A Method of Rendering CSG-Type Solids Using a Hybrid of Conventional Rendering Methods and Ray Tracing Techniques.

by

Marion Scott Cottingham B. Sc. (Hons)

A thesis submitted to the

Faculty of Science

for the degree of

Doctor of Philosophy.

University of Glasgow.

September 1988.

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

ACKNOWLEDGEMENTS.

The work to be described was carried out in the Department of Computing Science, University of Glasgow. I would like to thank Dr. A.C. Kilgour and Dr. R.A. Sutherland for their continuing support and advice throughout the period of research and writing this thesis. I am particularly indebted to Dr. A.C. Kilgour for his helpful suggestions and constructive criticism.

I would also like to thank the British Science and Engineering Research Council for their financial support while carrying out the work involved.

I am most grateful to my Mother for taking over my household duties, without whose help this thesis would not have been possible. Finally I would like to thank my sons Steven and Alan for being very understanding and helpful.

CONTENTS.

Section	Description	page
Summary		8
Chapter 1 :	Modelling and Rendering Solids : An Overview.	9
1.1	Introduction.	10
1.2	Representations for Solid Modelling.	12
1.2.1	Alternative Representations of the Primitives.	16
1.2.1.1	Primitives Represented by Half-Spaces.	16
1.2.1.2	Polyhedral Approximation of Primitive Solids.	17
1.3	Rendering Techniques.	20
1.3.1	Hidden-Surface Removal Methods.	21
1.3.1.1	Scan-line Methods.	22
1.3.1.2	Ray Tracing Techniques.	24
1.3.1.3	Fast Access to Required Data.	26
1.3.2	Shading and Illumination.	27
1.3.2.1	Illumination Models.	27
1.3.2.2	Smooth Shading Techniques for Polyhedral Models.	31
1.3.2.3	Anti-Aliasing.	33
Chapter 2 :	Ray Tracing.	34
2.1	History	35
2.2	Using the CSG Representation.	40
2.3	Overview of Roth's Paper.	41
Chapter 3 :	Consideration of Roth's Method of Ray Tracing.	45
3.1	Improving the Efficiency of Roth's Method of Ray Tracing.	46
3.2	Problems in Building Ordered and Balanced CSG-trees.	51

Section	Description	page
Chapter 4 :	The Proposed Hybrid Method.	52
4.1	Additional Structures for Fast Access of Data.	55
4.1.1	Scene Trees: A New Structure for CSG-trees.	56
4.1.1.1	Input Sequences Containing Groups of Primitives.	59
4.1.2	Computing the Surface Definitions.	62
4.1.3	Calculation of Box Enclosures.	62
4.1.4	Indexing CSG-Tree Terminal Nodes.	63
4.1.5	Creating and Maintaining the Active Primitive List.	64
4.1.5.1	Path Through the Data Structure.	66
4.1.5.1.1	Creating a Path Through the CDS.	68
4.1.6	Creating and Maintaining the Span List.	69
4.1.6.1	The 'Polygon in Span' List.	70
4.2	Hidden-Surface Elimination.	72
4.2.1	Using Scan-line Methods.	73
4.2.2	Using Ray Tracing Techniques.	73
4.3	Algorithm for Proposed Hybrid Method.	75
4.4	Shading and Illumination.	78
Chapter 5 :	Presentation & Analysis of Results.	79
5.1	Comparison with Watkins' Spanning Scan-line Algorithm.	80
5.2	Comparison with Roth's Ray Tracing Techniques.	83
5.3	Comparison with Other Hybrid Approaches.	85
5.4	Performance Measurements.	87
5.4	Conclusions and Further Work.	89
References.		90

Section	Description	page
Appendix A :	The DIAMOND System.	102
1.1	Interacting with DIAMOND.	104
Appendix B :	A Compressed Data Structure.	107
Appendix C :	Compressed Data Structure for Rotational Sweep Method.	136
Appendix D :	Pseudo Ordering of CSG-Trees.	147

LIST OF FIGURES.

No.	Description	Following page.
fig. 1(a)	Special class of cell-decomposition called spatial enumeration representation (SE-Rep).	14
fig. 1(b)	CSG tree representation.	14
fig. 1(c)	Rotational Sweep.	15
fig. 1(d)	Translational Sweep	15
fig. 1(e)	Boundary representation (B-Rep).	15
fig. 1(f)	Isoluminance Contour Model	15
fig. 2	Winged-edge topology showing the various pointers associated with each edge.	18
fig. 3	Typical CSG primitive objects and their corresponding CDS.	19
fig. 4(a)	Wire-frame drawing of a cylinder with no hidden lines removed.	20
fig. 4(b) and (c)	Wire frame drawings of a cylinder with hidden lines removed.	20
fig. 5	Scan-line divided into spans according to intersection points.	23
fig. 6	Incident light and surface normal	28
fig. 7	Specular reflection.	29
fig. 8	Light rays reflecting from surface considered as a collection of small perfectly-reflecting surfaces.	30
fig. 9	Gouraud's 'smooth shading' technique.	31
fig. 10	Phong's 'smooth shading' technique.	31
fig. 11 (a)	Point by Point Shading.	35
fig. 11 (b)	Result of Point by Point Shading.	35
fig. 12	Pinhole Camera Model.	36
fig. 13	Sight rays being traced through the environment.	36
fig. 14	Components of light reaching the viewer from point A.	36
fig. 15	Classifying a ray.	40
fig. 16	Example of scene tree structure.	58

No.	Description	Following page.
fig. 17(a)	Drawing of vice.	58
fig. 17(b)	Binary tree structure representing the vice.	58
fig. 17(c)	Scene tree structure representing the vice.	58
fig. 18	Logical internal representation for input sequence.	61
fig. 19(a)	Template primitive with box enclosure.	62
fig. 19(b)	Deduced box enclosure in screen coordinates.	62
fig. 19(c)	Exact minimum box enclosure in screen coordinates.	62
fig. 20(a)	CSG-tree with index.	63
fig. 20(b)	CSG-tree with index ordered by y.	63
fig. 21(a)	Template primitive with box enclosure.	65
fig. 21(b)	Deduced box enclosure in screen coordinates.	65
fig. 21(c)	Exact minimum box enclosure in screen coordinates.	65
fig. 22	Active-list ordered by x for given scan-lines.	65
fig. 23	Active-list with associated path lists.	66
fig. 24(a)	Spans of a scan-line.	69
fig. 24(b)	Span list with associated 'polygon in span' sub-lists.	69
fig. 25(a)	Image with initially active polygons shaded.	77
fig. 25(b)	Image showing three areas and block 1.	77
fig. 26(a)	Initial state of underlying data structure.	77
fig. 26(b)	State of underlying data structure at a particular point.	77
Photograph 1	Wire-frame image of three cubes.	89
Photograph 2	Shaded image of three cubes with ray traced pixels coloured blue.	89
Photograph 3	Shaded image of three cubes.	89
Photograph 4	Wire-frame image of a sphere and cube.	89
Photograph 5	Shaded image of a sphere and cube.	89

SUMMARY.

This thesis describes a fast, efficient and innovative algorithm for producing shaded, still images of complex objects, built using constructive solid geometry (CSG) techniques. The algorithm uses a hybrid of conventional rendering methods and ray tracing techniques.

A description of existing modelling and rendering methods is given in chapters 1, 2 and 3, with emphasis on the data structures and rendering techniques selected for incorporation in the hybrid method.

Chapter 4 gives a general description of the hybrid method. This method processes data in the screen coordinate system and generates images in scan-line order. Scan lines are divided into spans (or segments) using the bounding rectangles of primitives calculated in screen coordinates. Conventional rendering methods and ray tracing techniques are used interchangeably along each scan-line. The method used is determined by the number of primitives associated with a particular span.

Conventional rendering methods are used when only one primitive is associated with a span, ray tracing techniques are used for hidden surface removal when two or more primitives are involved. In the latter case each pixel in the span is evaluated by accessing the polygon that is visible within each primitive associated with the span. The depth values (i. e. z-coordinates derived from the 3-dimensional definition) of the polygons involved are deduced for the pixel's position using linear interpolation. These values are used to determine the visible polygon.

The CSG tree is accessed from the bottom upwards via an ordered index that enables the 'visible' primitives on any particular scan-line to be efficiently located. Within each primitive an ordered path through the data structure provides the polygons potentially visible on a particular scan-line.

Lists of the active primitives and paths to potentially visible polygons are maintained throughout the rendering step and enable span coherence and scan-line coherence to be fully utilised.

The results of tests with a range of typical objects and scenes are provided in chapter 5. These results show that the hybrid algorithm is significantly faster than full ray tracing algorithms.

CHAPTER 1.

MODELLING AND RENDERING SOLIDS : AN OVERVIEW.

1.1. Introduction.

Computers are capable of producing astonishingly realistic pictures, which are often achieved by employing clever techniques that are equally astonishing in the amount of CPU time they require. The resemblance between state-of-the-art computer generated pictures and photographs of the real thing can be quite amazing. Most practical applications of computer graphics however do not require photograph-like images. This thesis deals with the problem of rendering mechanical parts in circumstances where speed of picture generation is more important than a high degree of realism.

There is no doubt that computer produced images are a valuable aid for communicating information. Such images can exploit the massive parallel processing capacity of human vision. The saying "A picture is worth a thousand words" has been found to have a strong element of truth by psychologists.

For decades industry has recognised that drawings are the most effective method of conveying geometric information between humans. It is a natural progression for computer graphics to provide an effective method of human-computer communication. Indeed for some applications that involve vast amounts of numerical data, computer generated images are the only reasonable way of communicating the information. The earliest computer graphics systems, developed in the 1960's, produced wire-frame drawings that were essentially the same as the traditional drawings done by draughtsmen etc. [Cottingham 1988a (Appendix D)]. These drawings were done on random-scan display devices that were limited in capability to plotting continuous lines and curves.

In the early 1970's raster-scan display devices were introduced enabling shading to be applied to images. These devices brought about a major change in the techniques used for describing objects. The aim now was to render shaded images rather than imitate the drawings previously done by draughtsmen.

The display screen of a raster-scan device is divided into a rectangular array of picture elements (pixels). When an image is generated, each of these pixels has an intensity value stored in a block of memory that is called a 'frame buffer'. Early versions of frame buffers typically used disks and drums for storage [Newman 1979]. This meant that raster-scan devices were rather expensive

because of the rotating memory required. By the early 1970's the decrease in cost of integrated-circuit shift registers made it more cost effective to use them in place of the rotating disks or drums. Unfortunately shift registers, disks and drums all require 'serial' access, each pixel being accessed in order once per revolution of a disk or drum, or once per cycle through the shift-registers. This resulted in an average delay of 1/50 or 1/60 of a second to change each pixel, so that even minor changes required several seconds. This was not acceptable for interactive applications.

Nowadays, random access integrated memory circuits are used for frame buffers. These overcome the latency problem and provide for faster interaction. Since the introduction of raster-scan devices there has been a continuously increasing ratio of processing power to cost. This has made them more and more attractive and accessible to a larger number of users. Their increase in popularity has brought about an increase in the availability of graphics software and demand for higher resolution display devices. Computer graphics is now well established and is still expanding into new applications as well as extending existing ones.

Today's raster devices also enable colour. Each element in the frame buffer is extended to contain the intensity value for each of the three primary colours (red, green and blue). These can be represented by a minimum of 8 bits; for example 3 bits are allocated for red, 3 for green and 2 for blue. There is no doubt that adding colour and shading greatly enhances realism in an image, enabling complex objects to be easily and unambiguously interpreted.

This thesis introduces a new method for producing acceptably realistic shaded images on a raster-scan display device. A hybrid of conventional rendering methods and ray tracing techniques is used to render unsculptured mechanical parts that are built using constructive solid geometry (CSG) methods (see section 1.2). Special attention is paid to efficiency, both in storage of the internal model and rendering algorithms used to generate the image. The main aim is to strike the correct balance between realism and processing time for engineering-type applications.

1.2. Representations for Solid Modelling.

In computer graphics the rendering process starts from an internal model. This internal model has to provide all that is necessary to describe a scene and for an image to be rendered. In a polyhedral representation, the model will contain the coordinates of vertices defining all the solids in the scene, and the topology showing how these vertices are connected (e. g. edges and faces). For a mathematical representation, it will contain some formulae for defining surface patches. For example, the surface of a sphere will be represented by a quadratic, and a Bézier surface patch will be represented by a parametric vector-valued function.

Geometric modelling is the term given to the actual construction of this internal model. There are currently three major types of internal model :

wire-frame model	:	an object is represented by a collection of lines.
surface model	:	an object is represented by a collection of surface elements.
solid model	:	a complex object is represented by some consistent model of true volume.

One of the main objectives when modelling a scene is realism (or precision). The degree of realism is influenced by both the complexity of the internal model and the rendering algorithms applied. In general, a high degree of realism requires a complex internal model and sophisticated rendering algorithms that require a lot of processing time to generate an image.

One method of reducing the drawing time is to build a few internal models of the same object, each at different levels of detail [Clark 1976]. Models with lower levels of detail may then be used for quick generation of the image for interactive viewing, or when less detail is required such as when the object is viewed from a distance. Models with higher levels of detail are used for high-

quality image generation or close-up viewing.

The most important of these internal models is the solid model. Solid modelling systems for CAD/CAM have been under development since the late 1960's and are now beginning to enter the commercial market in force. They are becoming increasingly popular in civil and mechanical engineering, architecture and other applications that utilise spatial features. A distinguishable feature of such systems is the unambiguous representation of solids [Requicha 1980, Requicha 1982]. Because solid modelling is so widely tried and tested, well understood and provides an unambiguous representation, it was decided to adopt this as the internal model for the work described in this thesis. There are six well-known representations :

primitive instancing :

The concept of families of object types is introduced (e.g. families of blocks, prisms etc.). An individual object within a family is called a primitive instance. These are defined by tuples of the form -

(**family, a b c**)

where **family** is a string identifying the 'type' and **a, b** and **c** are parameters providing position and size information. The number of parameters is dependent on the family of objects involved.

Because the number of different families is unrestricted, it is impossible to write general algorithms for computing properties of the solids represented. Each family must be treated as a special case. In particular, there is a lack of algorithms for combining instances of these primitives in order to build new and more complex objects.

cell decomposition :

Cells are represented by volume elements that have an arbitrary number of sides. These are sub-solids and are used to represent the volume as well as the surface. Cell decompositions are a generalisation of triangulations,

i. e. polyhedra may be triangulated into tetrahedra. However it is difficult for humans to decompose a curved solid.

spatial enumeration :

This is a special case of *cell-decomposition.* Space is partitioned by a 3-dimensional grid, into volume elements (voxels). Each voxel is a cube. Solids are represented by the list of voxels that they occupy (see fig. 1 (a)).

This representation is suitable for applications requiring box-like objects, and so is reasonably efficient in certain architectural applications. However, objects containing surface areas of high curvature require a tremendous amount of voxels, making this representation inefficient for representing most mechanical parts.

constructive solid geometry (CSG) :

Constructive Solid Geometry (CSG) methods represent complex solids by collections of simpler solids (or primitives). These are typically blocks, spheres, cylinders, cones and tori that are combined using the Boolean set operations union ' + ', difference ' - ' and intersection ' & '.

Internally these primitives and operands are stored as a tree structure, called a CSG tree. This is normally a binary tree that has the primitive solids stored at the external nodes (see fig. 1 (b)). The internal nodes contain the sub-solids resulting in the Boolean operands being applied to the two branches/primitives beneath them in the tree. The root node contains the complex solid being modelled. Certain classes of objects such as unsculptured mechanical parts can be easily created using the CSG representation.

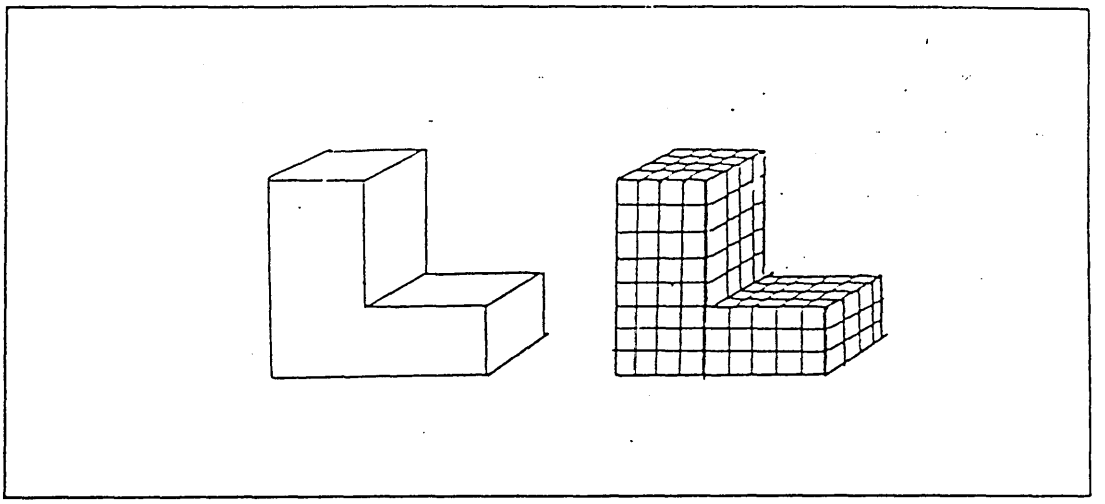
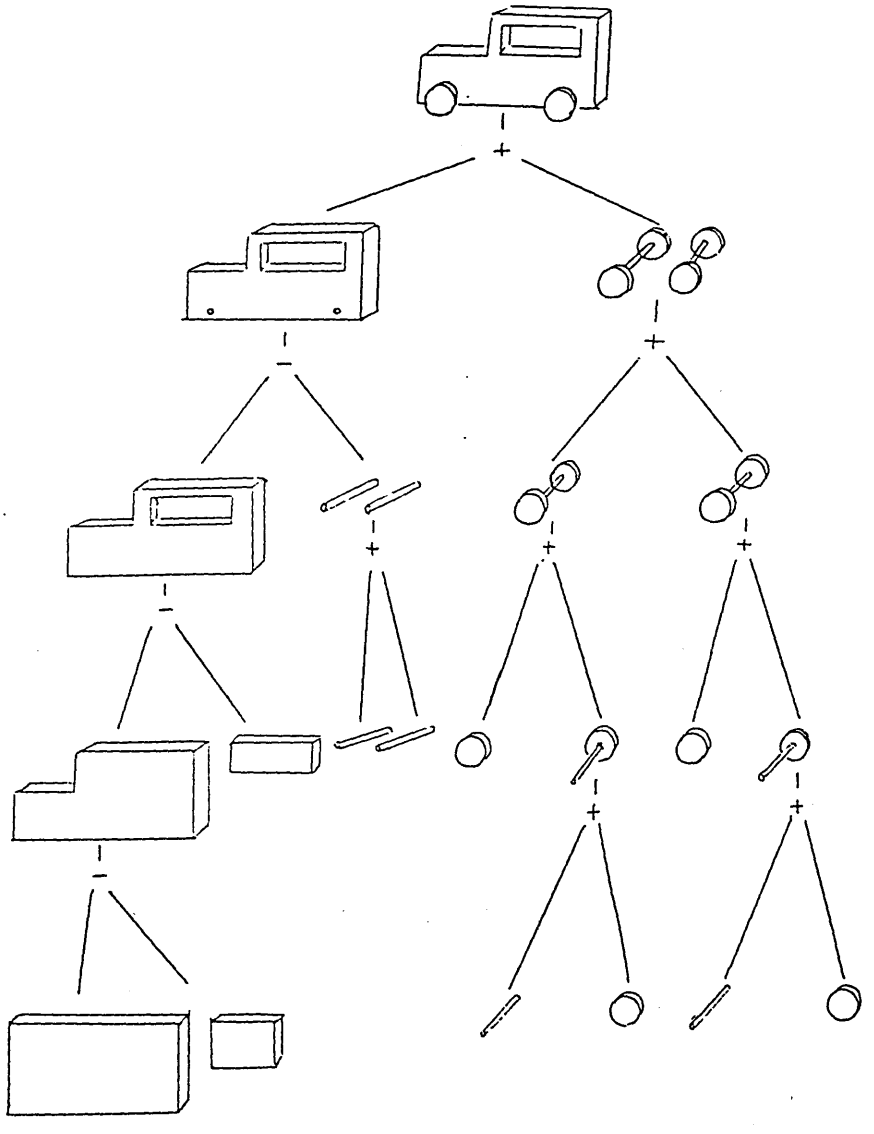


fig. 1(a) Special class of cell-decomposition called spatial enumeration representation (SE-Rep).

fig. 1(b) CSG tree representation.



sweep representations :

There are two types of sweep, rotational and translational. Rotational sweep is used for objects that have their symmetry preserved when rotated. The object is defined by a contour and an axis of rotation (see fig. 1 (c)).

Translational sweep is used for objects that are invariant in one direction. The object is defined by a contour and a trajectory (see fig. 1 (d)). The main disadvantage of this representation is the lack of algorithms available for computing the properties of the represented solids.

boundary representation (B–rep) :

The boundary, or surface of a solid, is represented as unions of facets, with each facet typically being defined by its bounding edges and vertices (see fig. 1 (e)). This representation is potentially capable of doing all that the CSG representation can do and more.

isoluminance contour representation :

The surface of a solid is divided into areas of constant intensity [Cottingham 1981, Conway 1988 (Appendix E)] (see fig 1(f)). At the rendering step, these areas are generated in depth order using the polygon-fill hardware that is available in most of the graphics devices currently available. It provides very realistic images which compare favourably with ray tracing (see section 1.3.1.2).

CSG and B-rep schemes are the best understood and currently the most important representation schemes for solids. There are some pure CSG systems such as SHAPES [Laning 1979] and earlier versions of SynthaVision [Goldstein 1979] and TIPS [Okino 1973]. These pure systems evolved towards having an additional surface representation for primitives. Dual representation is used in some CSG systems such as PADL [Voelcker 1978] , GMSolid [Boyse 1982] and later versions of Synthavision and TIPS.

The additional representation is derived from the CSG representation. For example, in GMSolid each type of primitive solid is represented by an exemplary

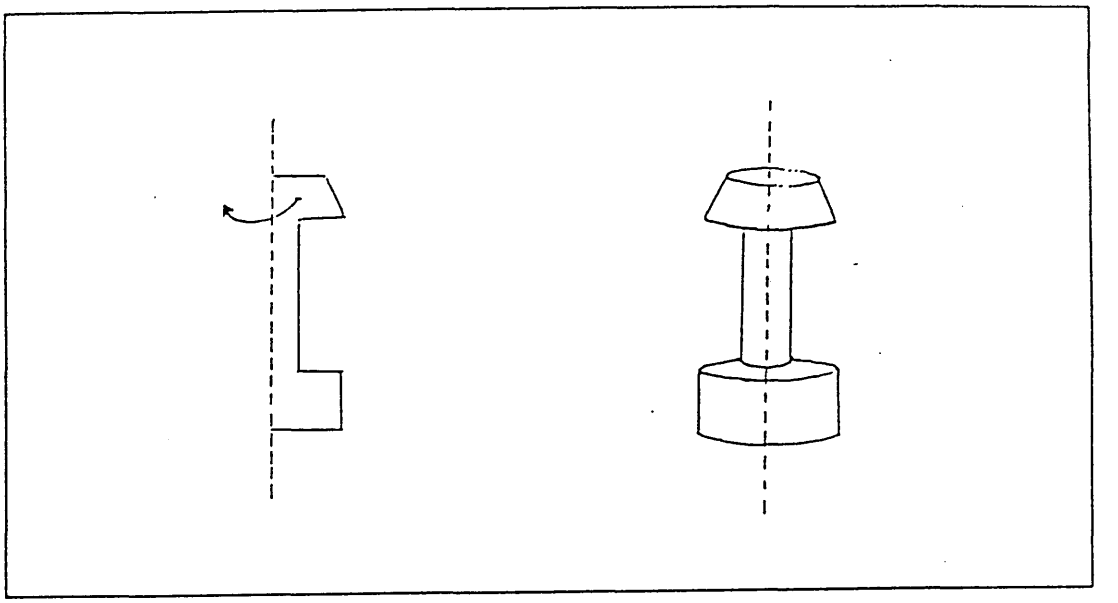


fig. 1(c) Rotational Sweep.

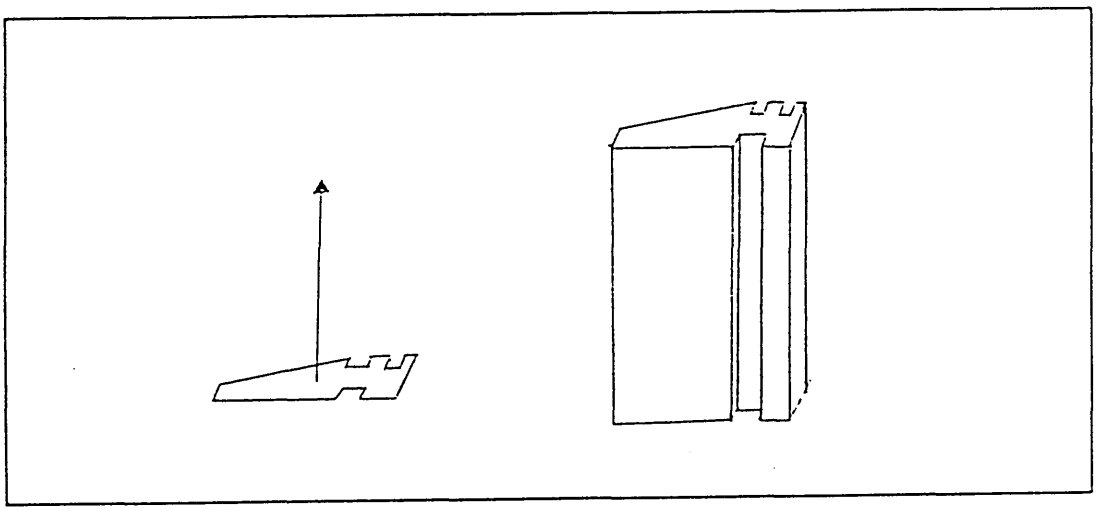


fig. 1(d) Translational Sweep.

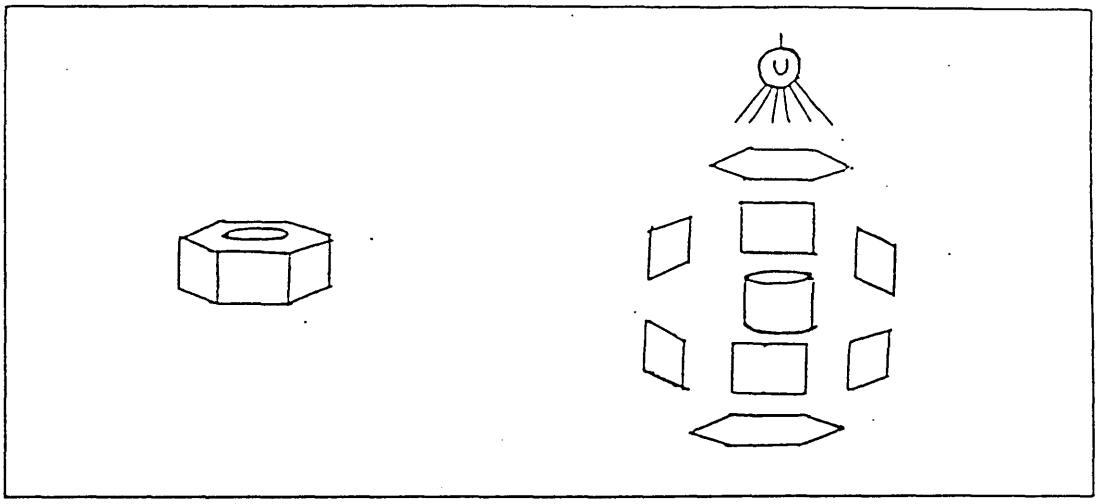


fig. 1(e) Boundary representation (B-Rep).

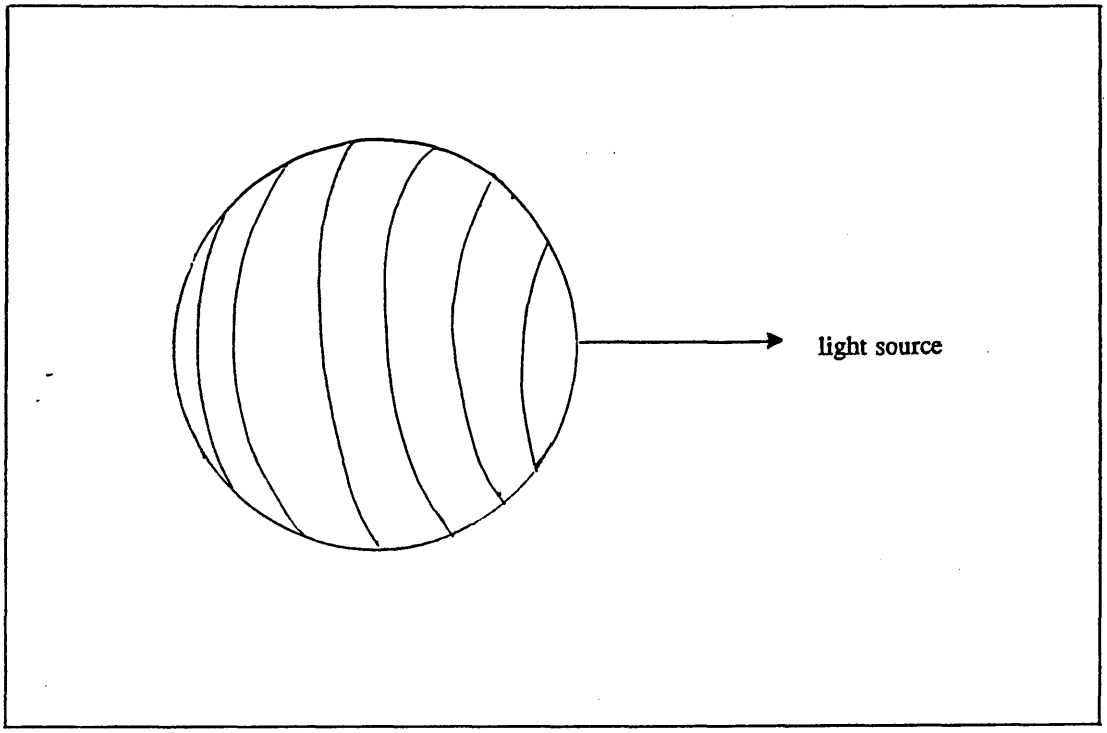


fig. 1(f) Isoluminance Contour Model.

surface definition and a transformation matrix [Boyse 1982]. These are defined in a 'local' coordinate space, with one feature placed at the origin (e. g. sphere's centre, cube's corner) and lengths in unit distance (e. g. sphere's radius, cylinder's height and width). When a solid is being constructed, the position and size of each primitive are provided and stored in a transformation matrix. This is used to transform the primitive type's surface definition from the object space to the scene coordinate space.

Nowadays, numerous commercial solid modelling systems exist [Requicha 1983]. It is often difficult to classify these systems into the simple categories above due to the lack of availability of technical descriptions of their internal organisations.

Because mechanical parts can be easily created from a small number of primitive types, the CSG representation was adopted for the present work. GMSolid's method of having primitive templates and transformation matrices are used to compute the surface definitions. This is less error prone than using B-rep since much less geometrical input is required.

1.2.1. Alternative Representations of the Primitives.

Many ways are available for representing CSG primitives. This section describes two that are commonly used in current CSG systems, viz half-spaces and polyhedral approximation.

1.2.1.1. Primitives Represented by Half-Spaces.

Halfspaces can be used as the simplest solids at the leaf nodes of CSG-trees (e. g. Shapes, TIPS). For example, a block is represented as the intersection of six planar halfspaces. A planar half-space is the infinite plane defined by the equation

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

plus all the points on one side of the plane. Using halfspaces can lead to invalid CSG representations, so checks must be made to ensure that compositions of solids remain bounded.

1.2.1.2. Polyhedral Approximation of Primitive Solids.

Polyhedral approximations can be used to represent the surface of the primitives at the leaf nodes of the CSG-tree. The primitive solid's surface is approximated by a collection of polygons. Vertices defining these polygons are normally defined in the Cartesian coordinate system with the x, y and z coordinates being sufficient to represent a 3-dimensional point.

This method of representation enables a smooth-shading technique (such as Gouraud or Phong, see section 1.3.2.2) to be used to restore the surface's smooth appearance. To obtain this in regions of high curvature the surface must be approximated by hundreds or thousands of facets. Therefore the amount of data storage required for this method of representation is usually very large, but this is becoming less pertinent as computer technology progresses. Because of the vast amount of data required, it is extremely important that the geometrical and topological information is stored in the most efficient and easily accessible manner. This information must include all that is required for an unambiguous representation of the solid(s) in question.

There are several different ways of representing and storing this information. Polygons could be defined by their vertices and could be stored as individual entities. The topology could be restored by comparing vertices of polygons; adjacent polygons would have at least one edge (i. e. two vertices) in common. However, using such a representation geometric information is duplicated, with a vertex common to four polygons being stored in four different places. Also finding two adjacent polygons could prove quite computationally expensive.

Two storage structures that avoid any duplication of data are the Winged-Edge data structure (see below) and the Compressed Data Structure (CDS)

[Cottingham 1985 (included as Appendix B), Cottingham 1987 (included as Appendix C)].

Baumgart's Winged-Edge Data Structure.

A well-known general structure for polyhedral representation is Baumgart's Winged-Edge data structure [Baumgart 1975]. This is used in the Design and in the Build-2 systems [Hillyard 1982]. It consists of four rings containing body, face, edge and vertex nodes. Each ring can be linearly traversed if required. The geometry is stored in the vertex nodes. The colour and intensity information is stored in the face nodes. The topology, showing the relationships between faces, edges and vertices, is stored in the edge nodes in the form of ten pointers. Figure 2 shows eight of these pointers, the two not shown are the next and previous edge nodes in the ring. The body node contains pointers to the 'first' and 'last' nodes in the vertex, edge and face rings.

The structure requires sixteen routines for node creation and manipulation and nine accessing routines, though this is considerably reduced in the recent generalisation of Guibas and Stolfi [Guibas 1985]. These routines enable the following access operations to be done efficiently -

- access all faces
- access all edges
- access all vertices
- access all edges belonging to a face
- access all edges meeting at a vertex
- access both faces meeting at an edge

It is a very powerful structure and can be used to represent virtually any polyhedron without duplication of data. However most of these access operations are not required when rendering an image. Because of its generality, it requires what seems an excessive amount of data to represent simple objects. For example, to represent a cube requires 1 body node, 6 face nodes, 8 vertex nodes, 12 edge nodes

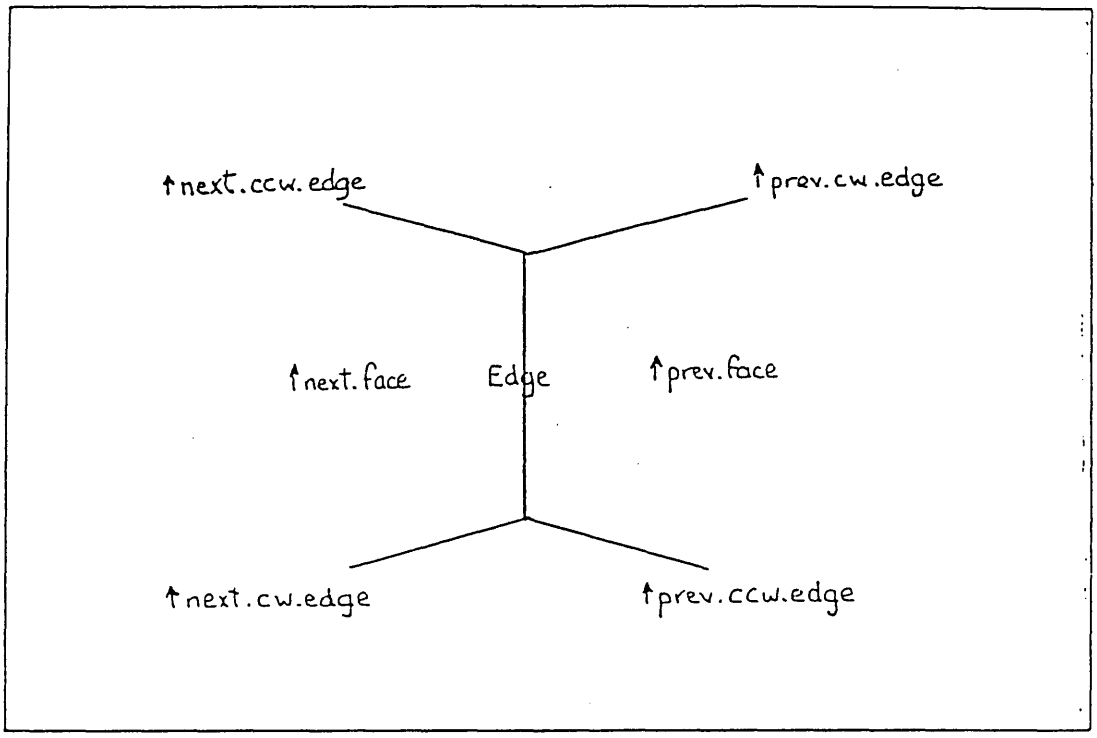


fig. 2 : Winged-Edge topology showing the various pointers associated with each edge.

and numerous pointers.

Vertices are the basic building blocks in approximating any surface by polygonal facets. Edges are defined by two vertices, facets are defined by edges and solids are defined by facets. The author has devised a simpler, more compact data structure which is less general than Baumgart's but is suitable for representing the CSG primitives. This is outlined below.

The Compressed Data Structure.

Using the Compressed Data Structure (CDS) only vertices are actually stored and the topology is deduced from the order of storage.

The CDS consists of an array with each element containing the coordinates of four vertices that define a facet. This limits the scope of polyhedral approximations to those in which exactly four edges impinge on each vertex, apart possibly from a small number of 'singularities', i. e. vertices with other than four edges.

Representing a block using the CDS requires only a two element array since there are only eight vertices in the cube. The type *block* implies that there are connections between the two facets, the order of storage of the vertices within an element enables the correct connections to be made.

The polyhedral representation of primitive solids is adopted for the hybrid system described in this thesis. Because of its simplicity the CDS is used for storing the vertex information of the polygons. Fig. 3 shows how the typical (CSG) primitives are stored using this data structure.

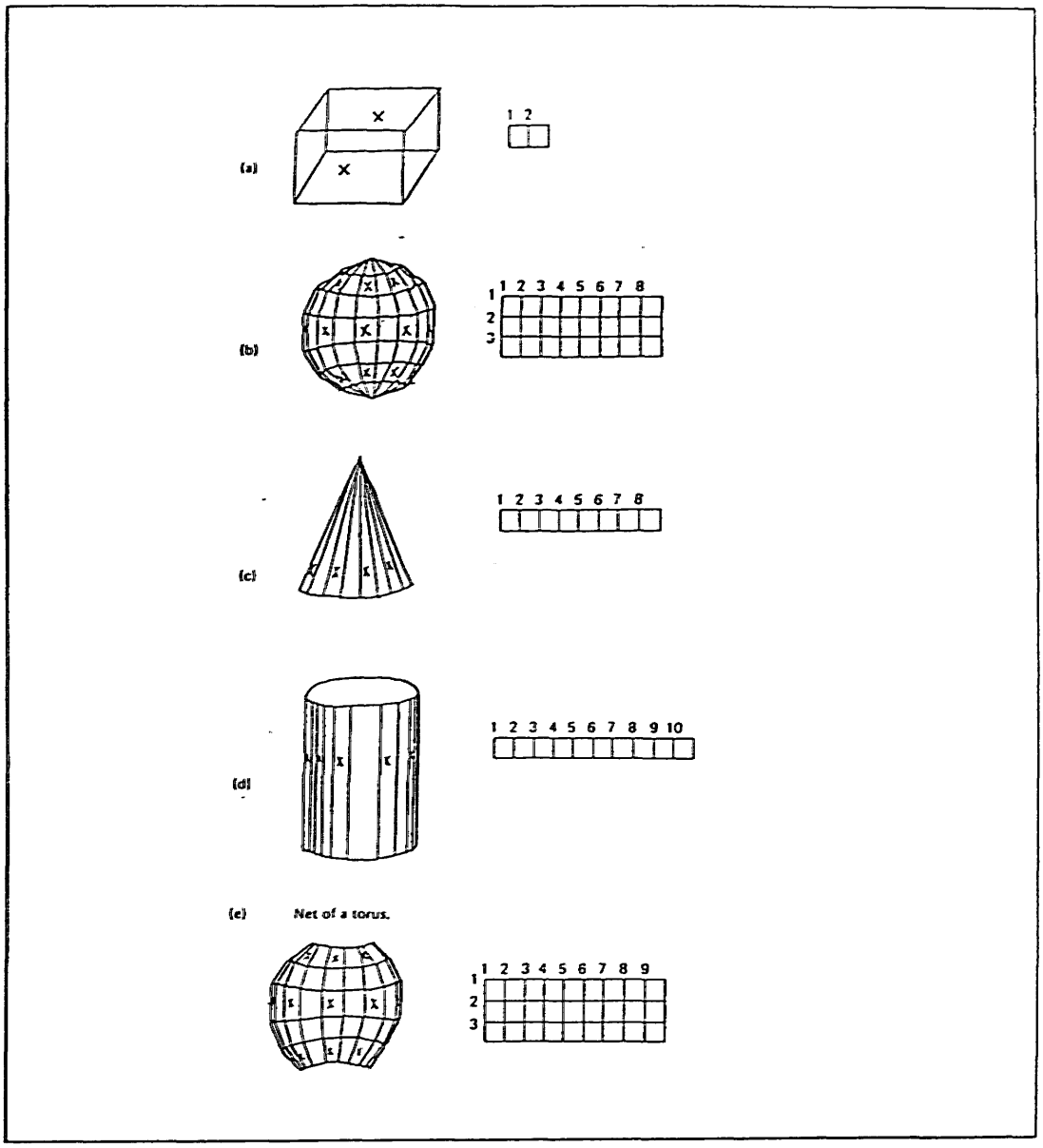


fig. 3 : Typical CSG Primitives and their corresponding CDS.

1.3. Rendering Techniques.

As computer graphics became established, research tended to concentrate on two major issues, namely improving realism and increasing performance. The two most common conventional techniques for achieving realism when generating a 3-dimensional scene are hidden-line/hidden-surface removal and shading. These techniques help to convey the relative depth and shape of solid objects in the scene and are discussed in sections 1.3.1 and 1.3.2.

Among the most efficient hidden surface methods are those based on a scan-line approach. These are discussed in the following section.

1.3.1. Hidden-Surface Removal Methods.

The problem of removing hidden-lines/hidden-surfaces is one of the more difficult tasks in computer graphics [Rogers 1985]. Hidden-line/hidden-surface removal is necessary to avoid any ambiguity or confusion. Figure 4 (a) shows a wire-frame drawing of a cylinder with no hidden-line/hidden-surface removal. This could be interpreted as 4 (b) or 4 (c).

There are many different hidden-line/hidden-surface algorithms in existence, some providing solutions for specialised applications. Algorithms that are designed for real-time rendering of images, e. g. flight simulation, are completely different from those that have been designed for producing photograph-like images, e. g. in computer animation, which must also include reflections, refractions, transparency and shadows, and often require several hours to produce an image.

In general, hidden-line algorithms work in the object-space where the objects are defined to determine the objects in the environment that are fully/partially visible. Hidden-surface algorithms often work in image-space where the objects are viewed (i. e. display screen) to determine what is visible at any particular position in the screen. The remainder of this section concentrates on hidden-surface methods since these are relevant to the application of generating the shaded images discussed in this thesis.

