Technical Report

UCAM-CL-TR-235
ISSN 1476-2986

Number 235

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Modelling and image generation

Heng Wang

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

ISSN 1476-2986

Summary

Three dimensional (3D) volume representation, processing and visualisation have gained growing attention during the last ten years due to the rapid decrease in computer memory cost and the enhancement of computation power. Recent development in massive parallel computer architectures and special purpose graphics accelerators also contributes to solve 3D volume manipulation problems which usually require large memory and intensive computation. Volumetric graphics is becoming practically possible and finding many applications such as medical image processing, computer aided design and scientific visualisation.

A volumetric object is usually represented in one of two forms: a large 3D uniform grid of voxels (volume elements), and a relatively compact non-uniform collection of volumes. Objects in the latter form are obtained by adaptive recursive decompositions. An octree is a special case in which each non-terminal volume is subdivided into eight sub-volumes. The problems of current implementation of octrees concern the speed and complexity of memory management. This dissertation looks into a novel approach of designing octree-related volumetric graphics algorithms based on Content-Addressable Memories (CAMs). A CAM is an architecture consisting of elements which have data storage capabilities and can be accessed simultaneously on the basis of data contents instead of addresses. It is demonstrated that the main features of CAMs, their parallel searching, pattern matching and masked parallel updating capabilities, are suitable for implementing octree-related algorithms.

New CAM algorithms are presented for transforming octrees, evaluating set operations (union, intersection, difference), displaying volumetric objects, calculating volumes, constructing octrees from other representations, and so on. These algorithms are remarkably simple and conceptually intuitive. The simplicity plays an important role in constructing robust 3D solid modelling systems. In addition to their simplicity many algorithms are more efficient than their conventional counterparts.

A new method has been developed to speed up the image synthesis algorithm of ray tracing using CAM octrees. It is aimed to reduce the number of ray-object intersection tests without significantly increasing overhead costs of storage and computation which are related to octree data structures and their traversals. The simulation results verify the expected improvements in speed and in memory management. Ray tracing can be accelerated by applying parallelism. Preliminary analysis shows possibilities of implementing the above CAM octree ray tracer on general parallel machines such as MIMD (Multiple Instruction stream, Multiple Data stream).

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Three dimensional modelling and image generation play an important role in computer graphics, image processing, and many scientific and engineering fields such as computer-aided design and computer-aided manufacture (CAD/CAM). During the last twenty years, extensive research has been pursued on modelling and image generation. Many problems in these two fields were identified and experimental systems were developed to meet increasing demands from applications.

The central problems concerning geometrical modelling systems are the need for powerful computational means to represent and process solid objects, input and output objects (for example editing objects and displaying them on screens). Graphical data have three primary forms: as a continuous geometric model, as a discrete pixel (pictorial element) image, and as a discrete voxel (volume element) image. The first form is used to represent objects in abstract concepts. The second form is an important means for visualising the abstract data on a raster graphics system. Algorithms related to these two forms have been extensively studied in early years of computer graphics research and are established as fundamentals of the subject. Unlike the previous two forms, voxel images have only gained attention of researchers in recent years. Although a lot of progress has been made in model representations and processing, more research is still needed particularly for improving system performance, geometric coverage, application coverage. Current approaches to solve these problems are to adopt high-level multiple-representations in geometric modelling systems and to use specialised accelerating hardware for different representations when suitable.

This chapter gives a brief review of three dimensional (3D) modelling schemes and their basic features. Requirements of 3D object processing and displaying from applications are summarised. Following these discussions we look into the main problems of current systems then propose a new approach of modelling on the basis of content-addressable memories. Finally comes an overview about the chapter organisation of the thesis.

1.1 Geometric Modelling

Geometric modelling is concerned with representing physical objects by symbolic structures called data. More specifically, 3D models are used to describe geometric properties of objects in 3D, and store them in the computer as some data representation. These data are manipulated using geometric algorithms to compute the properties of objects. The choice of representation strongly influences the design of a complete geometrical modelling system. The basic requirement for a good representation of an object is that it should have descriptive power and must be valid, complete, easy to be created and efficient in execution. Several representation schemes are widely adopted for 3D modelling. Each of them has some advantages or disadvantages and is suitable for different applications. They are listed below following the definition by Requicha [Requ80] and Voelcker *et al.* [Voel88].

1. **Wireframes:** A wireframe represents a solid using segments of 3D space curves. It is generalised from 2D graphics of lines and arcs. The scheme is used for historic reasons such as easy plotting and so on. Wireframes have two serious deficiencies: ambiguity and representing invalid solids. Thus they can not be used as primary representations.

2. **Boundary representations (Breps):** Breps are schemes in which a solid is represented by sets of faces that enclose it completely. These faces can be polygons for plane surfaces or patches in the case of curved surfaces. Each face is represented by its bounding edges or vertices. Breps have advantages for generating line drawing or raster images on a graphic display, for supporting graphic interaction such as picking up an edge or a vertex.

3. **Pure primitive instancing schemes:** The schemes are based on the notion of families of objects, each member of a family is distinguishable by a few parameters. Each object family is called a generic primitive. It can be a block, a cylinder, a cone, a sphere, a torus, and so on. These primitives are unambiguous, unique, concise. However, it is difficult to combine instances from these object families to create complex objects. Another drawback is that there is usually no uniform algorithm available for all object families.

4. **Constructive Solid Geometry (CSG):** A CSG model represents a solid by regularised Boolean combinations including unions, differences, intersections and complement of simple primitives. The model is presented in application programs as a binary tree whose nonterminal nodes are set operators and terminal nodes correspond to primitive solids. These primitive solids have two forms: one is based on pure primitive instances as described above, the other is based on general half-spaces. The virtue of CSG models is that they can be easily used in designing and generating certain types of objects which are often used in CAD.

5. **Sweep representations:** Sweep representations are schemes in which a solid is represented as a spatial region traversed by a plane or a moving object.

They have restricted description power and are used only on some occasions.

6. **Spatial occupancy enumeration schemes:** This is an approximated representation of a solid using a union of box-shaped cells occupied by the solid. The cells may be of a uniform size like 3D spatial grid or of non-uniform sizes generated by recursive decomposition. There are many structures to represent varying sized cells. One of them is a tree structure called an octree. Spatial occupancy enumeration schemes are unambiguous but storage expensive. They do not exactly represent geometric objects. Thus they are not suitable for internal representation in solid modelling systems, but are usually used as auxiliary tools. These schemes are useful in image modelling and processing, computer vision.

7. **Cell decomposition schemes:** Cell decomposition schemes are more general forms which represent a solid as a union of quasi-disjoint cells. Cells can be tetrahedra or other shapes. Typical cell decompositions are solid triangulations and finite element meshes. The latter ones are important representations used in the 3D finite element analysis for getting numerical solutions of differential equations. These schemes are unambiguous but not unique. They are also difficult to construct, especially for curved solids.

In defining the above schemes, we have assumed that objects are rigid solids. However natural objects include many non-rigid objects such as water and other deformable solids. In this thesis, representations of non-rigid solids are excluded.

Summarising the above representations, CSG models and Breps are suitable for modelling in CAD systems, and space occupancy enumeration models are means for 3D image processing. Primitive instancing and sweeping are used in both cases. Other schemes are found useful for some special applications.

1.2 Processing and Displaying Solids

The main purpose of modelling objects using the computer is to manipulate and to visualise them. We want to move and scale objects, calculate volumes, momenta and other properties of objects, finally display them on some media. Therefore algorithms in a solid modelling system must include model construction and maintenance, geometric transformations, boolean compositions, representation conversion, and visualisation. Many graphical processing algorithms involve some kind of search operations such as geometric searching, spatial indexing, intersection testing, and so on. The efficiency of searching operations determines the efficiency of many algorithms.

Algorithms for solid visualisation are also important parts of the core of a solid modelling system. Early graphics visualisation is based on line drawing due to constraints of display devices such as the storage tube CRT display and the random scan refresh display. These devices have been obsolete for a long time. Modern display devices are mainly raster CRT devices which allow shaded images and realistic images to be generated. These images give us the impression of solid areas, let us identify different material properties of objects and various lighting phenomena.

However, the computation cost increases tremendously with raster images especially for complex images. Many algorithms are of O(n^2) (n is the number of objects) complexity when they are implemented straightforwardly. Graphical system developers attempted to gain speed either by exploiting spatial locality of geometrical operations (for example [Tamm81]) or by using specialised hardware such as geometric engine and solids engine [Meag84]. By exploiting spatial locality, average performance of O($n \log n$) can be achieved for some algorithms. Still, we can only achieve performance of interactive time with these improved methods. In future, we shall expect computer graphics systems to generate sequences of high quality images in real time. In order to process and generate images in real time, two approaches are being pursued. One is a short term strategy and the other is a long term strategy.

The current short-term strategy is based on graphics supercomputers and specialised graphics co-processors. Examples of the state-of-the-art systems are Silicon Graphics IRIS 4D [Akel88] and Stellar GS-1000 [Apga88] and many other graphics workstations. These workstations serve mainly as high-performance polygon rendering processors and solve display problems. They support line drawing and polygon shading which have limited expressive power and can not generate images with complex lighting and motion blur. At the current stage, only computationally intensive ray tracing algorithms can handle such images in a simple and elegant way.

Computer graphics is a rapidly moving field. Its horizon changes with the growing power of general-purpose computers. It is very likely that graphics workstations will become widely used and next generation display systems will not be limited to simple shading. When real time ray tracing becomes available, hardware for simple shading may become as obsolete as storage tube display.

Solid modelling systems require not only display but also other functions such as set operations, mass computation, and so on. To implement set operations efficiently, schemes which are good at spatial locality should be employed. Jackins *et al.* [Jack80] and Meagher [Meag82] described the use of octrees for representing three dimensional objects of any shape (concave, convex, sculptured, and so on) to any specified resolution. These representations satisfy uniqueness, completeness and other requirements. They are also efficient in set operations (union, intersection and difference), geometric operations (translation, scaling and rotation), hidden surface removal and other operations such as calculating the volume and the center of gravity of an object. However, representing objects in octrees requires a large memory space and is only an approximation of objects. These two deficiencies limited the practical usefulness of octrees as the internal representation for solid modelling systems. Therefore most systems adopt an approach which uses Brep (and/or CSG) as internal representation and converts Brep (and/or CSG) to octree when an application demands it.

The long-term strategy of real time graphics is to exploit parallelism of graphics algorithms. The parallelism can be realised either through general-purpose multiprocessors or through specialised architectures optimised for applications. The advantages of using general-purpose multiprocessors are that we can accelerate algorithms for a wide range of applications including both processing and displaying. Moreover, users can easily update systems to fit the state-of-the-art algorithms.

Apart from the computational intensive problem discussed above, another factor that affects current systems of modelling and image synthesis is the software complexity. Graphics problems are so complicated that it is very difficult to write algorithms which are both efficient and simple. To guarantee the correctness of programs is even difficult. This is a serious problem which increases the cost for long-term software maintenance and makes the job of generating a robust system difficult. We must aim at keeping the design of software as simple as possible. The complexity of a solid modelling system comes from the following sources,

- Multiple representations are used in one system.
- Thus the system needs algorithms for operating on each representation and for doing representation conversion.
- Multiple representations may introduce inconsistency. Consistency checks are both complex and time consuming.
- Algorithms are mainly based on the von Neumann model of computation. Some of them are complicated due to the sequential nature of the von Neumann machine.

Among the current problems identified above, key issues concern computation time, memory management and software complexity. Most graphics systems involve a certain number of searches. Thus we need to find a kind of computer architecture which allows us to cope more efficiently with these problems. In 1950's, an architecture called associative memory was proposed [Slad56] and has been developed since. It is now known as content addressable memories (CAM). While a detailed description will be given in Chapter 3, we note briefly here that a CAM is an architecture which accesses the data by their content instead of addresses. It is well suited for searching and simple memory management. We are thus motivated to assess its potential in improving our 3D graphics systems.

1.3 Scope of the Thesis

This research explores efficient CAM algorithms for octree applications of modelling and space indexing. Instead of designing algorithms by breaking them down into sequential form, we construct our octree algorithms in terms of larger conceptual units of the task. With the assistance of CAM architectures, some common problems may be less important while others may become more significant. These problems are identified with respect to our octree applications. The advantages and limitations of CAM will be investigated.

In Chapter 2, we review octree data structures and algorithms, classify data types which include raster type and vector type octrees, then look at the three principal formats of octree representation, and finally analyse the space/time efficiency problems related to these data formats for several major operations. Chapter 3 introduces the concept of CAM. We first give an overview of general CAMs and then concentrate on a specialised Syracuse CAM which is designed for applications of quadtrees/octrees [Oldf87]. The main features and functions of Syracuse quadtree/octree CAM will be summarised. The storage of both raster and vector octrees in CAM are presented. This chapter serves as the foundation for the remaining chapters.

Although quadtrees [Will88b] and octrees are closely related in terms of the concept and implementations, they are different in many aspects in applications. Chapter 4 investigates general CAM octree algorithms for constructing octrees from other representations, displaying them with hidden surface removal, viewing cross sections of objects, calculating volumes, transforming objects and so on.

Chapter 5 applies CAM octrees to ray tracing algorithms, the latter being powerful but time expensive techniques for generating realistic synthetic images. Many researchers have used some kind of space subdivisions to speed up ray tracing by partitioning the object space and sorting objects in a space order. However, these accelerating algorithms based on conventional machine architectures suffer some problems. The major one is that they either require extensive memory or have slow space traversal. Another problem concerns their complexity in memory management. It will be shown that these problems can be readily resolved using CAM octrees. A simple and elegant CAM algorithm is proposed and tested. The new algorithm is designed with both memory and speed considerations in mind.

Chapter 6 deals with the specific CAM problem of reading back multiple responders after a search operation. We examine the importance of ordered-retrieval of CAM words in octree applications. The octree related orders are clarified. Then we discuss ordered-retrieval of CAMs with trit (three states) storage. Examples are used to demonstrate the usefulness of ordered-retrieval.

Chapter 7 concludes with a summary of the new material presented in this thesis. This chapter also includes a preliminary analysis on architectures for parallel processing of ray tracing. We also discuss CAM architectures in relation with other general-purpose parallel machines, and consider how CAM octree ray tracer should be configured into parallel architectures of multiprocessors or multicomputers. Finally some suggestions for future research are given.

Chapter 2

Modelling with Octrees

This chapter reviews the hierarchical data structure—the octree. An octrees is a three dimensional(3D) generalisation of a binary tree. A binary tree is a tree in which each node is pointed to by a node called its parent and the node itself may be a parent of two nodes which are specified by a left link and a right link. An octree is a tree of degree eight, each parent node has eight children. A parent node is also a nonterminal node. A terminal node (also called a leaf) contains the information about the object. Octrees are classified into two types according to contents of leaves and rules of decomposition. They are raster octrees for volume data and vector octrees for surface data. An octree can be stored in different formats depending on requirements of applications. Formats of octrees are distinguished by whether they have pointers or not. Three commonly used formats are one with a pointer structure, and two with pointerless structures. The first of the two pointerless octrees is called a treecode. A treecode is a simple linear structure which stores an octree in depth first order. The second pointerless octree is called a leafcode and is based on an octal numbering system. Algorithms related to octrees have been developed for different octree types and formats.

Octrees have been used as representation techniques in many application areas such as 3D image processing, volume data processing, solid modelling, computer vision and robotics. They are also useful for accelerating image generation in computer graphics.

We first introduce surface based imageries and volume based imageries, and review the data structures of octrees and algorithms for octree manipulations and constructions. We then analyse and compare three major formats of octrees, identify the problems related to the system performance of octree algorithms under conventional computer architectures. These analyses lead to adaptation of an architecture of content-addressable memory (CAM) which is described in the next chapter.

2.1 Introduction

Chapter 1 introduced briefly several techniques for 3D object representations. These representations play crucial roles in much computer-aided scientific and engineering research. They also influence strongly the software organisation of application systems and the efficiency of data storage and manipulation. Thus, the choice of representation schemes will determine the overall performance of a system.

Different applications require different representations. For example, in solid modelling systems the core representations are Breps and CSG while in 3D image processing systems the essential schemes are spatial occupancy enumeration and cell decomposition.

The difference between systems is also reflected in input data. The input to a system can be classified into two groups: data-generation based input and data-collection based input. The data-generation based input data is created by users through specifying the parameters for primitive geometries. It is often used in computer generated images and synthesised pictures that come out of mathematical equations. The data-collection based input data is real data about real objects. The data is represented by 3D arrays of numbers. It is obtained using modern image data collection techniques such as computer tomography, scanning electron microscopy, and digital stereoscopy. It is widely used in the computation and display of 3D structures of scenes, human organs, and many other real objects.

2.1.1 Surface-Based Imagery vs. Volume-Based Imagery

Three dimensional computer graphics representations can be classified according to two distinct data formats: surface data and volume data. The former is closely related with the above described data-generation based input. The latter corresponds to data-collection based input. Surface data represent a 3D object by describing boundaries between the object and the space, or boundaries between different materials by using lines, polygons, patches and other abstract geometrical concepts. They are very useful for visualising 3D objects because many traditional display algorithms deal with lines and polygons.

In recent years, the second format—volume data has received considerable attentions [Fuch89]. One reason is that it is sometimes difficult to convert volume data obtained by data-collection techniques into surface data. Algorithms for extracting geometries from 3D arrays are often complex. The results are sometimes ambiguous and manual intervention is necessary. The direct and straightforward manipulation of volume data is attractive for its simplicity and robustness. The volume data is unambiguous, and can represent both rigid objects and material mixture models [Dreb88]. Other factors also contribute to the usefulness of volume data.

- Firstly, 3D measurements are advanced thus allowing very high resolution 3D data to be recorded. The new devices of solid state cameras and lenses, and improved microscopes give more accurate approximated data to real objects.

- Secondly, new algorithms called volume rendering have been developed [Dreb88]. These algorithms make full use of 3D arrays of volume data, rather than using surface data found in such arrays. Such systems process and display directly with volume data.

- Finally, several new memory and processing architectures have been designed for 3D voxel-based imagery [Meag84, Kauf88]. These architectures make it possible for real time visualisation of large amounts of 3D data.

2.1.2 Uniform Subdivision vs. Hierarchical Subdivision

Two kinds of architectures have been designed for processing volume data. One is based on a non-hierarchical model which stores unit cubic cells (voxels) in a large 3D cubic frame buffer. The other is based on hierarchical octree encoding. We call the former a voxel model and the latter an octree model. In weighing voxel approach and octree approach, the primary considerations are speed and memory requirements.

Voxel-based systems are simple because they use a uniform array structure. Objects are stored in a cellular cubic memory with a real and discrete model. Objects are manipulated directly using voxel mapping and transformation. Since the number of cells are fixed in a voxel model, projections, manipulations and rendering are independent of the scene complexity. When a voxel-based approach is used for image synthesis, traditional hidden surface removal algorithms for surface data are no longer necessary. Other advantages of the voxel approach is that the number of voxels along each direction is not necessarily a power of 2. The numbers can be different along different axes. For example, in 3D medical data we can have a resolution of $256 \times 256 \times 100$.

However, the voxel-based approach requires both extensive memory and intensive computation. A typical cube frame buffer, with a moderate resolution of $512 \times 512 \times 512 \times 8$, requires 128 Mbytes of memory for data alone. For higher spatial resolution and 24-bit colour images, which often occur in computer graphics, Gigabytes of memory is necessary. On the other hand, the extremely large throughput of data forms the bottle-neck for performance of such systems. Therefore, the real potential of voxel-based systems rely on decreasing in the cost of memory and increasing in the compactness and speed of memory. Massive parallel processing architectures are also important.

The octree approach, on the other hand, organises blocks of voxels in a hierarchical manner. At the bottom level, each block represents a group of voxels of the same colour. This leads to a tremendous saving in memory and much less throughput of data. The tree structure also allows much quicker traversal than a sequential voxel array traversal. In the remaining part of the thesis, the octree-based approach will be studied in further detail.

2.1.3 Systems for Volume Data Processing

Systems in the category of the voxel-based approach are GODPA (Generalised Object Display Process Architecture) [Gold85], PARCUM (Processing ARchitecture based on CUbic Memory) [Jack85] and CUBE [Kauf88]. All these systems are hardware oriented. Systems with octree encoding are normally software oriented or combine hardware with software. One such example is the Insight system of Phoenix Data Systems [Meag84].

- GODPA is a voxel processor integrated into a complete physician's workstation. It has real time implementation of rotation, scaling, translation on grey scale data. The voxel processor is based on a multi-processor architecture. There are sixty four processor elements (PEs), each contains a double 128×128 image buffer. Each PE has access to a part of the object memory and holds a minipicture of the object. The output image consisting of all minipictures from PEs is transferred to the frame buffer for display.

- PARCUM is a system designed for storing and displaying solid objects. It was built around a three-dimensionally organised memory called the memory cube. An object is represented as 3D arrays of binary digits 1 or 0 and is written into corresponding cells of the memory cube. Object visualisation can be implemented by direct accessing of cells in the memory cube. The read access is carried out by reading 4^3 voxels in parallel. The process of visualisation includes projection and illumination for realistic display. The performance was not presented by the authors.

- Kaufman and Bakalash [Kauf88] designed a CUBE architecture for 3D volume visualisation. It is centered around a 3D cubic frame buffer of voxels, and it contains three processors, each can access the frame buffer. The functions of these processors are to input sampled or synthetic data, to manipulate and to display 3D images. These processors are:

  1. A 3D geometry processor for scan-converting geometry into voxels;
  2. A 3D frame buffer processor for 3D BitBlt (bit block transfer) and transformation;
  3. A 3D viewing processor which generates orthographic projections of images.

  The CUBE architecture requires huge memory and immense parallel processing to achieve real time display. Two special features were incorporated within the architecture: a unique skewed memory organisation for parallel retrieval or storage of voxels; a multiple-write bus. They allow the system to display an image of n^3 voxels in $O(n^2 \log n)$ time.

- Insight is a system designed for medical image analysis and planning. It is based on a solid processing engine embodied in a specialised hardware processor. The system combines the software and the special hardware of the

Solid Engine which is used for image generation, manipulation and analysis of 3D models. A central computer is used for communication, control, processor initialisation and application programs. A high speed 3D model memory is used to hold 3D models of octrees. The data in the 3D model memory are loaded from the central computer and are processed by the Solid Engine.

2.1.4 Octrees

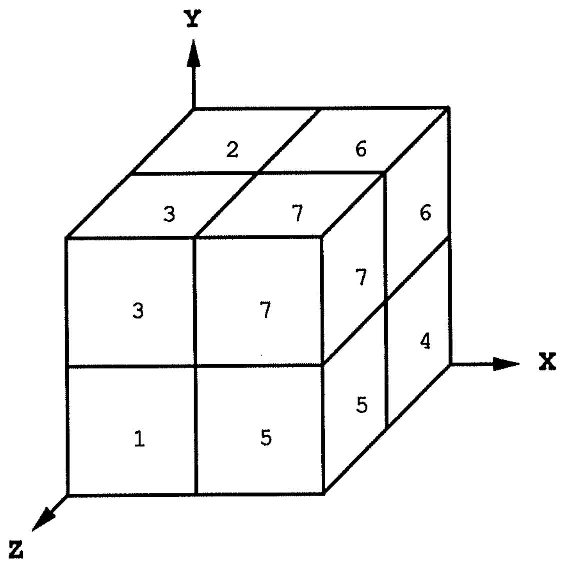
The octree approach of representing 3D objects is based on the principle of recursive decomposition (also called divide-and-conquer). It is a 3D generalisation of a binary tree. Its counterpart in 2D is a quadtree. An octree is created by recursively subdividing a cubic space into eight subspaces (referred to as octants) until the octant is homogeneous or until the specified resolution is reached. Octrees were first investigated independently by several groups of researchers [Jack80, Meag82, Srih81]. Each group paid attention to a number of specific problems. Jackins [Jack80] discussed the use of octrees for space planning, object rotation and translation. Meagher [Meag82] used them for geometric modelling and milling. Whereas Srihari [Srih81] referred to octrees as a means for 3D digital image processing.

Many algorithms have been developed during the last ten years [Same88a, Same88b, Chen88]. These algorithms can be classified into two categories: construction of octrees and manipulations of octrees. These algorithms will be reviewed in Sections 2.4–2.7.

2.2 Definition of Octrees

An octree is a tree with nodes of degree eight. Its root corresponds to the entire object space which is a cubical region with a length of 2^n (n is the depth of the tree). The root space is recursively subdivided into eight subspaces (octants or nodes) until the current space is homogeneous or the smallest cube is reached. The children of a node are of equal size, each is assigned a number chosen from 0, 1, 2, 3, 4, 5, 6, 7 as in Figure 2.1. These numbers are labels of nodes. A node can be either a nonterminal or a terminal node. Each nonterminal node (called grey node) has eight children again and each terminal node (called a leaf node) represents a data element. Each octant has a position and a size. The position of an octant is given by the coordinates of its zero-corner. The choice of zero-corner can be determined by the system programmer. In Figure 2.2 it is the left-bottom-back corner of the space. The size of an octant must be a power of 2. An octant of a given size can have only a restricted number of possible positions. For black and white images, the value of a leaf node is either 1 or 0 (marked full/empty or black/white), depending on whether it is inside or outside the object. Multiple colours are also used on some occasions to differentiate different objects. Figure 2.2 shows an example object and its octree.

The depth of an octree is the level of its deepest leaf which corresponds to a



	X	Y	Z
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Figure 2.1: Octant numbers of an octree space subdivision.

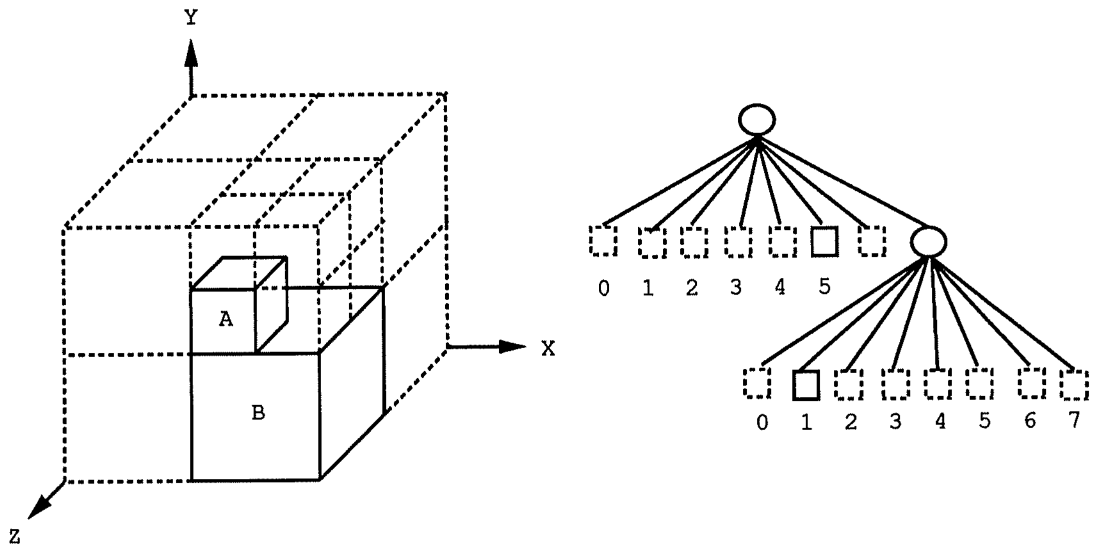


Figure 2.2: A sample object and the corresponding octree

voxel. The level of an octant (v) in an octree is defined as

$$level(v) = \begin{cases} 0 & \text{if v is a root octant} \\ level(father(v)) + 1 & \text{otherwise} \end{cases}$$

There are two data formats generally used in computer graphics: raster data and vector data. These distinct concepts of raster and vector data have been brought into octree definition too. Similarly, octrees have two distinct data types. Following the terms in [Same88a], we call them raster octrees and vector octrees respectively in this thesis. A raster octree models an object as a collection of cubes in 3D space. A vector octree models an object using the geometric abstractions of vertices, edges, faces, and polyhedral solids.

2.2.1 Raster Octrees

A leaf node in a raster octree is either full or empty (black/white). A nonterminal node is labeled grey. The example in Figure 2.2 is a raster octree. Raster octrees have some limitations when used for solid modelling systems. The first one is that a raster octree gives a rough approximation of an object. The accurate surface information which is needed for surface display is lost. The second limitation concerns the difficulty in conversion of a raster octree to other representations. The memory expensive is also a problem that restricts the use of octrees in complex solid modelling systems. Vector octrees are thus proposed to cope with these shortcomings of raster octrees.

2.2.2 Vector Octrees

Carlbom *et al.* [Carl85] and Navazo *et al.* [Nava86] introduced independently the definition of vector octrees which they referred to as extended octrees and polytrees respectively. The aims of using vector octrees are to reduce memory requirement and keep exact representations of objects. The cost of using this data format is the complicated node types and complicated algorithms. A leaf node of a vector octree is one of the following: a face cell, an edge cell, a vertex cell, a full or empty cell. Figure 2.3 illustrates these face, edge and vertex cells.

- A face cell contains a part of a polygon face of the object.
- An edge cell has a part of an edge of the object and parts of faces connected to the edge.
- A vertex cell contains one vertex from the object and parts of edges and polygon faces which are all converging on the vertex.
- A full cell is a homogeneous cell which lies entirely inside the object.
- An empty cell is a homogeneous cell completely outside the object.

