should be connected can be merged together using dilation. Currently segmentation is being used successfully to identify all parts of the human body (Section 3.7.9).

- opacity classification
  Often the straightforward mapping of value to opacity does not result in an accurate display of surfaces contained within the volume. In medical imaging, visualisation often has to display the interface between surfaces such as air/skin, skin/muscle and muscle/bone. These surfaces can be best visualised by setting the opacity according to a function of the gradient of the data [25]. Using this method the gradient is calculated using Equation 3.3 (page 37) and the opacity is assigned accordingly, such that voxels in the vicinity of the object interfaces (high gradients) will have high opacity.

The next step is to determine the representation of the volume. The image comprises of a regular grid of points, each of which can be assigned a certain colour. The image is calculated as a representation of what would be seen from a certain viewpoint, looking in a particular direction, under certain conditions. With the viewpoint and direction known a ray originating from each pixel in the image can be traced through the object domain.

The light and shade of each pixel is calculated as a function of the intensity along the ray originating from that pixel. The value of each pixel is governed by how the volume rendering model interprets the intensity profile and thus visualisation of the data is determined by the model.

The continuous intensity signal along the ray is reconstructed by taking samples at evenly distributed points on the ray and it is these samples that are evaluated by the model.

3.2.3 Levoy's rendering model

The rendering model popularised by Levoy [25] is that of assuming the volume to be back lit by a uniform white light. The light passes through the volume and is attenuated by the opacity it encounters on the path it tracks through the volume. The resulting light that reaches the image is the light that exits the volume. Each pixel is also given a colour which is determined by the *colour* of each sample along the ray. The colours are chosen in a colour classification process which is similar to the opacity classification process, except the colour is a triple of red, green and blue. The colour at each sample can also depend upon the lighting model chosen as will be described later.

Each voxel value in the volume is given by the function

$$f : \mathbb{R}^3 \Rightarrow \mathbb{R} \tag{3.1}$$

where $f(x) =$ measured value at x if $x \in \mathbb{N}^3$
otherwise $f(x) =$ value trilinearly interpolated from the 8 closest neighbours of x.

The quadruple $< r, g, b, \alpha >$ at each data point is calculated from the lookup table for the value at those points.

$< r, g, b, \alpha > = \mathrm{LUT}(f(x))$

where r, g, b, and α are the red, green, blue and opacity components respectively.

A function

$$g : \{r, g, b, \alpha\} \times \mathbb{R}^3 \Rightarrow \mathbb{R} \tag{3.2}$$

is defined such that
$g(\alpha, x) =$ the opacity at the point x trilinearly interpolated from its 8 closest neighbours.
$g(r, x) =$ the red at that point, and so on.

For each pixel $\mathbf{p}$ in the image, where $\mathbf{p} = < i, j >$ i=1,...,I_x, j=1,...,I_y, where I_x and I_y are the dimensions of the image in the x and y axes, a ray $\mathbf{R_p}$ is traced into the object domain. If the ray intersects the volume of data, the length of the ray passing through the volume will be $l =\mid \mathbf{R_p} \mid$. If the ray is sampled at intervals of τ, there will be $K = \frac{l}{\tau}$ samples, each given by $\mathbf{R_p(n)}$ where $n = 1, ..., K$. $\mathbf{R_p(1)}$ is the ray entry point and $\mathbf{R_p(K)}$ is the last sample before the ray exits the volume. The colour and opacity at each sample is given by the function g, and for the n^{th} red sample would be $g(r, \mathbf{R_p(n)})$. If attenuation of a back light with colour Background $= < B_r, B_g, B_b >$, is assumed as the rendering model, the resulting pixel colour triple $< p_r, p_g, p_b >$ is given by the final value of $< C_r, C_g, C_b >$ where for each pixel $\mathbf{p}$

$C = Background$
for $n = k$ **downto** 1 **do**
  $opacity = g(\alpha, R_p(n))$
  $C_r = (1 - opacity) \times C_r + opacity \times g(r, R_p(n))$

$C_g = (1 - opacity) \times C_g + opacity \times g(g, R_p(n))$
$C_b = (1 - opacity) \times C_b + opacity \times g(b, R_p(n))$
endfor
$p =< C_r, C_g, C_b >$

This algorithm contains no view dependent visual cues which may aid the understanding of the visualisation. These can be added in the colour accumulation stage by calculating the spatial point of each sample and applying a shading operator to it before compositing that colour. The simplest shading operator is to depth cue the data. The process of depth cueing involves calculating the distance z, from the ray origin (pixel) to the sample point. The colour at the sample point is then attenuated according to some function based upon that distance. The simplest function is to multiply the colour intensities by $\left(1 - \frac{z}{Z}\right)$ where Z is the maximum distance in the scene.

It is also possible to apply a shading technique such as Phong's, by determining a normal to the data at the sample point which can be used in a similar way to a surface normal. The normal can be calculated by trilinear interpolation from the normals of its 8 closest voxel neighbours, whose normals are calculated using difference operators.

For voxel x, y, z in a data set where each voxel is a unit cube, its normal G is calculated using,

$$\begin{aligned} G &= (g_x, g_y, g_z)\,, \\ g_x &= f\,(x+1, y, z) - f\,(x-1, y, z)\,, \\ g_y &= f\,(x, y+1, z) - f\,(x, y-1, z)\,, \\ g_z &= f\,(x, y, z+1) - f\,(x, y, z-1)\,. \end{aligned} \tag{3.3}$$

The result of shading is a value between 0 and 1 which can be used to multiply the colour for a sample before being composited.

3.2.4 Results

Using this algorithm and suitably defined classification functions, images such as that of Figures 3.1 and 3.2 are created.

But what is the intended interpretation of these images? Firstly the classification functions must be examined. In the case of Figure 3.1, values below 50 have been set to be transparent (that is $\alpha = 0$). Values between 50 and 100 have been set to $\alpha = 0.15$ and the colours set to $< r = 1.0, g = 1.0, b = 1.0 >$, values between 100 and 200 set to $\alpha = 0.15$ and the colours set to $< r = 1.0, g = 0.0, b = 0.0 >$ and above 200, $\alpha = 0.8$, $< r = 0.0, g = 0.0, b = 1.0 >$. The image indicates that values above 200 (up to 255) occur in two distinct ellipsoidal sections, and values between 50 and 200 occur in two larger ellipsoids, and a torus around the centre. In this case the interpretation is correct. The lighting model has given the eye the required cues to determine the *shape* of the volumes contained within the data, and the colouration and opacity have allowed the three separate ranges to be distinguished.

In the case of the CT image (the data having values -1117 to +2248), values below -300 have been set to be transparent and values between -300 and 50 set to $< r = 1.0, g = 0.79, b =$

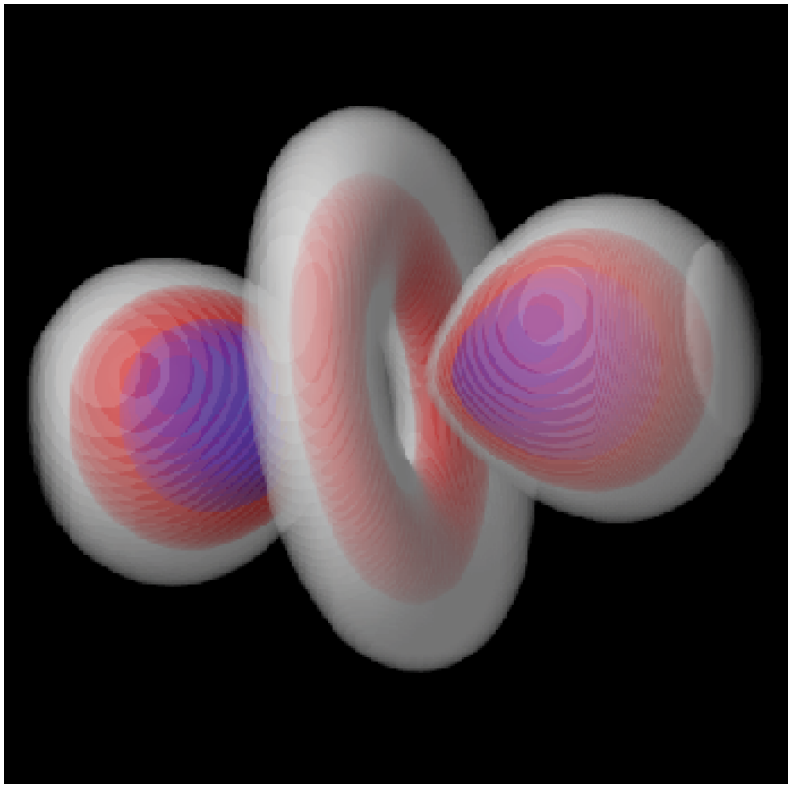


Figure 3.1: Volume rendering of AVS Hydrogen data set.

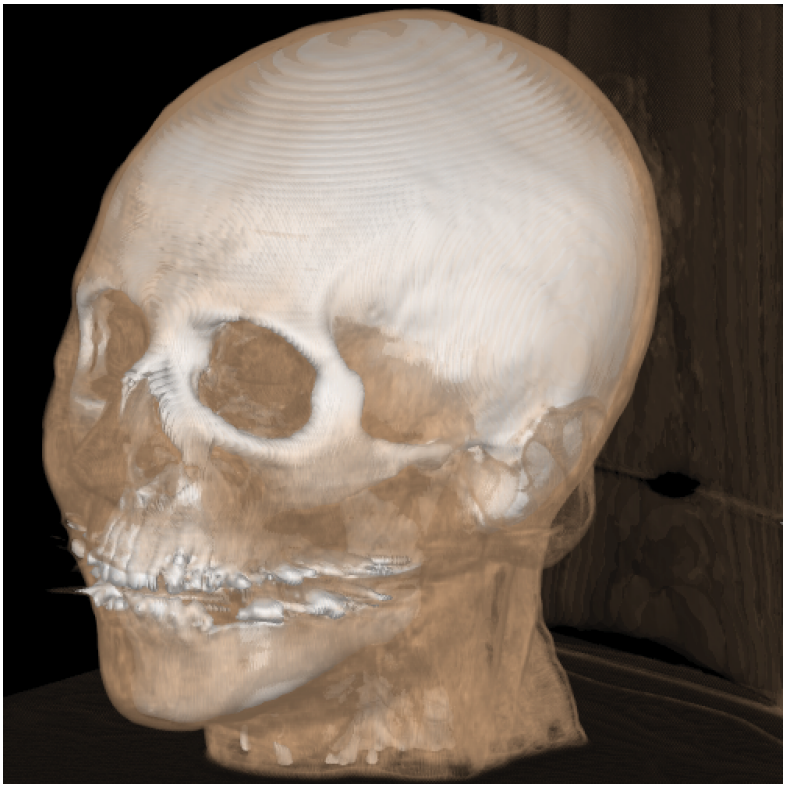


Figure 3.2: Volume rendering of CT head data set.

$0.6 > \alpha = 0.12$. These values contain the range for the skin in the CT images. The other range we are interested in are those values that produce the skull, 300-2248, and therefore these are set to $\alpha = 0.8$ and $< r = 1.0, g = 1.0, b = 1.0 >$. The interpretation is also correct in this case, the skin is mostly transparent, thus revealing the skull contained within. The lighting functions provide enough information to understand the surfaces – both exterior (skin) and interior (skull).

3.3 Volume Rendering Models

3.3.1 Maximum value images

For many applications specific viewing models other than the one mentioned in Section 3.2.3 are employed to present visualisations. The simplest is the widely used maximum value visualisation[1] [16]. For this approach the maximum sampled value along a pixel's ray is the value used for that pixel's colour. This method avoids the need for expensive shading calculations and composition operations, and therefore requires less computation than using a model such as Levoy's. The method is generally used to produce visualisations in which the data is rotated, so that the visualiser can identify areas of high value (hot spots). This type of visualisation results in images not unlike x-rays, and therefore is particularly useful in the medical profession where the users of such methods are familiar with x-rays.

The main problem with this method is the fact that images 180^o apart are identical. For animations of rotating objects this produces the effect of the object seeming to turn back and forth rather than rotate. Depth cueing can be employed to avoid this problem by attenuating the image according to the depth at which the maximum value occurs, and indeed, animations produced using depth cueing provide far more depth information than without.

3.3.2 X-Ray images

The maximum value images of Section 3.3.1 produce images that look like x-rays. X-ray images of the data can be produced by simulating how the rays pass through the volume, calculating the absorption due to the density of the volume. If the density, ρ, along each ray from 0 to l is

$$\rho = f(t), 0 \leq t \leq l, \tag{3.4}$$

the mass of material along the path of the ray is

$$\int_0^l f(t)dt. \tag{3.5}$$

This can be calculated quite simply using either Simpson's Rule (with three ordinates):

[1] Personal communication with K. S. Sampathkumaran of Washington, USA. in which he stated that for medical applications maximum value projection gave a valuable insight to the hotspots within the data

$$\int_0^l f(t)dt \approx \frac{1}{3}d\left(y_0 + 4y_1 + y_2\right), \tag{3.6}$$

or the Trapezium Rule:

$$\int_0^l f(t)dt \approx \frac{1}{2}d\left(y_0 + 2y_1 + \ldots + 2y_{n-2} + y_{n-1}\right), \tag{3.7}$$

where d is the distance between each sample. As $d \to 0$ the approximation becomes more accurate.

The image displayed is a function of the material density – for example, the inverse of the material density, and normalised to use the full range of brightness of the display. The result is an image which looks similar to an x-ray.

3.3.3 Standard model without normal shading

The main computational expense with the standard model is the shading calculation done at each sample position within the volume. If this could be removed, the amount of computation required can be dramatically reduced. One way would be to completely disregard shading operations and display the colour and opacities directly. This results in an image generated rapidly that contains no shading cues but does contain enough information to identify interesting areas within the data set.

Alternatively the expensive calculation of data normals can be disregarded, with the data being shaded using depth cueing. This gives the same look upon the data as the method without any shading at all, and in addition provides some feeling of the distance of the objects within the data. The drawback to both methods is the fact that valuable curvature information is not present in the images they produce.

3.3.4 Sabella's method

A major contribution to the subject of producing alternative images from 3D data sets is that of Sabella [26]. The volume data is treated as a varying density emitter, where the density is a function of the data itself. Sabella regards this to be an equivalent model to a particle system where the particles are sufficiently small. Rays originating from each pixel are traced into the volume data set as described in Section 3.2 and the volume is sampled as before. Sabella's viewing model calculates four parameters – the peak value occurring along the ray, and the distance where it occurs, the attenuated intensity, and the centre of gravity of the field along the ray. A combination of three of these four parameters are then mapped to the Hue, Saturation and Value (HSV) model.

The peak value and distance are self explanatory, whereas the attenuated intensity needs more explanation. It is taken to be the brightness, B, along the ray, where

$$B = \int_{t_1}^{t_2} e^{-\int_{t_1}^{t} \tau p^{\gamma}(\lambda) d\lambda} p^{\gamma}(t)\, dt \tag{3.8}$$

The variables and expressions of this equation are described in [26].

The term $\int_{t_1}^{t} p^{\gamma}(\lambda)\, d\lambda$ represents the number of particles in the volume along the ray that spans between t_1 and t, where

t_1 and t_2 are the ray entry and exit points of the value data,
t is the ray parameter, $t_1 \leq t \leq t_2$,
p is the density, and γ is a constant correction factor.

The term $e^{-\int_{t_1}^{t} \tau p^{\gamma}(\lambda) d\lambda}$ could be regarded as the transparency for simplicity, and is in fact proportional to transparency. The integral therefore calculates the contribution to the final intensity of particles along the whole ray, and the attenuation of the light due to the density of the particles.

It is shown [26] that equation 3.8 is the continuous form of the discrete equation

$$\sum_{i=1}^{n} b_i \prod_{j=1}^{i-1} \theta_j \tag{3.9}$$

where b_i can be considered as the brightness at sample i, and θ_j is the transmittance (that is transparency) of sample j, $0 \leq \theta_j \leq 1$. The term $\prod_{j=1}^{i-1} \theta_j$ calculates the attenuation due to all the samples *in front* of sample i, and therefore the product $b_i \prod_{j=1}^{i-1} \theta_j$ is the contribution of sample i to the intensity. The sum of all contributions gives the brightness, B.

A practical method of computation can be derived from this equation:

$B = 0.0$;
$o = 1.0$;
for $i = 0$ **to** *length* **do**
  $B = B + b_i \times o$
  $o = o \times \theta_i$
endfor

The peak value parameter is mapped to Hue, and the attenuated intensity is mapped to Value. Either the centroid, or distance is mapped to the saturation parameter. This has the effect of allowing colour to represent the maximum values, and the distance parameters to give depth information in the form of saturation, leading to what can most easily be described as fog. The attenuated intensity parameter gives an impression of the distribution of the data.

3.4 Efficiency Aspects of Volume Rendering

The volume rendering method as described in Section 3.2 is a costly process to compute due to the number of samples that have to be determined. In this section a new method which reduces this computation is presented. The method has been developed as a result of analysing

where work is done in the volume rendering process. This section introduces the idea of choosing the distance at which samples are made along the ray such that computation can be reduced. The method requires a bare minimum of precalculation (a few tens of mathematical operations), and works without constraints for arbitrary view points. In Section 3.4.1 the calculations required by the volume rendering process are examined. In Section 3.4.2 the process of choosing the sampling distance to reduce the number of calculations is introduced, and in Section 3.4.3 the effect this has on computational time and the images produced is investigated. Section 3.4.4 outlines some view artifacts this method introduces during animation loops, and suggests how these may be overcome. Section 3.4.5 offers conclusions for this method.

3.4.1 Computation involved during volume rendering

The algorithm has been sufficiently described elsewhere [25] for implementation, but in each case, although the spacing for the sampling is described, the value is left up to the reader. In the majority of cases a value of 1 unit will be chosen for simplicity, where each voxel in the data set has length 1 unit in each dimension. Figure 3.3 indicates the situation.

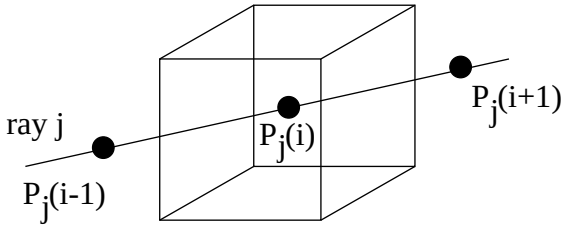


Figure 3.3: Ray passing through cube sampled at even intervals.

The spatial position of the i^{th} sample along the j^{th} ray is given by $p_j(i)$. The value which is sampled at that point is given by $S_{p_j(i)}$ where the sample is calculated by trilinear interpolation from the 8 neighbouring voxels that make up the cube that contains the point $p_j(i)$. The sampled value is usually the interpolation of the value and normal from each of the 8 neighbouring voxels [40] or the interpolation of the opacity and colour components, and normal from each of the 8 neighbouring voxels [25].

In order to calculate the normal of the sample $S_{p_j(i)}$ in Figure 3.3, the normals at each voxel $v_0, ..., v_7$ must be determined using an appropriate method, such as central differences. This involves the calculation of 8 central differences, each of which involves 3 subtractions, 6 voxel look ups (Equation 3.10) and one normalisation step.

$$\begin{aligned} G &= (g_x, g_y, g_z)\,, \\ g_x &= f\,(x+1, y, z) - f\,(x-1, y, z)\,, \\ g_y &= f\,(x, y+1, z) - f\,(x, y-1, z)\,, \\ g_z &= f\,(x, y, z+1) - f\,(x, y, z-1)\,. \end{aligned} \tag{3.10}$$

The normalisation step (Equation 3.11) involves 2 additions, 3 multiplications, 1 square root

and 3 divisions.

$$\begin{gathered} l = \sqrt{g_x^2 + g_y^2 + g_z^2} \\ g'_x = \frac{g_x}{l} \; g'_y = \frac{g_y}{l} \; g'_z = \frac{g_z}{l} \end{gathered} \tag{3.11}$$

To calculate the normals at each of the 8 cube vertices requires a total of 16 additions, 24 subtractions, 24 multiplications, 24 divisions, 8 square roots and 48 voxel look ups.

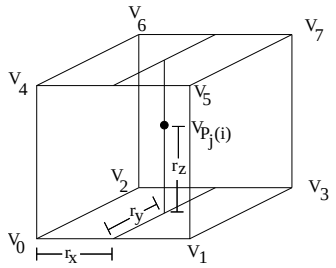


Figure 3.4: Trilinear interpolation within cube.

If the offsets in the cube for each axis are r_x, r_y and r_z (Figure 3.4), the values at $p_j(i)$ are calculated using trilinear interpolation as follows :

$$\begin{aligned} t_0 &= (v_1 - v_0)r_x + v_0 \\ t_1 &= (v_3 - v_2)r_x + v_2 \\ t_2 &= (t_1 - t_0)r_y + t_0 \\ t_3 &= (v_5 - v_4)r_x + v_4 \\ t_4 &= (v_7 - v_6)r_x + v_6 \\ t_5 &= (t_4 - t_3)r_y + t_3 \\ v_{p_j(i)} &= (t_5 - t_2)r_z + t_2 \end{aligned}$$

which involves 7 subtractions, 7 additions, and 7 multiplications for each value that must be interpolated and a total of 8 voxel look ups. Furthermore, if a value such as opacity is required, a further classification table look up is necessary. If the value being interpolated is the normal, each v_i is now a triple and the calculations are therefore trebled. Also each step must be normalised, so a further 7 normalisation calculations are required. A summary of the total required computational operations is given in Table 3.1.

operation	normal	rgbα	value
$+$	51	28	7
$-$	45	28	7
$\div$	45		
$\times$	66	28	7
$\sqrt{}$	15		
voxel l. u.	48	8	8
table l. u.		32	

Table 3.1: Calculations required for trilinear sampling process.

As can be seen from the table, normal interpolation is the most expensive operation, particularly with its 15 roots, and 45 division operations. When this is put into context with the operation of the algorithm, it is realised that for a large image (500×500) and a large data set ($256 \times 256 \times 256$) this normal interpolation can be done up to a maximum of 100 million times. (Number of rays $\times$ the maximum ray length). It is this figure that lead to the investigation of ways to reduce the calculation.

One immediate solution is to precompute the normals at each of the voxels in a preprocessing step. The problem with this is the extra storage required – for a data set of $256 \times 256 \times 256$ an additional 200MBytes of storage is needed (12 Bytes for a normal per voxel). As a result of the usual problems of memory size, disk size and swapping it is far better for them to be calculated on the fly. In reality far less normals are computed during image computation since not all voxels contribute to the final image when adaptive termination is used. In an experiment with the CThead data of 7 million voxels, only 2 million voxel normals were computed.

3.4.2 Choosing the stepping distance

By choosing the stepping distance to be 1 unit, the samples may occur anywhere within the data set, that is to say, there are no means by which the computational cost can be reduced by taking advantage of where sample positions occur. It can be seen that since these samples can occur anywhere within the cube, trilinear interpolation for the values must be used. If the samples were to occur within the face of each cube, then only bilinear interpolation would be required. This reduces the need to know the normals at all 8 neighbouring voxels to just needing to know them at the 4 face neighbouring voxels. The stepping distance can be chosen so this situation occurs by calculating how far along the ray must be travelled to cross between successive cube faces parallel to a particular axis. By starting the ray on that face boundary, it is ensured that by using the correct step distance, each sample will be in a face, and therefore only bilinear interpolation is required.

The stepping distance is calculated by determining which axis the ray travels fastest in – that is for a given length of ray which axis it will travel furthest in. The rays are then adjusted to start in the closest face perpendicular to that axis, and then each sample will occur in a face perpendicular to the axis. Face interpolation functions are used depending upon in which face interpolation has to take place (perpendicular to the x, y or z axis). The ray is adjusted simply by scaling the stepping distance of the *fast voxel traversal* algorithm of Amanatides and Woo [41] so that the quickest axis has a stepping distance of 1 (2 division operations), and the fractional part of the sample point position is altered so that it is zero in the fastest axis, and contains the fractional placement in the other two axes (2 divisions, 2 multiplications, 2 subtractions and 2 additions). The fast voxel traversal algorithm then continues as normal.

The effect this has is two-fold. Firstly it is no longer necessary to calculate the normal from 8 neighbours, but rather from 4, halving the number of central difference calculations. Secondly, bilinear interpolation is used:

$$t_0 = (v_1 - v_0)r_i + v_0$$
$$t_1 = (v_3 - v_2)r_i + v_2$$
$$v_{p_j(i)} = (t_1 - t_0)r_j + t_0$$

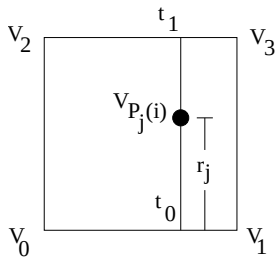


Figure 3.5: Bilinear interpolation within cube face.

Where r_i and r_j are the offsets along the two axes (Figure 3.5). This involves 3 subtractions, 3 additions and 3 multiplication operations for each value that must be interpolated, and a total of 4 voxel look ups. The total computational operations required are summarised in Table 3.2, with the old values in brackets.

operation	normal	rgbα	value
+	23 (51)	12 (28)	3 (7)
-	21 (45)	12 (28)	3 (7)
$\div$	21 (45)		
$\times$	30 (66)	12 (28)	3 (7)
$\sqrt{}$	7 (15)		
voxel l. u.	24 (48)	4 (8)	8
table l. u.		16 (32)	

Table 3.2: Calculations required for bilinear sampling process.

It can be observed that about 50% of the calculation can be saved using this method.

3.4.3 Testing

The effect this has on computational time was tested by implementing the standard volume rendering algorithm using trilinear interpolation and then modifying it to calculate the required interval size, therefore allowing bilinear interpolation to be used. The algorithm was implemented on a DEC 3000 model 400 Alpha workstation and tested on the UNC Chapel Hill CThead and superoxide dismutase electron density map (SOD) data sets, and the AVS Hydrogen data set. Since the new method increases the interval size, a certain amount of time will be saved due to the fact that less samples will be taken along the ray. In order to eliminate this effect the standard algorithm was modified so that it calculated the interval size, and used trilinear interpolation rather than bilinear interpolation (called Jump

in tables). Taking this time into account allows the true saving to be revealed. One final point is that the interval size can vary between 1 when the viewing angle is axis aligned, and $\sqrt{3}$ when the viewing angle is at 45^o to each axis. Therefore testing is carried out at various angles to give differing interval sizes. The results appear in Tables 3.3 and 3.4.

	Standard Ray Termination				Adaptive Termination			
Angle	Standard	Jump	Bilinear	Gain	Standard	Jump	Bilinear	Gain
45,45,45	113.16	69.18	39.10	26.6%	73.18	54.30	40.10	19.4%
0,90,0	111.53	113.78	65.73	41.1%	64.20	65.53	50.50	23.4%
45,0,45	114.46	84.21	47.11	32.4%	73.73	61.93	35.77	35.5%

Table 3.3: Computation times for Hydrogen data set.

	Standard Ray Termination				Adaptive Termination			
Angle	Standard	Jump	Bilinear	Gain	Standard	Jump	Bilinear	Gain
45,45,45	365.85	276.57	141.71	36.9%	96.88	84.40	54.61	35.3%
0,90,0	401.67	448.60	244.36	39.2%	103.23	112.50	74.18	28.1%
45,0,45	344.49	351.67	183.01	46.9%	98.68	100.98	64.68	34.5%

Table 3.4: Computation times for CThead data set.

The normal calculations dominate the volume rendering process, and by reducing the number required and simplifying the interpolation process, substantial savings can be obtained as indicated by the tables. In fact the computation time has been reduced by about 30%–40%. This reduction is to be expected since the amount of computation done during normal calculation is reduced by 50% and also the fact that normal calculation requires a large proportion of the running time when compared to the constant calculations that are made, such as calculating ray entry and exit points to the volume.

It is interesting to note (from the Jump column) that by increasing the step size, computational time also increases in some cases. This is because the step size is not significantly greater than 1, and some extra calculations involved when determining the number of samples outweigh any saving.

The method gives a better gain for the normal ray terminating method because a higher percentage of the computation time is spent calculating normals, for which the method offers a 50% speedup.

Examining the image produced shows very little degradation in quality. This is to be expected since the samples are still being taken at evenly spaced intervals and are taken in such a way that all values along a ray are used in the sampling process. There is no difference between images 3.6 (b) and 3.6 (c), since both have the same interval size. There is a difference between images 3.6 (a) and 3.6 (b) which have been produced with an interval size of 1 and 1.732 units respectively. This difference is caused by samples being taken with larger intervals using the same compositing formula:

$o_i^{out} = (1 - o_i)o_i^{in} + o_i$ for $i = 1, \ldots, n$

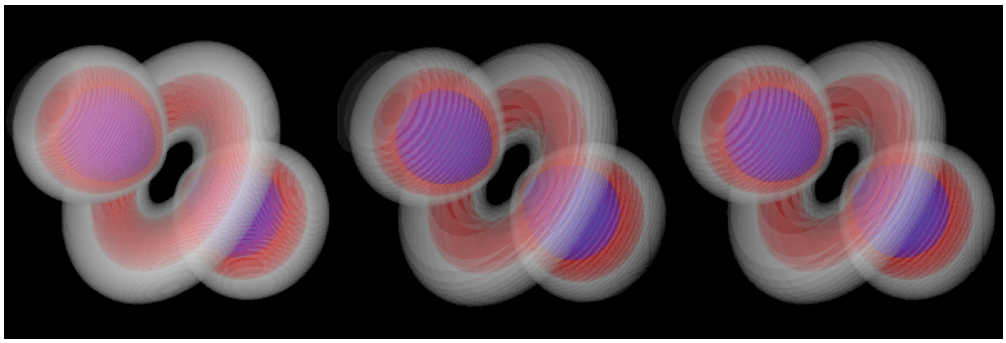


Figure 3.6: (a) Trilinear interpolation, (b) Increased interval size, (c) Bilinear interpolation.

where n is the number of samples, o_i is the opacity at sample i, and o_i^{out} and o_i^{in} are the opacities coming into and going out of i. The interval length does not come into this formula, and therefore in a volume of constant value, n samples with an interval length of m will give the same opacity for n samples with an interval length of 1.

If $m > 1$ the volume sampled with the larger interval size will seem more transparent than the volume traversed with an interval size of 1. This is quite acceptable since the opacity values are a subjective scheme to allow the presentation of the data. For static images there is no problem.

Figure 3.7 shows the resulting images for the CT head test data set.

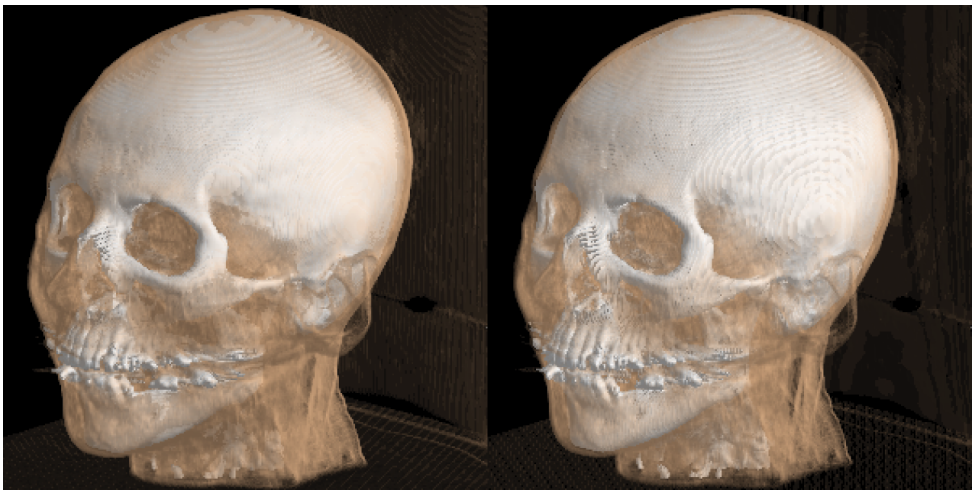


Figure 3.7: (a) Trilinear interpolation, (b) Bilinear interpolation.

3.4.4 Animation

As mentioned in the previous section, increasing the interval size, increases the transparency of the volume. This does not cause problems for static images, but for animation, where the viewpoint from which the data is seen from varies, the interval size is variable and during a rotation, the transparency may be perceived to rise and fall. A practical solution is to raise

each value in the opacity classification table to some power proportional to the step size. Figure 3.8 gives two images, with differing interval lengths, produced using the modified classification table. As can be seen, the transparency of the images corresponds quite closely.

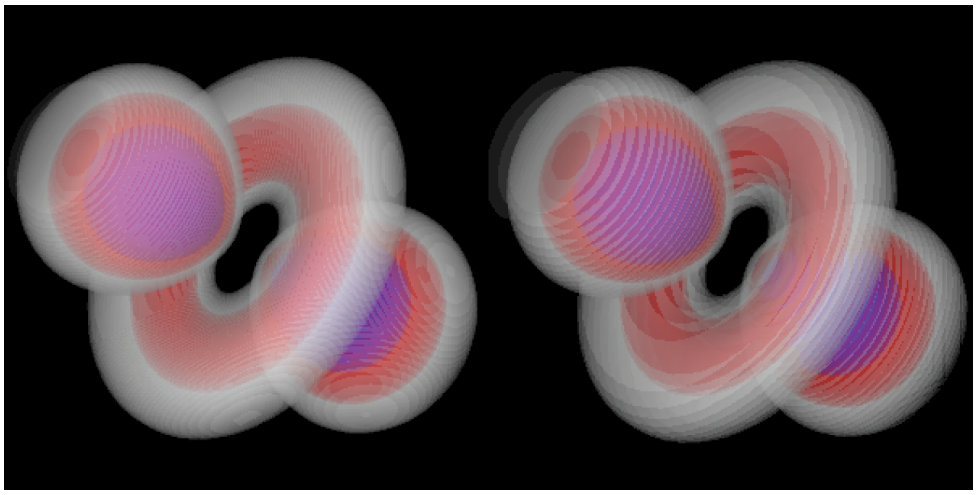


Figure 3.8: (a) Trilinear interpolation, (b) Bilinear interpolation.

3.4.5 Conclusions

By analysing exactly how many operations are carried out during volume rendering, it has been possible to concentrate on a method that reduces these operations by about 50%. This substantial reduction has been achieved through the use of a variable interval size which is dependent upon viewpoint, and is calculated in order to make each sample fall within a cube face. This allows bilinear, rather than trilinear, interpolation to be used, with the associated reduction in computational complexity. This saving has been analysed both qualitively and also through the effect it has on the execution times of the volume rendering process. Since normal calculation is the main burden of volume rendering, a reduction by 50% of the operations required should result in a large reduction in computation time. This reduction is observed to be around 30% and reasons for this were given in Section 3.4.3. The images produced by the method are shown to be good representations of the volume, and problems that arise during animation where the viewpoint changes have been investigated. A practical solution to avoid these artifacts has been given, and testing shows it to be acceptable. This method compares well with other acceleration techniques (Section 3.5) since it gives a similar reduction in time without the large trade off in image quality that other methods suffer. This method involves very little computation to determine the stepping distance, and suffers from no constraints on the viewing direction and view point.

3.5 Acceleration Techniques

The volume rendering process has been identified as being very costly in terms of computation, and many methods exist that accelerate the computation time. This section reviews several

acceleration techniques.

3.5.1 Adaptive rendering

Most of the expense involved in volume rendering is the fact that so many samples are taken along each ray during the compositing process. Since the number of rays is dependent upon image size, the larger the image, the more expensive it becomes to render the data set. Adaptive rendering attempts to reduce the workload by concentrating computation in areas where it is most needed. This technique can be applied quite simply to volume rendering by assuming the resultant image will contain large areas of coherency (colour and intensity). The principle is to ray trace the volume at a low image resolution and by treating four neighbouring pixels which have been ray traced as corners of a square the remaining pixels can be coloured using bilinear interpolation. This coarse image can be regarded as the first image in a sequence of images that progressively become more and more refined. The refinement process takes place by adaptively concentrating on areas with greatest change. If the pixels at the corners of the square vary by more than a given tolerance ϵ, the square is divided into four smaller squares and the new unknown corner points are computed. The process of computing the interior pixels using bilinear interpolation is repeated, and a new image can be displayed. This process is carried out until the corners of the square vary by less than ϵ or the size of the square is one pixel. At this point the image is correct to a tolerance of ϵ. The effect of this method is that computation is concentrated in areas where features change sharply, for example along the edge of an object. By relaxing the tolerance, less pixels are truly sampled, but more image defects become apparent, and so there is a trade off between image quality and speed. This method was also presented in [42, 38]. The image in the middle of Figure 3.9 is a volume rendering of the hydrogen data using the adaptive rendering method with squares of an initial size of 16, and a tolerance of $\epsilon = 0.0275$ (or a difference of 7 intensity values out of 256). The image on the right shows the pixels that were actually ray traced, the rest being interpolated from these. The number of rays traced is 34335, as opposed to 160000 for the image produced by the standard method on the left. Whereas the standard method took 129.4s, the method using adaptive rendering took 53.0s. All images are 400×400.

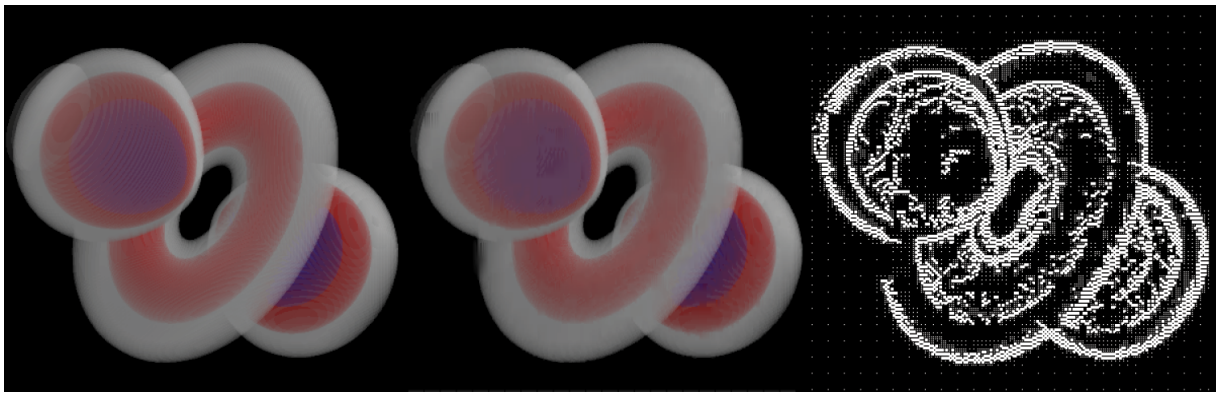


Figure 3.9: The adaptive rendering process.