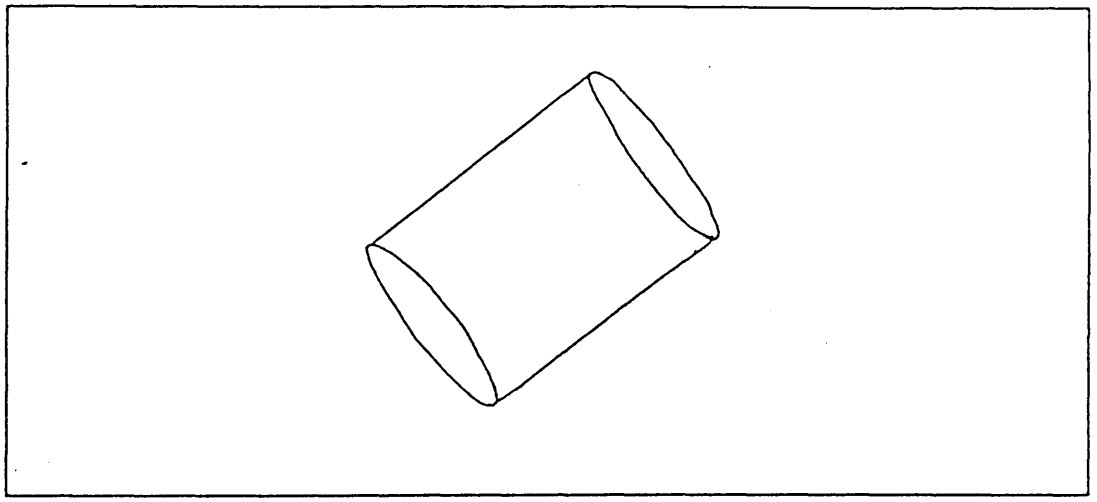


fig. 4(a) Wire-frame drawing of a cylinder with no hidden lines removed.

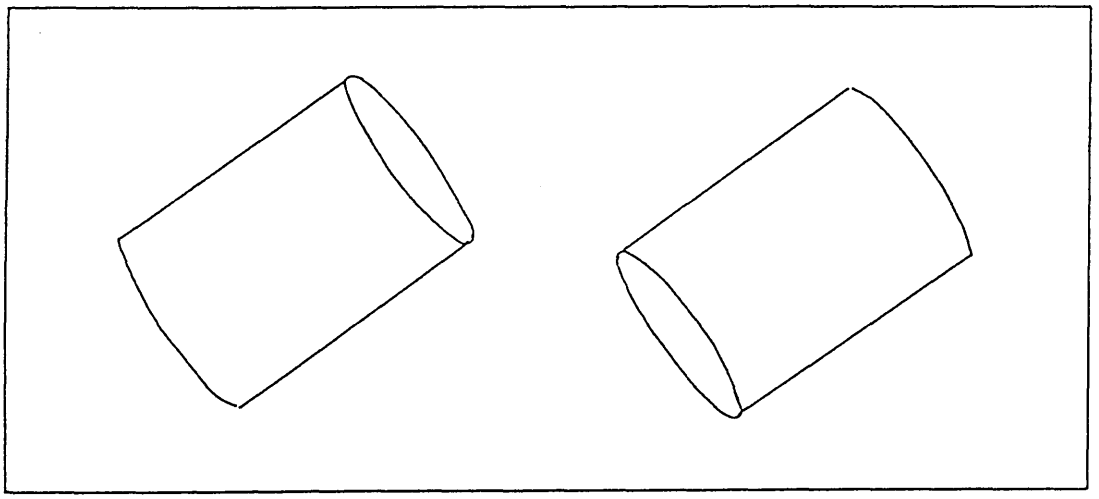


fig. 4(b) and (c) wire frame drawings of a cylinder with hidden lines removed.

Several hidden-surface algorithms are designed solely for raster-scan display devices. These algorithms concentrate on setting the intensity values at pixels in order to provide an approximation to a scene. In this section intensity will be used to mean both brightness and colour.

All hidden-surface algorithms involve sorting surface areas into some order [Sutherland 1974] to enable the visible surface, at a particular screen position, to be found and displayed. These algorithms tend to differ only in the sorting methods used and the order that geometric information is sorted into. Most of the algorithms require surfaces to be split into planar polygons. A few are designed to handle other representations such as parametric surface patches or mathematically defined objects.

Hidden surface methods often employ coherence properties to increase efficiency. There are three well-known coherence properties -

scan-line coherence	:	The properties that raster images on adjacent scan-lines are usually similar and adjacent pixels on the same scan-line are likely to have the same characteristics.
area coherence	:	The property that pixels lying within a small region of the display screen tend to be filled by the image of the same polygon.

span coherence : The property that sequences of pixels on a scan-line tend to be filled by the image of the same polygon. This is a 1-dimensional form of area coherence.

For efficiency, hidden-surface algorithms that generate the images of polygons by processing them one polygon at a time in an arbitrary order with respect to the screen's pixel order use area coherence to generate the pixels lying within their boundaries (e. g. Warnock's algorithm [Warnock 1969] and the z-buffer algorithm [Catmull 1975]). Hidden-surface algorithms that generate the images of polygons by processing them several at a time in scan-line order, called scan-line algorithms, use span coherence or scan-line coherence (or both) to generate adjacent pixels on each scan-line. The earliest such method was proposed by Wylie et al [Wylie 1967].

The hybrid method presented in this thesis uses the scan-line approach and processes several polygons at a time generating pixels in left-to-right order, one scan-line at a time, taking advantage of span coherence to minimise the work in moving from one pixel to the next and scan-line coherence to minimise the work in passing from one scan-line to the next.

1.3.1.1. Scan-line Methods.

There are several scan-line hidden surface algorithms available a well-known one is Watkins' spanning scan-line algorithm ([Watkins 1970, Newman 1973, Rogers 1985]). This algorithm generates pixels in scan-line order and works with vertices in screen coordinates taking advantage of scan-line coherence and span coherence methods. For efficiency the data is prepared by a pre-processing step as follows -

determine the maximum y-value for each polygon in the scene.
order polygons in decreasing maximum y-values.
for each polygon compute and store
- the number of scan-lines intersected by the polygon.
- a list of the polygon's edges.
- the four coefficients of the plane equation (ax + by + cz = d).
- the polygon's rendering attributes.

The following spanning scan-line algorithm then processes the prepared data -

for every scan-line *do*
begin
create/update active polygon list by examining ordered polygons;
create/update active edge list by examining active polygon list;
create/update span list maintaining increasing x order;
process active edge list;
end;

Each node in the active edge list contains the following information about each edge intersection -

1. the x coordinate at the intersection point.
2. the difference in x between intersection points on two adjacent scan-lines.
3. the number of scan-lines intersected by the edge.
4. an identifier for the polygon.
5. a flag to indicate if the polygon is active on current scan-line.

Processing the active edge list entails splitting the scan-line into segments (or spans) where the intersection points are located (see fig. 5). This reduces the hidden surface problem to the selection of the visible surface in each span of the scan-line. There are three kinds of spans possible -

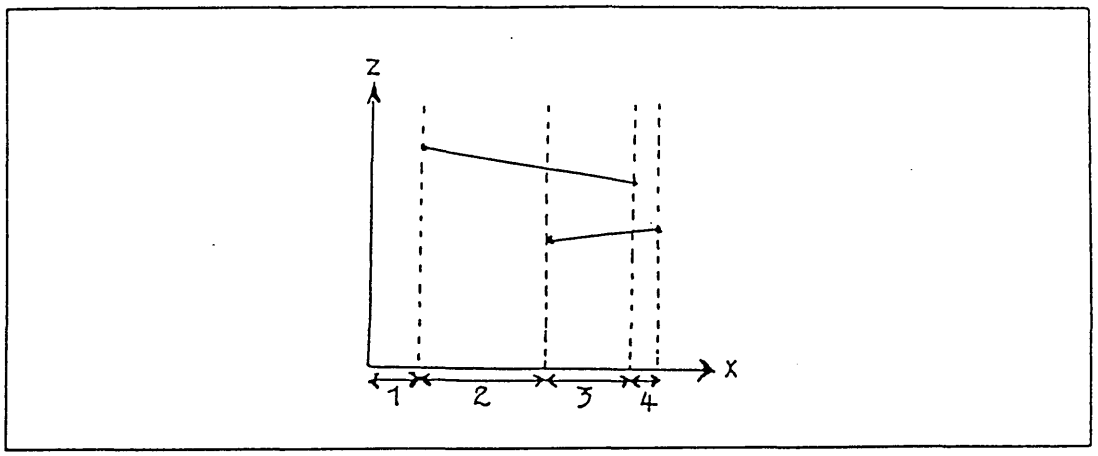


fig. 5 : Scan-line divided into spans according to intersection points.

1. empty span (span 1 of fig. 5).
2. span containing one surface (spans 2 and 4 of fig. 5).
3. span containing more than one surface (span 3 of fig. 5).

This algorithm takes advantage of the scan-line and span coherence properties that are described in section 1.3.1. Chapter 4 discusses how the algorithm's concepts are adapted for the hybrid method and used at both the primitive level and the polygon level.

1.3.1.2. Ray Tracing Techniques.

Ray tracing is to mathematically cast a ray from the viewpoint through every pixel in the screen. Hidden-surface removal is achieved by finding the first surface intersected by the ray. This provides the visible surface at the pixel corresponding to the ray [Atherton 1983]. There is no need to clip surfaces that partially/wholly lie outside the viewing window since no ray intersects such surfaces.

Without doubt, ray tracing (or casting) techniques are capable of producing the most realistic computer generated images currently attainable in computer graphics. Most of the problems that occur using conventional illumination models can be solved very easily. Accurate lighting models handling optical effects such as shadowing, reflections and refraction for transparent objects can be achieved in a simple and elegant way [Wijk 1984a]. For the current work ray tracing is used solely for hidden-surface elimination.

Ray tracing is relatively simple to program in comparison to other graphics algorithms [Kajiya 1983], but generally requires a large amount of computing time. This is mainly due to mapping the pixels onto the scene. The following ray tracing algorithm shows how this is done -

```
for every pixel in screen do
    begin
    cast ray through pixel;
    find first surface hit by ray;
    calculate intensity value;
    end
```

Computation time required can be kept to a minimum by using a simple illumination model whenever a high degree of realism is not required, or by using special hardware enabling parallel processing (see section 2.1). However the time required to find the nearest surface intersected could still be considerable.

The major part of any ray tracing algorithm is the ray-surface intersection calculation. This is normally done in one of two ways depending on the surface representation. If the surface is approximated by a collection of polygonal facets the ray-surface intersection test is done for every facet. The facet normals are used to distinguish the enter and exit points of the ray.

If the surface is defined analytically (e. g. quadric surfaces) points on the surface are computed directly by solving the equation for each ray, i. e. the roots supplying the intersection points. The simplest way to compute these points is to transform rays to local coordinate space before performing the ray-surface intersection tests. The alternative is to transform the equation.

After the ray-surface intersections have been found, the rays are decomposed into a collection of inside-the-solid and outside-the-solid segments [Myers 1982]. This is called classifying a ray. It can be determined from these segments the surface 'visible' at the associated screen pixel.

Ray tracing algorithms have been proposed for handling a wide class of surface representations. These include polyhedra and simple analytic surfaces [Roth 1982], general algebraic surfaces [Blinn 1982, Hanrahan 1983], parametric surfaces such as rational bivariate polynomials [Kajiya 1982] and Steiner patches [Sederberg 1984], surfaces with superimposed density distribution [Kajiya 1984], and biquadratic patches [Steinberg 1984].

It has been recognised that ray tracing is inefficient because it does not take advantage of any coherence properties [Atherton 1983]. This thesis introduces a

method of incorporating scan-line coherence and span coherence (discussed in chapter 4) into ray tracing. To enable this the order of pixel generation is left-to-right in scan-line order.

1.3.1.3. Fast Access to Required Data.

Finding the visible polygons from a large number of polygons can be very time consuming. Several ways of finding these have been devised to increase efficiency. One of these involves the partitioning of space [Woodwark 1982]. A model is recursively divided into sub-models according to spatial position until each sub-model reaches a certain level of simplicity. The processing starts with the sub-models positioned near the viewplane. The results are stored in a quad tree and provide for hidden-surface elimination; if all the quads corresponding to the front face of a sub-model have already been generated then that sub-model can be discarded without any further computation.

Another method uses a spatial index to link data to partitions [Tamminen 1984]. Tamminen converts a complex polyhedron into an octree-like block model and sequentially retrieves the blocks intersected by a ray. These blocks are ordered by distance to efficiently search for the visible face by location.

The current work introduces another method that operates in image space and provides access to data at two different levels. At the higher level, bounding rectangles are used for locating primitives that may be visible. At the lower level, paths through the data structure indicate polygons within each primitive that are visible. If only one primitive is associated with an area the front-facing polygons are generated. For areas associated with more than one primitive, ray tracing is used to find the polygon visible at a particular point. This method is discussed in more detail in chapter 4.

1.3.2. Shading and Illumination.

After the hidden-surface algorithm has found a visible surface, the intensity value has to be computed. The characteristics of the surface, its position and orientation to the viewpoint and the light source(s) must be considered. The intensity value will, according to the physics of light, have several different components. These are dealt with in the following sub-sections.

1.3.2.1. Illumination Models.

Diffuse Reflection and **Ambient Light.**

Every object in a scene reflects light depending on the reflectance property of its surface. Extremely dull surfaces will scatter light equally in all directions irrespective of the viewing direction. This is called diffuse reflection [Foley 1982].

The amount of light reflected from such surfaces can be computed using Lambert's cosine law:

$$I_d = I_p k_d \cos(\theta) \qquad \text{(a)}$$

where I_d is the intensity of the diffuse illumination, I_p is the intensity of the point light source, k_d is the level of dullness in the range 0 to 1, and θ is the angle between the light direction $\mathbf{L}$ and the surface normal $\mathbf{N}$. $\cos(\theta)$ can also be written as -

$$\mathbf{L} \cdot \mathbf{N}$$

where $\mathbf{L}$ and $\mathbf{N}$ have been normalised (see fig. 6).

In most real environments, parts of objects lying in the shade are not totally unlit. This is owing to ambient light which is propagated by multiple reflections of light from the many surfaces present in the environment. Adding the approximate ambient light term $I_a k_a$ to equation (a) gives :

$$I = I_a k_a + I_p k_d (\mathbf{L} \cdot \mathbf{N}) \qquad (b)$$

where I_a is the intensity of the ambient light and k_a is the amount of ambient light reflected from the object's surface.

Suppose two objects of the same colour and dullness factor overlap on the screen, and that they have the same surface normal. Using equation (b) they would both be given the same shading and would be indistinguishable from each other, therefore distance must also be considered in the illumination model. The energy of light decreases as the inverse square of the distance that the light travels from its source to the object's surface and onto the viewpoint. If D is this distance, then equation (b) is updated to :

$$I = I_a k_a + I_p k_d (\mathbf{L} \cdot \mathbf{N}) / D^2 \qquad (c)$$

However, this does not provide realistic shading when the viewpoint is very far away or very near to a surface. When the light source is an infinite distance away, solids appear to be evenly shaded with ambient light. When it is very near, surfaces sharing the same angle θ can have considerably different shades. These differences in shade can be overcome by replacing D^2 by $D + k$, where D is the distance between the perspective viewpoint and the surface, and k is a constant. This changes equation (c) to :

$$I = I_a k_a + I_p k_d (\mathbf{L} \cdot \mathbf{N}) / (D+k) \qquad (d)$$

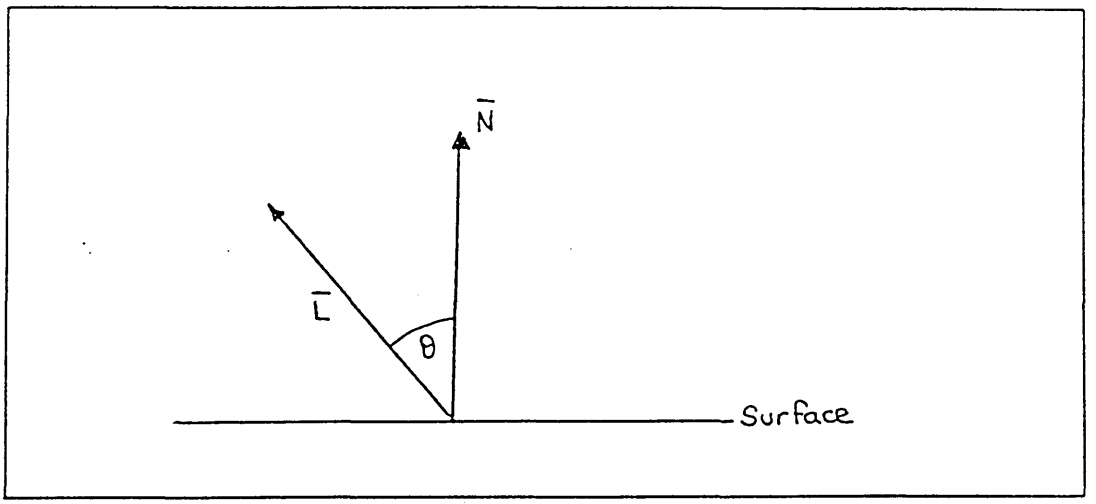


fig. 6 : Incident light $\overline{L}$ and surface normal $\overline{N}$.

If a coloured image is required then equation (d) must be used to compute the amount of diffuse reflection for each of the colour components of the intensity. For an RGB video monitor the colour components are red, green and blue. The parameters for intensity and reflectivity then become three-element vectors, one for each colour component. The amount of ambient light is uniform for any colour, since it is always the same colour as the point source.

Specular Reflection.

Specular reflection occurs on any shiny surface. An area of high specular reflection is often called a highlight. The colour of the highlight is always the same colour as the light source, or if reflected from another surface the colour of the incident light.

Specular reflection occurs only when the viewing direction is roughly coincident with the direction of reflection $\overline{R}$ (see fig. 7), (i. e. angle $\alpha = 0$). For surfaces that are perfect reflectors, such as mirrors, the viewing direction must be *exactly* coincident with the reflection direction. Most surfaces are not perfect reflectors: for them the intensity of the reflected light is at its highest level whenever the two directions are exactly coincident. This intensity decreases rapidly as the angle α increases. Phong Bui-Tuong approximates this rapid decrease by $\cos^n\alpha$, where typically $1 \leq n \leq 200$ depending on the surface [Bui-Tuong 1975, Foley 1982]. For perfect reflectors n would be infinite. $\cos^n\alpha$ provides a reasonable approximation for specular reflection but is based entirely on experimental observation.

The amount of light reflected depends on the angle of incidence θ. Let $\Omega(\theta)$ be the fraction of incident light that is specularly reflected, then equation (d) is written :

$$I = I_a k_a + I_p k_d(\overline{L} \cdot \overline{N})/(D+k) + I_p \Omega(\theta)\cos^n\alpha/(D+k) \qquad (e)$$

i. e. I = ambient + diffuse + specular.

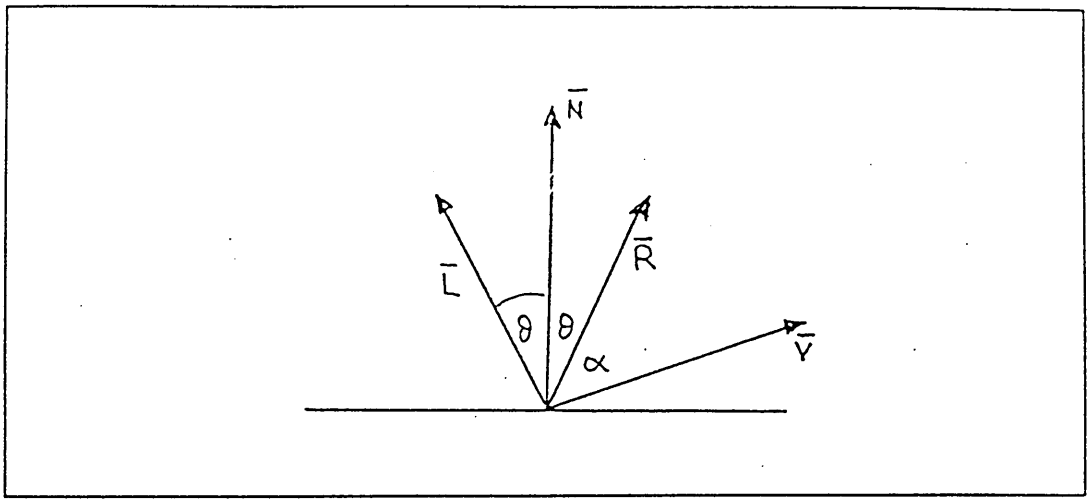


fig. 7 : Specular reflection.

Torrance and Sparrow present a theoretically-based model for reflected light [Torrance 1967]. This provides very realistic results and was used by Blinn [Blinn 1977] and Cook and Torrance [Cook 1982]. Using this model the surface is considered to be a collection of very small facets, each one being a perfect reflector (see fig. 8). Only facets that reflect light towards the viewer are considered (i. e. a, c, e, g and i). The geometry of the facets and the light direction determine the intensity and the direction of the specular reflection. Experiments have shown that there is a strong correlation between the actual reflection and the reflection provided by this model.

Shading algorithms aimed at realism must be able to simulate reflectance, specular reflectance, shadowing and diffuse illumination. Some applications also require simulation of surface texture and transparency.

Rendering mechanical parts does not require a high degree of realism, so texture and transparency are not considered further here. It is easily seen by comparing equations (d) and (e) that allowing for specular reflection greatly increases the workload and for this reason it will be omitted too. For this particular application it is assumed that the viewpoint will never be positioned very near or very far from the solid being modelled, therefore distance will also be ignored (compare equation (b) with (c) and (d)). Because parts of the solids are likely to lie in shadow, ambient light cannot be ignored. Therefore equation (b) is adopted to provide the minimum amount of realism required. The contribution of light reflected off other objects in the environment is ignored to avoid having to build an intersection tree (see section 2.1).

Applying even the simplest of these illumination models (see equation (a)) requires the calculation of an angle plus two multiplications. Smooth shading techniques (i. e. Gouraud shading) have been devised to minimise the cost of illumination calculations for surfaces that are approximated by polyhedra. Only points lying at the vertices have their intensity values calculated by the illumination model. The other intensity values are deduced using linear interpolation. This is much faster than numerous calls to the illumination model. These techniques are described in the next section.

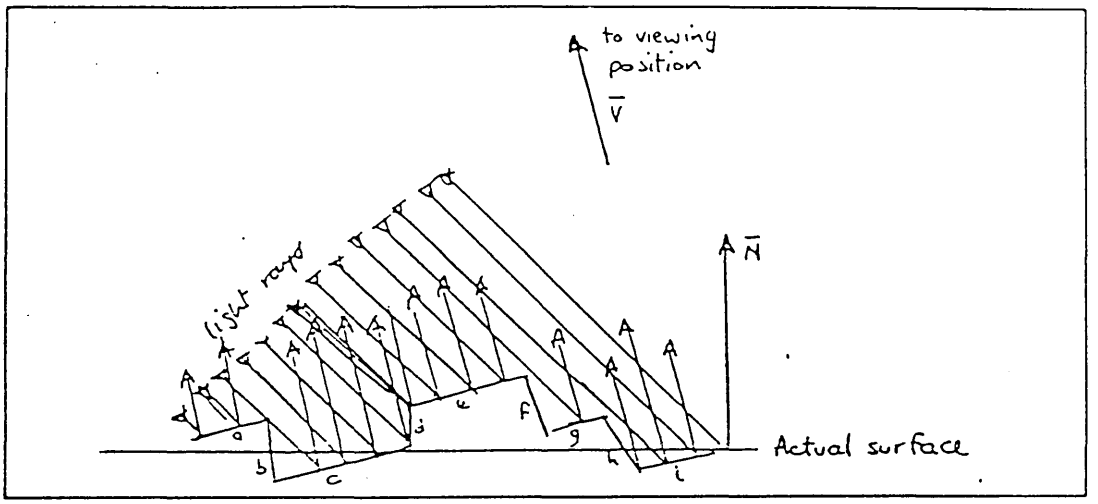


fig. 8 : Light rays reflecting from surface considered as a collection of small perfectly-reflecting surfaces.

1.3.2.2. Smooth Shading Techniques for Polyhedral Models.

Smooth shading techniques can be adopted to generate the images of objects that have their surfaces represented by planar polygons. These techniques use the illumination values at each vertex to compute the shades at pixels contained inside the polygon. The simplest of these techniques was introduced by Gouraud and used linear interpolation of colour and intensity values to eliminate intensity discontinuities and thus generate the appearance of smoothly curved surfaces [Gouraud 1971].

Using this technique surface normals are calculated for each polygon's surface. The normals at each vertex are computed by taking the average of the surface normals of all the polygonal facets that contain the vertex. These are used, with the desired illumination model, to compute the colour and intensity values at these vertices. The values at points lying inside the polygon and along the edges are deduced using linear interpolation.

In figure 9, the shading intensity at point L is determined by interpolating linearly between intensities at A and C. Similarly the intensity at R is determined by intensities at A and F. The intensity at point P can then be determined by interpolating linearly between intensities at L and R.

If colour is required the intensity value for each of the three primaries is calculated. The interpolation is done on each of these intensities individually.

However, this technique can tend to give a 'buzzing bees' effect when used for an animated sequence. This effect can make the surface appear to develop some form of moving microscopic life. The reason for this phenomenon is that the interpolation basis is fixed to the screen's surface rather than to points on the object's surface. This affects highlights too, with the shapes of highlighted areas being strongly influenced by the shape of the polygon rather than the surface orientation. Highlights lying within polygons, not incident on any vertex may be omitted altogether.

A smooth shading technique introduced by Phong tried to eliminate these problems by interpolating 'surface' normal vectors instead of shading intensities (see fig. 10) [Newman 1979]. A reflection model is applied to determine the colour and intensity values at each pixel as a function of the direction of the surface

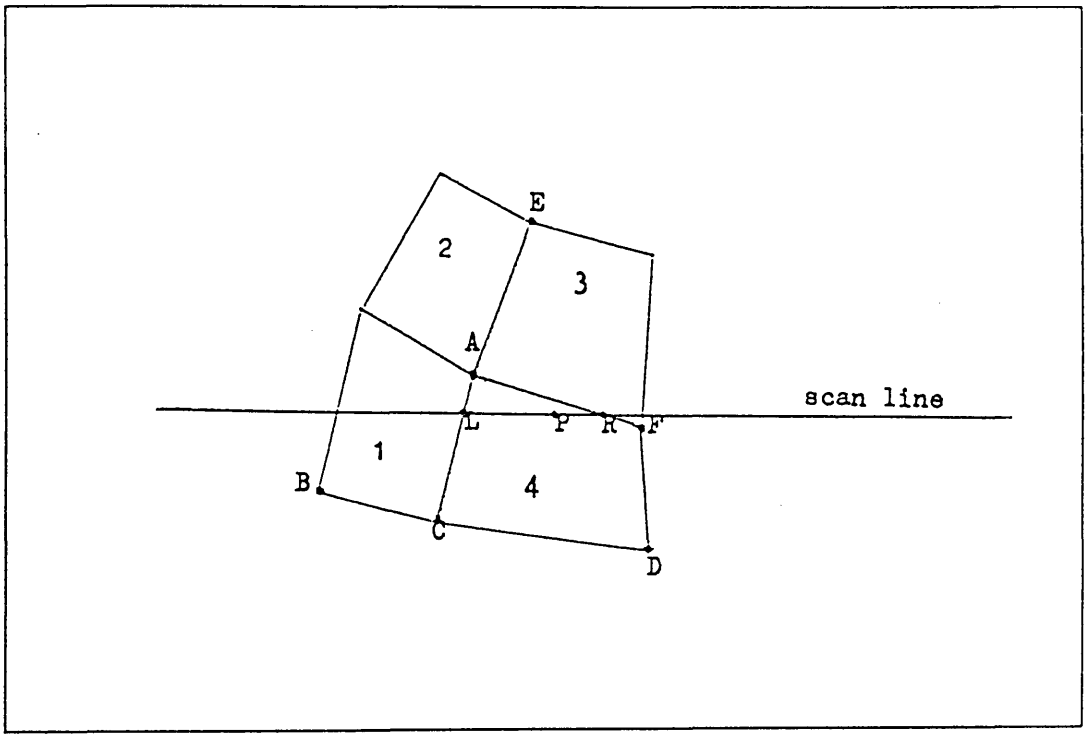


fig. 9 : Gouraud's 'smooth shading' technique.

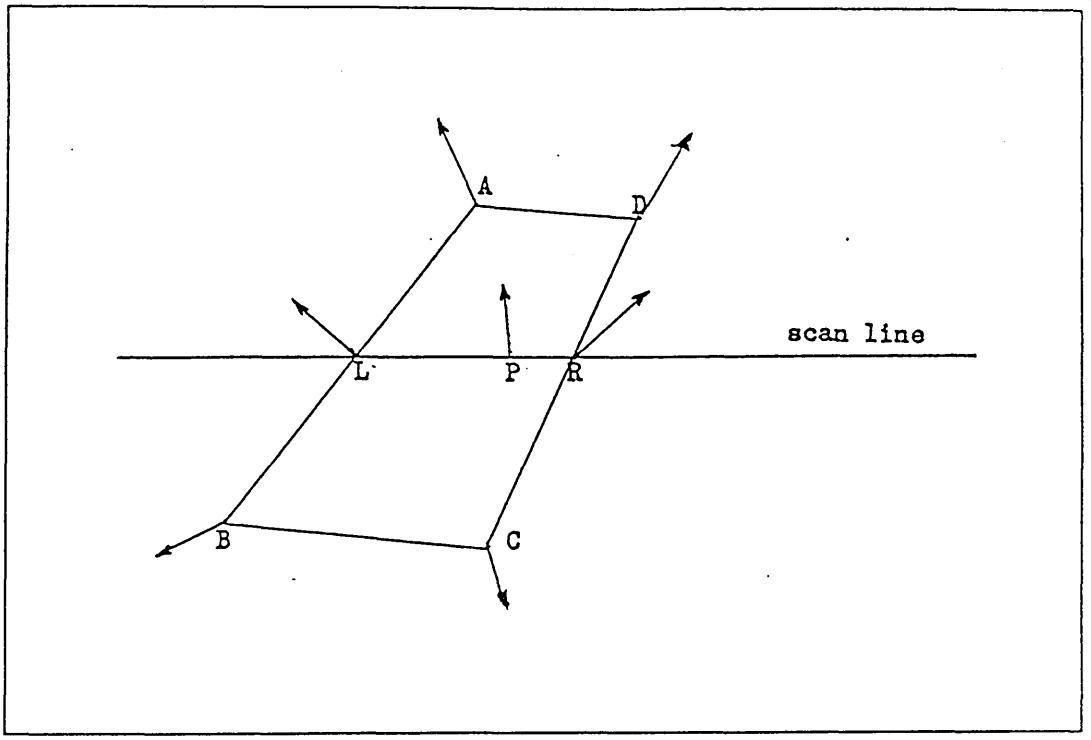


fig. 10 : Phong's 'smooth shading' technique.

normal. As with Gouraud's method the surface and vertex normals are computed. The vertex normals are used to interpolate the normals linearly between points lying inside the polygon and along the edges. Phong hoped to avoid the 'buzzing bees' effect by fixing the interpolation basis to the object's surface, however in certain situations this effect can be worse than when using Gouraud's method [Duff 1983]. Rotating an object and its light source in the image plane can cause unexpected results using either Gouraud or Phong shading. The shapes of the highlighted areas are more realistic using Phong's rather than Gouraud's method.

Phong's smooth shading technique has played an important part in the evolution of realistic image synthesis methods [Hall 1983]. The main disadvantage of using Phong's technique is the extra computation time involved in computing the three normal components at each point and in calling the shading model much more frequently than when using Gouraud's techniques.

Using any smooth shading technique, care must be taken not to smooth shade 'true' edges. For example, computing the normal of a cube's vertex by averaging the normals at the surfaces of all polygons containing that vertex, will yield spurious results. A 'true' edge must therefore be treated as a special case and the vertex given the same normal as the polygon being processed.

When rendering mechanical parts, the image rendered will be a 'still' image rather than a frame in an animated sequence, therefore the 'buzzing bees' effect produced by Gouraud's techniques is never envisaged. Specular reflection is being ignored so no highlighted areas will exist which was another disadvantage of Gouraud's method. Therefore Gouraud's techniques were adapted for efficiency in the new method.

1.3.2.3. Anti-Aliasing.

Aliasing is a common problem with images generated on raster display devices. The appearance of aliasing effects is due to attempting to map continuous lines or edges onto a raster-scan device which is discrete. It is an undesirable effect of point sampling techniques. Slithers of objects that are smaller than the space between adjacent sampling points may be overlooked. Linear or smoothly curved edges may appear jagged, giving a staircase effect.

The main problem with anti-aliasing for ray-tracing is that there is not enough information associated with each pixel [Fujimoto 1986]. The ray-object intersection point enables the sampling of only one point in the centre of the pixel.

There are many approaches to reducing the effects of aliasing [Crow 1981]. These follow two fundamental techniques. The first is to increase the sampling rate to a higher resolution than the display device; averaging or filtering methods are used to reduce these samples before displaying. The second approach is to treat a pixel as a finite area rather than a point, which is also equivalent to prefiltering the image.

The first approach is used for ray-tracing which is normally done using adaptive subdivision of pixels lying near large changes of intensity or near small objects.

For the purposes of the present work, the aliasing problem is disregarded.

CHAPTER 2.

RAY TRACING.

2.1. History.

Ray tracing, as applied to computer graphics, was introduced by Appel in 1967 [Appel 1967]. His paper described a scheme for the determination of visibility in a wire-frame image. A year later [Appel 1968] another paper was published describing experiments done in the automatic shading of line drawings, using a line-drawing display device. The possibility was recognised that 'machine generated photographs' might replace line drawings as the principal method of displaying information in engineering and architecture applications. However the techniques used for generating such images would have to be competitive with those currently in use for line drawings.

One of the many techniques Appel tested was ray tracing. His technique for point by point shading of an object (see fig. 11 (a) and (b)) was as follows-

1. Calculate minimum bounding boxes for the object in picture plane (screen) coordinates.

2. Generate raster of spots within this box, and compute the equation of the lines between each spot and the viewing position.

3. Project these lines into the scene and determine the first intersection point.

4. Determine the intensity value for this point and mark with appropriate symbol (e. g. ' . ', ' + ', ' # ', or ' * ').

However this method was found to require several thousand times more calculation time than conventional wire-frame drawing.

As with other rendering techniques different methods of tracing rays have evolved. These tend to differ in the direction that the rays are traced, and whether the viewing position lies in front or behind the picture plane.

In 1971 Goldstein and Nagel [Goldstein 1971] used ray tracing in an attempt to simulate the physical process of vision (i. e. how the eye's retina reacts to light being reflected from a scene). This was implemented on a raster-scan display device.

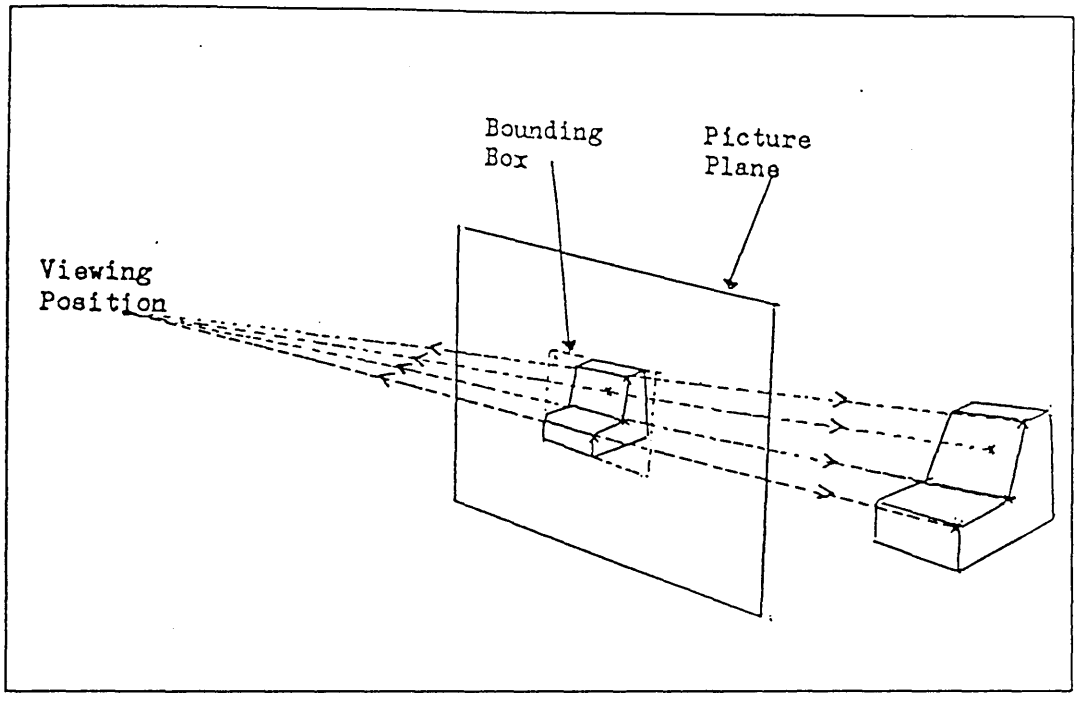


fig. 11(a) : Point by Point Shading.

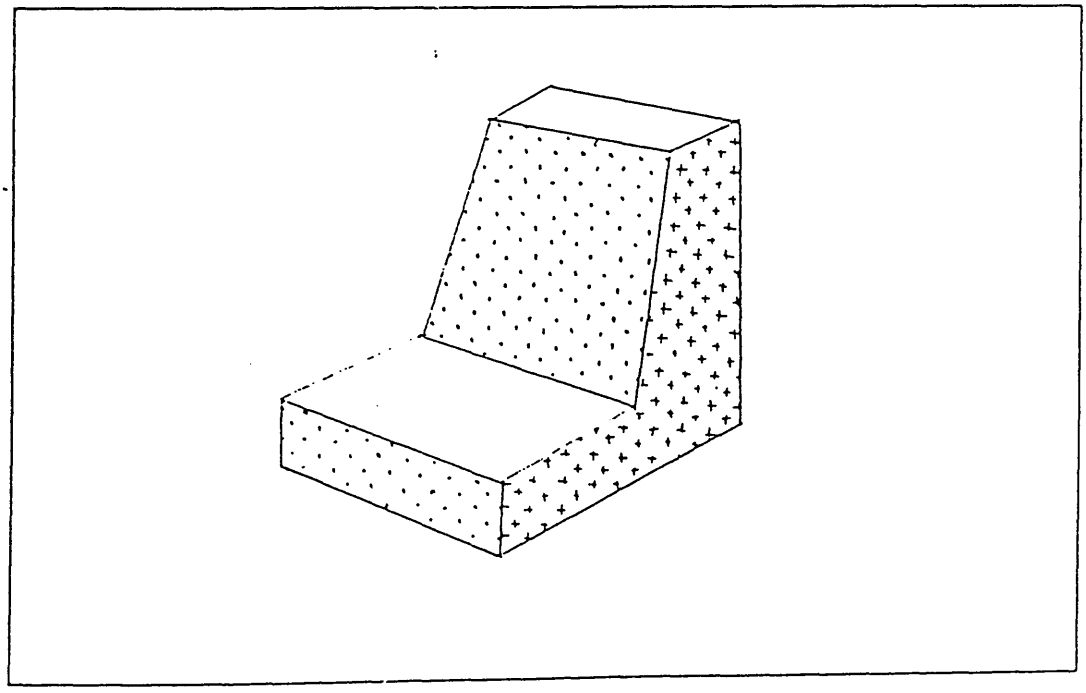


fig. 11(b) : Result of Point by Point Shading.

A pinhole camera was used for the model on which to base their visual simulation technique. The camera's film plane was analogous to the display device's screen and the pinhole to the focal point (see fig. 12). The film plane was subdivided into a two dimensional array of picture elements (pixels). The size of this array depended on the resolution required by the final results. The method was to cast a ray, from each pixel, through the focal point and into the object space. Ray-surface intersection tests were made to determine the first surface hit by the ray. The intensity value was then calculated by tracing a ray from this point to the light source.

It was noted that rays could also be traced in the opposite direction, starting at the light source, but in practice this approach would take longer to compute. Only a few of these rays would actually pass through the image plane and onto the viewer. An argument against this hypothesis is that if light rays were traced, and the light source was located at a different position from the view point, then the surface areas of the objects which were in shadow would never be hit by rays. Therefore such parts would never be drawn, being replaced by background colour/intensity in the final image rather than being lit by diffuse illumination.

This visual simulation technique introduced by Goldstein and Nagel was the impetus for most of the research into ray tracing techniques that followed. The pinhole camera model became the standard model in image processing [Roth 1982].

In 1979 Kay [Kay 1979] added further realism to the ray tracing modeller by attempting to solve the global illumination problem. It was recognised that colour and intensity at any point on a surface was partially dependent on the location, reflectance and additional characteristics of the other objects in the surrounding environment. Therefore a true image could only be generated by tracing each ray through the environment and computing the cumulative results (see fig. 13).

Rays were traced from the viewing position through each pixel and on through the scene. When a ray intersected an object the reflected ray was calculated. A hypothetical viewing position was then created at the intersection point and the viewing direction was made equivalent to the reflection direction (see fig. 14).

This process was recursively repeated. Accumulating these results was achieved using an intersection tree for each ray. This intersection tree was a

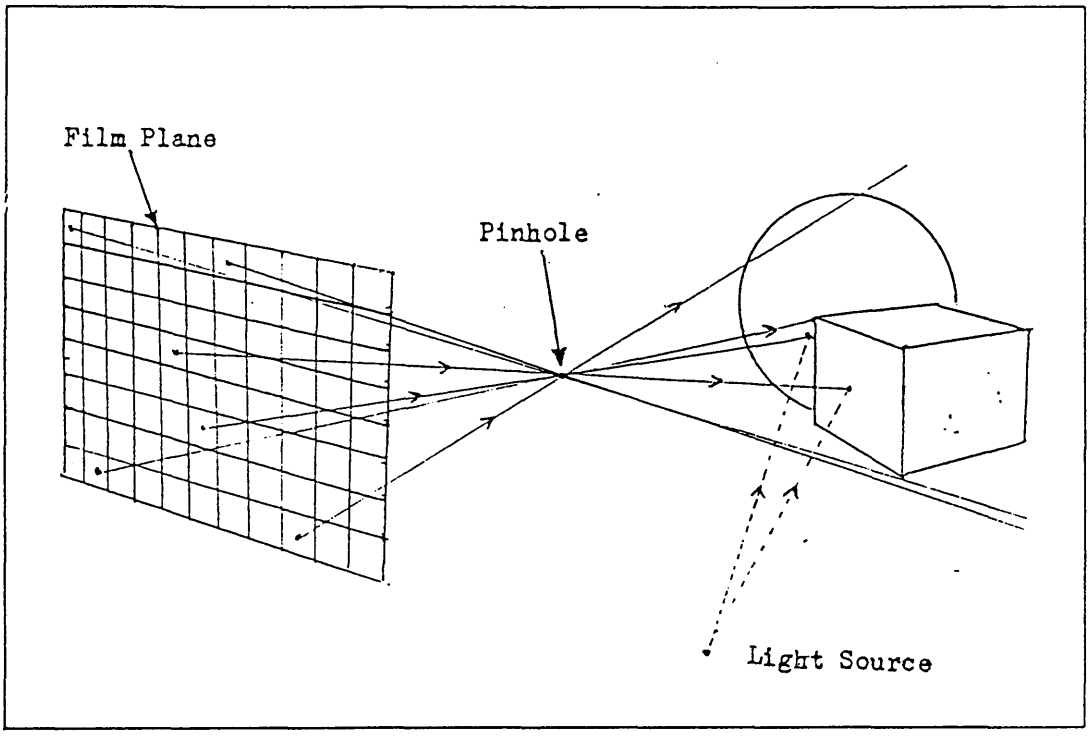


fig. 12 : Pinhole Camera Model.

binary tree with all nodes except the root storing surfaces intersected by the ray, the root corresponded to the initial viewing position. The final pixel intensity value was determined by traversing the tree and computing the intensity contribution of each branch according to the reflection model used.