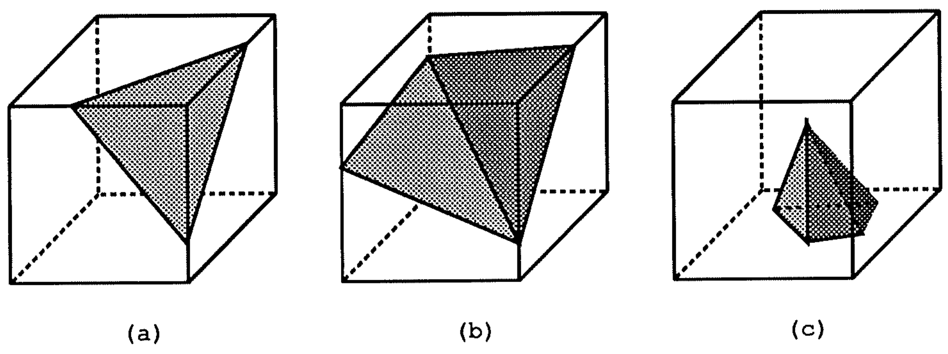


Figure 2.3: Leaf types of a vector octree: (a) a face cell, (b) an edge cell, and (c) a vertex cell

By introducing these complex cells, the number of nodes in an octree is reduced. However the space complexity of a vector octree is not easy to analyse. It is related to the position of the object in the space. There is a problem in the above definition of vector octrees when an object is not in a good position. An example is shown in Figure 2.4 (a) where a vertex lies very close to the boundary of a cell. Durst and Kunni [Durs89] called this situation a black hole. To avoid the possible black hole, they further generalised the polytree and introduced three new leaf types (Figure 2.5) in addition to the above five types. These new cells are edge', vertex', vertex" cells.

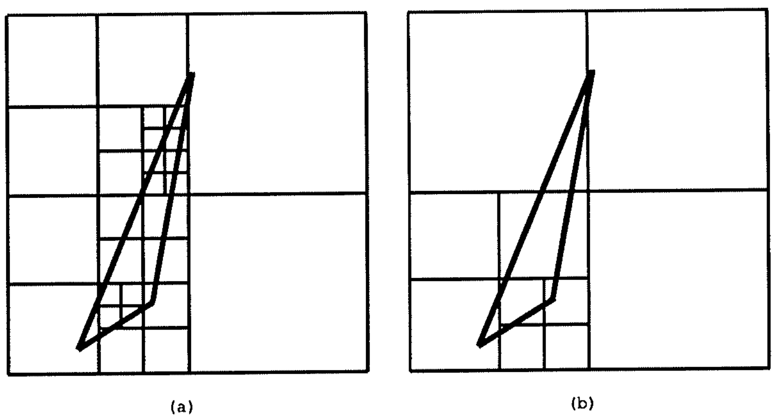


Figure 2.4: A 2D illustration of a problem with vector octrees: (a) a black hole exists when the object is in a bad position; (b) the problem can be solved by introducing more node types.

- An edge' cell contains polygon faces which connect to the same edge. The edge lies outside the cell.
- A vertex' cell has several edges and faces in it. All these edges and faces connect to the same vertex.

- A vertex" cell has neither vertex nor edge in it, but it has faces converging to a vertex outside the cell.

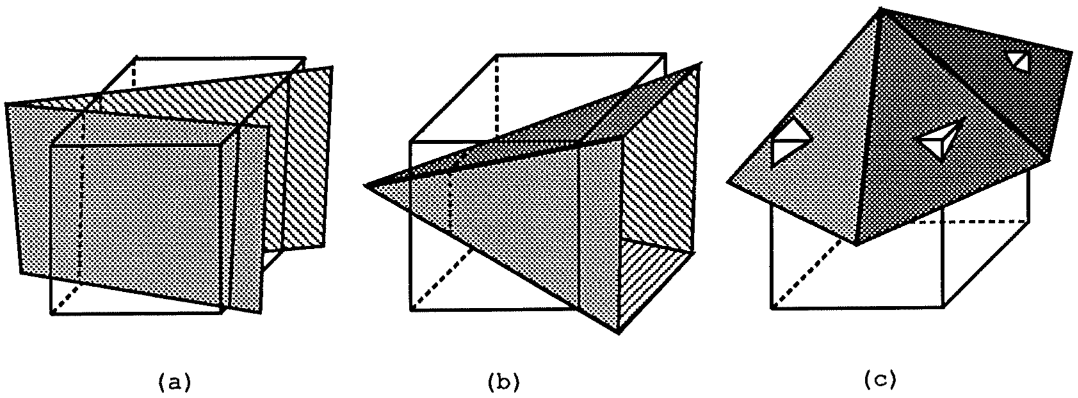


Figure 2.5: Additional leaf types for an extended vector octree: (a) an edge' cell, (b) a vertex' cell, and (c) a vertex" cell

Raster octrees and vector octrees differ only in their leaves. They can be organised in computer memory or disc storage using the same basic data structure. In the next section, several commonly used structures are introduced. The trees are raster octrees unless otherwise stated.

2.3 Data Structures of Octrees

For many algorithms in computer applications, choosing a proper data structure is crucial. This is also true for octree related algorithms. For the same data and operations, some data structures requires more (or less) memory and lead to more (or less) efficient algorithms than others. In considering the storage efficiency of an octree and execution efficiency of related algorithms, one must first know how the octree is stored.

Three principal ways of representing a quadtree/octree on conventional machines are: a tree structure with pointers [Hunt79a], a tree stored in preorder (known as a treecode [Oliv83a]) and a tree stored using locational keys (known as a leafcode [Oliv83b] or a linear octree [Garg82]). The last two formats are pointerless. In this thesis, these three formats will be referred to as pointer octrees, treecodes and leafcodes respectively. The efficiency of an algorithm depends on the encoding scheme adopted. Two other structures (a semi-pointer tree and an extendible cell structure) are also used in some literature but are less popular compared to the above encodings.

2.3.1 Pointer Octrees

Pointer octrees were the first of the three tree encodings used. Many early implementations of octrees are based on this approach. With this scheme each internal node (grey) of an octree has eight pointers to connect it with its eight children. These pointers are numbered as 0 to 7. The leaves do not have pointers but instead have a field to label the node as either full or empty. In some cases the label is black/white. To distinguish grey nodes from leaves, one extra bit is needed for each node. To save memory the position and the size information are normally not stored in nodes. They can be recomputed during implementation. To implement the tree traversal or other algorithms it is sometimes necessary to have one more pointer to link a node with its parent. This pointer is called a father link which is used to make the process of walking up a tree possible. The operation of walking up a tree is important for the neighbour finding algorithm which will be introduced later.

Octree manipulations with pointer octrees are simple and efficient. The subtrees can be visited in either a depth-first order or a breadth-first order. Node accesses are quick. However pointer trees have a large memory overhead for storing pointers. The size of a pointer field is normally 32-bit if implemented in C, although in theory it could be the base 2 logarithm of the number of nodes in the tree. Unfortunately, we shall have difficulty in allocating memory by this theoretical estimation since it depends on the tree size which is usually unknown before implementation. The overhead of storage for pointers make pointer octrees unsuitable for handling large trees which are often needed for 3D images and objects.

2.3.2 Treecodes

A treecode is a pointerless tree in which nodes are stored in an array structure with a specific order, typically a depth first order. These nodes are visited sequentially in the array. The sequence of records in the array represents implicitly the octree. With a treecode, every subdivided node is followed by its subdivisions, that is, the eight subnodes. Each node has a value. A nonterminal node has a negative value. A terminal leaf has a positive value representing its colour or other associated information. The treecode of the sample object in Figure 2.2 is

-1 0 0 0 0 0 1 0 -1 0 1 0 0 0 0 0 0

Nonterminal nodes (i.e -1 in the above data) can also be replaced by the average values of their subtrees. The position and the size of each node are not stored explicitly. Instead, they are calculated during recursive tree traversal. The pseudo C program of treecode traversal is shown in Algorithm 2.1.

A treecode requires less storage than a corresponding pointer tree. However the disadvantage of treecodes is that they are order sensitive. The spatial location and the size of a node are determined by its position in the array and in relation with other nodes. A tree must always be traversed from the beginning of the array even if users are interested only in a small number of nodes. This leads to an inefficient traversal algorithm especially when a tree is visited in a sequence other than the order in which it is stored.

```
#define HALVE(n) ((n) >>= 1)

/* defining an octant by the coordinates (x,y,z)
of its zero point and the size */
typedef struct {int x, y, z, size;} OCT;
int GREY = -1;
OCT Root = {0, 0, 0, 512};

/* vectors for computing each of the eight suboctants */
int x_table[8] = {0,1,0,1,0,1,0,1},
    y_table[8] = {0,0,1,1,0,0,1,1},
    z_table[8] = {0,0,0,0,1,1,1,1};

OCT SubOctant(int i, OCT octant)
BEGIN
    HALVE (octant.size);
    octant.x += x_table[i]*octant.size;
    octant.y += y_table[i]*octant.size;
    octant.z += z_table[i]*octant.size;
    return (octant);
END
PROCEDURE Traverse(OCT octant)
BEGIN
    ReadColour();
    IF (colour == GREY) THEN
      FOR i = 0 TO 7 DO
        Traverse(SubOctant(i, octant));
    ELSE Display(octant, colour);
END
main()
BEGIN
    Traverse(Root);
END
```

Algorithm 2.1: The pseudo C procedure of the treecode traversal algorithm

2.3.3 Leafcodes

In contrast to the above two encodings, a leafcode stores space addresses of nodes of an octree. Each leaf carries the information of its position and size as well as colour. The structure of a tree in leafcode is a simple array. Each element (corresponding to a leaf) of the array consists of three fields:

1. an octal number to represent the locational code of an octant's x, y and z coordinates;

2. a number to represent the level (or the size) of an octant;

3. a number indicating the colour of an octant.

Each octal digit in the locational code corresponds to one of eight octants at a certain level of octree space subdivisions, with the left-most digit for the first level of subdivisions. Therefore, these octal numbers express the path from the root to the given node. For black and white images, the colour can be ignored if storing only black nodes. Leafcodes for the black/white image of Figure 2.2 is shown below. The tree has only two black nodes for this object.

5X, 71

Leafcodes have the smallest number of nodes among the above three tree encodings. Less nodes are stored since only the nodes which contain object information are important, although each leaf may take more bits to specify its location. Other advantages of leafcode octrees include the following. Information about the spatial location and content of each octant are stored in the node. Therefore nodes in the leafcode format are not order sensitive. All the nodes can be processed in parallel for some operations such as translation, rotation by 90 degrees. However, searching a node is always slow on a conventional machine for a large data set. To improve searching, a tree must be kept in a sorted order and needs to be resorted after updating. Another disadvantage of leafcodes is that the maximum resolution must be set in advance and is fixed during implementation.

2.3.4 Other schemes

- Semi-pointer structures: These structures are mainly introduced in quadtrees [Will88b] and are equally applicable to octrees. They lie between pointer structures and treecodes. There are many variations of semi-pointer trees. Three of them are listed here: the sextree proposed by Oliver [Oliv86] (dectree in the case of 3D octree); Stewart's one-to-four tree [Stew86]; Williams's Goblin quadtree [Will88a]. These formats are basically reorganised treecodes with an additional pointer for each block of four nodes (eight nodes for octrees). The pointers allow subtrees to be skipped at any given level. A semi-pointer tree requires a little more memory compared to the most space efficient scheme of treecode, but offers greater flexibility for node access and tree traversal. The breadth-first traversal of trees is made simple.

- An extendible cell (EXCELL) structure: EXCELL is a scheme aimed at indexing and efficient accessing of geometric data. It allows the size and the shape of cells, which are rectangular parallelepipeds (RPs), to vary according to the spatial organisation of data. The structure has two parts: a data part and a directory. The data part consists of cells similar to the leaves of octrees. The difference is that here cells are not necessarily cubes. Cells are formed by recursively halving the space of interest in some order. The access of a cell is through the directory which is an array of pointers to the data cells. Each element of the array corresponds to a RP of minimal size. Given a point in

3D space, the minimal RP is calculated and located in the directory. Then following the pointer in the directory, the data cell can be found. The memory requirement is extensive for the directory.

2.4 Manipulations of Octrees

The main operations of octrees are those for transformations, display, cross section generations, set operations, volume computations, connectivity labelling and neighbour finding. The conventional algorithms for these operations are reviewed in the following sections. They will be discussed according to data structures used.

We shall concentrate mainly on raster octrees. For vector octrees, tree structure traversals are basically the same as that of raster octrees, except that contents and manipulations of leaves for vector octrees are more complicated. Applications of vector octrees are confined to accelerating image generation, set operations and display.

2.4.1 Geometric Transformations

Geometric transformations of octrees include translations, rotations and reflections. They are discussed below, each in one subsection.

2.4.1.1 Translation of Octrees

Translating a volume is to move the volume through displacements along each of three axes x, y, z. Translation of an octree is defined as follows. Given an octree encoding of an object (the source tree) and a translation vector of the object (T), the translation process will produce a new octree (the target tree) representing the translation of the original octree. This process involves reorganisation of the tree to build the target tree from the source tree.

Jackins and Tanimoto [Jack80], Meagher [Meag82] developed algorithms for translating pointer octrees. Their algorithm is based on intersection tests which overlap the space of the source tree and the space of the target tree by moving the latter with respect to the former, then mapping the colours of the source octants to the target octants. Oliver and Wiseman [Oliv83a] discussed translation of treecodes. Their algorithm is based on observations that growth of tree detail is along the edges of incursions. Those edges are places where new nodes will appear. Translation is implemented for three axes in turn, each time the tree is translated by a positive displacement (that is a movement along the positive direction of the axis). If the object is required to be translated by a negative displacement, an operation for reversing the space is implemented before translating. The image is reversed again after translating.

Translation of leafcodes is straightforward. It can be implemented by address manipulation. Each node can be translated independently. A translated leafnode may remain a leaf if it is translated by some multiple of its size, or otherwise may need to be broken into several leaves. The node splitting is then a key process. The resulting tree will need to be sorted and condensed.

2.4.1.2 Scaling of Octrees

Scaling, by a power of two, octrees encoded as pointer structures or as treecodes is simply to add or to delete levels at the root. Adding one level at the top reduces the object by half. The new root has eight children, one of them is the old root, others are empty leaves. The position of the old root in the new tree determines the position of the object in the new space. Similarly, doubling the size of an object in octree space is to select a node from eight nodes at the top level of the tree as a new root. All other brothers of the selected node are deleted. If scaling the octree of an object by a factor other than a power of two, we must rebuild the tree. Like translation of octrees, this process involves intersection tests.

Scaling leafcodes by powers of two is less easy than pointer trees and treecodes. Firstly, all leaves must be scaled independently by recalculating a new address for each leaf. Secondly, the address of each new leaf will depend not only on its corresponding old leaf but also on the reference point for scaling. Leafcodes are better than pointer trees for scaling by an arbitrary factor because tree reconstruction is fast with address manipulation . A scaled leaf may be split into several new leaves. The final tree will need to be condensed.

2.4.1.3 Rotations and Reflections of Octrees

Rotation of an octree by 90 degrees (or a multiple of it) about an axis is to rotate the octree about a line passing through the center of the octree space with a specific direction of rotation. Reflections include reflecting the octree about a plane passing through the center of the octree space. The plane either is parallel or has an angle of 45 degree to a face of the space. Both rotations and reflections are implemented by reordering the sequence of nodes.

For a pointer octree, the rotation process is simply a recursive permutation of the children of each grey node [Jack80]. The order of permutation is determined by the space organisation and the axis about which the octree is rotated. For the space organised as in Figure 2.1 and assuming we want to rotate an octree about the x-axis and move the z-axis into the y-axis, the rotation can be performed by applying the permutation (0,1), (1,3), (2,0), (3,2), (4,5), (5,7), (6,4), (7,6). The time complexity of rotating a pointer octree is O(N), i.e. linear in the number of nodes N in the tree.

Rotations and reflections of treecodes are less easy than for pointer octrees. It is because the permutation is performed at each level of the tree in a breadth-first order, whereas the treecode is organised in a depth-first order. In their paper, Oliver and Wiseman [Oliv83a] process the treecode one level at a time and make multiple scans of the input. Each scan skips over the subtree which separates one node from the next node at the same level. The algorithms are space efficient but not time efficient.

Rotations and reflections of leafcodes are very simple by applying the spatial permutations to all of the octal digits of each node.

$$0 \to 1\,;\,1 \to 3\,;\,3 \to 2\,;\,2 \to 0\,;\,4 \to 5\,;\,5 \to 7\,;\,7 \to 6\,;\,6 \to 4\,;$$

The mapping time is also linear in the number of nodes N in the tree. If only black nodes are stored then N represents the number of black nodes. Resorting is needed after rotating the original tree. The sorting takes $O(N \log N)$ time.

2.4.2 Set Operations

The basic set operations are union (OR), intersection (AND) and complement (NOT). The hierarchical data structures like quadtrees and octrees are especially useful for performing such operations. The algorithms mainly require a traversal of two input trees in parallel, comparison of corresponding nodes, then construction of a resulting octree. The rules for node comparison depend on the types and contents of nodes of the input trees. They are different for raster octrees and vector octrees. For raster octrees of black and white images, the set operations merge two input trees according to rules in Table 2.1 and Table 2.2.

Union	WHITE	BLACK	GREY
WHITE	WHITE	BLACK	GREY
BLACK	BLACK	BLACK	BLACK
GREY	GREY	BLACK	GREY

Table 2.1: Union of two raster octrees

Intersect	WHITE	BLACK	GREY
WHITE	WHITE	WHITE	WHITE
BLACK	WHITE	BLACK	GREY
GREY	WHITE	GREY	GREY

Table 2.2: Intersection of two raster octrees

The Table 2.1 can be explained as follows. If two nodes from the tree A and the tree B are both BLACK then the resulting node in the output tree is BLACK. If one node, say in A, is WHITE, then the corresponding node in the output is set to the other node in B. If nodes in both trees are GREY, the output node is set to GREY and the algorithm is applied recursively to children of A and B. The resulting tree must be condensed because the union of two trees can yield the situation where all children of a GREY node in the output tree are BLACK. The intersection of two octrees (see Table 2.2) is just as simple. A program for intersecting two octrees is listed in Algorithm 2.2. Again the resulting tree needs to be condensed, and this time all WHITE sons are merged. For vector octrees, the node types are more complicated and lead to complex decision making as demonstrated by Navazo [Nava89] (shown in Table 2.3).

```
PROCEDURE Intersect(OCT A, B)
BEGIN
  IF A = White OR B = White THEN output White
  ELSE IF A = Black THEN copy out the subtree of B
  ELSE IF B = Black THEN copy out the subtree of A
  ELSE
    output Grey
    FOR i = 0 TO 7 DO
      Intersect(SubOctant(i, A), SubOctant(i, B))
  ENDIF
END
```

Algorithm 2.2: Pseudo C procedure for the octree intersection algorithm

Intersect	W	B	F	E	V	G
W	W	W	W	W	W	W
B	W	B	F	E	V	G*
F	W	F	W,F,E,G	W,F,E,V,G	W,E,F,V,G	G'
E	W	E	W,F,E,V,G	W,E,V,G	W,E,V,G	G'
V	W	V	W,F,E,V,G	W,E,V,G	W,V,G	G'
G	W	G*	G'	G'	G'	G

Table 2.3: Intersection of two vector octrees: G* means that the grey node and its descendent are copied to the output tree; G' indicates that further intersection of the leafnode with the descendent of the grey node is required.

The time complexity of octree set operations is related to the data structure used for tree encoding. Union and intersection of pointer octrees are performed in a recursive manner, with an execution time proportional to the number of nodes in the output octree. The union/intersection time of treecodes is linear in the sum of nodes from two input octrees. This is because each node of a treecode is determined by its position in the data array with respect to the nodes before it. To locate an octant one must visit each node in turn. There is no simple way to skip over a group of nodes or subtrees. For leafcodes, the time for the intersection/union operation is linear in the sum of black nodes of two input trees.

The complement of an octree is formed by reversing the colour of its leaves. Black nodes are changed to white nodes and vice versa. The grey nodes remain unchanged. The execution time is proportional to the number of nodes in the original tree for all three tree formats.

2.4.3 Volumes and Other Integrals

Computation of volumes and other integral properties of solids with complex shape are necessary in scientific and many other fields. Lee and Requicha [Lee82] summarised several methods for volume calculations using octrees and other cell decomposition schemes. They calculated volumes of solids represented by CSG. The CSG model is first converted into an octree, then the octree is traversed in preorder and the sizes of black nodes are accumulated to obtain the volume. The execution time is proportional to the number of nodes in the octree.

2.4.4 Displaying Octrees

There are two main approaches to display 3D objects stored as octrees. They are front-to-back traversals and back-to-front traversals. Since all octants are already in a sorted order, the hidden surface removal is very easy. It is simply to follow a specific sequence to visit each node of an octree. With front-to-back display, octree nodes which are closer to the viewer are visited before those farther away nodes. Front-to-back methods are suitable for special projections of octrees such as face, edge, or corner views. The advantage of this approach is that each leaf in the tree is visited at most once. If a leaf is completely obscured then it is not visited at all. In back-to-front display, nodes farther away from the viewer are visited before close ones. The projection of the current octant overwrites the painted region of any octant visited earlier.

Doctor and Torborg [Doct81] developed a front-to-back display algorithm for face views. In their algorithm, a display generator transforms an octree-encoded object into a quadtree-encoded image. The relation of an octant and a quadrant is simple since the edges of the octant and the projected quadrant coincide. Both opaque and semi-transparent objects are discussed. Yamaguchi *et al.* [Yama84] proposed triangular quadtrees for front-to-back display of octrees with corner views. The algorithm restricts the image to be an isometric view. With this special view direction, projections of edges of cubes in an octree divide the space into triangles, thus a triangular quadtree is formed. Again, edges of triangles coincide with edges of octants of the original octree. Front-to-back display with an arbitrary view is more difficult. No simple rule exists to determine the relation between the edges of the current octant and previously projected octants. One needs to test the intersection between the projection of an octant and the quadrants in the image plane. Therefore, for arbitrary views it is simpler to use back-to-front display.

Frieder *et al.* [Frie85] proposed a display algorithm for recursive back-to-front read out of voxel-based objects. A similar approach was adopted later by Gargantini *et al.* [Garg86] for leafcode display. In their method, the farthest suboctant was examined first, followed by its three face neighbours (in any order), followed again by the three face neighbours of the closest suboctant, and finally by the closest suboctant itself. To get this order, a program called `Pipeline` was used to reorder the input leafcode. The program requires repeated searches of the input octree. Searches slow down the algorithm's implementation. Meagher [Meag82] adopted back-to-front display for pointer octrees. He used a predetermined recursive display sequence 0 1 2 3 4 5 6 7 for the octree space shown in Figure 2.1. Similarly, octrees can be displayed with other sequences of 0 2 4 6 1 3 5 7 or 0 1 4 5 2 3 6 7. These three sequences give the same final image of the object. With this method, the programmer must decide in advance the sequence to be used. Changing the viewing position may result in different display sequences, or require rotating the octree.

Displaying treecodes in an order other than the stored depth first order is generally inefficient. It is due to the lack of ability to do random node access in treecodes. The process may need more than one scan of the octree. Oliver [Oliv84] proposed two methods for fast display of treecodes. The first algorithm generates an image of a plane section of an octree and will be summarised in the next section. The second method displays an orthographic projection of an octree in a front-to-back manner like Doctor and Torborg's algorithm mentioned above. Oliver chose a special numbering order (Figure 2.6 (b)) to store the octree. Thus it becomes possible to make a single scan of the tree. However, if the viewing position changes, a different order may be required thus extra traversals are necessary to reorder the tree. Scanning a treecode is often inefficient because many nodes must be visited before the wanted one is found. To make quick skips over those nodes, Oliver further introduced a sedectree (sedecim is the Latin word for sixteen) in which each parent node has sixteen child nodes. The first eight nodes are pointers to the offsets and the other eight nodes are normal octants.

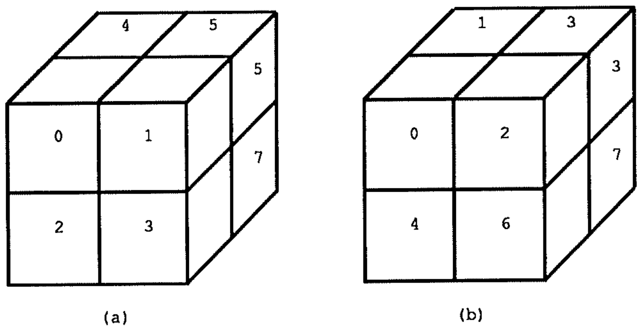


Figure 2.6: Oliver's ordinal numbers of octants for two treecode algorithms: (a) display a plane section which is parallel to the front face of the image space; (b) display an orthographic projection of an octree in a front-to-back manner.

Noting that only those nodes at the border of the object are contributing to the final image, several groups of researchers developed algorithms which separate border octants from inner octants and display border octants only. Chien and Aggarwal [Chie86b] presented a multi-level boundary search algorithm which detects and labels the interface between the black/white nodes (surfaces) of a pointer octree. The algorithm avoids time-consuming neighbour finding operations by using top-down neighbour computation in conjunction with tree traversal. Gargantini *et al.*

[Garg86] proposed another two algorithms which are modifications of the back-to-front leafcode display algorithm. These two algorithms detect first the 3D border of the given object, then project the surface voxel of the object onto the screen. The difference between the two methods is that one processes black and white images and the other deals with general images with grey-level or different colours. However, the disadvantages are that the two methods result in a large number of border voxels and the time is increased for displaying these small voxels.

2.4.5 Cross sections of octrees

A cross-section of an octree is an image of a voxel-thick slice from the octree space. It is generated by testing the intersection of the octree with a plane. The plane can be either perpendicular to any axis of the octree space, or with an arbitrary orientation. The first case corresponds to an orthogonal cross section and the second one is an arbitrary cross section. An orthogonal cross section can be generated by traversing the octree, finding the nodes that intersect the plane, and building the corresponding quadtree from those intersected nodes. The plane-octant intersection test is simple since the plane is parallel to two of six surfaces of each octant. For an arbitrary cross section, there is no simple relation between the plane and the octants. The quadtree corresponding to an arbitrary cross section must be built either top-down or bottom up.

Yau [Yau84] discussed algorithms for generating quadtrees of cross sections from pointer octrees. Her algorithms included orthogonal cross section generation, and arbitrary cross section construction. Two algorithms have been proposed for generating arbitrary cross sections. One is a top-down method and its key operation is to detect octant-plane intersections. The other is a bottom-up method which scan-converts the plane and determines the colour of each pixel by using the point enclosure test. The pixels are then grouped into quadrants to build the quadtree.

Oliver's algorithm [Oliv84] generates cross sections from treecodes. Here a treecode is numbered in such a way that either the first four octants or the last four octants will be chosen during the depth first traversal. Two more offset nodes are added to make the algorithm jump over the irrelevant nodes quickly. This modified tree is called a dectree, in which each parent has ten children.

2.4.6 Neighbour Finding

The neighbours of an octant are those nodes which are spatially adjacent to the octant. Each octant in an octree space has six faces, twelve edges and eight vertices. So each node has three kinds of neighbours: face neighbours, edge neighbours, and vertex neighbours as in Figure 2.7. Given a node and a direction, the node can have either one face neighbour which is equal to or larger than itself, or several smaller face neighbours (Figure 2.8).

There are two main reasons for locating the neighbour of an octant. The first one is that the value of a node, or an operation applied to a node, may depend on the value of its neighbour. One such example is the connected component labelling

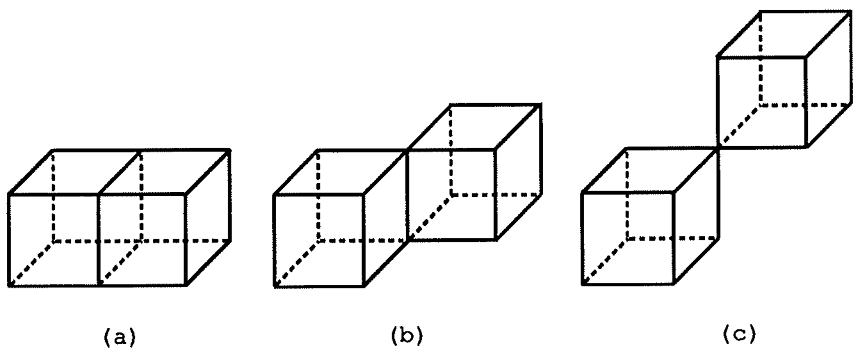


Figure 2.7: Neighbours of an octant: (a) face, (b) edge, (c) vertex neighbours

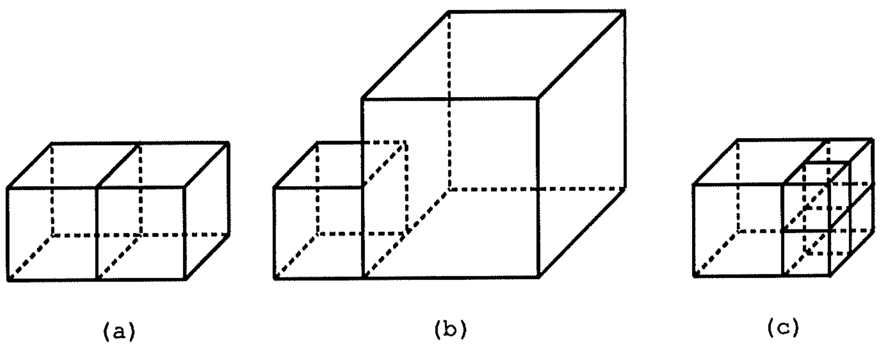


Figure 2.8: Neighbours of an octant: (a) equal, (b) larger, (c) smaller in size

algorithm. The second reason is that some applications need to visit octants in the space in spatial order. This is most useful for accelerating ray tracing. Connected component labelling algorithms visit all neighbours of an octant, while moving between adjacent octants in space involves only two neighbouring nodes each time. Motion is possible in the direction of a face, an edge, or a vertex. Motion is also possible between nodes of different sizes.

There are several techniques for neighbour finding. The most straightforward method is to use point location. From a node, we compute the coordinates of a point which just exits the current node space. This point must be located in one of the neighbours of the node. Then the point location operation is performed. It starts at the root of the tree, compares the center of the current space with the point, and determines which of the eight subtrees contains the point. The process is repeated recursively until a leaf is reached. The worst case time complexity for locating an octant is linear in the depth of the tree. Point location can also be implemented in a different way by converting the coordinates of the point to the octal code of a unit cube and then checking whether it corresponds or belongs to any octant. This is most often used for leafcodes. Its execution time is proportional to the log of the number of nodes in the tree since it involves a binary search to find a specific node.