3.5.2 Template-based volume viewing

In Section 3.4 it was shown that the process of trilinearly interpolating sample positions and function gradients along the ray is the most computationally intensive component of the volume rendering process, thus any method that avoids this will reduce the computation required. The template based method of Yagel and Kaufman [40] calculates a ray template which is a path of voxels through the volume. This template can be moved over the image construction plane in such a way that the volume is sampled uniformly without gaps (Figure 3.10). The ray template indicates which voxels are to be sampled, and these voxels are sampled without trilinear interpolation, with their gradients calculated using central differences. This is valid if it is assumed that the value in the volume does not vary over the cube represented by one voxel, which is not generally the case. The template is constructed so that it is either 6-way or 26-way connected. In this case 26-way connection is chosen because it results in less samples and allows the volume to be sampled uniformly without repetition. The image is formed by projecting and resampling the image construction plane to the desired dimensions.

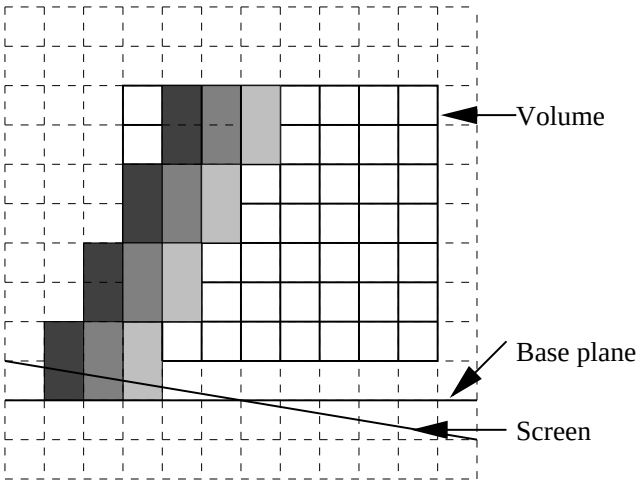


Figure 3.10: Using template to sample the volume.

3.5.3 Adaptive termination

The original volume rendering algorithm composites colour and opacities in a back to front manner. If at any time a fully opaque sample is encountered, the samples composited up to that point have no bearing on the final pixel colour since they are obscured. This is an undesirable situation since expensive interpolation and gradient operations are effectively ignored and do not make any contribution. In order to prevent this, the ray should be traced in a front to back manner so that in the event of an opaque sample, the ray can be terminated, and so no further samples need to be taken along the ray [43].

Using the definitions from Section 3.2.3 the front to back algorithm for each pixel $\mathbf{p}$ is

$C = Background$

$o = 1.0$
$n = 1$
while ($n < K$ and $o \neq 0.0$) **do**
  $C_r = C_r + o \times g(\alpha, R_p(n)) \times g(r, R_p(n))$
  $C_g = C_g + o \times g(\alpha, R_p(n)) \times g(g, R_p(n))$
  $C_b = C_b + o \times g(\alpha, R_p(n)) \times g(b, R_p(n))$
  $o = o \times (1 - g(\alpha, R_p(n)))$
  $n = n + 1$
endwhile

The front to back compositing function is slightly more complicated, but also allows the use of adaptive termination to stop the rendering process when a useful image has been calculated. In the compositing function it is observed that as the accumulated transparency o approaches 0, the sample taken does not contribute significantly to the final pixel colour [43]. In fact the contribution is less than o, so for $o < \epsilon$ where ϵ is some small tolerance level, we can *adaptively terminate* the ray no matter how far it has travelled through the volume. Since the opacity along the ray reaches high values very quickly, when passing through slightly opaque solid matter, the ray is terminated long before it has passed through all of the volume, and thus large amounts of unnecessary computational effort can be saved. If the tolerance ϵ is relaxed rendering time is reduced although the image contains more artifacts.

3.6 Comparison of Methods

For the comparison of all the different methods, three test data sets were used – AVS Hydrogen, University of North Carolina (UNC) CThead, and UNC superoxide dismutase electron density map (SOD). Each test program was written in C on a DEC Alpha 3000/400 workstation using as much common code as possible to make timing comparisons fair. Each data set was rendered from a particular viewpoint under certain conditions for an image size of 400×400 to give the time. Such a large image was chosen in order to allow the differences between the methods to be more marked. The images were compared qualitatively (how they looked). The results for the CThead data are given in Table 3.5, results for the Hydrogen data are given in Table 3.6, and for the SOD data in Table 3.7.

For the CThead (Figures 3.11–3.13) there is no qualitative difference between any of the images using the standard viewing model. From these comparisons it would seem that the bilinear method with adaptive termination is the one that produces the best results quickly for this data set, being over 7 times faster than the method without any optimisations. An animation loop was created for the CThead (see accompanying video) using the adaptive termination technique, with and without the adjustment of the step size to enable bilinear interpolation. Using the normal method, the animation took 2hrs 6mins 12secs to produce, as opposed to 1hr 16mins 6secs with bilinear interpolation. There is no visual difference between the two which would suggest that the bilinear method can be used where high quality accurate images are required using the least computational time.

The template method is by far and away the fastest method – taking only 10 seconds for the

Method	Adaptive Termination	Figure	Time (secs)
Standard	No	3.11(a)	731.37
Standard	Yes ($\epsilon = 0.1$)	3.11(b)	190.01
No shading	Yes ($\epsilon = 0.1$)	3.11(c)	81.09
Template	Yes ($\epsilon = 0.1$)	3.12(a)	10.10
Bilinear	Yes ($\epsilon = 0.1$)	3.12(b)	98.25
Jumps	Yes ($\epsilon = 0.1$)		140.93
Bilinear	No		292.64
Jumps	No		503.63
X-ray	N/A	3.12(c)	97.66
Maximum	N/A	3.13(a)	99.71
Depthmax	N/A	3.13(b)	106.23
Sabella	N/A	3.13(c)	133.61

Table 3.5: Results of different methods using CThead data.

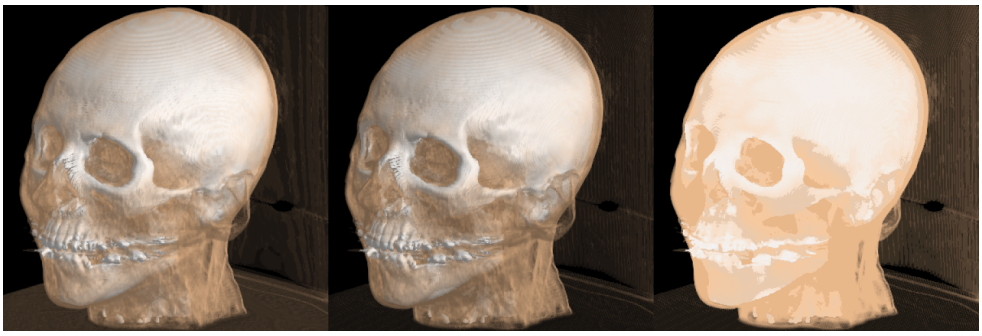


Figure 3.11: (a) Standard method (b) Adaptive Termination (c) No shading.

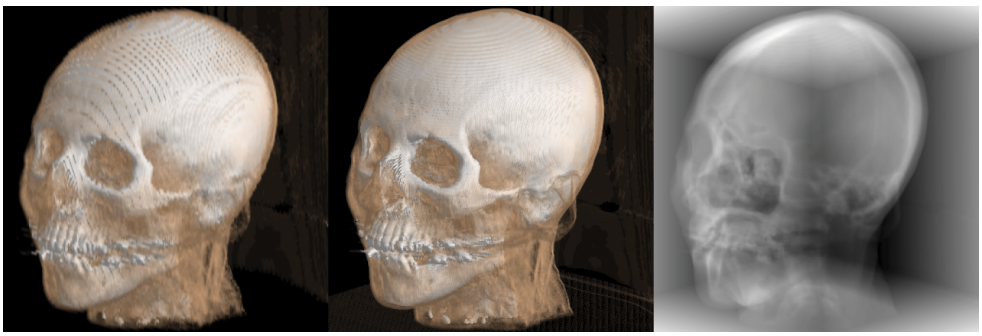


Figure 3.12: (a) Template method (b) Bilinear method (c) X-ray method.

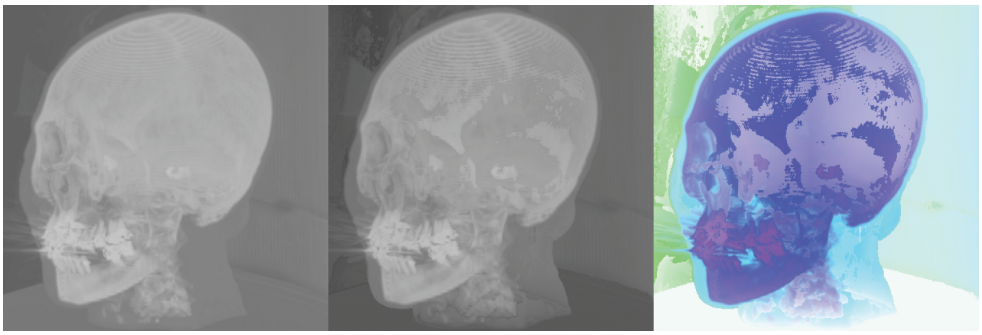


Figure 3.13: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

test data, but produces a very coarse image. The image is adequate to gain a rough idea of the data and is ideal for quick visualisations, since it shows the main features. The problem with this method is that the time is not scalable, that is, it is constant for all image sizes, unlike the other methods, which are scalable. It was found that a 100×100 image created using the bilinear method, and scaled up took half the time of the template method to compute, and produced a comparable image, the only difference being the fact that the bilinear image was slightly smoother than the template image, and contained less sharp delineations of features which had provided good indicators in the case of the template method.

The x-ray, maximum, and maximum with depth cue methods produce alternative views of the data in reasonably quick times. The maximum and x-ray methods produce similar images in much the same time. When using the maximum method to produce animations, the head looks as if it is swinging from side to side, rather than rotating, due to the fact that images 180^o apart are identical. This situation was somewhat improved by using the depth cueing method, and produces a better animation.

Finally the Sabella image of the skull is interesting, but it can only be used to show how the image differs from the standard methods. The image it produces in the case of the CThead serves no real purpose since the method was not intended for use on such data sets.

For the Hydrogen data set (Figures 3.14–3.16) the image produced by the Sabella method shows the data distribution using colour. Fogginess shows depth, and together they convey a lot of information, showing the structure of the data. The image looks good, but it needs effort to interpret the information.

The x-ray, maximum and maximum depth images all look very similar, and show the two *hot spots* and a general fuzziness around the toroidal section.

The only difference between the adaptive method and the standard method is that the adaptive image looks duller because brighter areas to the back of the volume are not contributing, due to the ray terminating. It was found that by making the error bound ϵ, tighter, this effect is reduced, and the time to produce the image increases. It was found that for $\epsilon = 0.05$ the image produced by the adaptive method looks identical to the image produced by the standard method and took 146.28s, a saving of 25%. The template method produces a coarse image which looks very fuzzy. It gives a rough idea of the data spread, but the image is weak and dull. This is due to the fact that less samples are taken along the ray, and therefore less

Method	Adaptive Termination	Figure	Time (secs)
Standard	No	3.14(a)	202.32
Standard	Yes ($\epsilon = 0.1$)	3.14(b)	130.94
No shading	Yes ($\epsilon = 0.1$)	3.14(c)	30.67
Template	Yes ($\epsilon = 0.1$)	3.15(a)	1.95
Bilinear	Yes ($\epsilon = 0.1$)	3.15(b)	55.00
Jumps	Yes ($\epsilon = 0.1$)		104.45
Bilinear	No		69.88
Jumps	No		131.88
X-ray	N/A	3.15(c)	26.76
Maximum	N/A	3.16(a)	25.98
Depthmax	N/A	3.16(b)	29.87
Sabella	N/A	3.16(c)	36.62

Table 3.6: Results of different methods using hydrogen data.

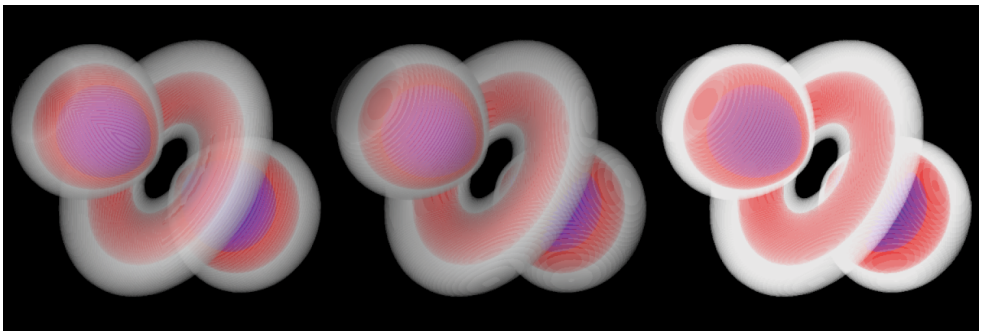


Figure 3.14: (a) Standard method (b) Adaptive Termination (c) No shading.

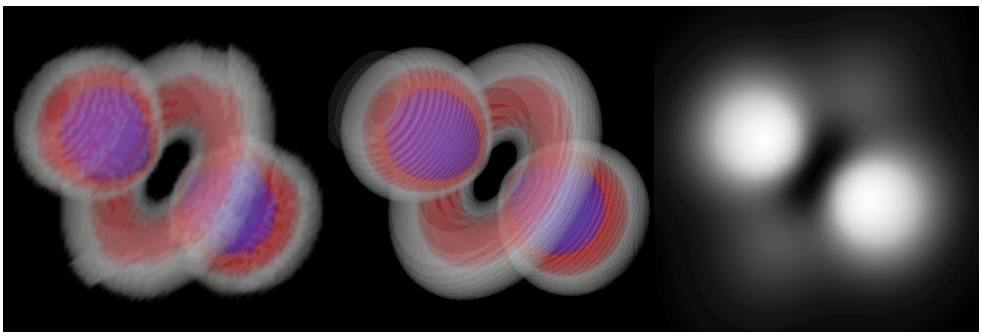


Figure 3.15: (a) Template method (b) Bilinear method (c) X-ray method.

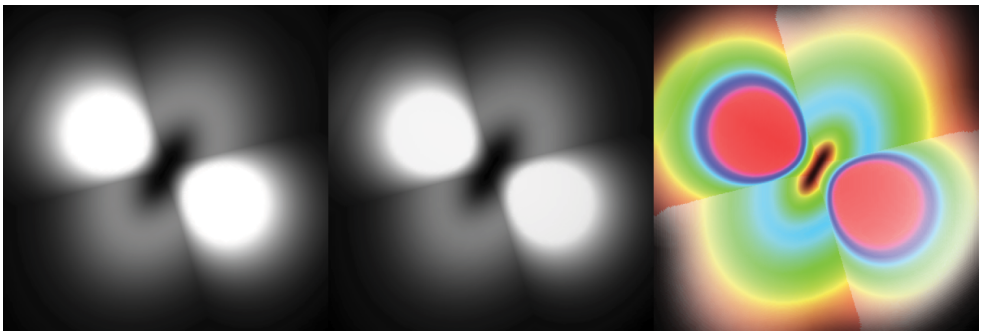


Figure 3.16: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

light is contributed to the final intensity. The method without shading also produces the image quickly and gives a good impression of the data, although the lack of shape information from the shading detracts from the image. The bilinear method produces good clean images, but they are slightly more transparent, and therefore dull, because less samples are taken along the ray. Nevertheless, the images are still very useful.

Method	Adaptive Termination	Figure	Time (secs)
Standard	No	3.17(a)	108.80
Standard	Yes ($\epsilon = 0.1$)	3.17(b)	99.95
No shading	Yes ($\epsilon = 0.1$)	3.17(c)	56.01
Template	Yes ($\epsilon = 0.1$)	3.18(a)	3.37
Bilinear	Yes ($\epsilon = 0.1$)	3.18(b)	51.68
Jumps	Yes ($\epsilon = 0.1$)		73.55
Bilinear	No		49.81
Jumps	No		72.98
X-ray	N/A	3.18(c)	43.79
Maximum	N/A	3.19(a)	42.44
Depthmax	N/A	3.19(b)	47.83
Sabella	N/A	3.19(c)	59.66

Table 3.7: Results of different methods using SOD data.

This problem also occurs with the SOD data (Figures 3.17–3.19), but the bilinear method results in an image that is even more weak and dull. Any impressions of curvature are lost as shading is not enforced over large coherent areas. In this case the method produces useful images, but they do not contain as much information as some of the other models and methods.

The image produced by the template method suffers even more from dullness and is very fuzzy. The method is still useful though, because it is so fast, and could be used to produce animations quickly to gain an overall impression of the data.

The image produced by the standard method with adaptive termination is duller than images produced without, and in this case the saving in time is not so significant, and in conclusion not worth the slight degradation in image quality. A good impression of the hot spots contained

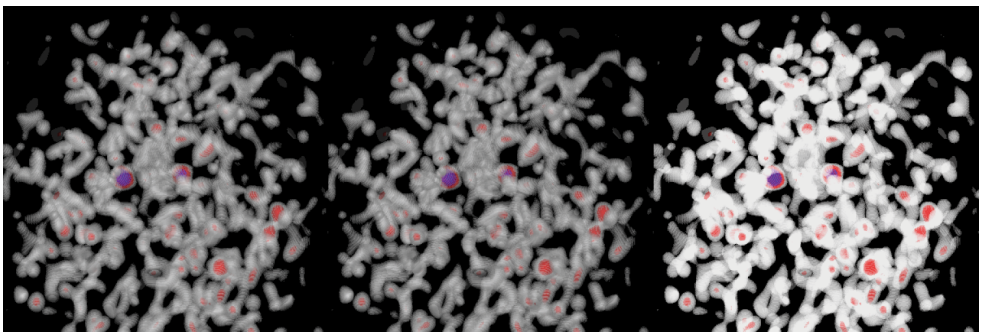


Figure 3.17: (a) Standard method (b) Adaptive Termination (c) No shading.

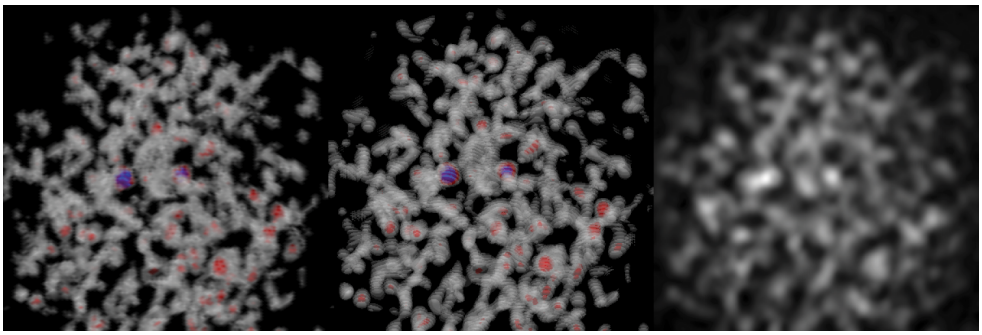


Figure 3.18: (a) Template method (b) Bilinear method (c) X-ray method.

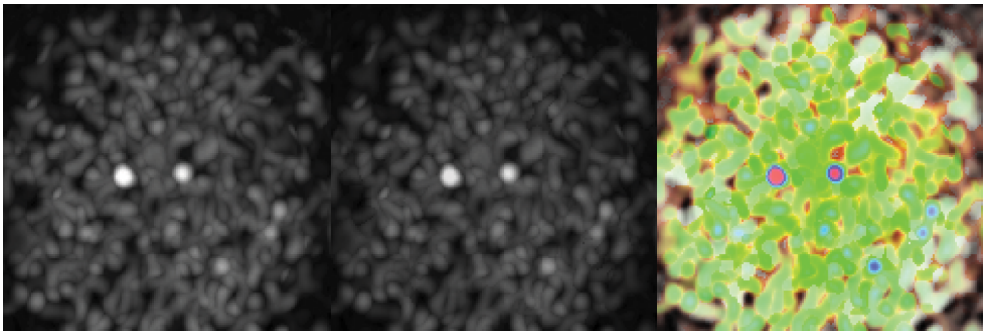


Figure 3.19: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

within the data can be gained through images produced without shading, also through images produced using the x-ray, maximum and maximum with depth cueing methods, which also bring out the blobby structure of the data quite well.

Finally the image produced using the Sabella model produces what is arguably the best image, with a clear indication of hot spots, with a good feel for the lumpy structure of the data, conveying the depth of the data well.

3.7 A Review of Other Volume Rendering Techniques

3.7.1 Introduction

The main viewing models available for the rendering of 3D data sets have been described and compared in Sections 3.2 and 3.3. The more important acceleration techniques have been reviewed in Sections 3.4 and 3.5, with comparative tests made about their speed and image quality in Section 3.6. This section is a review of the other techniques and systems that are available for the visualisation of volume data.

3.7.2 Splatting

One large area that has been focussed on is that of projecting the volume data set onto the image plane to produce the image. This differs from previously described techniques in that rather than a pixel by pixel traversal of the image, and a mapping from image space to object space, the volume is traversed in a cell by cell order with a mapping from object space to image space.

In footprint evaluation [44], Westover suggested projecting each sample within the data set onto the image plane, and *spreading* the energy of the sample according to its *footprint.* The footprint is obtained by applying the view transformation to a generic footprint which can be calculated by assuming that the reconstruction kernal is a sphere. The kernal's footprint is determined, and the energy spread is calculated according to some function, such as a Gaussian. The relative merits of footprints of various sizes (5×5 to 101×101) are described in the paper. The advantage of this method is the fact that each sample is projected onto the image and rays no longer have to be traced into the volume to integrate samples to find their contribution. This allows the application of parallel processing techniques since each processor can operate on a subset of the data rather than requiring access to the data as a whole.

This work was extended by Crawfis and Max [45] where the authors present an ideal reconstruction formula. Their goal was to produce a formula which produced splats with smooth densities, so that the structure of the individual splats was not visible. Their example is that of a cube of $n \times n \times n$ voxels, each emitting an intensity of 1, and having no opacity. The projection of such a cube should be as constant as possible. The function they calculate produces a variation of at most 1 part in 256. They also describe how to map textures onto the splats with the use of hardware texture mapping (Silicon Graphics Rendering Machines

and Iris Explorer) to produce animations, and motion blurred images of 3D vector fields.

The work is also extended by Laur and Hanrahan [46], this time by approximating splats with Gouraud shaded polygons. The work they present is concerned with the production of real time rendering for interactive applications. Real time rendering is achieved by encoding the data as an octree, and projecting nodes of the octree using the splatting technique. Each node in the tree is associated with an error, the idea being that the node with greatest error is the node to be divided next in the refinement process. Using this method the user would be able to move a low resolution representation of the object in real time, which would be progressively refined whenever the user paused to consider the image.

3.7.3 Curvilinear grids

One aspect of volume rendering that does not receive so much attention is that of the display of curvilinear grids. The grids usually arise as the result of computational space being warped around an object of interest. A typical example is a computational fluid dynamics simulation of an aircraft wing, where the mesh has a much lower resolution away from the wing, than in the vicinity of it. Distances between neighbouring points can vary by a factor of 10,000 over the whole data set. Conventional ray casting methods run into difficulty since rays can enter and exit through any cell face, and also re-enter cells (Figure 3.20). Max et al. [47] went some way to addressing this problem by dividing non-convex cells up into convex cells and non-convex cells which are guaranteed to have only one ray span intersection for a given viewpoint.

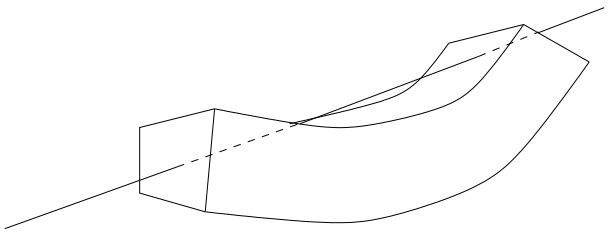


Figure 3.20: Ray can exit and re-enter curvilinear cells.

Alternatively, the large curvilinear grids could be resampled by a regular grid, and then ray casted. This method is problematical, since resampling would have to be carried out with a small step size to retain accuracy. This would result in a large increase in the volume of data. A larger step size would not result in such an increase in the amount of data, but has the problem that fine details in the data would be lost.

These problems can be avoided by using projection methods [27, 48, 49, 50]. Wilhelms and Gelder [48] show how accurate integration and composition can be performed on cells, in order to project the cells correctly onto the image plane. Wilhelms [50] also compares the projection and ray casting methods and shows that a speed up of 50–150 times is quite reasonable. A comparison of the different methods for projection was made by Williams [49], in which various methods are described for the projection of tetrahedral meshes. The order of traversal presents a problem, but once solved [51, 52], cells can be projected easily, usually in the form of hardware Gouraud shaded polygons.

It is also worth mentioning the work presented by Frühauf [53], which essentially solves many of the problems associated with the ray casting of curvilinear grids. In this paper the author shows how computational space can be unfolded into a regular grid. The same transformation can be performed on the light rays so that instead of sampling along straight rays in curvilinear space, samples are taken along curved rays in a regular computational space. This solves the problems of finding neighbours, and the ray/cell intersections. The ray direction is stored at each node as a vector, and some *particle tracking* technique, such as Runge-Kutta, is used to determine where the next sample occurs along the ray. Results show that very little error creeps into the ray paths, which therefore allows the production of accurate images.

3.7.4 Data compression

The data sets requiring visualisation are often very large. In fact they are usually larger by an order of magnitude, than data sets that can be handled easily. A typical data set can be around 16 MBytes. Techniques which reduce the volume of data can often have the advantage that images are also quicker to produce [54, 55].

Udupa and Odhner [54] extend their data model [24] to cater for volume rendering. In their method they assume (for a large reduction in data and image rendering times) that the opacity functions (of Section 3.2.2) have been defined using gradient operators such that voxels away from object boundaries contribute very little, if at all, to the image. This allows them to define a data structure which encodes the data as a *shell* around an object boundary. The shell is then projected using a back-to-front or front-to-back method, and since only voxels within the shell rather than all the voxels in the scene domain are projected, large reductions in rendering time are achieved. They report shell rendering is nearly 1000 times faster than a straight forward implementation of a standard ray casting method.

Volume compression was the focus of the work of Ning and Hesselink [55], where data is replaced by indices to a codebook. Using vector quantization, the authors use nearest neighbour mapping of every k-dimensional input vector X to some vector X_i selected from a finite codebook of candidate vectors. The original data can be replaced by the index i, which allows the data to be compressed. They chose the voxel value, normal and gradient magnitude for a block of b^3 voxels as the vector to be compressed, and found that a compression ratio of 5:1 could be achieved for practical data sets. The rendering times of the data set increased by 5% due to the overhead of checking a table entry to uncompress the data. They also show how faster rendering can be achieved by rendering the codebook entries from the viewpoint and compositing the result along each ray. Since the codebook has only a few entries, rendering of the codebook entries takes negligible time. The rendering of the data is now reduced to a composition of the codebook renderings that occur along the ray which avoids the need to perform costly trilinear interpolation and shading calculations at each sample along the ray. A speed up of about 10:1 was achieved using this method of volume rendering, although there is loss of image accuracy since the codebook is only representative of the data, and not all possible ray entry and exit points for a given view can be catered for.

3.7.5 Spatial techniques

Spatial techniques seek to explore coherency within the data to accelerate rendering. An additional data structure is constructed using the data which can then be used to accelerate the rendering, usually by allowing rays to skip over portions of the volume that are uninteresting. The most popular spatial subdivision technique is that of the octree [56, 57], which was applied to volume rendering by Levoy [43]. In a preprocessing step an octree is constructed where each node is divided if it contains a voxel which has non-zero opacity. Each divided node branches into eight subtrees, one for each of the subvolumes constructed when the volume is divided in half in the x, y and z directions. The ray is traced through the octree, using information within the nodes to jump over large areas of non-contributing (zero opacity) voxels. Most of the computation during volume rendering is the trilinear interpolation of sample values along the ray from surrounding voxels. By knowing in advance that all values within a subvolume are going to produce an opacity of zero which would result in no contribution to the image, the ray can safely skip over that subvolume to the next sample point. This would avoid the costly sampling process for all samples along the ray within that subvolume, resulting in a reduction of computational cost and therefore time. Although a speed up was reported by Levoy [43], work in Chapter 4 showed that perhaps this was not the case. In addition to this problem, any change of opacity function would require the octree to be recomputed.

The fact that the octree does not speed up ray casting is also mentioned by Yagel and Shi [58]. The authors suggest that the additional cost of tracing the ray through the octree can be avoided by storing the octree information at the empty voxels as uniformity information. Using this process, empty voxels are assigned a value which represents which level of the octree they are in, which can then be used to cause the ray to leap forward to bring it to the first voxel beyond the uniform region.

Other methods are discussed by Yagel and Shi [58] in which either an additional volume, or the data volume itself is used to store proximity flags or values. The idea behind this is to place a shell of flags, or shells of values representing distance, around the object which will enable an efficient ray tracking algorithm to switch from an efficient space jumping rapid traversal algorithm, to an accurate ray traversal algorithm. The proximity flags would indicate a surface was near, or more efficiently, the shells of distance values would indicate how far a ray can jump without encountering an object. These methods suffer from the fact that 3D preprocessing is required that must be repeated for any change in the data.

Yagel and Shi [58] also show how a method retaining starting depths for rays can accelerate ray casting of successive images in a rotation by avoiding the need to calculate a traversal of empty voxels in front of the object. They show how to avoid some of the problems associated with rotated objects covering objects previously visible and report a speed up of 2 to 6 times. It should be noted that this speed up applies to ray traversal, and since in many applications it is the ray compositing step that is the most time consuming (see Section 3.4.1), a significant speed up in the volume rendering process may not be observed.

Spatial techniques are also used by Subramani and Fussell [59], in order to store interesting voxels in a $k-d$ tree. This tree is obtained by using a median cut space partitioning algorithm to efficiently divide space up so that either branch, at any one time, removes a similar number of voxels. Bounding volumes are also used at nodes to efficiently encapsulate the voxels

within each subtree. This leads to an efficient subdivision of space – the $k-d$ tree, which can be efficiently ray traced. The authors report a reduction of over 90% in the amount of data that needs to be considered in order to produce the image, although they do not discuss rendering acceleration times.

One other acceleration method is the polygon assisted ray casting (PARC) technique of Avila et al. [60]. The object is simply approximated by polygons which can be projected onto two depth buffers – a front projection and a back projection. These two depth buffers will contain the start and end points for each ray of the image, and in this way the ray does not traverse parts of the volume that do not contribute to the final image. The problem with this method is that of how to choose the polygonal representation of the object. This can be achieved by using the faces of subvolumes of the data which contain a surface, and merely requires the decision of how fine the subvolumes should be. The method has the advantage that costly unnecessary ray sampling is avoided, and therefore produces images with less computation.

3.7.6 Shading techniques

The problem of interactive shading on less powerful graphics workstations was addressed by Fletcher and Robertson [61]. Interactive shading is desirable, since it allows researchers to move light sources around their data in real time, thus giving a better understanding as to the three dimensional nature of their data. Another benefit is the experimentation of light source positions during the production of animation sequences without the need of re-rendering each frame. Interactive shading is achievable by computing a restricted table of n normals (where usually $n = 256$). The normal calculated for each pixel in the rendered image, is then mapped to the normal in the table that has the closest matching direction to it. In recalculating the image with different shading parameters, only the intensities resulting from shading the normals within the table need to be computed. These values can then be mapped back onto the image to produce the new image. This results in a substantial reduction of computation – to a level most graphics workstations can handle. The main problem is the fact that only a restricted subset of normals can be reproduced accurately. One example image [61] is that of a hemisphere which has been shaded with this technique, and has resulted in a hexagonal faceted image, each hexagon centred about a known normal. Their solution is to dither the normals using a process similar to the Floyd-Steinberg algorithm for dithering. They report images can be shaded at rates of 47 to 139 fps on a DEC 5000/200.

3.7.7 Volume seeds

Ma et al. [62] present a method in which the user is given interactive control of the final volume rendered image. The user can alter variables such as the opacity table and voxel colours in real-time through the use of a *cache*. The idea is that the volume is ray traced from the given viewing direction, and the value of each sample along the ray is stored in a three dimensional array. This array is simply traversed, and voxel values are looked up in the classification table in order to perform the compositing step. Since costly trilinear interpolation of the sample values is no longer required, real-time compositing of the image is achievable.

The user is also given control of the *volume seeds* technique. The user can place a seed anywhere in the volume with the effect that voxels close to the seed are enhanced, and those further away are made more transparent. The example given [62] is that of placing a seed in the throat to make the spine in the neck area easier to see. The effect is achieved by increasing the opacity of close voxels, and reducing the opacity of further away voxels using a simple function. Real-time control is achieved by combining the seed placement with the volume caching to give users full investigative control over their data.

3.7.8 Systems and environments

Techniques can be regarded as the precursors to fully fledged systems that provide environments for data visualisation. Commercial examples would be those of AVS, Data Explorer and the public domain Khorus packages. These all work on the data driven flow model and are based on the linking of modules to form networks (Figure 3.21).

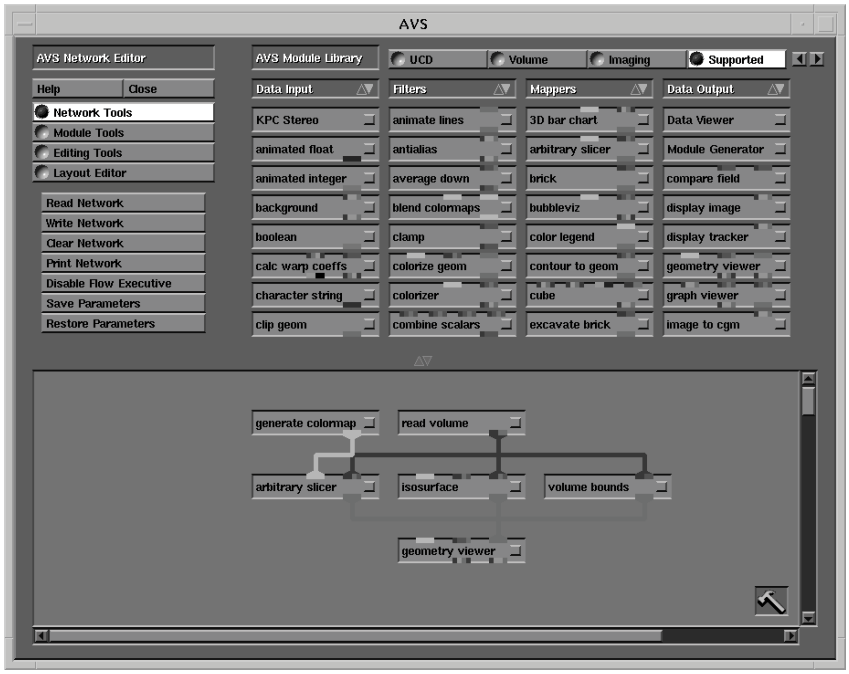


Figure 3.21: Network editor and module palette from AVS^{TM}

The commercial packages tend to be well known, and widely used, but very often it is the *homegrown* software that incorporates innovative features.

A simple but effective system is reported by Corrie and Mackerras [63]; the authors point out that one of the major benefits of their system is its flexibility in producing images over systems, such as AVS, which limit the user to a number of fixed rendering techniques. This flexibility is achieved by using a shading language, rather than a user interface. It is therefore flexible and extensible. The example visualisations and code shown in the paper [63] are for the standard shading model for volume rendering [25, 27], Sabella's model [26] and a propriety shader for the Dominion Radio Astrophysical observatory. The authors report that allowing a higher level shading language does result in a loss of efficiency when it comes

to computing the images, with the example that their implementation of standard volume rendering takes about 1.7 times as long to render an image.

Montine [64] takes a low level approach to the task of providing a visualisation language and provides just the basic functions for visualisation. These functions are classified as to which object they operate on (volume – the original data, environments – lighting conditions and viewing parameters, and images). To visualise a set of data the user must provide the appropriate calls to all the functions. This requires low level knowledge from the user as to what the available functions are, but in this way provides the user with a powerful environment.

Smart particles are introduced by Pang and Smith [65] as a way of providing a framework for visualisation. Using existing techniques usually associated with 3D data sets, such as isosurfacing, particle tracking and volume rendering, the authors endeavour to show how an effective, but easy to use system can be made using the SPARTS (smart particles). The sparts are sprayed into the data set in much the same way as paint is sprayed from spray cans. Sparts can be of several forms such as surface seeking, volume penetrating, or flow tracking. Sparts can also operate on the data, and leave messages for other following sparts. The authors indicate that their system allows the user the flexibility of exploring their data using a simple and intuitive tool – the spray can. It is an effective tool since complex visualisations can be built up by using different combinations of sparts.

An example of a visualisation environment actually being used would be that of Yoshida et al. [38] in which the authors demonstrate the effectiveness of their system in support of neurosurgical planning (CliPSS – Clinical Planning Support System). They show the simple step by step process a user follows to produce data which can be used for surgical planning. Each slice is edited interactively to remove unnecessary data, such as the bed. The surgeon can use an extraction function to extract the volume of interest (VOI) by defining two threshold parameters and a seed. CliPSS then uses region growing to extract the VOI by grouping voxels together until a boundary is found. Boundaries are distinguished by moving a $7 \times 7 \times 7$ filter across the volume, calculating statistics locally from the voxels. The surgeon is given the option of rendering the VOI as a surface (isosurfacing) or as a volume (volume rendering). Surgical planning is supported by using a stereotactic frame which fits over the head of the patient. Using this frame a correlation between the computer model and patient can be made. The surgeon can plan entry points, direction and depth to probe for a target lesion on the computer. The surgical instruments can be attached to the stereotactic frame and positioned accurately over the lesion. The surgeon then knows the direction and depth to which he must probe. Using this tool operations are made much safer for the patient.