Kay added further reasons for tracing sight rays rather than light rays. These were -

1. The guarantee that each pixel would have an intensity value.

2. At lower levels of the tree, it may be possible to decrease the number of rays required as these rays get further away. This would be impossible to do if the light rays were traced.

3. Multiple light sources only require one sight ray tree, but if light rays were traced, these would require one tree for each light source.

4. The real shape and size of a light source can be correctly modelled using a sight ray system. Only point light sources can be modelled if light rays are traced.

Whitted [Whitted 1980] also traced rays from the viewing position to the surfaces through the environment and onto the light sources. His shading algorithm was the first to accurately simulate 'true' reflection, shadows and refraction as well as the effects normally already simulated by conventional shaders. Simulating 'true' reflection meant that objects not directly visible to the viewer could be visible through mirrored reflections in other objects. This would be impossible to achieve using conventional hidden-surface elimination techniques. He generated very realistic effects and some astonishing images [Roth 1982].

In 1981 Nelson Max [Max 1981] used ray tracing to render a natural scene containing islands, ocean waves and sky. Rays were reflected by the ocean and checked to determine whether they hit any other object on their way to the sky.

Roth [Roth 1982] described a method of ray tracing objects represented by CSG trees (see section 1.2). Roth's aim was to achieve fast image generation

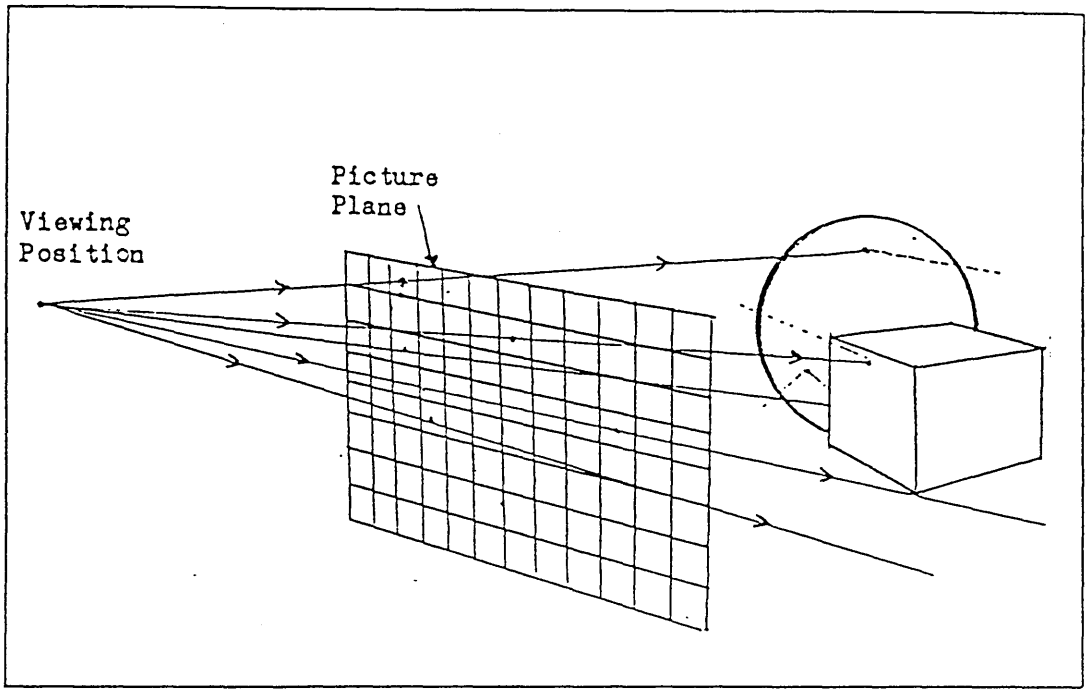


fig. 13 : Sight rays being traced through the environment.

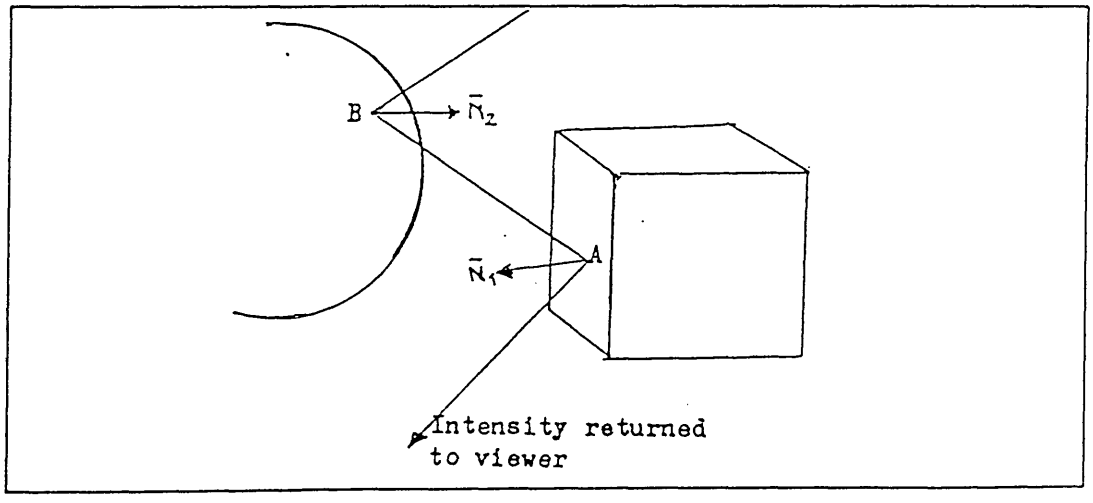


fig. 14 : Components of light reaching the viewer from point A.

for interactive modelling rather than realism. His paper was the impetus for a major part of this research and is discussed in more detail in section 2.3 and chapter 3.

In 1983 Hall and Greenberg [Hall 1983] added a few improvements to the illumination model introduced by Whitted in 1980. These improvements included a hybrid reflection model, a spectral sampling method, the ability to define the position, size and shape of light sources, and adaptive depths of intersection trees.

Ray tracing was incorporated with the Cook and Torrance [Cook 1982] illumination model (see section 1.3.2.1) to generate a high quality image [Bouville 1984]. This illumination model is based on a theoretical analysis of light reflection. It undoubtedly provided the best results that had ever been obtained to date.

In 1984 Cook [Cook 1984] introduced motion blur, depth of field, penumbras, translucency and fuzzy reflections into ray tracing methods. This was achieved by distributing the directions of the rays according to the analytic functions they sampled.

A method of ray tracing a new class of objects was introduced by J. van Wijk [Wijk 1984b]. These objects were defined by sweeping spheres of varying radii along a 3-dimensional trajectory. The basic problem was that of ray-surface intersection. J. van Wijk also introduced another new class of objects defined by sweeping planar cubic splines [Wijk 1984c].

Also in 1984, H. Weghorst, G. Hooper and D. Greenberg [Weghorst 1984] introduced a selection of bounding volumes aimed at reducing the computing time of the ray-intersection test. The paper discusses how to determine optimal bounding volumes. The characteristics looked at include the cost of intersecting the volume, the item complexity and the projected void areas.

In 1985, D. Toth [Toth 1985] extended the use of ray tracing to the rendering of surfaces which could not be generated using existing methods. He solved the ray-surface intersection directly using multivariate Newton iteration.

Because rays were processed individually, computation time could be minimised by using special hardware to enable several rays to be processed simultaneously. As far back as 1979 D.S. Kay [Kay 1979] suggested

spreading the load of ray tracing between a large number of small processors. This is possible when no coherence properties are utilised and each sight ray is traced independently of its neighbours.

M. Dippe and J. Swensen [Dippe 1984] introduced an algorithm which adaptively subdivided a scene into sub-regions that had approximately the same uniform load. The algorithm was mapped onto independent computers, each communicating with a few neighbours.

A.L. Thomas [Thomas 1984] describes how specialised hardware and the development of VLSI circuits may be used to speed up ray tracing. Primitive operations were identified and implemented in hardware.

The recent introduction of machines such as INMOS transputers for parallel processing is likely to have a considerable impact on computer graphics in the near future. A transputer is a VLSI component consisting of a computer on a chip containing a processor, memory and communication links for connection to other transputers [Walker 1985]. A network of these can operate concurrently.

Ray tracing systems cover a wide range of applications, one of which is solid modelling. One of the most famous of these systems is the SynthaVision system [Goldstein 1979, Sorensen 1982, Kinnucan 1983], which uses the CSG representation as the internal model. This was developed by Mathematical Applications Group Inc. (MAGI). Using this system the Walt Disney animators built props and scenes for use in the sci-fi movie TRON. Complex objects are modelled by combining primitive solids (see section 1.2). This system has 18 different types to choose from.

Ray tracing using the CSG representation is discussed in the next section.

2.2. Using the CSG Representation.

Using conventional rendering techniques, CSG-trees are traversed and pairs of objects combined using the set of Boolean operands. This entails computing new boundaries for the intersecting facets of the two objects being joined. There are many possible methods of achieving this. One simple approach introduced by Yamaguchi and Tokieda [Yamaguchi 1984] was to use triangulation. This simplified finding the intersection points; triangles are planar and therefore only one line of intersection exists.

Ray tracing avoids recomputing boundaries by finding the visible surface at each pixel. A major part of the workload in ray tracing is finding the ray-surface intersection points to compute this surface. Computing the intersection points for a single primitive is much simpler than computing them for a complex solid. Therefore finding the intersection points usually involves searching the CSG tree for primitives intersected by the ray. When an intersection is found the enter and exit points are stored. Using a binary tree structure for the CSG-tree, this classification propagates through the internal nodes to the root (see fig. 15). The Boolean operator is used to 'combine' the enter and exit points of the left and right primitives/sub-solids in the tree. This method, or similar, is used by the SynthaVision, GMSolid, PADL-1 and -2 systems.

Alternative methods to traversing the CSG tree for each ray can be used if the CSG tree representation is evaluated i. e. is converted from a 'composite object' representation into a 'single object' representation [Jansen 1985]. If using a polyhedral approximation, this is achieved by intersecting the polygons and merging them into a new model.

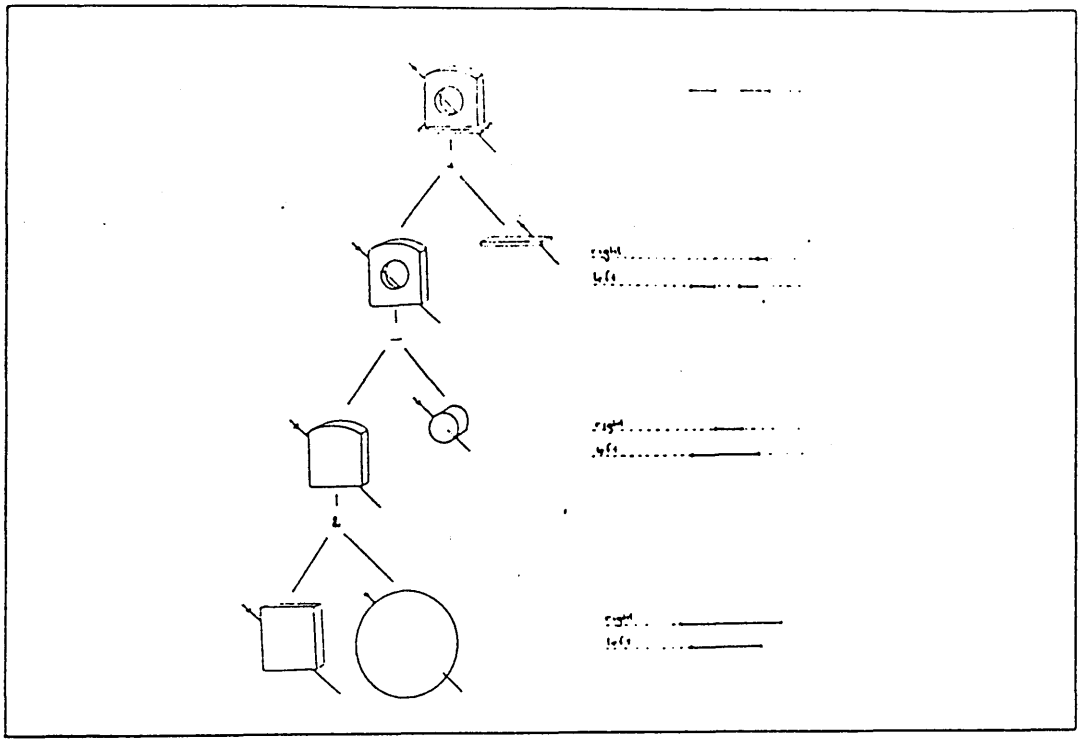


fig. 15 : Classifying a ray.

2.3. Overview of Roth's Paper.

Roth used the ray tracing methods described above. Points on the primitives' surfaces were generated directly by the ray-surface intersection points. Ray-primitive intersection tests were done in the local coordinate space where the primitives were defined analytically. The rays were transformed using the screen-to-local transformation matrix of the primitive concerned. The ray-surface intersection calculation was then reduced to a normalised form.

Three coordinate systems were used to accomplish this. These were -

local coordinate system.

As with some conventional CSG systems (e. g. GMSolid), primitive objects were defined in their own *local coordinate system.* An analytic surface definition exists for each primitive type; the characteristics were the same for any number of solids of the same type. Whenever a new primitive solid was created, its size and position were used to create a scene-to-local transformation.

global coordinate system

Objects in the scene were defined in the *global (or scene) coordinate system.* When an object changed its position or size, changes occurred only in the global coordinate system. The definition of the primitive object remained unchanged in the local coordinate system. The scene-to-local transformation matrix was updated to reflect the change.

screen coordinate system

The third coordinate system was the *device screen coordinate system.* This was linked to the scene coordinate system by the screen-to-scene transformation matrix.

A reduction in computation time was achieved by combining the screen-to-scene transformation and the scene-to-local transformation and storing the resultant screen-to-local transformation. This required multiplying the two matrices and storing the result for each primitive, but halved the number of transformations that the rays had to undergo. There is normally a far greater number of rays than there are primitives.

Rays were classified by finding the ray-surface intersection points for each primitive in the scene. These were then combined as before. Roth suggests that the combine operation should be done in three steps -

1. Compute the 'inside' and 'outside' segments from both the left and right sub-trees in sorted order. This forms a list of in-and-out segments, some of which may overlap.

2. Re-class segments as inside or outside according to the Boolean operator used in the combine.

3. Merge overlapping segments together into one segment.

However Roth assumed at step 3 that different primitive surfaces had the same photometric information, this may not be so. For example, intersecting a red cube and a blue sphere what colour would the segment containing the two newly merged segments adopt.

Ray tracing CSG-trees using Roth's methods is very time-consuming. Consider generating an image on a screen of size $p \times p$ pixels, using a CSG tree containing q primitives. This requires -

1. $(\log_2 q - 1)p^2$ $(q > 1)$ recursive procedure calls.
2. $(q - 1)p^2$ classification combines.
3. qp^2 ray direction calculations
4. qp^2a ray-surface intersection tests (where a is the average number of surfaces per primitive).

Typically p^2 will be $\geq \frac{1}{4}$ million.

Roth suggested the introduction of box enclosures for improving efficiency [Roth 1982]. These box enclosures were cuboids that were defined by six numbers; the minimum and maximum x, y and z-coordinates. The x and y values in screen coordinates defined a rectangular screen area that enclosed the image of the solid/sub-solid and the z values in local coordinates provided the depth bounds.

To enable these box enclosures to be computed with minimal effort, a bounding cuboid in local coordinates for each primitive type was pre-defined and stored with the primitive type's characteristics. The box enclosures in scene coordinates were easier to compute if local-to-scene transformation matrices were available. Therefore the inverse of the scene-to-local transformation matrices were computed and stored for each of the primitives. The box enclosure for a primitive was computed by transforming the eight vertices of the primitive type's cuboid into screen coordinates using the primitive's local-to-screen transformation matrix. The minimum and maximum x and y values of the projected points were stored, along with the minimum and maximum z values of the unprojected points. With the exception of the cube primitive, it was more efficient to transform these vertices than it was to find the maximum and minimum x, y and z values of the vertices defining the surface.

The enclosures were computed and stored for every node in the CSG tree. Prior to the drawing step, the tree was traversed and the box enclosures computed at the primitive (external) nodes. These were then combined using the Boolean operators to provide boxes at internal nodes. For a tree containing q primitives these combinations required $6(q - 1)$ comparisons.

At the ray classification step, the tree was searched from the top downwards, directed by the box enclosures, for branches that were likely to contain 'visible'

primitives. If the pixel that a ray passed through did not lie within the box enclosure at a node then that node and its left and right branches in the tree were ignored. It is easily seen how this reduced all 4 of the requirements listed above. In particular, if all the boxes enclosed a fraction A ($A \leq 1$) of the screen's pixels then a maximum of $A(\log_2 q - 1)\mathbf{p}^2$ $(\mathbf{q>1})$ recursive procedure calls may be required to process the pixels lying within primitives' box enclosures. The search for 'visible' primitives at the remaining $(1 - A)\mathbf{p}^2$ pixels terminates before reaching the bottom of the tree.

All four ray tracing requirements, listed above, depended on the spatial distribution of the primitives in the tree. Requirements 1 and 2 depended on the number of nodes that were visited to classify the rays. Requirements 3 and 4 depended on the number of positive ray-in-box tests. Therefore using box enclosures, for maximum efficiency the external nodes in the tree had to be ordered in such a way that the box enclosures at internal nodes were as small as possible.

Requirements 1 and 2 depended on tree organisation. In general fewer nodes need to be visited in a well-balanced tree than in a badly-balanced tree which, in the worst case, is a linear list. Therefore the tree had also to be balanced. The definition of a completely balanced tree is that there must be at most a difference of one between the number of nodes at the left-subtree as there is at the right-subtree, irrespective of the position of the node in the tree.

Both ordering and balancing the tree kept the number of recursive calls required to a minimum. These calls included the additional pixel-in-rectangle test.

The additional requirements needed to implement box enclosures in a binary CSG tree containing q primitives were -

1. $8q$ local-to-scene transformations to get primitives' depth values.
2. $8q$ scene-to-screen transformations to get primitives' screen rectangles.
3. $6(q - 1)$ comparisons to compute boxes at internal nodes.

The next chapter considers Roth's methods and suggests improvements that can be made.

CHAPTER 3.

CONSIDERATION OF ROTH'S METHOD OF RAY TRACING.

3.1. Improving the Efficiency of Roth's Method of Ray Tracing.

This section introduces proposals for improving the efficiency of the method used at the ray creation step described in Roth's paper. These are delaying the computation of a ray's direction until required and deducing rays using linear interpolation.

Using Roth's ray tracing methods [Roth 1982] for modelling solids, the following restrictions are made -

1. The display screen is defined as being the x-y plane centred at the origin in the world coordinate system.
2. The viewing position is placed along the negative z-axis, in the world coordinate system.
3. The visible solids are positioned in the direction of the positive z-axis.

These restrictions ensure that rays are evenly distributed in the primitives' own local coordinate system where the primitive objects are defined.

At the ray tracing step, rays are cast from the viewing position through every pixel in order left to right, top to bottom. A ray is defined by its screen position (i. e. the pixel it passes through) and direction, which are stored in homogeneous coordinates. Computing a ray's direction requires three subtractions.

For a screen size of $p \times p$ pixels, the ray creation step would require -

$$3p^2 \text{ subtractions} \qquad \text{(a)}$$

Efficiency can be improved at this step by delaying the computation of the ray's direction vector until it is actually required, i. e. the search has permeated down to a primitive node and the result of the pixel-in-rectangle test is positive. Only then is the direction vector computed. In general, using this method most of the direction vectors for rays passing through pixels that will be assigned

background intensity are not computed. The saving at this step is proportional to the area of the screen that all the primitives' box enclosures do not cover.

Roth transforms the ray into the local coordinate system for a primitive. Since the transformation matrix is a 4×4 array, the usual matrix multiplication would require 32 multiplications and 24 additions to transform both the starting position and the direction vector. For a primitive with a box enclosure covering $l \times l$ pixels and full resolution would require -

$$32l^2 \text{ multiplications and } 24l^2 \text{ additions} \qquad \text{(b)}$$

Roth suggests increasing efficiency by omitting multiplication by 0 or 1 and addition of 0. The last column of the transformation matrix is of the form -

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ \# \end{bmatrix}$$

where # will be 1 assuming no dilatation is required. The transformation of the ray's starting position is of the form -

$$[x\ y\ 0\ 1]$$

therefore only 6 multiplications and 6 additions are necessary. The transformation of the ray's direction is of the form

$$[x\ y\ z\ 0]$$

and so requires 9 multiplications and 6 additions. The full ray transformation for a primitive with a box enclosure as above now requires -

$$15l^2 \text{ multiplications} + 12l^2 \text{ additions} \qquad \text{(c)}$$

Efficiency can be improved further by using linear interpolation to deduce rays in the local coordinate space. Whenever a primitive's parameters are input, the 4×4 transformation matrix and the box enclosure are computed. The increments

required to deduce rays can then be computed. These would have to be re-computed if the viewing position changed.

To deduce a ray, only three of the four rays passing through the corner pixels of a box enclosure are created. After these rays are transformed into local coordinates, the increments in the x and y directions are computed. If the viewing position is static the computation can be done at the input stage, this saves time at the drawing stage. The ray creation step for q primitives now only requires -

$$9q \text{ subtractions per scene} \qquad \text{(d)}$$

compared with (a). Now only three rays need to undergo any transformation, the other rays being deduced. Each deduction requires only 6 additions. For a primitive with a box enclosure covering $l \times l$ pixels requires -

$$45 \text{ multiplications } + (6l^2 + 36) \text{ additions} \qquad \text{(e)}$$

which compares very favourably with (c).

Additional work is required to compute the twelve increments needed for the ray's start and direction in x, y and z coordinates for both the 'horizontal' and 'vertical' directions. This involves -

$$12q \text{ subtractions } + 12q \text{ divisions per scene} \qquad \text{(f)}$$

These increments are computed once and stored in the primitive concerned for use each time a ray-surface intersection test is required.

Timing on the VAX 11/780.

Timing measurements were done on the VAX 11/780 at the Department of Computing Science, University of Glasgow. The results are given in the table below. For the purpose of these measurements a primitive with a box enclosure of 500 × 500 pixels was assumed.

As can be seen from the table, the creation of the 250,000 rays corresponding to pixels lying within the box enclosure required 2.3 CPU seconds (see (a)).

At the transformation step the first three columns of the matrix used to transform rays into local coordinates consisted of real numbers, the last column contained [0 0 0 1] as expected. Using only integers in the matrix reduced the time dramatically but in general this is unrealistic. To create the rays and transform them by multiplying by all the matrix entries required 115.8 CPU seconds (see (b)).

Increasing efficiency using Roth's suggestion of omitting multiplication by 0 or 1 and addition of 0 reduces this time to 68.9 CPU seconds (see (c)).

Finally, creating the three corner rays, transforming them and deducing all the others in the local coordinate space required 4.4 CPU seconds (see (e) and (f)). However this time would increase slightly with the number of primitives involved; 10,000 primitives requiring approximately $\frac{1}{7}$ of the multiplication/division operations and $\frac{2}{3}$ of the addition/subtraction operations required by (c).

	CPU seconds per 250,000 rays.	
(a) Roth's ray creation	2.3	
(b) Creation + Full matrix mult	115.8	
(c) Creation + Reduced matrix mult	68.9	
(d) Creation of 3 corner rays	0.0	
(e) Creation + Deducing rays	4.4	

Roth uses the analytic definition of a primitive which requires considerably less storage space than the polyhedral approximation, but before testing for ray-surface intersections, rays must be transformed into the local coordinate system. Since the number of rays will normally be far greater than the number of vertices required to approximate the primitive's surface, it will generally be more efficient to transform the vertices which is the approach taken in the current work.

The improvements proposed in this section were introduced before the introduction of the underlying data structures, that were to enable direct access to required primitives and polygons. After these data structures were introduced rays were mapped onto polygons defined in screen coordinates. Therefore a ray's direction vector in scene coordinates was no longer required.

3.2. Problems in Building Ordered and Balanced CSG-trees.

As discussed in section 2.3, maximum efficiency is gained using box enclosures and a spatially ordered and balanced binary CSG-tree. This section deals with the problems faced in attempting to build such a tree. The problems of ordering are dealt with first.

The most efficient way is to arrange primitives according to the order that they are required by the ray tracing algorithms. For rows of pixels this is in decreasing order of maximum y values. These values are easily obtained from their box enclosures. For columns of pixels this is in increasing order of minimum x values. It may be possible to achieve both of these orderings on any one scan-line of pixels but it is generally impossible to achieve them for an entire image, which is required by the static nature of the binary tree structure.

Strict ordering of the primitives by y-values may not be possible using binary trees due to the ordering dictated by the intersection and difference operators. Since the difference operator is non-commutative

i.e. $\mathbf{a - b \neq b - a}$

and non-associative

i.e. $\mathbf{a - (b - c) \neq (a - b) - c}$

the order of the left and right primitives/sub-solids associated with it must remain static. The intersection operator is associative and commutative so the left and right primitives/sub-solids can be swapped if required. However they must still remain as primitives/sub-solids of the same internal node associated with the operand. This restriction can affect the balancing of the tree.

The ordering problem can be solved by the introduction of an index structure to depict the order for accessing primitive nodes during the search for ray-surface intersections [Cottingham 1988b]. Using these indices it no longer matters whether the tree is ordered or balanced since it is no longer traversed from the top downwards during the search. Introducing indices is discussed in section 4.1.4.

CHAPTER 4.

THE PROPOSED HYBRID METHOD.

The main aim of the work in this thesis is to render unsculptured mechanical parts, balancing realism with processing time. When generating an image it is desirable that relevant data can be directly accessed. Conventionally scan-line methods achieved this by ordering all the polygons, initially in decreasing maximum y value and at the scan-line level by increasing minimum x value. There may be several thousand polygons per scene, therefore this sorting could be very time consuming. Using Roth's ray tracing methods fast access of relevant data is achieved by ordering and balancing CSG-tree nodes. Generally there will be far fewer nodes than there are polygons, which will require less time to sort than scan-line methods. However it is not usually possible to have the CSG-tree built with nodes in the desired order for processing a whole scene (see section 3.2).

The proposed rendering method avoids ordering polygons and balancing CSG-trees, by the introduction of underlying data structures [Cottingham 1988b]. A description of these is given in section 4.1.4 and the proposed algorithm is given in section 4.3.

A dual representation is used for the data with scenes being represented by pseudo CSG-trees, called scene trees (see section 4.1.1) and primitives by polyhedral approximations to their surfaces (see section 4.1.2). Rendering the image is done in scan-line order. Box enclosures are used for fast recognition of potentially visible primitives and to restrict the number of pixels that require their intensity values to be actually computed. Only pixels lying within at least one primitive box enclosure have their intensity values evaluated. No clipping is required since box enclosures are clipped to the viewport when they are created.

With the addition of underlying data structures, it is possible for the primitive nodes (stored at the external scene tree nodes) to be traversed in the order required by the routines generating the image. The underlying data structures can also be used with conventional binary CSG-trees to achieve the same ordering, thus avoiding having to order and balance the CSG-tree nodes. These data structures enable primitives active on a scan-line and polygons incident on a pixel to be directly identified and accessed. These additional data structures are described in more detail in section 4.1. Trees are no longer traversed from the top downwards making it possible to introduce scene trees which are no longer stored in a binary tree structure. Scene trees require fewer nodes than CSG-trees to represent most scenes, however the proposed hybrid method can be used with either tree structure.

Hidden surface elimination is accomplished using a hybrid of conventional scan-line methods and ray tracing techniques. These methods and the conditions under which they are used are discussed in section 4.2.

The proposed algorithm is provided in section 4.3 showing how the underlying data structures are maintained and used at the rendering step and how they determine what hidden-surface elimination method is used.

Linear interpolation (Gouraud shading) is used to evaluate the intensity at each pixel.

4.1. Additional Structures for Fast Access of Data.

This section describes the data structures required to enable direct access to the primitives and polygons required at the drawing step, and to take full advantage of the efficiency gained by employing scan-line coherence and span coherence properties. The word size for storing a 'C' language pointer varies depending on the compiler used. The pointers discussed in this section are assumed to require a $\frac{1}{2}$ word, which is the storage required by the compiler used.

Pseudo CSG-trees, called scene-trees are described in section 4.1.1. However the underlying data structures (i.e. index etc.) are not dependent on the tree structure used and will work equally well with either scene trees or binary trees.

After the CSG-tree has been completely built, an index can be added to enable all the primitives in a scene to be ordered by y values (see section 4.1.4). The addition of an active primitive list enables all the primitives incident on a particular scan-line to be identified and ordered by x values (see section 4.1.5). The addition of a path list for each active primitive enables all the polygons incident on a particular scan-line to be identified and ordered by x within each list. Path lists are described in section 4.1.5.1.

The addition of a span list enables all the primitives incident on a particular span (or pixel) to be easily identified (see section 4.1.6). The addition of a 'polygon in span' list within each span enables direct access to the polygon(s) in the path lists, that are incident at a particular pixel within the span.

There is a pointer to the first node in each of these lists and a pointer to the current node being processed in each list except for the index list. The 'first pointer' to the index list provides access to the next primitive that will become active.

The 'first pointers' to the active primitive list and the span list are used to re-initialise the 'current pointers' after each scan-line has been processed. The 'current pointers' in these lists are used during the processing of a scan-line and are updated as each active primitive/span are completed. The 'first pointer' in the 'polygon in span' list is used to re-initialise the 'current pointer' after each pixel (and scan-line) has been processed.

4.1.1. Scene Trees : A New Structure for CSG-trees.

The addition of an index (see section 4.1.4) enables primitive nodes to be accessed in required order. Therefore, the CSG tree no longer requires to be traversed from the top downwards. This makes it possible to store the CSG tree in a different data structure from the normal binary tree structure. The proposed structure, called a scene tree, is introduced in this section, however the hybrid method is not dependent on its use. The scene-tree structure allows primitives to be grouped using brackets. The next section describes input sequences grouped in this way.

The proposed structure consists of four levels of nodes (see fig. 16). At the lowest level, the external nodes contain the data for the primitive instances. These are built as a one-way linear list for access reasons. Each node contains information about a primitive such as its type, size, defining vertices, directly associated Boolean operand and a list of pointers to its group (or bracket) nodes, if any, that enclose it in the input sequence.

At the next level there are group nodes. These are built as a two-way linear list and are stored in the same order as the opening brackets are input. Each of these nodes contain pointers to the two nodes that correspond to the first and last primitives that are enclosed.

The numbers attached to the brackets in the following input string show the position of each set of brackets in the list -

$$({}^{1}({}^{2}\ \mathbf{a}_{p0} + \mathbf{b}_{p1}\ )^{2}\ - \mathbf{c}_{p2}\ )^{1}\ \&\ ({}^{3}\ ({}^{4}\ \mathbf{d}_{p3} - \mathbf{f}_{p4})^{4}\ \&\ ({}^{5}\mathbf{g}_{p5} + \mathbf{h}_{p6})^{5})^{3}$$

The following table shows the contents of the brackets list :

bracket list position	1	2	3	4	5
Boolean op	-	+	&	-	+
nested level	1	2	1	2	2
open (prim no.)	P0	P0	P3	P3	P5
close (prim no.)	P2	P1	P6	P4	P6
next bracket	2	3	4	5	#
prev bracket	#	1	2	3	4

The primitive nodes are accessed at the rendering step and pointers are followed to the group nodes to find their associated Boolean operands. There are four types of nodes that are of interest -

1. Primitive is associated with only '+' operator(s).
2. Primitive is on left-hand side of '-' operator.
3. Primitive is on right-hand side of '-' operator.
4. Primitive is associated with an '&' operator.

Type 1 primitives will be fully rendered whereas the others will probably have a piece cut off to reflect the Boolean operator. The following pseudo-code shows how the associated types are found -

```
if current primitive is in a group then
    begin
    set current bracket to bracket pointed at by current primitive
    for 0 to nested level
        begin
        if current bracket's operator is '-' then
            check and return type 2 or 3
        if current bracket's operator is '&' then
            return type 4
        set current bracket to previous bracket in list
        end
    end
```

```
if current primitive's operator is '+' then
        return type 1
if current primitive's operator is '-' then
        return type 3
if current primitive's operator is '&' then
        return type 4
if next primitive's (to current primitive) operator is '-' then
        return type 2
```

The complex solid nodes are at the level above the group nodes. These are built as a one-way linear list and have pointers to the four nodes corresponding to the first and last primitive and first and last group input for the solid. These nodes are only used for deleting/updating part/all of the complex solid and are not required at the rendering step.

At the highest level there is a scene node. This forms the root of the scene tree and contains pointers to the two nodes corresponding to the first and last complex solids input. Figure 16 gives an example of the whole structure.

The introduction of groups and storing the Boolean operators at the primitive nodes, in general enables a CSG-tree to be represented by fewer nodes. To represent a solid containing p primitives the binary tree structure requires $(2p - 1)$ nodes. In comparison the new structure requires $(p + b + 1)$ nodes, where b is the number of bracket nodes. Normally b will be much less than p. For example, fig. 17(a) shows a drawing of a vice. A possible input sequence to create such an object could be -

```
(( a + b ) - ( c + d + e + f + g )) + h + i + ( j - k ) + l
    + (( m + ( n - o )) - ( p + q + r + s )) + t
```

This combines 20 primitives. The corresponding CSG tree in binary form requires 39 nodes and is shown in fig. 17(b). Figure 17(c) shows the tree represented by the new data structure containing 27 nodes, the ordered set of indices is also shown.

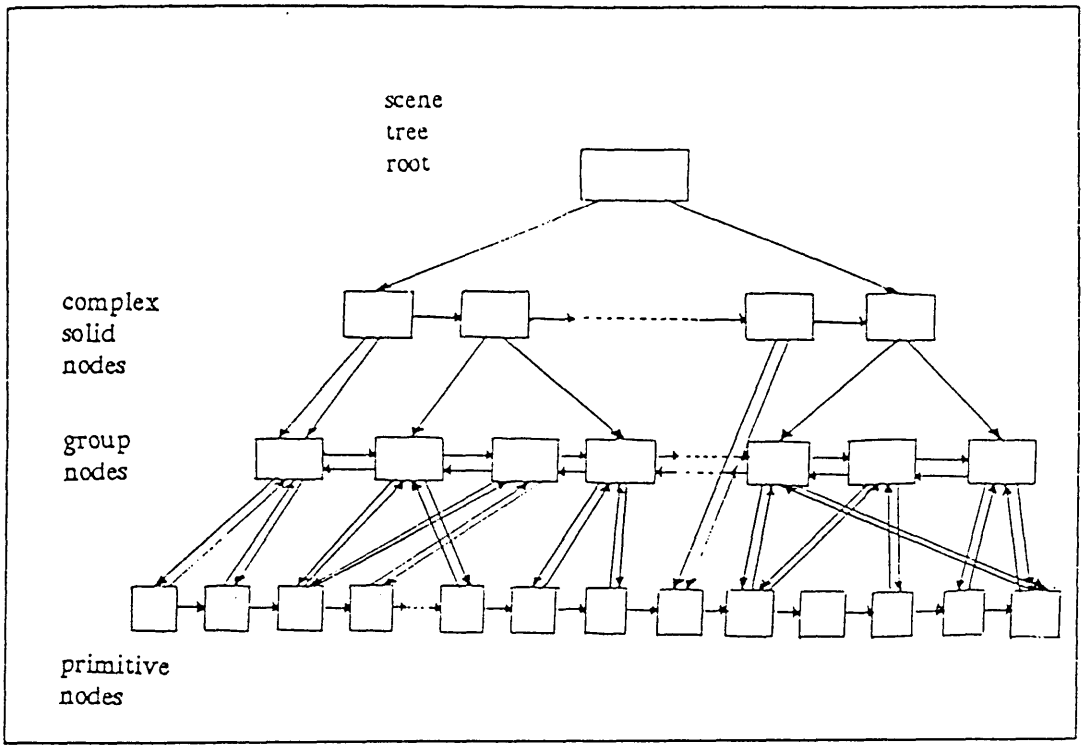


fig. 16: Example of scene tree structure.

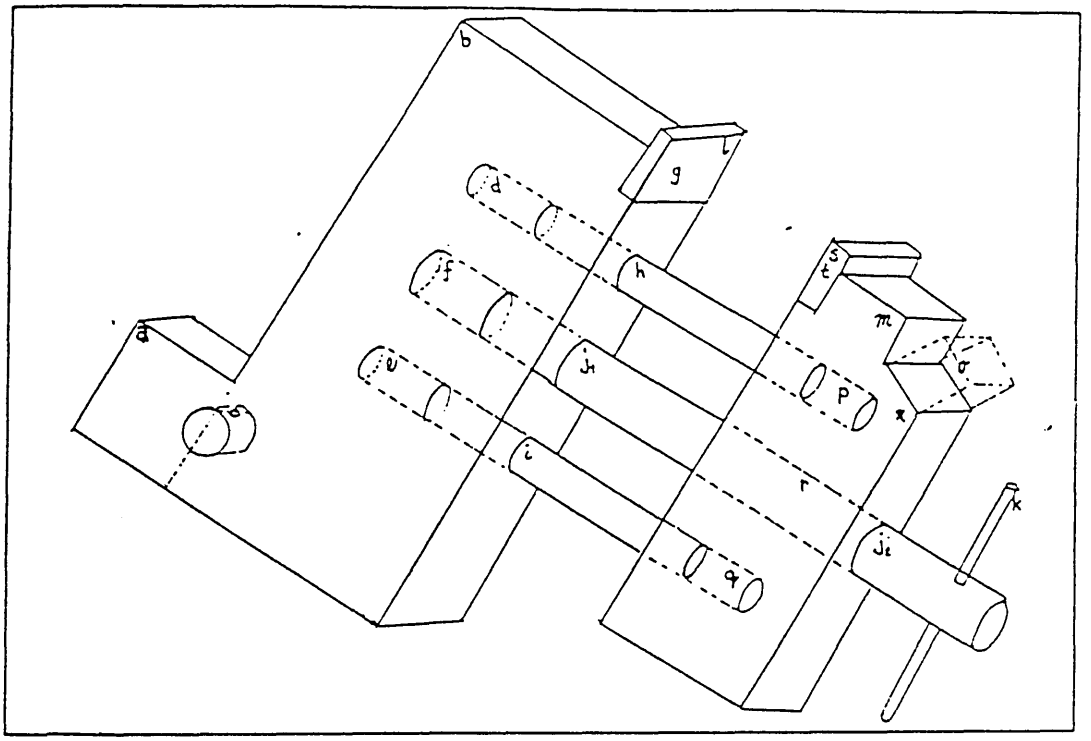


fig. 17 (a) Drawing of a vice.

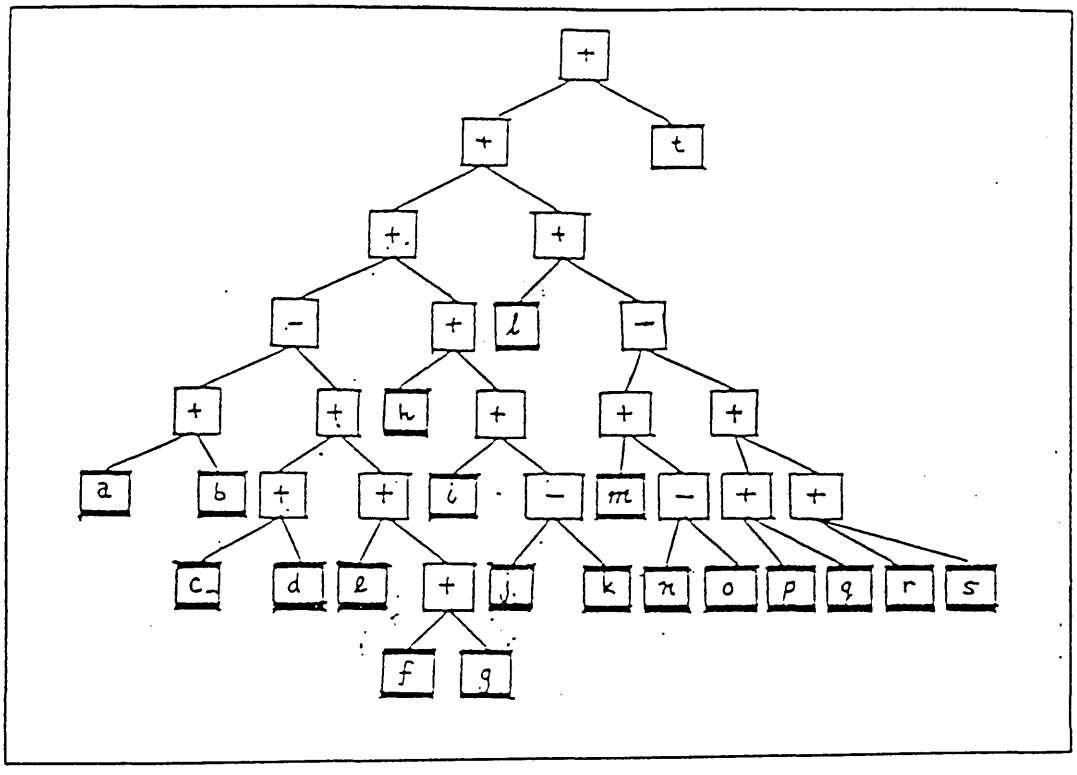


fig. 17 (b) Binary tree structure representing the vice.

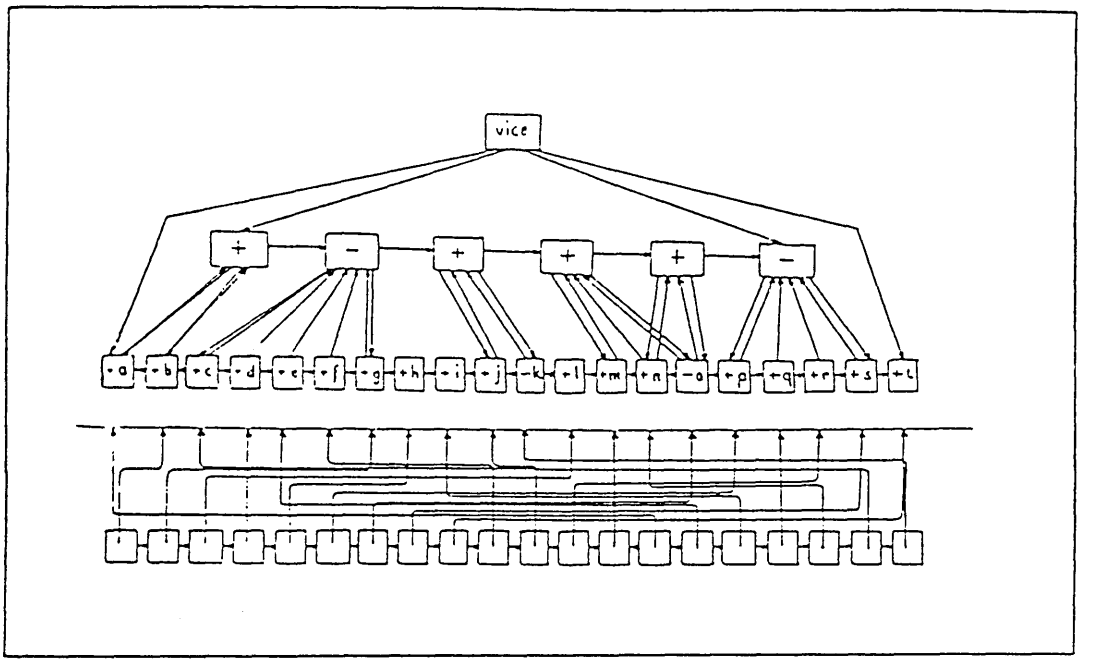


fig. 17 (c) Scene tree structure representing the vice.

4.1.1.1. Input Sequences Containing Groups of Primitives.

Three distinct input entities are used for the creation of a scene:

complex solid : A complex solid is a complete entity that forms part/all of the scene. It is identified by a name and comprises a collection of one or more primitive(s)/primitive group(s).

primitive group : There are three distinct types of groups:

(i) a group of primitives that are combined to form a sub-solid.