Another method [Same89a] is a bottom-up neighbour finding. It adds father links to the tree and computes the path to the neighbour by following the links to find the nearest common ancestor of the node and its neighbour. This method is simple for the cases which search face neighbours that are of a size equal to or greater than the node itself. The third method is to build explicit links from a node to its neighbours [Hunt79b]. These links are named ropes and are defined as a link between two face adjacent nodes. The two nodes are of equal size and at least one node is a leaf. The memory requirement is increased for storing ropes. Both the second and third methods are more complex in locating neighbours with a size smaller than that of the original node. They are also storage extensive.

2.4.7 Condensation of an Octree

A minimal octree is a tree which has a minimal number of nodes for representing an object. In contrast, the same object may be stored as a non-minimal octree with more nodes. Those extra nodes often form groups of sibling nodes of the same colour as shown in Figure 2.9. They can be replaced by one larger node.

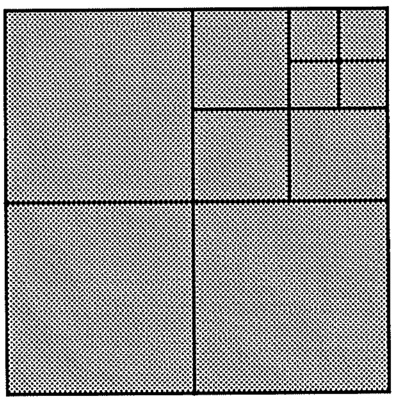


Figure 2.9: A 2D representation of nodes of a non-minimal octree

An octree needs to be condensed after operations which result in a situation where eight sibling nodes in some part of the space are of the same colour. To condense an octree is to delete children of a grey node when all those leaves below have an identical label. The grey node is replaced by a leaf with the label. During condensation, each node of the input tree must be visited once in order to identify its colour. Therefore the condensation process has a time complexity which is proportional to the number of nodes in the input tree.

2.5 Constructing Octrees from Image Models

Octrees can be constructed from two groups of representations: image models and object models. Image models include 3D arrays of numbers (3D digital images), and its two variations: run-lengths and serial sections, as well as silhouette images which are 2D projections of 3D scenes. The data in image models are mainly generated by data collection techniques. They are important in the field of image analysis and processing, pattern recognition, computer vision. Object models include sweep representations, CSG and boundary representations. They are used as object descriptions, generated by users and then stored in computers. In this section, we discuss image model to octree conversion. The algorithms for converting object models to octrees will appear in the next section.

2.5.1 3D Arrays

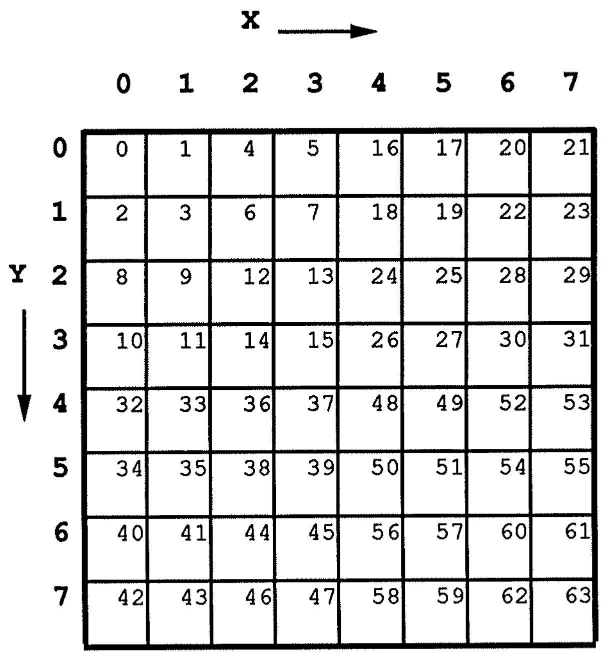
A simple method to build an octree from a 3D binary array is to extend the algorithm developed by Samet [Same80] for quadtree construction from a 2D array. This is a recursive method which successively subdivides the array and scans elements of the array in a depth first order (also named Morton order). Figure 2.10 is an illustration of scanning a 2D array in Morton order with the space origin in the top-left corner. A node of a tree is visited after its children are visited. A leaf node is created only when it contains a maximal block of elements, all these elements being of the same colour. Minimal octrees are generated with this method. The time complexity is proportional to the number of voxels in the array. The memory requirement is that for storing the 3D array and the output octree.

The above simple algorithm is storage extensive. To save storage, Samet [Same81] proposed an improved algorithm for constructing quadtrees by processing the image one row at a time and merging identically coloured sons as soon as possible. This is a bottom-up method. It can be extended to 3D and implemented by voxel insertion. All the voxels in odd-numbered slices are inserted into the current tree. Merging operations are only possible for even-numbered slices. The algorithm cannot give much memory saving for 3D cases since the minimum intermediate storage is that for storing the octree with the first slice at voxel size. The total number of nodes for an octree with such a slice is

$$\frac{2}{3}(4^{(D+1)} - 1) - 1$$

While D is the maximum depth of the tree. For the example shown in Figure 2.11 (D = 2), the number of nodes is 41. When image resolution increases (for instance D = 8, resolution 256 × 256 × 256), this number is 174,761.

The algorithm is more complicated. The key procedure is to add a neighbour voxel of a node to the tree. It also adds non-terminal nodes to the tree when necessary. Adding a neighbour of a node in a specified direction consists of the following processes: traversing ancestor links until a common ancestor of two nodes is found; and then descending along a path which is the reflection of the ascending



Y \ X	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

Figure 2.10: Scanning a 2D array in Morton order.

path about the common boundary between two nodes. If a common ancestor does not exist, then a non-terminal node is added with its other sons being white. Another important procedure is a `merge` process which checks at each even-numbered slice for possible merges with previous slices. The inner loop of the algorithm is slow although its asymptotical complexity is still proportional to the number of voxels in the input. Each node in the tree requires one more field to hold a pointer to its ancestor. So memory overhead for the final octree is larger than that for the previous simple method.

Both the above top-down and bottom-up algorithms build pointer octrees. It is clearly shown that the bottom-up method is much more complex due to the node insertion operation which is time consuming for pointer octrees. Leafcodes, on the other hand, make the bottom-up approach simpler and straightforward. Mark [Mark86], Shaffer and Samet [Shaf87] discussed efficient leafcode-based algorithms which convert raster data in the form of run-lengths to quadtrees and octrees.

2.5.2 Run-Lengths

A run-length is a widely used representation which stores each row of a raster image as a sequence of maximal runs of pixels (or voxels) of the same value.

Shaffer and Samet [Shaf87] proposed an algorithm for constructing quadtrees from run-lengths by using a table of active nodes. The construction process is as follows. For each pixel in a run, if the pixel has the same colour as an appropriate active node, do nothing. Otherwise, a node is inserted with the current pixel as

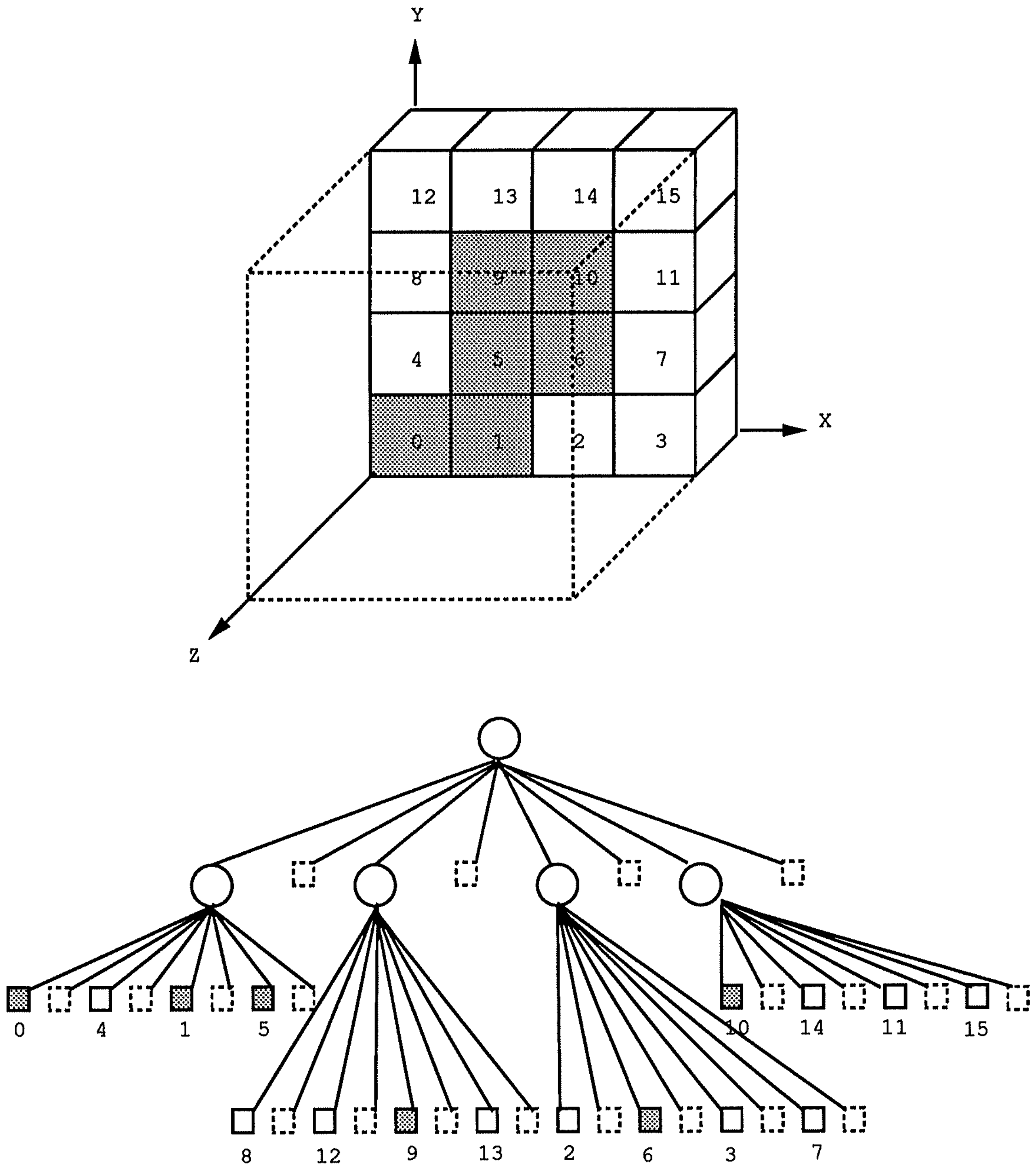


Figure 2.11: An octree for one slice of a digital image

the first pixel and with the largest possible size. This node is also added to the active node table. An active node is removed from the table when the current pixel is the last pixel of the node. The key operation is to locate the smallest active node which contains the specified pixel. This needs a search operation. To avoid search, an access array is used. Each element of the array provides a pointer to the corresponding active node in the table. This algorithm generates a minimal quadtree. The authors claimed that their algorithm is optimal by using the number of insertions as a metric, since it yields at most one insertion per node in the output. However, the insertion process is not constant and nodes are not inserted in the preorder. The resulting tree must be sorted. The time complexity for sorting is $O(n \log n)$ where n is the number of nodes in the output tree. To maintain the table of active nodes a certain amount of additional storage is needed. The above idea can be extended to octrees using a larger table.

2.5.3 Serial Sections

Yau and Srihari [Yau83] proposed an algorithm for constructing octrees from a series of cross-sectional images represented as quadtrees. The algorithm builds an octree from 2^n slices of quadtree images which are orthogonal to one axis. It merges each pair of adjacent slices in a bottom-up way. At the first stage, the 2^n slices are merged into $2^{(n-1)}$ generalised slices. This process is repeated n times and finally the last two generalised slices are merged to obtain the final octree.

2.5.4 Orthogonal Silhouettes

The above octree construction algorithms may become useless if we cannot get 3D binary arrays or run lengths of objects in the first place. In practice, the 2D images of an object can be generated much easier by using a camera. Such an approach is often used for robot vision. A series of 2D images from different view points can be obtained and used to construct the octree of an object. Chien and Aggarwal [Chie86a] proposed to take three orthogonal face views (front, top, side) of an object to build an octree. We call it the face-view algorithm here. The images for three orthogonal views can be easily converted into quadtrees, each quadtree corresponding to one face. However, three face views are not enough for representing objects like cones and objects with holes. Three face views of a cone will result in an octree with pyramid shape. More views are necessary in those cases. Veenstra and Ahuja [Veen85] added six edge views (one of them is shown in Figure 2.12) to refine the octree built previously from face views. More complex corner views as in Figure 2.13 can also be introduced.

These algorithms are based on pointer trees. Here we only give a brief summary of the face-view algorithm proposed by Chien and Aggarwal [Chie86a] as follows:

1. generating the quadtrees of the three silhouettes from the three orthogonal views of an object;

3	7,3,2	7,6,2	6
1	5,1,0	5,4,0	4

Figure 2.12: An edge view of an octree space

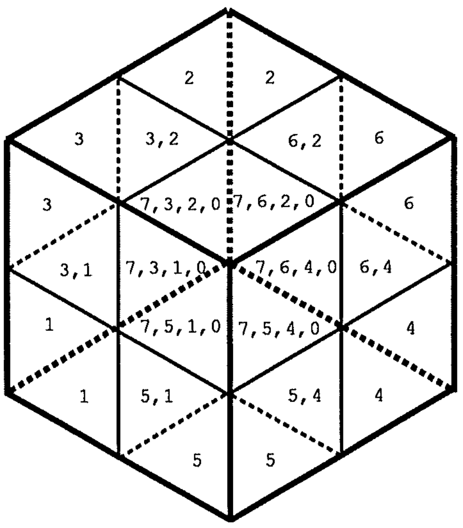


Figure 2.13: A corner view of an octree space

2. transforming each quadtree along the associated viewing direction to obtain a pseudo-octree representing a cylinder swept through the related viewing direction;

3. Intersecting the three pseudo-octrees to generate a final octree of the object.

For each view, a pseudo-octree is generated by assigning two numbers known as labels to every node of the quadtree. Figure 2.14 shows the correspondence between quadtree orders and pseudo-octree labels for each of the three face views. The process of intersecting three pseudo-octrees to generate the final octree consists of three procedures: `Intersect-3` and `Intersect-2` which perform three and two pseudo octree intersections respectively; `Convert` which traverses the resulting pseudo-octree to produce an output octree. The rule for intersection is as follows. Starting from the root node, the program traverses the three pseudo-octrees in parallel. If the nodes from the three pseudo-octrees are grey, `Intersect-3` is performed on the eight combinations of their children. If two are grey and one is black then the

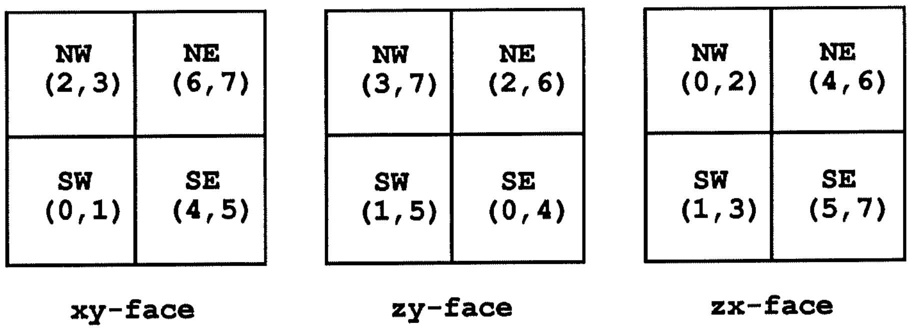


Figure 2.14: The quadrant numbers and the corresponding pseudo octree labels

outcome is determined by the two grey nodes through `Intersect-2`. If one node is grey and the other two are black, the output is the subtree of the grey node. If all three nodes are black, the output is a black node. If at least one node is white, then the output node is white. Since it is possible that after intersection all eight children of a grey node are white, the resulting tree may not be a minimal tree. Condensation must be performed on the output tree.

2.6 Constructing Octrees from Object Models

2.6.1 Rectangular Parallelepipeds

Franklin and Akman [Fran85] designed an algorithm for constructing octrees from a set of rectangular parallelepipeds (RPs) approximating an object. This is a bottom-up approach for building leafcodes. The RPs are evenly spaced in one plane (for example xy-plane) and have different lengths along the third axis (z). These RPs are either obtained by ray casting methods which cast parallel rays through the xy-plane, or converted from run-lengths.

2.6.2 General CSG Models

The simplest algorithm for constructing octrees from CSG models is to build an octree for each primitive solid then to merge all these primitive octrees using CSG set operators. Lee and Requicha [Lee82] proposed an efficient method to build octrees from CSG solids. Their algorithm has two major improvements to the above straightforward method. Firstly, it has a simple cell-primitive classification method for decomposing a primitive object into variable sized blocks. The algorithm uses a modified cell classifier to determine whether an arbitrary cell is wholly inside/outside any primitive of CSG or is intersecting with it. Instead of examining eight vertices of an octant, the cell classifier checks only the center of the octant. In order to do this, two boundaries are derived for each solid at each level of subdivision. The solid is compressed to form an inner-bound as well as expanded to get an outbound for testing in/out relations. These boundaries are computed in a preprocessing phase and stored in main memory. If the central point of an octant is outside the corresponding outbound, then the octant is white. If the point is inside the inner-bound, the octant is black. Otherwise, the octant intersects with the boundary of the solid and is marked grey. Further subdivision is necessary. The second improvement is an efficient recursive subdivision strategy. Instead of obtaining the final octree of a CSG model by combining the primitive octrees, the algorithm evaluates set membership classifications during recursive subdivisions. The combination tables (for example Table 2.4 for intersection of two primitives) are consulted for each cell during subdivisions.

		Classify (cell, B)		
		IN	OUT	INTERS.
Classify (cell, A)	IN	in	out	subdivide
	OUT	out	out	out
	INTERS.	subdivide	out	subdivide*

Table 2.4: Combination table for classifying a cell against a CSG tree (A intersected with B), * denotes that a non-minimal subtree may appear in this case.

2.6.3 Boundary Representations

Conventionally, two approaches have been employed for building octrees from boundary representations: top-down and bottom-up. The former includes the divide-and-conquer method with point enclosure tests [Tamm84a], the method of interior/exterior classification [Meag82], and the method of connected component labelling [Tamm84b]. The top-down methods are most useful for generating pointer octrees and treecodes although they are applicable to leafcodes too. Gargantini *et al.* [Garg86] use a bottom-up method to generate leafcodes by interior filling. Tang and Lu [Tang88] proposed another bottom-up method which generates leafcodes by transforming coordinates of voxels to octant addresses and then condensing them into octree nodes.

- **Cell classification:** This method consists of two steps: an intersection test and a point enclosure test. In the first step, each octant is examined to see whether it intersects any of the object's faces. If it does, then the octant is subdivided and the test is repeated for each suboctant. Otherwise the octant could be either interior or exterior to the object. This is determined in the second step in which the program counts the number of times that a ray originating from a point inside the octant intersects the faces of the object. An even number means that the octant is outside the object. Otherwise, it is inside. The above tests require intensive computation especially when the number of faces of the object increases. Tamminen *et al.* [Tamm84a] used a spatial indexing technique to speed up the process of point enclosure tests. By improving the efficiency of tests, the divide-and-conquer method becomes practical for objects with a large number of polygons.

- **Interior/exterior classification:** This method classifies an octant as black when it is interior to the object, white if it is exterior, otherwise subdivides the octant. An octant is classified as an interior one when it is interior to all the bounding faces of the object. If it is outside any face then the octant is an exterior one. This classification requires comparisons between all eight vertices of an octant and each face of the object. The input data (a list of faces) are repeatedly scanned for testing each octant. Essentially, this is a "brute-force" method and is not practical for objects with a large number of polygons. Moreover, it is only applicable to convex objects. For concave objects the above classification criteria do not hold. An octant can be interior to a concave object when the octant is outside some faces of the object. On the other hand, a cell can be exterior to an object with holes while it is interior to many faces of the object. Figure 2.15 shows one example of an object with a hole.

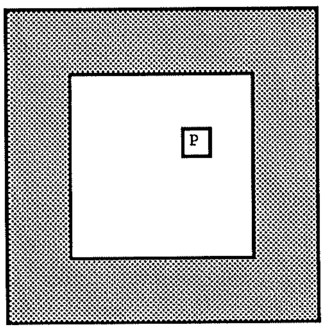


Figure 2.15: The problem of interior/exterior octant classifications for an object with a hole. The probing cell is interior to the external faces and exterior to the internal faces of the object.

- **Connected component labelling:** This algorithm was developed by Tamminen and Samet [Tamm84b]. It comprises of two phases. In the first phase, each polygon of the object is converted into a linear image tree in which black leaves represent the surface area of the polygon. All trees corresponding to polygon faces of the object are then merged into a single tree representing the boundary of the object. In the second phase, the white leaves corresponding to the interior of the object are changed to black by labelling connected octants. This method is faster than the cell classification algorithm, but is also complicated involving projection, conversion, overlaying and labelling. It creates a non-minimal octree which needs further condensation. The time complexity of the algorithm is related to the surface area of the object.

- **Bottom-up:** Tang and Lu [Tang88] designed an algorithm which scan-converts the object into an array of voxels and then processes these voxels to get a leafcode. It requires O(V) time to encode the Cartesian coordinates of voxels into octant addresses and another O(V) time to condense voxels into leafnodes. Here V is the number of voxels in the interior of the object. Preprocessing is needed in their method for exploiting surface coherence and edge coherence to speed up scan-conversion of planes and lines. It is obvious that the method is comparatively slow when the object has a big volume but a relatively small number of polygon faces.

2.7 Constructing Vector Octrees

Carlbom *et al.* [Carl85], Brunet and Navazo [Brun85] employed different methods to construct vector octrees from boundary representations of polyhedral objects. A polyhedral object consists of a list of polygon faces. Each polygon face has a list of edges. An edge is determined by its two end points and each edge can belong to at most two polygons. A point has a spatial location and can be shared by several edges. With the above constraints, the relation among the number of faces (f), the number of edges (e), and the number of vertices (v) is derived, which is known as Euler's Formula, as below

$$v - e + f = 2$$

The approach by Carlbom *et al.* is based on a modified Sutherland-Hodgman clipping algorithm for the special circumstances of polytrees. Firstly, it recursively clips polygons into subspaces until the space contains only one of vertex, edge, face or empty. Secondly, empty octants are classified into black or white using ray casting techniques. The drawback of Carlbom's method is that it creates many polygon segments with pseudo-vertices and pseudo-edges during the process of polygon classification.

Brunet and Navazo's method (Algorithm 2.3) is also a recursive decomposition process. The program keeps lists of faces and vertices associated with the current octant during subdivisions. The subdivision stops when the octant is one of the face, edge, vertex, full or empty cells. The clipping procedure computes the above lists of faces and vertices from the input lists. For each face on the facelist, several tests are implemented in turn as listed below. The complexity of each test is increasing in the sequence.

1. checking if any vertex of the face is inside the octant;
2. intersecting the octant with the bounding box of the face so that trivial non-intersecting faces are rejected quickly;
3. detecting whether the plane of the face intersects the octant;
4. detecting if any edge of the face cuts any face of the octant;
5. examining if intersection of the face and the octant is void.

```
PROCEDURE Build-octree(octant,facelist,vertexlist)
BEGIN
    FOR i=1 to 8 DO
        calculate the suboctant[i] of the octant
        clipping(suboctant[i],facelist,vertexlist,facelist1,vertexlist1)
        IF no faces in facelist1 THEN
           IF the node is interior THEN output(BLACK)
                                   ELSE output(WHITE)
           ENDIF
        ELSEIF only one face THEN output(FACE)
        ELSEIF only one edge THEN output(EDGE)
        ELSEIF only one vertex THEN output(VERTEX)
        ELSEIF suboctant size > 1 THEN output(GREY)
                    Build-octree(suboctant[i],facelist1,vertexlist1)
        ELSE output(GREY, pointer to the faces)
        ENDIF
    ENDFOR
END
```

Algorithm 2.3: Pseudocode for Brunet and Navazo's vector octree construction algorithm.

The algorithm is valid both for plane faces and curved surfaces, but different clipping procedures are employed.

2.8 Applications

In comparison with other 3D representations (for instance CSG, Breps, wireframes), octrees have several advantages: hierarchical data organisation, regularity, uniqueness and spatial addressability. These features facilitate efficient set operations, volume computation, display with hidden surface removal, point location, neighbour finding, and so on. The above octree algorithms need only integer operations and thus are suitable for fast manipulation by special-purpose hardware. There were attempts to build octree machines. However, as will be discussed later, there are problems with octree machines when they are used for geometric modelling and milling. Octree machines have not gained popularity in real applications. In the last ten years, hardware acceleration of floating point operations has become widely used and general computer power has increased tremendously. Octree engines developed in the early years of the decade seem even less powerful. Nevertheless, as general spatial data structures, octrees are still attractive. Octrees have been used in three dimensional image processing, computer graphics, geometric modelling and image rendering. Recently, octrees and other hierarchical data structures have received considerable attention [Same88a, Same88b] and have extensive influence on the design of solid modelling and image processing systems.

2.8.1 3D Image Processing

In image processing octrees show their suitability as a representation scheme for 3D images of any complexity. The objects may be concave, convex, irregular shapes, or have holes and disjoint parts. Such images are often seen in medical image processing (for example, images of a brain). These images are difficult in general to define using other 3D modelling techniques. Here, octrees are more compact than 3D voxel arrays and have the flexibility to probe certain subsets of the data. Efficient representations using octrees can save storage and improve the performance of many tasks such as display of cross-sectional images and the quantitative measurement of 3D objects.

2.8.2 Geometrical Modelling

During the early years of research into spatial data structures, octrees were used in geometric modelling systems for Boolean operations, geometric operations and display by hidden surface removal [Jack80, Meag82]. However octrees only approximately represent geometric objects and are memory extensive. These limitations restrict octrees to be used as the internal representations of geometric objects. Nowadays octrees are mainly used as intermediate representations for the above mentioned applications as well as volume calculation, finite element analysis, surface triangulation and exploitation of spatial locality.

2.8.3 Accelerating Image Generation

The realistic image generation algorithm of ray tracing benefits from the octree data structure through spatial indexing. By indexing objects using an octree, a ray can be tested against objects in a spatially sorted order and can also bypass many irrelevant objects which are far from the ray's path in the space. Thus the number of ray-object intersection tests is reduced considerably. These tests are most time consuming and can cost up to 90 percent of the total processing time for a straightforwardly designed ray tracer.

2.9 Space and Time Complexities of Octrees

To measure a particular format of octrees we often use two criteria—the space efficiency and the time efficiency. The space and the time often trade-off each other. The space efficient schemes usually require longer time to process each element. On the other hand, the time efficient data structures often require larger memory space. Since the cost of memory is becoming much cheaper these days, the time efficient schemes are more appealing on general purpose workstations. However, memory access time still remains a major problem for processing a large amount of data in real time. With improved CPU and special purpose VLSI chips, the processing time becomes faster than the memory enquiry. Therefore the space efficient schemes may also play an important role in real time special purpose systems. We compare the memory required by the three major octree formats—pointer octrees, treecodes and leafcodes, then identify, based on the review in previous sections, the major operations in tree traversals, transformations and constructions.

2.9.1 The Space Analysis

- **Pointer Structures:**

  A grey node in a pointer octree has eight pointers, each pointer uses 32 bit integers on a conventional machine. A leaf node has one data field to indicate its colour. The field can be two bits for black and white images or several bits for colour images. Here we assume 8 bits to record 256 different colours or grey scales. Each node requires one additional bit to distinguish the grey nodes from the leaves. The number of grey nodes (G) is determined by the number of leaves (L) in the tree by the following formula:

  $$G = \frac{1}{7}(L-1)$$

  The total storage (the number of bits) for a pointer tree is

  $$(8+1) \times L + (32+1) \times G$$

- **Treecodes:**

  Each node in a treecode has only one record to indicate its content and one extra bit to distinguish between leafnodes and grey nodes. Again assuming 8 bits per node, the total storage is $9 \times (L+G)$ bits.

- **Leafcodes:**

  The memory requirement of a leafcode is determined by the resolution of the octree space. On conventional machines, each node is represented by a group of three numbers: the level of the node in the tree (this decides the size of the node); the locational code which indicates the node position; the colour of the node. Assuming the level of the root node is 0 and the maximal level of the tree is D, we need $\log_2 D$ bits to represent the first number, $3 \times D$ bits for the second number and another 8 bits to store the colours. The storage required for each node is

  $$\log_2 D + 3 \times D + 8$$

  For a tree of depth 6, we need $29 \times L$ bits. The L is the total number of non-white leafnodes.

An example of a sphere of unit size with the center located at the octree space center is used for demonstration. The results are listed in Table 2.5.

Samet and Webber [Same89b] discussed how to reduce the memory requirement of a pointer tree by packing pointers. They assumed that pointers need not be

	Pointer Trees	*Treecodes*	*Leafcodes*
No. of Nodes Stored	24,769	24,769	9,598
No. of Black Nodes	9,598	9,598	9,598
Minimal Memory Required	297,225	222,921	278,342

Table 2.5: Analysis of memory requirements for the three octree formats.

larger than necessary for distinguishing nodes. Thus different trees can use pointers which are different in size. The size is a function of the total number of leaves, approximately $\log(\frac{8}{7}L)$ for octrees. Therefore for some examples a packed pointer tree requires less memory space than the corresponding pointer-less tree. However, the implementation of such schemes is more difficult. We must know in advance how many leaves exist. When the number of nodes in a tree increases, reallocation of the pointer space may be necessary. The packed pointer tree is more suitable for a static database.

2.9.2 The Time Complexity

Now we analyse the time complexity of the above three octree formats. The key operations of octree algorithms are tree traversal, random octant query and attribute query (for instance colour search). Tree traversal is the basic operation for many octree algorithms. For example algorithms of set operations traverse two input trees in parallel and apply set operators to the corresponding nodes in two trees. Algorithms like displaying an octree and computing volumes traverse one tree in some required order. Random octant query, on the other hand, is most useful for algorithms of neighbour finding, connected component labelling. Attribute query includes colour searches and so on. The efficiency of the above operations determines the overall performance of an octree system.