3.7.9 Anatomical atlas

Karl Heinz Höhne and his colleagues at the University Hospital Ependorf, Hamburg, are working on a project to segment and label the whole of the human anatomy [66, 67, 36, 34, 35]. Their main aim is to provide a hypermedia system for anatomical study, enabling students to contol their exploration of the body. They used their generalised voxel model [66] to store the available information about each voxel – its grey level value as obtained from CT or MRI, and its membership to parts of the body, or functional areas or a lesion. In addition to this

model, and the graphical capability of the system, is a knowledge base which encodes the relationship between all the different regions (for example of the brain). The method by which the model is constructed, and the functionality of the system is described in [36] with additional information about producing the images of blood vessels in [67]. In order to isolate each entity within the brain, they first perform a semi-automatic segmentation to extract voxels belonging to the skull, brain or ventricular system. From this model an anatomist can examine the data slice by slice and identify and label regions. Each region is extracted by selecting it using region filling, thresholding and pixel selection. Basically the anatomist would point to an object, and define data thresholds which would identify the voxels that belong to the object as those that can be reached by a path of voxels with values between the threshold values. For troublesome regions pixels can be included or removed in a way similar to using a paint package. Once an object has been selected it is given an entry in the knowledge base indicating its name, and functionality – for example [66], "*gyrus calcarinus is part of the lobus occipitalis*".

With this model the user can explore the human head with ease. A typical enquiry could be to start with the whole head, and indicate a cut and depth. This cut would be performed and the user can point to the revealed anatomy and query the database to find out its nature. The database will provide a label to each object and show the user where the object occurs in the knowledge tree. Cut planes can be taken, but could ignore specified objects such as eyes. X-ray images can be simulated with the addition of being able to enquire what has contributed to the resulting intensity along the ray. The knowledge tree itself can be queried – for example the user could select an object from the tree, and ask for it to be displayed.

The resulting system is a powerful exploration tool, giving users a very natural feeling of dissection as they explore the brain. Höhne estimated that about one person year was spent on the segmentation of the brain during the construction of the brain atlas. The group is now extending its work to the whole of the human anatomy [35].

3.7.10 Image segmentation

It has already been mentioned in the previous sections that image segmentation is a necessary tool in order to extract the various parts of the voxel model, so that each object can be examined in isolation. The problem is that there is no one method that can automatically segment complex objects from voxel data. Most methods rely on interaction, in the form of the user selecting voxels within a slice that are within the volume of interest (VOI) and filling the volume in 3D within a certain range. The range is defined using thresholds and the region is defined simply as all those voxels that can be reached from the start voxel by a path of voxels that have values that lie between the thresholds. The thresholds can be defined by allowing the user to probe the data – displaying the values of voxels pointed to by the user within a slice. Examples of this method can be found in [36, 37, 38, 39].

Algorithms which perform automatic segmentation on images (for example [30]) are used to perform semi-automatic segmentation of voxel data. The regions produced using an automatic method will often have false connections (or bridges) to other regions, or will be split from regions that they should be joined to. One simple method to solve the first case is to shrink the voxel model by n voxels (where n is small, typically $n = 1$), and then grow it by

n voxels. This will have the effect of removing links of n voxels wide [68, 66]. Other methods use the image processing techniques of erode and dilate to erode away the bridges or dilate the region to join it to others [69, 66].

Yoo et al. [70] show the difference between syntactic and semantic classification. In syntactic classification, geometric clipping is used to cut away parts of the volume not required, and intensity values are mapped to opacity using mappings such as gradient value. They point out that these operate on the whole volume and cannot distinguish between features such as the different bones in a leg. Semantic classification isolates the different parts of the volume interactively. Firstly the regions are segmented automatically, and connected in the form of a branching tree. This tree is traversed interactively by the user in order to select the object of interest. They remark on the effectiveness of their system by giving the example that the extraction of the brain, which previously took 40 minutes, now takes about 10 mouse button clicks in a few seconds.

3.8 Conclusion

This chapter has been concerned with the production of images from volume data using the method of volume rendering. A thorough comparison of the more popular viewing models was given in Sections 3.2 and 3.3. The new work presented in the chapter was concerned with the acceleration of image production (Section 3.4), which was compared to the existing acceleration methods of Section 3.5. Section 3.6 presented a thorough comparison, in terms of computational requirements and image quality. A review of other volume rendering methods was given in Section 3.7.

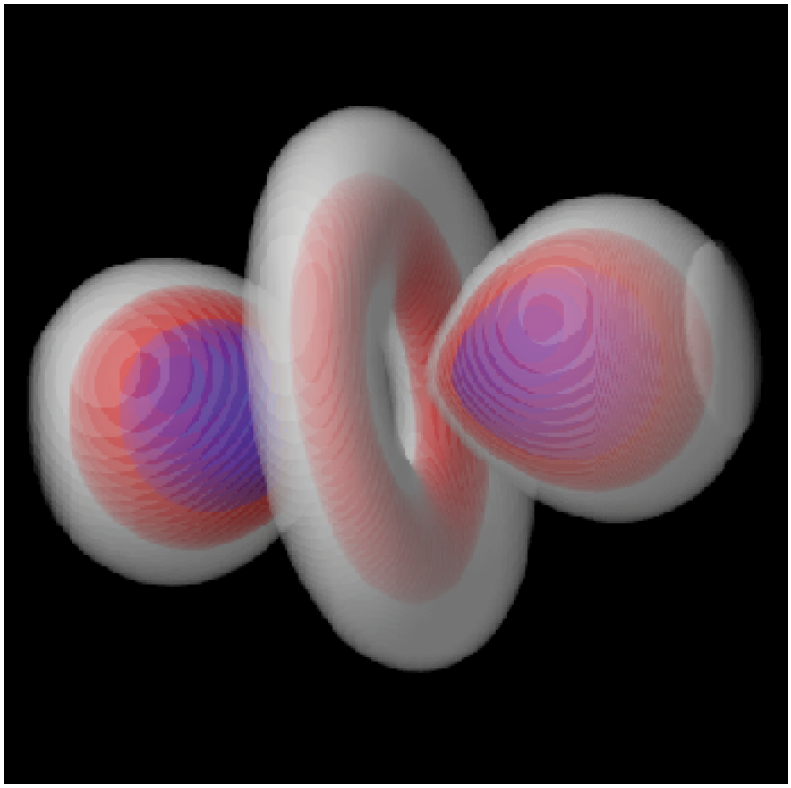


Figure 3.1: Volume rendering of AVS Hydrogen data set.

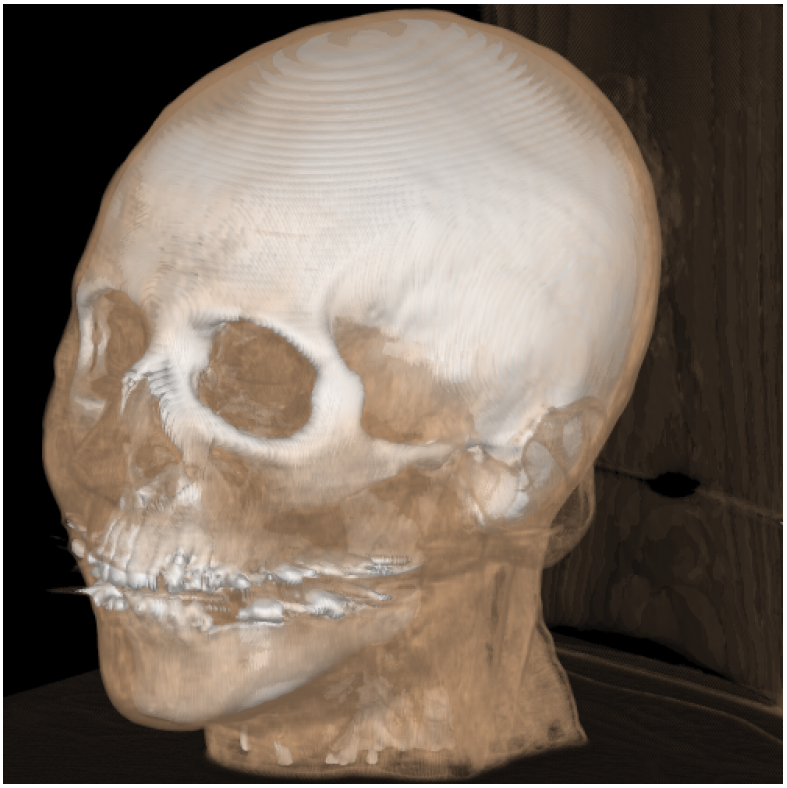


Figure 3.2: Volume rendering of CT head data set.

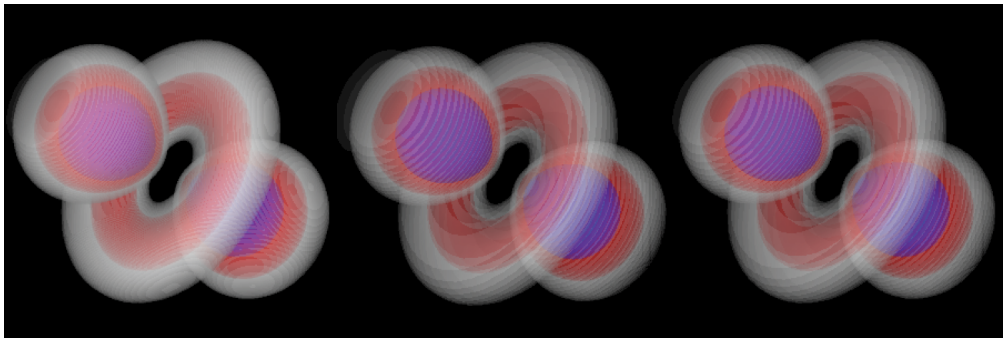


Figure 3.6: (a) Trilinear interpolation, (b) Increased interval size, (c) Bilinear interpolation.

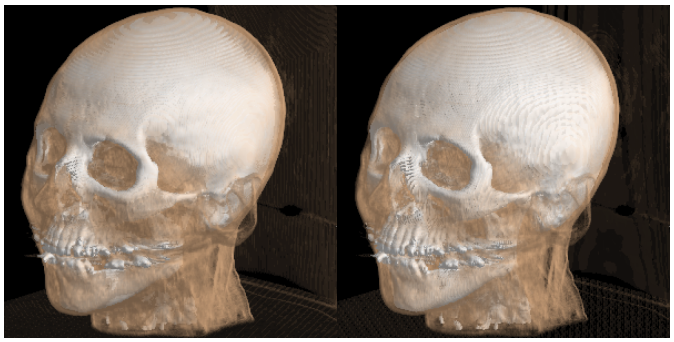


Figure 3.7: (a) Trilinear interpolation, (b) Bilinear interpolation.

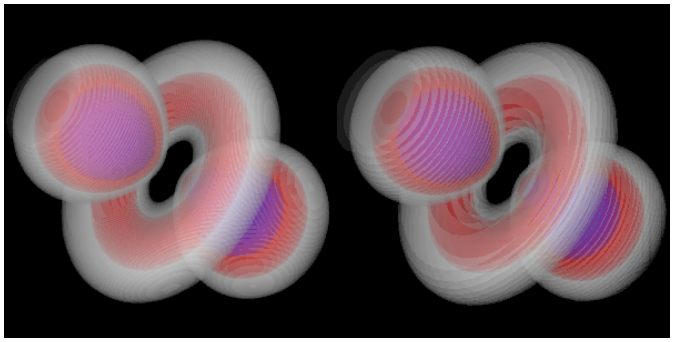


Figure 3.8: (a) Trilinear interpolation, (b) Bilinear interpolation.

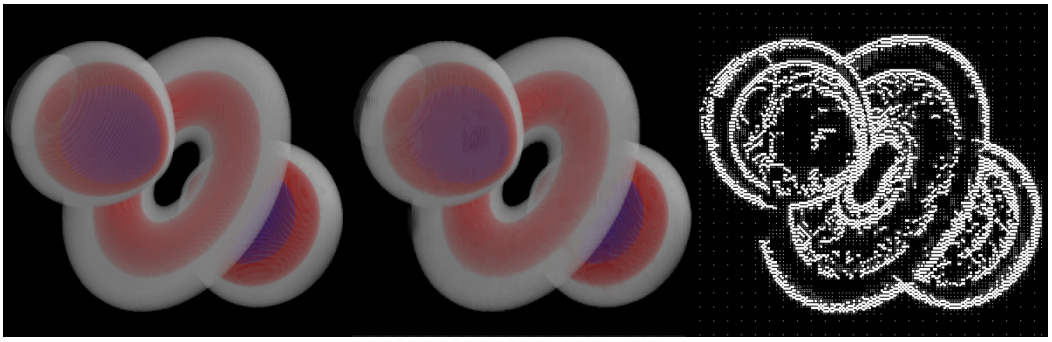


Figure 3.9: The adaptive rendering process.

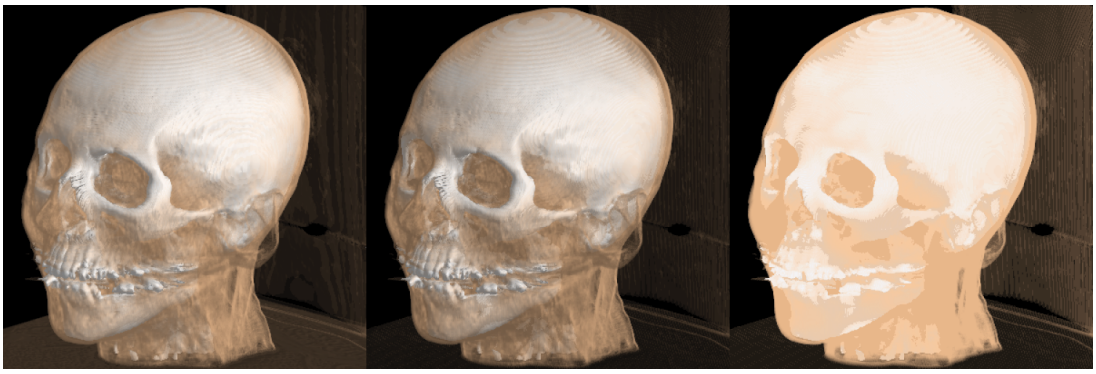


Figure 3.11: (a) Standard method (b) Adaptive Termination (c) No shading.

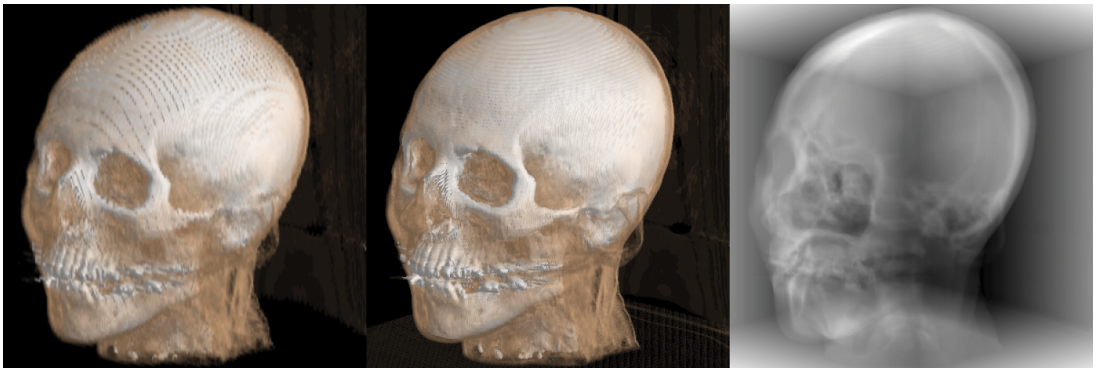


Figure 3.12: (a) Template method (b) Bilinear method (c) X-ray method.

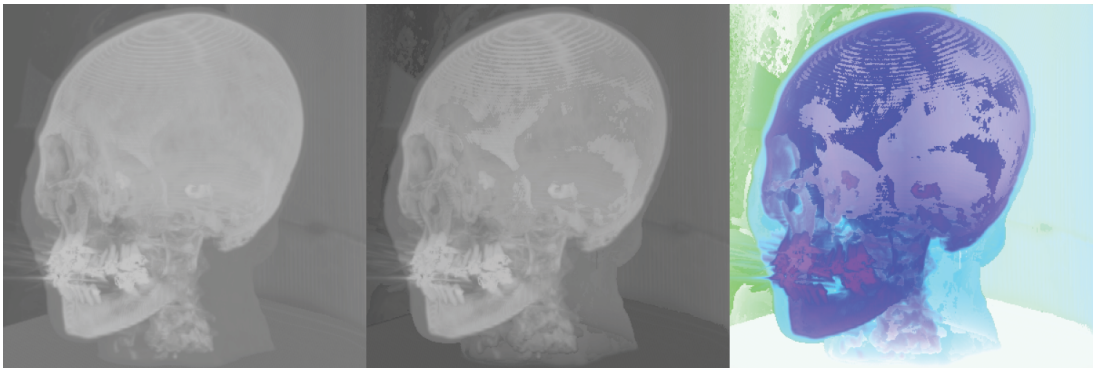


Figure 3.13: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

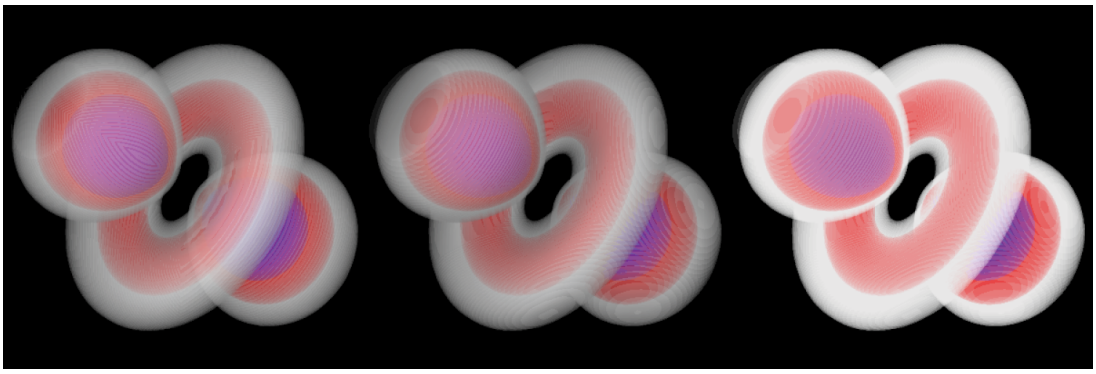


Figure 3.14: (a) Standard method (b) Adaptive Termination (c) No shading.

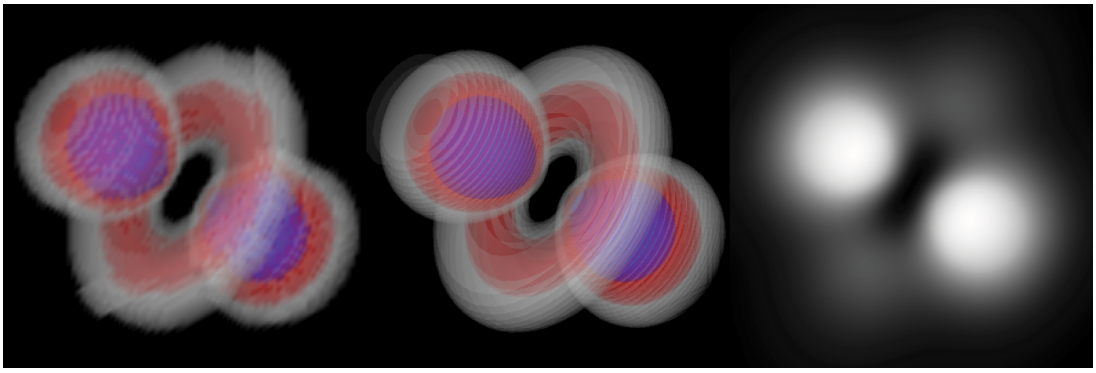


Figure 3.15: (a) Template method (b) Bilinear method (c) X-ray method.

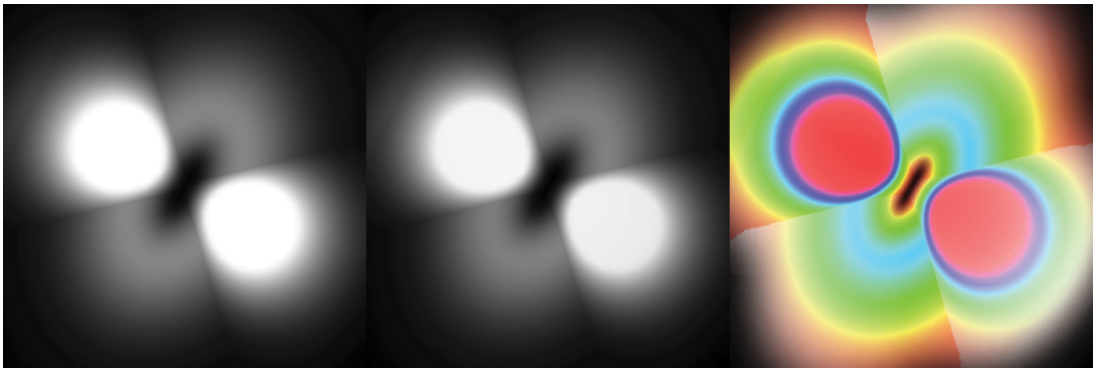


Figure 3.16: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

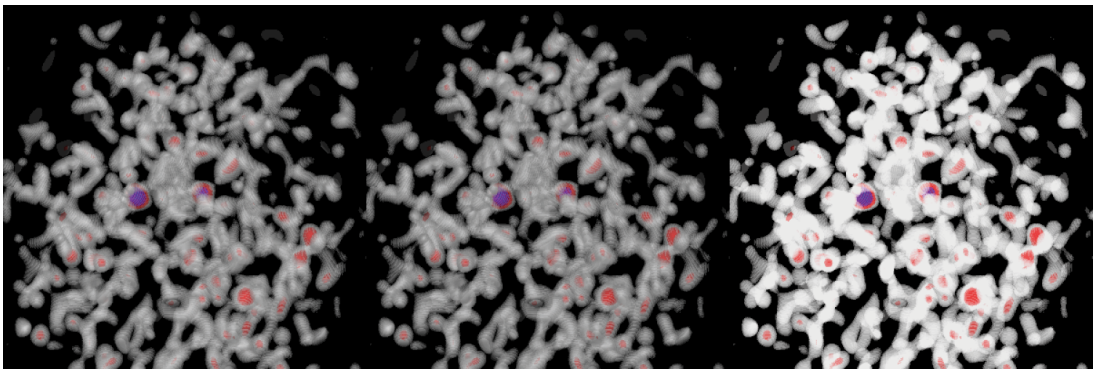


Figure 3.17: (a) Standard method (b) Adaptive Termination (c) No shading.

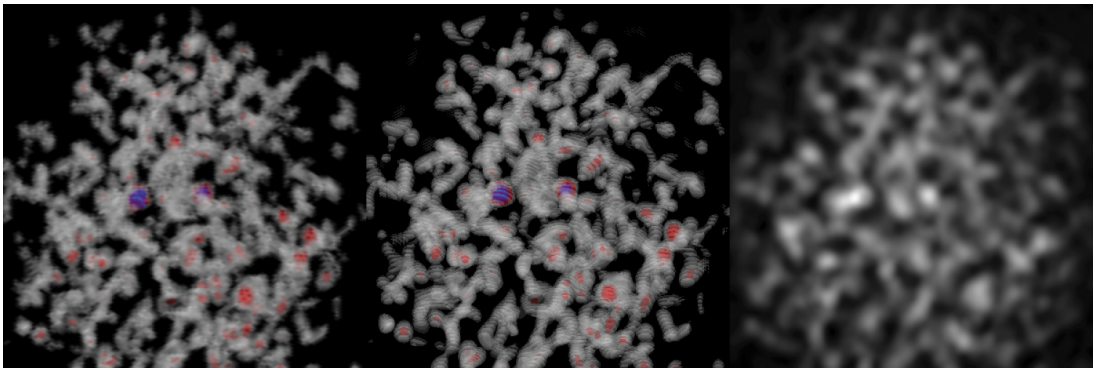


Figure 3.18: (a) Template method (b) Bilinear method (c) X-ray method.

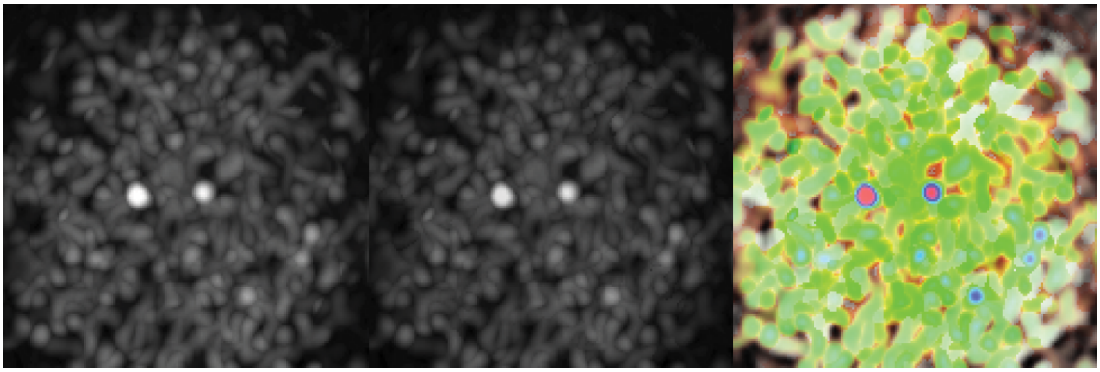


Figure 3.19: (a) Maximum method (b) Maximum with depth cue (c) Sabella's method.

Chapter 4

Direct Surface Rendering of Objects

4.1 Introduction

In previous chapters visualisation of three dimensional data has been achieved through reconstructing an isosurface from the volume data set (Chapter 2) or through a volume rendering of the volume data set (Chapter 3). In this section the structure of the volume data itself is used to derive the visualisation. The underlying voxels themselves are either projected onto the image plane, or rendered through ray casting.

In Section 4.2 the existing algorithms for the display of these voxels are described. They include algorithms that project the voxels – a back-to-front projection algorithm, a front-to-back projection algorithm and a boundary detection algorithm. The so called semi-boundary construction is also presented which allows a compact storage of the voxels that comprise a surface. Of the ray casting methods, two are presented which simply sample the ray at discrete locations in order to find the surface which has been determined as either part of a segmentation process, or as an isosurface of a particular threshold. Finally a fast voxel traversal algorithm is described.

The shading techniques for the voxel surface are presented in Section 4.3. Firstly the standard equations for rendering are given in Section 4.3.1. In the following sections the different shading techniques that are available are described (Z-buffer shading (Section 4.3.2), Gradient shading (Section 4.3.3), Constant shading (Section 4.3.4), Normal-based contextual shading (Section 4.3.5) and Grey-level gradient shading (Section 4.3.6)).

A new method for producing high quality images from voxel data is presented in Section 4.4. The method is described in detail, and then compared to some of the shading techniques of Section 4.3.

In Section 4.5 a new method which allows real time cutting operations to take place on volume data is presented. It is compared with existing methods, and tested on various data sets.

Finally conclusions for this chapter are given in Section 4.6.

4.2 Determining Voxel Visibility

Frieder et al. [71] propose the back-to-front (BTF) projection algorithm. This algorithm is not strict in the sense that the projection of voxels is not in true back to front order, although it is shown that if voxel a is projected before voxel b then b is not obscured by a. The algorithm that achieves this property is quite simple, and is best understood with the use of a 2D analogue (Figure 4.1). In this figure the voxels A, B, C and D are occupied, and are to be projected onto the screen. Assuming the x and y axes are arranged as they are in Figure 4.1, such that the origin is furthest away from the viewer, then either a traversal in which the x coordinate varies fastest ($ABCD$), or a traversal in which the y coordinate varies fastest ($CADB$) will lead to a correct rendering of the screen. This is because if part or all of a voxel with coordinates (x, y) is obscured by a voxel with coordinates (x', y') then $x' \geq x$ and $y' \geq y$ and so by projecting the voxel at (x, y) before that of (x', y') the correct image is achieved for this particular arrangement. If the viewer is in a different position, x and y must be chosen to increase or decrease accordingly. The algorithm can be extended to 3D quite easily, and now if the origin is furthest from the viewer, the 3D voxel data set is traversed in increasing x, y and z. The choice of which index varies fastest can be made arbitrarily, and therefore can be chosen so that the data is read a slice at a time, thus avoiding unnecessary *paging*.

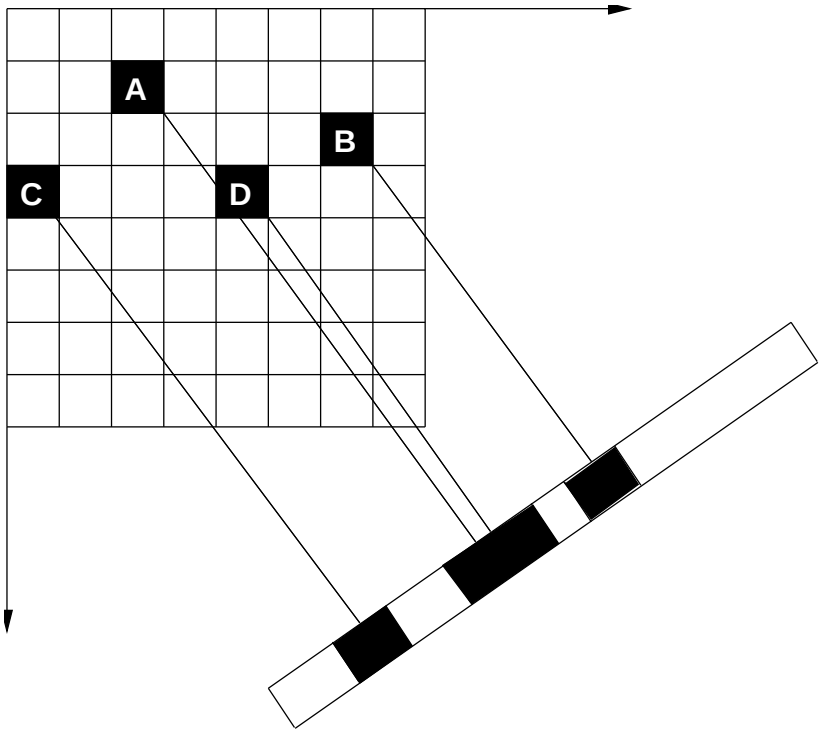


Figure 4.1: Back-to-front traversal yields correct image.

The shading of images is achieved by using the Z-buffer method (Section 4.3.2), or gradient shading (Section 4.3.3) of the Z-buffer, with particular attention given to the production of a continuous image. Particular problems arise when the size of a voxel is greater than the size of a pixel, because not all pixels within an object will be painted, since each voxel is projected

onto one pixel. A continuous image is achieved by scaling the size of the voxels by a constant amount depending upon the view point.

The natural progression of the BTF method is the front-to-back (FTB) method [72]. For this method the volume is traversed opposite to that of the BTF method so that unobscured voxels are painted first. Voxels further back which are obscured by those already projected are not displayed. The problem with this simplistic view is that the visibility cannot be determined without projecting the voxel, and checking the image, which is just as expensive as projecting the voxel and painting it. Reynolds et al. [72] show that arbitrary rotations about the object can be achieved using rotations of α, β and γ about the x, y and z axes respectively. They further show that a rotation α followed by β results in lines parallel to the x axis remaining parallel to the x axis. The rotation γ can be achieved by an image rotation. Using this information they project rows of voxels, scanline by scanline using look-up tables, and updating a dynamic screen data structure which represents unpainted scanline pixel subsets. Further projection of voxels is restricted to those that project onto unlit pixels as given by the dynamic screen data structure, and can be determined easily by taking into account the row and scanline coherence.

Gordon and Udupa [73] give a fast boundary detection algorithm. The algorithm determines all those cube faces that share a 1-cube and a 0-cube within the volume, i.e. those faces that make up the boundary of the object. The algorithm requires the voxel data to be in the form of a binary array so that each exterior cube has a value of zero (0-voxel) and each interior cube has a value of one (1-voxel). Given any boundary face in the volume, all other boundary faces that are connected to this one can be found. In other words, the boundary of one connected volume can be tracked from just one initial boundary face. The algorithm is similar in principal to the simplicial pivoting method also used in *chain of cubes* (Chapter 2; Section 2.4.3). The result is a mesh of cube faces which can then be displayed. The problems with this method are that it will only extract the connected object from the seed face, and any other objects within the volume will not be found. In addition to this, the mesh will contain many cube faces, and so will therefore be quite large, and since there are only 6 possible face orientations, the shading of the surface will appear to have false edges. This shading problem will be discussed further in Section 4.3.4.

Udupa and Odhner [24] create a *semi-boundary* representation of the voxel model. Their representation has advantages over other representations as it stores the voxel model compactly and allows various operations to be carried out efficiently. They define

$$U_s(\mathbf{C}) = \{c \in C \mid g_s(c) = 1\} \tag{4.1}$$

where g_s is a mapping of C (the scene domain) onto $\{0, 1\}$ using a segmentation function s. $\mathbf{C}$ is the pair (g, C) and is called a scene. Hence U_s is all those voxels that are 1-voxels.

They define the 6-neighbours of a 1-voxel c to be (along the principal axes)

$$n(c) = \{d \mid \text{for some } j, 1 \leq j \leq 3, |c_j - d_j| = 1 \text{ and } c_i = d_i \text{ if } i \neq j\} \tag{4.2}$$

where c_i and d_i are the i^{th} coordinates of the voxels c and d respectively.

The *semi-boundary* S_s of the scene $\mathbf{C}$ for a given segmentation function s is a tuple (D, Δ), where

$$D = \{c \mid c \in C, g_s(c) = 1, g_s(d) = 0 \text{ for some } d \in n(c)\} \tag{4.3}$$

and Δ, the encoding function, assigns to every element c of D a 6-bit number called the neighbour code. This neighbour code indicates the value of the segmentation function for the neighbours of c.

From the definition it can be seen that the semi-boundary contains all those, and only those, 1-voxels that have a 0-voxel as a neighbour. 1-voxels which are surrounded by 1-voxels are left out of the representation, and it is this fact that results in the compactness of the data structure. It should also be obvious that a 1-voxel that is surrounded by 1-voxels will not contribute to an image of the object and hence by removing it from the rendering process will accelerate the rendering process without any loss of information.

The image of the data set is created by projecting the semi-boundary onto the viewing plane using the appropriate view transformations and orthographic projection. Each voxel is projected onto the image plane, and the pixels it covers are shaded according to a calculated intensity. As mentioned before, care must be taken with the projection to ensure that pixels are covered, and no gaps occur.

The semi-boundary is a compact representation of the surface model, and as such allows operations to be carried out efficiently on the data. The operations Udupa and Odhner discuss are cutaway, osteotomy, mirror reflection and segmental movement. The cutaway option is to perform a slicing of the data set and remove the cutaway portion completely. The osteotomy allows the user to specify a cut outline and depth of cut. The semi-boundary is then split into two objects, and using the segmental movement function the user can move the segmented object around the domain using six degrees of freedom. Finally mirror reflection allows the model to be reflected about planes.

The semi-boundary allows data to be represented compactly, and as such to be operated on interactively for quite modest data sets. The display of the semi-boundary can be accelerated because visibility of voxel configurations can be determined in a preprocessing step for each of the 8 possible viewing octants. Through the use of look-up tables for visibility, normal information and transformation, the semi-boundary can be projected very efficiently. They report that preprocessing takes about 10 to 15 minutes per data set, and that the display of the semi-boundary takes about 2 to 5 seconds on a Sun 4/110C. In their opinion the method is 3 to 5 times faster than other approaches. It must be decided whether or not it is worth waiting for the initial preprocessing step in order to achieve the near interactive manipulation of the data set.

The above methods all project the voxel volume onto the screen, whereas the technique employed by Höhne et al. [66, 74, 35] is to track a ray from each pixel into the volume, sampling at evenly distributed points along the ray in order to build an intensity profile. A surface voxel is identified as one which exceeds a preselected intensity threshold, or one which has been classified as part of a segmentation process. It is this voxel that is then used to determine the intensity of the pixel using one of the methods of Section 4.3.

An earlier method by Tuy and Tuy [75] also used ray casting. A ray is sampled at intervals of some distance d. Once a surface transition occurs (i.e. moving from outside the object to inside) the depth at that point is used to create the shading. The distance d has to be chosen carefully to avoid missing thin objects, and to minimise computational time.

The most costly part of these ray casting algorithms is the tracking of the ray through the volume. In [41] Amanatides and Woo show how fast traversals of such a volume are achieved. The algorithm is of use for both the cuberille structure of volume data, and for partitions of 3D space that usually occur when some sort of bounding scheme is introduced for use with ray tracing. The particular application they give is for the latter partition, although it can be easily adapted for the former.

The idea is that a ray can be represented by

$$\mathbf{r} = \mathbf{u} + t\mathbf{v}(t \geq 0) \tag{4.4}$$

where $\mathbf{u}$ is the ray origin, $\mathbf{v}$ is the direction vector of the ray, and t is the distance travelled along the ray. Using this equation and the regular spacing of the grid, the distance travelled along the ray between successive voxel boundaries can be calculated for each dimension. For example in Figure 4.2 to move to an adjacent voxel in the y direction $\sqrt{10}$ units must be skipped over, and to move to an adjacent voxel in the x direction $\frac{\sqrt{10}}{3}$ units must be skipped over. By continually keeping track how far the ray has traversed through the volume, and how far it has traversed in each dimension, the minimum of three variables can be taken to see which boundary is crossed over next. The algorithm is presented in full by Amanatides and Woo [41], and requires just two floating point comparisons, one floating point addition, two integer comparisons and one integer addition per iteration.

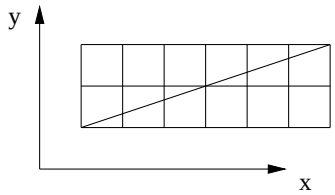


Figure 4.2: Fast voxel traversal.

4.3 Determining Voxel Contribution to Pixel Intensity

4.3.1 Shading equation

The image of the surface is produced by calculating the intensity of each pixel using the following lighting equations [29].