(ii) a primitive(s) / primitive group(s) that immediately precede and succeed an intersection or difference operator (grouped together to avoid ambiguity).

(iii) several primitives with a common operand (grouped together for convenience).

primitive solid : A primitive solid is an entity that forms part/all of a complex solid. It belongs to a class of basic primitive objects, viz block, cylinder, cone or sphere.

Round brackets are used to denote the groups, which allows a variety of input formats. Firstly it permits sub-solids to be identified by enclosing them in a set of brackets. (In some CSG systems this is achieved by naming a group of primitives). The sub-solid can then be transformed (i.e. translated, rotated etc.) as a single entity.

Secondly grouping can be used to produce an unambiguous input sequence. For example, suppose a solid object consists of two sub-parts a sphere **S** and a cube **C** that contains a cylindrical hole **H**. The expected input would be -

$$S + (C - H)$$

The operator preceding a primitive in the input string is considered to be the directly associated operator for that primitive. This operator is stored in the primitive node. For convenience it is assumed that the first item of input for a complex solid or primitive group is a primitive that is directly associated with a union operand. Therefore the first primitive does not require to be preceded by a Boolean operator. The above input sequence is interpreted by the system as -

$$+ S + (+ C - H)$$

In the above example the grouping is used to indicate which primitives are associated with each of the operators, so that they can be correctly combined. For example omitting the brackets for the solid above gives -

$$S + C - H$$

which could be interpreted either as -

$$S + (C - H)$$

or

$$(S + C) - H$$

which would provide different results if object **H** intersects with object **S**.

Note that it is possible for a primitive to have several operands associated with it. For example in the input sequence -

$$S + (C - H)$$

H has a ' - ' operand directly associated with it and a ' + ' indirectly associated with it.

Finally grouping can be used to collect primitives sharing the same operator and its preceding operand. For example, suppose a solid consists of a cube

containing three holes, a possible input sequence is -

$$(C - (H_1 + H_2 + H_3))$$

A possible internal representation is shown in fig. 18. It is often simpler to decompose a complex solid using a mixture of these input methods.

Using the Backus-Naur form of notation, the production rules for the context-free grammar to define all legal regular expressions for an input sequence are as follows -

< expression>	::=	< subexpr>
	|	< factor>
	|	(< expression> & < expression>)
	|	(< expression> - < expression>)
< factor>	::=	(< subexpr> + < subexpr>)
< subexp>	::=	< terminal>
	|	< terminal> + < subexp>
< terminal>	::=	U | Y | O | S

As usual the metasymbol < ....> denotes a non-terminal symbol, ::= is read as "is defined to consist of" and | is read as "or alternately". The terminal symbols denote the primitive types U - cube, Y - cylinder, O - cone and S - sphere.

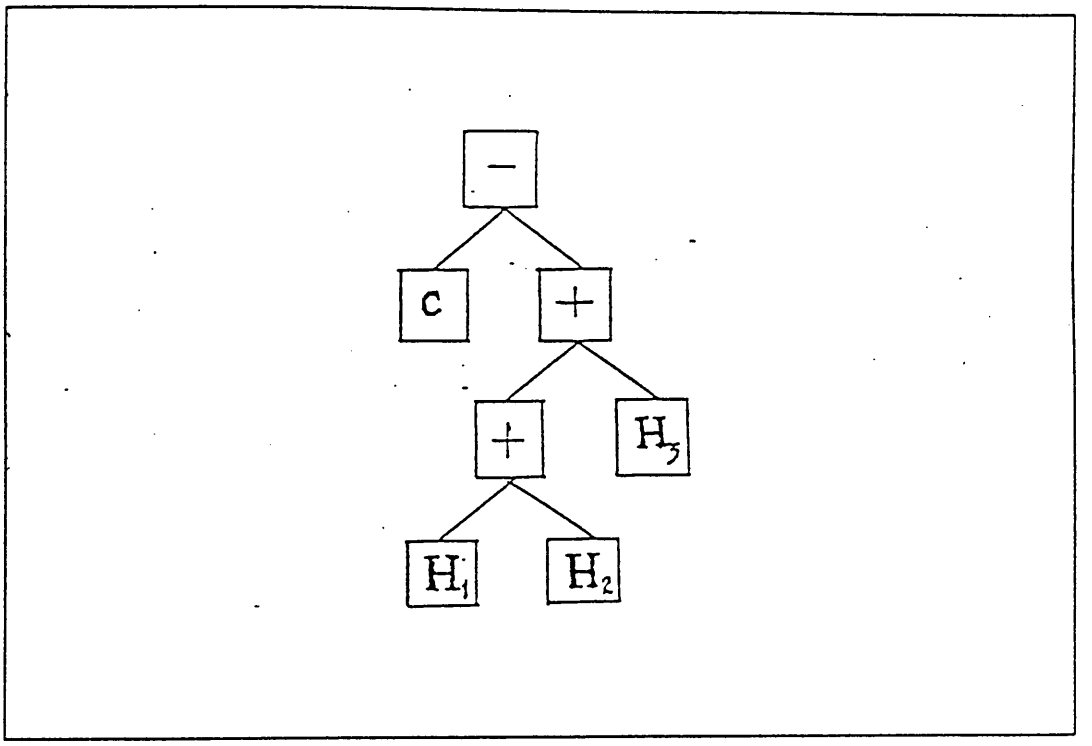


fig. 18 Logical internal representation for input sequence.

4.1.2. Computing the Surface Definitions.

As in the GMSolid modelling system a template primitive is stored for each primitive type. These have their surfaces approximated by polygons that are defined in the local coordinate system (see section 1.2). As with some other systems there is a routine for the creation of every primitive type.

When a primitive's type and parameters are input, the routine for the given type generates the 3D coordinates of the vertices required for a polyhedral approximation to the surface. These vertices are generated by passing the 'local' vertices approximating the template primitive through the local-to-scene transformation matrix. The screen position, normal and intensity value for each vertex are then computed and stored.

4.1.3. Calculation of Box Enclosures.

During the input step the box enclosure for each primitive is computed by transforming the template's bounding cuboid. As in Roth's methods the x and y coordinates of this cuboid are transformed to screen coordinates to provide a bounding rectangular screen area. Roth transformed rays into the local coordinate space and used the z coordinates of the template cuboid to define the depth bounds. The hybrid method works in the scene coordinate space to determine the depth, therefore the z coordinates of the bounding cuboid are transformed into scene coordinates to provide the depth bounds that are required by the hidden-surface algorithms.

The box enclosure may be slightly larger than necessary using this method, but only 16 coordinates are compared (see fig. 19 (a) and (b)). To achieve an exact minimum box enclosure, as shown in fig. 19 (c), would require $2n$ comparisons (where n is the number of vertices).

Box enclosures are clipped to the viewport to ensure all pixels processed actually lie within the viewing area on the screen. If two of the edges of a box enclosure lie outside the viewport, these edges are redefined to lie along the edge of the viewport. If all of the edges lie outside the viewport the box is set to a NULL value and its associated primitive ignored. The box enclosures are used at

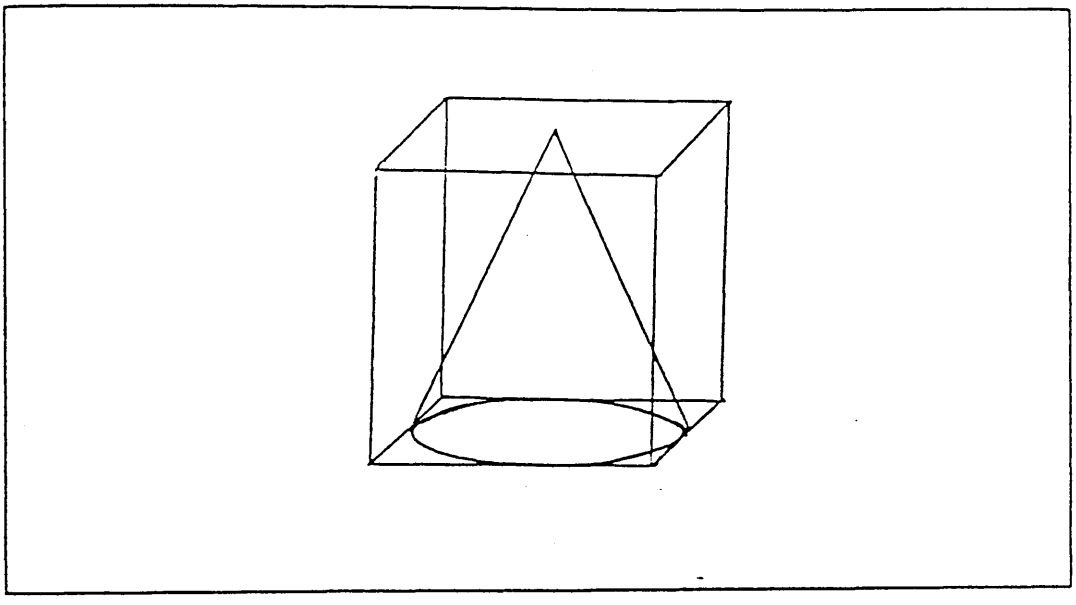


fig. 19 (a): Template primitive with box enclosure.

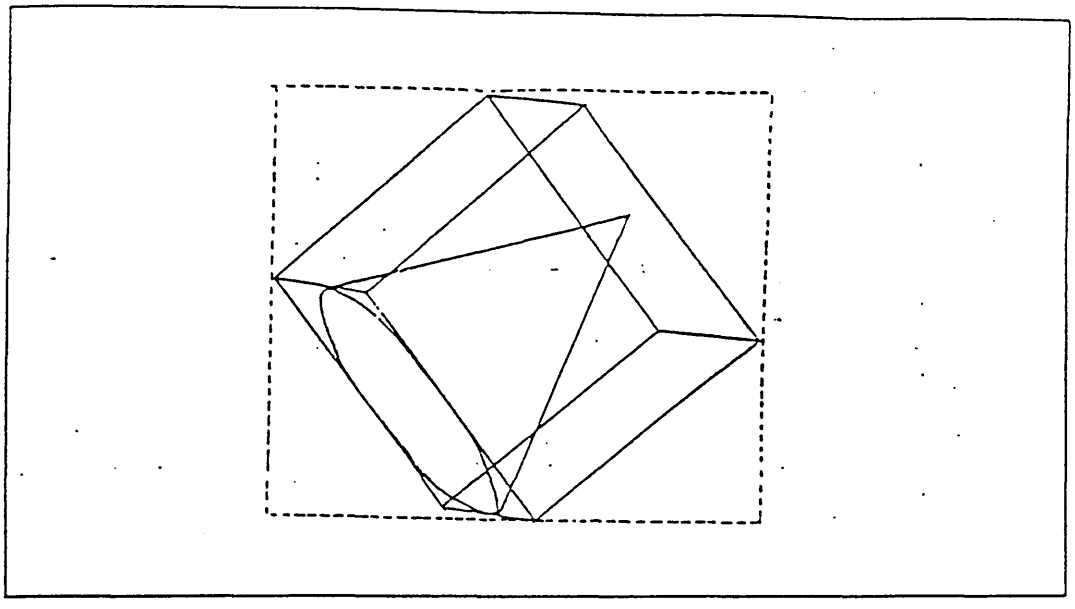


fig. 19 (b): Deduced box enclosure in screen coordinates.

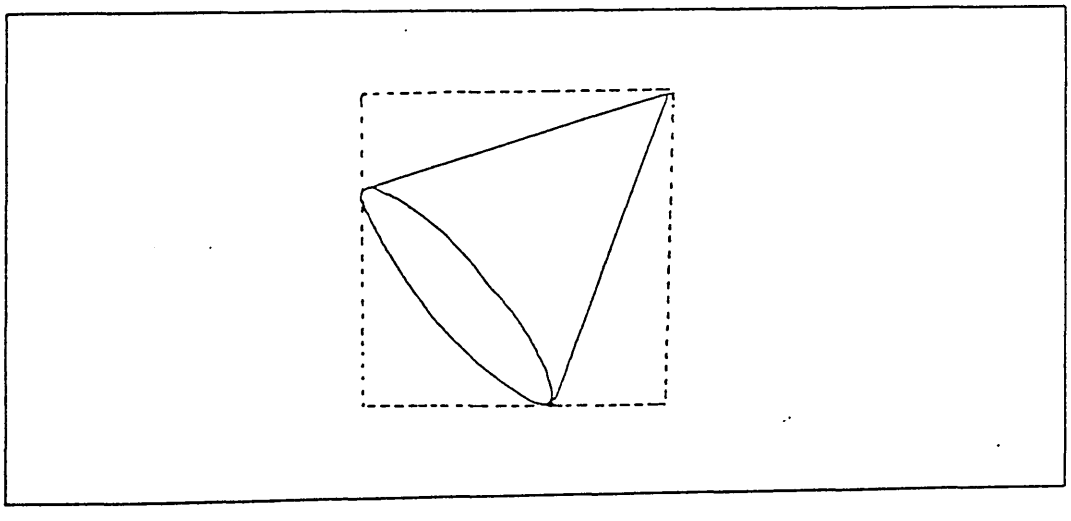


fig. 19 (c): Exact minimum box enclosure in screen coordinates.

the rendering step to determine the pixels to be processed.

4.1.4. Indexing CSG-tree Terminal Nodes.

Accessing data for image generation, generally requires that it is sorted into some order. Section 3.2 outlines the difficulties in attempting to impose an order on the terminal nodes of a CSG tree. In the present method these difficulties are avoided by the introduction of an ordered index containing pointers to the primitives stored at external nodes (see fig. 20 (a)). The index enables direct access to the primitive information that is required by the rendering algorithms.

The index is created as a linked list directly after the definition of the scene has been input and immediately before the rendering step. Index entries are ordered in decreasing order of the maximum y values (in screen coordinates) of the associated primitive box enclosures (see fig. 20 (b)). When ordering the index only the contents of the nodes change, the nodes themselves maintain the same position in the list. This results in fewer changes of addresses. To swap the positions of two nodes in the list requires four pointers to be updated, swapping their contents only requires updating the two pointers to the external (primitive) nodes.

An advantage of accessing primitives directly, instead of by a tree traversal, is that the evaluation of pixels can be restricted to those that lie within primitive box enclosures. The box enclosures at internal nodes in the tree no longer need to be computed.

Another advantage is that some Boolean operators may not be required. Using the traditional top downwards traversal of the tree, nodes containing Boolean operators are accessed and their contents noted during the traversal, primitive nodes are accessed last. It is always necessary to access all of the primitive nodes in a tree during the rendering step therefore the Boolean operands are found for all primitives. Accessing the tree from the bottom upwards, the primitives are accessed first, so it is not necessary to find the Boolean operands associated with some of the 'hidden' primitives (see section 4.2.2). Finding the Boolean operands in a binary tree structure requires the tree to be traversed up to

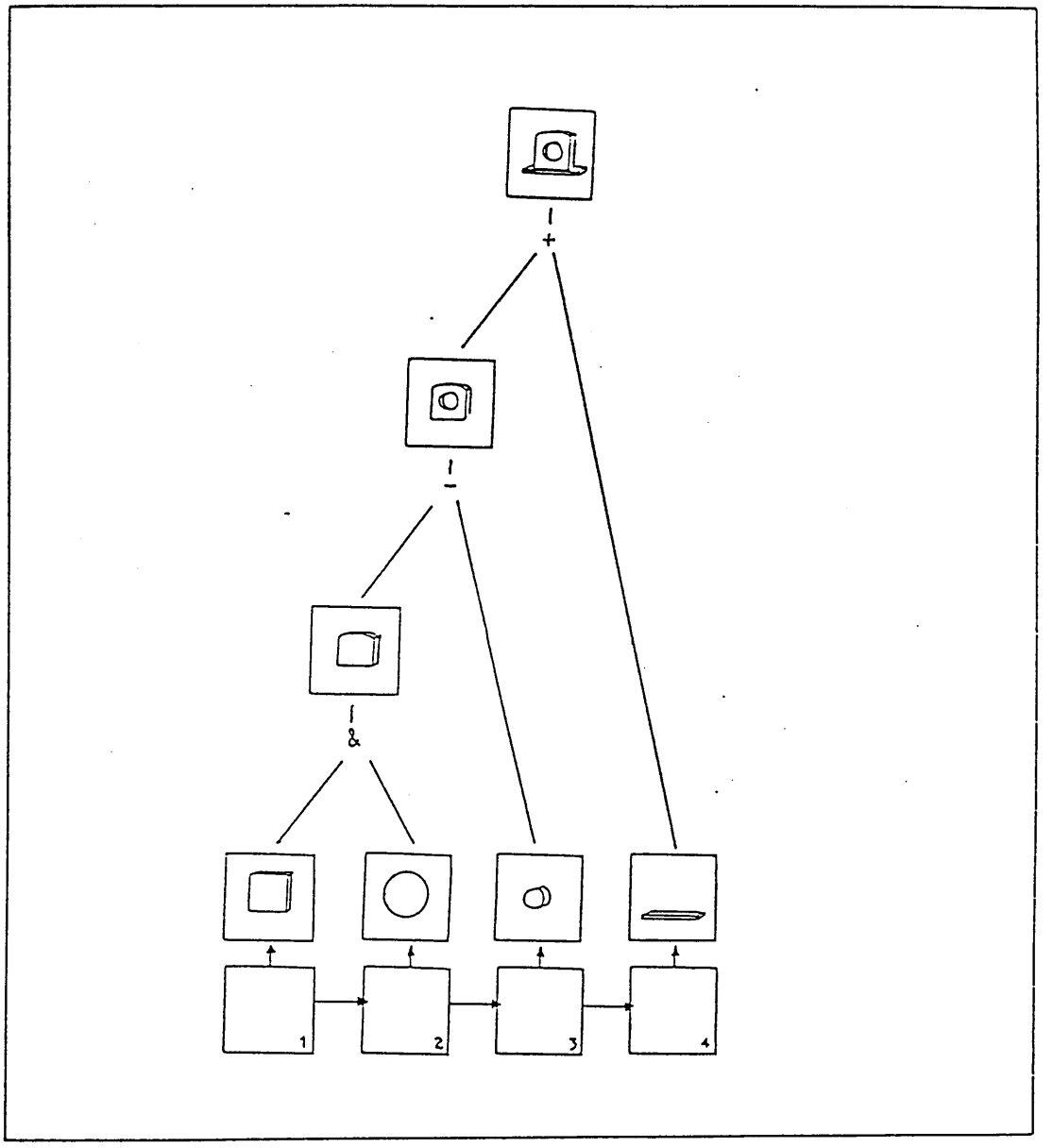


fig. 20(a) : CSG-tree with index.

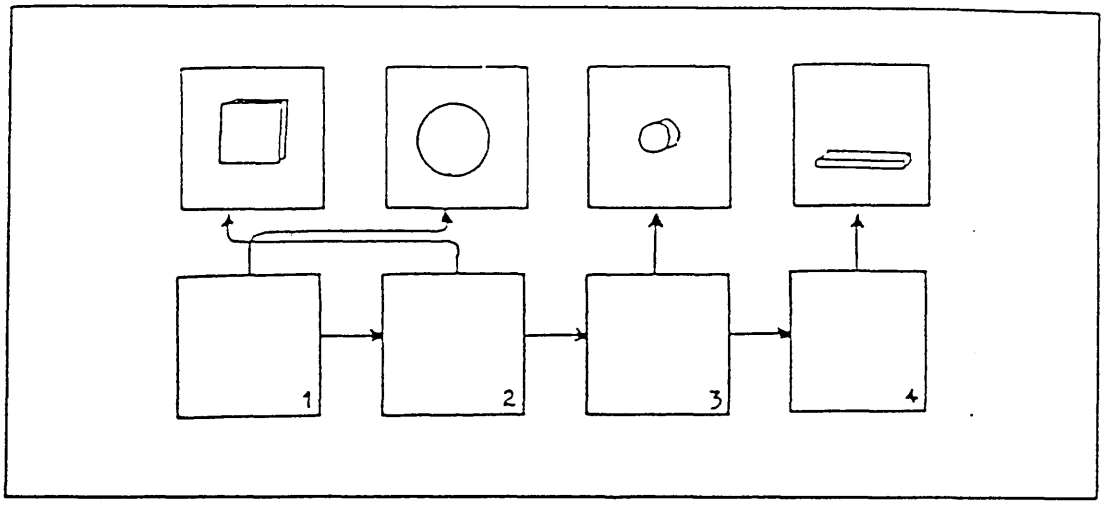


fig. 20(b) : CSG-tree with index ordered by y.

the root.

Another advantage of using an index is that primitives associated with difference and intersection operators can still be correctly ordered. Using a binary tree structure primitives are restricted to being stored at the left or right branches of these operands. This may cause the box enclosure at the internal node to be much larger than actually required, adding to the inefficiency of the top downwards traversal.

Ordering in the x direction is achieved by creating an active primitive list containing all the primitives intersected by the current scan-line. Throughout the rendering step nodes are transferred from the 'front' of the index to this active primitive list. Thus at any particular time the index only contains pointers to those primitives that have their box enclosure lying below the current scan-line being processed. After the rendering step the index is empty. The active primitive list is discussed in the next section.

The storage requirement for the index depends on the number of primitives in the scene. Initially the index will require a node for every primitive in the scene. Each node contains a pointer to a primitive and to the next node in the list. Therefore the initial storage requirement for **P** primitives is **P** words. This requirement will diminish as the image is generated.

4.1.5. Creating and Maintaining the Active Primitive List.

The active primitive list is created/updated by transferring nodes from the beginning of the ordered index. Initially the first node in the index list contains the pointer to the primitive that has the top of its box enclosure 'nearest' to the top of the display screen. A node is created in the active primitive list for this first node and any subsequent nodes that have their box enclosure starting at the same level.

New nodes are placed into their correct position in the active list, which is ordered by increasing order of minimum x values. In general only a few boxes will become active in any one scan-line. Any boxes becoming inactive are deleted from the list. The following pseudo-code shows how ths is done -

```
set current to first node in active list;
while not   end of active list   do
    begin
    if current node not active do
        remove current node;
    set current to next node;
    end;
set current to first node in index list;
while current node is active do
    begin
    insert into active list in correct order;
    set current to next node in index list;
    end;
```

Because box enclosures are approximated their size may be larger than required (see fig. 21 (a), (b) and (c)). Therefore the index provides only an approximate ordering of the primitives, which may cause a primitive's associated node to be maintained in the active list for longer than is really necessary.

The active primitive list will normally remain constant over several adjacent scan-lines (see fig. 22). Changes occur when a box enclosure's top/bottom occurs at the current scan-line. The first entry of the index provides the scan-line where an addition to the list will occur. All the elements in the active list are examined to determine the next scan-line where a deletion from the list will occur.

The active primitive list is maintained throughout the rendering step and is non-existent afterwards. It is possible for the list to be completely deleted and re-created several times throughout this step. This happens whenever scan-lines containing no active primitives occur after the image generation has begun and the index is not empty.

Whenever a node is added to the active primitive list a path list is created with a node for each polygon incident on the current scan-line. Pointers to the first and current path nodes are stored in their associated active primitive node. More information is provided about paths in the next section.

The active primitive list enables the scan-line to be divided into the spans (or segments) that enable overlapping primitives to be identified. These determine the hidden-surface elimination technique that is to be used and are discussed in section

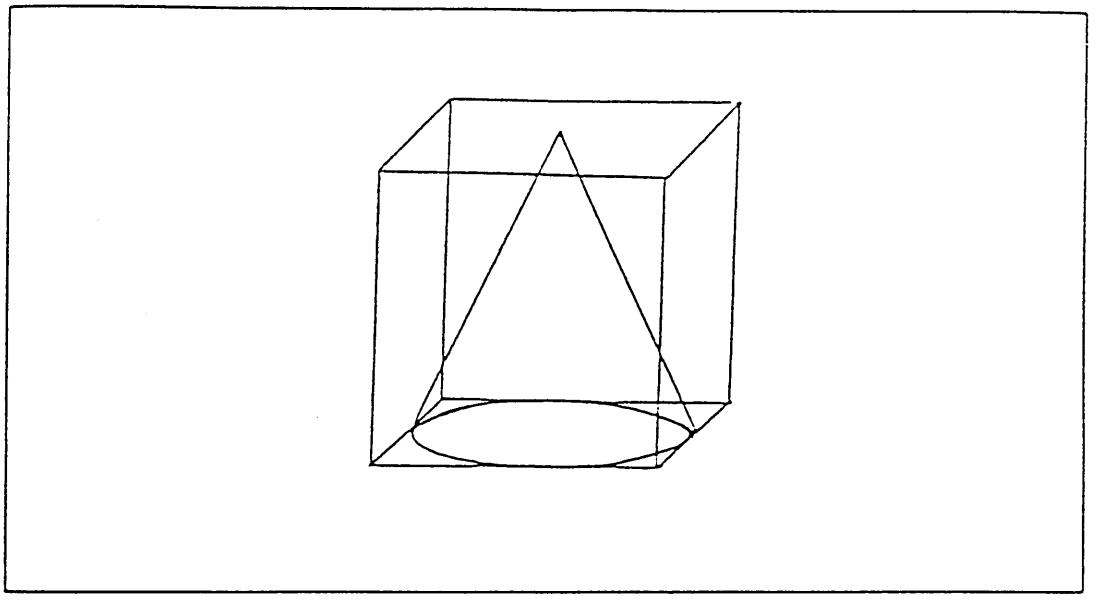


fig. 21(a) : Template primitive with box enclosure.

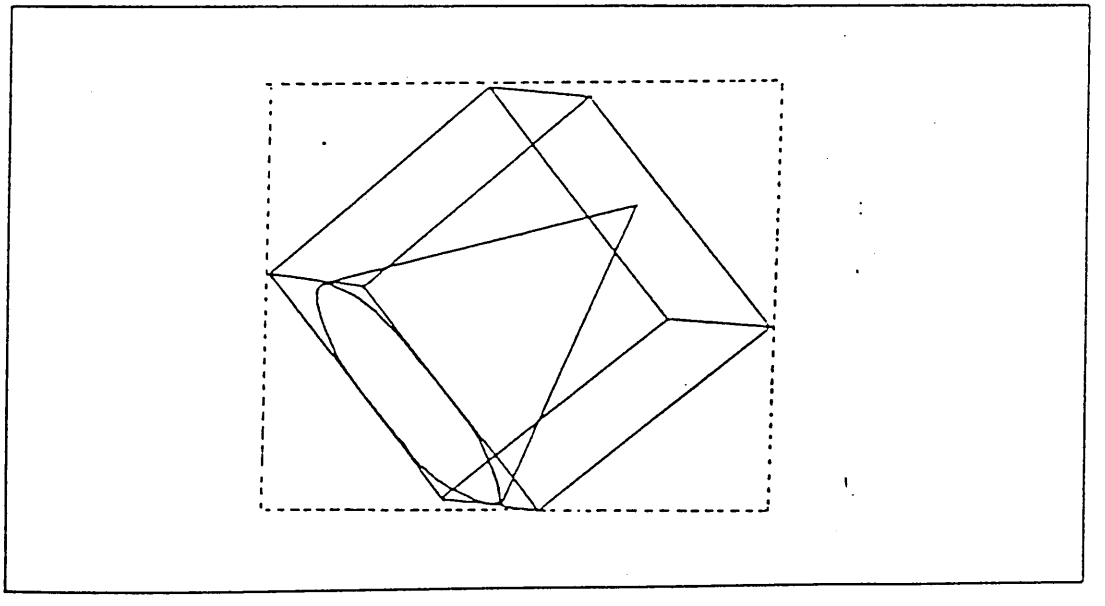


fig. 21(b) : Deduced box enclosure in screen coordinates.

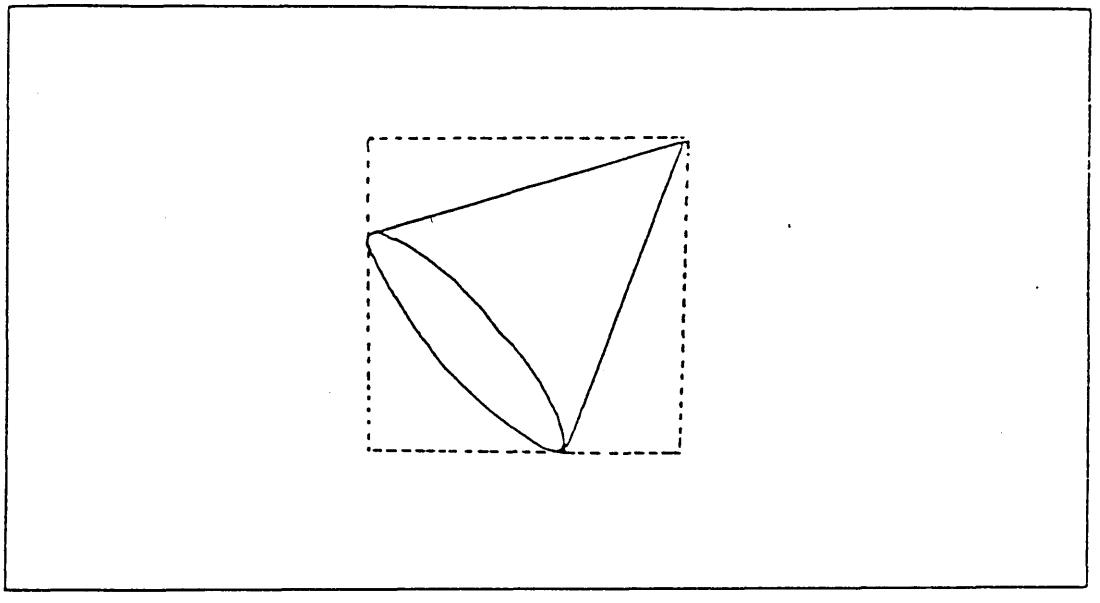


fig. 21(c) : Exact minimum box enclosure in screen coordinates.

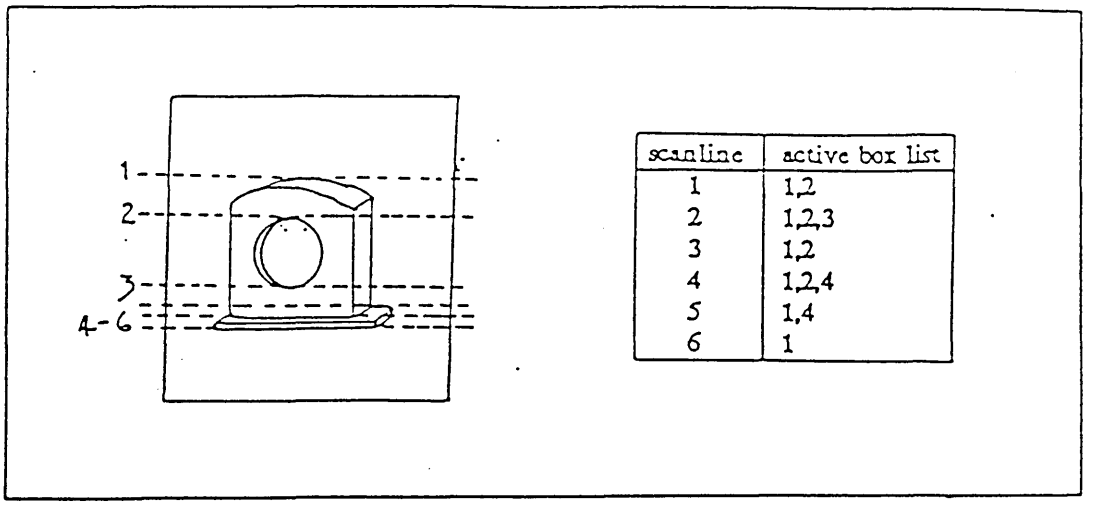


fig. 22 : Active list ordered by x for given scan-lines.

4.2.

The storage requirement for the active primitive list is dependent on how many primitives are active on any one scan-line. Each node contains a pointer to a primitive and the first and current 'visible' path nodes, space has to be reserved for the first and current 'invisible' path nodes too whether or not these are required (see section 4.2.2). There are also pointers to the next and previous nodes in the active primitive list. Each node requires $3\frac{1}{2}$ words.

4.1.5.1. Path Through the Data Structure.

In the proposed hybrid method fast access to visible polygons is achieved with the aid of paths through the primitive data structures. A path is a linked list of nodes, each one containing a pointer to a facet that is incident on the current scan-line. The facets pointed at by the path list at any particular scan-line are not dependent on the data structure used. A path is created when a primitive first becomes active (see fig. 23). Path nodes are maintained in increasing order of the x values of the first edge intersected.

The initial creation of a path list entails finding the facet(s) intersected by the current scan-line. Because scan-lines are generated in order, this is achieved by searching the data structure for the facet containing the vertex with the maximum y value. A fast method for finding this vertex when using the CDS is given in section 4.1.5.1.1.

If conventional hidden-surface methods are used the path node is accessed and the following information is retrieved -

the two x positions of the edge/scan-line intersection points.
the two intensity values at the edge/scan-line intersection points.
the increment in intensity between two pixels.

This information is used to linearly interpolate the intensity values at the pixels lying between the two x positions and within the current span. The path node is not updated to reflect the current intensity value.

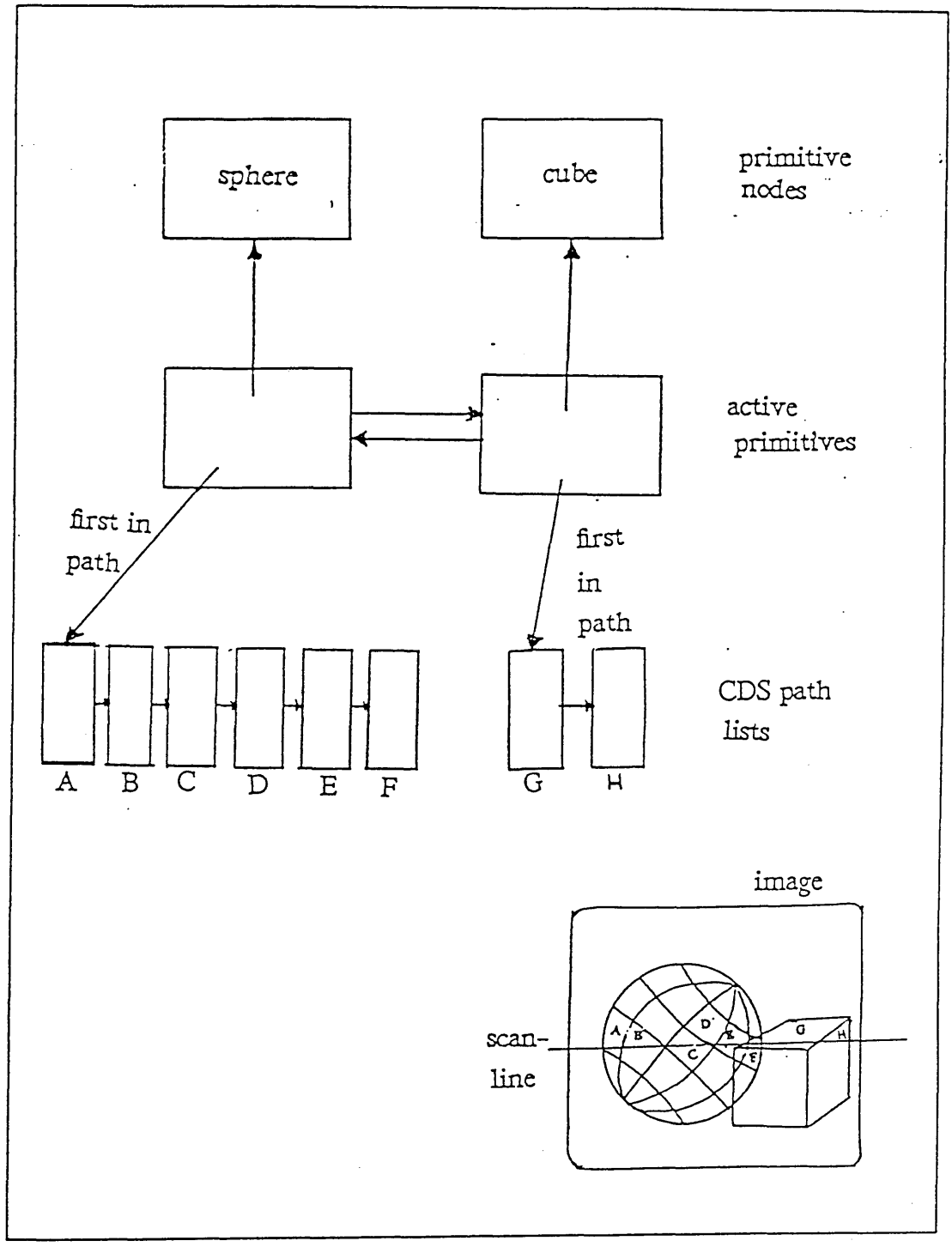


fig. 23 : Active list with associated path lists.

If ray tracing techniques are used and the polygon associated with the path node contains the current pixel, the following information is retrieved -

the depth value at the current pixel.

the intensity value at the current pixel.

Both these values are updated in the path node and will be retrieved by the next ray.

After each scan-line has been completely processed the information in the path nodes is updated to reflect the changes between scan-lines. The depth and intensity values between pixels are reset to the edge values. At any particular time each path node contains the following information -

pointer to visible facet on scan-line.

the two edges intersected by scan-line.

the two x positions of the edge/scan-line intersection points.

the two depth-values of the edge/scan-line intersection points.

the two intensity values at the edge/scan-line intersection points.

the following increments reflect the difference between two adjacent scan-lines :

the increments in x along the two edges.

the increments in depth along the two edges.

the increments in intensity value along the two edges.

the intensity value at the current pixel (or at the first edge intersected if not yet active).

the depth value at the current pixel (or at the first edge intersected if not yet active).

the following increments reflect the difference between two adjacent pixels:

the increment in intensity.

the increment in depth.

the top-most and bottom-most vertices.

the next and previous path nodes in the list.

Each path node exists until the scan-line reaches the bottom-most vertex of its associated facet. Because the box enclosure is an approximation (see fig. 21) the top-most/bottom-most facet may differ from the top/bottom of the box

enclosure. Therefore it is possible for an active primitive to have an empty path list. After a primitive has been generated its associated path list will be empty.

The path through the front facing polygons is required for all primitives to determine their depth. It is this path that is created initially when a primitive becomes active. A path through the back-facing polygons may also be required if the primitive is directly or indirectly associated with an intersection or difference operand and is not 'hidden'. For efficiency the creation of the path through back-facing polygons is delayed until required. Both 'front-facing' and 'back-facing' paths may be used by the ray tracing algorithms to determine the correct polygon intersected. This is discussed in more detail in section 4.2.2.

Path nodes each require $21\frac{1}{2}$ words of storage. The number of path nodes required is dependent on the number of polygons intersected by a particular scan-line.

4.1.5.1.1. Creating a Path Through the CDS.

The compressed data structure (CDS) [Cottingham 1985 (included as appendix B),Cottingham 1987 (included as appendix C)] is a 2-dimensional array. Each element contains the data required for one facet. Using the CDS a minimum amount of work is required for ordering and searching because of the inherent nature of the data structure i. e. adjacent facets in the scene are also 'adjacent' in the data structure. This is also an advantage when moving from one scan-line to the next, when paths are updated to reflect the polygons that are incident on the new scan-line.

The path list contains pointers to either the visible facets incident on a particular scan-line or the invisible facets (see section 4.1.5.1). No path list will contain pointers to both types. This will approximately half the number of facets of interest when creating the path list.

Because of the correspondence between facets and their array positions, a search is made to find one facet intersected by the particular scan-line. Creating a path containing visible facets, the search initially begins by finding one visible facet. An algorithm for achieving this is given in appendix B, section 4.2.

The search then continues for a facet intersected by the scan-line. The data contained at each element in the array includes the 2D and 3D coordinates of the vertices defining a facet, therefore it is easy to compute if the scan-line intersects the visible facet found. If it does not the 2D y-values of the facet and the visibility of adjacent facets determine whether to consider the adjacent array element lying to the 'north', 'south', 'east' or 'west' of the intersected facet. This procedure continues until an intersected facet is found.

Other facets intersected by the scan-line are found in a similar manner but coherence is applied. If the facet lying to the 'north' of the facet found is intersected by the scan-line, the next facet to be examined is the one lying to the 'north' again. If this facet is not intersected, the facets to the 'north-east' and 'north-west' are examined, and so on. Facets to the 'south' of the first facet found are also examined. Eventually all the visible facets intersected by the scan-line are found and pointed to by the path list.

Creating a path list for invisible facets is done similarly.

4.1.6. Creating and Maintaining the Span List.

The span list is created by processing the active primitive list and dividing the scan-line into spans with boundaries where the scan-line enters and exits each active primitive's box enclosure (see fig. 24 (a)). The span list is updated whenever the active primitive list is updated. The span list reduces the hidden surface problem to determining which primitive is visible from a selection of the possibly visible primitives in each span of the scan-line. There are three types of spans possible -

1. empty span (spans 1 and 7 of fig. 24 (a)).
2. span containing one primitive (spans 2 and 6 of fig. 24 (a)).
3. span containing more than one primitive (spans 3, 4 and 5 of fig. 24 (a))

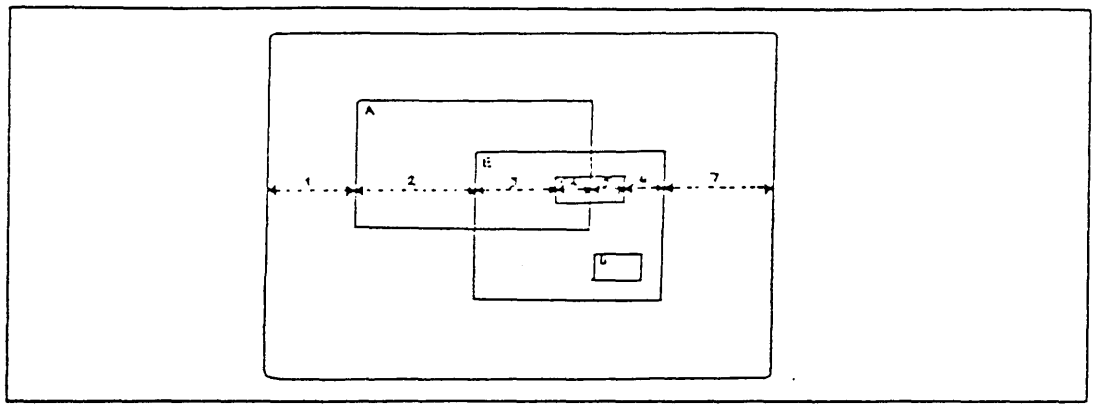


fig. 24(a) : Spans of a scan-line.