- **Pointer Trees** are very simple for top-down tree traversal. The nodes in a tree can be visited by following the pointers in any predetermined order. The worst case performance for visiting a random octant is proportional to the maximal level (D) of the tree. The colour search requires the whole tree to be traversed in the worst case, thus its performance is related to the number of nodes in the tree.

- **Treecodes** have a major disadvantage in that tree traversal is order sensitive. If the order of traversal is different from the order in which nodes are stored, traversal algorithms need to use a buffer for storing some nodes or need to rescan the tree. Therefore the numbering of eight brother nodes in treecodes influences the complexity of algorithms. No order is universally efficient for all algorithms. The worst case performance to locate a random octant is proportional to the number of nodes (N) in the tree. Searching a node with a specific colour requires $O(N)$ time too.

- **Leafcodes** can be traversed in an order other than the sorted order of locational codes. However the process requires searching. On conventional computers a search operation has a complexity related to the number of black nodes (n) in the leafcode tree. The worst case performance of traversing the whole tree is $O(n \log n)$. The time to locate a random octant is $O(\log n)$. The colour search needs $O(n)$ time.

Table 2.9.2 summarises the above time complexity analysis.

	Preorder Traversal	*General Traversal*	*Locating Octant*	*Colour Search*
Pointer Trees	O(N)	O(N)	O(D)	O(N)
Treecodes	O(N)	*	O(N)	O(N)
Leafcodes	O(n)	O(n $\log n$)	O($\log n$)	O(n)

Table 2.6: Analysis of the worst case time complexity of the three octree formats for basic operations. (NOTE: * in the table stands for O(N) when using buffers or $O(N^2)$ when rescanning)

2.10 Summary

The memory extensive structures of pointer octrees have simple inner loops for pointer following and are efficient for most tree manipulation algorithms. Treecodes are the most space efficient representations but have tree traversal problems. They are suitable for algorithms using special preorder traversal. They are not efficient for algorithms which need breadth-first traversal or random node access. Algorithms which involve searching attributes are not efficient with both pointer trees and treecodes. These algorithms are implemented by traversing the tree exhaustly. Algorithms of neighbour finding are complicated with pointer trees and treecodes, requiring some additional fields in the basic data structure (for example father pointers and extra links). The memory requirement of a leafcode lies in general between its counterparts in a pointer tree and a treecode. One drawback of leafcodes is that their manipulations involve complex inner loops for encoding and decoding locational codes. However, this can be solved using special hardware when high performance is required. Advantages of leafcodes over pointer trees and treecodes are:

1. Leafcodes can be constructed in either a bottom-up or a top-down manner.
2. Walking from one octant to another, and neighbour finding are simple in leafcodes.
3. Leafcodes are suitable for hardware implementation. With specialised hardware encoding and decoding operations are fast while recursive subroutine calls are slow.

4. Leafcodes are suitable for parallel processing since nodes can be processed independently.

Most octree algorithms involve some tree traversals and the execution time is determined by the number of nodes in the octree. A leafcode has the smallest number of nodes among the three formats. Therefore leafcodes are expected for efficient operations. For general traversal, the key problem of leafcodes concerns the search operation. On conventional architecture, to improve the searching process, leafcodes are normally stored in sorted order. The sorting process need $O(n \log n)$ time and a search requires $O(\log n)$ time. By introducing new computer architectures, as we shall discuss in the next chapter, the searching problem can be solved and octree leafcodes are becoming very promising with advantages in both storage saving and execution efficiency.

Chapter 3

Content-Addressable Memories

3.1 Introduction

In Chapter 2, we have shown that leafcodes have some advantages over pointer structures and treecodes. These include storage efficiency, spatial address coding, simple space traversal, bottom-up construction, and parallel node processing.

However, leafcodes have one major shortcoming which concerns their data organisation on conventional architectures. A leafcode is stored as a linear array in conventional Von Neuman machines. Locating a node in the array requires a search operation which is slow with sequential processing. Although various algorithms have been studied to improve the efficiency of search on sequential machines [Sedg88] the best algorithms need $O(log(n))$ time for searching one node, plus $O(nlog(n))$ time for preprocessing to sort the array. Therefore, search is a major problem. However, solid modelling systems require intensive searches for spatial location and other operations. To resolve such problems we are motivated to investigate applications of searching machines in solid modelling systems. Content-addressable memories are such architectures designed for searching and related operations.

3.2 Content-Addressable Memories

Content-Addressable Memories (CAMs), initially called associative memories [Hanl66], are generally described as a collection of elements which have data storage capabilities and can be accessed simultaneously on the basis of data contents instead of specific addresses. CAMs were first investigated by Slade and McMahon in 1956 [Slad56] and have been actively studied since then. A recent monograph on CAMs was given by Kohonen [Koho87]. An extensive review of research on CAMs can be found in it.

3.2.1 Concepts of CAMs

There are many different ways to classify CAMs—by general concepts; by functionality; by evaluation techniques and so on.

At least three concepts of CAMs exist. The first concept regards CAMs as extensions of conventional memories for improving performance of a variety of logical search and match operations. The second assumes a large CAM acting as a special purpose primary data storage and retrieval device. The third concept concerns an associative processor which contains a CAM with cells capable of arithmetic and associative logic in addition to the search and decision logic. In this thesis, we use the first of these concepts. We think of a CAM chip as a special hardware peripheral within a general purpose computer. The CAM hardware is a subsystem controlled by a conventional host computer which sends instructions to the CAM, transfers data between its internal memory and the CAM and executes various tests on the CAM. Since only some applications need content addressability, a CAM subsystem is used only where it provides an advantage over conventional facilities.

CAMs can also be classified by functionality. These are:

- Exact match CAMs: These CAMs select data which match precisely a global constant.
- General comparison CAMs: The search operations in these CAMs are based on comparisons of equal, not equal, greater than or equal to, less than, and less than or equal to a global constant.
- Functional memories: These CAMs have the binary '0' and '1' as well as the "don't care" value '*' in storage.

When classifying CAMs using their evaluation techniques, we have:

- Fully parallel CAMs: which perform all operations in parallel on all cells of all words.
- Bit serial CAMs: in which operations are parallel on the same bit of each word and serial for each bit of a word.
- Word serial CAMs: which perform each operation one word after another.
- Block oriented CAMs: where several blocks of bits or words are executed in parallel.

The major advantages of CAMs over conventional memories are that a CAM supports an effective and natural way of organising and retrieving data, reduces software complexity and increases computing speed and power. For most applications, the software simplicity is as important as execution efficiency in order to develop and maintain large and complicated systems. In addition to all the above advantages, CAMs are particularly suitable for solving searching problems.

3.2.2 General Features

CAMs have several basic features:

1. Data is stored in any order and no addresses are needed;
2. Data access involves searching for part of the record (known as the key);
3. For a fully parallel CAM, the key in a search register is compared with every word stored in the CAM in one memory cycle.
4. CAM provides some means for multiple response handling.

Multiple response handling involves two separate tasks: one is to perform data transformation operations (that is to write to a group of selected words) in one memory cycle; the other is to select a word in the matched words for reading out to a sequential device. We call the former a `multiple-write` operation, the latter a `multiple response resolver`. Some octree algorithms can be more efficient when the `multiple-write` operation is used. In addition to `multiple-write`, there is also a `single-write` operation when a CAM search yields only one response.

The multiple response resolver is crucial and unique to CAMs. Any CAM search may have zero, one or more responders. To read these responders back to conventional sequential memory, it is necessary to select responders one after another in some order. The order may be random, based on physical locations of words, or based on the values of words. One example of selecting words according to the values is to read out words in a sorted order either ascending or descending. CAMs also need to identify the number of responders of a search operation. Several schemes for this measurement were discussed in the survey paper by Parhami [Parh73]. The simplest scheme gives a binary indication showing either 'no responder' or 'some responders'. A more popular scheme is the one providing a ternary indication: 'no responder', 'exactly one responder', or 'more than one responder'. Two other sophisticated schemes provide an exact or approximate count of the number of responders. Users can select a particular CAM design from these schemes according to the application requirements, hardware complexity and manufacturing cost.

3.2.3 Current Developments in Hardware

The concept of CAM is not new and has been around for thirty years. However for more than two decades only chips with small memories were manufactured because of implementation problems and fabrication cost. Not until recent years have the advances in VLSI technology made it feasible to design a CAM chip with wider words and trit (0, 1, or *don't care*) storage. A *don't care* is a cell state that matches either 0 or 1 presented in a search pattern. The New York State Center for Advanced Technology in Computer Applications and Software Engineering (CASE center) at Syracuse University is one of the research organisations actively working on the design and development of CAMs for a wide area of applications [Brul88].

A CAM is composed of cells which are capable of storage and pattern matching. Each cell operates simultaneously to compare the incoming search pattern with its stored data. The simplest CAM cells can be built from static RAM circuits by adding transistors to provide the comparison functions. A static cell requires 15 transistors for storing a `trit`. Wade and Sodini [Wade87] have developed a 5-transistor dynamic cell for `trit` storage.

As an example we summarise some basic features of Wade and Sodini's dynamic CAM cell. The cell layout is shown in Figure 3.1. The cell can store trit values. To store a '0' transistor T1 is on, a '1' T2 is on, a '*' neither transistor is on. Search operations are carried out in parallel using a row match line passing through every cell of each word. Typical cell search times are in the 50–100ns range. A single chip can have 64 rows of 32 cells. Several such chips can be cascaded vertically to form a CAM with a large number of words. The entire CAM, or selected rows and/or columns, can be searched or written in a single memory cycle. These advances thus made it possible to design practical application systems on the basis of CAMs. Applications of CAM are mainly found in pattern recognition, image processing, evaluation of logic programs and database systems. The cost and speed of CAMs are determined by

- the cost and access time of memory cells;
- the degree of parallelism and interconnection of cells;
- the amount of logic for each element.

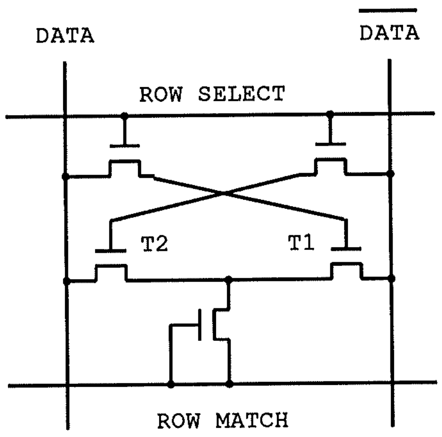


Figure 3.1: Wade and Sodini's dynamic CAM cell (courtesy of Richard Williams).

3.3 Syracuse Quadtree/Octree CAM

Oldfield *et al.* [Oldf88] have proposed a new CAM chip for storing and processing quadtrees and related spatial applications. This chip is a pack of Wade/Sodini cells described in the above section. The chip is currently being designed at Syracuse University and has been simulated at Computer Laboratory, University of Cambridge [Will88b]. Before using the CAM, one should be aware of the overall architecture and functions of this specific type of CAM. Here we give a brief summary of the architecture and features. Its functionality will be described in full detail in the next section.

3.3.1 Architecture

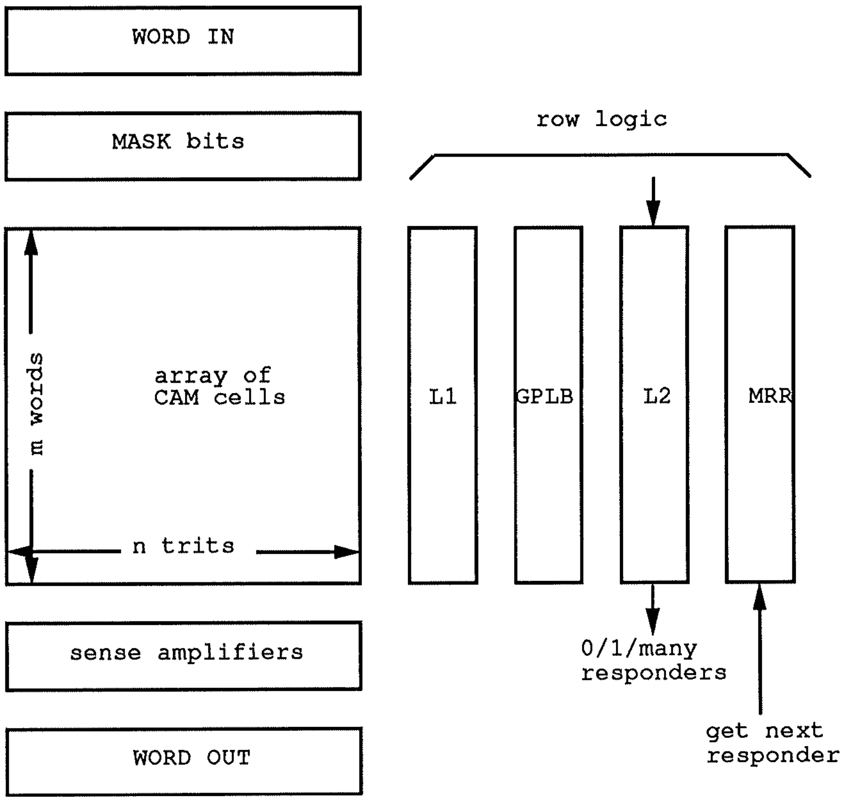
Figure 3.2 shows the floor plane of the Syracuse CAM chip. The chip is organised around an array of words (m), each word containing a number of cells (n). Each cell can store information as one of three states: a *don't care* state, which is represented by $*$, and conventional 0, 1 states. The storage for three states is referred as trit storage, analogous to bit storage. Each word has a bit to determine whether it is in use or free. The design is a general purpose one without any assumptions about the contents of each word. Each word consists of a single physical field of a fixed width. For current research we assume that the width is 32 trits. The application will decide the logical fields within each word.

A simple row logic is added to the CAM design. It is a hardware built into the CAM chip to handle multiple responders and to provide complex searches which are combinations of a pair of simple searches. The row logic which contains a multiple response resolver (MRR) and a general-purpose logic block (GPLB), is repeated identically on every row and is controlled by vertical micro-code control lines. Two searches can be combined using one of the sixteen GPLB functions. These functions will be listed in Section 3.4.

	0	1	*
0	T	F	T
1	F	T	T
*	T	T	T

Table 3.1: Comparing a cell state with a query.

The most important operations of CAMs are search and write operations. When a search operation is called, a search pattern is presented as a word of trits and is compared with all words simultaneously to see if any word matches. A search operation processes all rows and all columns in parallel. The $*$ in a search pattern can match any cell state, while the $*$ in a cell can match 0, 1 or $*$ in the search pattern. Table 3.1 demonstrates cell comparisons. Table 3.2 shows a search pattern and a group of CAM words to be compared. After a search operation, CAMs need to distinguish whether there are zero, one or more than one matching words. If multiple responders are yielded for a search, the MRR works to select one responder to be read back into a sequential device. The MRR will continue to select subsequent responders until all responders are read back. The current design supports two kinds



GPLB = General Purpose Logic Block
MRR = Multiple Response Resolver
L1, L2 = 1 bit latches

Figure 3.2: The floor plan of Syracuse CAM chip (courtesy of Richard Williams).

Search Pattern	**1*1***
Trit Words	0*1*
	1000
	1011
	11**
	.
	.
	.

Table 3.2: Comparing all words with a query (matching words are shown in boldface)

of write functions: a `single-write` and a `multiple-write`. The `multiple-write` operation writes to selected words in parallel within a single memory cycle. During the write operations a register is used to mask out columns in selected words which are not to be changed.

A single chip can have 128 words of 32–64 trits on it. Hundreds of such chips can be cascaded vertically to get a CAM of reasonable size. An octree application needs several thousand words for a simple scene. The typical execution time for a search or a `multiple-write` operation is of the order of 100ns. Chip cascading does not affect the cycle time.

3.3.2 Interface

The interface has two separate parts: a low level interface and a high level interface. The low level interface is built on top of the general purpose CAM chip. It is application independent. In our CAM simulator, the low level interface works as a procedural interface which models the chip at a functional level. The memory operations include the following:

- The *NoOp* function evaluates the row logic.
- The *SetWriteMask* function sets the mask register to the specified value. A 0 bit in a column will disable subsequent writing to the column.
- The *SearchCAM* function presents a word with a trit pattern to be compared with the CAM. All active words are compared with the pattern. The response bit is set on each matched word.
- The *WriteCAM* function writes the word into all responding rows on all the columns selected by the mask register.
- The *ReadCAM* function reads the next responder and resets its response bit.

The high level interface is designed according to applications. Different applications, such as quadtrees for geographical information systems and octrees for 3D modelling, can be built on top of this interface. At this level, a physical CAM word is divided into a number of logical fields. The interface designer must know the number of logical fields and the width of each field. This information is provided according to the requirement of the particular application. For quadtrees/octrees and related spatial organisations three fields are normally required, because that each node in a quadtree or an octree has three sources of information: the tree identity; the position and the size of the octant; the colour associated with the octant. These three fields are named as: the id field which keeps the tree identifier; the location field to store the locational code of a node; the colour field which records the content of a node. Each word in the CAM represents one node of an octree.

The applications see the high level interface as a group of procedure calls. Each procedure has parameters related to the above logical fields. The details of functions of the high level interface for octree applications will be described in the next section.

This interface is similar to William's [Will88b] interface for quadtrees except that the field width and some functions are different.

3.4 CAM Octrees and Functions

3.4.1 Storing Octrees in CAMs

A CAM octree is an octree stored in CAM in the leafcode format. Each leaf contains an identity, a locational code and a colour. The locational code can represent the position and size of each octant in one record as described in Section 2.3. The sample image of Figure 2.2 with an object of two black nodes labelled as A and B is used again here. The corresponding CAM content is listed in Table 3.3 and the background nodes are ignored. In Table 3.3, a CAM word is divided into three fields: *id*, *location* and *colour*. The sum of the widths of three logical fields must not exceed the width of a CAM word.

	Id	*Location*	*Colour*
A	01	111001$***$	10
B	01	101$******$	10

Table 3.3: CAM contents for the sample object

3.4.1.1 The ID Field

The *id* field is used to identify different octrees which are stored together in a CAM at the same time, and to indicate "free" words. It is assumed to have a width of 3 trits. But only bit patterns are actually stored. All words are initially set to free. An entry for a new leaf of an octree is added to the CAM by finding a free word and replacing it with the *id* for the new entry. If no more free words are available, the CAM is then full. The CAM system responds with error messages indicating CAM overflow. Entries are deleted by marking them as free. This can be done either by a `single-write` operation or by a `multiple-write` operation.

3.4.1.2 The Locational Field

The *location* field stores spatial information about leaves of an octree. It is used to locate an octant in the space. We assume that the space origin is at the left-bottom-back corner of the universe, although other coordinate systems are also possible. Different coordinates influence only the encoding and decoding of locational codes. The encoding process is as follows. The bit patterns of the x, y and z coordinates of the vertex on an octant's left-bottom-back corner are calculated. The size (s) of the octant is indicated by replacing the bottom m bits (where $m = \log_2 s$) of the x, y and z patterns with $*$. The trit patterns of x, y and z are then obtained. Interleaving these trit patterns yields the locational code. For example, the bit patterns for the node B in Figure 2.2 are x (100), y (000) and z (100). When the size is considered, the bit patterns are replaced by the trit patterns as x (1**), y (0**) and z (1**). The x, y and z trit patterns are interleaved to give the locational code (101******) for the node B as in Table 3.3. The width of the *location* field is determined by the image resolution (for example 9 trits are required for an $8 \times 8 \times 8$ image).

3.4.1.3 The Colour Field

The *colour* field indicates the colour or other information associated with each octant. It can store colours for raster octrees and indices for vector octrees. With raster octrees, multiple colours can be handled. The width of the field determines the number of different colours which can be distinguished. For example a 5-trit colour field can store 32 different colours. With vector octrees, three trits in the *colour* field are reserved for indicating node types. Indices to surfaces are held in the remaining part of the *colour* field. Details of vertices, edges and faces are stored in the conventional main memory. The width of the field depends on the maximum of the following three numbers: the number of vertices, the number of edges and the number of faces.

3.4.2 Basic Data Types and Constants

We have mentioned trits several times. The structure of TRITS is a basic data type associated with the Syracuse CAM architecture. It is defined as a single record in the high level interface as well as in the applications. A TRITS has two parts: a data and a dont_care. The data structure of TRITS and some basic constants are listed below.

```
typedef struct {unsigned data, dont_care;} TRITS;

TRITS ANY      = { 0, ~0}, /* ***...*** */
      FREE     = { 0,  0}, /* 000...000 */
      WHITE    = { 1,  0}, /* 000...001 */
      BLACK    = { 2,  0}, /* 000...010 */
      RESERVED = {~0,  0}, /* 111...111 */
```

Other constants include the maximum depth of an octree, the size of the octree space, and the field masks for write operations. A field mask is related to the width and the position of the field within a CAM word and is determined by applications. In the octree application, they are determined by the maximum allowed tree depth.

```
typedef struct {int x, y, z, size;} OCT;

#define DEPTH       ()  /* the lowest level of octree subdivision */
#define SIZE        ()  /* the size of the root octant */
#define ID          ()  /* select 'id' field for write operations */
#define LOCATION    ()  /* select locational field for write operations */
```

```
#define COLOUR        ()  /* select colour field for write operations */
#define ALL           ()  /* select the whole word */
/* and so on */
```

3.4.3 Functions

Basic functions of octree CAMs can be classified into two categories. One includes those involving CAM hardware operations. The other includes those high level functions for octree utilities. The former is generally independent of applications and only the widths of the three logical fields are influenced by applications. Whereas the high level utility functions are application dependent, for instance related to quadtrees or octrees. The CAM functions in the first category are listed below in Algorithm 3.1. We give only procedure names and descriptions.

```
int Search (id, location, colour)
   TRITS id, location, colour;
   /* return TRUE if a search yields any responders */

int ReSearch (op, id, location, colour)
   short op;   TRITS id, location, colour;
   /* search the CAM again, after a Search() operation,
   using a GPLB operation on previous responders---return TRUE
   if any responders remain. This allows two consecutive searches
   to be combined with any GPLB operations */

int MultipleResponse ()
   /* return TRUE if a search has more than one responder */

void SingleWrite (mask, id, location, colour)
   unsigned mask;   TRITS id, location, colour;
   /* write to the field (indicated by the mask) of the current
   (or first) responder */

void MultipleWrite (mask, id, location, colour)
   unsigned mask;   TRITS id, location, colour;
   /* write to the field (indicated by the mask) of all responders */

void Read (id, location, colour)
   TRITS *id, *location, *colour;
   /* read the current (or first) responder */

int NextResponder ()
   /* get the next responder---return FALSE if last responder */

void AddEntry (id, location, colour)
   TRITS id, location, colour;
   /* add a new word to the CAM by writing to a free entry */
```

Algorithm 3.1: A list of high level functions for CAM hardware operations.

These procedures hide the CAM implementation details. From the user's point of view, a CAM system is a group of functions operated with several logical fields. The underlying interface will convert these logical fields into a single physical field and call the low level CAM interface for CAM hardware operations. The actions taken by these procedures involve various low level system operations including that of setting various registers; performing 'No Op' function to set each selected line to high; searching or reading the CAM and so on. The GPLB operations of the ReSearch function include sixteen actions which are listed in Algorithm 3.2.

```
typedef ENUM {
        GPLB_CLEAR,             0
        GPLB_A_NOR_B,           1
        GPLB_A_AND_NOT_B,       2
        GPLB_NOT_B,             3
        GPLB_NOT_A_AND_B,       4
        GPLB_NOT_A,             5
        GPLB_A_XOR_B,           6
        GPLB_A_NAND_B,          7
        GPLB_A_AND_B,           8
        GPLB_A_EQUIV_B,         9
        GPLB_A,                10
        GPLB_A_OR_NOT_B,       11
        GPLB_B,                12
        GPLB_NOT_A_OR_B,       13
        GPLB_A_OR_B,           14
        GPLB_SET               15
} GPLB_OP;
```

Algorithm 3.2: Sixteen GPLB operations

The high level utility functions for octree applications are listed in Algorithm 3.3. The encoding and decoding procedures depend on the choice of coding schemes (interleaving in x, y, z order or z, y, x order) and space origins. The procedure `RunList()` is a key operation which is used in some algorithms of the following chapters. Its function is to split one dimensional orthogonal lines (also called rays) into segments which fit the octree subdivisions. It generates a list of integers from the bit patterns of the coordinates and length of a ray. Each integer gives the position of a segment of the ray. The length of the segment can be derived from the position of its subsequent segment. All segments are fitted into the spatial restriction of octrees (that is powers of 2). Rectangles in 2D and rectangular parallelepipeds in 3D can be processed by calling the `RunList()` procedure several times and each processes along one axis.

```
void DecodeParall (location, x, y, z, w, h, d)
   TRITS location;   int *x, *y, *z, *w, *h, *d;
   /* return the position, width, height and depth of a rectangular
parallelepiped for the locational code */

TRITS EncodeParall (x, y, z, w, h, d)
   int x, y, z, w, h, d;
   /* return the locational code for the rectangular parallelepiped */

void DecodeOct (location, octant)
   TRITS location;   OCT octant;
   /* return the octant from the locational code */

TRITS EncodeOct (octant)
    OCT octant;
    /* return the locational code for the octant */

int DecodeNumber (n)
   TRITS n;
   /* return the colour from the trit code */

TRITS EncodeNumber (i)
   int i;
   /* convert the colour to the trit code */

int Size (location)
   TRITS location;
   /* return the maximum of the width, height or depth
   of a rectangular parallelepipeds in the locational code */

TRITS SetTrit (word, position, value)
   TRITS word;   int position, value;
   /* set the specified trit in selected words to the required value */

int RunList (start, length, list)
   int start, length, list[];
   /* return a list of integers for a run-length */
```

Algorithm 3.3: A list of octree utility functions of CAM high level interface.

The above procedures of the high level interface can be supported by simple front-end hardware because that they are relatively standard for octree applications. The efficiency of the overall system can be further improved with this additional front-end hardware.

3.4.3.1 The RunList() Procedure

Given a ray with a position and a length, we want to fit the ray into the space of an octree subdivision. This process involves splitting the ray into several maximal segments, each of which must align properly within an octant (see Figure 3.3). A segment has some features which are used to determine where we can split the original ray. For a segment at level i of an octree subdivision, we know that it has a length of $2^{(D-i)}$ (where D is the maximum depth of the octree) and a start coordinate which is divisible by the length. The number of segments varies for different rays, but the upper bound of this number is determined by the depth of octrees.

$$MaximumNumberofSegments = \begin{cases} 1 & \text{if } DEPTH = 0 \\ 2 \times \text{DEPTH} & \text{otherwise} \end{cases}$$

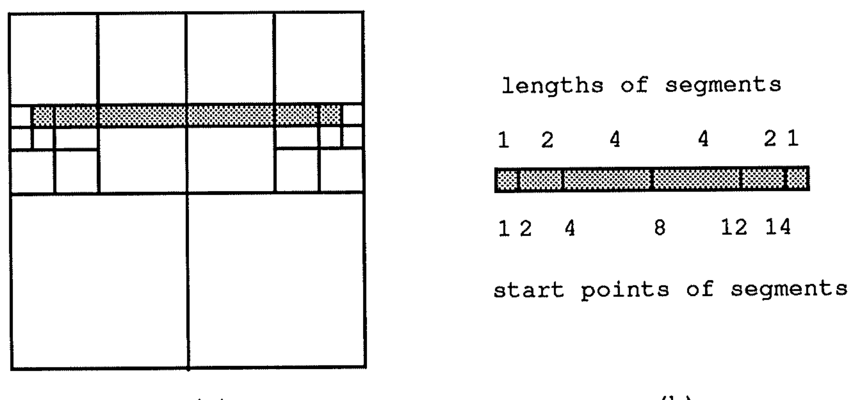


Figure 3.3: Generating a RunList from a ray: (a) a ray in a 2D region of quadtree subdivision, (b) the same ray is split into maximal segments.

These segments can be computed either recursively or sequentially. The recursive procedure [Akma89] searches for the longest segments at each stage until the current length equals 1. For the ray in Figure 3.3, the segments are obtained in the following order (the number shown below is the start position of each segment),

```
Segments   (4)  (8)
           (2) (12)
           (1) (14)
```

The sequential splitting process finds the segments in an increasing order of start positions. For the same ray shown above, the order is (1) (2) (4) (8) (12) (14). We employ the sequential splitting algorithm in our `RunList()` procedure because some algorithms are sensitive to the order.

3.5 The Advantages of CAM Octrees

The most obvious advantages of CAMs are fast searching, parallel updating and pattern matching capabilities. Each search requires only a single memory cycle as it simultaneously compares every entry in the CAM with the given search pattern. Responding words can be updated in parallel by the `MultipleWrite()` operation which processes all responders in a single memory cycle. Pattern matching allows rectangular parallelepipeds (RPs) of different shapes and sizes to be compared in order to test volume intersection. These capabilities provide us some interesting features for octree processing as follows.

- **Colour queries:**

  1. Searching for octants of a specific colour can be implemented as a colour search. The search pattern has *don't cares* in the *id* field and the *locational* field, and a required colour in the *colour* field.
  2. Searching for nodes that are not a given colour can be done by combining two searches using a GPLB ($A \cdot \overline{B}$) operation. The first search activates nodes which can be any colour. The second one searches among the above active nodes those which are not of the given colour using the `ReSearch()` operation.

- **Space queries:** A space search involves an enquiry with a specific locational code pattern. Responses to the spatial enquiry are the stored octants intersecting with the enquiry volume. The simple space queries benefit from CAM trit storage and pattern matching facilities. The query space can be one of the following:

  1. An octant of any size, of which two extreme examples are the root space and a voxel.
  2. A restricted RP, for example a scan line or a slice. The restriction is on the relation of the start coordinate and the length of the RP along each axis.