The intensity I, of the pixel is given by the following depth cueing equation

$$I = s_0 I' + (1 - s_0) I_{dc} \tag{4.5}$$

$$s_0 = \frac{Max - z_0}{Max}$$

where Max is the maximum depth of any part of the object from the view plane, and z_0 is the depth of the part of the object from the current pixel. I_{dc} is the intensity of the depth cue colour and I' is the calculated intensity of the surface,

$$I' = I_a k_a + I_p(k_d cos\theta + k_s cos^n \alpha) \tag{4.6}$$

where (see Figure 4.3)

- I_a is the intensity of the ambient light source
- k_a is the ambient-reflection coefficient ($0 \leq k_a \leq 1$) which can be regarded as the amount of light reflected from a material
- I_p is the intensity of the point light source
- k_d is the diffuse-reflection coefficient of the material ($0 \leq k_d \leq 1$)
- θ is the angle between the direction vector of the light source L, and the surface normal N.
- k_s is the specular-reflection coefficient ($0 \leq k_s \leq 1$)
- n is how *tight* the specular reflection is to be
- α is the angle between the direction of light reflection R, and the viewing vector V. (Phong shading).

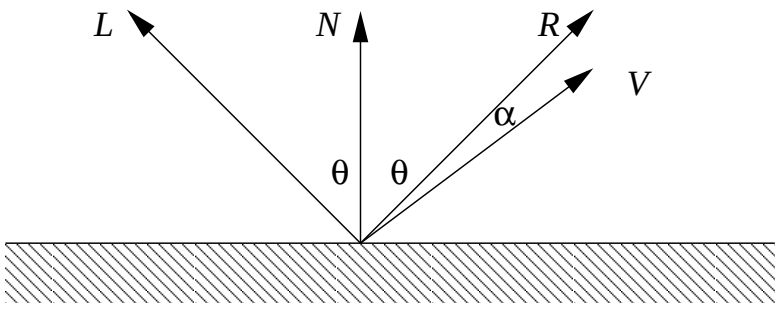


Figure 4.3: Rays involved in ray tracing.

These equations are still not complete [29], but they model most effects necessary for the display of voxel data. In the following discussion each of the lighting techniques are introduced along with the simplifications to the lighting equations that they make.

4.3.2 Z-buffer shading

The simplest shading method is that of distance only shading, also known as Z-buffer shading due to the technique used to produce the images [29]. For this method the distance from the view plane to the surface voxel is calculated and $cos\theta$ and $cos^n\alpha$ are set to 1. In other words neither the surface normal nor the light and view vectors are calculated. Distance only shading produces smooth looking images with no visible object curvature which is a result of the fact that adjacent pixels have similar distances from the view plane, and therefore little variation in intensity. The main problem is that human perception relies on edge information to give cues as to the shape of the surface, and without these distance only shading is of limited use. The advantage of this method is the fact that no extra computation is required to compute the images, since the image is calculated from the Z-buffer.

4.3.3 Gradient shading

The method of gradient shading was intended to be a post-processing step that takes place on Z-buffers. The idea is to use a Z-buffer produced by some technique and produce the shading by calculating the surface gradients from the depth data. The Z-buffer contains the distance from each pixel to the visible surface along a ray traced from the pixel to the surface according to the view transformation. The Z-buffer is of the form $Z = z(x, y)$. The surface normal N can be computed by calculating the gradient ∇z at that point.

$$\nabla z = \left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, 1 \right) \tag{4.7}$$

At pixel (i, j) $\frac{\partial z}{\partial x}$ can be approximated by the backward difference

$$\delta_b = z(i, j) - z(i - 1, j) \tag{4.8}$$

by the forward difference

$$\delta_f = z(i + 1, j) - z(i, j) \tag{4.9}$$

or by the central difference

$$\delta_c = \frac{1}{2}(z(i + 1, j) - z(i - 1, j)) \tag{4.10}$$

and similarly for $\frac{\partial z}{\partial y}$.

For smooth objects δ_c is a good approximation to the gradient, but in the region of object edges, incorrect gradients can arise. These can be avoided by using forward and backward differences in problem regions which is achieved using weights as described in [76]. Images shaded using this method convey far more depth and curvature as compared to distance only shaded images. The disadvantage is that the blocky structure of the voxels is obvious, and detracts from the image. The advantages of this method is that it does not require any extra calculations or storage for object normals as the processing takes place on the Z-buffer. Secondly, the shading has a time complexity proportional to the image size rather than object complexity and is correspondingly faster when large and complex medical objects are viewed.

4.3.4 Constant shading

The normal at a surface boundary is calculated by simply determining the normal of the cube face the ray has hit. Using this method, the angles θ and α can be determined as the dot products of the unit vectors $L \cdot N$ and $R \cdot V$ respectively. This method is similar to methods which shade polygonal meshes using constant shading. The problem associated with this method is the fact that false edges can occur in the object (Figure 4.4). In Figure 4.4 the object boundary has normal b, but as can be seen the voxel boundary has normals a or c depending upon which face is hit by the ray. The resulting images with the false edges present confuse the interpretation of the object. The only advantage of this method is the fact that the normal calculations can be precomputed since only 6 possible normal orientations are available.

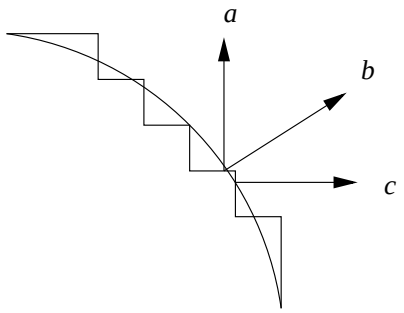


Figure 4.4: False edges occur when using just voxel faces for shading.

4.3.5 Normal-based contextual shading

For this method the orientation of the face the ray has hit is found. The normal to the surface at that point is then determined using information about the orientation of the four neighbouring faces, one neighbouring each edge of the face struck by the ray. This information provides a total of 25 different orientations of the surface normal which can then be used to shade the image. The advantage of this method over the previous methods is the fact that the shading information is *image independent*. The normals are calculated from the object description, and as such are view independent resulting in images that are coherent during animation. The disadvantage is the fact that there are only 25 possible different normal orientations, and therefore a restricted number of intensities available for shading. For information on the hardware implementation of discrete shading of voxels the reader is directed towards the work of Kaufman et al. [77, 78].

4.3.6 Grey-level gradient shading

The previously discussed methods all rely on creating the normal data either from an image based representation of the object, or from an intermediate mesh representation of the boundary of the object. We can regard this as producing only an approximation of the true surface normal since the intermediate representation, whether it be the image or mesh, is only an approximate description of the surface. The voxel data comes in the form of grey-scale

values which can be used for the accurate determination of surface normals in the following way. The grey values in the volume represent the density of the relative amounts of tissue (air/skin, soft tissue/bone) in the surrounding voxels. As a result of this the grey values around a voxel represent the surface orientation and as such can be used to calculate the normal at a surface interface. To calculate the normal N at a voxel at a location (i, j, k),

$$\begin{aligned} G_x &= v(i+1, j, k) - v(i-1, j, k) \\ G_y &= v(i, j+1, k) - v(i, j-1, k) \\ G_z &= v(i, j, k+1) - v(i, j, k-1) \end{aligned}$$

$$N_i = \frac{G_i}{\sqrt{|G_x|^2 + |G_y|^2 + |G_z|^2}},\ i = x, y, z$$

$$N =< N_x, N_y, N_z > \tag{4.11}$$

where $v(i, j, k)$ is the grey level value of the voxel at location (i, j, k). Using Equation 4.11, the gradient is calculated using the 6 neighbours of a voxel.

The gradient can also be calculated from all 26 neighbours using

$$G_x = \sum_{\beta=-1}^{1} \sum_{\alpha=-1}^{1} v(i+1, j+\alpha, k+\beta) - v(i-1, j+\alpha, k+\beta)$$

$$G_y = \sum_{\beta=-1}^{1} \sum_{\alpha=-1}^{1} v(i+\alpha, j+1, k+\beta) - v(i+\alpha, j-1, k+\beta)$$

$$G_z = \sum_{\beta=-1}^{1} \sum_{\alpha=-1}^{1} v(i+\alpha, j+\beta, k+1) - v(i+\alpha, j+\beta, k-1) \tag{4.12}$$

Problems occur in the vicinity of small, thin objects since the assumption that surrounding voxels give an indication of surface orientation is no longer valid. One solution to this problem is similar to that given in Section 4.3.3 in that the normal is determined adaptively from the surrounding grey level gradients. The gradient is taken to be the *central*, *forward* or *backward* differences of neighbouring voxels depending upon their values with relation to voxel (i, j, k). If the value of voxel $(i-1, j, k)$ and $(i+1, j, k)$ are both smaller, the gradient is taken to be the forward or backward difference depending upon which is the lower of the two, otherwise the central difference is used. A similar calculation is made if they are both larger than the value of voxel (i, j, k).

4.4 Direct Surface Rendering

4.4.1 Introduction

In this section a new technique for the production of high quality images is presented. This method follows the ray casting paradigm of Amanatides and Woo, and determines the surface

normals using the 6-neighbour method. The main difference is the fact that this method determines the *exact* surface boundary using trilinear interpolation. The normal to the surface boundary is calculated using trilinear interpolation from the gradients of the eight neighbouring voxels which make up the cube containing the resulting surface. The images resulting from this method are more accurate than those obtained using the previously discussed methods, and by using the accurate normals obtained directly from the grey level data the surface can be rendered using high quality shading. The algorithm for direct surface rendering is introduced in Section 4.4.2. The images produced using this method are compared with those produced by previous methods in Section 4.4.3.

4.4.2 Direct surface rendering of volume data sets

The algorithm is similar in principle to the ray casting algorithms of Section 4.2 and Chapter 3. The main difference to those of Chapter 3 is that it is only interested in the surface contained in the volume rather than the intensity and colour of every voxel, as determined by integrating along the ray.

It is regarded, in particular by the research of Höhne et al. [74], that volume rendering is not as useful at determining the *true* surface when such a surface exists. It is far better to visualise the surface of, for example, the skin or bone for CT scans, than obtain a view of the volume as a whole. Using volume rendering, voxels far away from surface voxels will also be contributing to the view of those surface voxels, and so give a false impression of the surface. Using the direct rendering technique an accurate view of the surface is generated with no other voxels contributing to the surface.

It is different to the techniques of Section 4.2 in that the surface is accurately determined, rather than using the coordinates of the hit voxel.

The surface is defined as a set consisting of all points in the volume space whose values are equal to a predefined threshold value τ, and it may be composed of disconnected pieces. For each pixel in the image plane, a ray is cast into the volume and is traced through cubes on its path until it hits a transverse cube. A cube, bounded by eight voxels, is said to be transverse if at least one of its voxels is inside the surface and one outside the surface. Transverse cubes can be identified by comparing values of its bounding voxels against the threshold value τ and setting an eight bit flag, $b_8b_7b_6b_5b_4b_3b_2b_1$, as

$$b_i = \begin{cases} 0 & \text{if the value of voxel } i \leq \tau; \\ 1 & \text{if the value of voxel } i > \tau. \end{cases} \tag{4.13}$$

A cube is transverse if its flag is of the binary value between 00000001 and 11111110, and Figure 4.5 shows such a cube. Once a transverse cube is found, a further check is made to see if the ray intersects the surface contained within the cube. The values at the ray entry and exit points, namely α and β respectively, are calculated using an interpolation function $f(\delta)$. Given a point δ lying on one of the six square facets of a cube, $f(\delta)$ is defined as the bilinear interpolation of the values associated with the four voxels which bound the facet. The ray intersects the surface if the ray span defined by α and β is transverse (Figure 4.6),

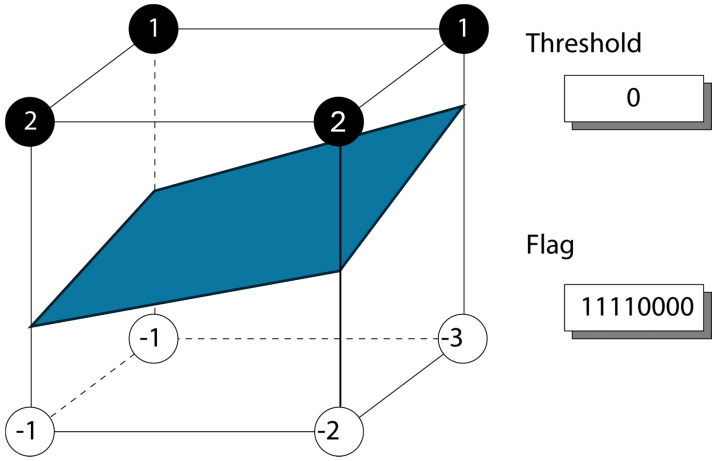


Figure 4.5: Determining if a cube is transverse.

that is, $f(\alpha) \leq \tau \leq f(\beta)$ or $f(\beta) \leq \tau \leq f(\alpha)$ The actual intersection point γ on the surface is calculated as

$$\gamma = \alpha + (\beta - \alpha)\left(\frac{f(\gamma) - f(\alpha)}{f(\beta) - f(\alpha)}\right) \tag{4.14}$$

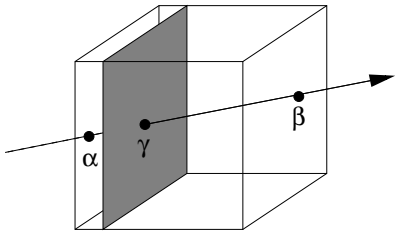


Figure 4.6: Calculating the surface intersection point γ.

Once the intersection point γ has been found, the surface normal at γ, and hence the intensity, can be computed. The normal at γ is calculated by trilinearly interpolating from the gradients of the eight voxels bounding the cube. The gradient, N at a voxel located at (i, j, k) is calculated using the difference approximation of Equation 4.11. With the normal, a shading technique, such as Phong shading, can be used to calculate the intensity at γ using the lighting equations of Section 4.3. Textures can also be mapped onto the surface for evaluating a second function in the data set. This is of particular use when each voxel is associated with a vector of samples. In computational fluid dynamics, for example, a surface representing constant pressure in a volume can be displayed with temperature mapped on to it.

A substantial amount of computation is required for tracking a ray through the volume and testing whether or not cubes are transverse along the path. To reduce this, cubes in a volume are examined in a preprocessing step. This results in a volume of binary data, where each bit indicates whether or not a cube is transverse. The speed of voxel traversal can also be improved using one of the two algorithms in the literature, namely the fast voxel traversal

algorithm [41] and the octree-based algorithm [43]. The fast voxel traversal algorithm allows a very quick traversal with just 2 floating point comparisons, 1 floating point addition, 1 integer addition, 2 integer comparisons for each transverse cube and a bit test and an integer comparison for each non-transverse cube. If a transverse cube k is encountered, the entry point α_k is calculated and $f(\alpha_k)$ is linearly interpolated. A flag is set to indicate the entry point to the next cube to be examined, regardless of whether it is transverse or not. The ray is then continued into the next cube $k+1$ where again the entry point α_{k+1} and $f(\alpha_{k+1})$ are calculated. The ray entry and exit points and their associated values are now known for the ray span in cube k and the surface intersection point can be determined. If the ray span is not transverse, the traversal continues with the next cube, for which the calculated entry point is already known. The octree-based traversal algorithm makes use of the octree data structure to skip over empty areas of a volume. This allows many non-transverse cubes along the ray path to be jumped over in a single operation. The drawback of this algorithm is that many calculations are required to find out which part of the octree the ray enters next when it leaves the previous cell. It has been found that this cost outweighs the cost of traversing more voxels using the previous method.

4.4.3 Comparison of methods

The depth only shading (Section 4.3.2), gradient shading (Section 4.3.3), 6-neighbour shading (Section 4.3.6) and 26-neighbour shading (Section 4.3.6) methods are compared with the new direct surface rendering method in this section. The methods are all used on three data sets, the CThead data set from the University of North Carolina, Chapel Hill, the Hydrogen data set from AVS Inc. and an artificial data set of a sphere.

The depth shading produces images that lack any surface detail whatsoever (Figures 4.7(a), 4.9(a) and 4.11(a)). The shape of the objects can be determined, along with some cues as to what is close to the viewer, and what is far away. Apart from that there is very little else that can be determined from the image about the object in question. The gradient shading definitely improves matters as can be seen in Figures 4.7(b), 4.9(b) and 4.11(b). The curvature of the surface is now obvious, and is a result of the fact that lighting has produced appropriate shading cues. The problem with this method is that the blockiness of the surface is apparent. This is a result of the underlying voxel model determining the surface normals rather than the surface represented by the voxel model itself. The situation is improved greatly by the use of surface normals produced by the grey level data as depicted in Figures 4.7(c), 4.9(c) and 4.11(c). In these cases the six neighbours of a voxel adequately describe the surface and produce an image that seems to be quite accurate. Problems arise in the region of thin objects, but those could be overcome by using the adaptive techniques for producing the gradient. The surface is not quite as smooth as it could be as shown by the images of Figures 4.8(a), 4.10(a) and 4.12(a) where the surface normals have been produced by considering all 26 neighbours of a voxel. The result is a much smoother surface, as can be seen in particular on the smooth sphere, and the top of the skull. The problem with enlarging the region from which the gradients are produced is the fact that more thin surface details and thin surfaces will be lost than that of the 6-neighbour method. This can be seen clearly in the skull and hydrogen images where larger regions of black occur. In comparison the direct surface rendered images are far superior (Figures 4.8(b), 4.10(b) and 4.12(b)), producing in

the case of the sphere, a perfect image. The advantage of having accurate surface boundary detection and accurate normal computation is particularly obvious in the Phong shading of the image.

In addition to being a high quality rendering technique, the direct surface rendering technique does not suffer from the coherence problems that the previous methods do. The most obvious occurrence of the problem is when an animation of the data is produced. Using the previous methods strong aliasing patterns would be apparent in the data. This is a result of either the shading being dependent upon the view of the data, or the fact that the surface is not accurately determined. Using direct surface rendering, the surface and shading parameters are determined accurately and do not vary greatly for small adjustments of the camera (viewer's) position and direction. The result is that coherent animations can be produced using this method, without any of the aliasing problems usually associated with animations of such data. See accompanying video for animations of a skull rotating, brain rotating, an animation of the hydrogen data set with varying threshold (τ) ($10 \leq \tau \leq 254$), and an animation of the superoxide dismutase data set with varying threshold (τ) ($1 \leq \tau \leq 179$).

We draw the conclusion that this method produces the most accurate, and visually pleasing images of all the methods and as such allows effective visualisation of the underlying data.

As an example of some of the images that can be produced using the direct surface rendering method see page 91. In five of these images, several differently coloured partial mirrors have been placed at the edges of the data set, and in all of the images one bi-directional light has been used. The top left image is that of an artificial data set of a sphere with blue, white and green mirrors. Note the sharp high-light from the Phong shading and the smooth gradual change from light to dark across the sphere. In the top right image the AVS lobster data set has been used to demonstrate the fine detail that can be displayed using this method. The AVS hydrogen data set has been used in the middle left image. The remaining images have been produced using the public domain data sets from the University of North Carolina of superoxide dismutase, CThead and MRbrain. The mirrors make both the brain and the face of the MRbrain data set visible in the same image. These images show that the direct surface rendering method can be used not only to present data accurately, but also in a visually pleasing way.

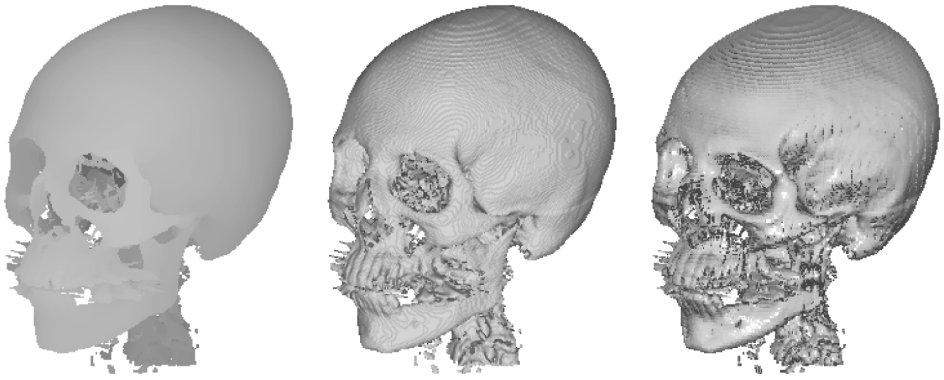


Figure 4.7: Shading of CThead (a) Depth only (b) Gradient (c) Grey-level data.

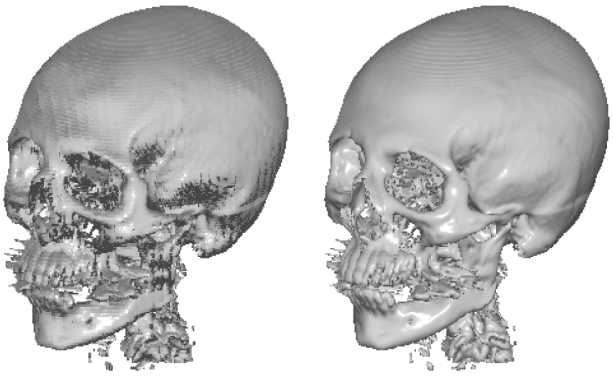


Figure 4.8: Shading of CThead (a) 26-neighbours (b) Direct surface rendering.

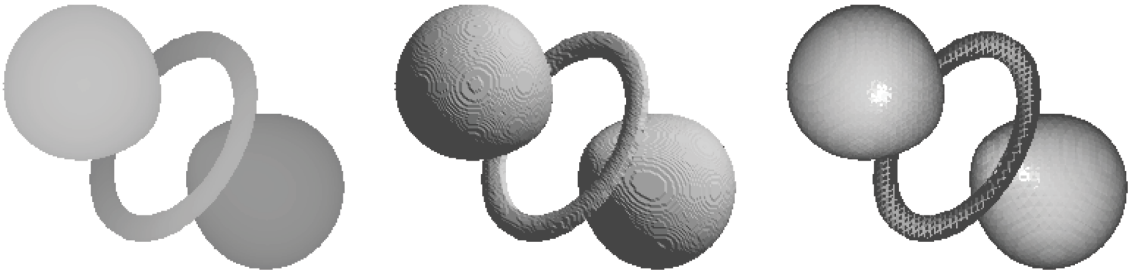


Figure 4.9: Shading of Hydrogen data (a) Depth only (b) Gradient (c) Grey-level data.

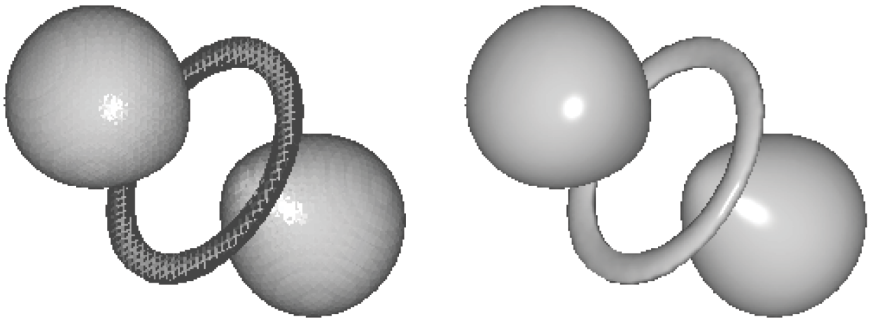


Figure 4.10: Shading of Hydrogen data (a) 26-neighbours (b) Direct surface rendering.

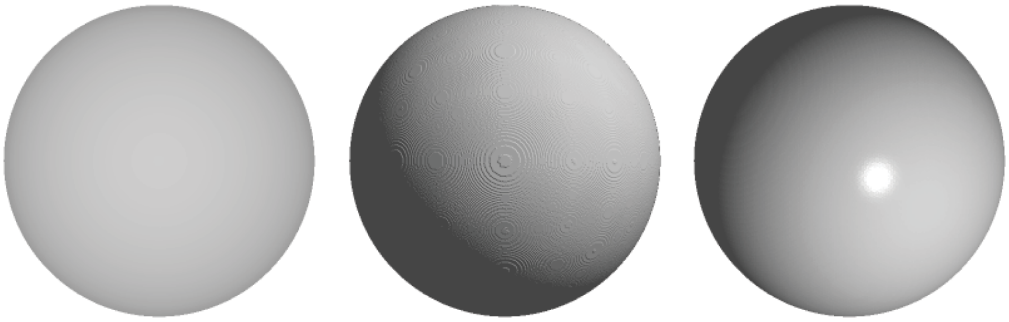


Figure 4.11: Shading of sphere (a) Depth only (b) Gradient (c) Grey-level data.

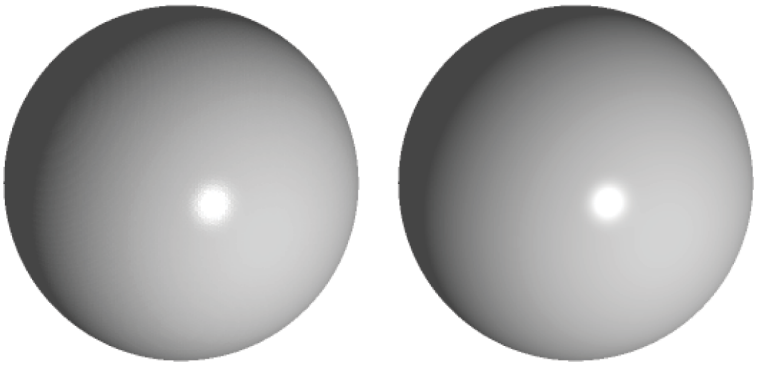


Figure 4.12: Shading of sphere (a) 26-neighbours (b) Direct surface rendering.

4.5 A New and Efficient Method for Real Time Cutting Operations

4.5.1 Introduction

Volume visualisation has a variety applications in the fields of medical imaging, flow visualisation, seismic studies, and microscopy amongst others. In many applications, a user often wishes to see different parts of the data, such as the interior of a surface, a cross section or a nested object. Therefore it is necessary to provide the user with interactive cutting operations for removing obstructing parts of the volume. Although basic techniques for volume visualisation have been well studied during the past few years, no method has yet been found in the literature for allowing real time cutting operations on large data sets. Some researchers approach the problem by using costly custom hardware to render objects as close to real time as possible, whereas others concentrate on approaching the problem on the software side.

This section introduces a new software method which allows real time cutting operations for displaying surface interiors. The background for existing cutting techniques is given in Section 4.5.2, and the new method which allows real time cutting techniques is described in Section 4.5.3. This method uses the direct surface rendering method which was presented in Section 4.4.

4.5.2 Background

Existing volume visualisation methods include surface reconstruction (Chapter 2), direct volume rendering (Chapter 3), and forward projection (Chapter 3; Section 3.7.2). The surface reconstruction method operates on cubes, each composed of eight voxels, where each voxel represents a value in the 3D data set. By traversing cubes in the volume, it reconstructs a mesh composed of triangles with each triangle's vertex on the isosurface of some predefined threshold value. The disadvantage of this method is that a triangular mesh is created which must be stored and then displayed. For some large data sets, such a mesh can have in excess of a million triangular facets. In addition to the overhead of storing such a mesh, it can take some time to display the mesh without the aid of a highly sophisticated hardware renderer. Cutting operations take place upon such a representation by traversing the mesh,

and intersecting each triangle with each of the cut planes. The cutting of the mesh can be accelerated by using some partitioning structure, such as a *binary space partitioning* (BSP) tree [29], but still requires much computation. The triangular mesh must then be displayed, and so a highly sophisticated hardware renderer that can cope with large triangular meshes is essential to achieve real time manipulation.

The direct surface rendering method treats the volume as a cloud of densities and produces a display image by casting rays into the volume. Each pixel in the image plane represents the accumulated intensity of light and colour reaching the eye after passing through the volume. A ray is traced through the volume for every pixel in the image plane, sampling the volume at regular points where values such as colour and opacity are linearly interpolated. The colour and intensity of each pixel are obtained by compositing the sampling values along the corresponding ray. The method suffers from the drawback that extensive calculation is required to interpolate values at every sampling point, and thus is not suitable for real time operations.

Direct volume rendering does allow surface interiors to be visualised using an opacity map that defines wholly or partially opaque voxels. As the opacity map is usually defined as a function of voxel values, the method cannot deal with the situation where an object of interest is of the same value as an obstructing object. An alternative way of defining an opacity map was proposed by Ma et al. [62], with which nested objects can be displayed by increasing opacity around the object of interest and decreasing opacity away from it. However, in many applications, the shape of an interesting object is often arbitrary and its position is usually unknown. Moreover, with any algorithm based on direct volume rendering, the whole volume has to be retraced when the opacity map changes.

The forward projection method projects voxels in the volume directly onto an image plane where each voxel is drawn as a footprint. A footprint may represent the energy density or the projected shape of a voxel, and is composited with all other footprints on the image plane with an intensity determined by the voxel location. Though the method is considered to be faster than the surface reconstruction and the direct volume rendering method, the resulting images generally lack accuracy and quality. Any operations on the data set require the whole data set to be traversed and re-projected onto the image plane.

4.5.3 Real time cutting operations

A cut is defined by a set of cutting planes, C_1, C_2, ..., C_m whose plane normals are N_1, N_2,..., N_m respectively. Given a viewpoint and the position of an image plane, two depth buffers, namely the near depth buffer D_{near} and the far depth buffer D_{far}, can be constructed from the cutting planes. Both depth buffers are of the same resolution as the image plane, and each element of the buffers corresponds to a pixel in the image plane. Elements in D_{near} are initialised to zero and those in D_{far} are initialised to a maximum value MAX_DEPTH. For each pixel p in the image plane, cutting planes are classified into three groups according to the ray $R(p)$ cast from the pixel. They are planes parallel to the ray, (that is, $R(p) \cdot N_i = 0$), planes facing the same direction as the ray ($R(p) \cdot N_i > 0$) and planes facing the opposite direction ($R(p) \cdot N_i < 0$). For a plane C_i that is parallel to the ray, we set

$$D_{near}(p) \leftarrow MAX\_DEPTH, \text{ and } D_{far}(p) \leftarrow 0 \tag{4.15}$$

if p is in the negative halfspace defined by the plane, that is, $C_i(p) < 0$. For a plane facing the same direction as the ray, the depth of the intersection point between the ray and the plane is computed and $D_{near}(p) \leftarrow max(D_{near}(p), depth)$. Similarly, for a plane facing the opposite direction, the depth of the intersection point is computed and $D_{far}(p) \leftarrow min(D_{far}(p), depth)$.

These two depth buffers are then used to compare with the surface points found by casting rays into a volume to determine whether or not a surface point should be displayed. Everything of a depth in front of the near depth buffer and behind the far buffer, is cut away. Figure 4.13 shows an example where a cut is made on an outer sphere to reveal the inner sphere.

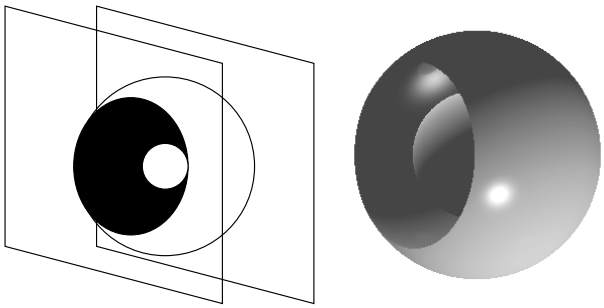


Figure 4.13: A cut revealing an inner sphere.

In order to avoid the whole image needing to be retraced whenever a change is made to the cutting planes, the direct surface rendering algorithm discussed above casts rays into the volume, keeping track of all surface intersections, until the ray exits the volume. An intersection buffer is used to store information about all surface intersections for each ray, which includes the intensity resulting from shading a surface point and a depth for each surface intersection along the ray. The layout of the intersection buffer is illustrated in Figure 4.14.

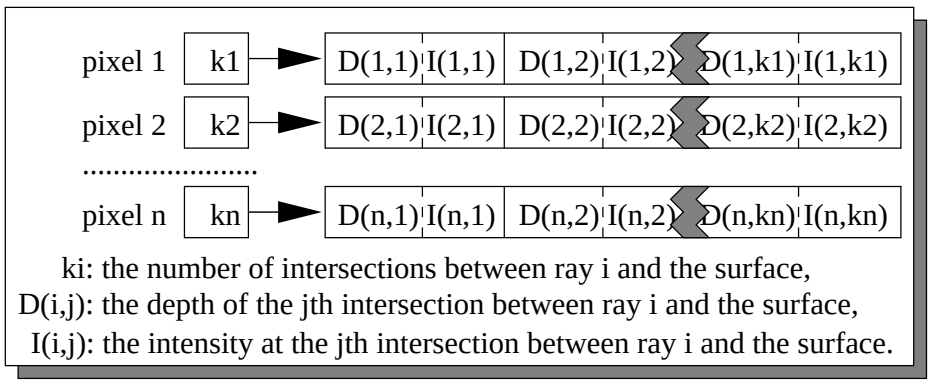


Figure 4.14: The layout of the intersection buffer.

With this method, a user is allowed to specify interactively a combination of cutting planes. These planes are then scan-converted to form the two depth buffers. The image is produced

by simple comparison operations. For each pixel in the image plane, the depth of each intersection is compared with the near and far depth buffers until the first valid intersection is found, and the pixel is drawn according to the stored intensity. After the initial preprocessing stage to create the intersection buffer, there is no need to retrace rays through the volume at all unless a new viewpoint is required. In addition to intensity, other quantities could be stored, such as surface normals, which could allow the user to move lights over the object interactively. Multiple surfaces with different thresholds can be stored in the buffer along with transparency and colour information, allowing semi-transparent surfaces to be manipulated.

4.5.4 Testing

The direct surface rendering algorithm, together with cutting operations, has been implemented in C on a DEC Alpha 3000 model 400 workstation and tested on various known data sets. The data sets used include CThead (Figure 4.15(a)) and MRbrain (Figure 4.15(b)) from the University of North Carolina, and hydrogen (Figure 4.15(c)) from AVS^{TM}. The timing obtained for a 300x300 image plane and the space usage is reported in Table 4.1.

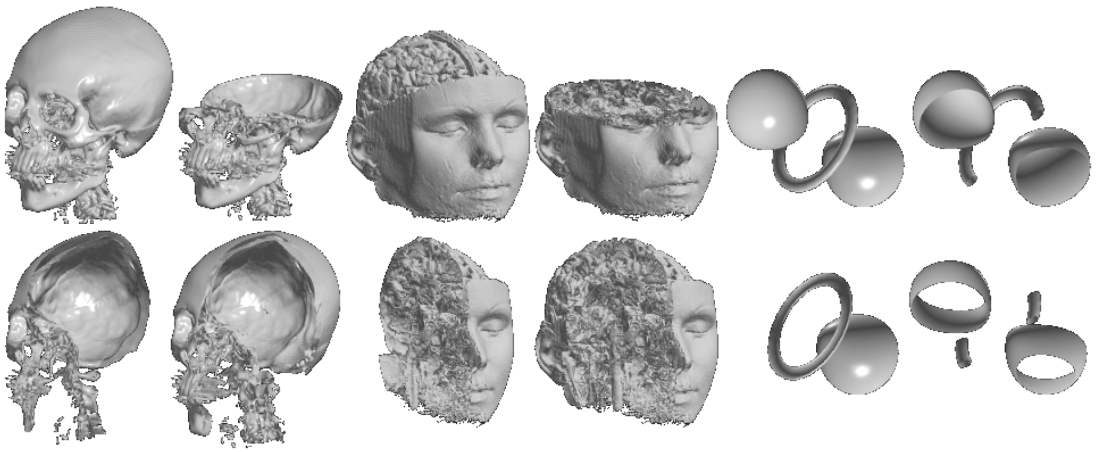


Figure 4.15: Cuts performed on (a) CThead (b) MRbrain (c) Hydrogen.

In Figure 4.15(a), a cut has been made in order to reveal the interior of the skull. Figure 4.15(b) demonstrates that cutting operations can be used to visualise interior surfaces with the same threshold as the exterior surface. In this case the texture of the brain, and the nasal passages are clearly visible. As shown in Figure 4.15(c), by specifying a pair of cutting planes a band of the surface can be displayed. The computational times for producing each image from the intersection buffer are given in the extract image column. The time taken to extract an image depends upon the complexity of the surface and values in the depth buffers. The more intersections there are along the ray before a valid intersection is found (or the ray passes beyond the far depth buffer), the more comparisons are required. The marching cubes column gives the time taken to extract a surface at the same threshold value and the storage size of the polygon mesh is given in the final column.

As can be seen from Table 4.1 interactive cutting operations can be performed on volume data sets. This has been achieved since the display of an image requires merely simple comparison

Fig. No.	Cast Image (secs)	Cast Buffer (secs)	Extract Image (secs)	Buffer Size (MBytes)	Marching Cubes (secs)	Mesh Size (Mbytes)
4.15(a)	21.35	43.50	0.057	2.13	29.28	12.37
Top right			0.087			
Bottom left			0.093			
Bottom right			0.093			
4.15(b)	18.98	49.86	0.06	3.96	39.20	24.68
Top right			0.12			
Bottom left			0.15			
Bottom right			0.13			
4.15(c)	12.35	14.07	0.03	0.56	0.80	0.24
Top right			0.060			
Bottom left			0.056			
Bottom right			0.033			

Table 4.1: Testing results

operations on two depth depth buffers constructed from cutting planes and an intersection buffer which has been set using the direct surface rendering algorithm in a preprocessing stage. Real-time cutting provides users with flexibility during the visualisation of complex volume data sets.

4.6 Conclusion

This chapter concentrated on the production of images from voxel data by directly visualising the voxel data. This is in contrast to the methods introduced in previous chapters which visualised such data by creating an intermediate surface which can be displayed (Chapter 2) or by sampling the data along rays (Chapter 3).

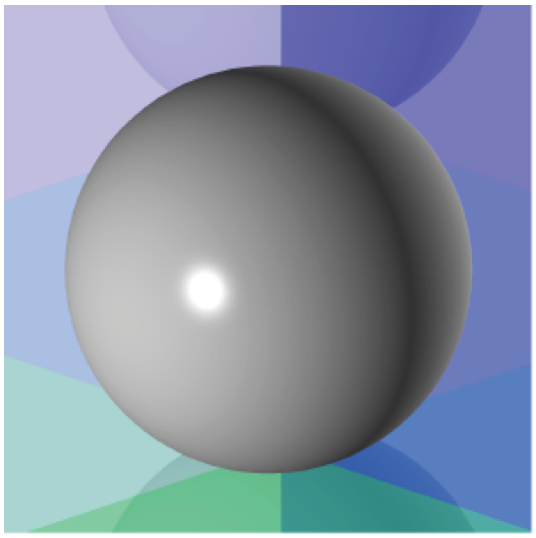
In Section 4.2 algorithms to extract and display objects embedded in voxel data were reviewed. The shading methods available for the presentation of such models were described in Section 4.3 and presented with the aid of the standard lighting equation.