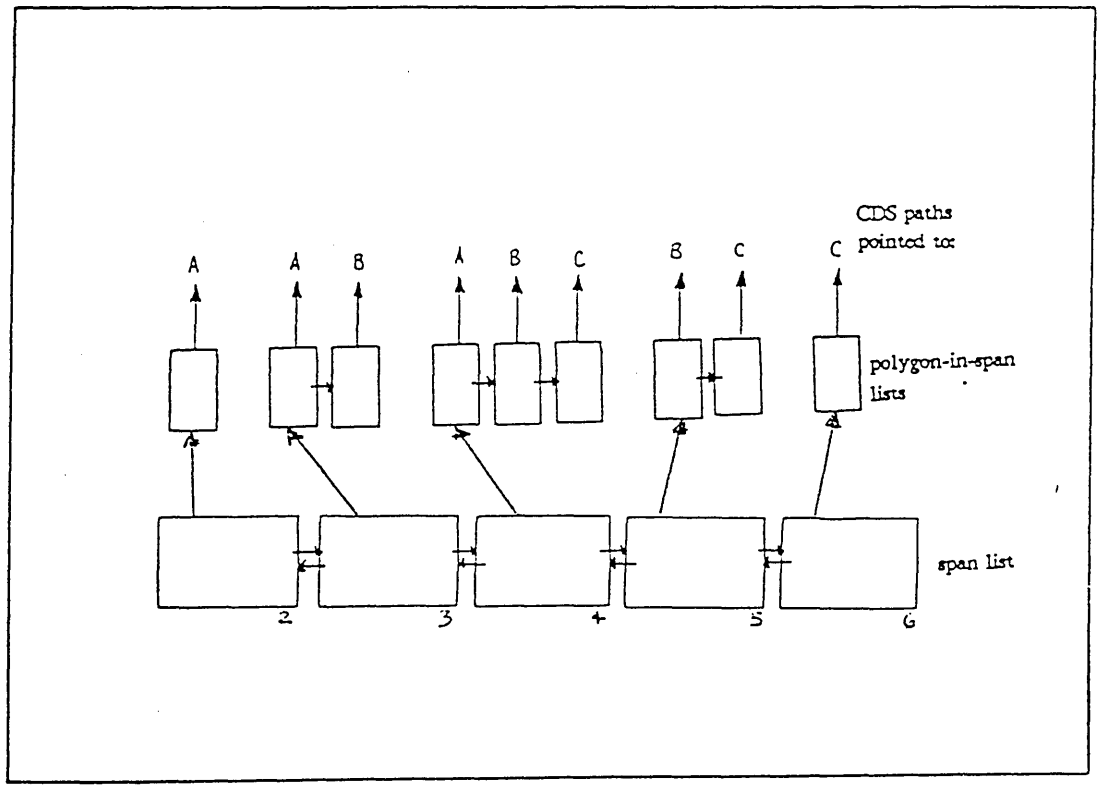


fig. 24(b) : Span list with associated 'polygon in span' sub-lists.

A span node is created for each non-empty span of the scan-line. These are stored as a linked list (see fig. 24(b)) and contain the following information -

start and stop x values for span.
number of primitives active in span.
pointers to first and current nodes in the 'polygon in span' list.
pointers to the next and previous nodes in the span list.

The span list is traversed during the generation of a scan-line, providing access to the visible polygons in the 'polygon in span' lists. There is one 'polygon in span' list for each node in the span list. The next sub-section describes the 'polygon in span' list in more detail.

Each node in the span list requires 5 words of storage. The number of nodes required is dependent on the number of box enclosures active at a particular scan-line and whether they overlap.

4.1.6.1. The 'Polygon in Span' List.

A path list is created for each primitive when it becomes active (see section 4.1.5.1) and is maintained (in order) to reflect polygons active on any particular scan-line. Each node in the 'polygon in span' list points at a path node to provide direct access to a polygon that is potentially visible at a particular pixel within the span. The number of nodes in the 'polygon in span' list will depend on the number of primitives active within the span.

The 'polygon in span' list contains a 'first' pointer to indicate the path node that contains the polygon incident at the first pixel in the span. Initially the required polygon will be found by traversing the path nodes of the active primitive involved. Thereafter it is updated at the end of each scan-line to reflect changes between scan-lines.

The 'polygon in span' list also contains a 'current' pointer to indicate the path node that contains the polygon incident at the current pixel. At the start of a span the 'first' and ' current' pointers point at the same path node. The

pointer to the current path node is updated to reflect changes between pixels, which occur each time the 'closing' edge of a polygon is encountered.

For each pixel in a span the 'polygon in span' list is traversed providing the information required to accomplish hidden-surface elimination. This is discussed in the next section.

The storage requirement for a node in the 'polygon in span' list is $1\frac{1}{2}$words. The number of nodes required is dependent on how many primitives overlap at any particular time.

4.2. Hidden-Surface Elimination.

In the proposed rendering method hidden-surface elimination is achieved using a mixture of conventional scan-line methods and ray tracing techniques. The following provides two different categories of hidden-surface elimination and how they are dealt with -

back–facing polygons :

The back-facing polygons of any primitive are invisible. It is usual for systems using polyhedral approximation schemes to cull such polygons and ignore them during the rendering step. In the hybrid method back-facing polygons are distinguished by their NULL intensity values at their four vertices. They cannot be completely ignored because they may be required by the ray tracing algorithms (see section 4.2.2).

intersecting solids:

Using conventional rendering methods it is normal to compute new boundaries where two or more solids intersect. This is no mean feat, using polyhedral approximation it involves finding and re-defining all the 3-dimensional polygons that are affected by the intersection. This type of hidden-surface elimination is much simpler to achieve using ray tracing techniques. All the ray-polygon intersection points are computed, the intersection point of an existing surface that is nearest to the viewing position provides the polygon that is visible (see section 4.2.2). The surface definitions remain unchanged which could be a distinct advantage for any future updates. The hybrid method does the ray tracing in a similar fashion but delays the computation of intersection points occurring at back-facing polygons until required.

4.2.1. Using Scan-line Methods.

Scan-line methods are only used for hidden-surface elimination where there is only one primitive active within a particular span. No checks for visibility are required within such a span. Using these methods the visible facets in the span are found by accessing path nodes via the associated 'polygon in span' list (see section 4.1.6.1).

All the pixels within the span are generated for front facing facets, back-facing facets are ignored. This is possible because all the primitive surfaces are convex and so will only have one front-facing polygon at any particular pixel. Evaluating a pixel involves calculating its intensity value using the information stored in the path node (see section 4.4).

Adding solids containing concave areas (e.g. torii) to the list of allowable primitive types would involve inserting an extra field in the primitive definitions to enable these solids to be easily identified. Such solids may require two or more path lists to exist simultaneously (see section 4.1.5.1). Areas of the screen containing these solids would always be generated using ray-casting.

4.2.2. Using Ray Tracing Techniques.

Ray tracing techniques are used for hidden-surface elimination where there is more than one primitive active within a particular span. Using these methods pixels are generated individually but coherence is applied between rays passing through adjacent pixels. When ray tracing methods are used the depth and intensity values at the ray-facet intersection points are retrieved from the path nodes.

To find the visible surface at a pixel doing a top downwards traversal of the tree, the rays undergo a classification step (see section 2.2). Generally no coherence between rays is attempted.

In the proposed hybrid method, finding the polygons intersected by the ray is reduced to finding the polygons containing the pixel associated with the ray.

These polygons are found by following all the 'current' path pointers stored in the 'polygon in span' list. It is possible for a ray not to intersect any polygon. If one or more polygons are intersected then all the ray-polygon intersection depths are retrieved from the path nodes to enable the first surface intersected by the ray to be found. The 'nearest' polygon of a primitive 'directly' associated with a ' + ' operand (see section 4.1.1.1) is computed from the depths list. If this primitive is not indirectly associated with a ' - ' or a ' & ' operand then the polygon 'exists' and the ray-polygon intersection point provides the visible surface at the current pixel. Otherwise further work is required to determine an 'existing' polygon visible at the pixel.

If the primitive containing the 'nearest' polygon is directly associated with a '+' operand and is indirectly associated with a ' - ' operand, the path must also be created for its back-facing polygons. This path will supply the depth bound for the primitive. The primitives directly associated with the same '-' operand will also have their depth bounds computed. These bounds will determine whether the '+' primitive contains a hollow concave area or a hole.

If the primitive containing the 'nearest' polygon is indirectly associated with a ' & ' operand the polygon incident on the other primitive(s) involved in the intersection are used to determine the depth.

After the ray associated with the first pixel in a span has been cast, the other rays can use the information of their immediately preceding ray. The 'nearest' polygon will remain so until a new ray intersects an 'existing' polygon that has a smaller depth value than the 'nearest' polygon hit by the previous ray. The 'initial' ray for a span is stored for the duration that the span node exists, and is updated to reflect changes between scan-lines.

4.3. Algorithm for Proposed Hybrid Method.

The following pseudo-code provides the outline for the proposed hybrid rendering algorithm. In this code ordering is done in screen coordinates and 'box' is used as a synonym for 'primitive box enclosure' i. e. the maximum and minimum x and y values in screen coordinates as described in section 2.3. Ordering by y is achieved using the maximum y values of the primitive box enclosures, ordering by x is achieved using the minimum x values. The term block is used to mean a group of adjacent scan-lines where no new primitives are encountered and no currently active primitives become inactive.

```
for   (each primitive in scene)   do
      begin
      compute the surface definition;
      calculate box;
      end;     /* for */
create and order index list in decreasing y order;
set scan-line to top of box of first primitive in index;
set finished to false;
while   ( not finished)   do
      begin
      create/update active primitive list and span list maintaining
                                              increasing x order;
      create path list for newly active primitives and link into
                                              'polygon in span' lists;
      if   (active list empty)   then
            begin
            if   (index list empty) then
                  finished := true ;
            else
                  set scan-line to max y of first node in index list;
            end
      else
            begin
            find last scan-line in current block of scan-lines;
            for   (every scan-line in current block)   do
```

```
        begin
        while   (span list not all processed)   do
            begin
            if   (only one box incident in span)   then
                draw_conven( );
            else
                ray_cast( );
            end;     /* while */
        update path list(s) and 'polygon in span' list(s);
        end;     /* for */
    end;     /* if */
end;     /* while */
```

As a pre-processing step an index is created and ordered. How this is achieved is described in section 4.1.4. The index enables direct access to the primitives at the external nodes of the CSG-tree and provides the rendering routines with the areas of the image where pixels are to be evaluated. Pixels outside these areas are ignored.

Throughout the rendering step entries from the front of the index are 'transferred' into the active primitive list. This is done whenever a box becomes incident on the current scan-line. A node is created for insertion into the active primitive list and is set to point to the primitive stored at the external node previously pointed at by the index node. The new node is inserted into the active primitive list maintaining increasing x order. The index node is no longer required and is deleted. Creating and maintaining the active primitive list is described in section 4.1.5.

The scan-line is split into spans (or segments) according to the x values of the box enclosures pertaining to primitives in the active primitive list. The pixels for each scan-line are generated for one span at a time. If only one box is associated with a span all the pixels within that span are generated using conventional scan-line methods (see section 4.2.1), otherwise ray tracing techniques are applied (see section 4.2.2).

Considering the image shown in figure 23, the index would originally contain pointers to two primitives. The current scan-line would be set to the top of the

sphere's box enclosure. The sphere primitive would be transferred to the active primitive list from the front of the index list. The path list for the newly active primitive would be created and would contain the two nodes associated with the shaded polygons (see fig. 25 (a)). The span list would be created with one node starting and stopping at area A's boundaries (see figure 25 (b)). The current block ends at the top of the cube's box enclosure. Scan-lines in block 1 are generated using conventional scan-line methods. Figure 26 (a) shows the underlying data structures prior to the rendering step. The 'current' in the span/path list points to the same node as the 'first' in the span/path list.

Figure 26 (b) shows the state of the underlying data structure at the scan-line shown in figure 23 and at the start of the scan containing the cube and the sphere (i. e. area B in fig. 25). The first span has been processed and so the 'current' pointer is pointing to polygon E, this will be reset to A after the whole scan-line has been processed. The 'first in span' and the 'current in span' pointers are the same for the two spans that are yet to be processed.

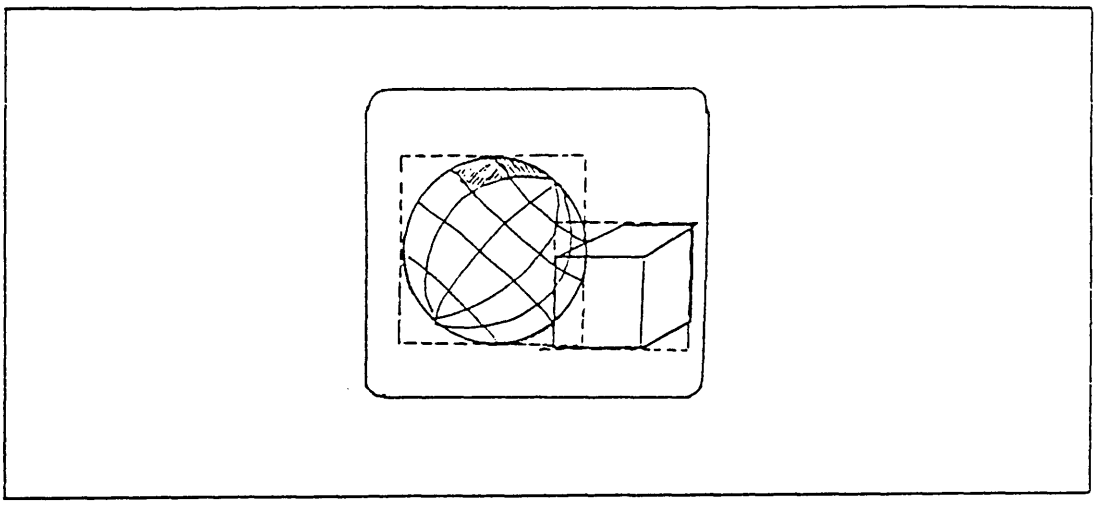


fig. 25 (a) Image with initially active polygons shaded.

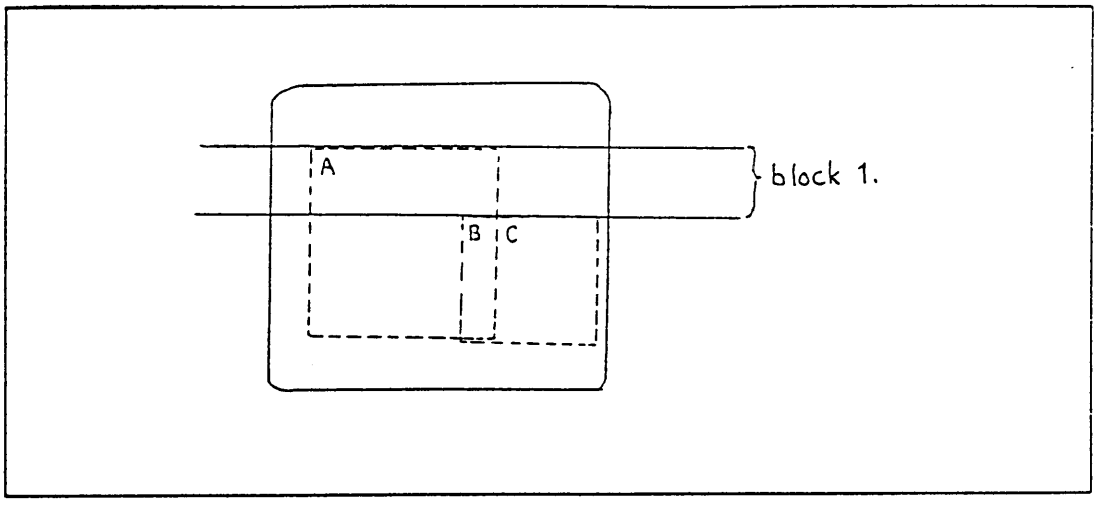


fig. 25 (b) Image showing three areas and block 1.

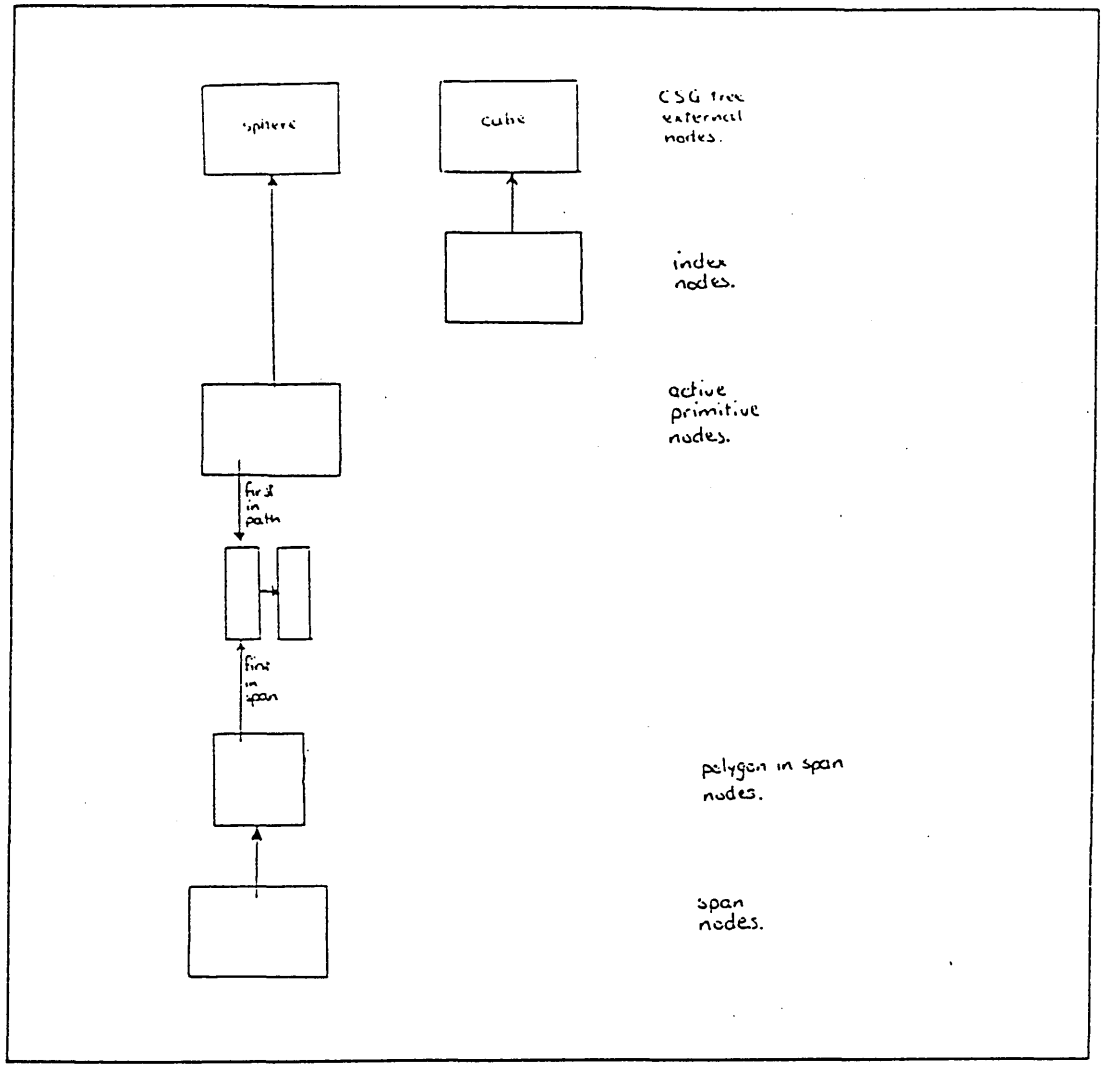


fig. 26 (a) Initial state of underlying data structure.

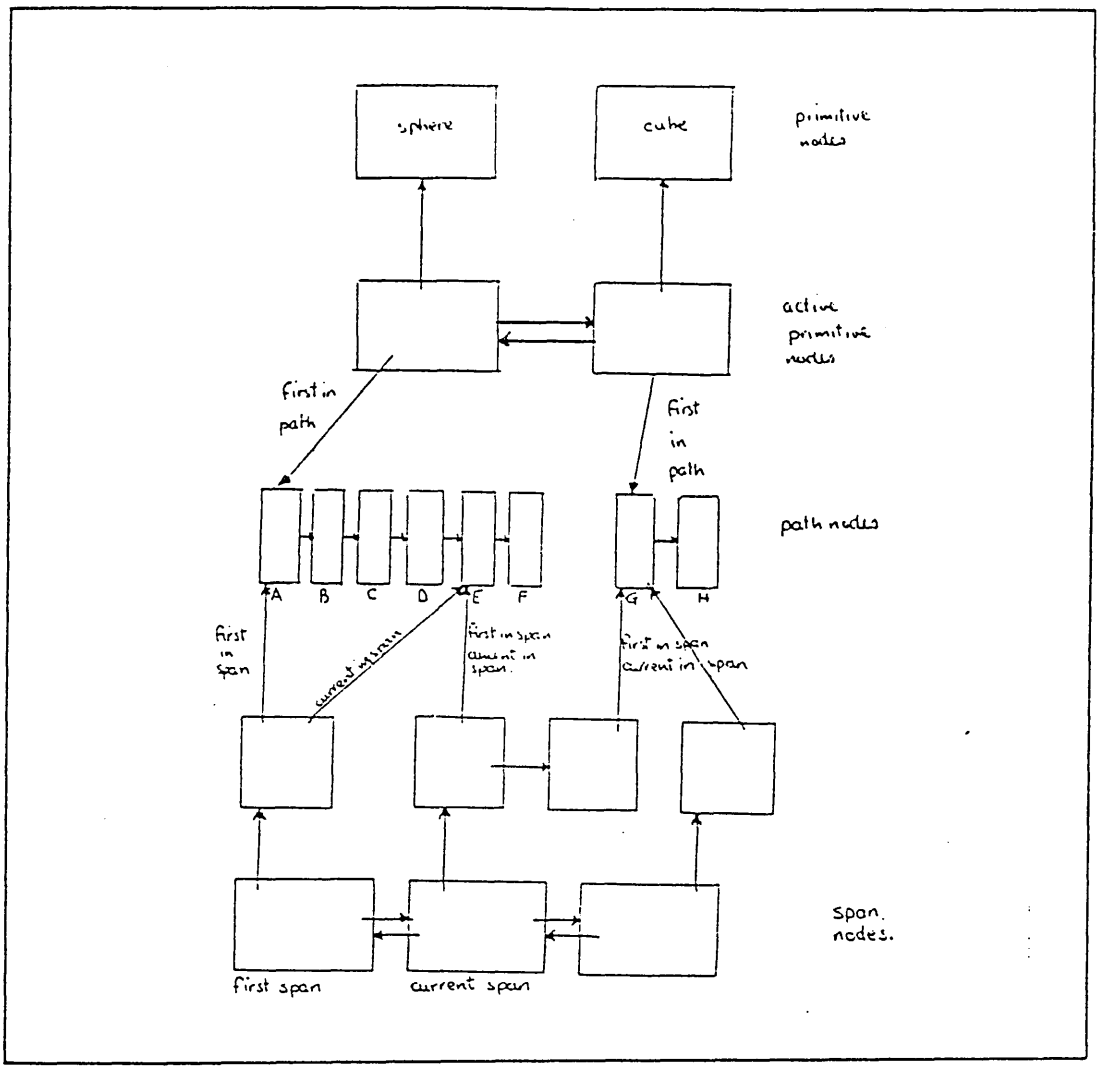


fig. 26 (b) State of underlying data structure at a particular point.

4.4. Shading and Illumination.

Traditionally ray tracing systems compute the intensity value of each pixel individually, each ray being independent of its neighbours. Conventional rendering methods enable smooth shading techniques to be applied, which is generally more efficient because the illumination model is called less frequently. Using the proposed hybrid method both conventional rendering methods and ray tracing techniques take advantage of the efficiency of Gouraud's smooth shading methods by linearly interpolating the intensity values at points on the screen (see section 1.3.2.2). Ray tracing is used purely for hidden-surface removal and does not provide for any sophisticated illumination model.

There are only two edges intersected by any scan-line since all the polygons in the hybrid system are convex quadrilaterals. Linear interpolation is applied to compute the intensity values at the scan-line/edge intersection points and to provide the intensity values at each of the pixels lying between these two points.

CHAPTER 5.

PRESENTATION & ANALYSIS OF RESULTS.

5.1. Comparison with Watkins' Spanning Scan-line Algorithm.

In the proposed hybrid method, data is accessed at two different levels. At the higher level the screen positions of primitives are provided by box enclosures and at the lower level the screen positions are provided by the vertices defining the facets. A feature of Watkins' scan-line hidden surface algorithm, described in section 1.3.1.1, is the initial sorting of polygons. The same concepts are adapted for the hybrid method but are applied at a higher level to primitive box enclosures.

Another feature of Watkins' algorithm is the concept of a span to reduce the number of depth calculations. The boundary of Watkins' span is determined by the end points of the line computed by intersecting the polygons with the scan plane. The spans containing more than one polygon/'scan plane intersection point' require the depth to be computed to determine which polygon is visible. If two of the 'intersection' lines cross within a span, the intersection point is computed and the span subdivided. The number of spans required for any scan-line is dependent on the number of polygons incident at that scan-line and how many of their associated 'intersection' lines cross, this could be very large.

In the proposed hybrid method the concept of a span is used to determine which primitives might be visible at any particular point. Box enclosures determine the extent of the spans rather than polygons which were used in Watkin's method, therefore fewer spans will be required. The boundary of a span is determined by the scan-line/boxes intersection points. Each span is contained in part/whole of one or more box enclosures. No new boxes become active within a span and no active boxes become inactive, which is different from Watkins' concept where depth was also considered. The span is used as a means of fast access to the primitive(s) incident within the span and to determine what rendering technique is to be used for generating the image within the span.

Unlike the polygons dealt with by the spanning scan-line algorithm box enclosures are always rectangular which enables spans to be easily created since minimum and maximum x and y values are simple to compute. There will be fewer box enclosures incident on any scanline than polygons. Using the spanning scan-line algorithm, a list of polygon's edges intersected is maintained. Using the hybrid method, the x-coordinates of the start and finish (i.e. 'box enclosure

edge'/'scan-line intersection points') of each span remains constant while the associated box enclosure(s) is active, therefore there is no need to update the positions of edges intersected. The spanning scan-line algorithm computes the coefficients of the plane equation and the rendering attributes, these are not required using the hybrid method since no image will be generated at this level.

In the proposed hybrid method path nodes provide direct access to polygons active on a particular scan line. A path node identifies two edges that the scan-line intersects, and is 'equivalent' to two of the edge nodes used in the Watkins algorithm. Storing these edges enables new edges to be efficiently computed whenever the end of an active edge is encountered. Using the CDS a polygon has always two edges intersected because each polygon is convex and has exactly four edges, therefore one path per polygon is adequate. Using a general data structure, such as Baumgart's winged-edge, polygons may have any number of edges, therefore several path nodes may be required per polygon, one for each set of enter/exit points. Extra checks would have to be incorporated whenever an active edge became inactive, to handle the additional case when two paths must be combined.

Watkins sets a flag in the edge nodes to indicate if the associated polygon is active on the current scan-line, this is not necessary in the proposed method since only path nodes of polygons incident on the current scan-line actually exist.

In the proposed method, the intensity values at the scan-line/edge intersection points are computed and increments stored for the difference in intensity between scan-lines and between pixels. These are required for Gouraud's smooth shading method.

If two primitives occur within a span, (i.e. two box enclosures overlap) the proposed method employs ray tracing techniques to determine which primitive is visible at any particular pixel in the span, rather than Watkins' method of computing the depths for each polygon span. To enable the ray tracing routines to determine the 'nearest' polygon, depth information is stored at the path nodes. The depths (i.e. z-coordinates) at the scan-line/edge intersection points are computed and stored with their corresponding increments for the difference in depth along edges between scan-lines and between pixels.

At the rendering step, if ray tracing is required, the depth values are computed linearly in the same way as intensity values are when using Gouraud shading. The

depths of polygons associated with a span containing only one primitive are not required and such polygons are generated using conventional rendering methods. The x positions of scan-line/edge intersection points are calculated using linear interpolation in the same way as Watkins' algorithm.

5.2. Comparison with Roth's Ray Tracing Techniques.

The most dramatic increase in efficiency from the method used by Roth is gained by accessing the CSG tree bottom upwards. In section 2.3 there are four requirements listed for generating an image using ray tracing. The first requirement is -

$$(\log_2 q - 1)p^2, \qquad (q > 1) \text{ recursive procedure calls.}$$

Accessing the primitives 'directly' via the span nodes, enables any evaluation to be restricted to only those pixels contained in box enclosures. Therefore no recursive procedure calls are required.

The second requirement is

$$(q - 1)p^2 \text{ classification combines.}$$

These classification combines will generally be reduced by not having to classify some primitives which are either completely 'hidden' or are drawable using conventional methods.

The requirement of computing ray directions is not appropriate because these are no longer required in the new method. The CPU time required to create 250,000 rays (i.e. direction vectors) on a Pyramid computer is 64.8 seconds. However this could be reduced by delaying the creation of rays until they are found to pass through a pixel associated with a box enclosure. Typically objects created by CAD systems will be centred in the screen and may cover at most around 75% of the pixels, therefore 25% of the rays need not be created.

The requirement of ray-surface intersection points is dependent on the method used to represent the primitives. Using ray tracing techniques the objects are commonly represented by an analytic definition to avoid ray tracing numerous polygons. The hybrid method of finding ray-surface intersection points is reduced to a 'ray-in-polygon' test that is carried out in screen coordinates.

Three requirements needed to implement box enclosures in a binary CSG tree are given at the end of section 2.3. The third requirement of computing the

boxes at internal nodes is no longer required, since these boxes are used to direct the top down search for a primitive.

5.3. Comparison with Other Hybrid Approaches.

This section discusses two other hybrid approaches and compares them with the proposed hybrid method.

Jansen [Jansen 1985] exploits CSG coherence by using spatial subdivision techniques combined with CSG list priority and ray tracing algorithms. Using his subdivision techniques the object space is subdivided into cells. Each cell has a bucket into which associated polygons are stored. Prior to the ray tracing step polygons are clipped against cell planes and tested for intersection with other polygons in the same bucket. This pre-processing step is independent of the viewing position. Altering the viewing position only affects the order that the cells are processed. When the viewing position is known cells lying behind or outside the viewing cone are discarded. Cells that are partly inside the viewing window have their polygons clipped. Polygons are then sorted within each bucket.

The ray tracing step initially intersects rays with cells to reduce the number of polygons that have to be tested for intersection. Only polygons in the buckets of the cells intersected then have to be considered. Coherence is achieved using a CSG list structure, a C-buffer and an item buffer. The CSG list structure stores the various sequences of primitives that are intersected by rays during the CSG tree traversal. The C-buffer contains a pointer to an element in the list structure for every ray. The item buffer uses the C-buffer to set up a pointer to the visible polygon for every pixel. A z-buffer is also used to avoid errors in the treatment of coincident facets and sorting errors.

The memory requirements for the three buffers are very large, the proposed hybrid method does not use buffers at all. Using Jansen's method, as a pre-processing step, 'object space' polygons are placed in cell buckets, then compared for overlap and clipped before sorting. Therefore the number of polygons actually stored is greater than the original number of polygons representing the complex object. The proposed method works in the image space, the only preprocessing required is to order the primitive box enclosures by maximum y-values. The overlapping and clipping of polygons is achieved automatically by the ray casting algorithms (without splitting polygons). The sorting step is not required either since the order of polygons is already inherent in the CDS's.

Timings are given by Jansen for the CSG list priority and the ray tracing algorithms for generating the image of two cylinders. The conclusion made is that this method shows no improvement on the reported scanline algorithms [Atherton 1983, Crocker 1984]. Therefore no timing comparisons were made for this method, a comparison between the performance of the scan-line algorithm given by Atherton and the performance of the hybrid method is given in section 5.4.

Sears and Middleditch [Sears 1984] adopt a similar approach to the proposed hybrid method in that the screen is divided into sections according to the primitives' box enclosures (in screen coordinates). However, they consider the whole screen. Edges of the box enclosures are extended to the edges of the screen, dividing the whole screen into rectangular regions. A pseudo CSG tree is constructed for each region, containing associated primitives and their Boolean operands.

At the ray tracing step, coherence is achieved by restricting ray/primitive intersection tests to the CSG tree associated with the region of the screen that the ray's pixel lies in. In comparison, the hybrid method uses 'poly-in-span' nodes to indicate the primitives active within a span of a particular scan-line.

Sears and Middleditch state results are significantly inferior to Atherton's results, therefore no comparison is made between their results and the results of the proposed hybrid method.

5.4. Performance Measurements.

This section analyses the results achieved using the proposed hybrid rendering technique described in chapter 4.

Generating the image of a cube with a box enclosure containing approximately 17,500 pixels takes 3 minutes 55 seconds using the scan-line rendering methods. Generating the same image using the ray tracing techniques employed in the hybrid method requires 11 minutes 55 seconds, which is almost four times as long. This time would be increased dramatically if rays were traced through all the pixels in the screen and intersection tests done to all the facets of the cube. Both methods above sent Tektronix graphics characters to a Cifer terminal, and both timings were done running the system on a VAX 11/780 and did not include the data creation step. The shading was done using a 4 x 4 array to provide 17 different intensity values.

Previous systems employing ray tracing techniques considered all pixels in the screen. The proposed hybrid method considers only pixels that lie within box enclosures, which is significantly faster (see section 3.1). The ray-surface intersection test is done in screen coordinates and so is reduced to being the ray-in-polygon test. In earlier ray tracing algorithms no coherence properties were employed during ray tracing, each ray was computed independently. The hybrid method employs scan-line coherence and span-coherence properties to increase efficiency, each ray using information from an 'adjoining' ray whenever possible. Therefore the timing for ray tracing given above would be considerably higher if normal ray-tracing methods were employed.

The efficiency gained from interchanging between conventional rendering methods and ray tracing techniques rather than using full ray tracing is dependent on the image being generated. Let G be the ratio denoting the number of pixels that are enclosed by only one box enclosure compared to all pixels enclosed. From the timing above, the approximate time to generate the image using the hybrid method compared with pure ray tracing is given by -

$$\frac{1}{4} G \times \text{ray-tracing-time.}$$

This ray tracing time does not include any hidden-surface computation other than the removal of back-facing polygons, since only one cube is involved. This time will increase with the number of overlapping box enclosures since the depth and intensity values at each ray-primitive entrance point must be updated for each new ray generated.

A Pyramid computer, linked to a Vectrix display device, was used to compare the time required to generate the image of a sphere. The sphere was approximated by a polyhedron consisting of 140 facets with front-facing facets shaded by 150 different shading intensities. The sphere covered an area of approximately 35,000 pixels. Normally such high precision would not be required, the sphere was created for testing purposes. Using conventional methods and shading each facet individually took 8mins. 20 secs. to generate the image. Using ray tracing techniques with box enclosures and the primitives stored in the compressed data structure (i.e. ray-in-quadrilateral tests) took 15mins 36secs. Taking full advantage of the coherence properties and the direct access facilities to required facets as provided in the hybrid method, enabled both these timings to be substantially reduced. Shading facets in scan-line order required 4mins. 54secs using conventional techniques. Ray tracing required 10mins. 25secs to render the same image of the sphere.

Photograph 1 shows a wireframe image of three cubes. Photograph 2 shows the same cubes generated as a shaded image, the areas coloured blue show the pixels that were generated using ray tracing techniques. Photograph 3 shows the cubes generated as a shaded image with the true intensity values drawn by the ray tracing algorithm. The wire frame image (photograph 1) took less than 2 seconds to draw. The shaded image of the three cubes shown in photograph 3, required only 18 secs. There were only 17 different intensity values allowed.

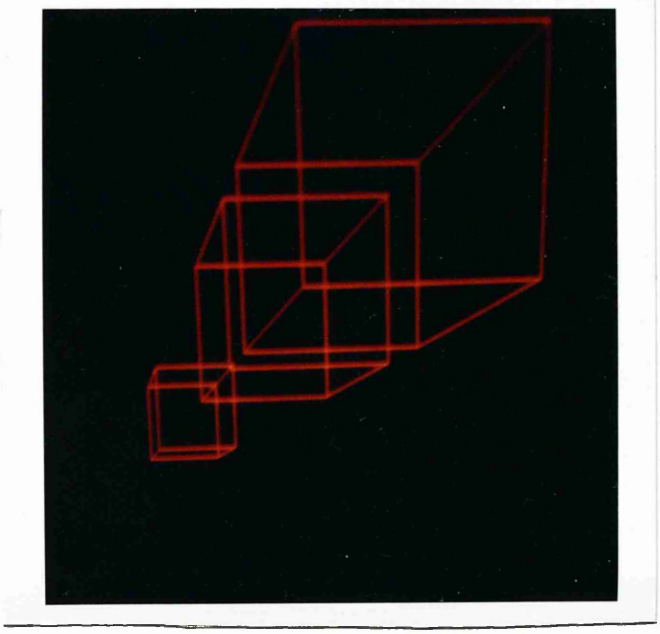
Photographs 4 and 5 show a wireframe and a shaded image (respectively) of a sphere and a cube combined. The wire-frame image in photograph 4 required 6 seconds. The shaded image in photograph 5 required only 23secs, as before only 17 different intensity values were allowed.

5.5. Conclusion.

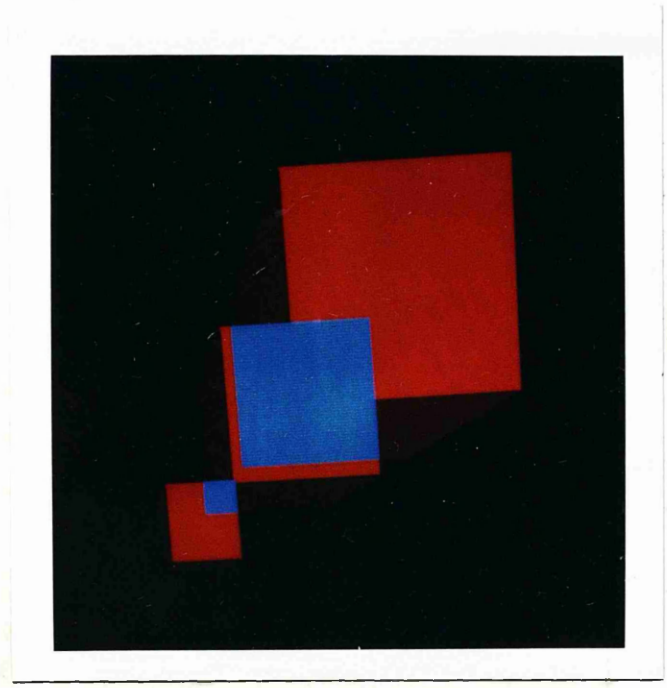
The aim of this thesis is to generate images, striking the balance between realism and processing time, for engineering type applications. The tests done so far indicate that the proposed hybrid method comes nearer to achieving this aim than any of the ray tracing methods or hybrid methods previously available.

If an application requires photograph like images, the hybrid system could be used for quick generation of images, for previewing in a similar fashion to the way that wire frame images are used. The bottom upwards tree traversal is not dependent on the method used to determine ray-surfaces intersection points. Therefore, for final images surfaces could be defined analytically and a sophisticated illumination model used with this traversal. This still increases efficiency over most of the previous ray tracing methods because it restricts the evaluation of pixels to those lying within box enclosures and provides direct access to the primitives. The additional data structures would no longer include path nodes for this final image, since these are dependent on the polyhedral approximation representation.

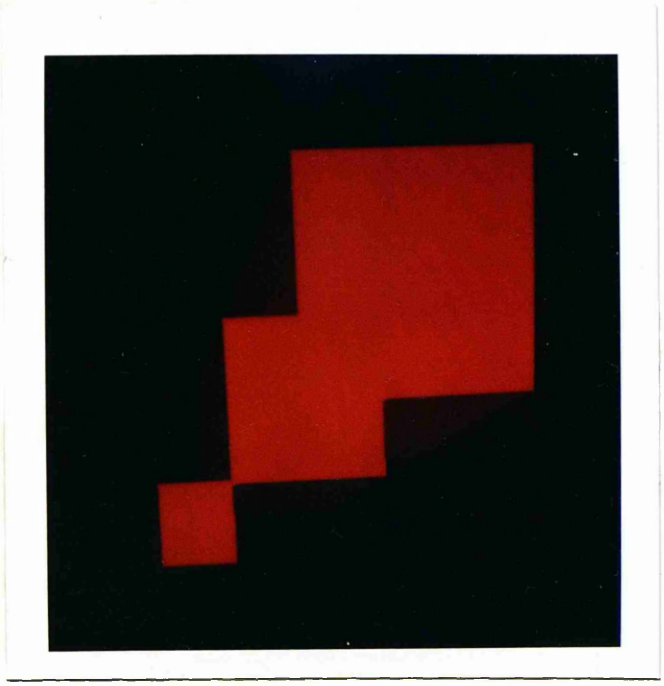
The proposed hybrid method does not include any anti-aliasing, since speed of generation had to be balanced with realism. The resulting image would be improved if a minimal amount of anti-aliasing was done. Polygons could be anti-aliased at their edges. These edges are easily obtainable so it would be simple to over sample and add some pre-filtering method around these points.



Photograph 1 : Wire-frame image of three cubes.



Photograph 2 : Shaded image of three cubes with ray traced pixels coloured blue.



Photograph 3 : Shaded image of three cubes.

REFERENCES.

Appel 1967.

"The Notion of Quantitative Invisibility and the Machine Rendering of Solids", A. Appel, Proc.ACM National Conf., 1967, pp.387-393.

Appel 1968.

"Some Techniques for Shading Machine Renderings of Solids", A. Appel, AFIPS Proceedings, Vol.32, Spring Joint Comp. Conf. 1968, pp.37-45.

Atherton 1983.

"A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry", P.R. Atherton, Computer Graphics, Vol.17, No. 3, July 1983, pp.73-82.

Baumgart 1975.

"A Polyhedron Representation for Computer Vision", B.G. Baumgart, Proceedings AFIPS National Computer Conf. 1975, pp.589-596.

Blinn 1977.

"Models of Light Reflection for Computer Synthesized Pictures", J.F. Blinn, Computer Graphics, Vol.11, No.2, 1977, pp.192-198.

Blinn 1982.

"A Generalization of Algebraic Surface Drawing", J.F. Blinn, ACM Trans. on Graphics, Vol.1, No.3, July 1982, pp.236-256.

Bouville 1984.

"Generating High Quality Pictures by Ray Tracing", C. Bouville, J.L. Dubois & I. Marchal, Eurographics Conf. Proc., 1984, pp.161-177.

Boyse 1982.

"GMSolid: Interactive Modeling for Design and Analysis of Solids", J.W. Boyse & J.E. Gilchrist, IEEE Computer Graphics & Appl., Vol.2, No.2, March 1982, pp.27-40.

Bui-Tuong 1975.

"Illumination for Computer-Generated Pictures", Phong Bui-Tuong, CACM, Vol.18, No.6, June 1975, pp.311-317.

Catmull 1975.

"Computer Display of Curved Surfaces", E. Catmull, Proc. IEEE Conf. Computer Graphics Pattern Recognition Data Struct., May 1975, p.11.

Clark 1976.

"Hierarchical Geometric Models for Visible Surface Algorithms", J.H. Clark, Comm. ACM, Vol.19, No.10, 1976, pp.547-554.

Clark 1980.

"A VLSI Geometry Processor for Graphics", J.H. Clark, Computer, Vol.13, No.7, July 1980, pp.59-68.

Clark 1982.

"The Geometry Engine: A VLSI Geometry System for Graphics", J.H. Clark, Computer Graphics, Vol.16, No.3, 1982, pp.127-133.

Conway 1988.

"The Isoluminance Contour Model", D.M. Conway, M.S. Cottingham, Proceedings of AUSGRAPH 88, Melbourne, July 4-8, 1988, pp.43-50.

Cook 1982.

"A Reflectance Model for Computer Graphics", R.L. Cook & K.E. Torrance, ACM Trans. on Graphics, Vol.1, No.1, Jan 1982, pp.7-24.

Cook 1984.

"Distributed Ray Tracing", R.L. Cook, T. Porter & L. Carpenter, Computer Graphics, Vol.18, No.3, July 1984, pp.137-145.

Cottingham 1981.

"Movies : Chapter 3", M.S. Cottingham, Honours Thesis, Department of Computing Science, University of Glasgow, 1981.

Cottingham 1985.

"A Compressed Data Structure for Surface Representation", M.S. Cottingham, Computer Graphics Forum, Vol.4, No.3, September 1985, pp.217-228.

Cottingham 1987.

"Compressed Data Structure for Rotational Sweep Method", M.S. Cottingham, AUSGRAPH 87 Conference Proceedings, 4-8 May 1987.

Cottingham 1988a.

"Computers in Society", M.S. Cottingham, L.M. Goldschlager, A.J. Maeder, R.T. Worley, ANZAAS Centenary Congress, Sydney, May 16-20, 1988.

Cottingham 1988b.

"Pseudo Ordering of CSG-trees", M.S. Cottingham, Eurographics 88 Conference, Nice, France, September 12-16, 1988 (to appear).

Crocker 1984.

"Invisibility Coherence for Faster Scan-line Hidden Surface Algorithms", Computer Graphics, Vol.18, No.3, July 1984.