- **Size queries:** All octants of a specific size can be found by searching for either a 0 or a 1 (not a *don't care*) in specific trit positions of the *locational* field.

- **Combinations:** A colour search can be combined with any space query or size query.

If implemented with conventional octrees, the above searches are complex and need many calculations. With CAMs, only minimum computations combined with a few pattern matching operations are necessary. A search can be operated with any of the three logical fields. It is possible to search the *id* field for all nodes in a specific octree; to search the *colour* field for a specific colour or index; to search the *locational* field for looking up different spaces. It is unique to CAM octrees that searching by any colour is as efficient as searching by locations. The features described above are the keys to many efficient CAM algorithms which will be presented in this thesis. They will be fully explained in Chapters 4 & 5.

3.5.1 Examples

Figure 3.4 shows examples of locational searches, with their corresponding search patterns in Table 3.4. Items which can be searched in the *locational* field are octants (for instance ** 000****** **), RPs (for example left half space: ** 0******** **), and so on. An enquiry octant may match one of the following: an equal-sized octant, a larger octant, one or more smaller octants or nothing (when the background is not stored). For instance, the search of the root space (** ********* **) gives the nodes A and B as responders.

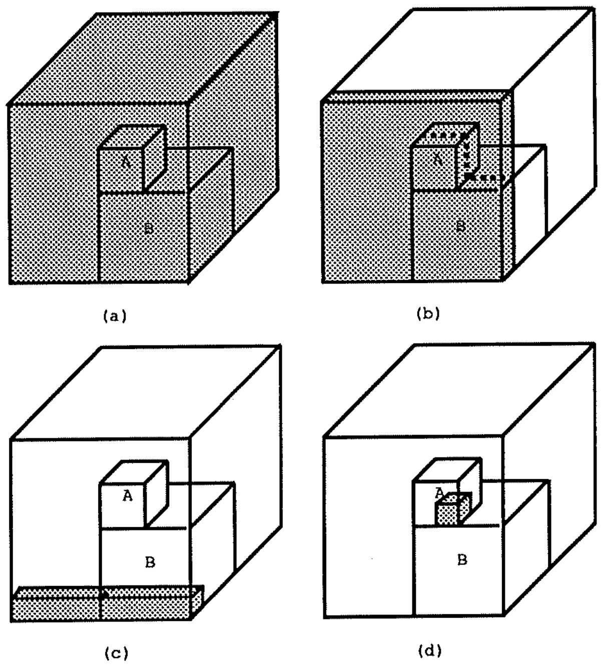


Figure 3.4: Examples of locational queries.

Search Space	*Search Pattern*	*Matches*
(a) root	*********	A,B
(b) slice	**1 **1 **1	A,B
(c) line	*01 *01 *01	B
(d) voxel	100 010 110	A

Table 3.4: Search patterns as CAM words for locational queries.

3.6 The White Nodes

With pointered octrees and treecodes, all white nodes must be kept in tree structures because the coordinates of a node are determined by its relation to other nodes in data structures. However, for leafcodes white nodes are no longer necessary. Each node has its location and size stored in the record and can be detected by decoding it. Therefore white nodes are often not stored in leafcodes. By ignoring white nodes, the number of records of an octree can be reduced. For some images, white nodes can be up to 80–90 percent of total nodes. Having less records is important for conventional leafcodes not only for the sake of saving memory but also for speeding up node enquiry. This is because the searching speed on a conventional machine is determined by the number of records. The node inquiry time can be significantly reduced by not storing white nodes.

A CAM octree is a leafcode by nature. Its main differences from a conventional leafcode are that a CAM octree is stored in CAM and its node enquiry is implemented in parallel. The time spent for searching a node is not determined by the number of nodes of a tree. Therefore it is less important, in terms of speed of the query, whether to keep white nodes or not . However, as will be shown below, some algorithms can be simpler if white nodes are stored while others are less sensitive to them.

When white nodes of a CAM octree are stored, they are treated in the same way as nodes with colours. Algorithms like inverting black and white spaces, flood-fill, scan conversion, and so on can benefit from simple white node operations and therefore are very elegant and efficient. However, in octree applications, the above algorithms are used less often than other algorithms such as volume computation in solid modelling and back-to-front display in medical image processing. Here, the background information is irrelevant.

3.6.1 Recovery and Deletion of White Nodes

As noted before, searches in a CAM octree are not related to the number of nodes in the tree. Some algorithms such as interior filling and inverting black/white spaces are simpler and more efficient when background nodes are stored. Others are better if white nodes are not stored. If we did not store white nodes in the original CAM octree, we may need to recover them for some stages when requirements arise. On the other hand, we may also want to delete white nodes after some operations. These processes are shown in Algorithms 3.4 & 3.5 in pseudo C code.

```
PROCEDURE recover_white(TRITS id, OCT octant)
BEGIN
    location = EncodeOct(octant)
    IF (!Search(id, location, ANY)) THEN
        AddEntry(id, location, WHITE)
    ELSE IF (MultipleResponse()) THEN
        FOR i = 0 TO 8 DO
            recover_white(id, SubOctant(i, octant))
    ELSE  /* Only one response */
        Read(&dummy, &location, &dummy);
        IF (octant.size > Size(location)) THEN
            FOR i = 0 TO 8 DO
                recover_white(id, SubOctant(i, octant))
        ENDIF
    ENDIF
END
```

Algorithm 3.4: Algorithm for recovering white nodes of a CAM octree

When an enquiry returns no responder, the space corresponds to a white node. If a search returns multiple responders, or returns a single responder which is smaller than the enquiry space, the enquiry octant needs further subdivision. The complexity of the algorithm is O(N), if using the number of searches in CAM as the measuring metric. The number N is the total number of nodes including black, white and unstored grey.

The algorithm for deleting white nodes simply contains a `Search` operation and a `MultipleWrite` operation. Its complexity is O(1) while its counterpart on conventional leafcodes is O(n) (n is the number of leaves in the tree).

```
PROCEDURE delete_white(TRITS id, OCT octant)
BEGIN
    IF (Search(id, EncodeOct(octant), WHITE)) THEN
        MultipleWrite(ID, FREE, ANY, ANY)
    ENDIF
END
```

Algorithm 3.5: Algorithm for deleting white nodes of a CAM octree.

3.7 CAM Vector Octrees

A CAM vector octree is stored similarly as a CAM raster octree except that the `colour` field is divided into two subfields. We call the first one a subfield for mark trits (see Table 3.5) which indicate the type of a node. We chose Durst and Kunii's definition [Durs89] of vector octrees (called integrated polytrees) in which eight leaf types are distinguished. The node types are black, white, face, edge, edge', vertex, vertex', vertex". The black and white nodes are homogeneous nodes and can be grouped as one type. Hence seven types must be indicated. Three trits are thus used for node types. The second subfield is either an index to the address of a face, an edge and a vertex, or a colour indicating the black/white/other colour of the node. The index is used as a reference to the location of elements of Brep in the cases where the node is one of faces, edges or vertices.

Mark Trits	*Node Types*
000	black/white
001	face (f)
010	edge (e)
011	edge (e')
100	vertex (v)
101	vertex (v')
110	vertex (v")

Table 3.5: Three trits are used in the locational field to mark the node types of a CAM vector octree, and the corresponding node types are listed.

Additional functions for encoding and decoding vector octrees are listed below.

```
TRITS EncodeIndex (ch, i)
  char ch; int i;
  /* IF ch = 'v' OR 'va' OR 'vb', return the trit code for the given vertex */
  /* IF ch = 'e' OR 'ea', return the trit code for the given edge */
  /* IF ch = 'f', return the trit code for the given face */

int DecodeIndex (n)
  TRITS n;
  /* return the index of vertex, edge, or face for the given trit code */

int DecodeType (n)
  TRITS n;
  /* return the node type for the given trit code */
```

Algorithm 3.6: A list of additional functions for vector octrees

Chapter 4

Operations on CAM Octrees

4.1 Introduction

Chapter 2 reviewed algorithms for octree manipulation, display and construction. In this chapter new algorithms are presented for the above operations, based on the novel CAM octree architecture described in Chapter 3. We shall show that most new algorithms are more efficient and significantly simpler than their counterparts on conventional architectures. This is especially true for octree construction algorithms.

As the CAM octree data structure is a variation of the leafcode data structure, CAM algorithms show their outstanding advantages for operations suited to leafcodes. Most conventional operations have execution time proportional to the number of nodes in the input tree, while for the CAM version of algorithms, some are found linear in the depth of the maximum space subdivision, some are linear in the number of nodes in the output tree.

The following sections describe general algorithms of octree manipulation which include geometric transformations, set operations, volume computations, displays, cross section generations, condensation and neighbour finding. We then discuss the general trends of conversion between octrees and other representations. This is followed by several new algorithms for constructing raster octrees and vector octrees from other representations.

4.2 General Algorithms

4.2.1 Geometric Transformations

4.2.1.1 Rotations and Reflections

Algorithms for 90 degree rotations and reflections of CAM octrees are similar to their leafcode counterparts which involve permutations of octal numbers in locational codes. In leafcodes, the mapping from the old value to the new value is implemented for each octal digit of each node in turn. It takes O(n $\times$ D) time for permutation and an additional O(n $\log n$) time for sorting the output octree where n is the number of nodes and D is the depth of the tree. The node mapping in CAM octrees is more efficient by exploiting special CAM capabilities of parallel `search` and `multiple-write`. Corresponding to one octal digit of a leafcode, a trit-triple is used in CAM. Mapping is implemented for each of the eight values of trit-triples in the locational field. This can be done globally using `search` and `multiple-write` operations.

The mapping process starts from the most significant trit-triple columns which correspond to the left-most three columns in the *locational* field. The following operations are implemented. Firstly, the program searches on these columns for any triples which have *don't cares* only, and puts the responding nodes into a temporary tree by multiple-writing their *id* field with a reserved value. This operation isolates the *don't cares* from other values. It is necessary since the transformation of *don't care* remains unchanged. On the other hand, this operation must be implemented before digit mapping of other values in the columns because the *don't care* triples would respond to all enquiry for other values. Secondly, after isolating *don't care*, the program searches and transforms triples with a specified value into the new value, and marks their *id* field as the above mentioned *id* in one memory cycle. There are eight values in total to be searched and transformed. Finally, the nodes with the reserved *id* is set back to the original *id*. The process is repeated for other columns until all columns are transformed.

The execution time is independent of the number of nodes in the tree. The number of searches and writes is proportional to the maximal depth of the space subdivision (D) rather than the number of nodes in the octree. Thus the performance of CAM octree transformation algorithms are much efficient in comparison with the performance of their conventional leafcode counterparts.

Rotations about an arbitrarily-oriented line need to reconstruct the tree. Here node splitting and condensing are involved. In these cases, the nodes in the original tree must be read out from CAM sequentially, then rotated and split to form new leaves. The resulting tree may need to be condensed as the newly formed tree may be a non-minimal tree (see Page 27). Here, CAM can be used to speed up the process of tree reconstruction and condensation, which will be discussed later.

4.2.2 Set Operations

The improvements of octree algorithms for the three basic set operations (AND, OR, NOT) using CAM depend on the nature of the operations. The union and intersection operations require traversal of two input trees in parallel and comparison of their corresponding nodes. If implemented by top-down recursive traversal, then the execution time of CAM octree union and intersection algorithms is O(N) (N is the number of nodes including grey, white and black in the output octree). This is the same as that for the pointer tree which has been discussed in Section 2.4.

The CAM octree can also be traversed in a different order, for instance a size order. As the union and intersection operations concern only the black nodes, the program can visit one black node each time and use it as the mask for processing nodes occupying the same space in the other tree. It would be better if the mask space selected is large. In these cases a large node in one tree may correspond to many small nodes in the other tree and the `multiple-write` operation can play an important and effective role. Therefore the largest black nodes are visited before the smaller ones. With this approach, the worst case complexity is O(n) where n is the number of black nodes in the output thus the union and intersection operations are more efficient than in the previous approach.

The NOT operation inverts the black and white spaces of an octree. In solid modelling systems, this means that the object and its surrounding space is inverted. Although this is seldomly used in 3D applications, the process of inversion is extremely simple with a CAM octree. It is a swap procedure which has three pairs of a `search` operation and a `multiple-write` operation, each pair uses only two memory cycles. The first pair searches and marks all white nodes with a temporary *id*. The second pair changes all black nodes to white. The third pair changes all nodes with the temporary *id* back to black nodes.

4.2.3 Volumes and Other Integrals

Volume computations with conventional octrees are implemented by traversing the octree and adding up the volumes of its black nodes. All the nodes in the tree are visited once thus the time complexity of the algorithm is proportional to the number of nodes in the input tree. The CAM octree volume computation algorithm depends on the functionality of the multiple response resolver of a particular CAM architecture. The current design of Syracuse CAM distinguishes zero, one, and many responders. No exact counting of the number of responders is supported. With this design the volume computation is performed by searching the black nodes and then reading them out sequentially. As in the case of leafcodes, the complexity of the algorithm based on Syracuse CAM is proportional to the number of black nodes in the tree.

Identifying the number of responders of a CAM search is an important part in many CAM designs. There exist four schemes for counting the number of responders [Parh73]. They are:

1. a binary indication—no responder or some responders;
2. a ternary indication—no responder, exactly one responder or more than one responder;
3. an approximate count of the number of responders;
4. an indication of the exact number of responders.

Foster and Stockton [Fost71] designed a circuit which counts none, one, two, ..., N responders to a search in a CAM. They used only "full adders" as elementary building blocks and minimised propagation delay for the extra counting function.

Assuming a function for counting the exact number of responders is available, volume calculation of a CAM octree becomes even more efficient. The algorithm is simply implemented by several searches, each followed by a count. Each search enquires for octants which have a specific size and have the colour of the required object. Such searches can be easily implemented in octree CAM which provides functions for size and colour searches as demonstrated in Section 3.5. The maximum number of possible sizes of octants in the space is determined by the maximal depth (D) of the space subdivision. Therefore, volume can be calculated with time complexity of O(D).

However, counting the exact number of responders can be implemented only on a small scale. It is not practical to design an architecture to count a large number of responders since the size of the counting register is limited. Still the algorithm can be reorganised to fit into a small counting register, such as one with a size of 64. This means that it can distinguish 0, 1, ..., 62, more than 62 responders. We noticed that the maximum number of nodes at each level (l) of the subdivision is 8^l while the maximum number of nodes for a specific object (or colour) at each level is even smaller $(8^l - 1)$. Thus the above simple CAM octree volume computation algorithm is redesigned in a recursive way. Starting from the largest size, the following process is repeated for each size in turn. The whole octree space is searched for octants of the specified size. If the number of responding nodes is less than 62, we multiply the number of responders with the size then add the result to the volume. If the number of responders is more than 62, then the current enquiry space is subdivided and each subspace is searched again for octants of that size. The process ends after all possible sizes have been examined. The subdivisions in this algorithm are dynamic. For some small octrees, this adaptive algorithm can be as efficient as the previous simple CAM volume computation method based on CAMs with an unlimited counting register.

4.2.4 Displaying Octrees

As discussed in Section 2.4, the algorithm for displaying leafcodes has one major shortcoming concerning searching operations. Node searching is necessary when traversing the tree in an order other than ascending order. On conventional machines searching for an element from a data file with a sorted linear array needs $O(\log n)$ time. The process for searching a large file, for example a data base of a typical 3D image with more than 100,000 nodes, is inevitably slow. Whereas in CAM octree searching is easy. There is no need for sorting and the display algorithm is simple.

The CAM procedure starts from the top level of the tree and searches downward recursively. If more than one octant responds to the space query then the space is subdivided into eight suboctants. The procedure is repeated for the eight suboctants in turn. The order of visiting is as follows. The farthest suboctant is visited first, then its three face-connected brothers, three edge-connected brothers and finally the vertex-connected brother. The farthest octant is determined by the viewing position and can be easily derived from the viewing normal vector. This process continues until there is only one hit returned, then the corresponding leaf node is displayed.

We can also use CAM to display the border of an object represented as an octree. The colour field is divided into two parts. The first part has one trit used to indicate whether the node is a border node or an inner node. It is initially set to 0 (corresponding to an inner node). The second part of the colour field is as usual an index to the specific colour. Before displaying, a boundary finding subroutine is called to mark all the leaves which have at least one white neighbour node as border leaves. Node marking is done by changing the first part of the colour field from 0 to 1. When displaying the octree, the program searches for surface nodes and visits the responders in the back-to-front order. The number of border leaves, thus the number of CAM read operations, is far less than the total number of the nodes in an octree.

The above algorithm has two advantages. Firstly the boundary finding is easy for CAM octrees by using a neighbour finding technique which will be discussed later. Secondly the returned border nodes are leaves of the CAM octree and are not necessary voxels. Compared to Gargantini's border display method [Garg86], the CAM method requires no extra memory for storing 3D border voxels, and uses less time for projecting border nodes onto the screen since the number of leaves to be displayed is small. However, the border must be recomputed if two octrees are merged to form a new octree through union or intersection. The CAM octree border displaying algorithm can be further improved by marking only those leaves which have at least one visible face. A visible face has a surface normal pointing to the eye position. Such an improvement makes the number of nodes to be displayed even smaller. The choice of the visible faces depends on the eye position.

4.2.5 Cross section generation for octrees

The algorithm for generating orthogonal cross section images of a CAM octree is very simple and efficient. It involves only one `CAM search` operation and several `CAM read` operations to get a quadtree which corresponds to a cross section of the octree. An enquiry plane of the cross section is constructed and encoded in the locational code (as demonstrated in Figure 3.4 of Section 3.5). Then the CAM is searched with the above locational code. All octants which intersect the plane will respond to the search. By reading and decoding these responders we obtain quadrant addresses of responding leaves and thus the required quadtree. The resulting quadtree may need to be condensed since some nodes may have the same colour and at the same time form a complete group for a larger quadrant.

In the CAM algorithm it is not necessary to calculate the distance of each node to the plane, as is required in the corresponding algorithm of pointer octrees. It is also not necessary to complicate the data structure, as in the treecode algorithm (for example introducing the dectree [Oliv84]). The complexity of the CAM algorithm is proportional to the number of nodes of the output quadtree of the cross sectional image.

4.2.6 Condensation

The condensation algorithm for conventional octrees examines each node in turn in a depth-first order, to see whether eight sibling leafnodes are of the same colour. If they are, these nodes are deleted and the colour of their parent is changed from grey to the child colour. The algorithm visits every node once, since there is no other way to detect the node colour without visiting it. The cost of condensation is therefore proportional to the number of nodes in the input tree. Using the CAM capabilities of colour queries as demonstrated in Section 3.4, the new condensation algorithm is derived as follows.

Our CAM algorithm is a recursive process. Starting at the root space, the program searches the current space. We assume that all background nodes are stored in CAM. Therefore the above search will yield either one responder or multiple responders. A single responder does not need to be processed further. In the case of multiple responders, the program reads back the first responder and decodes its colour. Then the space is searched again using a `ReSearch()` operation to check if there is any octant which is not of the above decoded colour. If no responder returns after the `ReSearch` operation, then all the nodes in the current space have the same colour. They can be condensed as a single output node. These octants are deleted in one memory cycle by a `Multiple-Write` operation. A new node with the size of the current space is added to the CAM octree. Otherwise the nodes in the current space have different colours. The process is repeated for each of eight sons of the current space. The complexity of the algorithm is linear in the number of nodes of the output tree.

4.2.7 Neighbour Finding

To locate the neighbours of an octant in a CAM octree, we can make enquiry spaces such that each is one voxel thick and has a cross-section which equals the face of the seed octant. Searching the CAM octree with the above spaces, we can get 6-connected neighbours. By adding more enquiry spaces at the edges and corners of the seed octant, 26-connected neighbours can be obtained. Neighbour finding algorithms are generally complex for all three conventional octrees because they involve random traversal of the tree. In contrast, the CAM version of neighbour finding is very simple—even trivial. Again it has benefited from the trit storage, the parallel searching and pattern matching capabilities of the CAM architecture.

A closely related algorithm is connected component labelling. It can be implemented for each object in turn. We select, from an object, an octant as the seed, then give the octant a new colour as the label to the object. Then by using the neighbour finding algorithm, we can locate those octants which have the colour of the object and are neighbours to the seed. These octants are updated with a reserved colour. The next octant with the reserved colour is retrieved and becomes the seed, and the process repeats. The whole process stops when all octants with the reserved colour have been processed. The time is linear in the number of nodes in objects. The neighbour finding can also be used for boundary discovery and so on.

4.3 Conversion between Representations

As shown in Chapter 1, each representation from Breps, CSG, octrees, and so on has its advantages over other schemes for some specific applications in geometric modelling systems. In order to make use of the best scheme for each application, multiple-representations are needed in one system. Algorithms for efficient conversion between different representations are then essential for a system of multi-representations. When general octree algorithms are improved using CAM, the efficiency of conversion algorithms becomes more important for overall system performance. In some cases it can be the main bottle-neck of an octree based solid modelling system. In the remaining part of this chapter we shall concentrate on approximate conversions from various representations to raster octrees and exact conversions from boundary representations to vector octrees. The conversion from raster octrees to other representations is only possible for some special cases [Kuni85] and will be excluded.

The following sections examine algorithms for constructing octrees using CAM. The techniques for converting image models to octrees will be discussed in the next section, followed by a section of constructing octrees from object models of boundary representations.

4.3.1 Top-down and Bottom-up Approaches

An octree can be built from other representations top-down using divide-and-conquer methods, or bottom-up by node insertion. With the top-down approach, the number of leaves of an octree is determined by the surface area of the primitive. If an octree is defined in a cubic domain of $2^n \times 2^n \times 2^n$, the space complexity of the surface area is of order 2^{2n}. The bottom-up approach, on the other hand, has a computation time determined by the volume of the primitive (of order 2^{3n}). The bottom-up approach is less efficient than the top-down approach in terms of time complexity (2^{3n} vs. 2^{2n}). However, the bottom-up approach has simple tests for both convex and concave objects whereas the top-down approach suffers some difficulties when handling concave primitives. A mixed bottom-up and top-down approach is introduced later in this chapter for constructing CAM octrees from Breps.

4.4 Constructing Octrees from Image Models

Here two image models—run-lengths and orthogonal silhouettes are discussed. They are typical examples of image models. These two models have distinctive characteristics and are useful in many practical applications. Some other image models can be easily converted to run-lengths.

4.4.1 Run-Lengths

Williams [Will88b] has proposed a CAM algorithm for converting run-lengths to quadtrees. His algorithm was designed for trees where the background (white) nodes are stored in CAM in the same way as the foreground nodes. His algorithm is similar to Shaffer and Samet's conventional algorithm [Shaf87] which has been summarised in Section 2.5, except that CAM capabilities of parallel searching and updating are used for restructuring and improving the efficiency of the conventional counterparts.

Shaffer and Samet used a table to store a list of active nodes and insert a node into the quadtree when it is not a part of a larger node. Their algorithm also used an access array for speeding up node access. By using the CAM, the above complex book-keeping operations for accessing the active node table and array are eliminated. Nodes (including intermediate ones) are inserted into the CAM and can be easily updated later if further processing requires a node to be changed.

The whole space is initialised to a single background node first. Then for each run-length the following procedure is repeated. The current run-length is split into segments using the `RunList()` procedure of Section 3.4. Each segment is extended along the other axis to a maximum width and is used as an enquiry space to detect intersecting nodes. If there are multiple responders then we examine whether these responders have the same colour as the run-length. The nodes with nonmatching colours needs to be updated. If only a single node responded and it has a different colour from that of the run-length, the node is updated too. How to update a node depends on whether the size of the node is larger than the width of the segment. If it is, then the node is split into smaller nodes. Those new nodes which have the same width and overlap with the segment are assigned the colour of the run and other new nodes keep the old colour. Otherwise the colour of the node is changed.

The same principle is applicable to octrees. The above algorithm can be upgraded to octree systems easily. The simplification of the CAM algorithm over the conventional counterpart is more significant for 3D octree applications. With the conventional algorithm the size of the table and the access array, as well as the execution complexity increase tremendously when the dimension gets higher. But in the CAM algorithm 2D and 3D procedures are nearly the same except for the number of parameters of procedures. The sorting is eliminated because the order of nodes is no longer important for CAM-based algorithms.

4.4.2 Orthogonal Silhouettes

We have reviewed Chien and Aggarwal's algorithm for constructing pointer octrees from orthogonal silhouettes in Sections 2.5.4. Their method has several shortcomings as listed below:

- It needs to construct three pseudo octrees which are explained in Section 2.5.4. The process involves mapping each node of the quadtree in a silhouette view as twin-son-nodes of the pseudo octree. The process is sensitive to the numbering sequence of the pseudo octree.

- To build the octree, the algorithm intersects the above pseudo octree by traversing them in parallel. The program for intersecting pseudo octrees is complicated, involving several procedures to combine child-nodes for different tests, to check intersections of either three nodes or two nodes, and to convert from a pseudo octree to an octree.

- It does not guarantee the minimal output octree. Sometimes it generates eight sibling nodes which are all white—thus the final octree needs to be condensed.

The CAM algorithm for converting the three orthogonal silhouettes to an octree is intuitive and simple. Corresponding to the above three steps, we describe the CAM procedures here:

- **Storing silhouette views in the CAM:**

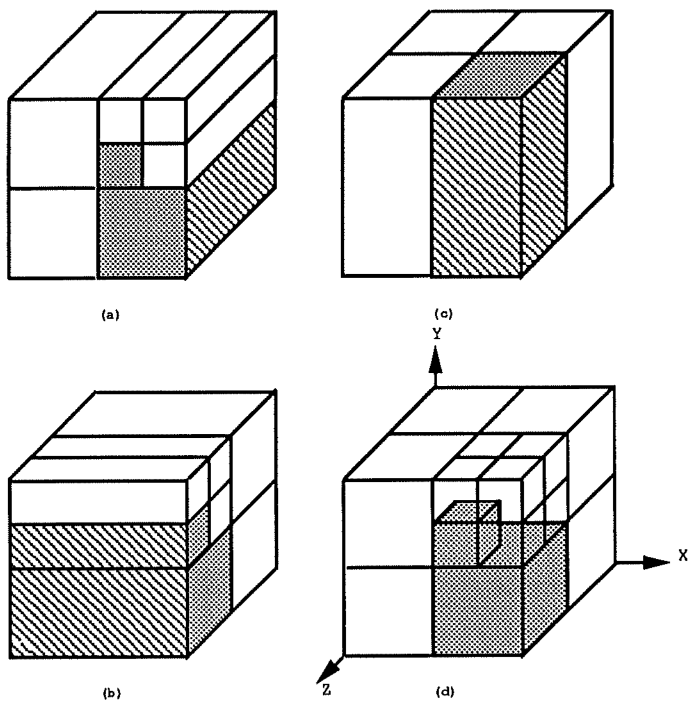


Figure 4.1: The three silhouette views ((a)(b)(c)) of an object are stored as groups of sweep volumes in the CAM, and the octree of the object (d) is obtained by intersecting these volumes.

Each of the three silhouette views can be represented as a group of rectangular parallelepipeds (RPs). Each RP is stored in CAM as a word. The length of each RP equals the maximum size of the octree space. Its cross section corresponds to the area of a node in the quadtree from one of the three silhouette views. Figure 4.1 is an example of the three silhouette views of an object: (a) (b) (c) are projected RPs and (d) is the represented object. The locational codes for these RPs for the front, top and side views are listed in Table 4.1. These codes are derived from the coordinates of the zero-corner of RPs. Once the coordinate is decided the codes will not be influenced by the numbering systems selected by different users.

	Id	*Location*		*Colour*
		x0y0z0	*x1y1z1*	
(a) Front View	001	0 1 *	* * *	0
	001	1 1 *	0 1 *	0
	001	1 1 *	1 1 *	0
	001	1 1 *	0 0 *	1
	001	1 1 *	1 0 *	0
	001	0 0 *	* * *	0
	001	1 0 *	* * *	1
(b) Side View	010	* 1 1	* 1 1	0
	010	* 1 1	* 1 0	0
	010	* 1 1	* 0 1	1
	010	* 1 1	* 0 0	0
	010	* 1 0	* * *	0
	010	* 0 1	* * *	1
	010	* 0 0	* * *	0
(c) Top View	011	0 * 0	* * *	0
	011	0 * 1	* * *	0
	011	1 * 0	* * *	0
	011	1 * 1	* * *	1
(d) Output Octree	100	1 1 1	0 0 1	1
	100	1 0 1	* * *	1

Table 4.1: CAM contents for the three orthogonal views and the output octree.

- **Intersecting sweep volumes of the three views:**

  Algorithm 4.1 demonstrates the simple CAM process of intersecting sweep volumes of the three views and constructing the octree. It is much simpler than the conventional one discussed above.

- **No condensation needed:**

  The white octants are not stored in the output octree. The final tree is minimal therefore there is no need for condensation. The complexity of the algorithm is determined by the output octree.

```
PROCEDURE build-octree(TRITS id, OCT octant)
BEGIN
  location = EncodeOct(octant)
  IF Search(ANY, location, WHITE) THEN
    IF Search(idx, location, WHITE) AND
      !Search(idx, location, BLACK)
         the octant is white
    ELSE IF Search(idy, location, WHITE) AND
           !Search(idy, location, BLACK)
         the octant is white
    ELSE IF Search(idz, location, ANY) AND
           !Search(idz, location, BLACK)
         the octant is white
    ELSE FOR i = 0 TO 7 DO
           build-octree(id, SubOctant(i, octant))
    ENDIF
  ELSE AddEntry(id, location, BLACK)
  ENDIF
END
```

Algorithm 4.1: Pseudo program for constructing an octree from the three orthogonal silhouettes of an object

4.5 Constructing Octrees from Object Models

4.5.1 Multiple-pass Top-down Construction of Octrees

This algorithm is an improvement of Meagher's interior/exterior classification method [Meag82] which has been reviewed in Section 2.6. By employing the CAM architecture, the new algorithm presented here achieves the goals of conceptual intuition, programming simplicity and execution efficiency.