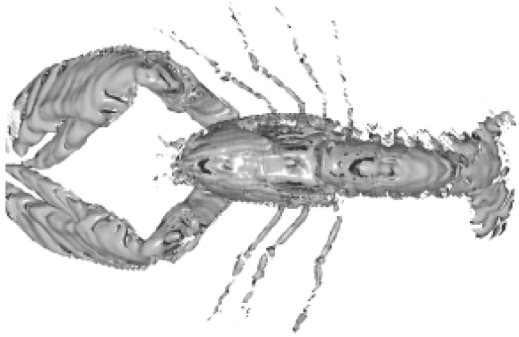
These methods were compared with the method of Section 4.4 which generates images of the highest quality. Those images are also coherent during the animation of the object, which was shown by the video produced of a skull rotating through 360^o.

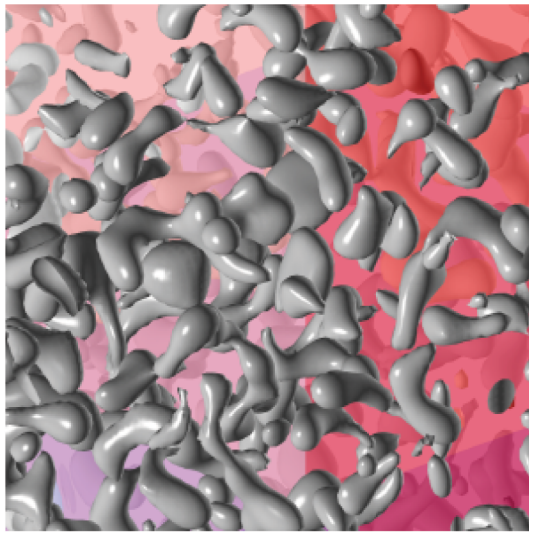
Interactivity was addressed in Section 4.5, where a method was presented which allows real time cutting operations to be carried out upon volume data. The method uses the information created by the direct surface rendering technique of Section 4.4, to produce high quality images which can be manipulated in real-time through the use of an intersection buffer.

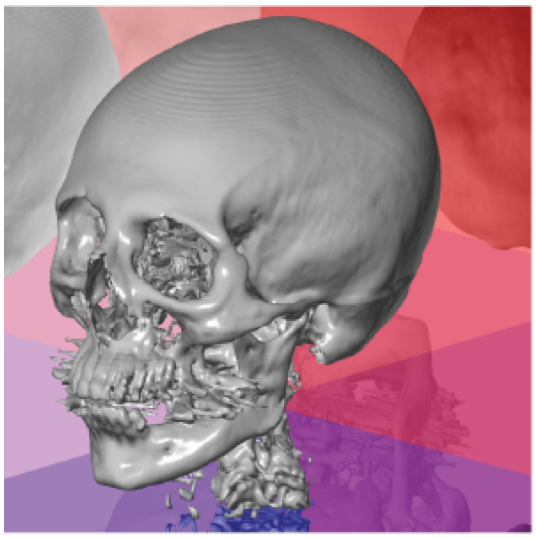
This work [1] was presented at the Eurographics workshop for Visualisation in Scientific Com-

puting 1994 (Rostock, Germany). It also appeared in "Visualization in Scientific Computing", pp 1–8, edited by M. Göbel, H. Müller and B. Urban, and published by Springer-Verlag.









Chapter 5

Volume Construction Techniques

5.1 Introduction

Previous chapters have concentrated on the production of images using techniques for volume visualisation. The volume data that has been visualised has been data that has been either calculated or scanned. These data sets can be considered to be *real* in the sense that they accurately simulate some process, or accurately describe some scanned object. This chapter investigates methods that construct *artificial* volume data. Artificial is meant in the sense that the objects represented by that data do not exist, or only exist as a computer model.

Sections 5.2 to 5.3 show how soft or blobby models are used to produce images of realistic natural objects such as liquid, flexible materials or skin. The theory behind such models is presented (Section 5.2.1), and the model is applied in order to produce accurate visualisations of simulations of liquid droplet movements (Section 5.3).

This chapter introduces the process of *voxelisation* (Section 5.4), in which objects from one source representation, are converted into volume data. Voxelisation can take place most easily on mathematical representations of objects, such as cones, spheres, quadrics and planes, and in contrast to this, Sections 5.4 to 5.6 describe a method for the voxelisation of data from triangular meshes. The volume visualisation technique can then be applied to the data in order to produce a graphical representation of the object. These sections give a practical approach to the voxelisation of data in the form of triangular meshes, and demonstrates the use of the method on various data sets.

5.2 Soft Objects

5.2.1 Introduction

A new class of objects was introduced by Blinn [79] which are often referred to as blobby objects. This class of objects has been found to be ideal for representing objects not usually represented well by computer such as soft tissue, flexible objects, cloth, liquids and natural objects.

Animations of such objects show fluid movements, and smooth blending. Liquid droplets can be animated so that they seem to move together naturally, preserving the volume of the liquid. Atoms modelled in this way have realistic looking electron clouds which stretch apart as the atoms separate. Flexible objects such as limbs and jointed objects move naturally, retaining a smooth surface during the movement. Musculature can be approximated using such objects, which can move realistically. The descriptions of these types of objects often contain words such as *natural*, *smooth* and *blended* because it is the representation of this class that these objects are most suited to.

They are a subset of the class of surfaces known as *implicit* surfaces. An implicit surface is defined according to the equation

$$F(x, y, z) = 0$$

and includes quadrics – ellipsoids, cylinders, planes etc., and superquadrics.

The field value at any point (x, y, z) created by a primitive R at a point (x', y', z') with strength b and decay a is expressed as:

$$D(x, y, z) = be^{-ar}$$

where $r = \sqrt{(x - x')^2 + (y - y')^2 + (z - z')^2}$.

The field value at any point created by a collection of primitives R_i is:

$$D(x, y, z) = \sum_i b_i e^{-a_i r_i} \tag{5.1}$$

The surface is defined using the function $F(x, y, z) = D(x, y, z) - \tau$. In other words the surface is an isosurface of threshold τ. As described by Blinn [79], Equation 5.1 represents the sum of each Gaussian bump centred at (x_i, y_i, z_i), with height b_i (strength) and standard deviation a_i (rate of decay). The variables a_i and b_i alter the *blobbiness* of the object.

In the case of superquadrics:

$$r = (x - x_i)^{\frac{2}{a_i}} + (y - y_i)^{\frac{a_i}{b_i}} + (z - z_i)^{\frac{2}{b_i}}$$

Blinn [79] describes a method of *rasterising* the blobby models which is similar to ray tracing primitives except that the ray/blob intersection points must be found numerically rather than analytically. An alternative is to evaluate the function at discrete points over a volume array, and use the isosurfacing technique to display the objects.

The isosurfacing method was used by Wyvill et al. [12] who proposed *soft objects*. These soft objects are similar to Blinn's blobby models except that the function they use becomes zero at a finite distance away from the placement point, and only approximate the exponential function. The use of a function with a finite field of influence reduces the complexity of the calculation algorithm since rather than requiring the field to be calculated for every voxel with every primitive as in the blobby model, *soft objects* only require the field to be calculated for those primitives by which they are influenced.

An approximation of the exponential function reduces the computation required to calculate the field. The function Wyvill et al. use is

$$C(r) = a\frac{r^6}{R^6} + b\frac{r^4}{R^4} + c\frac{r^2}{R^2} + 1$$

where $a = -0.444^*$, $b = 1.888^*$, $c = 2.444^*$ and r is the distance of the voxel from the centre of the primitive. $C(r)$ drops to zero at the radius of influence R, and has the advantage of being very cheap to compute compared to the exponential function.

5.2.2 Background

There is much work on the use of soft or blobby objects for the production of natural looking models, and the work of Opalach and Maddock [80] provides a good overview of the area. This section does not attempt to provide a review of the existing work, but does review some more up to date and varied research.

Muraki [81] used blobby models to produce a shape description of range data. The goal was to produce an accurate surface from range data using as few primitives as possible. Muraki found that the blobby model was ideal since very few blobs can describe quite complex objects. The object can be rendered by generating an isosurface from the model which can be compared to the range data. The process of comparison can provide a measure of how successfully the model description matches the range data, and by using an appropriate energy function, the process of minimisation can introduce and reposition blobs so that the isosurface closely approximates the surface described by the range data.

In the example images Muraki [81] shows that very few primitives can accurately model complex data. In one figure a bust of Mr. Spock is quite recognisable, using just 451 blobs. The main drawback is that the process is computationally expensive – taking "*a few days*". It is interesting to note that Muraki went on to work with wavelet transformation for volume data [82] which achieves a similar result – the representation of a complex object with few primitives – although it is far quicker to compute, taking minutes rather than days.

Volume morphing of data is a challenging research area, and the representation of objects by blob primitives results in a simple volume morphing algorithm. Each blob in one model is mapped to a blob or blobs in the second model. Parameters are adjusted for primitives such that they smoothly interpolate from one model to the second. Although no research has been found in the literature on this subject, there is a similar process that works on Muraki's wavelet transformation of volume data, by He et al. [83]. When comparing the result to somewhat simpler techniques, the inbetween volumes produce smooth connected surfaces, rather than surfaces with small disconnected parts that is more characteristic of the spatial domain volume morphing techniques [84].

One of the main problems associated with soft or blobby objects is that of interactively displaying and manipulating them. Although Opalach and Maddock [80] go some way to addressing this problem, there is still some difficulty in producing visualisations in real time. Most acceleration methods [80] aim to produce triangulations of the implicit isosurface from sampled data (Chapter 2), and fail to achieve interactive rates. One interesting alternative

is that of Witkin and Heckbert [85] where implicit surfaces are represented by particles displayed as disks. The particles can control the surface, in that particles can be *pulled* by the mouse and affect the underlying functions, and the surface can control the particles in the sense that their positions are influenced by the functions. The surface is sampled by these particles by placing a few randomly on the surface. They are then repelled from each other until an even distribution is achieved, upon which particles in less dense areas are split, randomly perturbed, and re-distributed. This adaptive process ensures that a good sampling of the surface is achieved, and also that a general impression of the surface is gained before the remaining fine details are filled in. The distribution is achieved by minimising energy constraints for the particles based upon the density in the local area and their mutual repulsion. Such a process proves to be effective, and Witkin and Heckbert [85] report that the surfaces can be sampled and manipulated interactively.

Blobby models have been used to simulate flexible objects melting by Terzopoulos et al. [86]. They describe the construction of a deformable solid as a system of particles connected by springs. Eight neighbouring particles making up a brick (hexahedral element) are connected using 12 springs along the edges, and 2 diagonal in each of the 6 faces. These springs model the flexibility of the model with respect to the external forces, the object's mass, and the tendency of the elastic object to restore its natural shape. External forces such as friction and processes such as non-penetrating surfaces are also simulated. By investigating the laws of thermodynamics the springs can be used to approximate the thermoelasticity (softening) and melting characteristics of heat-conducting deformable models. The spring's stiffness alters according to temperature to simulate softening, and by fusing springs, particles can become free components in a fluid, and may interact with other particles. The particles attract and repel each other so that collisions and fluid movement are effectively handled. The example they present [86] is that of a flexible material hitting a cool funnel. The funnel is heated and transfers the heat to the object. The object softens, and sags into the funnel, and as the temperature rises further it melts and runs out of the bottom of the funnel. The animation looks very natural, but one visual problem is the fact that the particles are large, and are visible in the *fluid* state. The extensions they suggest for the work is to introduce other physical effects such as to simulate heating through deformation or friction. By introducing a boiling point the transfer from a liquid to a gas could also be achieved. This work and the continuation of this work shows significant promise in the production of visualisations that closely resemble the physical world. This *physically based modelling* is an ongoing, challenging and interesting research area.

Much of the work has concentrated on using blobs for natural objects with one interesting exception. Reed and Wyvill [87] have used blobby models to simulate the natural phenomenon of objects struck by lightning. The lightning itself is described by a collection of connected finite length rays with which a particle system is used to generate the segments of a lightning strike, and then animate it. The objects struck by the lightning are represented by blobs which are made to glow by using the scalar field around the object to indicate brightness. They choose ellipsoids to create the implicit function F from which the surface is created as all those points p where $F(p) = 0.5$. They create the glow around the object in the volume defined as all those points p where $0.0 < F(p) < 0.5$. The glow is determined by calculating the closest approach of each ray to each ellipsoid and then calculating the maximum implicit value for all of those points. It is this value that determines the glow around the object. They

derive their method from the physical model of lightning, and adjust it to produce visually pleasing effects. The addition of the glow around the objects struck by lightning is achieved using the implicit field, and is an unusual and interesting application of blobby models.

5.3 Display of Implicit Surfaces

5.3.1 Introduction

There are two distinct approaches to the display of implicit surfaces, both of which are computationally expensive. The first approach is that of calculating the contribution of each primitive to each point in a regular 3D array. The surface can then be determined by extracting a polygon description of the isosurface of a value τ. This technique is expensive in that not only does the field have to be calculated, but also the isosurface must be generated (Chapter 2) and displayed.

The alternative approach is to calculate the closest approach of each object against each ray during ray tracing. Ray tracing has the advantage of allowing geometrical information from different sources to be easily combined with the blobby models. The closest approaches are evaluated in order to determine the implicit function value at each approach point, and then a numerical method such as Newton iteration is used to determine the exact point of the surface where the function has a value τ along the ray. The problem with this method is that many calculations are needed to determine the roots of the surface equation along each ray.

In this section the Direct Surface Rendering technique of Chapter 4 is applied to the visualisation of blobby objects. The grid of values is calculated for a scene, and the volume data is *ray traced* using the direct surface rendering technique. This has the advantage of replacing the complexity of finding exact surface intersections using numerical methods during ray tracing by stepping the ray through cubes. The accuracy of ray tracing is still retained. The advantages of this method over polygonalisation is the fact that far more effects, such as reflections can be modelled, and a large polygonal mesh does not need to be generated and displayed.

Firstly the technique is applied to data representing two blobs moving towards each other, and finally colliding. The simulation is very much like two droplets of water coming together to form one larger droplet. The second application is a simulation of a explosion. The data was originally computed using a particle system, and then a blob was centred about each particle. Both examples serve to demonstrate that the rendering method is accurate, and that blobs are an effective modelling component for natural objects.

5.3.2 Water droplets

This section demonstrates the various blob parameters available by using a simple example of two separated primitives moving towards the same point in space. To create the animation a field of 40^3 voxels was created, and the field value was calculated from the contributions of the two primitives. The field was then direct surface rendered to produce the image. The

implicit function was evaluated according to the method of Wyvill et al. [12] (Section 5.2), where $R = 30$ and the isosurface threshold used was $\tau = 0.5$.

At the start of the animation, the two droplets are separated by some (10 voxels) distance (Figure 5.1(a)). The droplets begin to move together (Figure 5.1(b)), and eventually ooze together (Figure 5.1(c)). As the centres continue to approach, the new single droplet becomes first of all ellipsoidal (Figure 5.1(d)), and then finally a sphere (Figure 5.1(e)). This animation is included on the accompanying video.

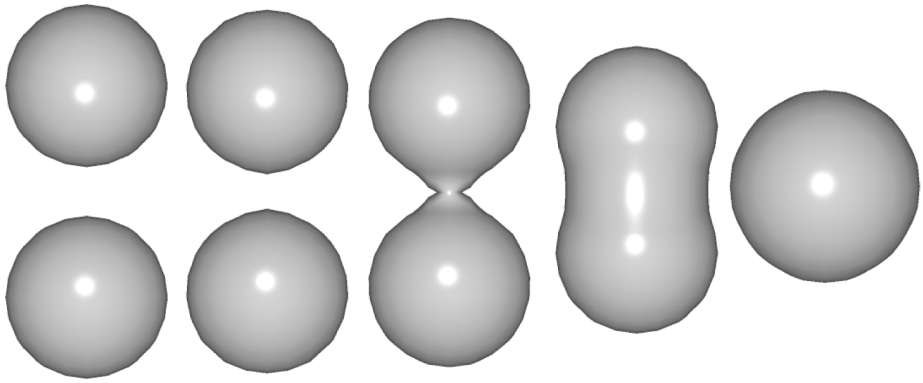


Figure 5.1: Selected frames from a 100 frame animation of droplets merging.

The merging of two water droplets is effectively modelled using blobby models, and the direct surface rendering technique works very well for the visualisation of these implicit surfaces. Each 200×200 pixel frame takes just over 2 seconds to ray trace (including field creation time) on a DEC 3000/400 workstation. The animation shows, that not only does this method produce accurate and visually pleasing images, but it also produces the images quickly. This is because the ray jumps over large intervals that contain no surface intersections with a minimum of computation, whereas with the previous ray tracing method, the surface would have to be found iteratively on each ray interval.

In Figure 5.2 two parameters are varied in order to show how they effect the blobs. R is varied vertically, and the isosurface threshold τ, is varied horizontally.

5.3.3 Explosion

The main aim of this example was to show the use of direct surface rendering for a number of moving blobs in a field. The choice of an explosion was made because it is visually impressive. The mechanism by which the blobs move in the field is controlled by using a particle system, where each blob is centred at a particle. The explosion is created by placing a number of particles at the base of the data set, and providing each particle with a random velocity and launch angle. Each particle also has a direction such that its neighbouring particles are equally angularly spaced as they race outwards from the centre of the explosion. In addition to this set up, the whole system is acted upon by gravity, and is bounded by the data set so that particles bounce off the walls. The motion of the particles is computed using the

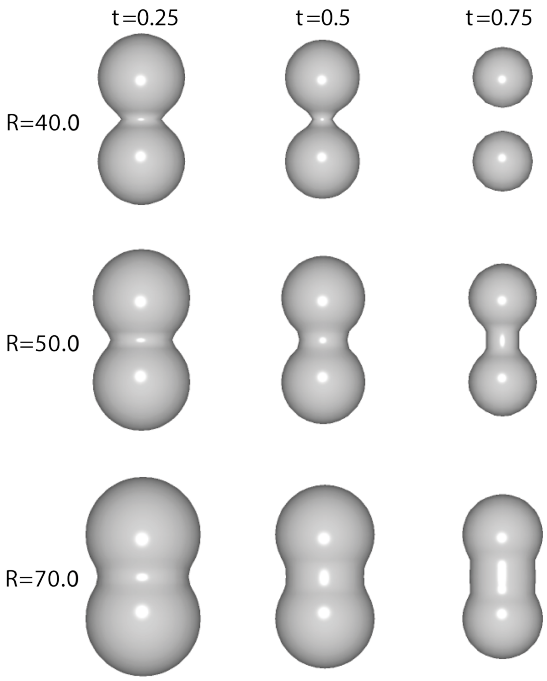


Figure 5.2: Effect of altering blob parameters.

equations of motion:

$$s = ut + \frac{1}{2}at^2$$

$$v = u + at$$

$$v^2 - u^2 = 2as$$

where s is the distance travelled, u is the initial velocity, v is the final velocity, a is the acceleration, and t is the time passed.

Given an initial velocity v and a launch angle θ, the horizontal velocity, h_v and the vertical velocity v_v are

$$h_v = v\cos\theta$$

$$v_v = v\sin\theta$$

Since gravity only acts on the vertical component, the horizontal component h_v remains constant, and the distance travelled horizontally S_x, after a time t is given by

$$S_x = v\cos\theta \times t$$

The distance travelled vertically S_y, is given by

$$S_y = v\sin\theta \times t + \frac{gt^2}{2}$$

where g is gravity (-9.81ms^{-2}).

Bounce is simulated by determining the horizontal and vertical components at the point of impact, and calculating a new launch angle and velocity based upon these components and the angle of collision with the wall.

At the start of the animation all of the blobs are at the centre of the base (Figure 5.3(a)). A few frames later they have begun to move outwards from the explosion (Figure 5.3(b)). Some particles move faster since the velocities are random within a certain interval. The paths are also determined by a random launch angle within a specified interval. These differences ensure that the blobs split apart irregularly at the start, demonstrating the *reluctance* of blobby models to part (Figure 5.3(c)). Shortly later some of the blobs reach the walls of the data set and bounce back (Figures 5.3(d–i)). The blobs continue to bounce around the volume, sometimes coalescing with each other as they pass close. This animation is included on the accompanying video.

```
50 50 50    {Size of volume data set}
       5    {Size of border}
      15    {Particles (integer)}
   0 0 5    {Initial position Z should = border (0 0 0 = centre of data
            set in x, y and z=0)}
   0 0 0    {Initial position variation in each axis}
      30    {Initial velocity in metres per second}
      10    {Initial velocity variation in meters per second}
   -9.81    {Gravity in metres per second squared}
      60    {Initial launch angle in degrees}
      10    {Initial launch angle variation in degrees}
       8    {Simulation time in seconds}
    0.05    {Time step in seconds}
     1.0    {Fraction of velocity retained after hitting a wall}
```

The animation simulates an energy preserving elastic bounce. This can be altered using a number between 0 and 1 representing the fraction of energy retained by the blob upon impact with the wall. The input file to the particle system simulator, shown above, produces the information for the 160 frame animation in a few seconds. Each file is then converted into volume data, and then direct surface rendered. Altogether it takes about 4 seconds to produce each frame using the direct surface rendering method. This compares very well to using the public domain ray tracer POVRAY which takes about 2 minutes per frame.

5.3.4 Conclusion

This section has applied the technique of direct surface rendering to the display of volume data constructed by using blobby or soft objects. The examples of the water droplets (Section 5.3.2) and explosion (Section 5.3.3) have shown that not only are the images of a high quality, but

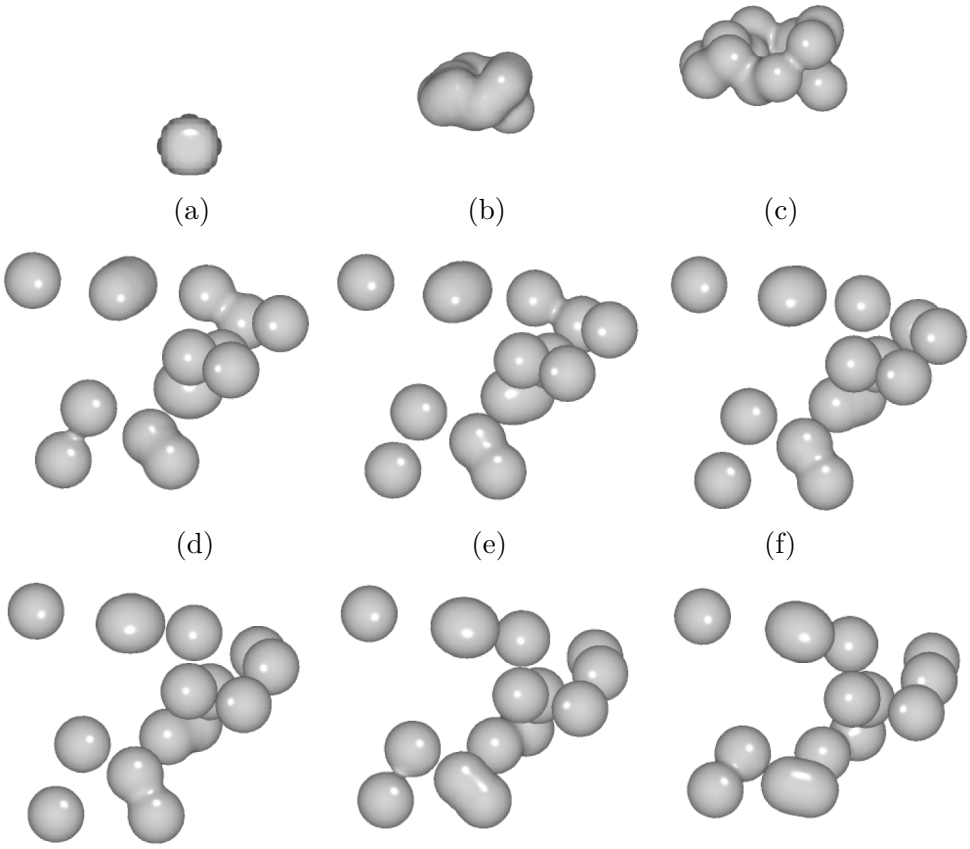


Figure 5.3: a–i. Bouncing blobs animation.

also that they are produced very quickly in comparison with a well known ray tracer. The reason for this is that the direct surface rendering method determines the surface using trilinear interpolation only within the volume element it occurs in rather than determining the surface intersection numerically from the field.

5.4 Voxelisation

5.4.1 Introduction

"... in 10 years, all rendering will be volume rendering." – Jim Kajiya at SIGGRAPH '91 [88]

As more visualisation packages and computer hardware become available for the visualisation of three dimensional data, it is becoming increasingly important to move away from surface based meshes and towards voxel data. The advantages of doing this are that surfaces from many different origins (CSG, spline based, meshes, volumetric data etc.) can be combined into one uniform format. It is especially suited to combining polygonal objects, gaseous objects (for example fog, cloud and fire), and volumetric data that has been collected by some device, for example a CT scanner. The reasoning behind this conversion into voxel data, and then volume visualisation is that the visualisation can be generalised into an all encompassing rendering process, as opposed to applying one or more of the rendering methods for each type of object description. Much of the work for rendering a scene is carried out during a preprocessing step, and then rendering can be achieved in one pass, permitting objects of the different sources and representations all to be treated in the same way and allow the consistent application of various visual effects. Data in this format can take advantage of existing software and hardware which has been optimised for volume visualisation and the result is usually a high quality, more rapid visualisation.

In the remaining part of this chapter the production of volume data from surface meshes is examined. An algorithm to convert closed triangular meshes into volume data is given along with optimisations which make the conversion realistically possible on most computers. The visualisation of this data is explored with a few examples given. The motivation behind this work is not only to develop a new step towards the goal of unifying data from several data sources, so that it may be visualised consistently, but also to produce volume data from triangular meshes in order to achieve realistic metamorphosis between volume data sets [89].

Most previous voxelisation algorithms convert solid objects into voxel data using spatial-occupancy enumeration [90, 71, 73, 29]. The result of this process is a three dimensional regular array of cubes, sometimes known as a cuberille. The main problem associated with this method is the fact that the model is limited by the resolution of the grid. The result of coarse grids is an image which is blocky in appearance due to the fact that only cube faces are displayed. Since there are only 6 cube faces, the resulting image only has 6 different normals with which to be shaded. Shading algorithms such as depth shading [76] and congradient shading [77] attempt to improve this situation, but only manage to reduce surface artifacts slightly.

Volumetric data is produced using filters [91], which can then be volume rendered to produce

anti-aliased images. The technique is described for objects such as spheres and cones, but detailed descriptions for triangular meshes are not given.

Where this work differs from previous research is that it provides the following:

- a detailed description of a practical approach to producing voxel data;
- production of voxel data which results in accurate images when rendered;
- details of optimisations which make the process realistic.

Details about the method are given in Section 5.4.2. Section 5.5 shows how the efficiency of the algorithm can be increased, and Section 5.6 discusses the results for some meshes.

5.4.2 Production of voxel data

Many methods exist for the visualisation of three dimensional voxel data, but they can be broadly divided into two categories – direct volume rendering and isosurfacing. Isosurfacing (Chapter 2) does not apply in this case since it aims to produce a surface mesh from volume data. Using this method a surface mesh would be produced from volume data, which had been produced from a surface mesh. The methods of volume rendering (Chapter 3) produce images directly from the volume data and by using accurate optical models [45, 92, 93, 94] the method can be used to effectively model various lighting effects such as scattered light, clouds and haze. Many of these methods have been implemented using hardware [95, 96, 70, 97], and as prices decrease these systems will become widely available.

Many methods produce images by converting the original values into opacity depending upon some function. This function usually states that regions of large gradient, that is object interfaces, are the most interesting parts of the volume and are to be visualised. Any voxel model produced from surface mesh data must be able to distinguish the object interface from the interior of the object and the surrounding empty uninteresting portions of the data set. One simple method would be to assign voxels a value depending upon whether they are inside or outside the object:

$$f(x,y,z) = \begin{cases} -1 & \text{if } (x,y,z) \text{ is outside the surface} \\ 0 & \text{if } (x,y,z) \text{ is on the surface} \\ 1 & \text{if } (x,y,z) \text{ is inside the surface} \end{cases} \tag{5.2}$$

The advantage of this function is that a three dimensional scan conversion algorithm can be employed to determine the state of each voxel within the domain. This results in a fast conversion of the object into voxel data, but the surface produced is highly dependent upon the density of the chosen voxel data. Images produced from such data often exhibit a blockiness [74] because a binary decision is being made at each voxel – it is either inside or not. In order to get a smoother description of the surface, a function which varies smoothly in the region of a surface is required. A far better function is the distance function defined as:

$$f(x,y,z) = \begin{cases} -dist(x,y,z) & \text{if } (x,y,z) \text{ is outside} \\ 0 & \text{if } (x,y,z) \text{ is on} \\ dist(x,y,z) & \text{if } (x,y,z) \text{ is inside} \end{cases} \tag{5.3}$$

where $dist(x,y,z)$ is the distance from (x,y,z) to the closest point on the surface. This results in a volume that represents a continuous function sampled at discrete voxel locations, that is not so dependent upon the density of the voxel data. This data can be visualised using the direct surface rendering method of Chapter 4; Section 4.4.

5.5 Increasing the Efficiency of Voxelisation

The main problem with a naive approach to the production of the voxel data using the distance function is the fact that the complexity of the algorithm is so high. If the number of triangles in the mesh is N and the size of the voxel data set in each dimension is X, Y, and Z respectively, the distance function is calculated $NXYZ$ times.

Taking a mesh of 100 triangles and a volume of 100^3 voxels, for example, would result in 100 million calculations of the distance function. Since each function computation involves calculating the distance to a three dimensional triangle, and involves at least a square root, this is costly in terms of computer processing. Three areas for acceleration can be identified:

- reducing the number of voxels for which the distance has to be computed,
- making the distance calculation as efficient as possible,
- reducing the number of triangles which the distance has to be computed with.

These three areas are all addressed in the next sections.

5.5.1 Reduction of distance function calculations

By simple observation it can be seen that it is not necessary for the value of every voxel to be accurately computed. In the direct surface rendering phase, the algorithm examines each cube, bounded by eight voxels, along a ray path. When it encounters a cube which contains the object interface it determines the values at the ray entry and exit points. If those values indicate there is a surface interface along the ray, the position of the surface is determined, and a surface normal is calculated. Voxel values at the eight vertices of the cube need to be known in order to determine the surface. The 6-neighbours (two in each of the x, y and z directions) of each voxel also need to be known in order to calculate the surface normal. In other words, only the values of voxels near the actual surface are required in order to display the surface. At all other voxels an indication of whether the point is inside or outside the surface is sufficient. This leads to a practical method for reducing the computational cost by restricting the distance computation to those voxels near the surface interface.

Each voxel is associated with two fields, namely state and distance. The state field indicates whether a voxel is interior (1), exterior (-1) or on the surface mesh (0), and is first computed by scan-converting the object. The state function (Equation 5.4) can be calculated for each voxel (x, y, z).

$$f(x,y,z) = \sum_{k=z-1}^{z+1} \sum_{j=y-1}^{y+1} \sum_{i=x-1}^{x+1} \text{state field of voxel } (i,j,k) \tag{5.4}$$

For a voxel (x, y, z) there is no need to apply the distance function (Equation 5.3), if:

- its state field is zero, or
- $f(x, y, z) = 27$ or -27 and $f(x', y', z') = 27$ or -27 for each 6-neighbour (x', y', z').

It is obvious that there is no need to apply the distance function when the state field is zero which indicates the voxel is on the surface. The second condition shows that if a voxel and all its 26-neighbours are all interior (or all exterior) to the surface and all the 26-neighbours of its 6-neighbours have the same state, its distance value will not influence the surface display in any way. The first part states that a voxel need not be known if it is not used during linear interpolation of position, and the second part states that a voxel need not be known if it is not used during determination of the surface normal. This process eliminates many voxels from the expensive distance computation and identifies the voxels, and only those voxels, for which the distance function must be calculated.

If the number of voxels on each slice for which the distance must be calculated is R_i, the number of distance calculations is now $N \sum_{k=1}^{z} R_k$. Usually R_i is less than both X and Y, and therefore this method results in a reduction of an order of magnitude of the number of distance calculations.

5.5.2 Point to triangle distance calculation

During the computation of the voxel data, the distance from each voxel to the surface must be found. This requires the calculation of the distance between a 3D point, P_0 and a 3D triangle $P_1P_2P_3$. Since this is likely to be done many times, and is the most expensive operation in the algorithm, it is worth examining in depth to create efficient code for the function.

It can be observed that there are several possible cases:

- the point could be closest to a vertex of the triangle (P_1, P_2 or P_3)
- the point could be closest to an edge of the triangle (P_1P_2, P_2P_3 or P_3P_1)
- the point could be closest to an interior point of the triangle. ($P_1P_2P_3$)

It would be costly to calculate the distance to each possible case and then use that to determine the minimum distance. It is far better to determine which case is applicable, and

then calculate the distance only once for each triangle. Two methods were implemented and compared. The first calculates the distance in three dimensions, and the second is a method that reduces the problem to two dimensions. It will be shown that the two dimensional method is the more efficient of the two.

5.5.3 3D method

Approaching the problem in three dimensions requires the projection of P_0 onto the plane of triangle $P_1P_2P_3$ to create P_0' (Figure 5.4).

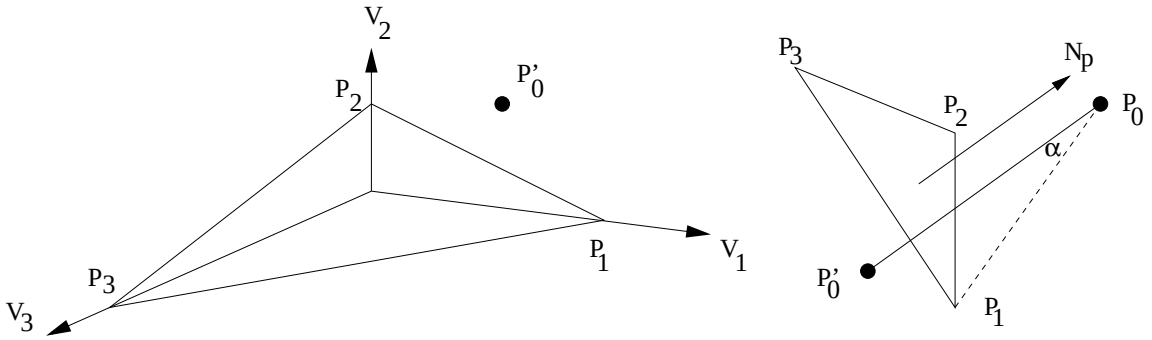


Figure 5.4: Calculating the distance of P_0 from $P_1P_2P_3$.

The normal N_p of $P_1P_2P_3$ can be calculated as:

$$N_p = P_1P_2 \times P_1P_3 \tag{5.5}$$

The angle, α between the normal N_p and P_1P_0 is calculated:

$$cos\alpha = \frac{P_1P_0 \cdot N_p}{|P_1P_0||N_p|} \tag{5.6}$$

The length of the vector P_0P_0' can be found using:

$$|P_0P_0'| = |P_0P_1|cos\alpha \tag{5.7}$$

The vector P_0P_0' can then be determined:

$$P_0P_0' = -|P_0P_0'|\frac{Np}{|Np|} \tag{5.8}$$

(The negative sign is used since P_0P_0' has direction opposite to that of N_p.)

P_0' can then be calculated as:

$$P_0' = P_0 + P_0P_0' \tag{5.9}$$

If it was the case that the projection of P_0 onto the plane lay within the triangle $P_1P_2P_3$, the length $|P_0P_0'|$ (Equation 5.7) would be the distance of P_0 from $P_1P_2P_3$.

If instead P_0' falls outside the triangle, the distance to the triangle is the distance to the closest edge or vertex to P_0'. In order to determine which edge or vertex P_0' is closest to, the position of P_0' in relation to the three vectors V_1, V_2 and V_3 must be found, (Figure 5.4) where:

$$V_1 = \frac{P_2P_1}{|P_2P_1|} + \frac{P_3P_1}{|P_3P_1|}, \quad V_2 = \frac{P_3P_2}{|P_3P_2|} + \frac{P_1P_2}{|P_1P_2|}, \quad V_3 = \frac{P_1P_3}{|P_1P_3|} + \frac{P_2P_3}{|P_2P_3|} \tag{5.10}$$

If $f_1 = (V_1 \times P_1P_0') \cdot N_p$, then $f_1 > 0$ if P_0' is anticlockwise of V_1. Similarly, f_2 and f_3 can be calculated for the other vectors. Using f_1, f_2 and f_3 the position of P_0' in relation to the vectors V_1, V_2 and V_3 can be determined. It further has to be determined if P_0' is inside the triangle, and if so the distance from the triangle is the distance calculated in Equation 5.7.

If P_0' is clockwise of V_2 and anticlockwise of V_1, it is outside the triangle if:

$$(P_0'P_1 \times P_0'P_2) \cdot N_p < 0 \tag{5.11}$$

and similarly for the other cases.

If P_0' is found to be outside the triangle, it is either closest to a vertex, or the side. For example, assume P_0' is closest to P_1P_2. (It is easy to apply the following to the remaining edges.)

The direction R, of P_0' to P_0'' is given by:

$$R = (P_0'P_2 \times P_0'P_1) \times P_1P_2 \tag{5.12}$$

where P_0'' is the projection of P_0' onto the triangle edge, and the angle γ is:

$$cos\gamma = \frac{P_0'P_1 \cdot R}{|P_0'P_1||R|} \tag{5.13}$$

The length $P_0'P_0''$ is calculated using:

$$|P_0'P_0''| = |P_0'P_1|cos\gamma \tag{5.14}$$

and $P_0'P_0''$ is:

$$P_0'P_0'' = |P_0'P_0''|\frac{R}{|R|} \tag{5.15}$$

The point P_0'', which is the projection of P_0' onto the line P_1P_2 is:

$$P_0'' = P_0' + P_0'P_0'' \tag{5.16}$$

Let

$$t = \frac{P_0'' - P_1}{P_2 - P_1} \tag{5.17}$$

If $0 \leq t \leq 1$, P_0'' is between P_1 and P_2 and the distance of P_0' from the line is $|P_0'P_0''|$ as calculated in Equation 5.14. The distance of P_0 to P_1P_2 is $\sqrt{(|P_0'P_0''|^2 + |P_0P_0'|^2)}$.