Crow 1981.

"A Comparison of Antialiasing Techniques", F.C. Crow, IEEE Computer Graphics & Appl., Vol.1, No.1, Jan.1981, pp.40-49.

Dippe 1984.

"An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", M. Dippe and J. Swensen, Computer Graphics, Vol.18, No.3, 1984, pp.149-158.

Duff 1983.

"Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays", T. Duff, Computer Graphics, Vol.17, 1983, pp.73-82.

Fiume 1983.

"A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", E. Fiume, A. Fournier & L. Rudolph, Computer Graphics, Vol.17, No.3, 1983, pp.141-150.

Foley 1982.

"Fundamentals of Interactive Computer Graphics", J.D. Foley & A. van Dam, Addison-Wesley Publishing Co., 1982.

Fuchs 1982.

"Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System", H. Fuchs, J. Poulton, A. Paeth, A. Bell, Proc. of the 1982 MIT Conf. on Advanced Research in VLSI, pp.137-146.

Fuchs 1985.

"Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes", H. Fuchs, J. Goldfeather, J.P. Hultquist, S. Spach, J.D. Austin, F.P. Brooks, J.G. Eyles and J. Poulton, Computer Graphics, Vol.19, No.3, 1985, pp.111-120.

Fuchs 1986.

"Quadratic Surface Rendering on a Logic-Enhanced Frame-Buffer Memory System", H. Fuchs and J. Goldfeather, IEEE Computer Graphics & Appl., Vol. 6, No. 1, Jan.1986, pp.48-59.

Fujimoto 1986.

"ARTS: Accelerated Ray-Tracing System", A. Fujimoto, T. Tanaka and K. Iwata, IEEE Computer Graphics & Appl., Vol. 6, No. 4, April,1986, pp.16-26.

Goldstein 1971.

"3-D Visual Simulation", R.A. Goldstein & R. Nagel, Simulation, Jan1971, pp.25-31.

Goldstein 1979.

"3D Modelling with the Synthavision System", R. Goldstein, First Annual Conf. on Computer Graphics in CAD/CAM Systems, M.I.T., April 1979, pp.244-247.

Gouraud 1971.

"Computer Display of Curved Surfaces", H. Gouraud, Univ of Utah Comp. Science Dept., Utec-CSc-71-113, June 1971, NTIS AD-762 018.

Gupta 1981.

"A VLSI Architecture for Updating Raster-Scan Displays", S. Gupta, R.F. Sproull & I.E. Sutherland, Computer Graphics, Vol.15, No.3, Aug.1981, pp.71-78.

Hall 1983.

"A Testbed for Realistic Image Synthesis", R.A. Hall and D.P. Greenberg, IEEE Computer Graphics & Appl., Vol.3, No.8, Nov.1983, pp.10-20.

Hanrahan 1983.

"Ray Tracing Algebraic Surfaces", P. Hanrahan, Computer Graphics, Vol.17, No.3, July 1983, pp.83-90.

Heckbert 1984.

"Beam Tracing Polygonal Objects", P.S. Heckbert & P. Hanrahan, Computer Graphics, Vol.18, No.3, July 1984, pp.119-127.

Hillyard 1982.

"The Build Group of Solid Modelers", R. Hillyard, IEEE Computer Graphics & Appl., Vol.2, No.2, March 1982, pp.43-52.

Jansen 1985.

"A CSG List Priority Hidden Surface Algorithm", F.W. Jansen, Eurographics Conf. Proc., 1985, pp.51-62.

Kajiya 1982.

"Ray Tracing Parametric Patches", J.T. Kajiya, Computer Graphics, Vol.16, No.3, 1982, pp.245-254.

Kajiya 1983.

"New Techniques for Ray-Tracing Procedurally Defined Objects", J.T. Kajiya, Computer Graphics, Vol.17, No.3, 1983, pp.91-102.

Kajiya 1984.

"Ray Tracing Volume Densities", J.T. Kajiya & B.P. Von Herzen, Computer Graphics, Vol.18, No.3, July 1984, pp.165-174.

Kay 1979.

"Transparency, Refraction, and Ray Tracing for Computer Synthesized Images", D.S. Kay, Master's Thesis, Cornell Univ., Jan 1979.

Kinnucan 1983.

"Solid Modelers Make the Scene", P. Kinnucan, High Technology, Vol.2, No.4, pp.38-44.

Laning 1979.

"Capabilities of the SHAPES System for Computer Aided Mechanical Design", J.H. Laning and S.J. Madden, Proc. First Ann. Conf. Computer Graphics in CAD/CAM Systems, Cambridge, Mass., April 1979, pp.223-231.

Lerner 1981.

"Fast Graphics Use Parallel Techniques", E.J. Lerner, IEEE Spectrum, Vol.18, No.3, March 1981, pp.34-38.

Max 1981.

"Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset", N. Max, Computer Graphics, Vol.15, No.3, August 1981, pp.317-324.

Myers 1982.

"An Industrial Perspective on Solid Modelling", W. Myers, IEEE Computer Graphics & Appl., Vol. 2, No. 2, March 1982, pp.86-97.

Newman 1973.

"Principles of Interactive Computer Graphics", W.M. Newman & R.F. Sproull, 1st Edition, McGraw-Hill, 1973.

Newman 1979.

"Principles of Interactive Computer Graphics", W.M. Newman & R.F. Sproull, 2nd Edition, McGraw-Hill, 1979.

Okino 1973.

N. Okino, Y. Kakazu and H. Kubo, "TIPS-1: Technical Information Processing System for Computer-Aided Design, Drawing and Manufacturing", Computer Languages for Numerical Control, J. Hatvany ed., North-Holland Pub. Co., Amsterdam, 1973, pp.141-150.

Requicha 1980.

"Representations for Rigid Solids: Theory, Methods, and Systems", A.A.G. Requicha, ACM Computing Surveys, Vol. 12, No.4, Dec. 1980, pp.437-464.

Requicha 1982.

"Solid Modelling: A Historical Summary and Contemporary Assessment", A.A.G. Requicha & H.B. Voelcker, IEEE Computer Graphics & Appl., Vol.2, No.2, March 1982, pp.9-24.

Requicha 1983.

"Solid Modelling : Current Status and Research Directions", A.A.G. Requicha & H.B. Voelcker, IEEE Computer Graphics & Appl., Vol. 3, No. 7, pp.25-37.

Rogers 1985.

"Procedural Elements for Computer Graphics", D.F. Rogers, McGraw-Hill, 1985.

Roth 1982.

"Ray Casting for Modeling Solids", S.D. Roth, Computer Graphics and Image Processing, 18, 1982, pp.109-144.

Sears 1984.

"Set-Theoretic Volume Model Evaluation and Picture-Plane Coherence", K.H. Sears, IEEE Computer Graphics & Appl., Vol.4, No.3, March 1984, pp.41-46.

Sederberg 1984.

"Ray Tracing of Steiner Patches", T.W. Sederberg & D.C. Anderson, Computer Graphics, Vol.18, No.3, July 1984, pp.159-164.

Sorensen 1982.

"Tronic Imagery", P. Sorensen, Byte, Nov. 1982, pp.49-74.

Steinberg 1984.

"A Smooth Surface Based on Biquadratic Patches", H.A. Steinberg, IEEE Computer Graphics & Appl., Vol.4, No.9, Sept. 1984, pp.20-23.

Sutherland 1974.

"A Characterization of Ten Hidden-Surface Algorithms", I.E. Sutherland, R.F. Sproull & R.A. Schumacker, Computing Surveys, Vol.6, No.1, 1974, pp.1-55.

Tamminen 1984.

"Ray-casting and Block Model Conversion Using a Spatial Index", M. Tamminen et al, CAD, Vol.16, No.4, 1984, pp.203-208.

Thomas 1984.

"Synthetic Image Generation", A.L. Thomas, University Computing 1984, Vol.6, No.3, Winter 1984, pp.148-160.

Torrance 1967.

"Theory for Off-Specular Reflection from Roughened Surfaces", K.E. Torrance & E.M. Sparrow, Journal of the Optical Society of America, Vol.57, 1967, pp.1105-1114.

Toth 1985.

"On Ray Tracing Parametric Surfaces", D.L. Toth, Computer Graphics, Vol.19, No.3, 1985, pp.171-179.

Voelcker 1978.

"The PADL-1.0/2 System for Defining and Displaying Solid Objects", H.B. Voelcker, A.A.G. Requicha, E.E. Hartquist, W.B. Fisher, J. Metzger, R.B. Tilove, N.K. Birrell, W.A. Hunt, G.T. Armstrong, T.F. Check, R. Moote & J. McSweeney, Computer Graphics (Proc. Siggraph 1978), Vol.12, No.3, Aug1978, pp.257-263.

Walker 1985.

"The Transputer", P. Walker, Byte, Vol.10, No.5, May 1985, pp.219-235.

Warnock 1969.

"A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures", Univ. of Utah Computer Science Dept., Rep. TR4-15, June 1969, NTIS AD 753671.

Watkins 1970.

"A Real-Time Visible Surface Algorithm", University of Utah, Computer Science Dept. Tech. Report UTEC-CSC-70-101, June 1970, NTIS AD762 004.

Weghorst 1984.

"Improved Computational Methods for Ray Tracing", H. Weghorst, G. Hooper and D. Greenberg, ACM Trans. on Graphics, Vol.3, No.1, Jan.1984, pp.52-69.

Whitted 1980.

"An Improved Illumination Model for Shaded Displays", T. Whitted, Comm. ACM, Vol.23, No.6, 1980, pp.343-349.

Wijk 1984a.

"Two Methods for Improving the Efficiency of Ray Casting in Solid Modelling", J.J. van Wijk, W.F. Bronsvoort, F.W. Jansen, CAD, Vol.16, No.1, 1984, pp.51-55.

Wijk 1984b.

"Ray Tracing Objects Defined by Sweeping a Sphere", J.J. van Wijk, Eurographics 1984 Conf. Proc., pp.73-82.

Wijk 1984c.

"Ray Tracing Objects Defined by Sweeping Planar Cubic Splines", J.J. van Wijk, ACM Trans. on Graphics, Vol.3, No.3, July 1984, pp.223-237.

Woodwark 1982.

"Reducing the Effect of Complexity on Volume Model Evaluation", J.R. Woodwark & K.M. Quinlan, CAD, Vol.14, No.2, 1982, pp.89-95.

Yamaguchi 1984.

"A Unified Algorithm for Boolean Shape Operations", F. Yamaguchi & T. Tokieda, IEEE Computer Graphics & Appl., Vol.4, No.6, June 1984, pp.24-37.

APPENDIX A : The DIAMOND System.

1. The DIAMOND System.

The DIAMOND (Dynamic Integrated Algorithm for Manifesting Objects Numerically Defined) system is an interactive menu-driven system for the creation and rendering of objects that are constructed using CSG methods. It is written in the C programming language which was chosen for its dynamic array capability and its close integration with the UNIX operating system that shares the same language.

Unfortunately there was no funding available at the time of this research to purchase a suitable CAD package that would supply the data required to represent the CSG-tree and to represent the surfaces for a range of primitive types. Therefore the DIAMOND system includes several data input and creation routines as well as the implementation of the proposed method. The principal components of the system have already been described in preceding chapters. The following table provides a list of the methods described along with the section in which they appear.

Table of Modelling and **Rendering Methods Adopted.**

Method Adopted.	Section
Solid Modelling - CSG representation, primitive templates and transformation matrices	1.2
Polyhedral approximation	1.2.1.2
Compressed Data Structure	1.2.1.2
Adapted scan-line and span coherence properties	1.3.1
Hidden surface removal	1.3.1
Basic illumination model	1.3.2.1
Gouraud's smooth shading techniques	1.3.2.2
Box enclosures	2.3
Additional structures to provide fast access to primitives and polygons	4.1
Algorithm for proposed hybrid method	4.3

1.1. Interacting with DIAMOND.

This section summarises how DIAMOND interacts with the user. Only the main menu and the "choose primitive" menu are given, since the data storage and data access routines are the most important features in this research. The system uses default values for as many variables as possible e.g. viewing position, light position and photometry information for the primitives. These provide the user with realistic values for each variable and enables a solid to be rendered with a minimum of input.

The first 'main' menu to appear on the screen is -

```
What do you want to do?

(for efficiency options 1, 2 and 5 should be chosen first!!!)

1.  change light variables.
2.  change viewing position.
3.  create a solid object.
4.  change existing solid object.
5.  change world coordinate system.
6.  draw scene - wire frame format.
7.  draw scene - shaded picture format.
8.  read data from macro file.
9.  stop writing to macro file.
10. exit from DIAMOND.
Choose one -
```

Whatever the choice the user is prompted for any further information required by the system. For example, if option 3 is chosen a prompt appears for the name of the complex solid to be created. After the name is input, another menu appears -

What primitive do you require?
1. cube.
2. cylinder.
3. cone.
4. sphere.
Choose one -

The system prompts the user for any parameters required for the primitive chosen. For a cube these parameters are the scene position of the bottom left corner of the front face and the dimensions in the x, y and z directions. For a sphere these parameters are the position of the centre and the radius. These parameters are used by DIAMOND to create a 4×4 transformation matrix for the primitive. This matrix is used to transform the coordinates of the vertices defining the appropriate template primitive. This transformation occurs immediately after all the information about a single primitive has been input. When the primitive has been completed another menu appears and prompts for the user to either finish the input session or select an appropriate Boolean operand to connect the next primitive to be input with the existing primitive(s). If the user indicates input is complete the main menu is redisplayed. If a Boolean operand is displayed the 'choose primitive' menu is redisplayed. This loop continues until the user finishes the input session.

The user may require a record of the inputs for any particular object/scene. The hybrid system provides a macro file facility; every character input is stored in a file named by the user. This is useful as a debugging/updating tool and can be used to check for typing errors, for correcting geometric errors and for expanding a complex solid. It also guarantees that the same scene can be re-generated as often as required with very little effort.

After the input for the scene is complete the 'main' menu is re-displayed to enable the user to render the image of the scene that has just been created. There are two choices available, '6' for a wire-frame image for fast generation of the scene, and '7' for shaded picture format.

The proposed hybrid algorithm is implemented and is used for shaded picture generation and forms the main part of the rendering routines. The wire frame images are generated using conventional methods. The additional data structures referred to in section 4.1 are created/updated whenever required. Hidden

surface elimination is achieved using conventional scan-line methods for removal of back-facing facets or ray tracing techniques as discussed in section 4.2. Gouraud's smooth shading technique is employed for all the shading calculations.

APPENDIX B : A COMPRESSED DATA STRUCTURE.

The following paper was published as a Departmental Reasearch Report No. CSC/85/R7 by the Department of Computing Science, University of Glasgow in 1985. It was also published in "Computer Graphics Forum", Vol. 4, No. 3 in September 1985.

A Compressed Data Structure for Surface Representation.

Marion S. Cottingham

Department of Computing Science,
University of Glasgow.

ABSTRACT

A standard method of simplifying the task of obtaining a shaded image of a solid object is to represent it by a polyhedron. Another method is to use sculptured surface modelling which represents surfaces by collections of surface patches. Using either method the surfaces can be approximated by polygonal facets, which are simple to shade according to photometry information.

To obtain a smooth image in regions of high curvature, the surface would typically be required to have hundreds or thousands of facets. Because of the many facets involved, it is extremely important that geometrical and topological information is stored in an efficient manner. This information must include all that is required for an unambiguous representation of the solid(s) in question.

The compressed data structure (CDS) is suitable for this purpose, and is capable of defining most surfaces. The structure is intended to minimize the amount of data stored, with as much information as possible being implied. The CDS can be easily generated knowing the order of the vertices defining the surface.

Keywords : data structure, surface modelling,
constructive solid geometry, ray casting.

1. Surface Representation Required by the CDS.

A surface can be approximated by a collection of facets, which are bounded by edges and vertices. Edges are straight line segments and vertices are normally defined in the Cartesian coordinate system, with the x, y and z-coordinates being sufficient to represent a 3-D point. Each facet is surrounded by other facets and may or may not be planar. Topological information is required to show how these facets, edges and vertices are connected.

The CDS is suitable for representing surfaces of polyhedra that can be split into rows and columns of quadrilateral facets. However there are some polyhedra that cannot be fully represented in this way. For example polyhedra approximating spheres, cones and cylinders contain anomalous triangular facets around the points where the axis of symmetry intersects with the surface (i.e. the poles). Using the CDS, the above solids are represented as far as possible by quadrilateral facets, and the triangular facets are dealt with as special cases (see section 5).

Anomalous (i.e. non-quadrilateral) facets occurring on the surfaces of complex solids can be easily identified by considering the number of edges meeting at each vertex in a simplified polyhedral version of the solid [Forrest 78] (see fig. 1(a)). These regions occur when the number of edges incident at one vertex is other than four (see fig. 1(b)). The surface of such a solid can be split into a collection of smaller surfaces, each having no anomalous facets. Figure 2 shows a surface being split into three sections. Each section would be stored in a separate CDS and the anomalous facets would be handled by their connections.

Another method of representing a surface is by sculptured surface modelling in which a mathematical model of surface patches such as Coons, Bezier or B-splines is used. The mathematics involved results in patches being nearly always four-sided. Surfaces are represented by a collection of patches stored in rectangular arrays. The patches that are not four sided lie in the anomalous regions which occur under the same circumstances as the anomalous facets mentioned above. This method has the advantage of an improvement in contouring and shading, but the associated algorithms are less efficient than polygon based algorithms due to the non-linear mathematics involved [Clark 76]. However, after a solid has been constructed these patches may be transformed into polygons and stored in CDS's.

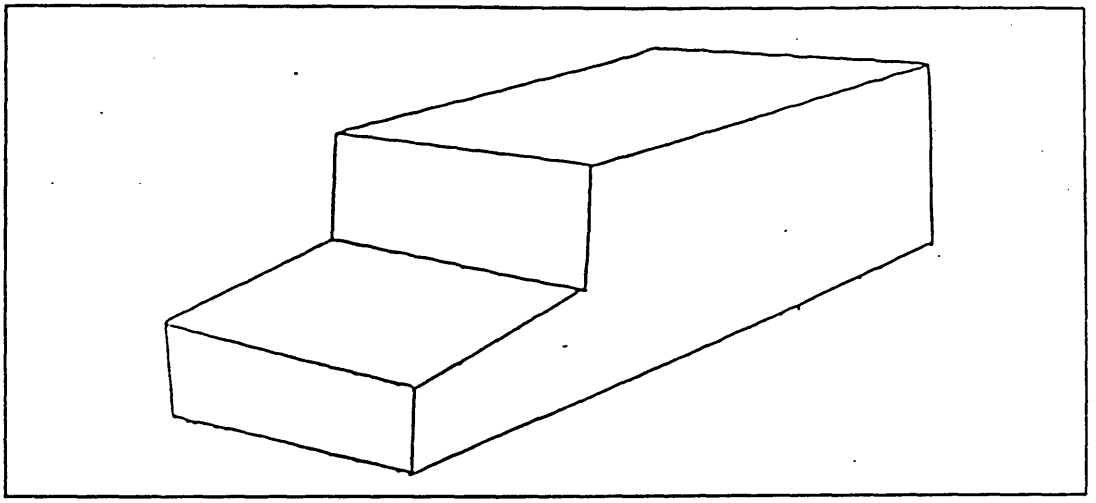


fig. 1 (a) Simplified polyhedral version of solid.

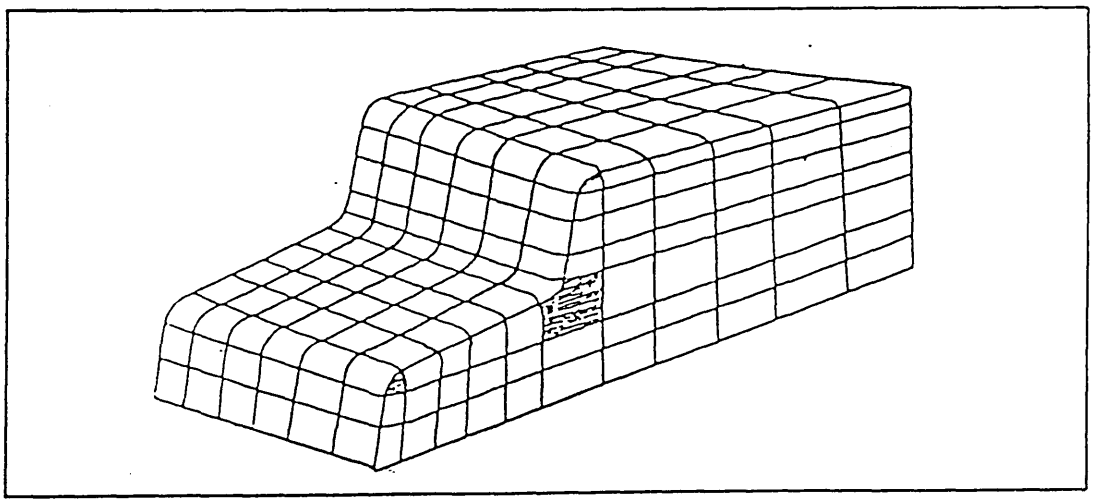


fig. 1 (b) Surface with anomalous facets shaded.

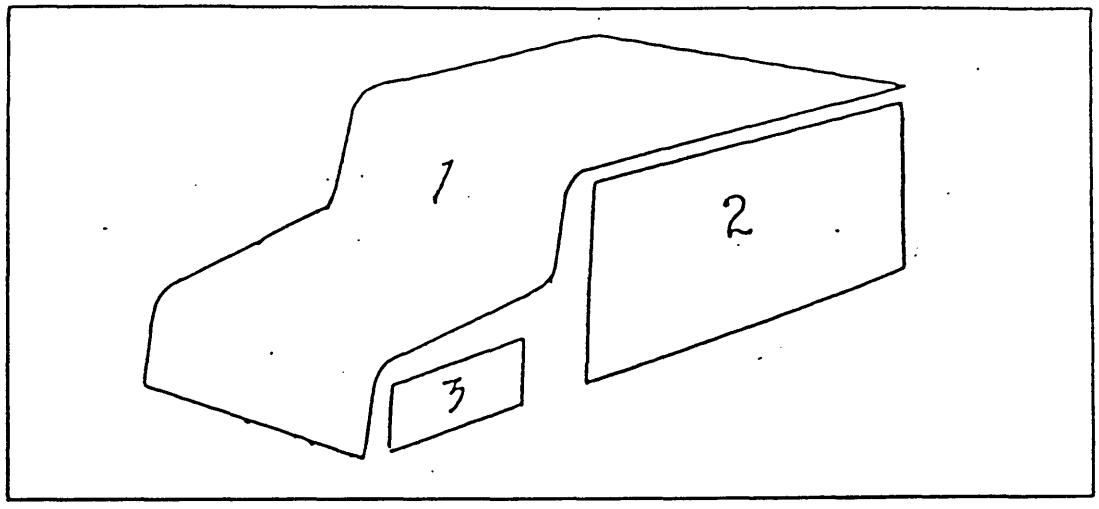


fig. 2 Surface split into three sections with no anomalous facets.

2. The Compressed Data Structure.

The CDS consists of a 2-dimensional array, in which every element contains the coordinates of the four vertices that define one quadrilateral facet. Defining every facet by an array element would require each vertex to be stored four times, since a vertex is common to four facets. Instead, only one in four facets is stored in the array. These are marked 'x' in fig. 3(a).

The vertices of the other facets can be easily retrieved from the relevant array elements. This can be achieved because of the correspondence between the position of the facets on the surface relative to each other and the order of storage in the array. Figure 3(a) shows numbered facets and their corresponding storage positions within the array.

The contents of each element in the array would typically include the x, y and z-coordinates of the four corner vertices defining facet f1 (see fig. 4), and data required, such as normals and diagonal information, for all four facets (f1..f4) in the 'group'.

The following 'C' language code shows how the CDS can be traversed and a wire-frame image generated -

```
typedef struct    {        /* screen coordinates */
   int x-coord, y-coord;
   int intensity;
}    COORDS_2D;

typedef struct    {        /* vertex coordinates */
   float x-coord, y-coord, z-coord;
}    COORDS_3D;

typedef struct    {
   COORDS-2D coords2;
   COORDS-3D coords3;
}    VERT;

typedef struct    {        /* four vertices defining a facet */
```

fig. 3 Only facets marked 'x' have their defining vertices stored.

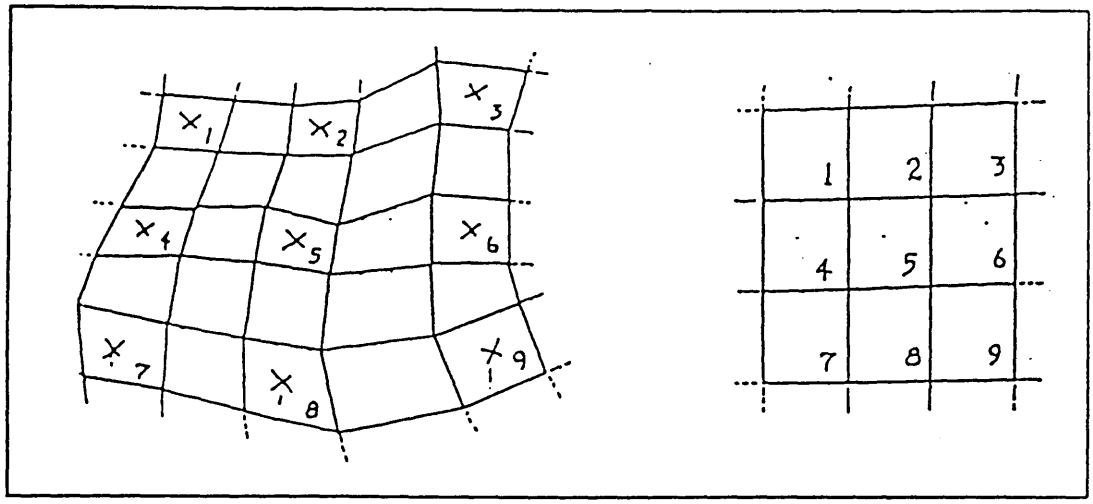


fig. 3 (a) Quadrilateral facets and corresponding CDS.

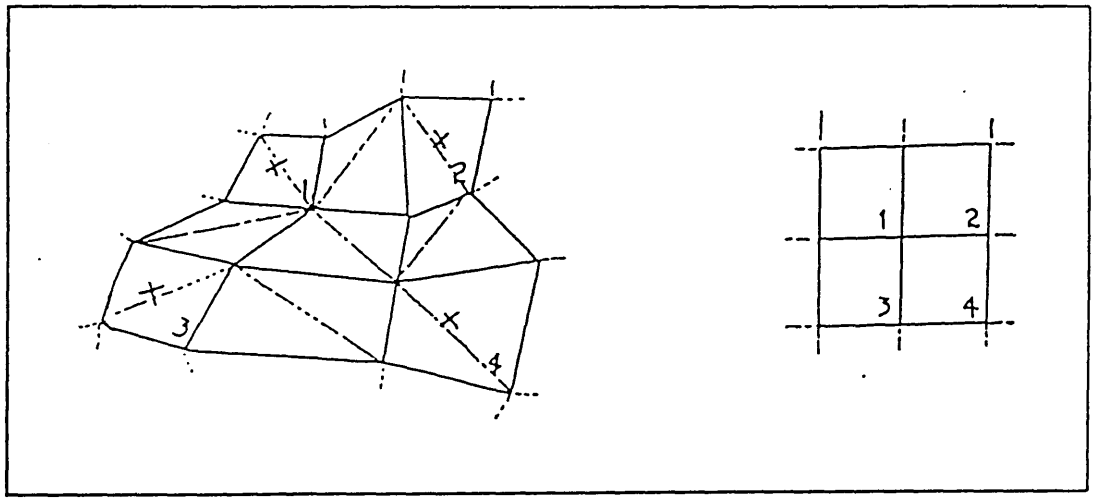


fig. 3 (b) Triangular facets and corresponding CDS.

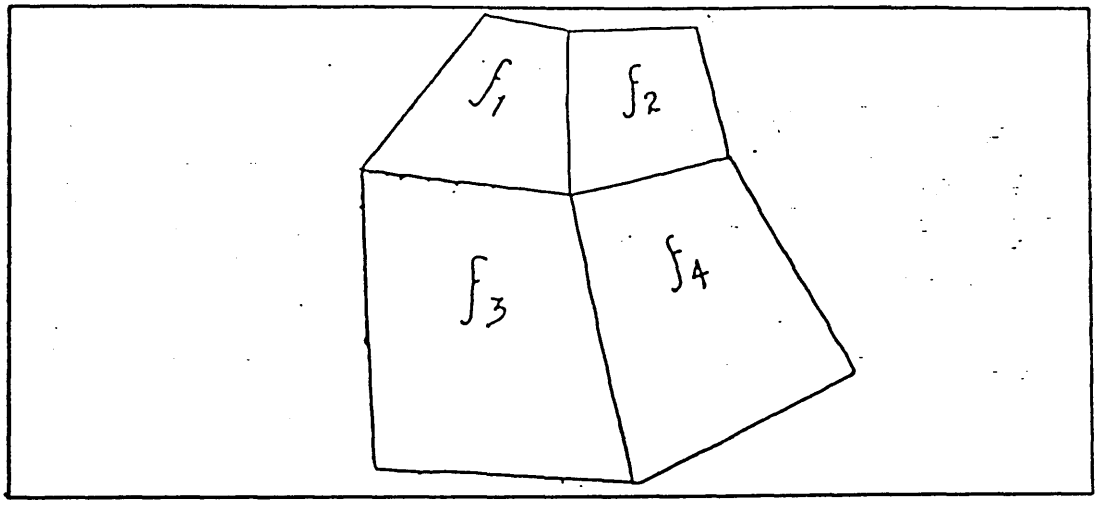


fig. 4 A group of four facets, only f1 has all its vertices stored in a single array element.

```
    VERT four_verts[4];
}   DS_FACETS;


draw-wire( )        /* generate wire-frame image */
{
    extern int max-hor,max-vert;
    int index1,index2;

    for     (index1 = 0; index1 < max-vert;index1++)    {
        for     (index2 = 0;index2 < max-hor;index2++)    {
            draw-facet(index1,index2);
            if      (index2 < (max-hor-1))
              draw-hor-connections(index1,index2);
        } /*end for */
        if      (index1 < (max-vert - 1))
          draw-vert-connections(index1);
    }     /* end for */
}       /* end draw-wire */
```

```
draw-facet(index1,index2)
int index1,index2;
{
     extern DS-FACETS *CDS, *cur-facet;
     extern int max-hor,max-vert;
     int count;

     cur-facet = CDS + (index1 * max-hor) + index2;
     for (count = 0;count < 4;count++)     {
          if (count != 3)
               draw-line(count,count+1);
          else
               draw-line(count,0);
     }     /* end for */
}       /* end draw-facet */


draw-hor-connections(index1,index2)
/* draw lines connecting current and next facet */
int index1, index2;
{
     extern DS-FACETS *CDS, *cur-facet, *next-facet;
     extern int max-hor;

     cur-facet = CDS + (index1 * max-hor) + index2;
     next-facet = cur-facet + 1;
     draw-between-line(1,0);
     draw-between-line(2,3);
}       /* end draw-hor-connection */


draw-vert-connections(index)
/* draw lines connecting current row and next row of facets */
int index;
{
     extern DS-FACETS *CDS, *cur-facet, *next-facet;
```

```
    extern int max-hor, max-vert;
    int index2;

    cur-facet = CDS + (index * max-hor);
    next-facet = CDS + max-hor;
    for (index2 = 0;index2 < max-vert;index2++)    {
        draw-between-line(3,0);
        draw-between-line(2,1);
        cur-facet++;
        next-facet++;
    }    /* end for */
}      /* end draw-vert-connections */
```

In order to imply the edges between vertices stored in the array, facets must be quadrilateral, the following restrictions enforce this -

1. Each facet must have exactly four edges.

2. Each edge must be defined by exactly two vertices.

3. Each vertex not incident on the perimeter of the surface, must be associated with four facets.

When the surface is being rendered it may be desirable for facets to be planar. The CDS handles non-planar facets by introducing a diagonal edge, thus changing the facet into two triangular facets, which are planar by definition (see fig. 3(b)). The diagonal chosen for the split is dependent on the curvature of facet being stored. The choice is stored in the appropriate array element.

If the surface of a solid is being represented it will be continuous, and will require connection information. If it is represented by a single CDS some wrap-round connections will provide the relationship between either the facets defined by the bottom and top row elements or the facets defined by the first and last column elements, or a mixture of both. If the surface has been split into sections then the connections between these sections must be stored. Data describing connections will contain a name, array elements and vertices involved. This information will be kept in a record containing global information for the solid.

To make correct connections for solid surfaces, a further restriction has to be introduced. All rows/columns must contain the same number of facets (i.e. vertices on the perimeter of the surface must be associated with exactly two facets).

When a surface is represented by facets, it is usual to restore realism by using some smooth shading technique [Newman 79]. It must therefore be possible to distinguish between "true" edges (i.e. where there is a discontinuity in the normal direction) and spurious edges that are introduced by the approximation to an originally smooth surface. It is desirable that these

spurious edges are undetectable after the smooth shading technique (e.g. Gouraud or Phong) has been applied. Array positions pertaining to facets that adjoin "true" edges are noted for use at the drawing step.

3. A Hierarchical Structure for Surfaces.

Many surfaces contain large areas that have very little change in curvature and therefore can be closely approximated by relatively few polygonal facets. These surfaces may also contain small areas containing lots of detail, which requires a relatively large number of facets for a realistic image. When representing such solids, the restriction that rows and columns of facets must be complete, could lead to the whole solid being represented with the large amount of detail required for these "small areas".

To avoid this inefficiency, the concept of a sub-surface is introduced; the area requiring a relatively large number of facets is contained within a facet or several adjacent facets in the original surface. This sub-surface is defined in a separate array from the main surface and is pointed to by a pointer in the array element corresponding to the facet(s) containing the sub-surface. A sub-surface array element can also point to another sub-surface, thus forming a hierarchy of surfaces.

To accommodate this hierarchy the definition of DS-FACETS is extended to

-

```
typedef struct    {
        union     {
                        VERT four-verts[4];
                        CDS_header *first-vert;
                  }     facet-kind;
}       DS-FACETS;
```

with first-vert containing the address of the first vertex stored in the sub-surface's CDS array.

In principle sub-surfaces can be represented by a completely different data structure (such as Baumgart's Winged-Edge), but this would require additional routines for node creation, manipulation and access. The pointer(s) in the array indicating the sub-surface would then point to this new structure.

4. A Changing Scene.

Moving a surface nearer to the viewing position can cause the amount of data required to increase dramatically in order to produce a realistic image of that surface.

The number of facets required to represent a surface can be determined by a combination of two things, the screen resolution of the display device, and the area that its largest image (i.e. at closest position to viewpoint) will cover on the screen. An image covering a large area of the screen will require much more data than the same image covering a small area. Therefore representing a surface by a specific amount of detail puts a restriction on the minimum distance from which the surface can be viewed with realism [Clark 76].

The same surface drawn on display devices of differing resolution may require different numbers of facets to represent it. At the drawing step it is inefficient to have more than one facet corresponding to one screen pixel. It is therefore an advantage if the number of facets can be easily adapted to correspond to the area that the image covers on the screen, and to the resolution of the display device.

4.1. Adapting the Number of Facets According to the Environment.

The method of adapting the number of facets is extremely simple. As each surface is processed, the area covered by the maximum bounding box is calculated in screen coordinates (i.e. with the surface at its estimated nearest point to the viewing position). This area determines the number of facets required for maximum precision. It is also used at the drawing step for comparison with the current screen area covered. The number of facets required (i.e. CDS array accesses) depends on this comparison.

Figure 5 shows a subset of facets representing a solid. Part (a) shows the facets required when the viewpoint is at its nearest position. At the drawing step all the array elements are accessed. Part (b) shows the facets required when a smaller area is covered. When drawing, every array element is accessed, but only a subset of the data is used. Finally, part (c) shows the facets required for an even smaller area. When drawing, every second array element of every second row is accessed. Array accesses can continue to be reduced, until the whole array represents only one facet.

When a reduced number of accesses is required, none of the array elements accessed contain all four vertices of any one facet. The vertices of a facet are then defined as the top-left vertices of the four 'adjacent' array elements accessed (see fig. 6).

When the surface is fairly flat, calculating the maximum screen area is pointless, since these will vary immensely according to the object's orientation to the viewpoint. However, it is useful to calculate a "bounding cuboid" in world coordinates. Then each time the object is drawn, the "bounding cuboid" is rotated in the same way as the object. It is then simple to calculate the screen area covered for this cuboid, which can then be compared with the area of the surface at its current position, to determine the number of array accesses required. In such a case, it may be desirable to access every mth row element and nth column element, where m need not equal n.

As an image grows smaller, minor details tend to disappear quicker than more major details, such as "true" edges. Correspondingly, when reducing the number of accesses into the array, "true" edges must be taken into consideration. Array elements that correspond to facets adjoining "true" edges are always accessed until areas between pairs of "true" edges are no longer

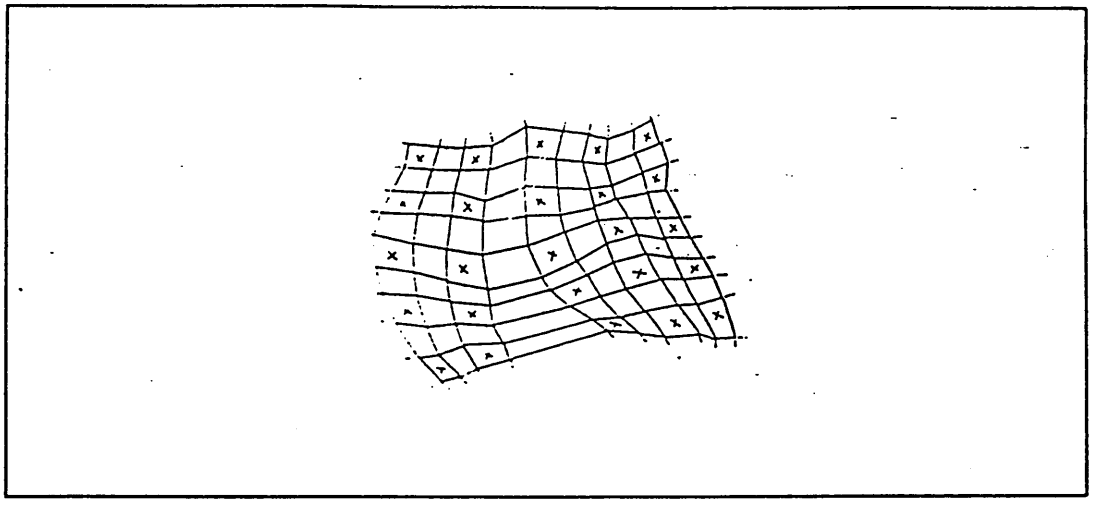


fig. 5 (a) Facets required when the viewing position is at its nearest point.

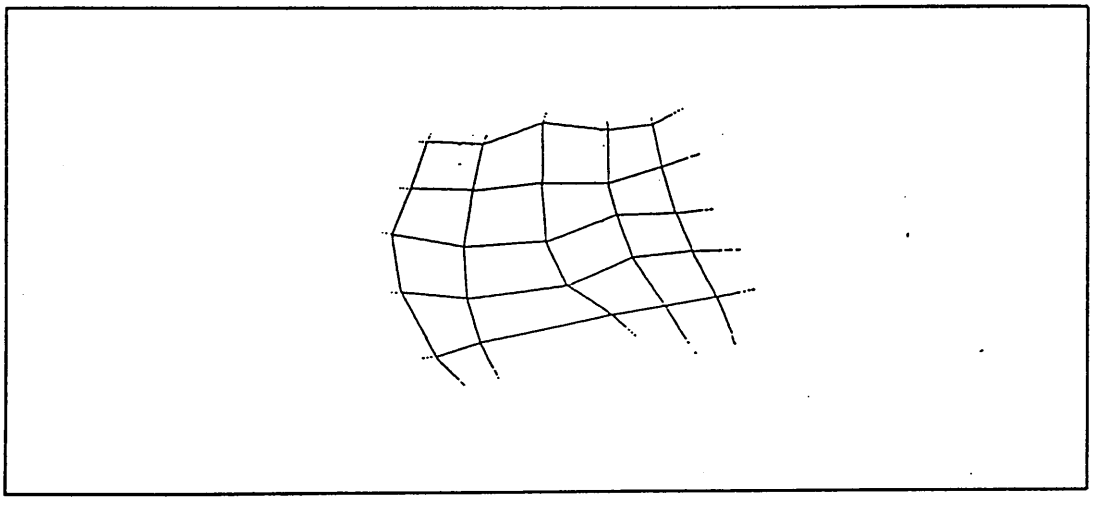


fig. 5 (b) Facets required when the viewing position is further away than at (a).

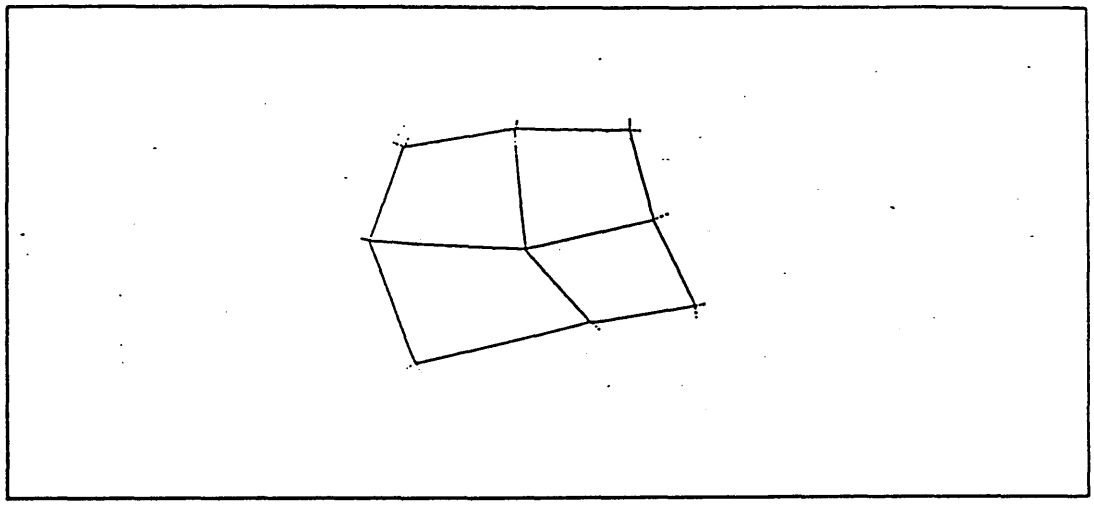


fig. 5 (c) Facets required when the viewing position is further away than at (b).

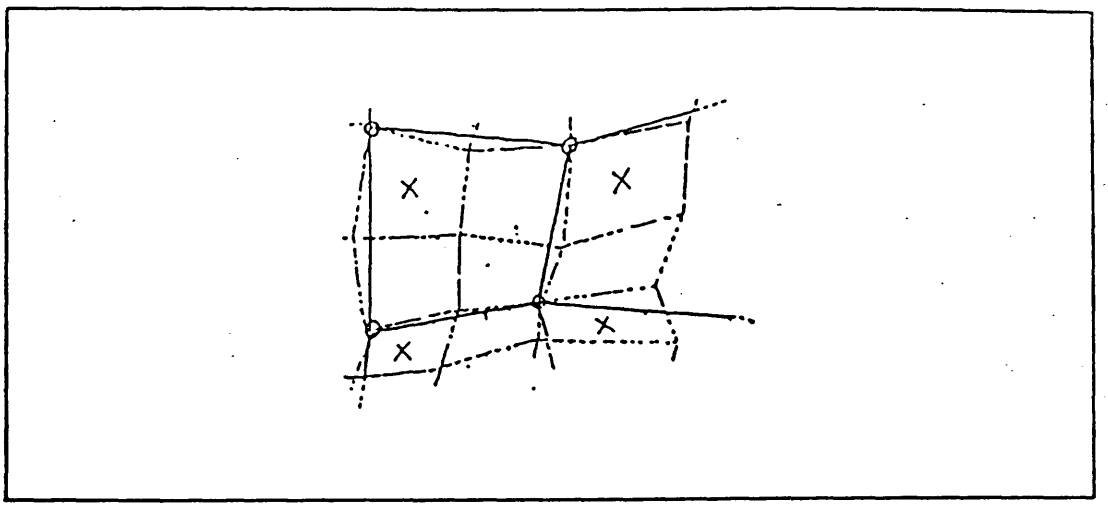


fig. 6 When a reduced number of vertices is required, the topleft vertices of array elements accessed define the new facet.

significant in size. These pairs of edges are then merged together to form new 'true' edges.