4.5.1.1 Observations:

The following features are noted:

1. A convex object can be derived by intersecting half spaces which are represented by planes enclosing the object. For each plane, the object space is divided into the interior and exterior of the plane. The object can be constructed by deleting the exterior part of each plane in turn.

2. To classify an octant as interior or exterior to a plane, we need to test at most two vertices of the octant against the plane. Moreover the second vertex can be tested based on the result of the first one with only two operations.

3. Updating a CAM octree is easy. Firstly, the updating process is not sensitive to the order in which octree nodes are stored. Secondly, searching octants by

contents is as efficient as by locations, both can be down in one cycle. Thirdly, the ability to handle multiple responders leads to the efficient deletion of a group of nodes.

Based on the above observations, a new CAM algorithm is designed. The algorithm applies the divide-and-conquer method to each polygon of the object in turn, leading to a multiple-pass, top-down algorithm. It starts with an initial octree consisting of a black root node. Each pass processes one polygon face from the object to update the current octree. So the input data is not rescanned and the octant-polygon tests at each pass are confined to one polygon only. Each test examines the spatial relationship of a polygon with an octant by testing two vertices (instead of eight) of the octant against the polygon. Hence, the total number of tests and the complexity of each test are substantially reduced. The main procedure is demonstrated in Algorithm 4.2. Octree updating is implemented by efficient octant insertion/deletion in CAM and is discussed in the next part.

```
/* This procedure builds an octree from Brep for a convex object. */
PROCEDURE build-octree (TRITS id)
BEGIN
  Read in Brep, and calculate the bounding box of the object.
  Calculate the parameters A, B, C, D of each polygon.
  /* Initialise the CAM octree as the BLACK root node. */
  AddEntry(id, EncodeOct(root), BLACK);
  /* Update the octree for each polygon of the object in turn. */
  FOR each plane DO
      update-octree(id, root, plane);
  ENDFOR
END
```

Algorithm 4.2: The CAM algorithm for constructing an octree from a convex object in Brep

For simplicity, 2D diagrams (Figures 4.2) are used to show the process of octree construction for a single object. The cases for multiple objects will be discussed later. The triangle represents an object. Its edges represent polygon faces enclosing the object. The initial CAM octree is a tree with a single node which occupies the whole object space as in Figure 4.2 (a). It is updated by the procedure `update-octree()` for each boundary face of the object in turn. The bold edge in each figure represents the polygon currently being processed. The octree space subdivision at the end of each pass is shown in Figure 4.2 (b), (c) and (d).

4.5.1.2 Updating a CAM Octree

The procedure `update-octree()` takes one polygon face, then changes the current octree by subdividing the space around the plane of the polygon and deleting octants

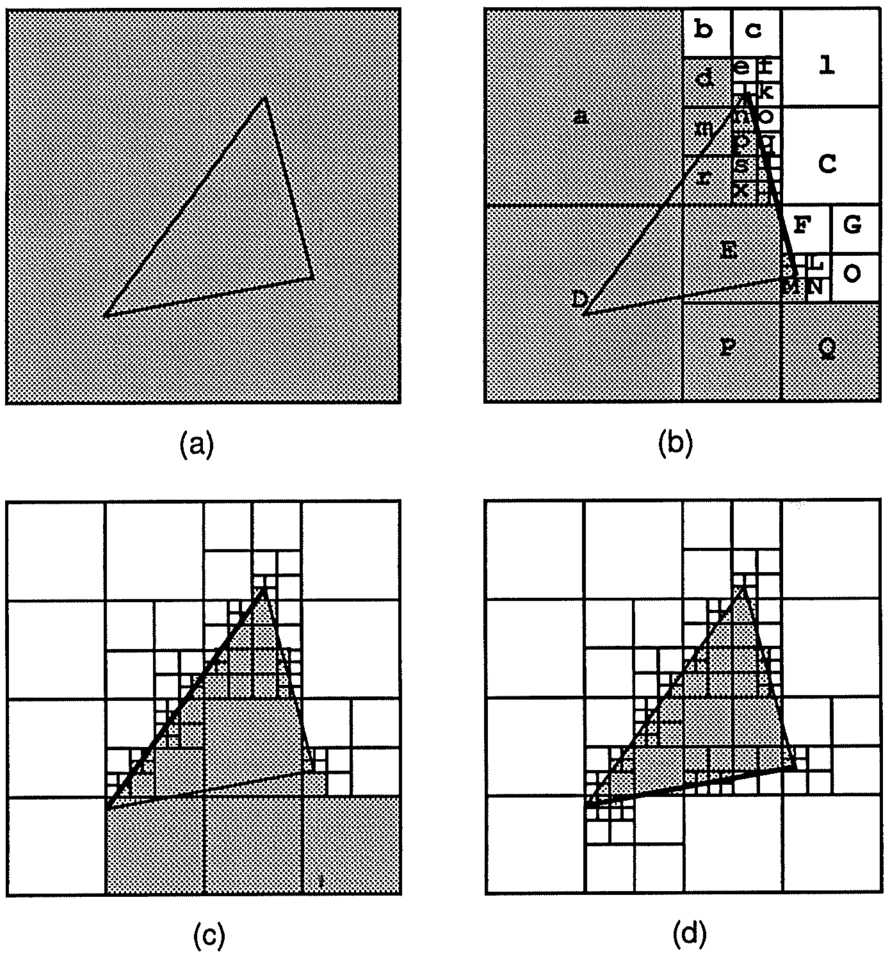


Figure 4.2: A 2D illustration of the stages of the multiple-pass top-down octree construction algorithm using CAM.

which are outside the plane. The process starts from the root space and tests the current space against the polygon face (for instance the right-facing polygon seen edge-on in Figure 4.2 (b)). There are several possible results for such a test.

- If it intersects the polygon, the space must be subdivided. For each of the eight subspaces, the CAM octree is searched for black nodes. If there is no black node in it then this subspace must have already been tested with previous polygons as outside the object and does not need to be processed further. The `update-octree()` procedure is called for those subspaces which contain black nodes.
- If the space is interior to the polygon face, no updating is required.
- If the space is exterior to the polygon face, the nodes under the space must be deleted. This is performed by searching the CAM octree using the current space, then updating the responders.

  – If there are multiple responders, all the responding nodes are deleted in one memory cycle by a `MultipleWrite()` operation.
  – If there is only a single responder, the responding node could be smaller than, equal to or larger than the enquiry space. In the first two situations the node is deleted by a `SingleWrite()` operation. Otherwise a white octant must be inserted into the larger octant using the subroutine `InsertOct()`.

```
/* Updating the current octree by processing one polygon face */
PROCEDURE update-octree(TRITS id, OCT octant, POLYGON face)
BEGIN
  SWITCH (Testing the relationship of the octant with the face)
  CASE intersect:
      IF octant.size > 1 THEN
        FOR i = 0 TO 7 DO
          suboct = SubOctant(i,octant)
          IF Search(id, EncodeOct(suboct), BLACK) THEN
              update-octree(id, suboct,facelist)
          ENDIF
        ENDFOR
      ELSE AddEntry(id, EncodeOct(octant), BLACK)
      ENDIF
      break
  CASE inside:
      break
  CASE outside:
      IF MultipleResponse() THEN
        MultipleWrite(ALL, FREE, ANY, ANY)
      ELSE
        Read(&dummy, &location, &colour)
        IF Size(location) <= octant.size THEN
          SingleWrite(ALL, FREE, ANY, ANY)
        ELSE
          InsertOct(id, location, colour, EncodeOct(octant), WHITE)
        ENDIF
      ENDIF
  END
END
```

Algorithm 4.3: The procedure for updating an octree against a polygon face

The subroutines `AddEntry()`, `EncodeOct()`, `SubOctant()`, `Size()` have been explained in Chapter 3. Their functions are as follows: add a node to a CAM octree; encode an octant into its locational address; compute a suboctant of an octant;

calculate the size of an octant from its location field. The procedure `InsertOct()` inserts a small octant into the space of a large octant by splitting the large one. The inserted part takes the new octant colour, while remaining parts are nodes with the old colour. This is demonstrated in Figure 4.3 where the node A is split into several subnodes.

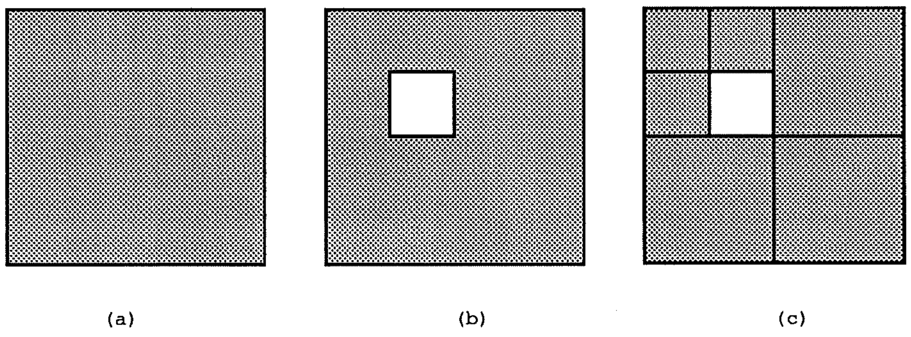


Figure 4.3: A node in (a) is split into several subnodes in (c) after inserting a small white node into a larger black node shown in (b).

4.5.1.3 Interior/Exterior Tests

An object is defined as a number of faces, each represented by its polygonal boundary. The plane in which a polygon lies is expressed by the equation [Fole90]:

$$Ax + By + Cz + D = 0$$

Where

$$\begin{aligned} A &= (y_k - y_j)(z_l - z_j) - (z_k - z_j)(y_l - y_j) \\ B &= (z_k - z_j)(x_l - x_j) - (x_k - x_j)(z_l - z_j) \\ C &= (x_k - x_j)(y_l - y_j) - (y_k - y_j)(x_l - x_j) \\ D &= -(Ax_j + By_j + Cz_j) \end{aligned}$$

Here (x_j, y_j, z_j), (x_k, y_k, z_k) and (x_l, y_l, z_l) denote any three non-collinear points on the plane. They should be chosen from vertices of the polygon in such an order that the resulting vector (A, B, C) points towards the "outside" of the object. The distance (d) of an arbitrary point (X, Y, Z) to the plane is

$$d = AX + BY + CZ + D$$

A positive (negative) distance corresponds to a point outside (inside) the plane. If $d = 0$, then the point is on the plane.

An octant is classified as interior (exterior) to a plane if all its vertices are inside (outside) the plane. If an octant is interior to all the planes which define a convex object, then the octant is inside the object. It is noted that only two out of eight vertices of an octant are important in testing the plane/octant relation. These two vertices lie on the opposite corners of an octant. Figure 4.4 illustrates the possible cases in 2D. For a 3D situation, four of two-vertex pairs cover all possible situations in which a plane and an octant can be compared.

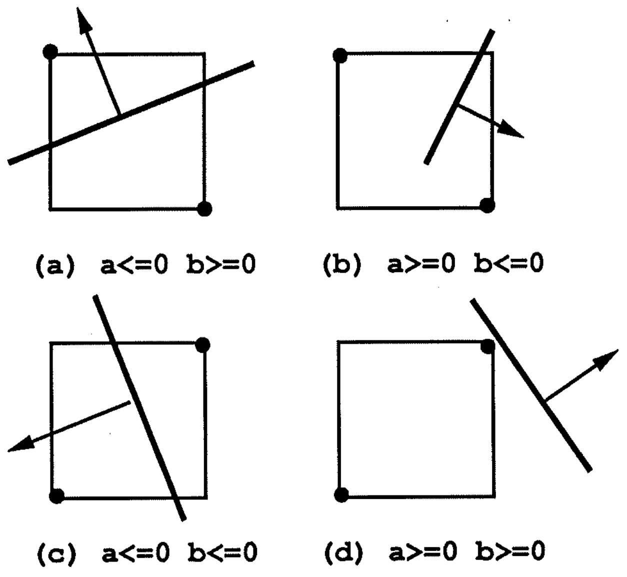


Figure 4.4: Testing two vertices to determine whether a square is inside/outside/intersecting to an arbitrary line.

The choice of two vertices is determined by the direction of the plane against which the octant is tested. The first vertex can be selected (for example the vertex $X1$, $Y1$, $Z1$ at left-bottom-back corner of an octant when $A > 0$, $B > 0$, $C > 0$; or $A < 0$, $B < 0$, $C < 0$) and examined by

$$d1 = AX1 + BY1 + CZ1 + D$$

The second vertex is $X1 + S, Y1 + S, Z1 + S$, where S is the size of the octant, and can be tested by

$$d2 = AX1 + BY1 + CZ1 + D + (A + B + C)S = d1 + ES$$

where the constant E equals the sum of A, B, C. It is fixed for each face and needs to be calculated only once. The total floating point operations for testing face-octant intersection are reduced to 4 multiplications and 4 additions.

We also relax the point-on-plane condition. Instead of using $d = 0$, $|d| < EPS$ is used, where EPS is a small constant which is determined by the voxel size and we chose that EPS equals half of the voxel size. If a polygon face intersects an octant on a corner then the octant does not need to be subdivided any further. In this way we can avoid generating many sibling nodes with the same colour. To reduce other

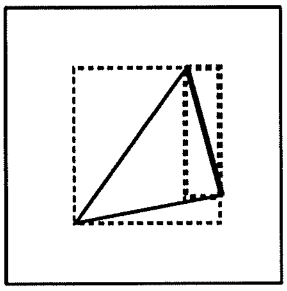


Figure 4.5: Bounding boxes for an object (the light dotted lines) and a polygon face (the heavy dotted lines)

redundant octant classifications, bounding box tests are used for the object and its polygon faces (see Figure 4.5).

The inside/outside/intersection test is summarised as follows:

1. Test whether the octant lies outside the bounding box of the object. If it does, the octant is outside;
2. Test whether two vertices of the octant are on the outside of the plane. If they are, the octant is outside;
3. Test whether the two vertices are on the inside of the plane. If they are, the octant is inside;
4. If one vertex is inside and the other is outside, test whether either of them is "on" the plane. If the "inside" vertex is on the the plane, the octant is outside (for example node F in Figure 4.2 (b)). If the "outside" vertex is on the plane, the octant is inside (for instance node E in Figure 4.2 (b)).
5. Test whether the octant is outside the bounding box of the face. If it is, the octant is considered to be inside. This test is implemented to avoid unnecessary subdivision and to keep the node for further processing in the subsequent passes (for instance nodes M and Q in Figure 4.2 (b)).
6. Otherwise, the octant intersects the plane.

4.5.1.4 Test Results and Analysis

The performance of the new algorithm is scene-dependent and was tested using two kinds of objects: a sphere approximated by 512 polygons and a cone approximated by 40 polygons. These two objects were chosen to compare our algorithm with the connectivity labelling method [Tamm84b] which is currently thought to be an efficient algorithm. In their paper, Tamminen and Samet tested a unit ball of 400 polygon faces and an object with 40 faces. The sphere and the cone are the closest examples in the data set available here. We have run examples of a sphere and several cones using different space resolutions and different object sizes. The purpose is to see how the number of polygon faces, object sizes and surface areas, and space resolutions influence the performance of the algorithm. The sphere of unit size (Figure 4.6) was tested at resolution 64. For the cone (Figure 4.6) two sizes were examined at two spatial resolutions of 64 and 128. The size of an object determines its surface area. The cone2 has one fourth of the surface area of cone1. The results are shown in Tables 4.2 and 4.3.

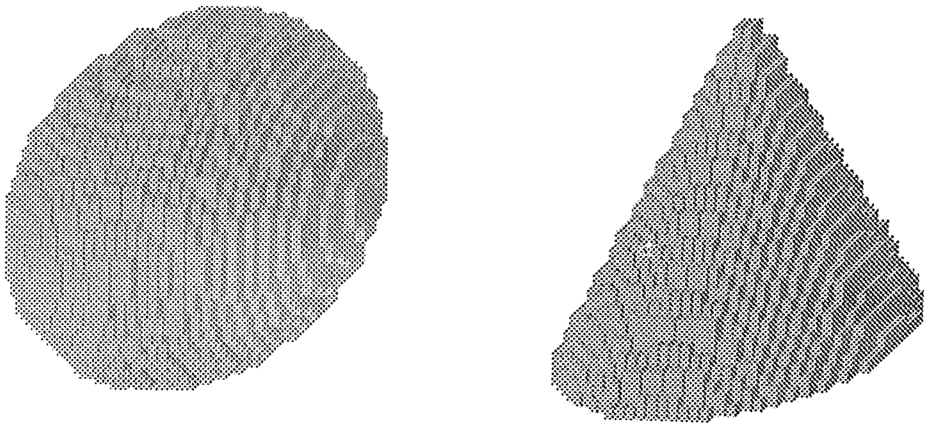


Figure 4.6: The octree images of a sphere and a cone

The algorithm has been tested with our CAM simulator on a MIPS machine. We estimated the run time by replacing the execution time of the low level subroutines which correspond to hardware functions with the CAM time estimated from the number of cycles. Table 4.2 contains simulation statistics for the hardware functions, namely the number of searches, reads, and so on. Since the time for one memory cycle is of the order of 100ns, the total CAM time is much less than one second.

	Sphere (512) Resolution 64	*Cone1 (40) Resolution 64*	*Cone1 (40) Resolution 128*	*Cone2 (40) Resolution 128*
CAM Searches	116,495	16,840	63,707	17,350
CAM Reads	13,257	3,124	12,325	3,047
CAM Writes	36,227	7,940	33,788	7,876
CAM Set-Masks	5,128	1,066	4,634	1,076
CAM Cycles	453,581	73,714	287,981	74,972

Table 4.2: CAM statistics for the multiple-pass top-down octree construction algorithm

Table 4.3 shows sizes of the output octrees and the timing statistics for various examples. We have listed both the number of black nodes and the total number of

	Sphere (512) *Resolution 64*	*Cone1 (40)* *Resolution 64*	*Cone1 (40)* *Resolution 128*	*Cone2 (40)* *Resolution 128*
Black Nodes	9,598	1,234	5,880	1,358
Total Nodes	24,769	3,521	15,153	3,681
Test Time	2.58sec	0.35sec	1.98sec	0.3sec
Total Run Time	8.76sec	0.67sec	4.97sec	0.68sec
Time Per Node	0.35msec	0.19msec	0.33msec	0.2msec

Table 4.3: The sizes of output octrees and estimated run times ($msec = 10^{-3}sec$) of the multiple-pass top-down octree construction algorithm

nodes (black, white and grey). Two timings of the new algorithm have been shown. One is the time for interior/exterior tests and the other is the estimated total run time. The total execution time of the algorithm is related to the number of nodes (black, white and grey). The memory required by this algorithm is determined by the number of black nodes of the final output octree.

From Table 4.3 we can see that the test time accounts for about one third of the total run time. The rest of the time is consumed by recursive calls of `update-octree()` and the procedure `SubOctant()` as well as other book-keeping calculations. A small amount of redundancy in calling the above subroutines is introduced by the multiple-pass top-down traversal. For example, there are repeated subdivisions (`SubOctant()`) at the root level for each polygon face, whereas there is only one subdivision at the same level for a single-pass top-down method. However, this overhead is mainly at the top levels and is small.

We did not compare our timings directly with those of the connectivity labelling algorithm proposed by Tamminen and Samet [Tamm84b]. The main difficulty for direct comparisons is that the two systems used different machines, object models and programming techniques. However, a part of their time statistics is listed in Table 4.4 to give an idea of the speed of their method. It is clear that the CAM based algorithm is much simpler and requires less memory. This makes it easier to integrate the octree construction algorithm into complex geometric modelling systems and to make long term maintenance of large solid modelling systems.

	Ball (400) *Resolution 64*	*Exc. (40)* *Resolution 64*	*Exc. (40)* *Resolution 128*
Total Nodes	NA	NA	12,331
Total Run Time	100sec	8sec	22sec

Table 4.4: The run times of the algorithm by Tamminen and Samet (on VAX11/750)

The above algorithm can be extended to multiple objects by updating the octree for each object in turn. Here we have assumed that all objects are separated from each other. In the case that all objects are mixed in one data base, it is essential to separate objects in the first place and assign each object an identifier. Polygons from different objects can be separated using connectivity properties of objects. The connectivity is expressed through definitions of edges and faces in Breps.

4.5.2 A Mixed Bottom-up and Top-down Algorithm

It has been noted previously of two trends for constructing octrees from Breps: the top-down approach and the bottom-up approach. The former is generally more efficient for convex objects and objects with a small number of polygon faces. The latter approach is mainly used for leafcodes [Atki86, Tang88] and has several distinct features:

- There is no restriction to object shapes. Concave objects and objects with holes can be processed as easily as convex objects.
- Efficient methods can be used to scan-convert polygon faces, curved surfaces and polyhedra into voxels. These scan-conversion methods often exploit space coherence of objects.
- The transformation of voxel addresses to octant addresses is straightforward.
- Octants do not need to be generated in octree pre-order as required by the top-down approach. The nodes are generated in a more or less random order which is only suitable for leafcodes.

However, there are some problems which influence the efficiency of algorithms based on the bottom-up approach:

1. The interior filling algorithm proposed by Atkinson *et al.* [Atki86] allows the space to grow inwards only. The input to the algorithm is border voxels, each is tagged as blocked or unblocked in each of its six faces. The space connected to the unblocked face is recovered and filled. The algorithm needs extra memory and pre-computation to provide block information.
2. Algorithms of [Tang88, Atki86] involve condensation at certain stages. The conventional algorithm for condensation is linear in the number of nodes in the input ($O(n^2)$ for [Atki86] and $O(n^3)$ for [Tang88]).
3. Both algorithms require sorting operations either to sort border voxels in the first phase ([Atki86]) or to sort the resulting octree at the last phase ([Tang88]).

We here design a new method that combines the features of the top-down and bottom-up approaches, and makes full use of the advantageous part of each approach. The CAM is used to improve the efficiency in several phases of the new algorithm. The mixed bottom-up and top-down algorithm is as following, with the corresponding steps illustrated in Figure 4.7 (a)-(d).

1. The bottom-up phase: It scan-converts each polygon face of the object into voxels and inserts them into the CAM. The result is a leafcode with black border voxels of the object. (Figure 4.7(a))

2. The top-down phase: It calls the procedure of background recovery (see Page 60) to get the rest of the octants of the octree space and adds them into the CAM as white nodes. (Figure 4.7(b))

3. Connected component labelling phase: It selects one octant inside the object as the seed and uses the CAM connected component labelling algorithm to change interior octants to black. (Figure 4.7(c))

4. Condensation phase: It condenses the output in Figure 4.7(c) to get the minimal tree (Figure 4.7(d)). The white nodes can be deleted if necessary.

Kaufman [Kauf87a, Kauf87b] reported two algorithms for efficient scan-conversion from 3D objects into discrete voxel-map representation. One [Kauf87a] scan-converts objects defined by planar polygons and the other [Kauf87b] converts objects with curved surfaces. Both algorithms use incremental processes which are generalised from the 2D algorithms and have only simple operations of integer additions and comparisons. Kaufman's algorithms guarantee that the border voxels from an object are linked without 6-connected tunnels which will cause internal cavities. The "lack of 6-connected tunnels" through the border insures that the interior octants are separated from the exterior ones by border voxels.

The scan-converted voxels are inserted to form a CAM octree of the black border. In CAM octrees border voxels can be inserted in any order without affecting the node search time or invoking any book keeping calculation. This allows voxels to be inserted according to the order of scan conversion which in turn is determined by individual polygon faces. This feature contributes considerablely to the execution efficiency and simplicity of the algorithm.

The complexity of the algorithm is determined by the complexity of each step. The computational complexity of Kaufman's scan conversion phase is related to the sum of the number of polygon faces of each object in the space. For each face the time complexity is $O(n \log(n) + v\ u)$, where n is the number of edges in a polygon, v and u denote the two larger sides of the box enclosing the polygon. The formula can be explained as following: the first term is the complexity for sorting edges and the second term is the count for inner loop execution. The complexity of other phases such as recovering background nodes, connected component labelling, condensation has been discussed previously (Pages 60, 66, 67). The new algorithm is applicable to objects of any shape.

4.6 Constructing Vector Octrees

We have reviewed in Section 2.7 the conventional vector octree construction algorithm proposed by Brunet and Navazo [Brun85]. Their method can be further

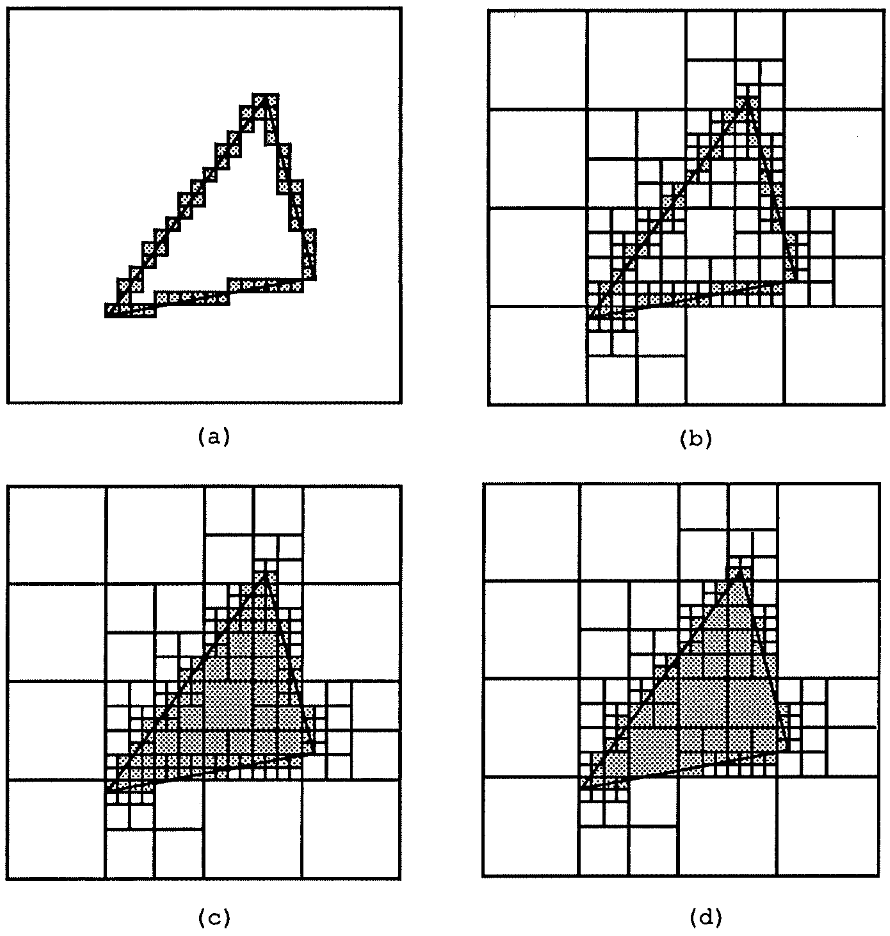


Figure 4.7: The mixed bottom-up and top-down octree construction algorithm: (a) bottom-up phase inserting boundary voxels into the CAM octree; (b) top-down phase recovering the non-boundary octants; (c) the octree after connectivity component labelling which separates internal and external octants and (d) condensing to get the minimal tree.

improved using CAM vector octrees. The structure of CAM vector octrees has been described in Section 3.7.

In Brunet and Navazo's method, polygons are clipped recursively against octants. For each octant, five different node types of a vector octree are examined during the tree construction in a sequence from simple to complex, that is black/white, face, edge and vertex. In our CAM algorithm (Algorithm 4.4), we reverse the order for examining node types and construct a CAM octree in several phases. In the first phase, some vertex nodes are examined. In the second phase face, white, face, edge and more vertex nodes are added. Finally the interior nodes and the exterior nodes are separated by using the CAM connectivity labelling algorithm.

The first phase—`build-vertex()` is simply a recursive procedure classifying each vertex point into a proper octant where it is located. The octant should be the

```
MAIN()
BEGIN
  get the list of faces
  load vertices into CAM with the id field as id3
  build-vertex(id, Root,faces)
  call the connected component labelling procedure
END
PROCEDURE build-vertex(TRITS id, OCT octant, LIST facelist)
BEGIN
  Search(id3, EncodeOct(octant), ANY)
  IF MultipleResponse() THEN
    FOR i = 0 TO 7 DO
      build-vertex(id, SubOctant(i,octant), facelist)
    ENDFOR
  ELSE
    build-face(id, octant, facelist)
  ENDIF
END
PROCEDURE build-face(TRITS id, OCT octant, LIST facelist)
BEGIN
  clip-faces(octant,facelist,facelist1)
  SWITCH (the number of faces in the facelist1)
  CASE 0: AddEntry(id,EncodeOct(octant),WHITE)
          break
  CASE 1: AddEntry(id,EncodeOct(octant),EncodeIndex('f',face))
          break
  CASE 2: IF (octant size <= 1) THEN
            AddEntry(id,EncodeOct(octant),GREY)
          ELSEIF (two faces share a common edge) THEN
            AddEntry(id,EncodeOct(octant),EncodeIndex('e',edge))
          ELSE FOR i = 0 TO 7 DO
                  build-face(id, SubOctant(i,octant), facelist1)
               ENDFOR
          ENDIF
          break
  DEFAULT:IF (octant size <= 1) THEN
            AddEntry(id,EncodeOct(octant),GREY)
          ELSEIF all faces share a common vertex THEN
              AddEntry(id,EncodeOct(octant),EncodeIndex('v',vertex))
          ELSE FOR i = 0 TO 7 DO
                  build-face(id, SubOctant(i,octant), facelist1)
               ENDFOR
          ENDIF
  END
END
```

Algorithm 4.4: The CAM algorithm for constructing vector octrees

largest possible space which contains one vertex at most. The point classification is well suited to CAM implementation. We can encode, in preprocessing, each of the vertices of objects as a voxel in a locational code format and store all these voxels in CAM with a special *id*. The *id* indicates that these nodes are different from nodes of the octree to be constructed. Starting from the root, the space is searched with the above *id* to see if any vertex lies inside. If there are multiple responders which means several vertices lying in the space, further subdivision is necessary. In such cases the process is repeated for each subspace unless the specific resolution has been reached. Otherwise there is no responder or only one responder, then the second phase starts. The second phase—`build-face()` clips the faces against the current octant, then examines the number of faces in the octant to decide the node types. This phase is similar to Brunet and Navazo's approach.