If $t < 0$, P_0 is closest to vertex P_1 and can be calculated as the distance $|P_1 - P_0|$.

If $t > 1$, P_0 is closest to P_2.

Therefore the distance of P_0 to $P_1P_2P_3$ has been calculated. It should be noted that Equations 5.5 and 5.10 can be precalculated for efficiency.

5.5.4 2D method

The second approach is that of converting the problem into a two dimensional problem. The simplest way to achieve this is by calculating the translation and rotation matrix to rotate the triangle $P_1P_2P_3$ so that P_1 lies on the origin, P_2 lies on the z axis, and P_3 lies in the yz plane.

This transformation matrix can be calculated once for each triangle in a preprocessing step, and can then be used to transform P_0 to P_0'. P_0 can be trivially projected onto the triangle's plane giving P_0' by ignoring its x coordinate since the triangle is in the yz plane. If P_0' is inside the triangle $P_1P_2P_3$, the distance from the point to the triangle is simply the x coordinate of P_0.

Using P_0', determination of the closest part of triangle $P_1P_2P_3$ to P_0 can be found by using the edge equation [98], and once determined the distance can be found in a standard way. The edge equation is simply:

$$E(x, y) = (x - X)\, dY - (y - Y)\, dX \tag{5.18}$$

for a line passing through (X, Y) with gradient $\frac{dY}{dX}$ with respect to a point (x, y). If $E < 0$ the point is to the left of the line, if $E > 0$ to the right, and if $E = 0$, it is on the line. Using Figure 5.5 we see the different possibilities.

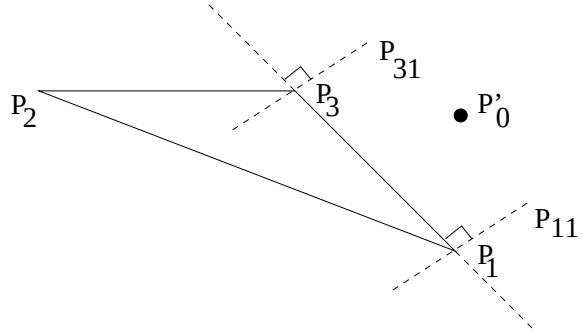


Figure 5.5: Calculating point position relative to triangle.

If P_0' is left of P_3P_1 it is closest to P_3P_1 if it is to the right of P_3P_{31} and to the left of P_1P_{11}. The proximity to the other edges of the triangle can be similarly determined. For P_0' to be closest to vertex P_1 it must be right of P_1P_{11} and left of P_1P_{12}, where P_1P_{12} is defined at right angles to P_1P_2. Using just these edge equations, the closest vertex or edge of $P_1P_2P_3$ to P_0' can be determined. The lines P_1P_{11}, P_1P_{12}, P_3P_{31}, P_3P_{32}, etc. and their directions can be precomputed, thus enabling simple applications of the edge equation to determine which part of the triangle the point P_0' is closest to. Once determined, the distance can be calculated to that part in the normal way.

5.5.5 Reduction of the number of triangles

As mentioned in the previous section the distance must be calculated to each triangle in order to determine the minimum distance to the surface. It is obvious that some triangles are far away from the voxel in question and should not have their distance calculated from the voxel. The complexity of the algorithm can be reduced further by determining these triangles quickly and eliminating them from the process of calculating the minimum distance from a voxel to a surface.

Using the methods of the previous sections results in a shell of known voxel values surrounding the surface which represents the original triangular mesh. From Section 5.5.1 we know that the furthest any calculated voxel can be from the surface is the distance covered by three diagonal voxels – or $2\sqrt{3}$ units.

A simple method of removing triangles from the process is that of rejecting immediately any triangle whose plane is greater than $2\sqrt{3}$ units away from the current voxel. Using the rotation method of the previous section the distance to the plane is known as soon as the appropriate transformation has been applied to find the transformed x coordinate of the voxel. Since this can be calculated immediately for each triangle, a minimum of calculation is carried out upon the triangle.

This does not remove triangles completely from the process, but does remove them from the *closest part* and distance calculations. It is shown by the results (Section 5.6) that the time to produce volume data is reduced by about 50% using this method.

5.6 Voxelisation Results and Conclusions

The voxelisation process was implemented in C on a DEC 3000/400. The three acceleration steps of Section 5.5 were included in the implementation. Several meshes were voxelised – octahedron, dodecahedron, soccerball, teapot, and chess pieces. In each case the triangular mesh was converted into a voxel data set of size 60^3 (216,000 voxels), for which the timing is given in Table 5.1. The octahedron and soccerball (Figure 5.6) show that the voxelisation process is effective for data which contains flat faces and sharp edges which we wish to be retained. The queen chess piece (Figure 5.7(a)) demonstrates that fine surface details are not lost by the voxelisation process. The smooth curvature of the piece is retained, and small details such as the ridge halfway up, and the knob on the top are still visible. Although

Data set	No. of triangles	Distance computations	% of voxels calculated	Time taken (seconds)
Octahedron	8	99576	5.12%	0.817
Dodecahedron	36	1170754	14.65%	4.316
Soccerball	116	3850470	15.24%	12.699
Teapot	252	5717484	10.98%	20.166
King	3080	17740398	2.67%	93.563
Queen	2600	16378497	2.92%	85.297
Bishop	2360	14092809	2.76%	76.314
Pawn	1600	14791639	4.28%	73.597
Knight	1524	10191575	3.09%	53.548
Rook	1600	18501156	5.35%	94.980

Table 5.1: Table showing voxelization timings

voxelised at 60^3 the piece only occupies an area of 15 voxels squared at the base, and about 3 voxels squared at the apex. Figure 5.7(b) shows pawn data combined with CT head data and demonstrates that one single visualisation technique can be used to display data which is originally from different sources. In fact the facets visible at the base of the pawn show that in this case the voxelisation is limited by the original triangular mesh.

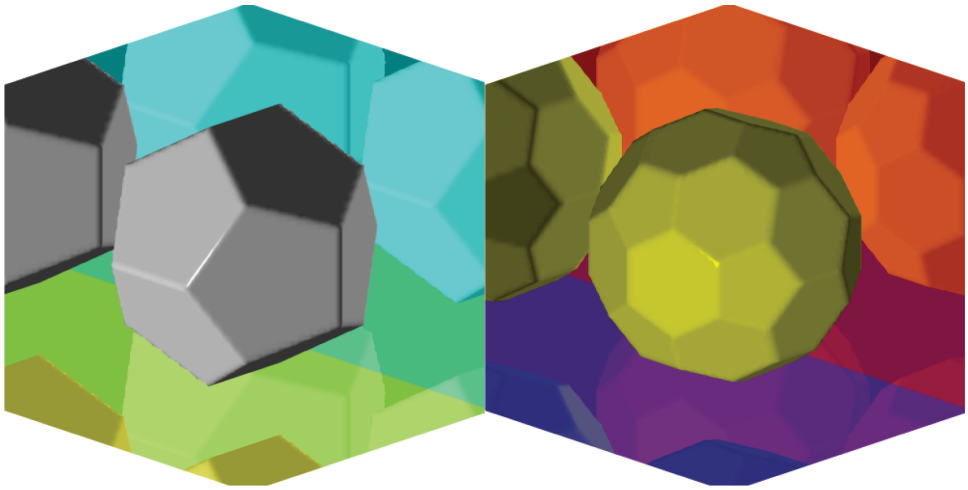


Figure 5.6: (a) Voxelised dodecahedron. (b) Voxelised soccerball.

The mesh data set and the number of triangles in the data set is given in the first two columns of Table 5.1. The third column gives the number of times the distance was computed between a voxel and a triangle. The percentage of the number of voxels that enter the distance calculation to the number of total voxels is given in column 4, with the time taken to produce the voxel data, in seconds, given in the final column.

In order to show the results of the acceleration techniques over the naive method an additional table (Table 5.2) shows the results for the king data set which can be regarded as typical. The first line gives the timing for the brute force naive method. The second line gives the

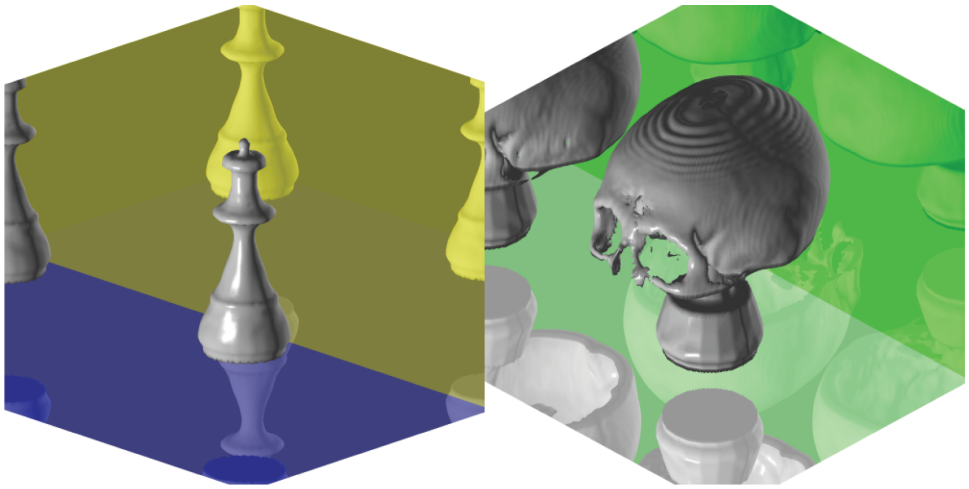


Figure 5.7: (a) Voxelised chess piece (b) Voxelised CThead and pawn.

timing for a method using the first two optimisations, with the final giving the timing for all three methods.

Algorithm	Time taken
Naive method	50 mins.
With 3.1 and 3.2	144.294 secs.
With all methods	93.563 secs.

Table 5.2: Table showing voxelization timings for different acceleration techniques

Future extensions to this work could involve extending visual effects by incorporating shadows, fog and global illumination complementing the already existing reflectance model.

In conclusion, a useful method for the voxelisation of triangular meshes has been presented, along with the visualisation process. This method works well and robustly for arbitrary closed meshes, and computes the data in a realistic time. In order to achieve this, three acceleration methods were described and implemented. The first was to reduce the number of voxels the distance is computed for, the second was to achieve an efficient implementation of the point to triangle distance function, and the third was to reduce the number of triangles the distance calculation is performed upon. The results of the method have been visualised using a general volume visualisation technique, and inspection shows the images to be accurate with respect to the original model.

5.7 Conclusion

This chapter has presented two methods for the construction of artificial volume data. The first method presented (Sections 5.2 to 5.3) showed how natural objects could be represented using *blobby* or *soft* objects. The direct surface rendering algorithm of Chapter 4 was applied to the volume data representation of these particular objects, and was shown to be an effective

method for the visualisation of the objects, both in terms of accuracy and efficiency. The two examples used to demonstrate this were the coalescing of two water droplets, and the explosion of some particles modeled by a particle system.

The second method of construction was that of voxelisation (Sections 5.4 to 5.6) which creates volume data from some source representation type. In this chapter the voxelisation of triangular meshes was investigated, and a method was presented to achieve this. In addition to this, several acceleration methods were described and compared which together make the voxelisation of large triangular meshes realistic. The acceleration methods included the reduction of the number of voxels for which a value has to be computed for, the reduction of the number of triangles which enter into the distance calculation, and the implementation of an efficient three dimensional point to triangle distance calculation function. The volume data was visualised using the direct surface rendering method, and several examples of such visualisations were evaluated.

This voxelisation work [2] was presented at the 13th Annual Eurographics Conference (UK Chapter), and appeared in the conference proceedings.

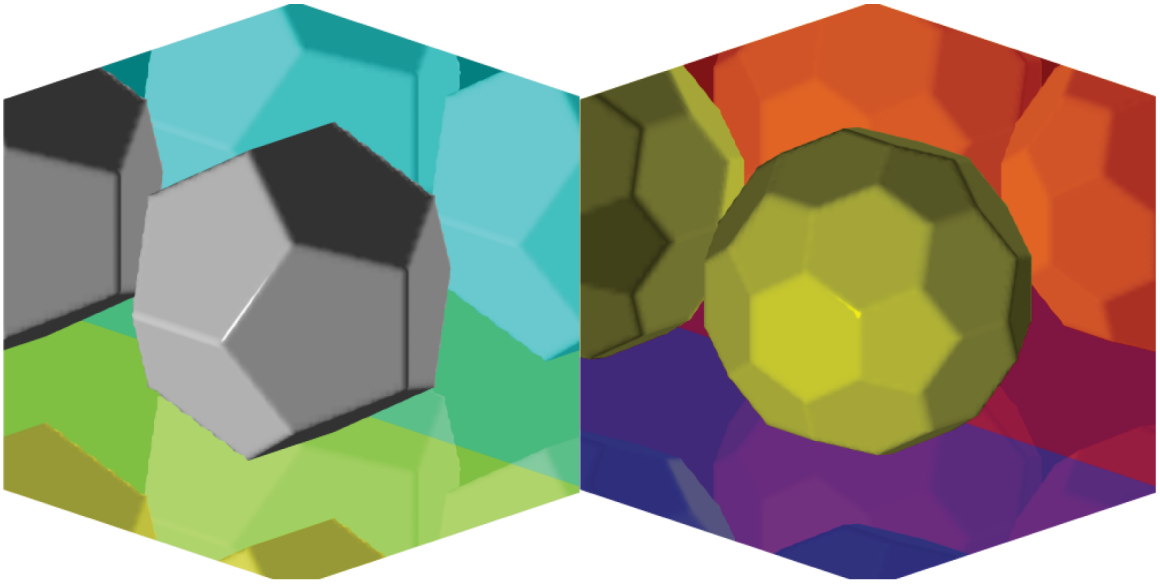


Figure 5.6: (a) Voxelised dodecahedron. (b) Voxelised soccerball.

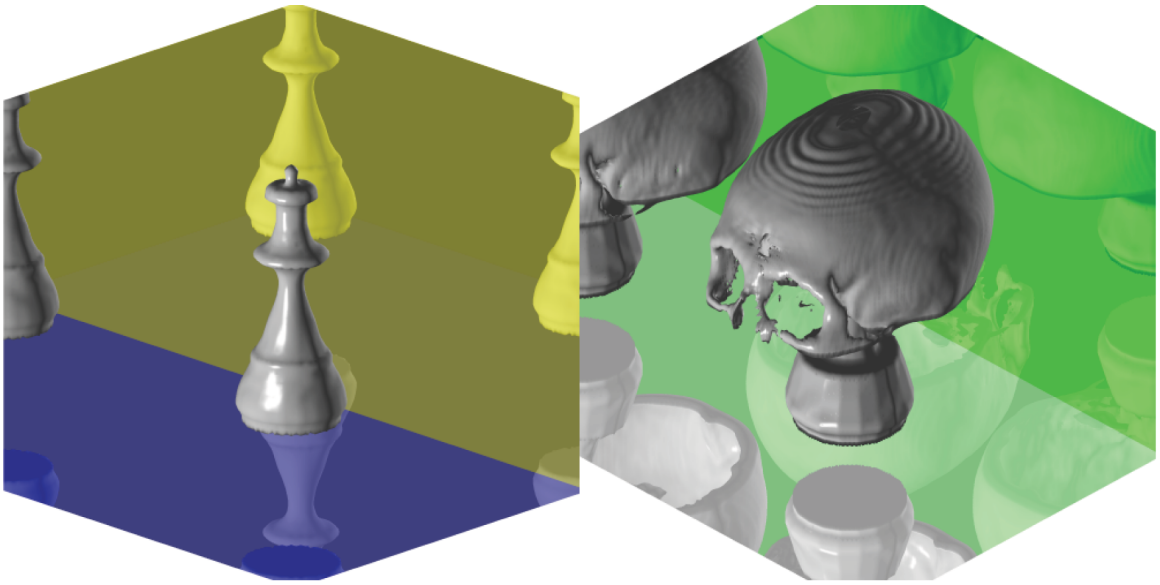


Figure 5.7: (a) Voxelised chess piece (b) Voxelised CThead and pawn.

Chapter 6

Contour Methods

6.1 Introduction

In a variety of applications the data is available as a series of contours which represent the cross section of a surface when sliced by the planes the contours reside in. The surface construction algorithm determines the surface using the contours. This chapter examines the existing algorithms and presents a new algorithm that has none of the short comings of the existing algorithms. In Section 6.2 the simple triangulation methods are reviewed. These methods only cope with cases where only one contour exists on each slice. The more general case of many contours on each slice requires more complex algorithms which are the subject of the review of Section 6.3. Section 6.4 shows how shapes (cones) can be fitted to the contours to approximate them and aid the reconstruction process. Section 6.5 introduces the more general problem of reconstruction from unorganised points, and finally remarks on these sections are given in Section 6.6, along with the identification of major problems. A new algorithm for surface reconstruction is presented in Section 6.7 that copes with the cases that create difficulties for other algorithms. The context of the problem is given in Section 6.8 and the approach is described in Sections 6.9 and 6.10. The algorithm is tested on classical and real problems in Section 6.11, and the results are discussed in Section 6.12. Finally the chapter is concluded in Section 6.13.

6.2 Simple Triangulation

The original solution to the construction of surfaces from contour data is due to Keppel [99]. For this method the form the data takes is constrained to be an anti-clockwise sequence of points on the contour. Each contour is a cross section of constant z with the points having positions which are x, y values. Given this definition of the contour, the convexity and concavity of any three successive points can be determined easily by drawing a circle with an arc passing through the points. If the arc has positive curvature it is anti-clockwise and convex, or if it has negative curvature it is clockwise and concave. Using this method the concave and convex regions of the contour can be determined, and treated separately during the tiling process.

Keppel points out that the tiling process must be guided somehow since it is impossible to ennumerate every triangulation of successive contours. For two contours of 12 points each there are around 10^7 triangulation combinations. In order to solve this problem, the contours are represented by a graph (Figure 6.1), where each vertex in the graph represents a possible span, and each edge is a triangular tile. Any tiling of the two contours is a path through the graph, and the tiling chosen is one such that the path is a minimal cost path measured with respect to some cost metric. A cost is associated with each edge in the graph which when totalled over the path is an indication of how good the surface is with respect to the cost metric.

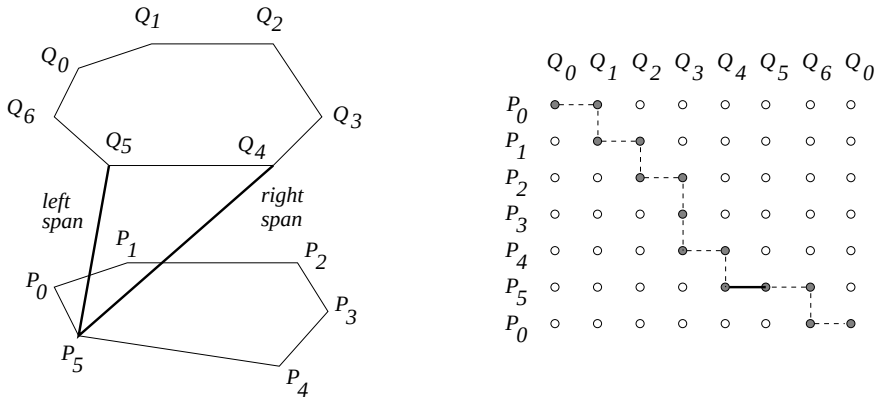


Figure 6.1: (a) Two adjacent contours. (b) Graph representing contours.

The cost metric Keppel chooses to produce an optimal surface is known as *maximise volume*. The idea is that each edge in the graph will represent the volume that triangle will add to the surface by treating it as a face of a tetrahedron. The problem of concavities is solved by trying to minimise the volume of the surface in those regions.

This method was later used for a practical application by Fuchs et al. [100], where the graph problem was studied and solved using a divide and conquer algorithm rather than the heuristics of Keppel. They show that the surface can be represented by paths in the graph, and that they can be found by successively subdividing the graph in order to minimise the search space. The measure that they try to minimise is the triangular facet area with additional measures to prevent the twisting of the surface.

Sloan and Painter [101] review the graph solving problem and compare common techniques. They also give a detailed description of a new technique in which they select pessimistic paths in the graph search which they show leads to a quicker solution, as opposed to selecting paths that minimise the search space. They explain that this is because that subdividing unequally results in subgraphs that will take approximately the same time to search, since the larger can take advantage of *bottlenecks* to remove nodes from the search, leaving less nodes for the more complex search for the optimal solution in the smaller subgraph.

The method of Fuchs et al. was also used by Cook et al. [102] with application to measuring volumes of surfaces represented by contour data. Specifically, they were interested in measuring the volume of human organs such as the liver.

Ganapathy and Dennehy [103] use heuristics[1] to triangulate between contours. They use a method of normalising the contour perimeter to 1. The next tile is chosen so that the difference in value of the distance traversed along the top contour with that of the lower contour is minimised. This measure has the advantages that it can be calculated cumulatively using local information and that previous contour spans contribute to the current calculation. It also has the advantage of requiring only $M + N$ steps to calculate the tiling. This is a result of choosing the tiling according to local information, and not minimising some measure of the complete tiling.

Heuristics are also used by Cook et al. [104]. They map the contours onto star-convex regions by assuming contours change direction slowly and smoothly, and have no sharp projections. The centroid of the upper region is determined, and the star-convex region for the lower contour is defined such that every line connecting each point of the lower contour to the centroid of the upper region passes through the region. A tiling between the two mapped contours is determined using heuristics which are based on the dot product of the vectors joining the centroids and the points to be tiled. Once the tiling process is complete, the mapped contours are returned to their original configuration retaining the tiling connections. The mesh produced is then used for the display and measurement of the area and volume of the reconstructed object.

6.3 Correspondence and Branching Problems

The main problem associated with this basic tiling method is the fact that the algorithm is restricted to the generation of surfaces from contour data which have just one contour for each value of z. Any object which is branching or coalescing in nature cannot be tiled. More specifically the problems encountered are referred to as the correspondence and branching problems, and are stated as:

- *Correspondence problem* – given two slices with arbitrary numbers of contours within them, which contours from one slice correspond to which contours on the other slice? Without establishing the correspondence between the contours, the tiling algorithm cannot be used.
- *Branching problem* – given two slices where M contours correspond to N contours $(M, N > 0)$, how should the tiling process be carried out?

Shantz [105] dealt with the many to many branching problem by connecting contours within the slice plane using as few as possible minimum distance connections. This has the effect of creating a single contour on each slice, which represents all the contours on that slice. The tiling then takes place using the method of Fuchs et al. [100] which can handle the case of mapping just one contour to one. The problem with this method is that it can only handle contours that are very similar in shape in adjacent slices. It can only handle multiple contours on a slice if they approach each other at one point rather than having a complex interface (Figure 6.2).

[1] Optimal methods solve the problem using graph methods to find a near optimal solution with respect to some measure. Heuristic methods solve the problem using local information only as a basis for choosing the tiling.

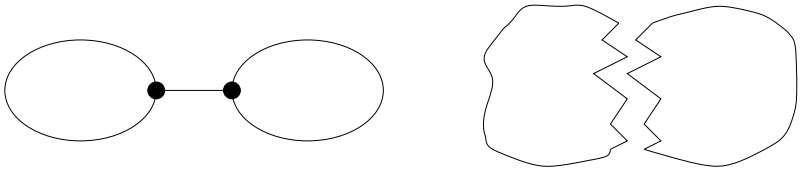


Figure 6.2: Approach at (a) A single point (b) A complex interface.

Another solution to these problems was proposed by Christiansen and Sederberg [106] in which they determine corresponding contours as those that overlap in any way. Once a correspondence between contours has been determined, contours in successive slices are bounded by boxes and mapped onto a unit square. The reasoning behind this is that successful contour triangulation is more likely when successive contours are mutually centred and similar in size and shape. Triangulation takes place on these contours using the very simple method of selecting the next triangle with the *shortest diagonal*, thus removing the need to resort to costly graph searches for the reasons mentioned above in the method proposed by Ganapathy and Dennehy. Branching is coped with by introducing a new vertex between the closest point on the two branches and combining them along with this point to make one contour. Tiling takes place on these two normalised contours and is then mapped back to the original vertex positions. This method copes well with simple branching but has difficulty when branching contours have a complex interface, which is more often than not the case. In these cases it is suggested the user should manually triangulate the path between the two branches.

Boissonnat [107] creates a surface from contours by projecting the Voronoi diagram of two neighbouring contours onto a plane parallel to the planes of the contours. The resulting 2D graph is used to create the 3D Delaunay triangulation between the two slices. He proves that this is possible for such a restriction on the points (the points occur in two 2D planes and are connected in each plane in the form of contours) and that the method for doing so is optimal. The resulting triangulation is a solid, of which the external faces make up the surface of reconstruction.

This method is very successful in the difficult situations where there are multiple contours with differing numbers upon successive slices. The method fails when adjacent slices differ too much, in addition to which the circumstances for failure cannot be identified. In such cases more slices must be taken through the object. The images presented in the paper show the method working for lungs and a heart, although they are not of a high fidelity.

Ekoule et al. [108] propose an algorithm which handles arbitrarily shaped contours, with multiple contours on each slice plane allowed. For their contour triangulation method they use a simple minimum edge length heuristic which is similar to Christiansen and Sederberg's [106] shortest diagonal heuristic. They divide the contour into convex and concave regions in a similar way to Keppel [99] and create a representative tree structure. Each contour is mapped onto its convex hull with vertices in the concavities mapped using Euclidean distance to retain the relative distances between successive points. The tiling then takes place between the two convex contours trivially, producing a satisfactory triangulation. This triangulation is then retained as the vertices are mapped back to their original positions, thus providing a triangulation of the original complex contours.

The case of multiple branching (one contour to many) is handled by creating a new intermediate contour between the two adjacent slices to be tiled. Each slice is then tiled with this contour to produce the overall tiling. The intermediate contour is produced by creating one closed polygon which is composed of points of the many contours clipped against a polygon joining the centroids of the contours. This polygon is tiled with the single contour and vertices are moved halfway along the spans to create the intermediate contour. The single contour is trivially tiled to this, while each individual contour on the slice with many contours is tiled with its corresponding contour part in the intermediate contour. The centre part of the intermediate contour is then tiled horizontally using existing vertices and the process is complete.

In order to cope with many to many branching they determine which contours should link with which using a *superposition degree* estimate. Those that should link are then tiled together.

This method, whilst dealing with moderately complex cases, will have difficulty when adjacent contours do not have similar convex hulls. In addition to this the tiling is not successful in some cases of many to many branching. The problem occurs when the links do not form disjoint sets. For example (Figure 6.3(a)) some of the links are $a \rightarrow c + d, b \rightarrow d + e$ and $d \rightarrow a + b$. In this case the available contours are insufficient for their tiling method, they require the contours of Figure 6.3(b), which have links $c \rightarrow a + b$ and $c \rightarrow d + e + f$.

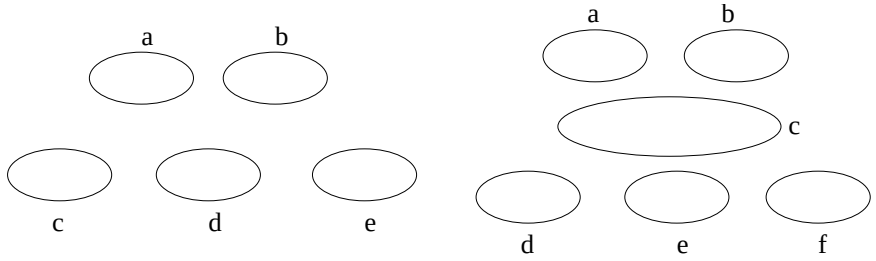


Figure 6.3: (a) Insufficient data (b) Sufficient data.

Giersten et al. [109] use user interaction to determine the connectivity between contours of adjacent slices. During the contouring process the previous slice is overlaid on the current slice in semi-transparent form and the user specifies successive contours. The contour slices are automatically aligned by using the centroid information provided by specifying corresponding contours. The slices are rotated and translated to achieve the correct alignment and overlap, and scaled to compensate for objects which may have been compressed as the result of the slicing process. They then proceed to reconstruct the data on an object by object basis. For each object they have a subgraph which indicates the corresponding contours on each slice, and their connectivity. Where a contour branches, or contours coalesce, they use a simple contour branching scheme based on that of Christiansen and Sederberg [106], and as such can only handle simple one to many cases. In addition to reconstruction they also apply 3D filtering techniques in order to produce smooth object surfaces.

6.4 Cones

Soroka [110] investigates the use of Generalised Cones (GCs) when applied to the reconstruction of objects from contour data. The three components of the GC are the *spine*, *cross section* and a *sweeping rule*. Soroka uses elliptical cones (ECs) which are described using elliptical cross sections, and by allowing the minor and major axis of the ellipse to vary linearly and independently over time. Each contour is determined to be either elliptical or complex. Complex contours are discarded and play no other part in surface reconstruction. On finding an elliptical contour the algorithm tries three rules in order, firing the first rule that applies. First it tries to extend an existing EC by adding the elliptical region to the end. Secondly it tries to create a new cone from the newly found single elliptical region. Thirdly it splits off a subregion of the cone in order to grow the cone into a complex region. Essentially this process attempts to model the general topological and geometrical information provided in the data, but suffers from a few problems.

The method can only model simple convex objects. Any contours or parts of the object regarded as complex are omitted from the final reconstruction. The practical application that Soroka presents is that of the reconstruction of a canine heart. The resulting model has the right ventricle missing because the contours comprising that object are classed as complex in that region. Another problem is that fact that the ECs cover the object, and not partition it. This is due to the fact that regions may be shared by several cones, and the result is that it would be impossible to use this geometric description to perform any quantitative analysis about the volume of the reconstructed object.

Myers et al. [111] present progress on the correspondence and branching problems. They use a descriptive language to indicate the contours present on each slice and the connection to contours on adjacent slices. This connectivity information is the result of an analysis of contour correspondence which is performed either manually or automatically. Two methods are suggested for automatically determining correspondence, namely the minimum cost spanning tree (MST) method and an algorithm using elliptical cylinders (ECs).

The method they use for ECs differs from that of Soroka [110] in that complex contours are not ignored. Rather they are approximated by a subcontour that is elliptical, and which is then treated in the same way as the other ellipses, either to extend a cylinder, or to create a new singleton cylinder. The goal of this process is to produce a small number of cylinders containing as many ECs as possible.

The next stage is to link cylinders together to create objects. They establish that cylinders can be linked in three ways. The cylinders A and B could link if one end of cylinder A joins to one end of cylinder B. The cylinder A could link to B and C if one end of cylinder A joins to the ends of cylinders B and C. Lastly the cylinder A could link to B if the end of cylinder A joins to an interior contour of cylinder B. In all cases the cylinder can be added to an existing object. If a cylinder does not connect with any existing object it is used to create a new object.

Their method also differs from Soroka's in that they use each contour only once when creating a cylinder. This results in the cylinders being a disjoint representation of the surface rather than a cover of the surface as was the case with Soroka's method. This restriction, while

avoiding the problems of Soroka's method, creates problems of its own, in that the result is order dependent. The order in which the ellipses are considered has a great bearing on the cylinders produced since a valid cylinder can be created by an ellipse being added to any singleton cylinder, when perhaps it would be better used elsewhere. They suggest examining all cylinders that could be created from three sections and keeping only those that extend to a length of three.

They also suggest a method using an MST. Firstly an ellipse is fitted to every contour. A graph is created using the contours as nodes, and a four dimensional value (x, y, A, B) is associated with each node, where (x, y) is the position of the contour, and A and B are the major and minor axes. All edges (i, j) are added where contours i and j lie on adjacent sections, and as such could possibly correspond. The cost for each edge (i, j) is calculated as the Euclidean distance between the nodes. The result is that the more similar contours are in position and orientation, the closer to zero the cost is. A MST is then computed, and if edge (i, j) exists in the spanning tree, contours c(i) and c(j) are expected to correspond. This tree contains interobject connections which are removed by traversing paths through the tree from unvisited nodes. After this stage the contour connectivity and object information is available in the form of that produced by the EC growing method. The problem with the MST method is that it does not find all the connectivity information because a tree rather than a graph is used. Also because connections are rejected on some criterion, rather than created, the tree may be left with edges that result in incorrect joins of disjoint objects.

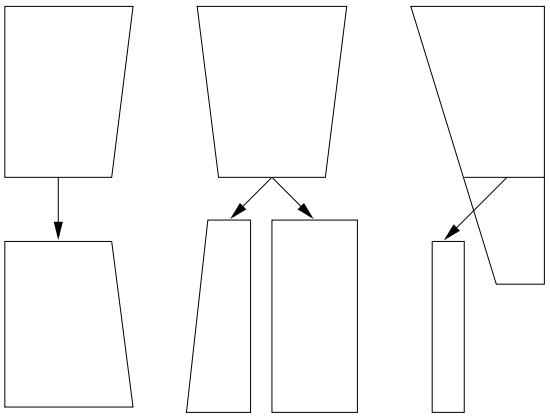


Figure 6.4: Branching – (a) One to one (b) Two to one (c) Internal.

From this model the branching structure of the object can be classified. In their method they only allow bifurcations. The three types of branching are – two cylinders are connected (Figure 6.4(a)), one cylinder branches (Figure 6.4(b)) and one cylinder interconnects (Figure 6.4(c)).

Where branching is involved they divide the possible cases into two – those that involve contours which have a simple interface (Figure 6.2(a)), and those that have a complex interface (canyon) (Figure 6.2(b)). The simple case can be tiled using the method of Christiansen and Sederberg [106]. The more complex case is solved by triangulating the polygon representing the canyon. The two contours are then connected and tiled using the one to one method. The canyon is found by finding the convex hull of the combined contours. This results in a list of points where the two edges from a point on one contour to a point on the second contour are

the edges that close the canyon polygon.

The techniques they describe are quite elegant (MST) and seem to work very well for many cases that other methods fail on. The main problem with this method is that although they indicate it can handle cases where many contours correspond to and branch into many, they give no examples. The canyon tiling for such arbitrary cases would become complex, assuming the correct connectivity could be found. Their method also suffers from the problem that it cannot handle major shape changes between slices, which is usual for these methods. Finally it cannot handle cases where one contour is entirely contained in another contour defining some hole within the object.

6.5 Reconstruction from Unorganised Points

Surface reconstruction from contour slices can be regarded as a particular case of the more general problem of reconstruction from disparate points. The general problem can be considered to be the reconstruction of the surface from a finite set of points in space. The restricted problem of surface reconstruction from contour slices is the reconstruction of a surface from a finite set of points where the points occur in a finite number of planes and are connected in some way.

The restricted problem arises when the data has been collected in a slice-by-slice manner, either automatically or manually. The general problem occurs when the data is collected by some object scanner or sensors (eg. cyberware scanners). This problem occurs during applications of reverse engineering – the automatic generation of CAD models from laser range data.

The problem has been approached by Boissonnat [112], using Delaunay triangulation. His first method works well for smooth objects with only minor variations in curvature. A point and its k-closest neighbours are projected onto a plane and the 2D Delaunay triangulation of the points is constructed. The triangulation is retained when the points are mapped back to their original configuration in order to give the surface triangulation. The second method, which works for a wider class of objects, creates the 3D Delaunay triangulation for the points. This triangulation consists of a tetrahedral description of the convex hull of the object. If the object is not equivalent to the convex hull, tetrahedra must be eliminated from the polyhedron until all the original sampled points are on the boundary. At this point the external faces of the tetrahedra is the triangular mesh description of the object.

Edelsbrunner and Mücke [113] also use 3D Delaunay triangulation to create the mesh, but have an additional control, α, which controls the level of detail. With $\alpha = \infty$ the mesh produced is identical to the convex hull of the original set of points. As α is decreased, the mesh decreases, until $\alpha = 0$ at which stage only the original points are present. A piece of the mesh disappears when α becomes small enough so that a sphere with radius α can occupy the space without enclosing any of the original points.

Turk and Levoy [114] produce meshes from $m \times n$ range images by considering each sample in the image to be a candidate vertex in the mesh. Four points in neighbouring rows and columns are selected and checked to see if they create triangles. If the distance between three

points is greater than some threshold no triangle is created since the surface may involve an unseen crevice in that area. The meshes from several scans from different angles are joined together in a process they call *zippering*. The final stage to the process is to optimise the mesh (see Section 2.3.4 of Chapter 2).

Most of these methods are computationally expensive – Turk and Levoy [114] state that the process of digitising an object and producing a mesh requires "*five minutes of user interaction and a few hours of compute time*". Boissonnat [107] applied the Delaunay triangulation method to the restricted problem, and took advantage of the restriction to reduce the computational expense. A future project could be to try to apply these methods to the restricted problem in an attempt to both accelerate these methods, and solve the restricted problem.

6.6 Remarks

From the above review of the various approaches it should be apparent that most strive to solve specific problems. The problem of tiling between two well behaved contours seems to be successfully managed, but by simply increasing the shape complexity of the contours, erroneous tilings start to occur. This problem can be overcome by mapping the contours to shapes that can be well tiled, for example Ganapathy and Dennehy's normalised length, Christiansen and Sederberg's mapping to unit square, and Ekoule et al.'s mapping to convex hull. The mapping of contours to some other shape is carried out so as to preserve the order and relative distances of vertices. The tiling process is carried out on the mapped contours, and the tiling is retained once the contours are returned to their original configurations. This allows a new class of contours to be handled, but the tiling process still requires contours to be similar in shape, and so if their mapped shapes are not similar, the method still does not produce a satisfactory tiling. A good test of any tiling technique is that of a spiral contour in one slice, and a convex contour in the next slice. This is regarded as being such a difficult case that J. Rossignac[2] of IBM research set a competition to produce a surface from such contours. None of the methods mentioned thus far can create a satisfactory surface from such contours, solely because they differ too greatly. Attempts have been made but as Boissonnat reports [107] the usual surfaces produced, intersect themselves.