4.2. Increasing Efficiency by Considering Visibility.

Only visible facets are considered at the drawing step. On average this will halve the number of array elements of interest when drawing solids. Since most solids have their visible facets adjacent to each other, after finding one visible facet it is easy to find the others.

In looking for one visible facet it would be inefficient to access all the array elements in turn. It is much more efficient to use some search method, such as the one described by the following code (higher level first) -

```
procedure visible;
begin
find largest 2^n × 2^n section in array;
while (visible facet not found and search not complete) do
    begin
    test if visible, the facet defined in every 2^nth column in every
    2^nth row in array (if not previously tested);
    decrement n (i.e. quarter each section);
    end while;
end procedure.
```

```
visible( )
      /* search current CDS array for visible facets, sparsely at first,
      examining more and more elements as the search progresses */
{
      extern DS_FACETS *CDS;
      extern int max_hor, max_vert;
      int found_flag,min( ),all_in_row,all_in_col,row,col,step_size;
      double pow( ),log2( ),d_two;

      d_two = 2.0;
            /* find largest 2**n × 2**n section in array */
      step_size                                                    =
      min(pow(d_two,log2(max_hor)),pow(d_two,log2(max_vert)));
      row = col = step_size - 1;
      found_flag = 0;
      while    ((!found_flag) && (step_size != 0))    {
            all_in_row = 1;
            while    ((!found_flag)  && (row < max_hor))  {
                  all_in_col = 1;
                  while    ((found_flag == 0) &&( col < max_vert))  {
                        if    ((odd(all_in_row)) || ((even(all_in_row))
                                    && (odd(all_in_col))))
                              found_flag = facet_visible(row,col);
                        all_in_col++;
                        col += step_size;
                  } /* end inner while */
                  all_in_row++;
                  row += step_size;
                  col = step_size - 1;
            } /* end while */
            step_size /= 2;
            row = col = step_size - 1;
      } /* end outer while */
} /* end visible */
```

```
int intlog2(n)
int n;
{
     register power = 0;
     int n;

     while (test <= n)
          { ++power;
          test <<= 1;
          }
     return (--power);
}     /* end intlog2 */


double log2(num)
int num;
{
     double pow_num,cur_num,d_two;

     pow_num = 1.0;
     d_two = 2.0;
     cur_num = pow(pow_num,d_two);
     while    (cur_num <= num)
       cur_num = pow(++pow_num,d_two);
     return(pow_num);
}
```

Generally, the above algorithm will only require a few array accesses. The other visible facets will be defined by the surrounding array elements.

For most convex surfaces there will be only one group of visible facets. However, it is possible for an object to be convex and yet fold itself over, such as a spiral shape. To take advantage of the knowledge above, such an object should be split into several non-folding sections and stored accordingly. If a surface has any concave areas there may be invisible facets mixed among the group of visible facets, in which case there is likely to be some correspondence between facets lying on edges of adjacent visible areas. The

facets where visibility changes will provide the contour of the object, which is useful in hidden surface and shadow calculations, and also anti-aliasing.

When the viewing position is moving relative to an object, facets changing in visibility will be positioned along the boundaries of the visible areas already in existence, or in areas that were previously hidden by the boundary. Using this knowledge will enable the visibility updates from frame to frame, to be made with the minimum amount of effort.

For hierarchical surfaces, if the facet containing a sub-surface is invisible and the sub-surface does not protrude then the whole of that sub-surface will be invisible. Otherwise, the array representing the sub-surface must be searched for visible facets in the same way as the 'main' array.

5. Solid Modelling Using Constructive Solid Geometry Methods and the CDS.

Using Constructive Solid Geometry (CSG) methods a complex solid is represented by a collection of simpler solids. These are combined by the Boolean set operations union, intersection and difference. This representation uses a binary tree, with leaf nodes being system primitives and internal nodes being compositions of these.

The system solids or primitives are typically blocks, cylinders, spheres, cones and tori. Some solid modelling systems use an unambiguous representation of these primitive solids [REQU82]. Polygonal representations are used in at least three contemporary solid modelling systems: IBM's GDP [WESL80], Matra's Euclid [BERN75] and Cambridge Interactive System's Medusa. The CDS could be used as the data structure for storing such representations.

The following paragraphs explain how the normal range of CSG primitives are represented using the CDS.

A block requires a two element array (see fig. 7(a)), since there are only eight vertices in the cube. The type 'block' implies that there are connections between the two facets, the order of storage enables the correct connections to be made.

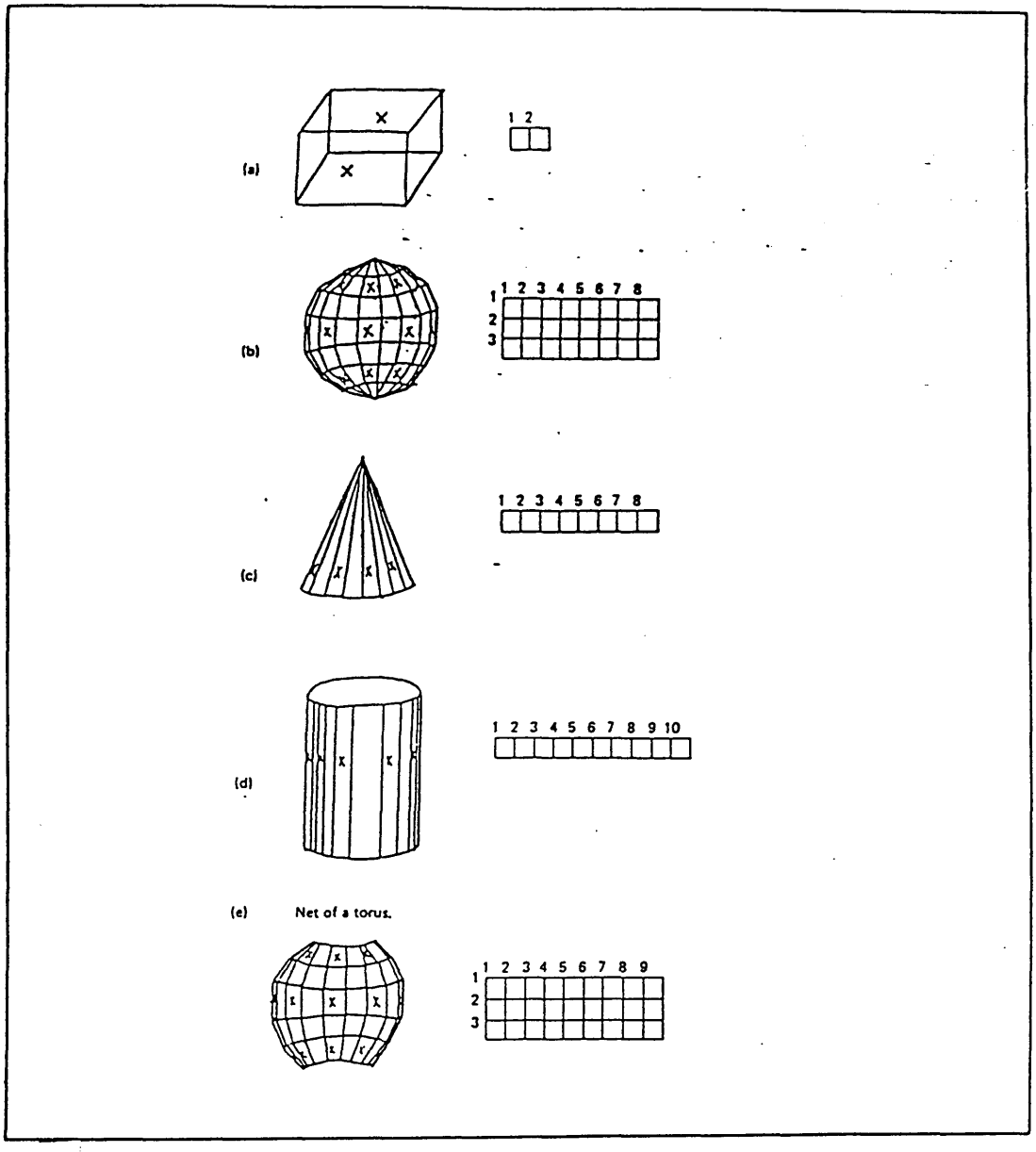


fig. 7 : Typical CSG Primitives and their corresponding CDS.

For a sphere represented by m rows and n columns of "quadrilateral" facets, the minimum requirement is an array of size $\frac{1}{2}m(n+1)$. The coordinates of the poles, are stored separately (see fig. 7(b)). The type 'sphere' implies that connections exist between the facets defined in the first row of the array and the top pole, between facets defined in the last row and the bottom pole, and between facets defined in the first and last columns.

A cone is represented as shown in fig. 7(c), and requires $\frac{1}{2}\ n$ elements in the array, where n is the number of facets on the side of the cone. The type 'cone' implies that the planar end is defined by the bottom vertices of all the facets. The cone's top can be stored in the top-left of element (0,0). No other top-left vertices and no top-right vertices need be stored. Facets defined in the first and last elements are connected.

A cylinder is represented as shown in fig. 7(d). The type 'cylinder' implies that the top plane is defined by the top vertices of the facets, and the bottom plane is defined by the bottom vertices, and that there is a connection between the first and last elements.

Finally, a torus is represented in much the same way as a sphere, requiring the same size of array (see fig. 7(e)). The type 'torus' implies that the facets defined by the first and last rows (and columns) of the array are connected.

When two primitives are combined their arrays must also be 'combined'. For union and intersection operations this can be achieved by examining the facets stored in each array and noting which facets lie in the intersection.

If the operation is union these facets will be invisible and will now point to the other primitive solid in the union. As in hierarchical structuring, see section 3. If the operation is intersection these facets are the ones required for the new solid. It may be possible to reduce the dimensions of the two arrays to accomodate only these required facets. Some connection would then be set up between these arrays, showing that they both form the same solid's surface.

The difference operation is a negative union. These facets will form part of the surface of the new solid, their normals will have to be updated to point in the opposite direction for hidden surface and intensity calculations. It may be possible to reduce the array containing these facets, since "non-intersecting"

facets are no longer required. The appropriate array elements for the other primitive would then point to this new array.

The above is a simplified approach, which assumes that the intersection of the solids happens along the edges of facets. In reality

a new boundary for the intersecting surfaces will need to be calculated which is no mean feat. This can be avoided if ray casting methods are used.

6. Ray Casting and the CDS.

Using ray casting techniques, sight rays are cast from the viewpoint through each pixel in the pixel plane (or screen) and into the scene. Hidden surface elimination is achieved by considering the first opaque surface intersected by a ray, this provides the visible surface at that screen point. Any transparent surfaces lying between the viewpoint and this surface would have to be considered. Clipping is accomplished by only considering surfaces that are hit by a ray.

The colour and intensity at any point on the surface of an object is dependent on its location with respect to the light source(s), it's reflection coefficients and the luminiferous quality of surrounding objects. When two objects are placed next to each other, every point on one has to be considered as a potential light source for the other [Kay 79]. Therefore if a realistic image is required, each ray must be traced through the environment and the results accumulated.

Whenever a ray intersects a surface, it can be reflected in many directions, resulting in the emission of several reflected rays. Each of these rays can then intersect another surface, resulting in the further emission of several more rays. This whole process can be described by a tree structure.

When tracing a ray, it is impossible to determine in advance, which objects will be intersected by the nth reflected ray (n>1). Therefore, it would be ideal if information about every facet on every surface was available at any particular time. The CDS will require less storage space than some of the other data structures currently used, (see section 7) thus enabling more information to reside in direct access store at any particular time. This implies that fewer or no I/O operations will be required during the ray tracing step, and that transfer time will generally be shorter.

6.1. Using CSG Methods.

If the object being traced is modelled using CSG methods (see section 5), then the CSG tree is traversed from the bottom upwards. Each ray is classified with respect to the primitives at the leaf nodes. The classification of a ray is the information regarding which parts of the ray are inside the primitive and which are outside. At each internal node the combine operators enable the classification to be made for that node.

Using Roth's ray casting methods [Roth 82] for modelling solids, the following restrictions are made -

1. The display screen is defined as being the x,y-plane centered at the origin in the world coordinate system.

2. The viewing position is placed along the negative z-axis, in the world coordinate system.

3. The visible solids are positioned in the direction of the positive z-axis.

These restrictions ensure that rays are uniformly distributed in the primitives' own local coordinate system where the primitive objects are defined.

Whenever an object's primitive type, size and position are defined by the user, a scene-to-local transformation matrix is stored along with any other data required for that object, such as a bounding box in screen coordinates. The scene-to-local transformation is for the rays, the primitive itself never undergoes any transformation. This allows the representation of a primitive to be defined once, its characteristics can then be given to any number of solids of the same type, each having a different scene-to-local transformation.

Prior to the drawing step, the CSG tree is traversed and the bounding boxes combined using the boolean operations. At the ray casting step rays are cast from the viewing position through every nth pixel (n>=1) in the order left to right, top to bottom. The tree is traversed for each ray, only accessing branches for which the ray lies within the bounding box. At the primitive node, rays have to undergo the appropriate scene-to-local transformations. Each transformation requires 15 multiplications and 12 additions. Thus for a primitive having a bounding box covering l x l pixels on the display screen, the transformation step requires -

$$15\left(\frac{l^2}{n^2}\right) \text{ multiplications, and } 12\left(\frac{l^2}{n^2}\right) \text{ additions.}$$

Because rays are uniformly distributed in the local coordinate space, only the four rays, which pass through the corner pixels of this bounding box, require to undergo any transformation. Linear interpolation can be used to deduce the other rays. It only requires 3 multiplications and 3 additions to deduce a ray, thus, for the same object as above, the results can be achieved with -

$$3\left(\frac{l^2}{n^2}\right) + 60 \text{ multiplications, and } 3\left(\frac{l^2}{n^2}\right) + 48 \text{ additions,}$$

which is a substantial saving in computation time for $\frac{l^2}{n^2} > 5$.

7. Minimising the Number of Ray-Facet Intersection Tests.

The ray casting step normally uses a substantial portion of the processing time, with most of this time being spent on finding the points where the ray intersects the surface. It is therefore important that this step be done efficiently.

It is often difficult to predict what facet of what object will be intersected next. However, using the knowledge that there is a correspondence between the facets and the order of storage in the CDS (i.e. adjacent facets are 'adjacent' in the array), it is possible to choose with some degree of accuracy, the array element that is most likely to give a positive result to the ray-facet intersection test. Rays are passed through adjacent pixels in the pixel plane, so if one ray hits a facet, then the next ray will usually hit either the same facet, or an adjacent one.

A bounding box test will enable the position of the next overlapping bounding box to be deduced. This ensures that the current object is not hidden by a new object until at least this position is reached. If a consecutive ray intersects the same object as the previous ray, the intersection test will initially be limited to the array elements around the row and column found for the previous ray. Objects containing concave areas must be tested to ensure that the area being investigated is not overlapped by another area from the same solid.

When two consecutive rays intersect within the same area of a solid, the 'direction increments' within the array are noted (i.e. changes in row and column). When they intersect in different areas, or objects, the array position containing the facet where the intersection occurred is noted. The intersection points of subsequent rays passing through the first row of pixels can be easily deduced, by considering another coherence property. The direction between one intersection point and the 'next' is likely to be the same as the direction between the 'next' and the 'next again'. When the same object is being considered, the row and column increments help to pinpoint the array position which defines the facet that the new ray will be most likely to intersect. The increments are noted each time a row and column are found.

All these increments and array positions are used by rays passing through the second row of the pixel array. The majority of these increments will remain the same from row to row. Array positions will normally undergo only small changes. Any differences are updated and used by the next row.

This section so far has concentrated on rows of pixels. A further reduction in ray-facet intersection tests can be achieved by storing increments for columns. These are the actual differences in the increments and array positions between consecutive rows of pixels, and can be stored as the updates are being done. At any one time, there will be increments and array positions stored for rays pertaining to only one row in the pixel array. The active bounding box list will supply the objects that these refer to.

The coherence properties applied above would generally not hold so tightly if using Kay's ray tracing methods, for a highly realistic illumination model, as the tracing of the ray continues and the ray has been reflected off several objects. For each of these objects, the idea of probing the array to find intersection points more quickly would still work with the same efficiency.

8. Comparison Between the CDS and Baumgart's Winged-Edge Data Structure.

The winged-edge data structure is based on a polyhedral representation, which the CDS can also handle. Using the winged-edge structure, topological information is stored explicitly in the edge nodes (see fig. 8), the geometry is stored

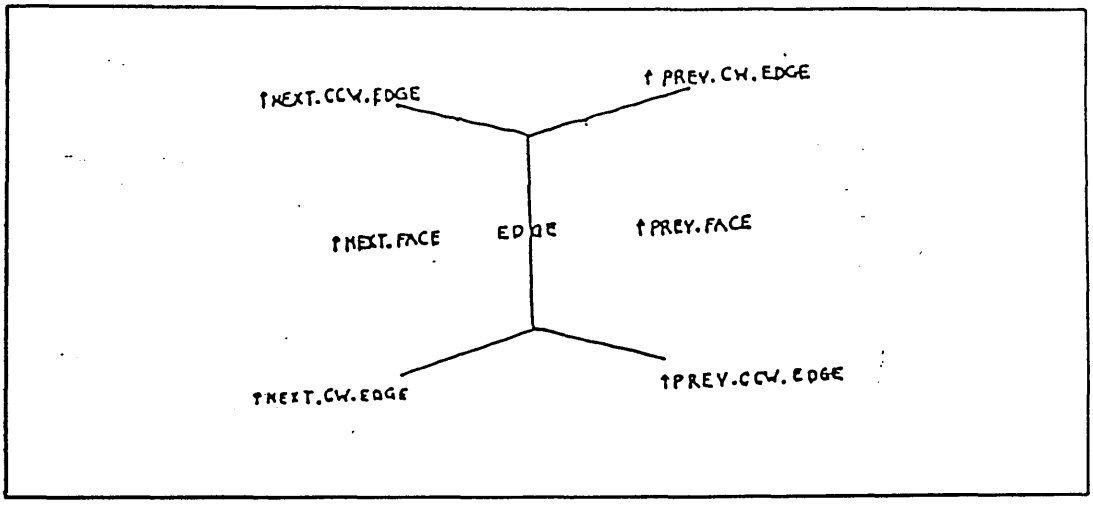


fig. 8 Winged-Edge topology, showing the various pointers associated with each edge.

in the vertex nodes and the colour and intensity information is stored in the face nodes. For any particular solid, these edge, vertex and face nodes are stored in three rings. In the CDS the topological information is implied by the array positions, the geometry information is stored in the appropriate array elements. Photometry information is stored globally wherever possible, otherwise it is stored in the array elements.

The winged-edge structure can handle a more general class of polyhedra than the CDS, in particular it can be used for storing polyhedra represented by facets which are not restricted to having a fixed number of edges. Using the pointers the addition or deletion of facets is simple. The CDS is restricted to polyhedra consisting of triangular pairs or quadrilateral facets, which must be in complete rows and columns. This makes it very difficult to add or delete facets, however, I am investigating whether it is worth using CDS ideas in a quad-tree structure instead of an array, with each tree node containing the same information as an array element plus necessary pointers. This will enable these operations to be done, but may have a detrimental effect on access time.

8.1. Access of Data.

The winged-edge data structure requires sixteen routines for node creation and manipulation and nine accessing routines. Direct access of data is impossible, one of the rings must be serially accessed until the required node is reached. Corresponding nodes in the other rings can be accessed by following pointers stored in this node.

The CDS being an array provides direct access enabling efficient search techniques to be used. No special creation or accessing routines are required, because of the simplicity of the structure.

8.2. Storage Requirements.

Consider one solid represented by n polygonal facets. Photometry information would require the same amount of storage using either structure, therefore this discussion is confined to the storage of topological and geometrical information.

The word size for storing a 'C' language pointer may vary in different compilers. I have chosen a $\frac{1}{2}$word, which is the storage required by the compiler I utilise, other compilers may take a word.

Baumgart's Winged-Edge Structure.

The body node contains six pointers to the start and end of the face, edge and vertex rings, requiring three words of storage. Only one body node is required per solid. The edge node contains the topological information and requires ten pointers. The number of edge nodes required in the representation of the solid described above is 2n, therefore, 10n words are required for the pointers. The face node contains only three pointers, requiring $1\frac{1}{2}$ words of storage. The solid has n face nodes, therefore requires $1\frac{1}{2}$n words for the pointers. The vertex node also contains three pointers. In addition the x, y and z-coordinates of the vertex are stored. The solid requires n vertex nodes, therefore $4\frac{1}{2}$n words of storage is required. The overall storage requirement for non-photometry data is therefore 3 + 16n words.

The Compressed Data Structure.

The CDS is a two-dimensional array that requires $\frac{1}{4}$n elements to store a solid represented by n facets. Each element contains the x, y and z-coordinates of the four defining vertices of the facet that it explicitly represents. Therefore the overall storage requirement is 3n words.

8.2.1. Result.

The winged-edge structure will require more than five times the amount of storage that the CDS requires for storing non-photometry data. The CDS enables direct access of data which is not possible using the winged-edge structure, but there are restrictions on how the surface is split into facets, these restrictions make it difficult to add or delete facets to an existing surface. Hence, storage requirements must be balanced against generality when deciding which data structure to use.

9. Conclusions.

The CDS allows a surface with numerous facets to be stored in a relatively small amount of space. Data can be easily and directly accessed from the CDS array. A direct correspondence exists between the positioning of facets and the array elements, which may allow spatial coherence properties to be efficiently exploited. However, to gain these advantages, there are restrictions on the way the surfaces are divided into facets. For some applications these restrictions will not be onerous. Operations that affect all the vertex coordinates (e.g. moving the viewing position) could be executed using one of the array processors which are commercially available today, the host would then be free to continue other tasks. Computers with limited direct access store can draw complex objects with fewer I/O operations, when using the CDS than when using most of the other data structures currently available.

10. Acknowledgements.

I would like to thank Dr. A.C. Kilgour for his helpful suggestions and constructive criticisms throughout the period of writing this paper. I would also like to thank the British Science & Engineering Research Council for their financial support while carrying out the work involved.

11. References.

[Baumgart 75] : "A Polyhedron Representation for Computer Vision", B.G. Baumgart, Proceedings AFIPS National Computer Conference 1975, pp 589-596.

[Bernascon 75] : "Automated Aids for the Design of Mechanical Parts", Y.J. Bernascon and J.M. Brun, Tech. Paper MS75-508, Society of Manufacturing Engineers, 1975.

[Clark 76] : "Hierarchical Geometric Models for Visible Surface Algorithms", J.H. Clark, Comm of the ACM, Oct76, Vol.19, no.10, pp 547-554.

[Forrest 78] : "A Unified Approach to Geometric Modelling", A.R. Forrest, Computer Graphics, Vol.12, no.3, Aug 1978, pp 264-269.

[Kay 79] : "Transparency, Refraction and Ray Tracing for Computer Synthesized Images", D.S. Kay, Master's Thesis, Cornell University, Ithaca, N.Y., Jan 1979.

[Newman 79] : "Principles of Interactive Computer Graphics", W.M. Newman and R.F. Sproull, Second Edition, McGraw-Hill 1979, pp 398-404.

[Requicha 82] : "Solid Modeling: A Historical Summary and Contemporary Assessment", A.A.G. Requicha and H.B. Voelcker, IEEE Computer Graphics and Applications, Vol.2, No.2, March 1982, pp 9-24.

[Roth 82] : "Ray Casting for Modeling Solids",

S.D. Roth, Computer Graphics and Image Processing, 18, 1982, pp 109-144.

[Wesley 80] : "Construction and Use of Geometric Models", M.A. Wesley, Computer Aided Design, J. Encarnacao, ed., Springer-Verlag, N.Y., 1980, pp 79-136.

APPENDIX C : COMPRESSED DATA STRUCTURE

FOR

ROTATIONAL SWEEP METHOD.

The following paper was published as part of the AUSGRAPH 1987 Conference Proceedings Perth, Australia 4th - 8th May 1987.

Compressed Data Structure for Rotational Sweep Method.

Marion S. Cottingham

Department of Computer Science,
Monash University.

PRECIS.

The rotational sweep method typically approximates the surface of an object by a collection of quadrilateral facets. A smooth-shading technique is applied to these facets to produce a shaded image. Obtaining a smooth image in regions of high curvature requires the surface to be represented by hundreds or thousands of facets. The large number of facets involved makes it extremely important that geometrical and topological information is stored in an efficient manner. This information must include all that is required for an unambiguous representation of the solid(s) in question.

The compressed data structure (CDS) stores such a representation efficiently, considering both volume of storage required and ease of access. The CDS is designed to minimize the amount of data stored with as much information as possible being implied. Therefore only geometrical information is actually stored, topological information is implied by the order of storage.

Keywords : compressed data structure, rotational sweep, surface modelling.

1. Rotational Sweep Method.

The rotational sweep method is used to define objects that have their symmetry preserved when rotated. An object is defined by a contour and an axis of rotation (see fig. 1(a)). The contour is typically defined by a collection of points.

The contour points are normally defined in the Cartesian coordinate system with the x, y and z-coordinates being sufficient to represent them. These points are rotated *n* degrees at a time, one full revolution around a given axis of rotation (see fig. 1(b)). At each step in the rotation the points defining the transformed contour (i.e. the vertices) are stored as part of the surface definition. This has the effect of dividing an object's surface into a collection of quadrilateral facets. Approximating the surface by a polyhedron, it is usual to apply a smooth shading technique to restore the surface's smooth appearance [Newman 1979].

Objects defined using the rotational sweep method often contain regions of high curvature that require a large number of facets to provide an acceptable surface approximation to enable the shading algorithms to generate a reasonably realistic image. It is therefore important that data is stored efficiently and that it is easily accessible to the rendering algorithms.

2. Compressed Data Structure.

The compressed data structure (CDS) [Cottingham 85] is suitable for storing the representation of objects' surfaces that are split into quadrilateral facets, such as those defined by rotational sweep methods. The surface definition must contain the coordinates of all the vertices defining the object's surface and the topology showing how these vertices are connected.

The underlying data structure of the CDS is a 3-dimensional array, making it simple to implement and use since array operations are already well understood in computing. The 3-dimensional array consists of a 2-dimensional array of facets with each element containing a 4-element array storing the four vertices that define one quadrilateral facet. A facet's edges are implied by the order of the vertices within this 4-element array; an edge is implied between adjacently stored pairs of vertices and between the first and last vertices in the array (see fig. 2).

Providing an array element for every facet would require each vertex to be stored four times, a vertex being common to four faces. Therefore only one facet per group of four has all defining vertices stored in a single 'facet' array element. Figure 3(a) shows facets that have their vertices stored in one 'facet' array element. The existence of the other three facets is implied by assuming that there are edges connecting adjacent facets in the 'facet' array (see fig. 3(b)). This implication is possible because the vertices are stored in a specific order and there is a direct correspondence between the position of facets on the surface with respect to their surrounding facets and their storage position within the array, i.e. facets 'adjacent' on the surface are stored at 'adjacent' elements within the array.

Each 'facet' array element includes the x, y and z-coordinates of the four vertices defining facet f_0 (see fig. 4) plus any photometry information required for all four facets (f_0 ..... f_3) in the 'group'. To enable the correct correspondence between facets and their array position, the following restrictions are made on the facets representing the surface -

1. Each facet must have exactly four edges.

2. Each edge must be defined by exactly two vertices.

3. Each vertex not incident on the perimeter of the surface, must be associated with four facets.

Using a polyhedral representation, hidden surface elimination is normally partly achieved by culling back-facing polygons. Back-facing polygons are usually detected by examining the normal vector of each facet. A normal is computed from the plane equation of a facet with vertex coordinates specified in a clockwise or anti-clockwise order depending on whether a left-handed or right-handed coordinate system is used [Hearn 86].

The left-handed coordinate system is adopted for the present work. The viewing position is fixed along the negative z-axis, making the order of the vertices stored within the CDS such that front-facing facets are in a clockwise order and back-facing facets are in an anti-clockwise order. Identification of back-facing facets within the CDS can thus be achieved without actually computing any plane

equations or calculating any normals.

3. Using the CDS with the Rotational Sweep Method.

3.1. Creating and Storing Data.

The rotational sweep method creates vertices in a predefined order, making it simple to store vertices at their correct position within the CDS.

The following pseudo-code computes the points required to approximate the surface of an object when given a set of points defining its contour, an axis of symmetry and the number of points required to be generated. The code also stores the points generated at the correct position within the CDS. Each point in the contour is rotated round the axis of symmetry n degrees at a time, where n = 360/'number of points required'.

```
allocate space for CDS_array;
row_num = 0;
cur_facet = CDS_array[row_num][0];
for each point in contour do
    begin
    read coordinates of contour point;
        /* place into CDS_array */
    if odd number points processed so far do
        cur_facet.v3 = point;
    else
        cur_facet.v0 = point;
    initialise angle to 0;
    for each point in rotation do
        begin
        increment angle;
        rotate point about axis of rotation by 'angle';
            /* place into CDS_array */
        if even number points processed so far do
            begin
```

```
                if odd number rotations done so far and
                    not in last column of CDS_array do
                    begin
                    set cur_facet to facet in next column;
                    cur_facet.v0 = rotated point;
                    end;
                else
                    begin
                    cur_facet.v1 = rotated point;
                    if cur_facet in last column of CDS_array do
                        cur_facet = CDS_array[row_num][0];
                    end;
                end;
            else   /* odd number points processed so far */
                if odd number rotations so far do
                    begin
                    set cur_facet to facet in next column;
                    cur_facet.v3 = rotated point;
                    end;
                else
                    begin
                    cur_facet.v2 = rotated point;
                    if in last column of CDS_array do
                        begin
                        increment row_num;
                        cur_facet = CDS_array[row_num][0];
                        end;
                    end;
                end;
            end;
        end;
```

3.2. Generating the Image.

At the rendering step the coordinates of the four vertices defining a quadrilateral are retrieved from the CDS. The following pseudo-code shows how this is achieved and how a wire-frame drawing can be generated using this data.

```
cur_row = 0;
for each row do
    begin
    cur_facet = facet_array[cur_row][0];
    next_facet = facet_array[cur_row][1];
    next_row_facet = facet_array[cur_row + 1][0];
    for each column do
        begin
            /* generate directly stored facet (f0) */
        draw_line(cur_facet.v0,cur_facet.v1);
        draw_line(cur_facet.v1,cur_facet.v2);
        draw_line(cur_facet.v2,cur_facet.v3);
        draw_line(cur_facet.v3,cur_facet.v0);

            /* generate edges connecting columns(f1) */
        if cur_facet not in last column do
            begin
            draw_line(cur_facet.v1,next_facet.v0);
            draw_line(cur_facet.v2,next_facet.v3);
            end;
           /* generate edges connecting rows (f3) */
        if cur_facet not in last row do
            begin
            draw_line(cur_facet.v3,next_row_facet.v0);
            draw_line(cur_facet.v2,next_row_facet.v1);
            end;
        set cur_facet, next_facet and next_row_facet to facets in next
        columns;
        end;
    increment cur_row;
end;
```

Each edge in drawn only once. The data structure enables common edges to be easily identified. The four edges of facet f_0 are drawn but only two edges of facets f_1 and f_3 are drawn. No edges of facet f_2 are drawn.

Generating a shaded image, facets are retrieved from the CDS in the same way as above. Calls to the 'draw_line' routine are replaced by calls to a 'shade_facet' routine that requires the four vertices defining a facet as its parameters. Additional code is included for generating facet f_2. The following pseudo-code shows how this is done -

```
cur_row = 0;
for each row do
    begin
    cur_facet = facet_array[cur_row][0];
    next_facet = facet_array[cur_row][1];
    next_row_facet = facet_array[cur_row + 1][0];
    for each column do
        begin
            /* generate directly stored facet f0 */
        shade_facet(cur_facet.v0,cur_facet.v1,cur_facet.v2,cur_facet.v3);
            /* generate facet f1 */
        if cur_facet not in last column do
            shade_facet(cur_facet.v1,next_facet.v0,cur_facet.v2,next_facet.v3);
           /* generate facet f3 */
        if cur_facet not in last row do
            shade_facet(cur_facet.v3,next_row_facet.v0,cur_facet.v2,
                        next_row_facet.v1);
           /* generate facet f2 */
        if cur_facet not in last row or last column do
            shade_facet(cur_facet.v2,(cur_facet+1).v3,
                        (next_row_facet + 1).v0,next_row_facet.v2);
        set cur_facet, next_facet and next_row_facet to facets in next
        columns;
        end;
    increment cur_row;
end;
```

The shade_facet routine smooth shades the polygon defined by the four vertices using a smooth shading routine such as Gouraud's or Phong's.

Around half of the facets used to approximate the surface in rotational sweep methods will be visible in the final image. Figure 5(a) shows all the facets drawn, figure 5(b) shows only front-facing facets drawn. In general, visible facets in convex areas are adjacent to each other on the object's surface. The correspondence between the position of facets on the surface and their position in the CDS array, means visible facets are generally stored at adjacent array elements. Therefore after finding one visible facet coherence properties can be used to find the others.

4. Conclusion.

The CDS stores a polyhedral approximation of an object in a relatively small amount of space. The underlying array structure makes it easy to implement since array operations are already well understood and used in computing. The array structure enables data to be directly accessed. Transformations affecting all the vertex coordinates, such as moving the viewing position, could be executed using one of the array processors commercially available. A polyhedral approximation of an object often requires a vast amount of information. Computers with limited direct access memory read and write parts of the data as required. Using the CDS such computers can draw complex objects with fewer I/O operations than when using most of the other data structures currently available.

5. References.

[Cottingham 85] : "A Compressed Data Structure for Surface Representation", M.S. Cottingham, Computer Graphics Forum, Vol.4,No.3,September 1985, pp.217-228.

[Hearn 86] : "Computer Graphics", D. Hearn & M.P. Baker, Prentice-Hall International Editions 1986.

[Newman 79] : "Principles of Interactive Computer Graphics", W.M. Newman & R.F. Sproull, 2nd Edition, McGraw-Hill, 1979.

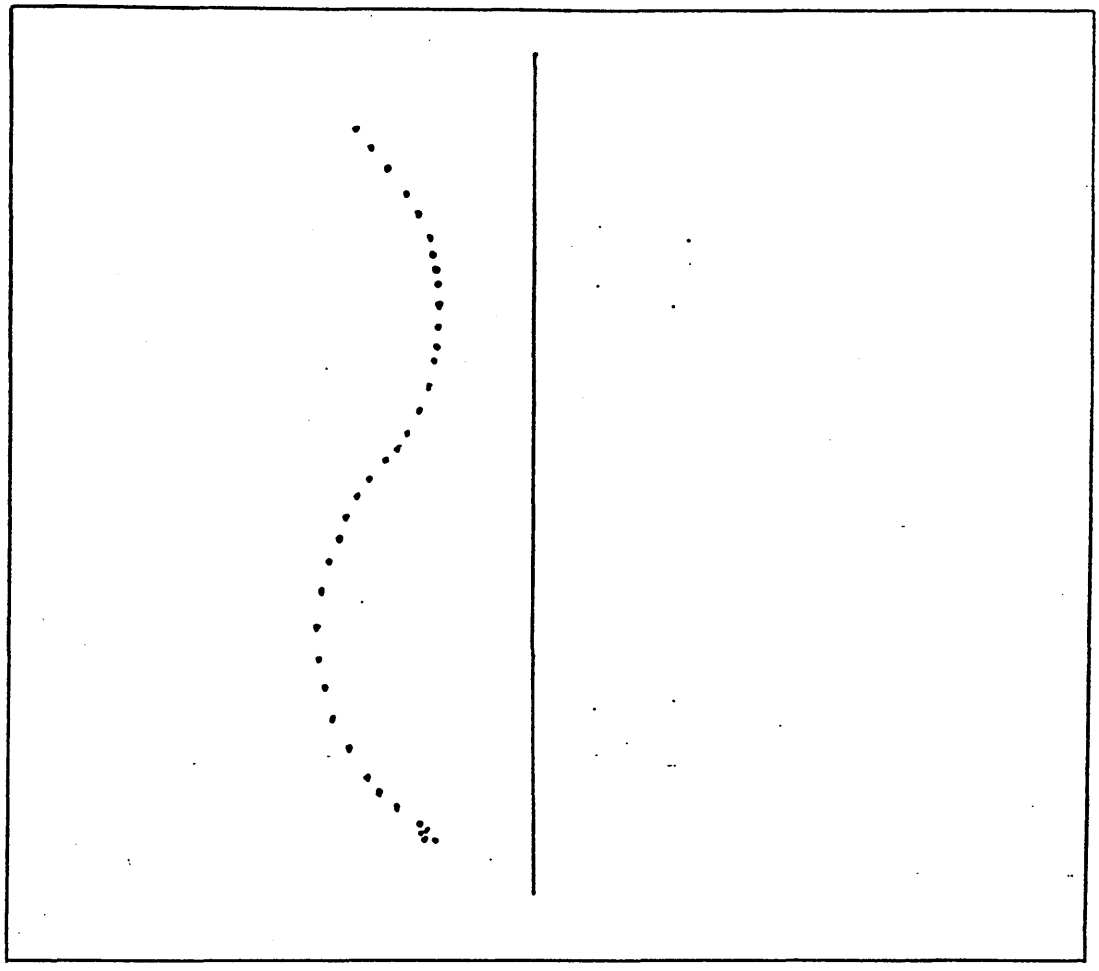


Fig. 1(a) : Vase defined by a contour and an axis of rotation.

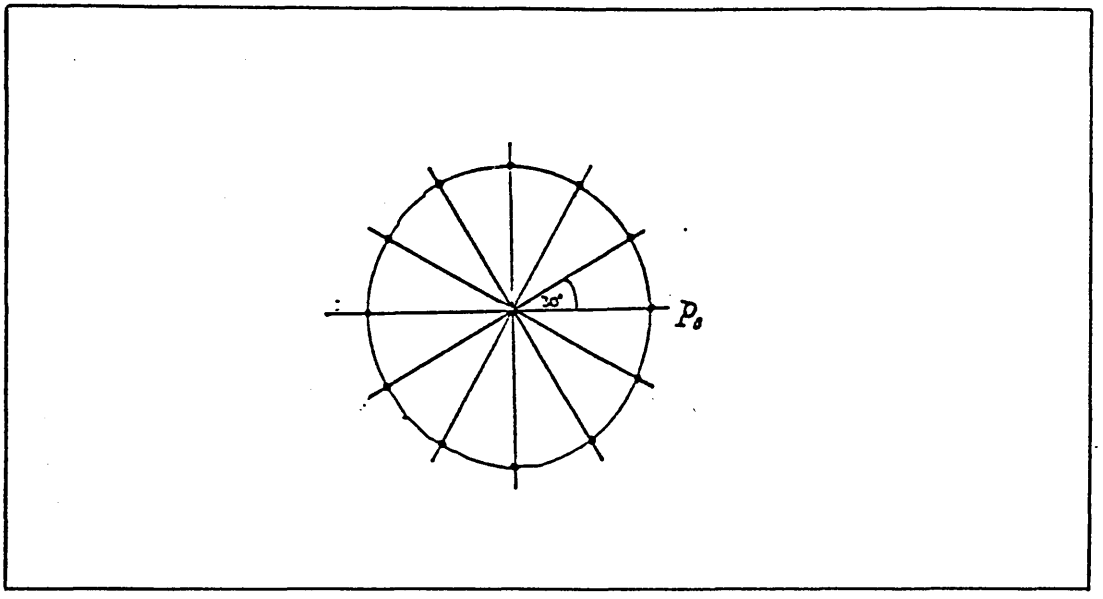


Fig. 1(b) : Point p_0 being rotated about axis of rotation 30° at a time.

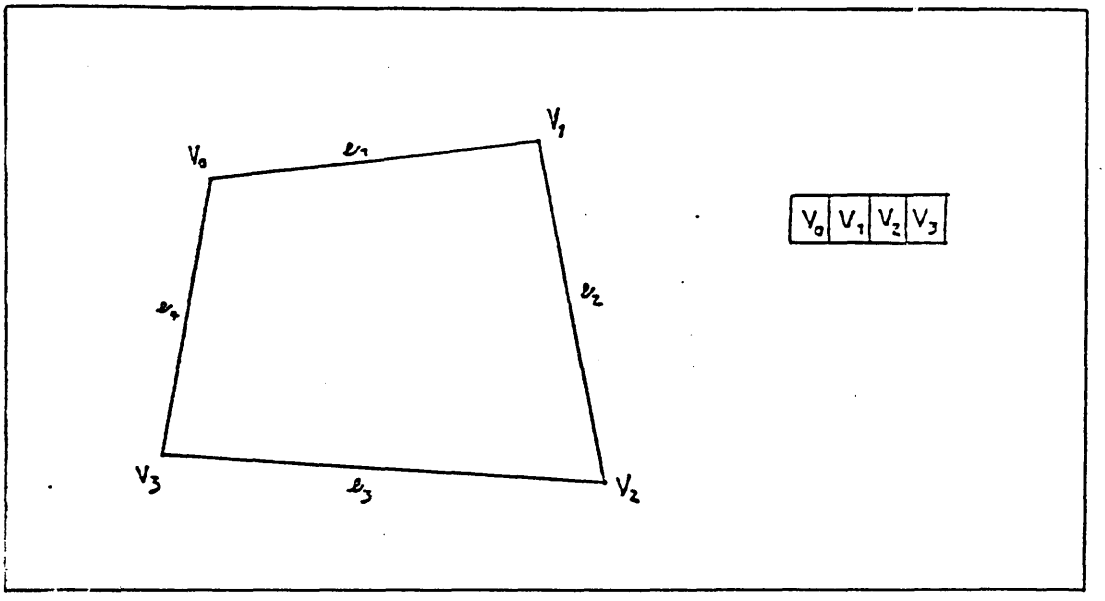


Fig. 2: The four vertices defining a facet are stored in order within the facet array element, enabling the facet's edges to be implied.

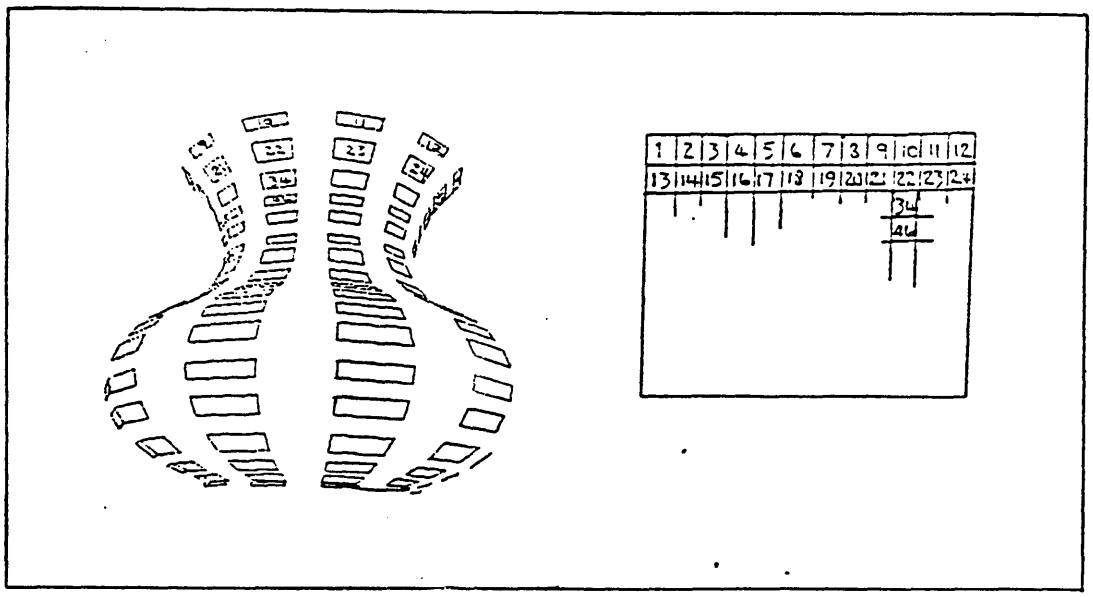


Fig. 3(a): One facet per group of four has all of its vertices stored in a single facet array element.