An obvious advantage of the new algorithm is that it has a simple vertex test. Unlike the conventional algorithm which tests each vertex against six bounding planes of an octant in turn, the CAM algorithm tests all vertices in one memory cycle. Another advantageous feature is that it eliminates the needs for clipping polygons at top levels during the construction process. It also uses less storage for storing intermediate face lists. However, the improvement in terms of speed may not be significant because most of the time of Brunet and Navazo's algorithm is spent on the expensive computation of non-trivial polygon clippings at the low levels of the tree. Our CAM algorithm improves performance at the top levels.

4.7 Space and Time Complexities of CAM Octree Algorithms

The memory space is determined by the number of nodes in the CAM. These nodes are usually black leaves of an octree. The time complexity of octree operations (tree traversals, locating an octant, colour searches and rotations) using CAM are listed in Table 4.5.

	Preorder Traversal	*General Traversal*	*Locating Octant*	*Colour Search*	*Rotations*
Leafcodes	O(n)	O(n log n)	O(log n)	O(n)	O(n × D)
CAM Octrees	O(n)	O(n)	O(1)	O(1)	O(D)

Table 4.5: Comparison of the time complexity of leafcodes and CAM octrees

Table 4.6 lists time complexity for set operations (union and intersection) and insertion/deletion operations. The CAM algorithms of octree union and intersection may show particular advantage over conventional algorithms when two input trees are significantly different in size or shape.

	Set Operations	*Insertion and Deletion*
Pointer Trees	O(N)	O(D)
Treecodes	O(N1+N2)	O(N)
Leafcodes	O(n1+n2)	O(log n)
CAM Octrees	O(n)	O(1))

Table 4.6: Analysis of the worst case time complexity of the three octree formats for set operations, node insertion and deletion. (N—the number of total nodes, n—the number of black nodes, 1 and 2 indicate the two input trees otherwise the output tree.)

4.8 Summary

In summarising new CAM algorithms for octree construction, we conclude that these algorithms extensively use the octant insertion operation. The insertion operation with CAM octrees has the following features:

1. It is conceptually simple and intuitive.
2. The execution efficiency for inserting an octant into an octree space is insensitive to the order of insertion.
3. An octant is inserted into a space which can be one of the following:
   (a) completely empty;
   (b) partially empty;
   (c) occupied by a larger octant;
   (d) occupied by an octant which has the same size as the octant to be inserted;
   (e) occupied by several smaller octants.

Random octant insertion is difficult in conventional octrees. It involves either tree traversals (as for pointer trees and treecodes) or intensive searches (as for leafcodes). Therefore it is always avoided when possible or optimised for special occasions. However this is no longer a problem when the CAM is employed.

The CAM octree construction algorithms and other CAM octree manipulation algorithms show the advantages of simplicity, clarity, consistency and general efficiency. These properties are very important for a large solid modelling system.

Chapter 5

Ray Tracing using CAM Octrees

5.1 Introduction

Ray tracing is a technique for creating realistic synthetic images. It is simple, elegant but computationally intensive. Octrees have been used to speed up ray tracing by partitioning an object space and sorting objects in a spatial order [Glas84]. The main problems of ray tracing acceleration with conventional octree structures concern memory management and octree traversal. The memory limits the size of an octree. Tree organisation and the node access speed influence the efficiency of tree traversals. Walking a ray through octants in a conventional octree is generally slow, involving ray-octant intersection and point location. These operations in turn affect the choice of the optimal level of space subdivisions and the algorithm efficiency. Clearly a better memory management for space subdivision techniques will help overcome these problems.

This chapter describes an application of CAM octree structures to accelerate ray tracing. A new algorithm is implemented based on the CAM architecture. The aim of our new algorithm is to simplify memory management and to speed up octree node access. The process of locating an octant for a given point is very simple with only one CAM `search` operation followed by one CAM `read` operation. To avoid ray-octant intersections, a new ray traversal algorithm named *adaptive 3D-DDA* (3D Digital Differential Analyser) is introduced. This results in efficient octree traversal and empty space skipping. Finally the problem of subdividing object space is discussed.

5.2 Background

5.2.1 The Ray Tracing Algorithm

Image rendering by ray tracing has become popular as a technique for image synthesis [Glas89]. The advantages of ray tracing lie in its simple and straightforward way of computation and the ability to display optical properties of objects in a scene. The simplicity of ray tracing algorithms leads to very little memory requirement for data structure and program code. Ray tracing algorithms have been widely used in obtaining realistic images which have been found difficult to generate with traditional scan line algorithms.

Ray tracing was originally developed for solving hidden surface problems of constructive solid geometry (CSG) objects [Gold71]. It was extended by Kay and Greenberg [Kay79] to generate images with reflection, refraction and shadows, then studied by Whitted [Whit80] and established as a powerful tool in computer graphics.

Ray tracing unifies visibility problems and lighting problems. By firing rays from an eye position, visible objects can be detected. By modelling the laws of optics, complex lighting and shadowing phenomena with multiple light sources can be handled. The latter is implemented by firing a ray at each light source, then total light energy at each point of the space can be calculated so that the shadow effect is determined. Both computation of visibility and shadows are straightforward, and mainly involve testing of ray-object intersections. Furthermore, objects with reflection and transparency can be modelled. Ray tracing has other applications such as removing hidden lines, computing volumes [Roth82], rendering penumbras and motion blur [Cook84].

5.2.2 Object Models used in Ray Tracing

Among most commonly used representations for modelling 3D solids there are CSG and Breps which have been described briefly in Chapter 1. Both representations can be mixed with a primitive-instancing scheme which has a group of basic objects (referred to as primitives): blocks, spheres, cylinders, cones, tori, and polygons. However, primitives are organised differently in CSG and Breps. In the former, solid objects are composed by combining primitives using boolean operators of union, intersection and difference. A composite object is represented as a binary tree. While in Breps, each primitive is taken as an individual object and thus processed independently.

Ray tracing is important for CSG models not only because it generates realistic images but also because it is a means to visualise objects from model definitions. The combination operators make CSG remarkably effective for designing solids but at the same time make CSG difficult to display. To display solids in the CSG model, one either converts them into Breps or uses ray casting [Roth82].

Differences between CSG and Brep lead to different ray tracing processes. Although ray-object intersection tests are basically the same for both models, ray tracing CSG is more complicated and involves point classifications against primitives. In CSG, if a ray is found intersecting several primitives of a composite object, all these intersection points (where the ray enters or leaves each primitive) must be stored and sorted by their distance from the ray origin. For primitives like tori, there could be 4 intersection points. The visible point is determined by evaluating in/out spaces from the above intersection points according to the boolean operators (union, difference and intersection) of CSG. Ray tracing Brep models needs no boolean operations. If considering objects without transparency, the visible point is the ray entry point which corresponds to the closest ray-object intersection point from the viewing position.

Besides CSG and Brep, more complicated models which contain mathematically defined surfaces such as parametric surfaces and implicit surfaces are used [Toth85, Barr86]. Ray tracing parametric surfaces are divided into two groups: one using numerical methods [Joy86] and the other by divide-and-conquer methods [Snyd87]. Methods in the first group are generally slow involving expensive evaluation of surface parameterisations. The divide-and-conquer approach subdivides a surface into simple triangles which are small enough to approximate closely the original parametric surface. This will yield a large number of polygons to be traced.

5.2.3 Previous Optimising Algorithms

The original ray tracing algorithm is very expensive in computation. The most time consuming calculation lies in testing ray-object intersections. Several techniques have been used to accelerate ray tracing by speeding up the intersection test itself or reducing the total number of intersection tests. Three approaches have been used to reduce the number of tests. The first one [Rubi80, Kay86] uses bounding volumes of objects to test simple bounding boxes or spheres before implementing complex object tests. The second group exploits spatial coherence of objects by subdividing object spaces. Algorithms using hierarchical subdivisions [Glas84, Kapl85] and non-hierarchical uniform subdivisions [Fuji86, Clea88] have been developed. The former is suitable for ray tracing of highly complex and unstructured environments. The latter is best for scenes with a large number of objects homogeneously distributed in spaces. The third group optimises ray tracing by ray classification [Arvo87] which is quite different from the bounding volume or spatial coherence approaches. Instead of preprocessing objects in spaces, ray classification algorithms dynamically explore ray coherence. Other optimising techniques include the item buffer approach for primary rays [Wegh84], the light buffer for shadow rays [Hain86], and a combination of hierarchical bounding volume with uniform space subdivision [Snyd87, Goh90].

5.2.3.1 Bounding Volumes

Bounding box algorithms [Rubi80] partition objects into groups, each group being bounded by a volume. The simplest bounding volumes are spheres or boxes. A group of objects bounded by a sphere are collected in a list. Groups of such lists are put into a higher level list. Hierarchies are thus built. There are two problems for this simple hierarchical bounding volume approach. First, bounding volumes are not tight so that many rays which hit bounding volumes miss objects. Secondly, each ray must trace top down through the object hierarchy. Organisation of the hierarchy affects the efficiency of these algorithms.

Kay and Kajiya [Kay86] used bounding volumes defined by plane-sets which enclose each object or a group of objects tightly, and introduced a special order for traversing the hierarchy of bounding volumes. Their traversal algorithm allows objects to be checked for intersection in approximately the order that each object would be encountered along the ray path. The main overhead of this algorithm is its inherent sorting.

5.2.3.2 Ray Classification

Arvo and Kirk proposed ray tracing by ray classification [Arvo87]. Their algorithm partitions rays into five-dimensional (5D) hypercubes, each containing a list of objects associated with it. Here the 5D corresponds to five degrees of freedom of a ray. They are the origin and the direction of the ray (x,y,z,u,v). Ray classification gains its efficiency by exploiting ray coherence. Since many neighbour rays tend to follow similar ray trees, a ray can benefit from classification for previous rays in a beam. Although working for all rays, the algorithm is at its best for primary rays and shadow rays, both these rays have good ray coherence.

Other optimisation techniques such as shadow cache, object sorting, back face culling for opaque solids are also used by Arvo and Kirk. Object sorting is implemented in preprocessing for the entire object database. Back face culling is done once for each beam. These techniques are most suitable for sequential processing. They will introduce algorithm complexities which are deficient for parallel processing. This shows one limitation of ray classification. The algorithm only has an advantage for sequential processing while subsequent rays can use the information obtained from processing of previous rays. When rays are processed in parallel on different processors, the coherence property is less used and overheads in ray classification become significant.

5.2.3.3 Space Subdivisions

Various space subdivision techniques for optimising ray tracing have been developed. They are hierarchical subdivisions—octrees and BSP (binary space partition) trees, and non-hierarchical uniform subdivisions. Subdivisions can be either in image spaces or in object spaces. Here we give a brief comment on subdivisions by octrees, BSP trees and uniform grids. Detailed comparisons of octree and uniform subdivisions will be discussed in Section 5.3.

- **Octrees:** Glassner [Glas84] used modified linear octrees due to the fact that horizontal walk between octants is easy in linear octrees. In addition, Glassner used a hash table and linked lists to trade-off speed and memory requirements. However, the inner loop for locating an octant for a given point is slow. It requires computing the hash name of the octant.

- **BSP trees:** Kaplan [Kapl85] used BSP trees in which each internal node has two pointers. A BSP tree classifies objects in a scene by dividing the space into half along each axis in turn. BSP trees have relatively large memory overheads. Typically, a BSP tree may contain more than 5,000 nodes including internal nodes and empty leaf nodes. But with node pointers the inner loop of node traversal may be relatively efficient.

- **Grids:** Fujimoto *et al.* [Fuji86] proposed a Spatially Enumerated Auxiliary Data Structure (SEADS) for fast ray tracing. With SEADS, the space is subdivided into a 3D array using a uniform subdivision. A ray is traced through this 3D array using a 3D-DDA (3D Digital Differential Analyser) method. Other researchers [Clea88, Aman87] use different voxel traversal algorithms for similar data structures.

Some combined methods have been proposed aiming at achieving better memory and speed trade-off, or to improve performance of space traversals. For example, Fujimoto *et al.* [Fuji86] also used the grid traversal method in octree traversal to replace ray-octant intersection tests. Snyder and Barr [Snyd87] combined list and grid structures to ray trace highly complex scenes.

5.2.3.4 Comparisons of Different Schemes

Space subdivisions have advantages in ray tracing all kinds of models including Brep and CSG defined objects [Wyvi86, Boua87], whereas the ray classification algorithm and Kay & Kajiya's efficient bounding volume algorithm are more suitable for objects in Breps and primitive-instancing schemes. Besides, the regularity of space subdivisions also makes them more suitable for some hardware implementation and parallel processing. Therefore, space subdivision algorithms remain useful.

The disadvantages of space subdivision schemes lie in their extensive memory requirements and long access time. Without memory constraints and the octant access complexity which limits the effectiveness of octree methods, octree efficient schemes could well be the way of the future. When the extra cost for octant traversal is no longer a problem, finer subdivisions of a space can be achieved. Simplification of memory management may play a key role in solving the space traversal problem. It is here where CAMs can be very useful.

5.3 Analysis of Space Subdivision Algorithms

5.3.1 Limitations of Conventional Octree Subdivisions

One problem with current octree subdivision algorithms concerns the choice of the level of space subdivision. This choice influences strongly the performance of octree based ray tracers. If subdivision is coarse, then a ray which hits nothing in a space must be checked against many objects in those octants with which the ray intersects. These checks can be reduced if the scene is subdivided finely enough. On the other hand, when subdivision is getting finer, the memory required for storing octants and time spent on walking through them increase. It is difficult to determine when subdivision is optimal to get the best performance of accelerated algorithms. This problem will be addressed further in Section 5.7. Here we explain why tree traversal is slow in a large conventional octree.

In octree and BSP tree methods, overheads of traversing an octree come from two sources. The first one is that given a point we need to locate an octant which encloses the point. The second one is ray-octant intersection to construct a point which is used to locate the next octant. Finding the point involves intersecting the ray with each of the six faces of the current octant to obtain the exit point of the ray. In cases where a ray passes through many octants before it hits any object or exits the object space, overheads in locating octants and ray-octant intersections become significantly large.

Hence there are several limitations for the octree approach:

- The efficiency of ray traversals through octants depends on the level of spatial subdivision.
- The optimal choice of the subdivision level is influenced by contents of scenes, and is difficult to determine in advance.
- For the above two reasons, the level is commonly limited to a small number, for example the maximum depth of 4. Thus the algorithm is unsuitable for a scene with a very large number of objects.

5.3.2 Limitations of Uniform Grids

Recently some studies [Aman87, Clea88] have been in favour of uniform subdivision algorithms as efficient ray tracing schemes. There are two reasons. First, voxel traversal for a uniform sized grid is very fast. For example it requires under 10 integer operations to move to the next cell [Clea88]. Second, it is a non-hierarchical structure so that locating a cell containing a ray origin can be done in constant time. However, the uniform subdivision scheme has several disadvantages. The main problems are listed below.

- It requires large memory. For example, at the resolution of 64 the number of voxels is 262,144. Most systems assume that the level of space subdivisions is small, typically with only a few hundred voxels.
- Such systems often use a fixed grid size for all scenery. The default grid size is commonly chosen as $10 \times 10 \times 10$ or $20 \times 20 \times 20$. Since scenes with different complexity may be suited to different grid resolution, a fixed grid may not always provide the optimum speed-up. It is also difficult to determine automatically what the optimal grid size is to achieve the best speed up for a particular scene.
- The scheme is sensitive to distribution of objects and to variation in the object's size. It works best when objects are uniformly distributed. The existence of a large background plane will destroy uniform distribution of objects around grid structures. This situation can make a large number of foreground objects concentrated in a few grid cells and leave most cells empty. In such a case system performance is slowed down and many empty cells need to be skipped on a ray path.

For uniform subdivisions, it is possible to reduce memory requirement by using a hashing scheme. Hashing provides some means for balancing memory requirement and time efficiency. Cleary and Wyvill [Clea88] used a hash table to store the entries of non-empty voxels. Their algorithm maintains two arrays. One is a full-sized linear array of 1-bit data to indicate whether a cell is empty or not. Assuming the cell array contains n^3 entries, the size of the linear array is n^3 bits. The other array is a smaller hash table of length M. To access a cell (i, j, k), an index (p) to the linear array is calculated using:

$p = i \times n \times j + j \times n + k;$

Then the linear array is checked. If the cell is non-empty, the hash table is consulted by computing the hash table address using the following function:

p mod M;

Therefore a small amount of extra computation is introduced in order to locate a voxel using the hash table. The choice of the table size M is important to the memory space and access speed. To consume less memory space M should be as small as possible. However, when the table is small, there is more chance for two cells to hash to the same address. There are two simple ways to handle the above problems [Sedg88]. One uses a technique named separate chaining which builds a linked list of the records for each table address thus colliding cells are chained together in the list. The other is hashing with linear probing which uses empty places in the table to solve collisions. When two records hash to the same address, then the algorithm probes the next position in the table.

It is necessary that the table length M is large enough. Large M can reduce the average length of linked lists thus sequential searches in separate chaining. With the linear probing method a large M is required to guarantee enough contiguous memory to hold all the non-empty grid cells. This means that we must be able to estimate in advance how many non-empty voxels will exist. The estimation will be difficult for arbitrary scenes. Assuming we have optimal choice of M which is just over the number of non-empty voxels, there is still a problem concerning time efficiency of probing. Sedgwick [Sedg88] showed that for a large table length M and with the table 90 percent full, linear probing will take about 50 probes for a worst case search. Therefore the hashing schemes seem unsatisfying for space and time efficiency when the number of non-empty cells is large. Unfortunately this will have to be the case for complex scenes.

5.3.3 Empirical Comparison of Space Subdivision Schemes

This section compares uniform subdivisions and octree subdivisions, then analyses advantages and problems related to them. A better data organisation is proposed to take advantages and avoid problems in these subdivisions. The proposed scheme will be explained later in Section 5.3.4. A simple 2D image with several randomly distributed objects is drawn in Figure 5.1 for analysis. We can see clearly from the figure that the proposed scheme is a variation of octree subdivisions with two main improvements. Firstly, empty cells are deleted. Secondly, each cell bounds tightly objects with which it intersects. All three subdivisions, the uniform, octree and proposed schemes, are at the same resolution (depth 3 for this example). In the uniform subdivision scheme there are 64 cells and in the octree scheme there are only 16 octants. The proposed scheme has the least number of nodes which is 9.

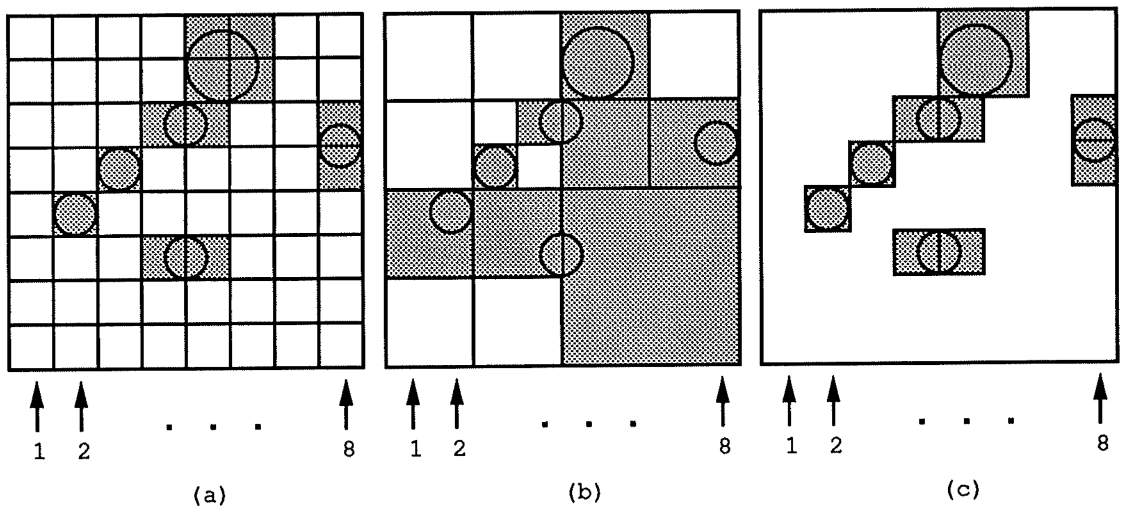


Figure 5.1: Comparison of different space subdivision schemes: (a) uniform subdivision; (b) octree subdivision; (c) proposed scheme

Table 5.1 shows the number of voxels (or octants) visited and the number of ray-object intersection tests involved for each sample ray in the scene of Figure 5.1. A uniform subdivision has a small number of ray-object intersection tests (7) and a large number of voxel traversals (48). An octree subdivision has a large number of ray-object intersection tests (13) and a relatively small number of octant traversals (23). However, to locate an octant for a given point in a conventional octree is more expensive than to locate a voxel in a uniform subdivision scheme. Now it is easy to see why ray tracers based on uniform subdivisions are generally faster than those based on octree subdivisions. The main reason is that with an octree accelerated ray tracer, the algorithm is slowed down not only by expensive octant traversals but also by the remaining large number of ray-object intersection tests caused by loose bounding octants. An octant bounds objects loosely when objects concentrate on a small part of it. Therefore, many rays that hit the octant miss objects in it thus tests are wasted. For example the ray number 7 in Figure 5.1 (b). Even if we can improve the efficiency of octant traversals, the number of ray-object intersection tests remains high for octree subdivisions. Cells in a uniform subdivision have tighter bounds to objects than those of octants in an octree subdivision.

5.3.4 Proposal of a New Scheme

The scheme proposed here (named a CAM octree ray tracer), is shown in Figure 5.1 (c). It combines several advantageous features of other subdivision methods. These features are the tight voxels of uniform subdivisions and the small memory of octrees. Memory saving is achieved by having various sized cells and ignoring empty cells. Table 5.1 shows that with the new scheme, the number of visited voxels is re-

	Uniform		*Octree*		*Proposed*	
Ray No.	*Voxels Visited*	*Inters. Tests*	*Voxels Visited*	*Inters. Tests*	*Voxels Visited*	*Inters. Tests*
1	8	0	4	1	0	0
2	4	1	2	1	1	1
3	5	1	3	2	1	1
4	8	2	5	2	2	2
5	3	1	1	1	1	1
6	7	1	3	3	1	1
7	8	0	3	2	0	0
8	5	1	2	1	1	1
Total	48	7	23	13	7	7

Table 5.1: Analysis of space subdivision algorithms used for accelerating ray tracing

duced to 7. It is far smaller compared to the corresponding numbers in the uniform subdivision and octree subdivision schemes. The number of ray-object intersections is the same as that of uniform subdivision. Therefore, the new scheme can be faster due to its small number of voxel traversals and ray-object intersection tests. For a high resolution subdivision (for instance depth 6), reduction in the number of voxel traversals with the new scheme will become more significant.

However, the question now is how can a ray quickly traverse the space shown in Figure 5.1 (c)? How can we decide the next voxel for a ray to visit? With conventional architectures, traversing such a space organisation may well be a complex task. Peng *et al.* [Peng87] have tried to use conventional linear octrees to store a similar spatial organisation. In order to find the next octant, they employ binary searches. They also tried to skip empty regions more efficiently using a heuristic method which finds the ray exit point for a given octant. The procedures of binary searches and ray-octant intersection tests remain the main sources of overheads. The two questions above can not be answered with simple yet efficient solutions using conventional architectures. We shall show how they can be resolved with CAMs.

5.4 Ray Tracing using CAM Octrees

As mentioned earlier, algorithms of ray tracing by octrees use up much CPU time for data structure traversal to find the next octant. It will be helpful to have some hardware assistance for time consuming octree traversal. This was our motivation to employ CAM which is very efficient for searching random cells in an octree. By proposing an algorithm using CAM octrees we attempt to overcome problems encounted in conventional octree-accelerated ray tracers and to improve their performance.

5.4.1 The Aims

Our aims are:

- simplifying memory management;
- speeding up the octree node access process by
    1. using CAM searches to locate an octant for a given point;
    2. removing ray-octant intersections.

In justifying our new technique as a good optimising algorithm or architecture, the following criteria should be met:

- low computation overheads for the new data structure;
- low memory overheads imposed by the specific data structure;
- dynamic optimisation for general scenery;
- simplicity (easy for maintenance).

We have noted in the previous section that the first source of overheads for octree traversal lies in locating an octant for a given point. This overhead may be readily overcome by employing a CAM architecture with which only one search operation is needed. The point location is simpler and more efficient than Glassner's one which computes the integer name associated with the octant. Another bottleneck of ray tracing with Glassner's octrees lies in the calculation of a point to be located. Therefore, the new algorithm must also avoid ray-octant intersections. This can be resolved by introducing an *adaptive 3D-DDA* technique in cooperation with CAM. Here we first give an overview of the CAM octree ray tracer. Details of *adaptive 3D-DDA* and CAM operations will be discussed later in the chapter.

5.4.2 The CAM Octree Ray Tracer: an Overview

The algorithm for speeding up ray tracing using CAM octrees is as follows. The object space is subdivided into an octree, and the non-empty octants are stored in the CAM. Each octant has its location, size and an index to a group of objects associated with it. Traversal of a ray through the octree space is implemented using an *adaptive 3D-DDA* algorithm incorporating some CAM operations such as `RunList()` and `Search()` (see Section 3.4). Instead of skipping voxel by voxel as in Fujimoto's 3D-DDA algorithm and in other uniform subdivision methods, the *adaptive 3D-DDA* method breaks a ray into several segments, each with a length as long as possible. The adaptive approach allows quicker empty space skipping especially when the ray is nearly parallel to any of the three coordinate axes in the object space. A ray segment is in turn split into one or more fragments by the CAM procedure `RunList()`, each fragment fits into the octree space. By comparing the locational codes of these fragments with the contents of the CAM, we obtain the octants with which the ray intersects. Each comparison requires only one search. After finding the closest intersected octant, objects associated with it are tested against the ray for intersection. The algorithm of ray-CAMoctree is listed below in pseudo C.

```
PROCEDURE Ray-CAMoctree(RAY ray)
BEGIN
  initialise adaptive 3D-DDA parameters according to the ray origin
  and direction;
  REPEAT
    calculate a ray segment based on adaptive 3D-DDA;
    compare the ray segment with CAM contents and get a list of ordered
    octants pierced by the ray segment;
    IF the list is not empty DO
      REPEAT
        get an octant from the list;
        FOR each object associated with the octant DO
          Intersect(ray, object);
        ENDFOR
      UNTIL no octant left in the list OR intersection is found;
  UNTIL intersection is found OR the ray exits the bounding box of space;
END
```

Algorithm 5.1: The pseudo-C code for intersecting a ray with the CAM octree model

Clearly the main parts of the CAM octree efficiency scheme are to build the CAM octree data structure and to trace a ray through it. The data structure and construction of CAM octrees are shown below, followed by a traversing technique which includes two parts: *adaptive 3D-DDA* and CAM operations.

5.4.3 Organisation of the Program

The basic ray tracer follows the software design described in Chapter 7 of the book: "An Introduction to Ray Tracing" [Glas89]. We chose this design for its clarity and generality. It is a standard method and is machine independent. The software, written in C, is in a modular fashion to make module interface simple and clear. The design adopted the object oriented programming philosophy with which files are organised by data structures of objects instead of by procedures. The aim is to make the code maintainable and extensible thus to make long term evolution of the project easier and more reliable. Acceleration techniques can be fitted into modules without major change in the system architecture of the basic ray tracer.

The features of the basic ray tracer include,

- Extensible multiple primitive types;
- Arbitrary level of diffuse reflection, spectral reflection, transmission;

- Multiple light sources;
- Simple operations on primitives.

For testing our CAM octree accelerating algorithm, we only consider a simple rendering model without texture, anti-aliasing and other things like complex ray distribution. The system contains an array of objects and each array element (an object) has several fields. The structure of an object is as follows:

```
structure object {
  unsigned short    object_type;
  double            bounding_box[3][2];
  char              *primitive;
  struct PrimProcs  *procs;
  struct Surf       *surf;
}
```

Each object has a type which is indicated in the `object_type` field. An object can be a primitive object or a composite object. Composite objects are used in accelerating techniques such as hierarchical bounding structures or space subdivisions, and so on. In the CAM octree algorithm introduced in this chapter only one composite object is used, that is the root space of the CAM octree. Primitive objects are polygons, spheres, cylinders, and the like. Each object is bounded by a simple `bounding_box` formed by three pairs of extents in x, y, and z directions. The `primitive` field points to a memory address where the geometric and other useful information of the object is stored. This field is type dependent and must be cast to different structures according to the type of a specific primitive. By using type cast, the overall structure of the program is made simple and clear. Primitive-dependent information is hidden in a data structure local to the file for each primitive group. New object types can also be added easily without changing the program structure. The pointer `procs` points to functions that perform known operations on object structures. These functions are:

- *Read:* Reading a primitive object from a data file;
- *Intersect:* Finding the intersecting point of a ray with an object;
- *Normal:* Calculating the normal vector at a surface point of an object;
- *Transform:* Translating or scaling an object or a ray;
- *Primitive-octant classification:* Classifying objects into octants.

The `surf` field points to the location where we store information about surface properties which include colour, reflectivity and transparency. Surfaces of a kind have a unique name and the name is defined before it is used. One surface can be associated with many primitive objects.

5.4.3.1 Indexing Objects in a CAM Octree

With our accelerating algorithm, objects (polygons, spheres, and so on) in a scene are organised using a CAM octree. Each octant in the CAM octree contains a group of objects. Organisation of groups is shown in Figure 5.2. CAM contents contain octants' locations and indices pointing to memory addresses on a child list. Only non-empty octants are stored. The child list is an array which stores groups of pointers to objects. Each group associates with a specific octant in the octree space. A group of primitives which intersect the same octant are indexed sequentially on the child list. The groups associated with different octants are separated by -1 in the list. The addresses indexed in CAM are the locations of the first child in each group. When an octant is found intersecting a ray, the first primitive in its associated group is located and tested against the ray. Other primitives in the same group are visited in turn until the number -1 is reached.