In addition to the fundamental problem of tiling between two contours most methods have trouble tiling branching contours. The approaches can be divided into those that do not cope with branching, Sloan and Painter's, Fuch et al.'s, Ganapathy and Dennehy's, and Soroka's, those that cope with simple branching (one to many, only when contours approach at a point), Shantz's, Christiansen and Sedeberg's, and Geirsten's, and those that cope with more complex branching (some many to many cases and contours that approach at complex interfaces) Boissonnat's, Ekoule's, and Myers et al.'s. Even the advanced methods do not cope well with moderately complex branching cases, and often require many slices in areas of mild complexity. The problem is compounded if contours have complex shape or branch arbitrarily many to many, and it seems that the more complex the case to be handled is, the more complex the solution is. Boissonnat [107] points out that mostly the inadequate

[2]Personal communication

solutions are a result of the methods not being able to tile horizontally under the contours in order to form bridges.

All the methods reduce the problem to that of determining the surface between two adjacent contours and carry out the tiling using heuristic or optimal methods and in the case of Boissonnat [107] Delaunay triangulation.

In addition to all of these problems, contour methods also usually lack ability to create surfaces with a definite thickness, for example a hollow cylinder as would be produced from the contours of Figure 6.5.

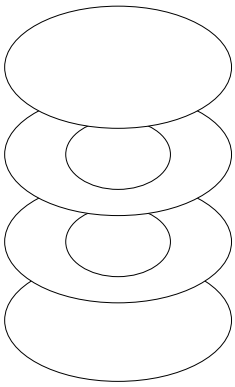


Figure 6.5: Contours with internal structure.

MacLeod et al. [115] were faced with the task of reconstructing the human thorax from over a hundred MRI slices. They describe the process as time consuming, since after digitisation of the contours, and alignment of the points, the resulting triangulated mesh must be edited by hand. User intervention is required in order to correct any inconsistencies of the mesh, involving a slice by slice check by hand of the whole surface that has been generated by the tiling process. To produce a surface from such data is tedious and can take of the order of one man day to do so.

The fact that so many problems exist for such a tiling method would suggest that an altogether new approach to the problem is required. The new approach should be able to cope with arbitrarily shaped contours with ease and produce *good* surfaces from the contour data quickly and with a minimum of user interaction. It should also be able to derive surfaces from many to many branching contours that is a smooth representation, and gives a good idea of what the original object was. Operation with a minimum of preprocessing to orientate the contours would also be a good requirement.

6.7 A New Approach to the Construction of Surfaces from Contour Data

In this section a new approach to the construction of surfaces from contour data is introduced. The process neither establishes a correspondence between vertices on one slice with those of the next as all other methods do, nor does it require preprocessing of the contours, or the contours to be similar in shape. The method is shown to be accurate, in that the surface produced, when intersected by planes corresponding to the original contour slices, produces

the original contours. It is consistent in that the surface is not dependent upon the order in which the slices are processed, or the ordering of the segments of any of the contours. It is efficient with regard to computation and ease of implementation. It is also flexible in that it handles arbitrary shaped contours that do not need to be convex, does not require the segments to be organised in such a way that they are placed end to end, joining the last vertex of the last segment to the first vertex of the first segment, and it does not require the segments to be organised such that a point is inside the contour if it is to the right, or outside otherwise. In fact it only requires that the contours be non-self-intersecting and closed, which is far less restrictive than the other methods.

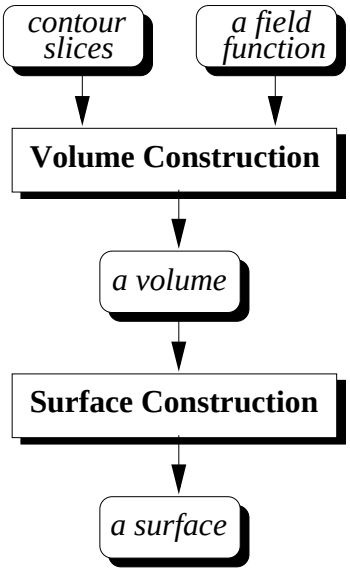


Figure 6.6: Two phases of the algorithm.

This new algorithm is organised as in Figure 6.6, which shows the input to the volume construction phase as a stack of contour slices and a field function. The result of this stage is a volume which is then used to create the surface in the surface construction phase. The background which lead to the development of this method is given in Section 6.8. Details about the input requirements are given in Section 6.9. Efficient calculation of the field function is the topic of Section 6.10 and results for some classical and practical problems are given in Section 6.11. Finally the approach is compared and summarised in the conclusions, Section 6.12.

6.8 Background

The impetus behind this research is the requirement for a good, robust method for producing surfaces from contour data. The data involved is that of a set of just 18 MRI slices through a human torso, from which models of the thorax, liver and lungs were required. It was also a requirement that the surface be as accurate and smooth a representation as possible. The MRI data did not contain any sharp delineation of objects, and as such was not suited to other methods of visualisation (Chapters 2 and 3). This *fuzziness* was due to the thickness

of the slices. In addition the contours varied greatly between slices due to the fact that there was such a large interval between scans (3cm). As a result of the fuzziness problem, it was best to identify the organs by eye, and outline them using digitised contours. Each slice was displayed in turn, and the contours of interest were outlined using mouse presses to indicate the positions of vertices on the boundary. The fact that the contours varied greatly between slices suggested that standard methods would not work well since this is the case they have most difficulty with. The need to visualise and perform mensuration upon surfaces from such contours led to the development of a new approach to construct surfaces from contour data.

6.9 Input Requirements

The input to the *volume construction* phase is a stack of contour slices S_1, S_2, ..., S_{N_z} and a field function $f(x, y)$. The function $f(x, y)$ is defined over all points of a plane in which contours of a slice reside such that

$$f(x,y) \begin{cases} < 0 & \text{if } (x,y) \text{ is outside all contours} \\ = 0 & \text{if } (x,y) \text{ is on a contour} \\ > 0 & \text{if } (x,y) \text{ is inside a contour} \end{cases} \tag{6.1}$$

In this phase, a two dimensional grid of size $N_X \times N_Y$ is uniformly mapped onto every contour slice. By defining a z-axis in the direction of the contour stack, each grid point can be considered as a voxel in an $N_X \times N_Y \times N_Z$ volume. The value associated with each voxel at location (x, y, z) is defined as $f(x, y)$ computed against all contours residing in plane z. The grid size determines the resolution of the surface to be constructed. The optimal accuracy can be obtained by choosing a fine grid or a grid with unequal intervals such that every contour vertex is located at a grid point. The original contours in each slice can easily be extracted from such a grid by taking all grid points where $f(x, y) = 0$, and linearly interpolating points that are not located at any grid point. This is valid since a contour is composed of a set of line segments, each of which is a linear interpolation between successive vertices of the contour. However, in practice, efficiency in computation and implementation can be achieved without sacrificing much accuracy by using a relatively coarse grid with equal intervals and an appropriate field function. The consideration of choosing a field function is to be investigated in the next section.

In the surface construction phase, the volume representation constructed from the contour slices is regarded as a continuous volume space, where the value of a non-voxel point is a trilinear interpolation of values of its eight neighbouring voxels that make up a cube containing the point. In comparison with the contour-based representation, this allows more general and consistent operations to be carried out. The process of extracting a surface from the volume is simply to identify all the points at which the trivariate function has a value of zero. The surface tiling algorithms of Chapter 2 may be used to construct such a surface in the form of a mesh of triangular facets which can then be rendered. The inbetweening of two successive contour slices becomes a trivial task of interpolating between two successive images in the volume.

6.10 Efficient Calculation of the Field Function

In the volume construction phase, a field function is computed with each voxel (i.e. grid point) in an $N_X \times N_Y \times N_Z$ volume. One simple field function is a decision function that determines if a point is interior, exterior to or on the contour boundary:

$$f(x,y) = \begin{cases} -1 & \text{if } (x,y,z) \text{ is outside all contours} \\ 0 & \text{if } (x,y,z) \text{ is on a contour} \\ 1 & \text{if } (x,y,z) \text{ is inside a contour} \end{cases} \tag{6.2}$$

The advantage of this function is that a polygon fill algorithm can be employed to scan grid points in each of the contour slices and determine their topological states in relation to contours. However, the surface produced is highly dependent upon the density of the chosen grid, and moreover, the finer the grid, the more discontinuous the surface in the z direction. This problem is illustrated in Figure 6.7, which shows two adjacent contour slices (Figure 6.7(a)) and the cross section of a surface constructed with each of three different grids (Figure 6.7(b)). Figure 6.8(a) shows a blocky surface resulting from the use of the function. The *blockiness* exhibited by the surface in Figure 6.8(a) is a result of the fact that essentially a binary decision has been made at each voxel – either it is inside the surface or not. This leads to a restricted number of possible surface configurations and as such surface normals, leaving the surface with large areas of flatness and sharp edges instead of a smoother continuous surface.

A far better field function is a distance function defined as

$$f(x,y) = \begin{cases} -dist(x,y) & \text{if } (x,y) \text{ is outside all contours} \\ 0 & \text{if } (x,y) \text{ is on a contour} \\ dist(x,y) & \text{if } (x,y) \text{ is inside a contour} \end{cases} \tag{6.3}$$

where $dist(x,y)$ is the distance from (x,y) to the closest point on the contour, or contours if there are more than one on the slice. This results in a volume that represents a continuous function sampled at discrete grid points and allows a surface to be constructed based on linear interpolation. As shown in Figure 6.7(c), the shape of the surface is not affected by the fineness of the grid in the same way as the simple field function. In contrast to Figure 6.8(a), Figure 6.8(b) shows a much more satisfactory surface produced using the distance field function. It is a much more natural surface than that of the one in Figure 6.8(a), and more like the surface one would have expected to have produced the contours.

The method described above is simple to implement, but implemented naively would result in an algorithm of $O(\sum_{i=1}^{N_Z} N_X \cdot N_Y \cdot N_{C_i})$, where N_Z is the number of slices, N_X and N_Y are the dimensions of the grid and N_{C_i} is the number contour segments in each slice S_i.

The calculation of the distance field would be computationally expensive, as for every grid-point, the distances to every contour segment in the same slice have to be calculated in order to find the minimum. Typically, for a volume data set with $200 \times 200 \times 25$ grid points and an average of 100 contour segments in each slice, 100 million distance calculations need to be performed. Since each distance calculation involves a computationally expensive square

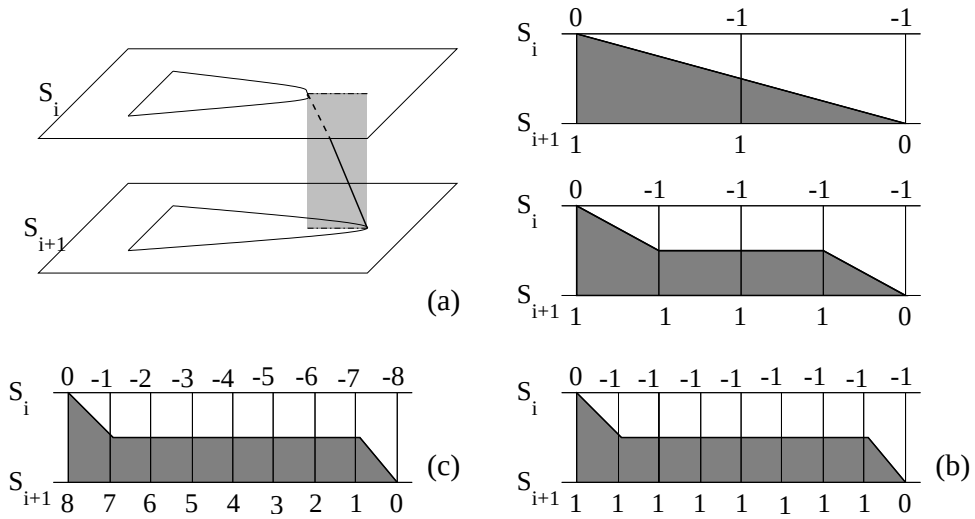


Figure 6.7: (a) Two adjacent contour slices. (b) Results of the simple field function with three different grids. (c) Result of the distance field function.

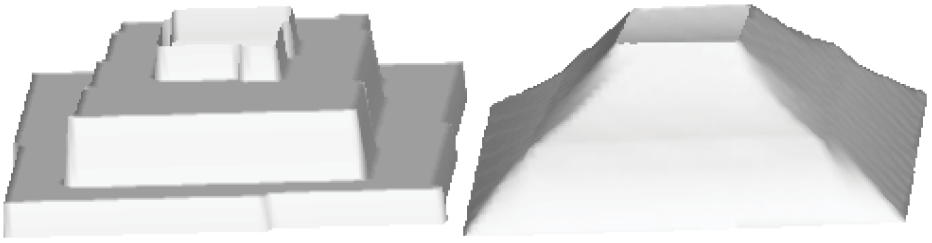


Figure 6.8: (a) Result of using simple field function. (b) Result of using the distance field function.

root this is obviously prohibitive and the prerequisite that the method should be quick to compute is not fulfilled.

The complexity of the algorithm can be reduced by partitioning each slice S_i into N_{C_i} sectors, each of which contains a contour segment C_i and those grid points closer to C_i than any other segments. The distance field value of each grid point can then be calculated using the field function against only one contour segment. Figure 6.9 shows an example of partitioning a plane consisting of two simple contours. The partitioning process is almost identical to that for constructing a Voronoi diagram except that bisectors between pairs of contour segments as well as vertices are calculated. The optimal solution to the Voronoi diagram is known as $O(N \cdot logN)$ [116]. Therefore the complexity of the partitioning stage is $O(\sum_{i=1}^{N_Z} N_{C_i} \cdot logN_{C_i})$ and the volume construction phase has a complexity of $O(N_X \times N_Y \times N_Z)$.

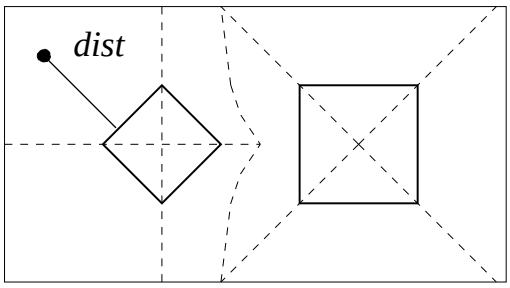


Figure 6.9: Partitioning a contour slice.

Although the number of segments to be considered for each grid point is reduced to one, the distance function still has to be computed for every grid point in each slice. As the implementation of a partitioning algorithm is far from being trivial, in most applications the actual saving on the cost of implementation and computation is usually very little if there is any.

By simple observation it can be seen that it is not necessary for the value of every grid point to be accurately computed. In the surface construction phase, the algorithm examines each cube, bounded by eight grid points. When it encounters a cube which contains the object interface it performs surface construction by linearly interpolating these grid points to find the polygon vertices of the surface, and also the surface normal at these vertices is calculated.

Voxel values at the eight vertices of the cube need to be known in order to determine the surface. The 6-neighbours[3] (two in each of the x, y and z directions) of each voxel also need to be known in order to calculate the surface normal. In other words, only the values of voxels near the actual surface are required in order to display the surface. At all other voxels an indication of whether the point is inside or outside the surface is sufficient. This leads to a practical method for reducing the computational cost by restricting the distance computation to those voxels near the surface interface.

Each voxel is associated with two fields, namely state and distance. The state field indicates whether a voxel is interior (1), exterior (-1) or on the surface mesh (0), and is first computed by scan-converting the object. The state function (Equation 6.4) can be calculated for each voxel (x, y, z).

[3]The 6 neighbours of a voxel comprise just its face neighbours, when considered as a cube.

$$f(x,y,z) = \sum_{k=z-1}^{z+1} \sum_{j=y-1}^{y+1} \sum_{i=x-1}^{x+1} \text{state field of voxel } (i,j,k) \tag{6.4}$$

For a voxel (x,y,z) there is no need to apply the distance function (Equation 6.3), if:

- its state field is zero, or
- $f(x,y,z) = 27$ or -27 and $f(x',y',z') = 27$ or -27 for each 6-neighbour (x',y',z').

It is obvious that there is no need to apply the distance function when the state field is zero which indicates the voxel is on the surface. The second condition shows that if a voxel and all its 26-neighbours[4] are all interior (or all exterior) to the surface and all the 26-neighbours of its 6-neighbours have the same state, its distance value will not influence the surface display in any way. The first part states that a voxel need not be known if it is not used during linear interpolation of position, and the second part states that a voxel need not be known if it is not used during determination of the surface normal. This process eliminates many voxels from the expensive distance computation and identifies those voxels, and only those voxels, for which the distance function must be calculated.

Although the pre-processing stage (i.e. scan conversion) still has a worst case complexity of $O(\sum_{i=1}^{Nz} N_X \cdot N_Y \cdot N_{C_i})$, if the number of voxels on each slice S_i for which the distance must be calculated is R_i, the number of distance calculations is now $\sum_{i=1}^{Nz} R_i \cdot N_{C_i}$. Usually R_i is less than both X and Y, and is much less if a fine grid is used, therefore this method results in a reduction of an order of magnitude of the number of distance calculations. Furthermore, the complexity of the distance calculation can be reduced to $O(\sum_{i=1}^{Nz} R_i)$, if combined with the partitioning method. In practise this partitioning stage is difficult to implement and compute. A far easier method is to divide the grid up into sectors, and distribute each contour segment into its appropriate sector. Measuring the distance to a contour then becomes the simple task of finding the distance to the closest contour segment within the sector. Any sectors outside of this distance can be trivially removed from a list of candidate sectors, thus greatly reducing the number of contour segments to compute the distance against, and hence speeding up the process.

6.11 Results for Classical and Practical Problems

In this section the surfaces constructed for various contour data sets are examined and discussed. The contours have been deliberately chosen such that they show how this approach can cope with contours that other methods cannot. They have also been chosen to show how *good* the surfaces are, that are produced by this approach.

The branching problem has already been shown to be a complex problem, particularly when many contours branch into many. To demonstrate how well this method copes with branching, five cases are presented. Firstly the simple case of one contour branching into

[4]The 26-neighbours of a voxel comprise its 6 face neighbours, 12 edge neighbours and 8 vertex neighbours.

two (Figure 6.10(a)). Secondly, one contour branching into two, and approaching at a complex interface (Figure 6.10(b)). Thirdly, one contour branching into four (Figure 6.11(a)). Fourthly, a particularly difficult case of three contours branching into five (Figure 6.11(b)), and finally one contour branching into two contours and then into three (Figure 6.12(a)).

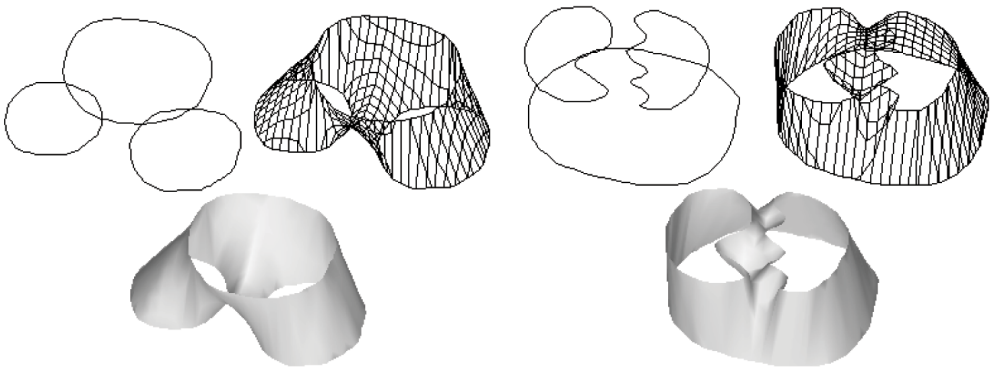


Figure 6.10: One contour branches into two at a (a) Simple interface. (b) Complex interface.

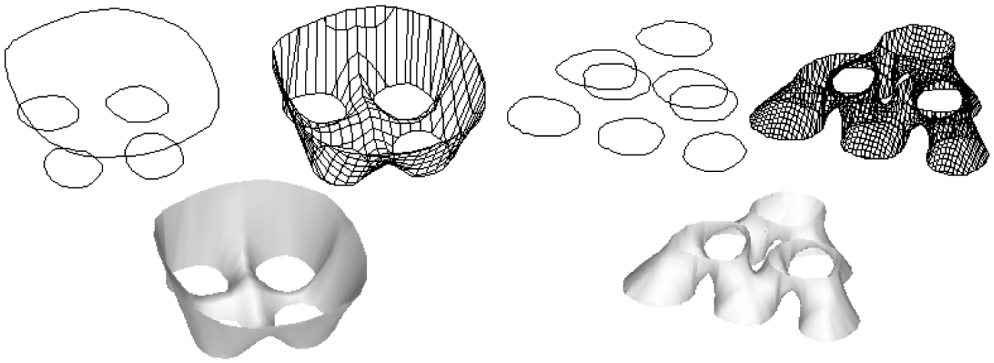


Figure 6.11: (a) One contour branches into four. (b) Three contours branch into five.

One contour branching into two can be effectively coped with by most tiling algorithms and the example is given to show that the surface is reconstructed just as effectively by this method. A far more difficult problem is that of branching into two contours where the contours approach at a complex interface. Previously user intervention or a complex tiling algorithm [117] would be required, but this method creates a surface without resorting to such techniques. To demonstrate one to many branching, one contour branching into four has been chosen. The resulting surface is a smooth representation of what one would expect the surface to look like. The difficult case of three contours branching into five also results in a natural looking surface and suggests that this method truly handles arbitrarily many to many branching without the complexity or failings of other methods. The final branching of one contour to two and then three contours has been highlighted to show that a case Ekoule et al.'s method cannot cope with (Section 6.3), can be solved using this approach. He stated there was insufficient information available to tile the case of two contour branching into three, and that more slices would be needed in the vicinity in order to determine the surface representation. As can be seen, this method requires no such information and produces a very natural looking surface. In fact, by adding the additional contour on top of the other

two slices, it is shown that the method correctly tiles around the *hole* that is present in the surface. The tiling is also very smooth and natural looking, which is a result of the distance function. By observation, this method robustly handles difficult many to many contour tiling cases with ease. It requires no extra information to be present with the contours, no user interaction and no tricky special procedures to create the tiling. The data is treated uniformly and consistently in every case in order to produce the surfaces.

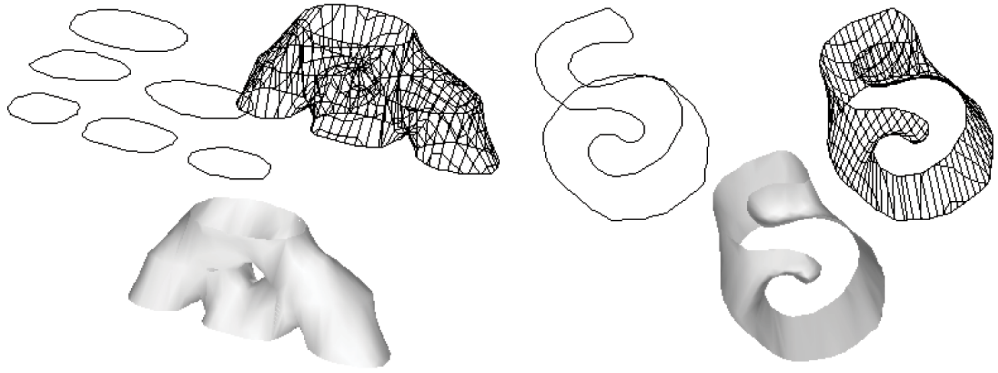


Figure 6.12: (a) One contour branches into two and then into three. (b) S to a circle.

The problem of greatly differing contours is also a major stumbling block for the tiling methods. To show that this method creates satisfactory surfaces for such situations, a number of cases have been chosen that involve quite complex contours in successive slices. The first is an S-shape corresponding to a circle (Figure 6.12(b)). Secondly, a thin slice successively becoming a square, a triangle and a circle (Figure 6.13(a)). Thirdly, a spiral becomes a circle (Figure 6.13(b)).

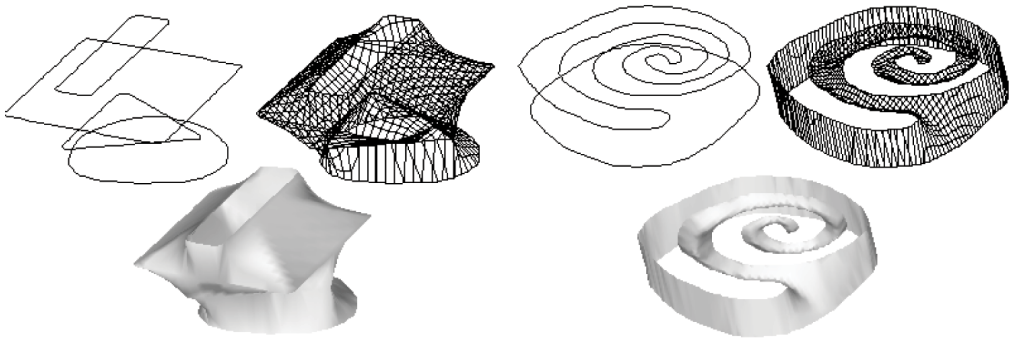


Figure 6.13: (a) Tube to square to triangle to circle. (b) Spiral becomes a circle.

The first example of an S-shape corresponding to a circle demonstrates that a mildly complex case produces a very reasonable, smooth surface. Such a contour pair would result in undesirable tilings using most methods. The method of Ekoule et al. [108] would produce the correct surface since both contours would be mapped to their convex hulls, and the tiling would take place between these contours. Problems would arise if the convex hulls of the contours differed greatly, and such a situation is given in the second case. Again the surface produced is convincing as a candidate surface. Finally as stated in Section 6.3 the mapping

of a spiral to a convex contour is an extremely difficult situation, but as can be seen by this particular surface, one that can be handled very well by this method.

The next example seeks to demonstrate how contour correspondence is determined. In the first example two similar contours do not overlap when viewed perpendicular to the plane, and result in two separate objects (Figure 6.14(a)). In the second case they do overlap very slightly and are treated as one object, namely as a tube (Figure 6.14(b)). The results may or may not match with a desired shape, but since both surfaces could be valid, it is difficult to say which is correct or not. However, it is commonplace to treat contours that do not overlap as separate objects and those that do as joined objects. This method certainly handles such cases consistently.

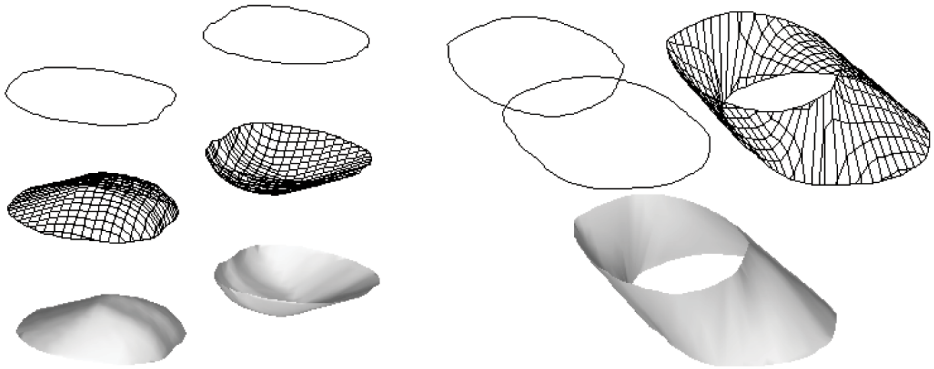


Figure 6.14: (a) Contours do not overlap. (b) Contours do overlap.

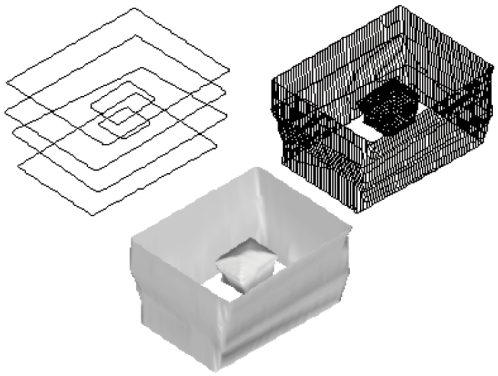


Figure 6.15: Internal object structure.

The final artificial problem shows a contour that has been constructed interior to another (Figure 6.15). This contour is present on the two middle slices, but not on the two outer ones. The reconstructed object has an exterior surface made from the joining of the four external contours, and an interior closed object made from the two internal contours. This method differs from the majority of other methods in that it handles such contour situations as contours that are describing the internal construction of an object. Effectively, this method can produce objects that have a determinable *thickness*. The two surfaces represent the boundary of the object, and are produced from the volume data, having their surface normals calculated from the data. The surface normals do indeed point outwards from the surface,

and so for the internal surface correctly point *inward* towards the empty centre of the object.

The practical application chosen is that of reconstructing the torso (Figure 6.16) and lungs (Figure 6.17) of a human male from just 17 and 12 contour slices respectively. The slices were obtained using a MRI scanner, and were first described in Section 6.8. Each contour was obtained by outlining the object within the slice using a mouse. The stack of slices were converted into a volume, and then a surface was reconstructed from them. Considering so few slices were used for the reconstruction the resulting surfaces are fairly detailed and smooth, with notable pectoral muscles, shoulder blades and abdomen on the torso.

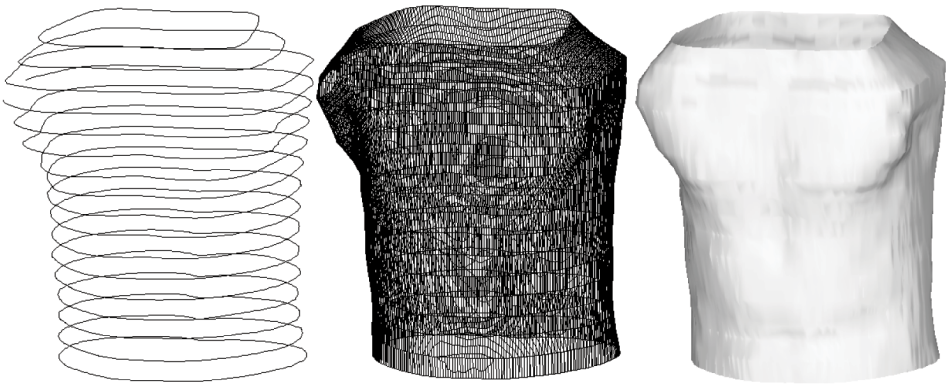


Figure 6.16: Torso reconstruction from 17 MRI slices.

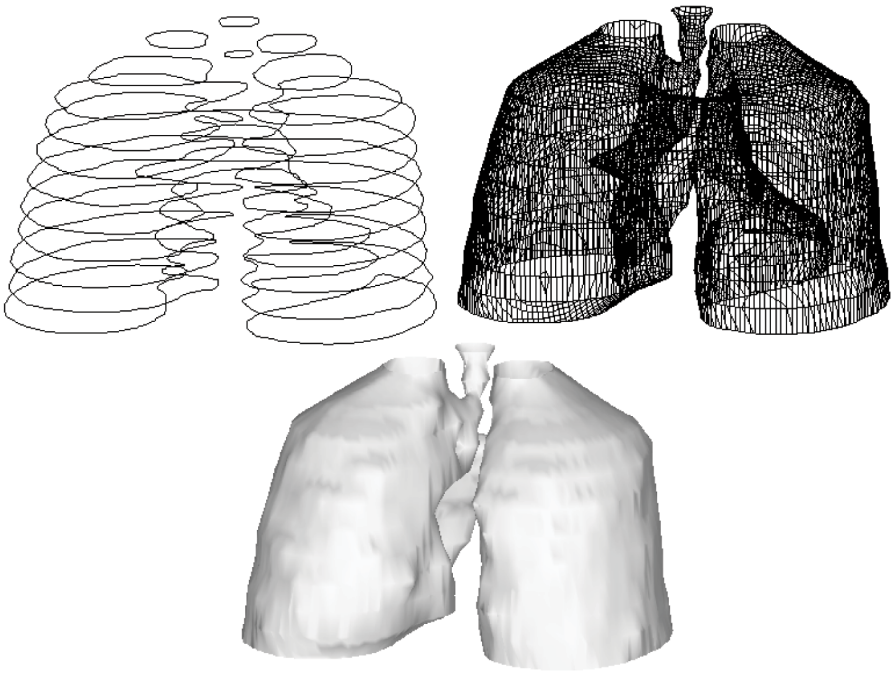


Figure 6.17: Lung reconstruction from 12 MRI slices.

The results of all this testing on a Dec Alpha model 3000/400 using C are recorded in Table 6.1, with timings given in seconds.

Fig.	Case	Contour Resolution	No. of Slices N_Z	Grid Size $N_X \times N_Y$	No. of Triangles	Volume Construction	Surface Construction
6.10(a)	1 to 2	200×200	2	50×50	531	0.133	0.02
6.10(b)	1 to 2	200×200	2	50×50	694	0.183	0.03
6.11(a)	1 to 4	200×200	2	50×50	630	0.133	0.02
6.11(b)	3 to 5	240×240	2	60×60	2892	1.000	0.07
6.12(a)	1-2-3	200×200	3	50×50	889	0.150	0.02
6.12(b)	S to O	200×200	2	50×50	770	0.200	0.02
6.13(a)	C-hulls	200×200	4	50×50	2332	0.300	0.05
6.13(b)	Spiral	240×240	2	60×60	2188	1.050	0.05
6.14(a)	Offset	200×200	2	50×50	1312	0.217	0.03
6.14(b)	Overlap	200×200	2	50×50	774	0.133	0.02
6.15	Interior	260×260	4	65×65	3768	1.083	0.10
6.16	Torso	512×512	17	64×64	8682	4.017	0.32
6.17	Lungs	420×420	12	84×84	10413	5.050	0.38

Table 6.1: Table showing testing results

6.12 Discussion

This method treats contour data in an altogether more consistent way than the previous method. The use of the field function converts the data into a volume of numerical data, from which the surface is derived. Since the surface is determined locally on a cube by cube basis, decisions about how *good* the tiling is, are redundant. Since the tiling can follow the surface arbitrarily through the volume, horizontal tilings are created automatically, without the need for user interaction or special tiling algorithms. The previous methods generated the surface by considering the contours of two slice planes at a time, whereas this method could be regarded as ignoring such structures and producing the surface globally from the volume as a whole by using local surfacing operations on cubes.

The result of using this new approach is that the data is treated consistently with no need for preprocessing. As an example it is often required that the contour segments are listed in an anti-clockwise manner, with the start vertices in close proximity. Such preprocessing is removed, and the contour segments and contour slices can be processed in arbitrary order.

The branching problem is handled quite well with all of the problematical cases being tiled efficiently. Satisfactory surfaces are created for complex branching situations involving complex contour to contour interfaces. The problem of tiling massively varying contours between slices is essentially solved. The method even produces a satisfactory surface for the highly complex spiral problem. The example tilings have also shown that objects with interior structures can be reconstructed from contour slices that have contours interior to other contours, and that the surface created encloses the solid object, and has the correct surface normals.

Finally, from Table 6.1 it can be seen that the time to create such surfaces is quite acceptable, even for the large problems of the lungs and torso. Taking as example the problem of

reconstructing the torso from MRI slices (first mentioned in Section 6.6). Using this method, one can rely on the surface produced from the contour data, and therefore there is no need to manually adjust badly placed tiles. A smooth surface can be produced from very few contour slices, and therefore the time required to isolate the object in the MRI scans is reduced. In the particular case of the torso reconstructed from 17 MRI slices, it took about 5 minutes to outline the object in all of the slices, create the volume data, and reconstruct and display the surface. The 5 minutes compares quite favourably with the day taken by MacLeod et al. mentioned in Section 6.6. The resulting surface is smooth, and compares very well with the surface they produce.

The advantages of this method enable it to be useful for the rapid visualisation of contour data. It is particularly suited to applications where few contour slices are available, and a good, smooth surface is required quickly. The user can select a few slices from the data, create the contour outlines and produce the surface automatically. Using other methods this would not be possible because contours would differ too greatly between slices, but using this method such a factor is not so important.

6.13 Conclusion

This chapter has reviewed the techniques that exist for the reconstruction of objects from contour data (Sections 6.2–6.6). It has shown that the failings of such methods are their inability to handle complex branching and differing contour shapes on successive slices. In addition to these main problems, these methods fail to reconstruct solid objects with holes, and require many restrictions on the contour data to be adhered to. In contrast a new method for the reconstruction of surfaces from contour data has been presented (Sections 6.7–6.12). This method correctly tiles complex cases, and is able to reconstruct objects with interior structures. It requires only two restrictions to be present with the data, namely that the contours are closed and non-self-intersecting, which is far less restrictive than other methods. The surfaces produced are smooth, and natural looking, and the method produces them by treating the data in a consistent and appealing manner. The many test cases demonstrate that this method is reliable, automatic, effective and computationally inexpensive when producing surfaces from contour data.

This work [4] was presented at Eurographics 1994 (Oslo, Norway), and appeared in Computer Graphics Forum 13:3, pp C-75–C-84, under the title "A New Approach to the Construction of Surfaces from Contour Data".

Chapter 7

Conclusions

This thesis has been a thorough investigation of the area of visualisation of regular three dimensional data. The main contributions are new methods for:

- reconstructing surfaces from contour data;
- constructing voxel data from triangular meshes;
- real-time manipulation through the use of cut planes;
- ultra high quality and accurate rendering.

Various other work has been presented which reduces the amount of calculations required during volume rendering, reduces the number of cubes that need to be considered during surface tiling and the combined application of particle systems and blobby models with high quality, computationally efficient rendering.

All these methods have offered new solutions and improved existing methods for the construction, manipulation and visualisation of volume data.

In addition to these new methods this work acts as a review and guide of current state of the art research, and has given in depth details of implementations and results of well known methods. Comparisons have been made using these results of both computational expense and image quality, and these could serve as a basis for the consideration of what visualisation technique to use for the resources available and the presentation of the data required. Reviews of each main visualisation topic have been presented, in particular the review of volume rendering methods covers much of the recent research. Complementing this is the comparison of many alternate viewing models and efficiency tricks in the most thorough investigation to this researcher's knowledge. During the course of this research many existing methods have been implemented efficiently, in particular the surface tiling technique, and a method for measuring the distance between a point and a 3D triangle.

In detail, the main results of this research are:

Chapter 2

- The new indexing method for volume data is equal in efficiency terms to the most efficient octree method.
- The realisation of the requirements for an efficient surface tiler, and the implementation of such a surface tiler.
- The solution of ambiguous cases through the use of an additional case table.