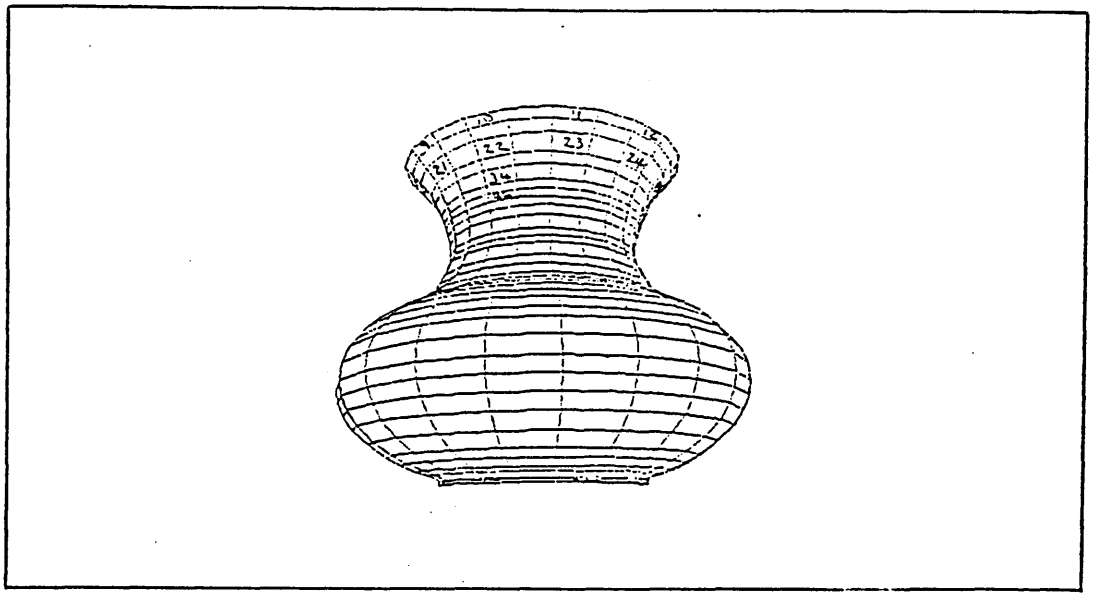


Fig. 3(b): Other edges are implied by the order of storage.

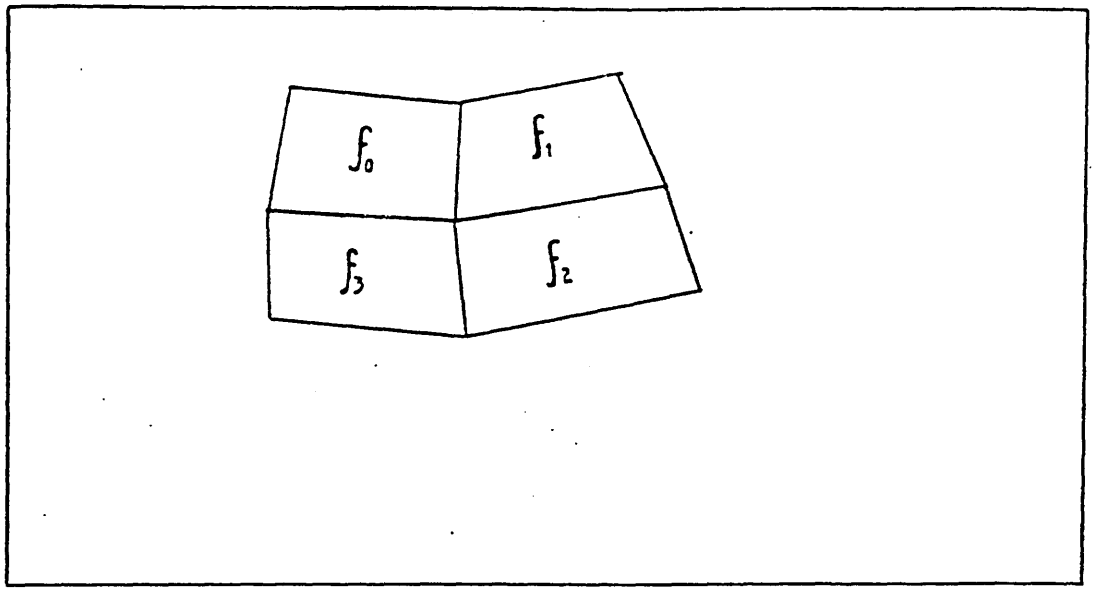


Fig. 4: The position of each of the four facets in the group.

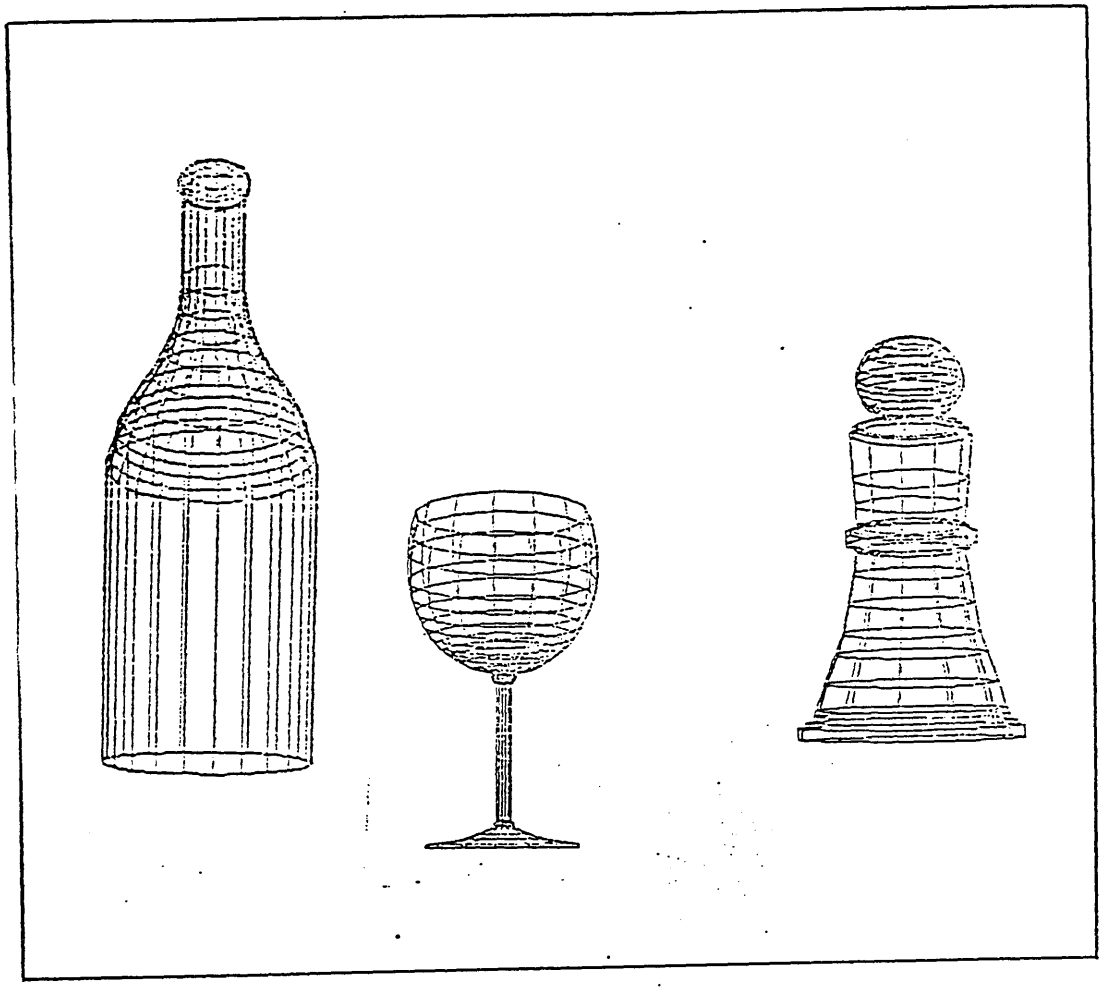


Fig. 5(a) : Objects with all facets drawn.

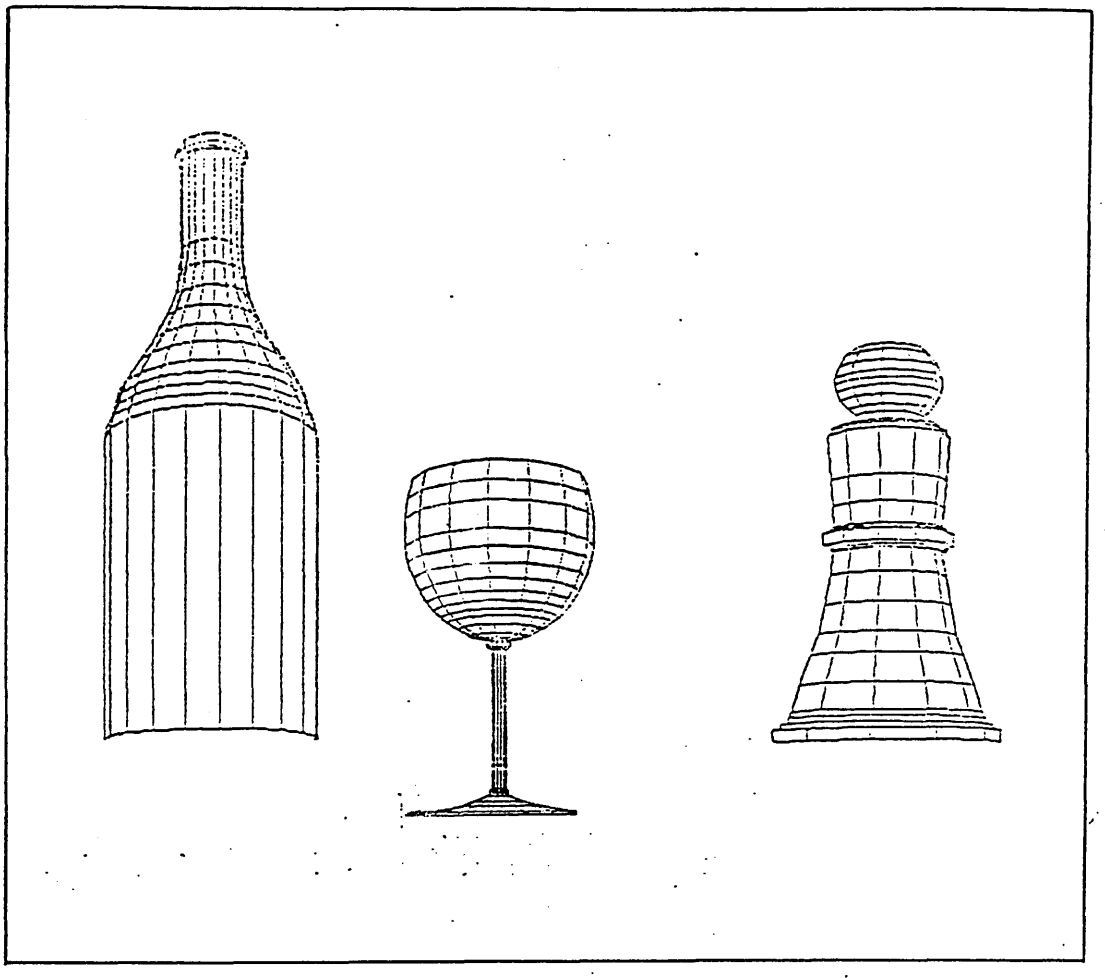


Fig. 5(b) : Objects with front-facing facets drawn.

APPENDIX D : PSEUDO ORDERING OF CSG-TREES.

The following paper was published as part of the EUROGRAPHICS 1988 Conference Proceedings Nice, France 12th - 16th September 1988.

PSEUDO ORDERING OF CSG-TREES

Marion S. COTTINGHAM

Department of Computer Science
Monash University
Melbourne, Australia

Using Constructive Solid Geometry (CSG) methods, it is usual for primitive object representations to be stored at the leaf nodes of binary trees. The major part of the work involved in generating an image of the object is finding what surface is visible at each pixel in the screen. Using conventional rendering methods this can be simplified by ordering the primitives by their screen positions and by their depths. Using ray tracing techniques this can be achieved by testing if rays intersect with primitives, the number of these intersection tests can be reduced by ordering. However it is not generally possible to order data (in any one direction) in CSG-trees where intersection or difference operators are involved.

This paper describes a method that enables 'local' three-way ordering of the data contained in CSG-trees that can be used with either conventional scan-line rendering methods or ray tracing techniques. This is achieved by the introduction of underlying data structures that dynamically change throughout the image generation step. Using this method, the primitive/polygon visible at a particular pixel can usually be accessed directly via pointers.

Keywords : Computer Aided Design (CAD), Constructive Solid Geometry (CSG), boundary evaluation, ray tracing, Compressed Data Structure (CDS).

1. INTRODUCTION

Constructive Solid Geometry (CSG) methods are used in most of the CAD/CAM packages currently available [1]. Certain classes of objects such as unsculptured mechanical parts can be easily created using these methods. Complex objects are represented by collections of simpler solids (or primitives). These primitives are typically blocks, spheres, cones, cylinders and torii that are combined using Boolean set operators union, difference and intersection. Internally these primitives and their associated operands are stored in a tree structure, called a CSG tree. This is typically a binary tree that has primitive objects stored at external nodes in the tree and their associated operands at internal nodes (see fig.1) The primitive objects are often defined by a second representation. The polyhedral approximation is a common way to represent the surface of the primitives at the leaf nodes.

After the CSG-tree has been built the data it contains is used to generate the image. A major part of this work involves identifying and removing hidden surfaces. There are many algorithms in existence for doing this. Sutherland et al categorised these into three classes object-space, image-space and list priority [2]. An 'object-space' method compares objects to determine which parts are visible, an 'image-space' method determines what is visible at each pixel and a 'list priority' method works partly in each space. Most of these algorithms sort surfaces either by screen positions (XY), by

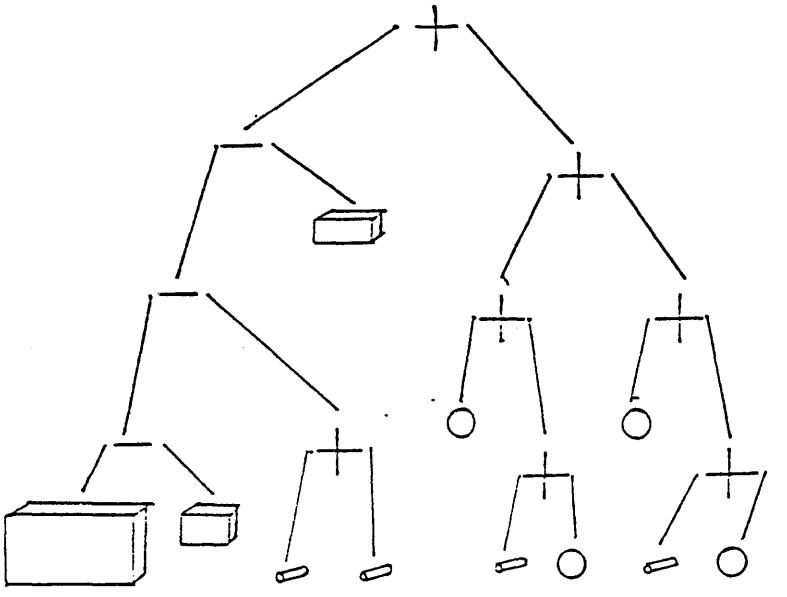


Fig.1 CSG tree representation

depth (Z) or by some other criteria. To increase efficiency some algorithms use scanline coherence (i.e. adjacent scanlines are very similar) or frame coherence (i.e. an image does not change much between frames in an animated sequence).

This paper introduces a method of accessing the data stored in CSG trees by scanline order, and by left to right pixel order within each scanline, and by increasing depth order within each pixel. This method also utilizes scanline coherence.

2. RENDERING METHODS

There are two main approaches to rendering the image of an object represented by CSG trees, conventional rendering methods and ray tracing techniques. Using conventional rendering methods the data must be preprocessed to provide a boundary evaluation for the whole complex object (see next section). Using ray tracing techniques, the tree is traversed by each sight ray to find the surface visible at a particular pixel (see section 2.2).

2.1 Conventional Rendering Methods

Generating an image using conventional methods requires computing a boundary evaluation for the complex solid. This usually involves traversing the CSG-tree from the bottom upwards, and at each branch combining the surface boundaries of their sub-objects according to the associated Boolean operand (see fig.2). The CSG-tree is conventionally a binary tree which restricts each evaluation to two sub-objects. After the boundary evaluation step has been completed, the root node contains the complete surface definition of the complex solid being modelled.

A common representation of the primitives is to approximate them by polyhedrals. This representation enables a smooth-shading technique such as Gouraud's or Phong's to be used [3, 4]. The boundary evaluation of a polyhedral representation is simplified to finding and updating affected

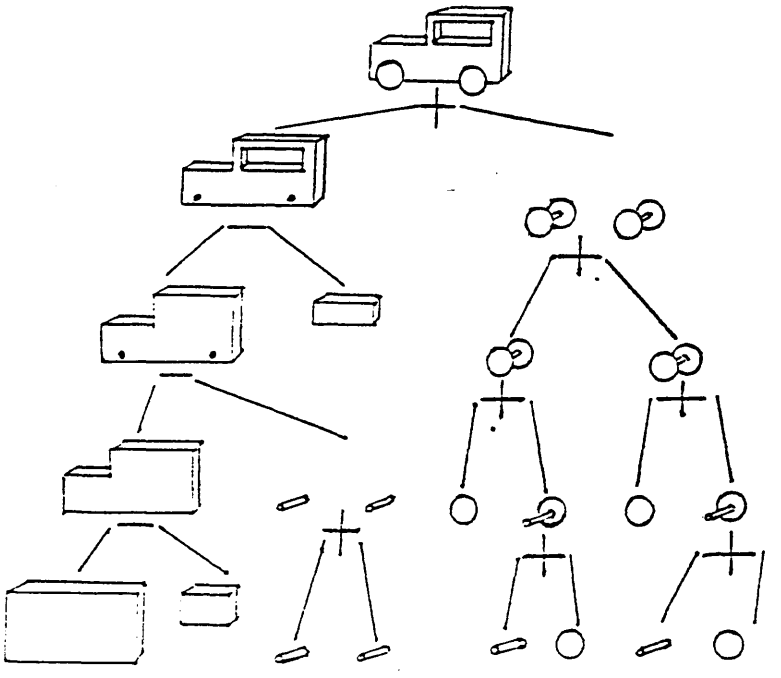


Fig.2 Boundary Evaluation

polygons, these are either deleted or re-evaluated according to the associated Boolean operands. Re-evaluating a polygon requires updating its existing edges by increasing/decreasing the number of vertices.

Ordering the leaf nodes by their primitives' maximum screen y-values can reduce the number of disjoint objects that have to be evaluated at any particular node. An unordered tree may have many disjoint objects at internal nodes resulting in several objects having to be considered simultaneously at any step in the evaluation. Figure 3(a) and (b) shows the ordered and unordered CSG-trees for the same object, with (b) showing a worst case with all the primitives remaining disjoint at internal nodes.

2.2 Ray Tracing Techniques

Ray tracing is to mathematically cast a ray from the viewpoint through every pixel in the screen and onto the objects in the scene. A high degree of realism can be achieved by bouncing the rays off reflective/transparent surfaces and tracing them through the scene until the light source is reached. However the mechanical-like objects normally represented by CSG-trees do not require such a high degree of realism, therefore this paper will concentrate on ray tracing for hidden-surface removal, providing an alternative approach to the boundary evaluation step of conventional rendering techniques.

Hidden-surface removal is achieved by finding the first surface intersected by each ray. This provides the visible surface at the pixel corresponding to the ray [5]. The major part of any ray tracing algorithm is the ray-surface intersection calculation.

Ray tracing a CSG-tree involves a bottom-up traversal of the tree for each sight ray, to find the intersection points where a ray enters and exits each primitive object (see fig.4). This decomposes the ray into a collection of 'inside-the-solid' and 'outside-the-solid' segments [6]. At each

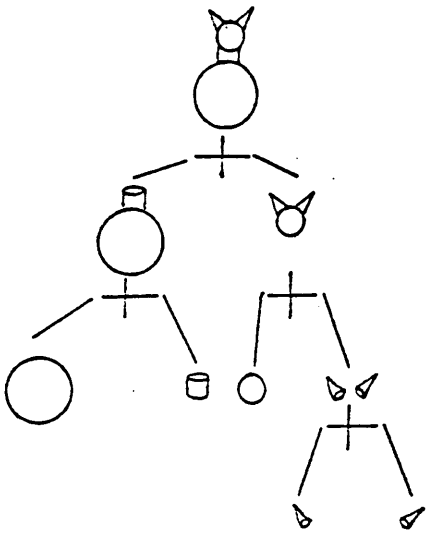


Fig.3(a) CSG-tree with nodes ordered

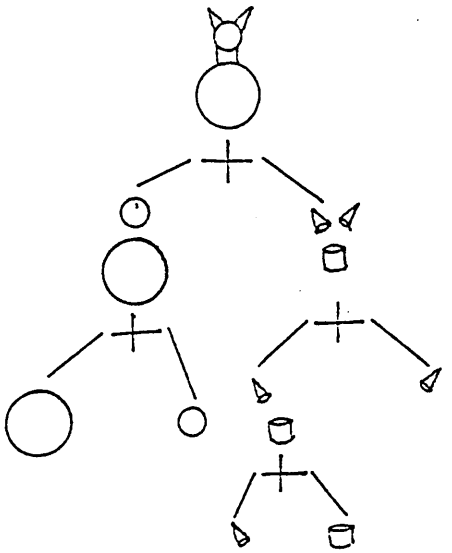


Fig.3(b) CSG-tree with nodes not ordered

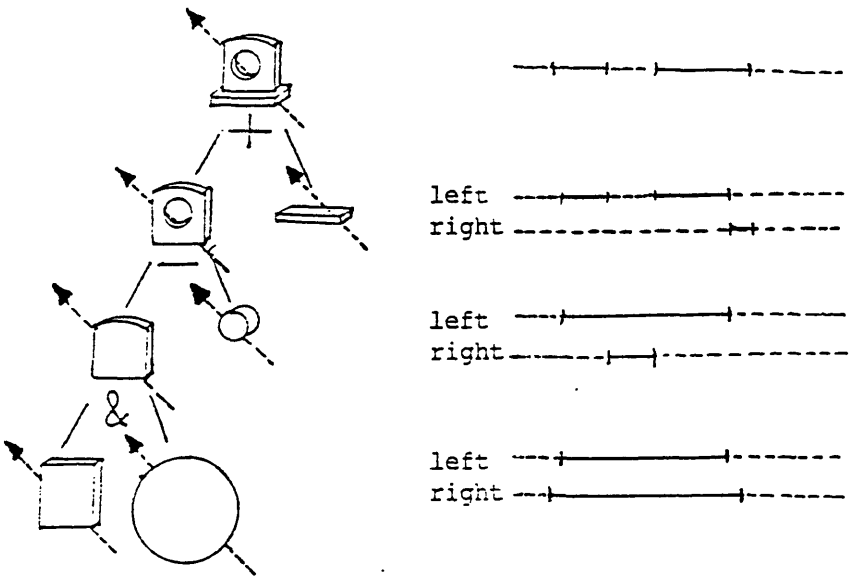


Fig.4 Ray Tracing a CSG-tree

level, these segments are combined according to the Boolean operands stored at internal nodes. After the whole tree has been traversed, the surface 'visible' at the associated screen pixel is determined from these segments.

Accessing all the CSG-tree nodes for every ray is expensive in computation. Roth [7] introduced box enclosures to reduce the number of node accesses required to process any one ray. A box enclosure is a rectangular area of the screen that encloses a primitive object. Before the drawing step the tree is traversed bottom-up and boxes combined at internal nodes according to the Boolean operands. After this traversal the box enclosure at the root node encloses the whole complex object. Figure 5(a) showsthe box enclosures for the CSG-tree shown in figure 3(a). If the pixel associated with a ray does not lie within a box enclosure then no further nodes in that branch need be accessed. This reduces the number of node accesses dramatically and rays are only intersected with primitives likely to provide a positive result.

In general box enclosures will be larger at internal nodes if the tree is unordered (see fig.5(b)). This implies that many more node accesses will be required for an unordered tree.

3. PSEUDO-ORDERING OF CSG-TREES

Using either ray tracing or conventional rendering techniques, it is more efficient if the CSG-tree nodes are ordered. There may be two levels of ordering to be considered, the primitive level and, for a polyhedral representation, the polygon level.

Considering the primitive level, if all the primitives in a CSG-tree are combined by union operators then leaf nodes can be strictly ordered by maximum y values (in screen coordinates). However, if any difference or intersection operators are used then strict ordering of the primitives is not generally possible; for correct results the positions of the two branches associated with either of these operators are static, making ordering impossible.

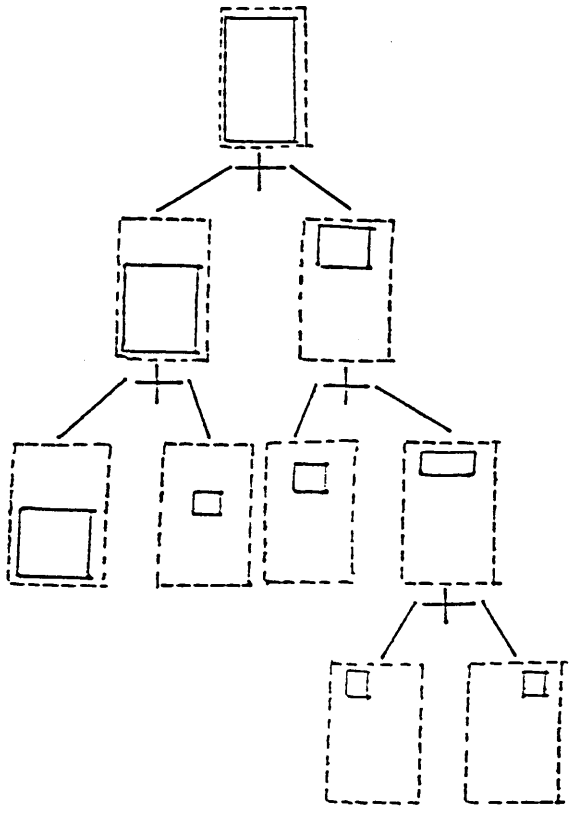


Fig.5(a) Ordered CSG-tree showing box enclosures

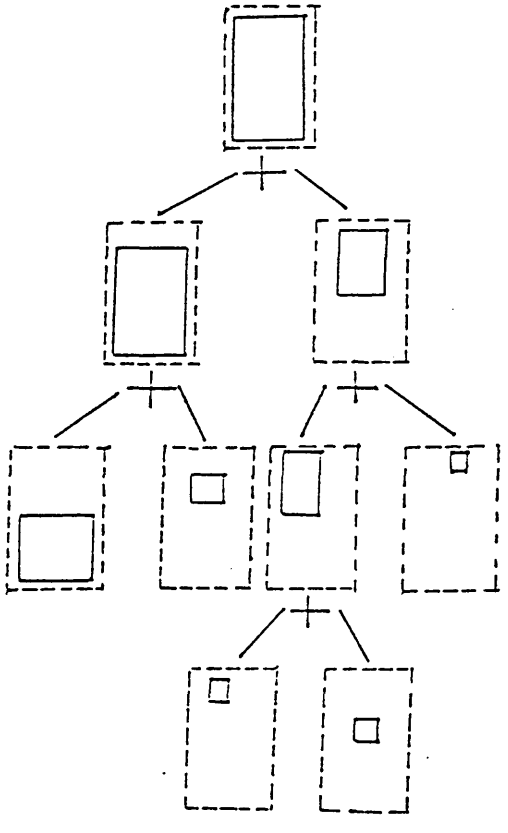


Fig.5(b) Unordered CSG-tree showing box enclosures

Assuming that ordering by **y** is possible, the next step is to order by increasing values of **x** local to each scanline. Generally it is impossible to build a binary tree that provides this two-way ordering for the whole scene. The main problem is that binary trees are inherently static structures. This paper introduces dynamic data structures that allow the CSG-tree primitives to be ordered in both the **x** and **y** directions for any particular scanline. The next section describes how this pseudo-ordering of the tree can be achieved dynamically throughout the rendering process.

If a polyhedral approximation is used to represent surfaces, additional underlying data structures can be used to access polygons in left to right pixel order within each scanline. These are introduced in more detail in section 3.2.

3.1 Ordering Primitive Nodes

The previous section outlines the difficulties in attempting to impose an order on the terminal nodes of a binary tree structure and how these difficulties can be overcome by the introduction of underlying data structures. The first of these is an index containing pointers to the primitives stored at external nodes (see fig.6). This enables ordering by **y**. This index is created as a linked list directly after the CSG-tree has been built and immediately before the rendering step (see fig.7). Index nodes are ordered by decreasing maximum **y** values (in screen coordinates) of primitives. This is made easier by employing Roth's method of box enclosures. Using an index, primitives associated with difference or intersection operators can still be strictly ordered since the CSG-tree nodes maintain their same positions. The order of nodes in this index is static throughout the rendering step although the number of nodes decreases as the rendering progresses and primitives have been completely processed.

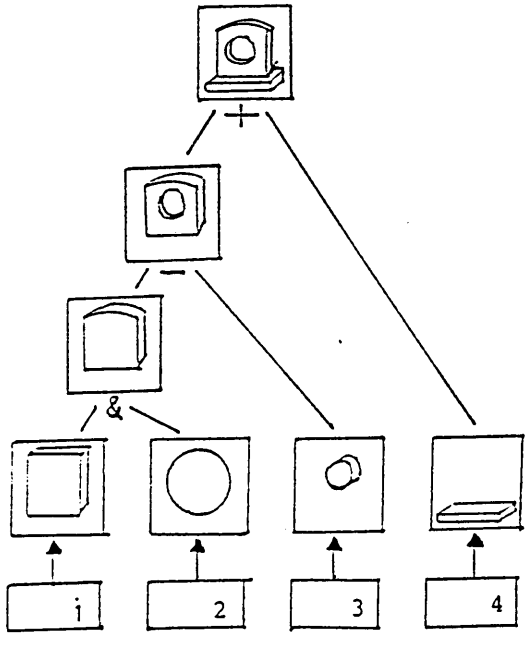


Fig.6 CSG-tree with index

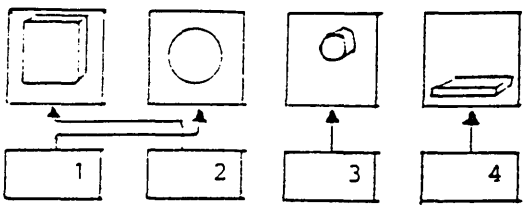


Fig.7 CSG-tree with index ordered by y

The second underlying data structure is the active primitive list. This enables primitives to be ordered by increasing **x** values within a scanline. For any scanline this list contains pointers to all the primitives intersected by the scanline. This active list is created/updated by transferring nodes from the beginning of the ordered index. Initially the first node in the ordered index contains the pointer to the primitive that is 'nearest' to the top of the display screen. This node is transferred to the active primitive list with any subsequent node that has its 'top' at the same level. The active list is ordered, in increasing order of minimum x box enclosure values, by inserting nodes into the list at their correct position. Nodes are removed when their box enclosures' minimum y values are incident on the previous scanline processed.

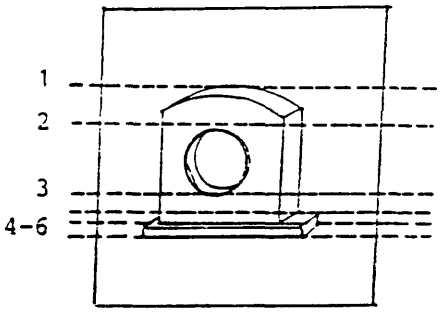
The active primitive list will normally remain constant over several adjacent scanlines and will only change when a primitive's top/bottom is incident on a scanline (see fig.8). The first entry of the ordered index provides the scanline where the next addition to the list will occur. The maximum 'bottom' of all the primitives in the active list provides the scanline where the next deletion from the list will occur.

The active primitive list is maintained throughout the rendering step, enabling each scanline to be divided into spans that are bounded by scanline/box enclosure intersections. Spans are discussed in more detail in section 4. A count is kept of the number of primitives associated with each span to enable overlapping primitives to be easily identified. This determines if any hidden-surface elimination other than the removal of back-facing polygons is required within a span.

3.2 Ordering Polygons

If a polyhedral representation is used, a third underlying data structure called a path provides fast access to visible polygons. A path is a linked list of nodes each containing a pointer to identify a polygon that is incident on the scanline. An individual path list is created for each primitive when it is transferred from the ordered index into the active primitive list and will remain in existence while the primitive is active (see fig.9). Path nodes are maintained in increasing order of the minimum **x** values of the scanline/polygon intersection points.

A path list will normally remain cosntant (i.e. contain nodes pointing to the same polygons, for a few scanlines, and will only change whenever a new edge is encountered. The initial creation of the path list entails finding the polygon(s) intersected by the current scanline. Because scanlines are generated in order, this is achieved by searching the data structure for the polygon containing the vertex with the maximum y value. It is assumed for this application that the polygons will all be convex, therefore there will be two edges



scanline	active primitive list
1	1,2
2	1,2,3
3	1,2
4	1,2,4
5	1,4
6	1

Fig.8 Active primitive list ordered by x for given scanlines

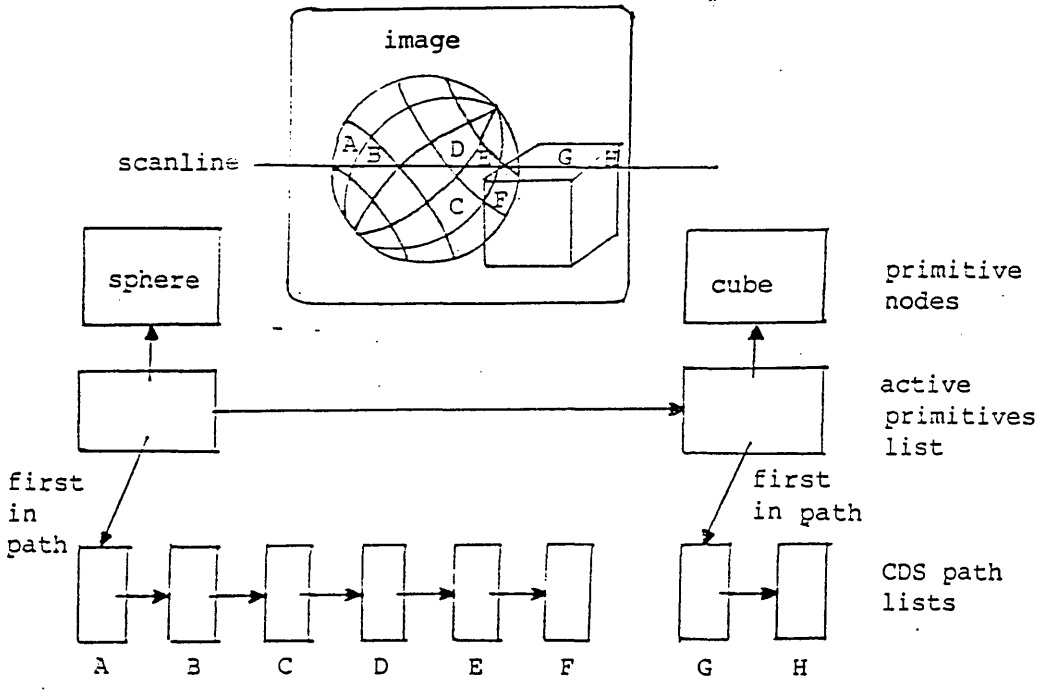


Fig.9 Active primitive list with associated path lists

intersected by any scanline. The following information is calculated and stored in each path node -

pointer to polygon incident on scanline.
the two edges intersected by the scanline.
the next path node in the list.

For each edge/scanline intersection point -
the x values in screen coordinates.
the z values in scene coordinates.
the depth values (if more than one primitive in associated span).

For updating between pixels -

the horizontal increment in depth values for both edges (if more than one primitive in associated span).

For updating between scanlines, the following is stored for the two edges -

the vertical increments in **x**.
the vertical increments in depth.

Additional information can also be stored for use by the smooth shading routines if required. For each path list there is a current pointer to indicate the polygon incident at the current pixel being generated.

Using conventional rendering methods a path node is used to generate the pixels lying along the scanline inside the associated polygon. If only one primitive is associated with the span containing a polygon, the enclosed pixels can be generated immediately. If more than one primitive is involved the current pointer for each path is followed and the depth value of each associated polygon is interpolated to reflect the depth at the current pixel. The polygon with the smallest depth determines the visible surface at that pixel.

Ray tracing techniques normally use an analytical definition of the surface and so would not usually require path nodes. Therefore ray tracing path nodes is not considered in this paper.

After each scanline has been completely processed, the information in the path nodes is updated to reflect the changes between scanlines. Each path node exists until the scanline reaches the bottom-most screen vertex of its associated polygon.

4. SPAN LIST FOR A SCANLINE

The span list is created by processing the active primitive list and dividing the scanline into spans with boundaries where the scanline enters and exits each active primitive's box enclosure (see fig.10(a)). The span list changes whenever the active primitive list is updated. The span list reduces the hidden surface problem to determining the visible primitive from several possibly visible primitives in each span of the scanline. There are three types of spans possible -

1. empty span (spans 1 and 7 of fig.10(a)).
2. span containing one primitive (spans 2 and 6 of fig.10(a)).
3. span containing more than one primitive (spans 3, 4 and 5 of fig.10(a)).

A span node is created for each non-empty span of the scanline. These are stored as a linked list (see fig.10(b)) and contain the following information -

start and stop x values for span.
number of primitives active in span.
pointers to first and current nodes in the path list (for each primitive).
pointers to the next node in the span list.

The span list is traversed during the generation of a scanline, providing access to the visible polygons in the path lists. There is one or more path lists for each node in the span list.

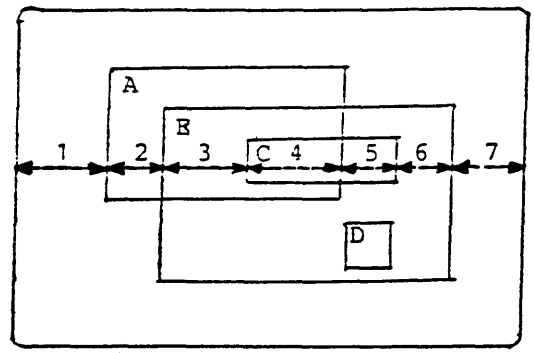


Fig.10(a) Spans of a scanline

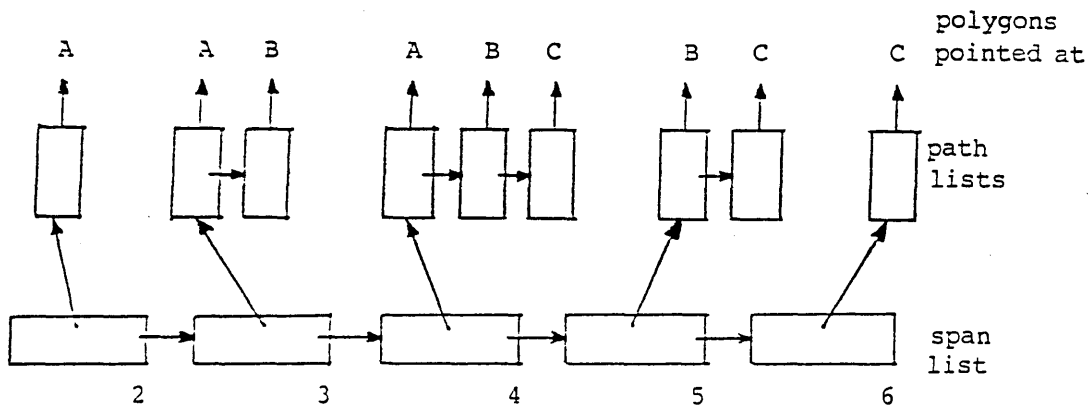


Fig.10(b) Span list with associated path lists

5. DATA STRUCTURES FOR POLYHEDRAL REPRESENTATION

There are several data structures in existence for storing the polyhedral representation. One of these is the compressed data structures (CDS) [8, 9] which uses a 2-dimensional array for storage. Each element contains the data required for one polygon. Using the CDS a minimum amount of work is required for one polygon. Using the CDS a minimum amount of work is required for ordering and searching because of the inherent nature of the data structure i.e. adjacent polygons in the scene are also 'adjacent' in the data structure. This is also an advantage when moving from one scanline to the next, when paths are updated to reflect the polygons that are incident on the new scanline. Another more general data structure is Baumgart's Winged-Edge [10]. This could also be used, but may require more searching to set up the path nodes.

6. CONCLUSIONS AND RESULTS

The underlying data structures enable pseudo-ordering of the primitives stored at CSG-tree leaf nodes. This enables primitives active on a scanline and the polygons incident at a pixel (if required) to be easily identified and directly accessed. Accessing the tree from the bottom upwards, the primitives are

accessed first, therefore the associated Boolean operand need only be found for the nearest 'visible' primitive. most of the information stored in the underlying data structures are pointers, these only require half or one word of storage depending on what hardware is being used. Accessing the data directly makes it feasible to generate the image in scanline order.

A Pyramid computer, linked to a Vectrix display device, was used to compare the time required to generate the image of a sphere. The sphere was approximated by a polyhedron consisting of 140 facets with front-facing facets shaded by 150 different shading intensities. The sphere covered an area of approximately 35,000 pixels. Gouraud's smooth-shading technique was used to interpolate the shading intensities. Using conventional methods, and shading one facet at a time, took 8 mins. 20 secs. to generate the image. Taking full advantage of the coherence properties and the direct accessing of data, that the underlying data structures described in this paper allow, enabled this time to be reduced.

REFERENCES

[1] Requicha, A.A.G. and Voelcker, H.B., Solid Modelling: A Historical Summary and Contemporary Assessment, IEEE Computer Graphics & Applications (March 1982) 9.

[2] Sutherland, I.E., Sproull, R.F. and Schumacker, R.A., A Characterization of Ten Hidden-Surface Algorithms, Computing Surveys (March 1974) 1.

[3] Gouraud, H., Computer Display of Curved Surfaces, Univ. of Utah, Comp. Science Dept., Utec-CSc-71-113 (June 1971).

[4] Newman, W.M. and Sproull, R.F., Principles of Interactive Computer Graphics, (McGraw-Hill, 1979).

[5] Atherton, P.R., A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry, Computer Graphics (July 1983) 73.

[6] Myers, W., An Industrial Perspective on Solid Modelling, IEEE Computer Graphics & Applications (March 1982) 86.

[7] Roth, S.D., Ray Casting for Modelling Solids, Computer Graphics and Image Processing (18, 1982) 109.

[8] Cottingham, M.S., A Compressed Data Structure for Surface Representation, Computer Graphics Forum (Sept. 1985) 217.

[9] Cottingham, M.S., Compressed Data Structure for Rotational Sweep Method, AUSGRAPH 87 Conference Proceedings (May 1987).

[10] Baumgart, B.G., A Polyhedron Representation for Computer Vision, Proceedings AFIPS National Computer Conf. (1975) 589.