Chapter 3

- The new stepping method produces images with a better quality/efficiency trade-off than most other acceleration methods, and the best trade-off if combined with other acceleration methods such as adaptive rendering.
- The important models and acceleration techniques have been thoroughly contrasted.
- An in-depth review of the current state of the art research has been made.

Chapter 4

- Real-time cutting operations are possible using an intersection buffer and the ultra high quality direct surface rendering technique.
- The main voxel shading techniques have been reviewed and compared.

Chapter 5

- Voxelisation of polygonal meshes has been shown to be achievable both efficiently and accurately.
- Display of such voxelised data can be best achieved using the direct surface rendering technique.
- The direct surface rendering technique can be used to efficiently visualise blobby models.
- Effective animations of blobby models can be achieved using particle systems.

Chapter 6

- The classical contour connection problems are shown to be solvable using a new approach.
- The new method is shown to be computationally efficient for practical problems.

- High quality images can be computed automatically.
- Hugely varying contours can be connected with this method.

Parts of this research have been presented at Eurographics 1994 (Oslo), the fifth Eurographics Workshop for Visualisation in Scientific Computing 1994 (Rostock), and the 13th UK Eurographics Conference 1995 (Loughborough).

Glossary

adaptive - altering the focus of the computation in order to concentrate on parts that are more important to the solution. Computation takes place in areas of rapid change, leaving parts that contribute less significantly to be computed last or not at all. For example adaptive termination, adaptive rendering and adaptive re-meshing.

aliasing - a problem associated with the discrete nature of object edge representation. A binary decision occurs at each pixel – it is either inside the object or outside. This leads to a *staircase* effect or *jaggy* at the edge of objects. It is most noticeable during animation.

ambient light - the light that falls upon an object that is an amalgamation of light from all sources, for example, light sources and reflections.

anti-alias - the process of filtering an image so that sharp edges are softened by *averaging* neighbouring pixel intensities, in order to remove large intensity differences between neighbouring pixels.

area light source - a source of light that occupies some finite area in object space. See also *shadow rays*.

area sampling - values are evaluated at various points in space. In *unweighted* sampling, the sum of the samples is divided by the total number of samples. In *weighted* sampling values are multiplied by constants according to their position. Typically central values are given higher weightings in order to preserve their contributions.

backface culling - a method of reducing the number of polygons of a polygon mesh to be displayed, by removing those that face away from the viewer in a preprocessing step. Polygons facing away are selected by determining the dot product of the normal of the polygon with the view vector.

bicubic surfaces - curved surfaces made up from parametric bivariate polynomial surface patches which are defined using three bivariate polynomials, one for each of x, y and z. Polynomials of various degrees can be used, but bicubic surfaces are made from cubic polynomials in both parameters.

bitmap - an image consisting of n pixels horizontally and m pixels vertically can be represented by $n \times m$ bits which determine whether a pixels is on (high intensity) or off (low intensity).

blobby models - surfaces created by isosurfaces of a certain threshold of field data, where each value has been calculated as the sum of the contribution of each primitive. The value

contributed by each primitive is calculated according to its strength and decay.

boundary representation (b-rep) - the term given to the representation of a surface boundary using vertices, edges and faces to approximate the object to be represented. See also *polygonal meshes.*

bounding volume - a complex object is completely enclosed by a simpler object such as a sphere or box. Computation such as visibility takes place on the bounding volume and then only takes place on the more complex object if valid. See also *octree.*

bsp tree - subdivides space by successively dividing space into pairs of subspaces using a plane of arbitrary position and orientation. Each internal node has two pointers, one for each side of the plane. Leaf nodes contain those objects that occur in the subspace represented by that path through the tree.

bump mapping - the normals computed for some surfaces result in the surface looking very smooth. In order to introduce a roughness into the object, the computed normals are perturbed slightly in a random direction. The new normal is used to shade the object to give a stipple, or bumpy, effect.

classification (of elements) - Elements are classified according to position, value, and neighbouring values. In this way elements can be built into groups which represent objects upon which it is desirable for computation to take place collectively.

coherence - If there is a mapping between objects at a low level, and objects at a higher level, taking advantage of the coherence of the objects is achieved by using computation upon the groups to reduce the computation required for processing the objects in that group.

colour table - an array of (usually) RGBα tuples, where α is *opacity.* The array is indexed by the value of the objects to be displayed (for example pixels or voxels).

compositing - the process of merging together all the colours and opacities that contribute to one pixel of an image. The composition can take place in a back to front manner or a front to back manner.

constant shading - one normal is evaluated for a polygon to be shaded, and the pixels the polygon projects onto are all shaded the same intensity calculated by the shading model using some function of the normal.

convolution - a mathematical technique for combining two functions. In computer graphics its refers to the application of a function known as a filter function to the discrete function defining the image. Such filter functions can be defined to *anti-alias* an image, detect edges or enhance certain features.

CT scan - an image produced of a cross section of an object using a computed tomography (CT) scanner. The image describes the amount of X-ray absorbed by each of many discrete points on one slice plane taken through the object.

deformable models - object descriptions that not only represent the physical appearance of the object but also the physical characteristics of the object. Such objects are often described by particles connected by *springs* which allow the objects to deform under external forces, and revert to their original state using internal forces. Such deformable models are ideally

suited to modelling elastic objects acted upon by physical forces such as gravity, friction, compressible force, extensible force and even heat.

depth cueing - the attenuation of each pixel's colour in an image due to the distance of the object from the pixel using the view coordinates. The pixel's intensity, I is

$$I = s_0 I' + (1 - s_0) I_{dc}$$

$$s_0 = \frac{Max - z_0}{Max}$$

where Max is the maximum depth of any part of the object from the view plane, and z_0 is the depth of the part of the object from the current pixel. I_{dc} is the intensity of the depth cue colour and I' is the calculated intensity of the surface.

depth only shading - the distance z, is calculated from each pixel in the image to the closest object in the scene to be displayed. The intensity of the image pixel I is $I = 1 - \frac{z}{Z}$ where Z is the maximum distance any object can be from any pixel.

directional light source - see *parallel light source.*

extended light source - see *area light source.*

footprint - the area covered by the projection of an object when it is projected onto a plane. For example, a spherical voxel will have either an elliptical or circular footprint.

global illumination - the light falling upon any object in the scene is accurately calculated from the contribution of light from every other object in the scene.

Gouraud shading - a technique for determining the intensity of the shading for a polygon. The normals are calculated at the vertices of the polygon, and are determined along the edges of the polygon using linear interpolation. Intensities at the edges of the polygon are calculated by the shading model using the interpolated normal. Intensity within the polygon is calculated according to the linear interpolation of the intensities at the edges of the polygon.

gradient shading - a method of using a z-buffer to calculated surface normals which can then be used to compute pixel intensities using a shading model. The z-buffer is an array $z(x, y)$ of depths for each pixel (x, y). The gradient at (x, y) can be calculated using

$$\nabla z = \left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, 1 \right)$$

At pixel (x, y) $\frac{\partial z}{\partial x}$ can be approximated by the backward difference

$$\delta_b = z(x, y) - z(x - 1, y)$$

by the forward difference

$$\delta_f = z(x + 1, y) - z(x, y)$$

by the central difference

$$\delta_c = \frac{1}{2}(z(x + 1, y) - z(x - 1, y))$$

or by a weighting of all three, and similarly for $\frac{\partial z}{\partial y}$.

hidden surface - the parts of a surface that are not visible to the viewer because they are obscured by other parts. Hidden surface removal is the process of determining the hidden surface so that it may be removed from the viewing pipeline.

inbetweening - the process of interpolating objects between two key objects. If the two key objects (for example keyframes for images) are V_0 and V_k, the process of inbetweening produces $k-1$ interpolated objects – $V_1, \ldots, V_{k-1}$, which smoothly transform object V_0 into object V_k according to some criteria.

intensity - pixels, light sources and discrete points have intensity associated with them. It is usually either a grey-scale value representing brightness, or a triple (I_r, I_g, I_b) where I_λ represents the intensity at that wavelength ($r =$ red, $g =$ green, $b =$blue). Often an additional intensity (I_α) indicates the *opacity* of the point.

interpolation - the function of determining a new vector between two existing vectors using the existing vectors and the distance between them and the vector to be determined. For instance the vector could be as complex as a whole image, or as simple as a single real number. Interpolation can be linear, where the in-between vectors vary linearly according to distance, or some higher order function.

isosurface - a surface of equipotential. For a function $f : \Re^i \to \Re$, the τ-isosurface, S, is defined as

$$S = \left\{ x^i : f(x^i) = \tau \right\}$$

Every point on the surface has a function value of τ, and no other points in the domain have a function value of τ. For $i = 2$, S would be referred to as isolines or contours.

light source - the set of points in space that illuminate the surrounding space. A light source is usually associated with a triple (I_r, I_g, I_b) where I_λ ($\lambda = r, g, b$) is the intensity of light for that particular wavelength (r=red, g=green, b=blue).

mach bands - the term given to the discontinuous lighting effect observed on polygon boundaries in a polygon mesh description of a surface when a lighting model such as Gouraud shading is used. For the case of Gouraud shading, pixel intensities are C^0 continuous across polygon boundaries, but not C^1 continuous. The eye interprets this discontinuity in the form of *banding*. A higher order shading model such as Phong shading creates C^1 continuity which removes much of this effect.

marching cubes - a popular algorithm for determining isosurfaces in 3D. Function values are evaluated at regular discrete points to make a 3D grid of data values (voxels). Eight neighbouring voxels (four on one slice of data, and four aligned in an adjacent slice) make up a cube, with which the surface can be determined using look up tables for each of the possible cases. Since each of the eight voxels can be inside or outside of the surface, there exist 256 cube configurations. The look up table indicates the triangles to be added to a triangular mesh, with their vertices interpolated from known voxel positions and values.

motion blur - the process of simulating the dynamics of moving objects by blurring them in the direction of motion. This occurs in the reality of photography or cinematography where the camera shutter is open for a finite length of time, and objects have a chance to move, and create multiple images (blur) on one frame. Simulating this from computer images, both

stills and animations, adds to the realism of the image.

MRI scan - an image produced of a cross section of an object using a Magnetic Resonance Imaging (MRI) scanner. The image describes the disturbance of the magnetic field by each of many discrete points on one slice plane taken through the object.

Nyquist rate - the lowest sampling rate that can allow a proper reconstruction of a signal. If the signal has a component with frequency f_λ which is the highest-frequency component, then the *Nyquist rate* is $2f_\lambda$. If less samples are used, the reconstructed signal may not be correct, and can represent high frequency components as a lower frequency signal. This misrepresentation is known as *aliasing.*

octree - a tree data structure where each node has up to eight children. It is most often used to partition 3D structures since the children of each node partition the space represented by the parent node. The process to be computed can be computed on a node. If successful the process is carried out on each of the eight children in turn. If unsuccessful, the part of the volume represented by that node can be ignored. For example – octrees are used as bounding volumes in ray tracing. If the ray being tested successfully intersects a node, it is tested against each of the children of that node, or if a leaf node, it tests for intersection with the objects within the volume bounded by the leaf. If the ray does not intersect the node, all the objects within that node are removed from the intersection computation.

opacity - usually a value between 0 and 1 which represents how dense an object is to light. If the opacity is 1 the object is opaque and obscures everything behind it, if the opacity is 0 it is completely transparent, and cannot be seen. Values between 0 and 1 represent, proportionally and linearly varying, the visibility of the object and objects behind it.

parallel light source - a light source from which light falls on all objects as if from the same direction. It occurs when the light is infinitely far away from the scene.

parallel projection - the production of images wherein the projectors are all parallel to each other and they are defined by a single direction vector.

particle systems - composed by a collection of particles that evolve over time. The particles have attributes associated with them such as mass, charge, or colour. The system evolves according to either random events, such as dying, introducing new particles and random motion, or by following equations for motion such as following trajectories according to initial velocity and gravity and following velocity vectors in numerical simulations.

perspective projection - the production of images using perspective. All the projectors pass through one point known as the centre of projection.

photorealistic - artificial images that closely resemble what you would see in reality, if positioned at the appropriate viewpoint in space and looking at the scene domain from the appropriate angle.

physically based modelling - creating an accurate behavioural model of an object based upon its physical properties and those of its surroundings. The motion and object shape are determined by the forces acting upon the object, and are calculated using interesting mathematics. **pixel** - (picture element) is the name given to a value in a 2D image.

point light source - source of light in which all light rays radiate outwards from a single point. Light bulbs can be best approximated by point light sources.

polygonal meshes - a form of object representation. The object's surface is approximated by a set of planar polygons in which each edge is used by at most two polygons, and each polygon is a closed set of edges. A visualisation of the object represented by the mesh is produced using any of a number of *rendering* techniques.

primary rays - originate from each pixel in the image in the direction of the view plane normal for **parallel projection**, or from the centre of projection in the direction of the pixel in **perspective projection**. The ray intersects surfaces that lie along the ray, and thus spawns *secondary rays* to compute the contributions to the pixel's intensity due to object *reflection*, *refraction* and *shadow*. These contributions are combined with the object's intensity to give the intensity of the pixel.

radiosity - models the lighting in a scene by using an analogy of thermal dynamics. The light in a closed scene follows the law of conservation of energy, and all light must be accounted for. The light at any surface is the sum contribution of all the light energy falling on that surface from all light sources and object reflections. Radiosity methods calculate the interaction of light in a view independent pre-processing step, and the resulting scene description can be rendered from this model using conventional techniques. Radiosity accurately models *global illumination* and removes the need for the *ambient lighting* term.

ray casting - the process of sending a primary ray from a point in space into a scene with direction. Upon encountering an object the ray terminates, and the intensity of the point is calculated according to the shading model.

ray tracing - is the process of sending a primary ray from a point in space into a scene with direction. Upon intersection with an object secondary rays are spawned in order to determine all contributions to the intensity of that point. Secondary rays determine contributions due to *reflection*, *refraction and transparency*, *shadows*, and *light sources*.

reconstruction - the process of determining an object from a partial description of the object. For examples, reconstruction of polygonal meshes from contours and reconstruction of objects from laser range data.

reflection rays - rays spawned from objects which are partial or total mirrors, in order to determine what is visible at the mirror. The rays are treated like primary rays, and may spawn other rays, including reflections rays, upon hitting an object. The origin of the ray is the intersection point with the object, and the angle of incidence is calculated using the direction of the incoming ray, and the surface normal, with the intersection point being the point of incidence.

rendering - the process of producing an accurate impression of a scene using any of the shading models, and any of the rendering techniques such as ray tracing or scan conversion.

scan conversion - the process of calculating pixel intensities from object descriptions in an ordered pixel by pixel, line by line way for each object.

scan line - one horizontal line in an image, derived from the line that a cathode ray tube sweeps (scans) across during anyone pass.

secondary rays - produced at the point of surface intersection during ray tracing. They are created in order to find the exact colour of the intersection point due to contributions from all objects that influence the light at that point, for example, light sources, shadows created by other objects and partial transparency of the object itself and objects that lie on the path between it and a light source. These rays are determined in the same way as primary rays except that the direction is some function of the view normal and surface normal, and the resulting colours contribute partially rather than totally to the final pixel colour.

segmentation - the process of determining similar objects with respect to certain defined parameters and grouping them together.

shadow rays - rays spawned from objects in order to determine whether or not any objects occur on a path between that point and each light source which may place the point in shadow. The origin of each shadow ray is the intersection point with the object, and the directions are the directions to each light source. If the light source is an area light source, one method is to trace n rays to random points on the area, and if m rays get through, the fraction of light falling on the point from that light source is $\frac{m}{n}$.

shadows (umbra/penumbra) - created by objects obscuring light sources from an object. In the case of area light sources the shadow will be a transition from points that can see most of the light, to points that can see no light at all. Those that are in complete shadow are said to be covered by the umbra, and those that are in partial shadow are in the penumbra.

spatial partitioning - the process of dividing space into smaller subspaces. The reasoning behind this is the fact that processes can be carried out on fewer objects in the particular subspace they relate to. See *octree* and *bsp tree* for examples of spatial partitioning techniques.

spatial subdivision - see *spatial partitioning.*

specular reflection - the highlight that can be seen on a shiny surface from particular angles. Perfect mirrors reflect light in only the direction of the reflection according to the angle of incidence, whereas any other shiny object reflects light unequally in different directions, and so the highlight is seen from different angles.

splatting - the process of projecting voxels onto an image plane, throwing them against the image so that they *splat* their colour and opacities on the image. The distribution of a voxel's energy is calculated, and the energy is added to each pixel it projects onto by compositing the colour and opacity at that point in proportion to the energy.

spline - the name given to a line (in 2D) or a surface (in 3D) that has been produced using knots (control points). To describe an object, a small number of knots are stored which can then be used to generate a smooth surface or curve. Spline curves can either be *interpolated* from the knots, i.e. the surface passes through each knot, or they can be *approximated* to the knots, i.e. the curve passes close but not necessarily through the knots.

stereo images - images that have been created from two images produced of the same scene at slightly differing angles and view points, in order to approximate the positions of the viewer's eyes. The images can be combined as one using one wavelength for one image, and a different wavelength for the second image (usually red and blue), and then viewing the composite through the appropriate glasses. Images can also be produced in the form

of random dot stereograms, or by displaying the two images alternately, and relying upon additional hardware (glasses) to be able to synchronise the left eye with the left image, and the right eye with the right image.

subsampling - the process of obtaining values more coarsely than is usual, and interpolating the inbetween values. For example, instead of tracing a ray for each pixel in an image, rays are traced coarsely over the whole image with extra rays being traced in the regions of high change. Other pixels are interpolated from known pixels in the process known as *adaptive rendering.*

supersampling - the process of obtaining values more finely than is usual, and averaging neighbouring values to produce a more accurate sample of the point being determined. For example, a form of *anti-aliasing* - is to supersample the image, and then average neighbouring values to produce a smaller, but more accurate image.

surface normal - the vector which points from a particular point on the surface, outwards away from the object so that the angle between the normal and the tangent to the surface in every direction, is the same in every direction.

threshold - a value which represents something interesting in a particular domain. For example, in 3D *isosurfacing* the threshold defines the surface in which we are interested in, and is calculated such that all points on the surface have a function value equal to the threshold.

transparency - the inverse of **opacity**.

viewing pipeline - refers to the process of transforming an object description (usually a mesh of vertices) to screen coordinates. It typically involves the view transformation of the object coordinates, clipping the object to the visible volume, and projecting the object. Often additional steps for mapping these coordinates to the screen coordinates are included.

volume rendering - the process of obtaining images from three dimensional volume data by treating the data as cloudy material. Each value in the data has a colour and opacity assigned to it during a classification stage. Rays are cast for each pixel into the volume, and the colour and opacity are sampled at evenly distributed points along the ray. These samples are composited using standard techniques to produce the accumulative colour and opacity reaching the pixel. Images produced by this method usually have several differently colour semi-transparent surfaces overlaid on an opaque surface, for example, semi-transparent skin overlyed on opaque bone for an image of a CT scan.

volume visualisation - the general term given to the process of displaying three dimensional data, and encompasses the methods of isosurfacing, volume rendering and splatting.

voxel - (volume element) the name given to a value in 3D space.

voxelisation - the process of producing 3D data from an object description, such as polygonal meshes, so that each voxel has a value. The data can then be rendered using a volume visualisation technique to produce an image of the original object.

Bibliography

[1] M. W. Jones and M. Chen. Fast cutting operations on three dimensional volume datasets. In *Visualization in Scientific Computing*, pages 1–8. Springer-Verlag, Wien New York, January 1995.

[2] M. W. Jones. Voxelisation of polygonal meshes. In *Proceedings of 13th Annual Conference of Eurographics (UK Chapter) (Loughborough, March 28-30, 1995)*, pages 160–171, March 1995.

[3] M. W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, December 1996.

[4] M. W. Jones and M. Chen. A new approach to the construction of surfaces from contour data. *Computer Graphics Forum*, 13(3):75–84, September 1994.

[5] M. W. Jones. Glossary. In *Proceedings of High Performance Computing for Graphics and Visualisation (Swansea, Wales, July 3–4, 1995)*, July 1995.

[6] W. E. Lorensen and H. E. Cline. Marching Cubes : A high resolution 3D surface construction algorithm. In *Proc. SIGGRAPH '87 (Anaheim, Calif., July 27-31, 1987)*, volume 21(4), pages 163–169. ACM SIGGRAPH, New York, July 1987.

[7] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[8] Å. Wallin. Constructing isosurfaces from CT data. *IEEE Computer Graphics and Applications*, 11(6):28–33, November 1991.

[9] M. J. Dürst. Letters: Additional reference to "marching cubes". *Computer Graphics (ACM SIGGRAPH)*, 22(2):72, April 1988.

[10] J. Wilhelms and A. V. Gelder. Topological considerations in isosurface generation – Extended abstract. *Computer Graphics*, 24(5):79–86, November 1990.

[11] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proc. Visualization 91*, pages 83–90. IEEE CS Press, Los Alamitos, Calif., 1991.

[12] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2:227–234, 1986.

[13] B. A. Payne and A. W. Toga. Surface mapping brain function on 3D models. *IEEE Computer Graphics and Applications*, 10(5):33–41, September 1990.

[14] H. Müller and M. Stark. Adaptive generation of surfaces in volume data. *The Visual Computer*, 9:182–199, 1993.

[15] H. Yun and K. H. Park. Surface modelling method by polygonal primitives for visualizing three-dimensional volume data. *The Visual Computer*, 8:246–259, 1992.

[16] R. S. Avila, L. M. Sobierajski, and A. Kaufman. Visualizing nerve cells. *IEEE Computer Graphics and Applications*, 14(5):11–13, September 1994.

[17] P. Ning and J. Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, November 1993.

[18] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proc. SIGGRAPH '93 (Anaheim, Calif., August 1-6, 1993)*, volume 27(2), pages 19–26. ACM SIGGRAPH, New York, August 1993.

[19] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Proc. SIGGRAPH '92 (Chicago, Illinois, July 26-July 31, 1992)*, volume 26(2), pages 65–70. ACM SIGGRAPH, New York, July 1992.

[20] G. Turk. Retiling polygonal surfaces. In *Proc. SIGGRAPH '92 (Chicago, Illinois, July 26-July 31, 1992)*, volume 26(2), pages 55–64. ACM SIGGRAPH, New York, July 1992.

[21] J. V. Miller, D. E. Breen, W. E. Lorensen, R. M. O'Bara, and M. J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. In *Proc. SIGGRAPH '91 (Las Vegas, Nevada, July 28-August 2, 1991)*, volume 25(4), pages 217–226. ACM SIGGRAPH, New York, July 1991.

[22] R. S. Gallagher and J. C. Nagtegaal. An efficient 3D visualization technique for finite element models and other coarse volumes. In *Proc. SIGGRAPH '89 (Boston, Mass., July 31-August 4, 1989)*, volume 23(3), pages 185–194. ACM SIGGRAPH, New York, July 1989.

[23] C. Zahlten and H. Jürgens. Continuation methods for approximating isovalued complex surfaces. In *Computer Graphics Forum, Proc. Eurographics '91*, volume 10(3), pages 5–19. University Press, Cambridge, UK., September 1991.

[24] J. K. Udupa and D. Odhner. Fast visualization, manipulation, and analysis of binary volumetric objects. *IEEE Computer Graphics and Applications*, 11(6):53–62, November 1991.

[25] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[26] P. Sabella. A rendering algorithm for visualizing 3D scalar fields. In *Proc. SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988)*, volume 22(4), pages 51–57. ACM SIGGRAPH, New York, August 1988.

[27] C. Upson and M. Keeler. V-Buffer : Visible volume rendering. In *Proc. SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988)*, volume 22(4), pages 59–64. ACM SIGGRAPH, New York, August 1988.

[28] L. Carpenter R. A. Drebin and P. Hanrahan. Volume rendering. In *Proc. SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988)*, volume 22(4), pages 65–74. ACM SIGGRAPH, New York, August 1988.

[29] J. D. Foley, A. Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, 2nd ed.* Addison-Wesley, 1990.

[30] A. Low. *Introductory computer vision and image processing.* McGraw-Hill Book Company (UK) Limited, Maidenhead, Berkshire, 1991.

[31] P. Burger and D. Gillies. *Interactive Computer Graphics.* Addison-Wesley, 1989.

[32] G. T. Herman, J Zheng, and C. A. Bucholtz. Shape-based interpolation. *IEEE Computer Graphics and Applications*, 12(3):69–79, May 1992.

[33] M. W. Vannier, J. L. Marsh, and J. O. Warren. Three dimensional computer graphics for craniofacial surgical planning and evaluation. In *Proc. SIGGRAPH '83 (Detroit, Michigan, July 25-29, 1983)*, volume 17(3), pages 263–273. ACM SIGGRAPH, New York, July 1983.

[34] A. Kaufman, K. H. Höhne, W. Krüger, L Rosenblum, and P Schröder. Research issues in volume visualization. *IEEE Computer Graphics and Applications*, 14(2):63–67, March 1994.

[35] A. Pommert, B. Pflesser, M. Riemer, T. Schiemann, R. Schubert, U. Tiede, and K. H. Höhne. Advances in medical volume visualization. Technical Report ISSN 1017–4656, Eurographics, 1994.

[36] K. H. Höhne, M. Bomans, M. Riemer, R. Schubert, U. Tiede, and W. Lierse. A volume-based anatomical atlas. *IEEE Computer Graphics and Applications*, 12(4):72–78, July 1992.

[37] T. R. Nelson and T. T. Elvins. Visualization of ultrasound data. *IEEE Computer Graphics and Applications*, 13(6):50–57, November 1993.

[38] R. Yoshida, A. Doi T. Miyazawa, and T. Otsuki. Clinical planning support system - CliPSS. *IEEE Computer Graphics and Applications*, 13(6):76–84, November 1993.

[39] D. R. Ney and E. K. Fishman. Editing tools for 3D medical imaging. *IEEE Computer Graphics and Applications*, 11(6):63–70, November 1991.

[40] R. Yagel and A. Kaufman. Template-based volume viewing. In *Computer Graphics Forum, Proc. Eurographics '92 (Cambridge, UK, September 7-11, 1992)*, volume 11(3), pages C–153–C–167. University Press, Cambridge, UK., September 1992.

[41] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Proc. of Eurographics '87, Amsterdam, The Netherlands*, pages 3–9, August 1987.

[42] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6:2–7, 1990.

[43] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[44] L. Westover. Footprint evaluation for volume rendering. In *Proc. SIGGRAPH '90 (Dallas, Texas, August 6-August 10, 1990)*, volume 24(4), pages 367–376. ACM SIGGRAPH, New York, August 1990.

[45] R. A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Proc. Visualization 93*, pages 261–266. IEEE CS Press, Los Alamitos, Calif., 1993.

[46] D. Laur and P Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proc. SIGGRAPH '91 (Las Vegas, Nevada, July 28-August 2, 1991)*, volume 25(4), pages 285–288. ACM SIGGRAPH, New York, July 1991.

[47] N. Max, R. Crawfis, and D. Williams. Visualization for climate modeling. *IEEE Computer Graphics and Applications*, 13(4):34–40, July 1993.

[48] J. Wilhelms and A. V. Gelder. A coherent projection approach for direct volume rendering. In *Proc. SIGGRAPH '91 (Las Vegas, Nevada, July 28-August 2, 1991)*, volume 25(4), pages 275–284. ACM SIGGRAPH, New York, July 1991.

[49] P. L. Williams. Interactive splatting of nonrectilinear volumes. In *Proc. Visualization 92*, pages 37–44. IEEE CS Press, Los Alamitos, Calif., 1992.

[50] J. Wilhelms. Pursuing interactive visulization of irregular grids. *The Visual Computer*, 9:450–458, 1993.

[51] P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[52] A. Van Gelder and J. Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering (extended abstract. In *Proc. Visualization 93*, pages 70–77. IEEE CS Press, Los Alamitos, Calif., 1993.

[53] T. Frühauf. Raycasting of nonregularly structured volume data. In *Computer Graphics Forum, Proc. Eurographics '94 (Oslo, Norway, September 12-16, 1994)*, volume 13(3), pages C–293–C–303. University Press, Cambridge, UK., September 1994.

[54] J. K. Udupa and D. Odhner. Shell rendering. *IEEE Computer Graphics and Applications*, 13(6):58–67, November 1993.

[55] P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *Proc. Visualization 93*, pages 11–18. IEEE CS Press, Los Alamitos, Calif., 1993.

[56] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics. *IEEE Computer Graphics and Applications*, 8(3):48–68, May 1988.

[57] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[58] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *Proc. Visualization 93*, pages 62–69. IEEE CS Press, Los Alamitos, Calif., 1993.

[59] K. R. Subramani and D. S. Fussell. Applying space subdivision techniques to volume rendering. In *Proc. Visualization 90*, pages 150–159. IEEE CS Press, Los Alamitos, Calif., 1990.

[60] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Towards a comprehensive volume visualisation system. In *Proc. Visualization 92*, pages 13–20. IEEE CS Press, Los Alamitos, Calif., 1992.

[61] P. A. Fletcher and P. K. Robertson. Interactive shading for surface and volume visualization on graphics workstations. In *Proc. Visualization 93*, pages 291–298. IEEE CS Press, Los Alamitos, Calif., 1993.

[62] K. L. Ma, M. F. Cohen, and J. S. Painter. Volume seeds: A volume exploration technique. *The Journal of Visualization and Computer Animation*, 2:135–140, 1991.

[63] B. Corrie and P. Mackerras. Data shaders. In *Proc. Visualization 93*, pages 275–282. IEEE CS Press, Los Alamitos, Calif., 1993.

[64] J. L. Montine. A procedural interface for volume rendering. In *Proc. Visualization 90*, pages 36–44. IEEE CS Press, Los Alamitos, Calif., 1990.

[65] A. Pang and K. Smith. Spray rendering : Visualization using smart particles. In *Proc. Visualization 93*, pages 283–290. IEEE CS Press, Los Alamitos, Calif., 1993.

[66] K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke. 3D visualization of tomographic volume data using the generalized voxel model. *The Visual Computer*, 6:28–36, 1990.

[67] A. Pommert, M. Bomans, and K. H. Höhne. Volume visualization in magnetic resonance angiography. *IEEE Computer Graphics and Applications*, 12(5):12–13, September 1992.

[68] K. D. Toennies, J. Udupa, G. T. Herman, I. L. Wornom III, and S. R. Buchman. Registration of 3D objects and surfaces. *IEEE Computer Graphics and Applications*, 10(3):52–62, May 1990.

[69] L. Caponetti and A. M. Fanelli. Computer-aided simulation for bone surgery. *IEEE Computer Graphics and Applications*, 13(6):86–92, November 1993.

[70] T. S. Yoo, U. Neumann, H. Fuchs, S. M. Pizer, J. Rhoades T. Cullip, and R. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics and Applications*, 12(4):63–71, July 1992.

[71] G. Frieder, D. Gordon, and R. A. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–60, January 1985.

[72] R. A. Reynolds, D. Gordon, and L. S. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38:275–298, 1987.

[73] D. Gordon and J. K. Udupa. Fast surface tracking in three dimensional binary images. *Computer Vision, Graphics, and Image Processing*, 45:196–214, 1989.

[74] U. Tiede, K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, and G. Wiebecke. Investigation of medical 3D-rendering algorithms. *IEEE Computer Graphics and Applications*, 10(2):41–53, March 1990.

[75] H. K. Tuy and L. T. Tuy. Direct 2D display of 3D objects. *IEEE Computer Graphics and Applications*, 4(10):29–33, October 1984.

[76] D. Gordan and J. K. Udupa. Image space shading of three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 29:361–376, 1985.

[77] D. Cohen, A. Kaufman, R. Bakalash, and S. Bergman. Real time discrete shading. *The Visual Computer*, 6:16–27, 1990.

[78] R. Yagel, D. Cohen, and A. Kaufman. Normal estimation in 3D discrete space. *The Visual Computer*, 8:278–291, 1992.

[79] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[80] A. Opalach and S. Maddock. Speeding up grid-data generation for polygonisation of implicit surfaces. In *Proceedings of 13th Annual Conference of Eurographics (UK Chapter) (Loughborough, March 28-30, 1995)*, pages 153–160, March 1995.

[81] S. Muraki. Volumetric shape description of range data using "Blobby Model". In *Proc. SIGGRAPH '91 (Las Vegas, Nevada, July 28-August 2, 1991)*, volume 25(4), pages 227–235. ACM SIGGRAPH, New York, July 1991.

[82] S. Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, 13(4):50–56, July 1993.

[83] T. He, S. Wang, and A. Kaufman. Wavelet-based volume morphing. In *Proc. Visualization 94, (Washington, D.C. October 17-21, 1994*, pages 85–92. IEEE CS Press, Los Alamitos, Calif., 1994.

[84] J. F. Hughes. Scheduled fourier volume morphing. In *Proc. SIGGRAPH '92 (Chicago, Illinois, July 26-July 31, 1992)*, volume 26(2), pages 43–46. ACM SIGGRAPH, New York, July 1992.

[85] A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24-July 29, 1994)*, volume 28(2), pages 269–277. ACM SIGGRAPH, New York, July 1994.

[86] D. Terzopoulos, J. Platt, and K. Fleischer. Heating and melting deformable models (from goop to glop). In *Graphics Interface '89*, pages 219–225, 1989.

[87] T. Reed and B. Wyvill. Visual simulation of lightning. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24-July 29, 1994)*, volume 28(2), pages 359–364. ACM SIGGRAPH, New York, July 1994.

[88] T. T. Elvins. A survey of algorithms for volume visualization. *Computer Graphics*, 26(3):194–201, August 1992.

[89] M. Chen, M. W. Jones, and P. Townsend. Methods for volume metamorphosis. In *European Workshop on Combined Real and Synthetic Image Processing for Broadcast and Video Production (Hamburg, Germany)*, November 1994.

[90] A. Kaufman. An algorithm for 3D scan-conversion of polygons. *Proc. of Eurographics '87, Amsterdam, The Netherlands*, pages 197–208, August 1987.

[91] S. W. Wang and A. E. Kaufman. Volume sampled voxelization of geometric primitives. In *Proc. Visualization 93*, pages 78–84. IEEE CS Press, Los Alamitos, Calif., 1993.

[92] N. Max. Optical models for volume rendering. In *Visualization in Scientific Computing*, pages 35–40. Springer-Verlag, Wein New York, January 1995.

[93] H. Meinzer, K. Meetz, D. Scheppelmann, U. Engelmann, and H. J. Baur. The heidelberg ray tracing model. *IEEE Computer Graphics and Applications*, 11(6):34–43, November 1991.

[94] W. Krueger. The application of transport theory to visualization of 3D scalar data fields. In *Proc. Visualization 90*, pages 273–280. IEEE CS Press, Los Alamitos, Calif., 1990.

[95] A. Kaufman and R. Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics and Applications*, 8(6):10–23, November 1988.

[96] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes5: A heterogenous multiprocessor graphics system using processor-enhanced memories. In *Proc. SIGGRAPH '89 (Boston, Mass., July 31-August 4, 1989)*, volume 23(3), pages 79–88. ACM SIGGRAPH, New York, July 1989.

[97] D. Cohen and C. Gotsman. Photorealistic terrain imaging and flight simulation. *IEEE Computer Graphics and Applications*, 14(2):10–12, March 1994.

[98] J. Pineda. A parallel algorithm for polygon rasterization. In *Proc. SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988)*, volume 22(4), pages 17–20. ACM SIGGRAPH, New York, August 1988.

[99] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, January 1975.

[100] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, October 1977.

[101] K. R. Sloan and J. Painter. Pessimal guesses may be optimal: A counterintuitive search result. *IEEE Transactions on Patter Analysis and Machine Intelligence*, 10(6):949–955, November 1988.

[102] L. T. Cook, P. N. Cook, K. R. Lee, S. Batnitzky, B. Y. S. Wong, S. L. Fritz, J. Ophir, S. J. Dwyer, L. R. Bigongiari, and A. W. Templeton. An algorithm for volume estimation based on polyhedral approximation. *IEEE Transactions on Biomedical Engineering*, 27(9):493–499, September 1980.

[103] S. Ganapathy and T. G. Dennehy. A new general triangulation method for planar contours. In *Proc. SIGGRAPH '82 (Boston, Mass., July 26-30, 1982)*, volume 16(3), pages 69–75. ACM SIGGRAPH, New York, July 1982.

[104] L. T. Cook, S. J. Dwyer, S. Batnitzky, and K. R. Lee. A three-dimensional display system for diagnostic imaging applications. *IEEE Computer Graphics and Applications*, 3(5):13–19, August 1983.

[105] M. Shantz. Surface definition for branching contour-defined objects. *Computer Graphics*, 15(2):242–270, July 1981.

[106] H. N. Chrisiansen and T. W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. In *Proc. SIGGRAPH '78 (Atlanta, Georgia, August 23-25, 1978)*, volume 12(2), pages 187–192. ACM SIGGRAPH, New York, August 1978.

[107] J. Boissonnat. Shape reconstruction from planar cross sections. *Computer Vision, Graphics and Image Processing*, 44:1–29, 1988.

[108] A. B. Ekoule, F. C. Peyrin, and C. L. Odet. A triangulation algorithm from arbitrary shaped multiple planar contours. *ACM Transactions on Graphics*, 10(2):182–199, April 1991.

[109] C. Giersten, A. Halvorsen, and P. R. Flood. Graph-directed modelling from serial sections. *The Visual Computer*, 6:284–290, 1990.

[110] B. I. Soroka. Generalized cones from serial sections. *Computer Graphics and Image Processing*, 15:154–166, 1981.

[111] D. Myers, S. Skinner, and K. Sloan. Surfaces from contours. *ACM Transactions on Graphics*, 11(3):228–258, July 1992.

[112] J. Boissonnat. Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics*, 3(4):266–286, October 1984.

[113] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, January 1994.

[114] G. Turk and M. Levoy. Mesh optimization. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24-July 29, 1994)*, volume 28(2), pages 311–318. ACM SIGGRAPH, New York, July 1994.

[115] R. S. MacLeod, C. R. Johnson, and M. A. Matheson. Visualization of cardiac bioelectricity - a case study. In *Proc. Visualization 92*, pages 411–418. IEEE CS Press, Los Alamitos, Calif., 1992.

[116] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[117] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Transactions on Graphics*, 3(2):135–152, April 1984.

List of Figures

List of Tables