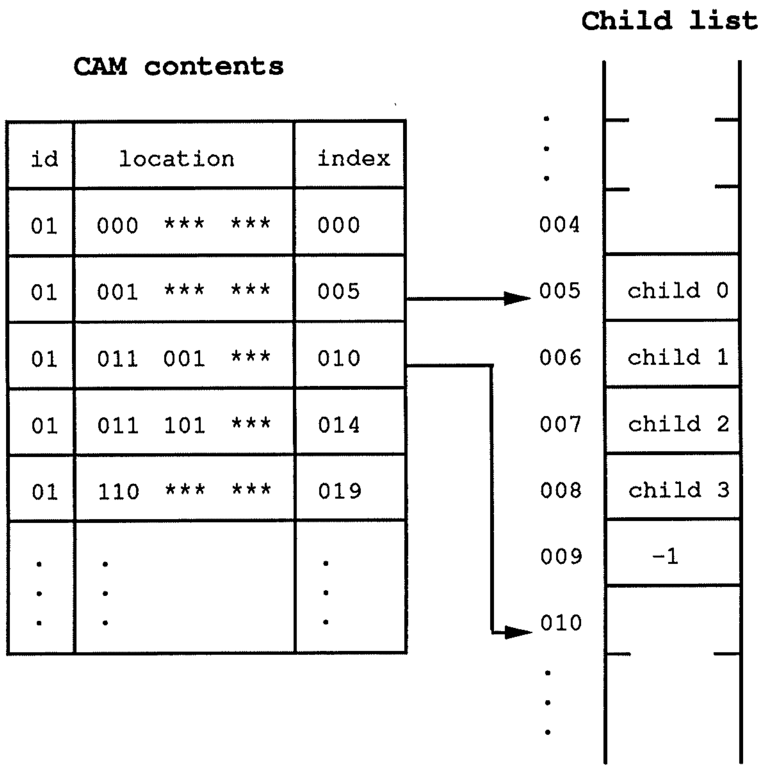


Figure 5.2: Indexing objects in a CAM octree: CAM contents and the child list.

A CAM octree is constructed in preprocessing by recursively subdividing the space into eight subspaces (octants) until the number of objects in an octant is less than some criteria (for example six objects) and they are bounded tightly by the octant, or the octant reaches the smallest size. The tightness of an octant around its associated objects is tested by further subdividing the octant and examining whether there are more than four empty suboctants.

The procedure `Sphere-Octant()` given below is an example of space subdivisions for primitives of spheres. It must be noted that only surfaces of spheres are important for the ray-sphere intersection test. Therefore, any octant which is completely inside or completely outside the testing sphere is considered as irrelevant to that sphere. Similarly, polygons can be classified with Sutherland-Hodgman's 3D window clipping algorithm. More discussion about subdivisions on polygonal object models will be given later.

```
PROCEDURE Sphere-Octant(SPHERE sphere, OCT octant)
BEGIN
  check the number of octant's vertices that lie outside the sphere
  IF the number is 0
    THEN the octant is completely in the sphere
  ELSE IF the number is 1 -- 7
    THEN add the sphere to the list associated with the octant
  ELSE /* all eight vertices lie outside the sphere */
    IF the octant is outside the bounding box of the sphere
      THEN the octant is completely outside the sphere
    ELSE IF the octant crosses any of three lines, each passes through
            sphere center and is parallel to one of three axes
      THEN add the sphere to the list associated with the octant
    ELSE the octant is outside the sphere
    ENDIF
  ENDIF
END
```

Algorithm 5.2: The procedure for classifying a sphere as inside/outside/intersection to an octant

Since our CAM octree is in locational code format, we must assume the maximum tree depth in advance. Considering that the current CAM simulator has 32-trits only, we set the maximum tree depth as 6. Therefore, the root of an octree is a cube with a size of 64. To make calculations of *adaptive 3D-DDA* simpler, all objects are scaled in preprocessing to a cubic space corresponding to the octree root. The eye position and lights' positions are scaled in the same way as well.

5.4.4 Adaptive Three Dimensional Digital Differential Analyser

Fujimoto *et al.* [Fuji86] were the first to use a 3D digital differential analyser (3D-DDA) for space traversal. The 3D-DDA algorithm is a 3D extension and modification of DDA which evaluates some functions incrementally. The DDA is commonly used in line or surface drawing on displays and plotters. Fujimoto and his colleagues also use 3D-DDA to traverse an octree structure, but results show that uniform grids are better than octrees in general. Figure 5.3 is an example of a 3D-DDA corresponding to a ray.

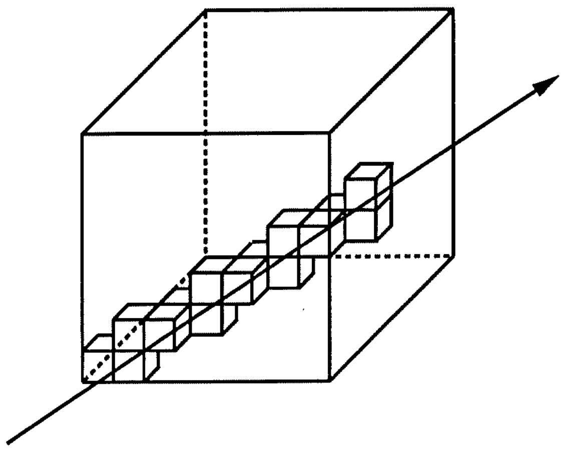


Figure 5.3: A ray and its corresponding 3D-DDA

Our *adaptive 3D-DDA* is similar to Fujimoto's 3D-DDA except that the adaptive algorithm examines a group of voxels in a single step. A ray is split into a series of segments, each segment has a cross section which is the unit square of a voxel (at the lowest level of the octree subdivision). The position and the length of the segment is calculated from the parameters which are determined by the ray origin and direction. Figure 5.4 gives two dimensional illustrations of the ray segments of our *adaptive 3D-DDA* in several typical cases. Since some ray segments do not fit the constraint of the octree space subdivision, they are further broken into fragments. These fragments (separated by dotted lines) can be seen in Figure 5.4 (b) (c). Related with Figure 5.4, Table 5.2 compares the number of voxels visited by the 3D-DDA method and the number of fragments examined for the same ray in our *adaptive 3D-DDA* method. The advantageous features of *adaptive 3D-DDA* are clearly shown in the results. In the best case, the ray can be tested against the object space in one step with a single segment. In general situations, the number of fragments is about one third of the number of voxels visited by the ray. Even in the worst case the number of fragments is less than two thirds of the number of voxels.

	3D-DDA	*Adaptive 3D-DDA*
(a) Best Case	8	1
(b) General Case	10	3
(c) Worst Case	14	8
Total	32	12

Table 5.2: Comparisons of traversal steps for 3D-DDA and for *adaptive 3D-DDA*.

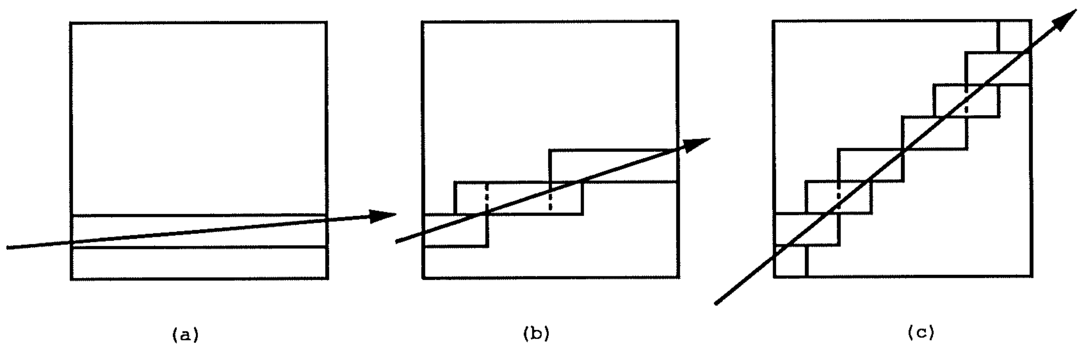


Figure 5.4: Two dimensional illustrations of adaptive 3D-DDA for rays of different orientations: (a) the best situation where the ray is nearly parallel to one axis (for instance the horizontal axis); (b) general situation; (c) the worst case where the ray enters from one corner of the world space and leaves at the opposite corner.

For each ray, the parameters required for calculating ray segments of *adaptive 3D-DDA* are shown in Figure 5.5 and constructed as follows,

1. Finding the coordinates of two points (pl, p2) where a given ray enters and exits the object space (for reflecting/refracting rays pl lies inside the object space);

2. Determining the main driving axis (MDA) which has the largest movement from entry point to exit point, the second driving axis (SDA), and the passive axis (PA);

3. Initialising the increments (sl, s2) and the control terms (cl, c2) in MDA, while the increments in PA and SDA are the unit voxel size.

Without losing generality, we choose the x-axis as MDA, y-axis as SDA and z-axis as PA to demonstrate the algorithm. The parameters sl, s2, cl and c2 are floating point data.

$$s1 = \frac{dx}{dz}; s2 = \frac{dx}{dy}$$

The control terms cl and c2 are coordinates in MDA of the points where the ray passes from one segment to the next. The parameters cl and c2 are updated by increments of sl and s2 respectively at some stages.

Calculation of each ray segment is implemented by simple comparison, addition and subtraction. Each ray segment is a parallelepiped which has a position indicated by coordinates sx, sy, sz, and a length sl. The position of a ray segment is determined by its left-bottom-back corner. Its width and height are unit size and are ignored in the data structure. The parameter sx is an integer truncated from floating point values cl or c2. The parameters sy and sz are initialised with the coordinates of the ray entry point and are increased by unit voxel size during the traversal. Each ray

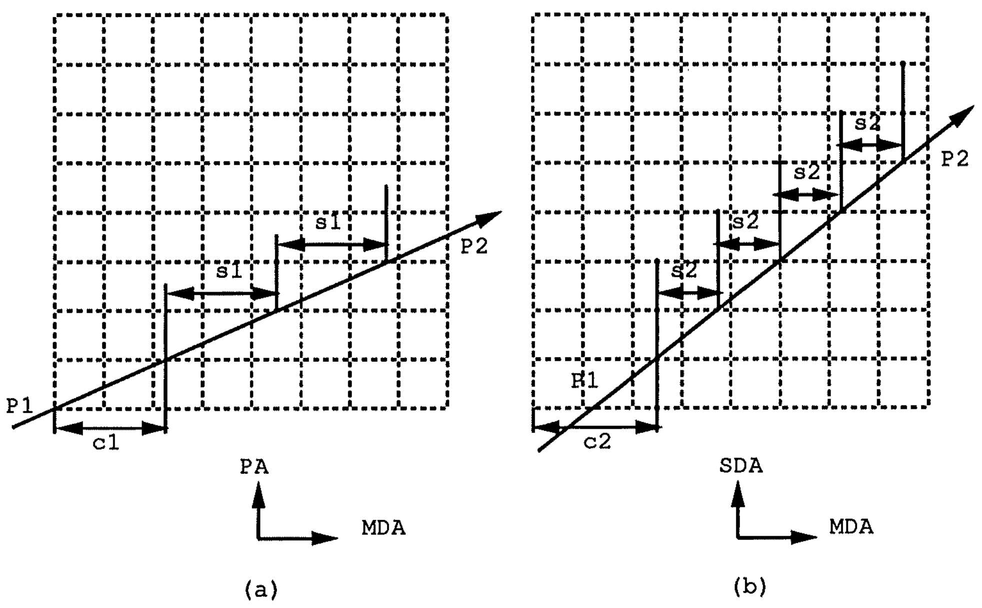


Figure 5.5: The parameters for calculating adaptive 3D-DDA: the increments (s1, s2) and the initial control terms (c1, c2)

segment is updated from the previous segment. The first segment is calculated from the ray origin if the ray is inside of the boundary of the scene space, or from the point where the given ray enters the scene space. The pseudo C code of the *adaptive 3D-DDA* algorithm is listed in Algorithm 5.3.

```
/* sx, sy, sz, sl are the coordinates and length of the current segment */
/* [x] is an integer truncated from the floating point number x */

initialise values of p1x, p1y, p1z, p2x, p2y, p2z, s1, s2, c1, c2
for current ray;
sx = [p1x]; sy = [p1y]; sz = [p1z];
WHILE c1 < p2x DO
  WHILE c2 < c1 DO
    sl = [c2] - sx + 1;
    IF intersect any object THEN exit;
    sx = [c2];
    c2 = c2 + s2;
    sy = sy + 1;
  ENDWHILE
  sl = [c1] - sx + 1;
  IF intersect any object THEN exit;
  sz = sz + 1;
  IF c2 < p2x DO
    sx = [c1];
    sl = [c2] - sx + 1;
    IF intersect any object THEN exit;
    sx = [c2];
    c2 = c2 + s2;
    sy = sy + 1;
  ENDIF
  c1 = c1 + s1;
ENDWHILE
WHILE c2 < p2x DO
  sl = [c2] - sx + 1;
  IF intersect any object THEN exit;
  c2 = c2 + s2;
  sy = sy + 1;
  sx = [c2];
  IF intersect any object THEN exit;
ENDWHILE
sl = [p2x] - sx + 1;
IF intersect any object THEN exit;
```

Algorithm 5.3: The procedure for computing ray segments of adaptive 3D-DDA

The tests in the program for intersecting a ray with objects are actually procedure calls. The procedure involves CAM operations and standard ray-object intersection tests. The number of operations of *adaptive 3D-DDA* for computing each ray segment is analysed here by counting instructions between two calls of the testing procedure. The results show that the *adaptive 3D-DDA* method has, on average, two floating point comparisons, two assignments from a floating point to an integer, three integer additions and one integer subtraction. The total number of operations is 8.

5.4.5 Operations of the CAM

CAM hardware acts as a peripheral device of a host computer. Its task is to find, in the right order, the octants which are non-empty spaces and are intersected with the ray currently being traced. To compare a ray segment with a CAM octree, a search pattern must be constructed whose locational code comes from the parallelepiped corresponding to the ray segment. Then a CAM `search()` operation is called. However the ray segment computed from the *adaptive 3D-DDA* method does not always fit into octree subdivision (see Figure 5.4 (b) and (c)). It is therefore necessary to check and break each unfit ray segment further into a group of fitted fragments, then to compare each fragment with the CAM octree. Segment splitting is implemented in the CAM high level interface with the subroutine `RunList()` shown in Section 3.4.

Since all the nodes in CAM are non-empty octants, the enquiry may yield zero, one or many responders. This means that the ray intersects zero, one or many octants respectively. These cases are considered in turn. A simple example is shown in Figure 5.6.

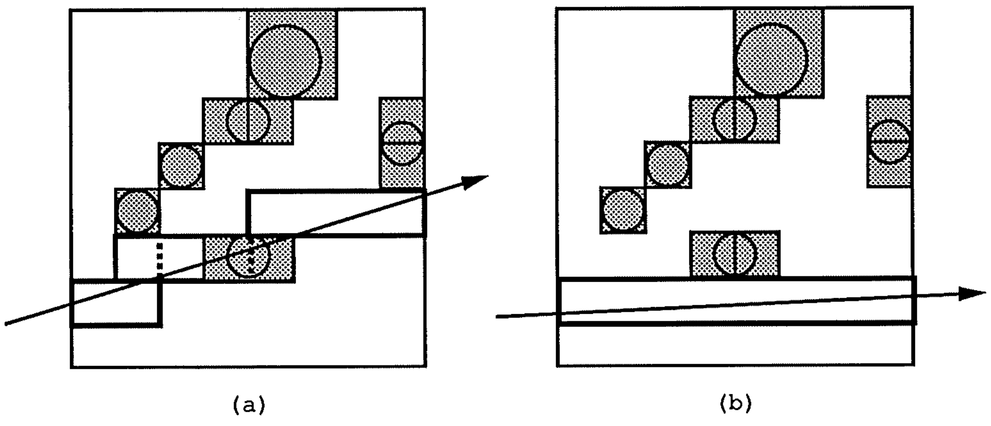


Figure 5.6: Locating the next octant

1. If a search returns no responders then the ray fragment hits nothing in space. The program continues for the next ray fragment.

2. If a search returns one responder, the ray fragment intersects one octant. This octant could either be the same as the octant hit by the previous fragment or be a new octant which has not been tested before. In the first case, the program jumps to the next fragment and continues. Otherwise, each object in the octant is tested in turn against the ray.

3. If there are multiple responders, the current ray fragment must be divided into two parts of equal size. The above process is repeated for each half in turn. This process is a recursive refinement of a ray fragment. Its purpose is to find the intersected octants in the right order.

If ordered retrieval CAM, which will be introduced later in Chapter 6, is available, refinements in the third case above can be removed. In this situation, multiple responders are read out sequentially in a sorted order. A ray must visit a group of octants in a specific order from the closest octant to the farthest one with respect to the ray's origin. Therefore, the program must detect the order required by each specific ray. The order can be either increasing or decreasing along the MDA. These and some related problems will be discussed further in Chapter 6.

5.5 Performance and Comparisons

5.5.1 The Influencing Factors

Several experiments were carried out to test the performance of our CAM octree ray tracer. This section presents results, compares them with the performance of the naive ray tracer which tests all rays against all objects and the performance of Glassner's octree based ray tracer. It should be noted that the comparison between algorithms is very subjective. There are several factors that may strongly influence the comparison of different accelerating algorithms. Firstly, the published timing data are based on different machines which have different architectures, speeds and floating point co-processors. Secondly, the language used in each system may be different. Although most ray tracers are implemented in C, some of them use assembler program to improve execution of heavily used procedures. There also exist systems in FORTRAN 77 [Fuji86] and in OCCAM running on transputers [Goh90]. Thirdly, the code organisation is also important for the execution efficiency. It depends on whether the data structure is a linked list or an array, whether the program is object oriented or is optimised by removing most procedure calls. Finally, comes the test scene complexity. As we have discussed in Section 5.3 different models tend to work best for different scenes. Thus it would be more sensible to compare the features of each algorithm than simply to compare the execution times.

One way to do the comparison is to normalise the average speed (CPU time) of each machine to Glassner's VAX 11/780 which has a floating point accelerator of 250 Kflops. Machine speed normalisation is unreliable in the sense that it depends upon what sort of operations are used in a particular code. Another method to compare different algorithms is to examine the speed-up of each accelerated algorithm to the corresponding naive ray-tracing algorithm. We adopt the latter approach in our analysis. Considering the above factors we think that more reliable comparisons of different efficient schemes should be derived as following. All efficient schemes should be based on the same naive ray tracer which uses a standard code and program structure. The speed up of each technique should be normalised to that of the naive ray tracer running on the same machine. Assuming that the tracing time of a naive ray tracer is 1.0, the relative performance (or speed-up) of accelerated ray tracers to the naive ray tracer is

$$SpeedUp = \frac{Time\ of\ Naive\ Ray\ Tracer}{Time\ of\ Accelerated\ Ray\ Tracer}$$

The higher this number is, the faster the accelerating technique is.

Since our experiments were carried out with the CAM simulator, most of the CPU time was spent on those parts that simulate CAM functions. The simulator is far slower than real hardware. On the other hand, our current simulator has only 32 trits. We can only test up to a depth of six of space subdivisions and index a maximum of 4096 groups of objects. Therefore, it is difficult to test a wide range of scenes using the simulator. Only some representative scenes are selected. We estimate the execution time of the new algorithm by profiling the program and subtracting the time which is known to be the part corresponding to hardware functions. The current CAM simulator and the CAM octree ray tracer involve many procedure calls in order to keep the design clean. The program can certainly be further optimised to improve execution speed. The time shown in this section represents the lower bound of the speed of our new CAM based algorithm.

5.5.2 The Scene Database

Haine [Hain87] proposed a Standard Procedure Database (SPD) package for testing rendering algorithms. In that database, he selected six scenes which are familiar to many graphics researchers and users. These scenes are generated by simple programs. Primitive objects in SPD are regular polygons, polygonal patches, spheres, cylinders, and cones. The number of objects in a scene can be adjusted directly in the program. Among six scenes, the recursive tetrahedral pyramid and the sphereflake are most widely cited. They are shown in the ray traced images of Figure 5.7 and Figure 5.8. The pyramid scene was introduced by Glassner, and then adopted by Kay and Kajiya [Kay86], as well as by Arvo and Kirk [Arvo87]. The sphereflake scene was used by others [Gree89, Prio88].

Here, we choose the two scenes of pyramid and sphereflake to test how much improvement can be achieved by using CAM octree data structures. In the pyramid example, the eye rays are not parallel to any of the three axes of the object space. This example is used to measure the algorithm for arbitrarily oriented rays. In this case, the eye rays are entering the object space from one corner and leaving from its opposite corner. It is nearly the worst case for *adaptive 3D-DDA*. In sphereflake scenes parallel eye rays were used to test the best case performance of *adaptive 3D-DDA*.

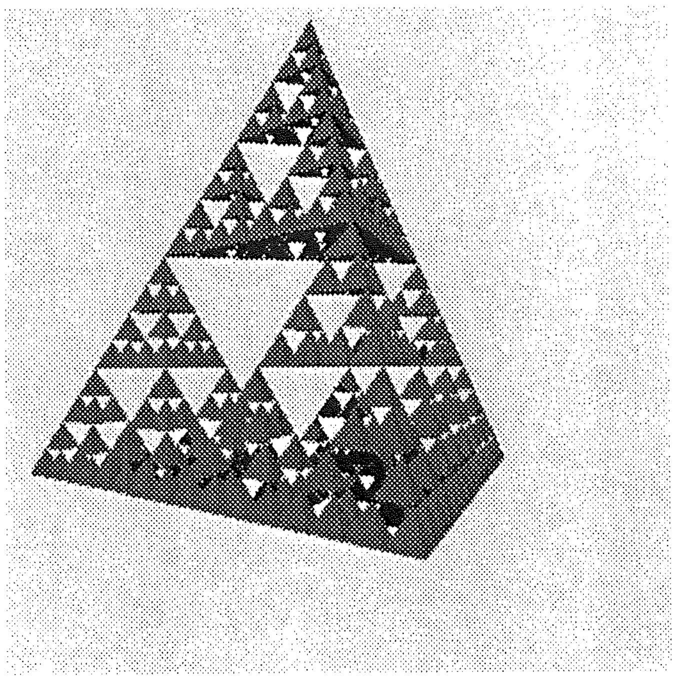


Figure 5.7: The image of the recursive tetrahedral pyramid

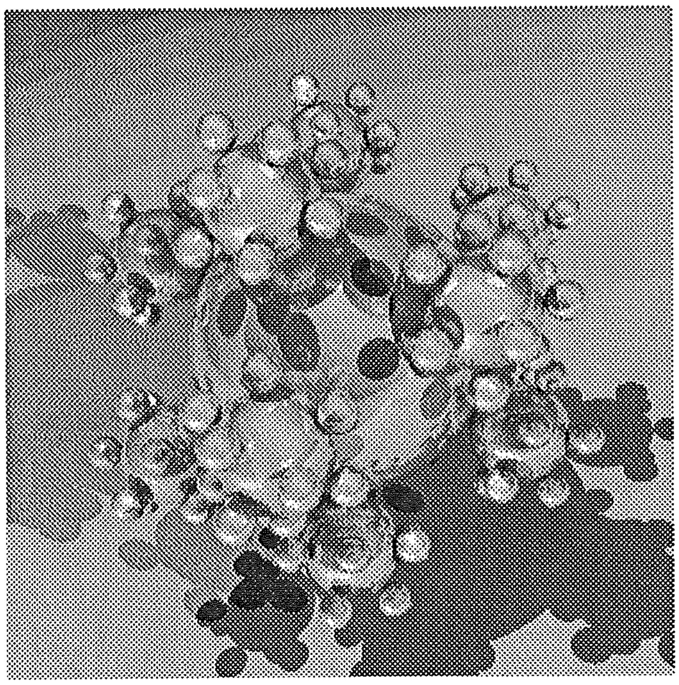


Figure 5.8: The image of the sphere-flake

5.5.3 Statistics

Table 5.3 gives the space and time comparisons of our new CAM octree ray tracer with the naive ray tracer and Glassner's ray tracers for the pyramid scene. The scene has 1024 polygons with an image resolution of 512 $\times$ 512. Only one light source is used and no reflection or refraction is considered. The viewpoints in our database are slightly different from those in Glassner's image. The number of octants stored in CAM is nearly half of that of Glassner's octree. The number of objects per octant is 4 in the CAM algorithm. The average number of intersections per ray is reduced tremendously without increasing very much the time for traversing rays through octants. From the statistics in Table 5.3 we can see that the speed of the CAM octree based algorithm is about 20 times as fast as Glassner's algorithm for this example.

	Resolution of Eye Rays	*No. of Octants*	*Aver. Inters. Per Ray*	*Execution Time (Sec.)*	*Speed Up*
Naive	(512,512)	1	1024	12,517	1.0
Glassner	(512,512)	473	25.6	NA	7.3
CAM	(512,512)	256	1.35	82	152.0

Table 5.3: Statistics of naive, Glassner's and CAM octree ray tracers for the tetrahedral pyramid scene

In the tetrahedral pyramid database, objects are in same size and are distributed regularly in space. Such object distribution is relatively rare in reality. Other examples tested are several scenes of sphereflakes. These scenes have the following distinct characteristics:

1. Objects in the scene differ significantly in size.
2. The objects' distribution is uneven in space.
3. There exists a background plane.
4. All balls are reflective but without transparency.
5. Three light sources are assumed.

We ran three models of sphereflakes, each with a different number of objects in the scene. The timing of these scenes is listed in Table 5.4. The timing results indicate that the execution time is almost linear in the number of objects in the scene. The more objects exist in a scene, the more significant the speed-up is. The number of CAM operations is listed in Table 5.5 for estimating the CAM hardware time. The real hardware time is far less than one second and thus can be ignored. Table 5.6 compares the uniform subdivision method with our CAM octree method. The CAM octree ray tracer is up to eighteen times as fast as the uniform grid ray tracer in these examples.

	Resolution of Eye Rays	*Level of Subdivision*	*No. of Octants*	*Execution Time (Sec.)*	*Speed Up*
Balls91	(100,100)	2	38	4.8	9.3
Balls820	(100,100)	4	341	12.2	42.0
Balls7381	(100,100)	5	2956	32.8	157.0

Table 5.4: Statistics of the CAM octree ray tracer for several sphere-flake scenes.

	CAM Searches	*CAM Reads*	*CAM Writes*
Tetra1024	904,393	157,947	256
Balls91	53,128	64,476	48
Balls820	203,467	126,942	623
Balls7381	428,711	205,381	3864

Table 5.5: Statistics of CAM operations of the CAM octree ray tracer

5.6 Analysis

The performance of the CAM octree ray tracer has been shown in the previous section. Here we shall undertake some analysis of the algorithm and its complexity. In our current implementation of the CAM octree ray tracer, we experimented with object models of Breps and pure primitive instancing schemes. Our algorithm can be extended straightforwardly to speed up ray tracing of CSG models.

5.6.1 The Time Complexity

Tracing a ray through an object space comprises of three stages. In the first stage the program tests intersection of the ray with the octree root space. If there is an intersection then the program passes onto the second stage where we first initialise the parameters for *adaptive 3D-DDA* then incrementally calculate ray segments. The third stage of the CAM ray tracer involves CAM searches for intersected octants. The responding octants are visited in a closest first order and objects associated

	Resolution	*Uniform*	*CAM*	*Uniform / CAM*
Tetra1024	(100,100)	9.1	2.9	3.1
Balls820	(100,100)	75.3	12.2	6.2
Balls7381	(100,100)	592.4	32.8	18.1

Table 5.6: Comparisons of ray tracers using uniform subdivisions and using CAM octrees.

with each octant are tested against the ray. The optimal performance of the CAM ray tracer depends on the balance of the above three stages. Other computations of ray tracers include calculating the lighting effect and generating secondary rays of shadowing, reflection and so on. For these secondary rays the above three stages are repeated.

In the tetrahedral pyramid example, the average number of ray-object intersections per ray is 1.35. All intersection tests consume 29% of the total execution time. The time for octant traversal is 48% of the total time. The rest of the execution time is spent on shading and other calculations. In comparison, we examined the corresponding statistics for Glassner's octree ray tracer [Glas84]. It gives an average of 25 ray-object intersections per ray. Even with such a large number of intersections per ray, the intersections used up less than 10% of total time. The overheads in octree traversal cost about 90% of total execution time.

The improvements of our CAM ray tracer to conventional octree accelerated ray tracers [Glas84, Kapl85] are mainly due to constant CAM searches, special CAM pattern matching capabilities, and the simple *adaptive 3D-DDA* algorithm. Because of the parallel CAM searching capability, the time to locate the octant which encloses a given point is O(1) instead of $log(n)$ as for conventional octrees. By using pattern matching and *adaptive 3D-DDA* we avoid calculating the point which is used for locating the next octant.

5.6.2 Parallel Primary Rays

The first and the second stages can be simplified for primary rays if we transform objects into the image space and subdivide the image space. In this case, all primary rays are parallel to each other and to one of the three axes of the space. Primary rays correspond to the best situations of the *adaptive 3D-DDA* algorithm. Each ray is a parallelepiped which has unit square and a length of the size of octree root space. There is no need to compute the parameters of *adaptive 3D-DDA* for parallel primary rays. The ray which does not hit any non-empty octant and thus hits nothing in the space can be detected in a single CAM `search`. Only secondary rays and shadow rays involve the *adaptive 3D-DDA* procedure.

5.6.3 Tight Bounding Octants

The tightness of a bounding octant is also an influencing factor. In our ray tracer, we developed a two pass preprocessing method to make bounding octants tighter. The first pass is a conventional object classification procedure. In the second pass, we check the tightness of octants which are larger than the smallest octant in the tree. It is implemented by examining how many suboctants of an octant are empty. If the number of empty suboctants is greater than five then the octant is replaced by its non-empty children. Table 5.7 shows some examples. We can also store irregular but tight cells, for instance some rectangular parallelepipeds, in CAM. This will save CAM entries and will not affect cell traversals by the *adaptive 3D-DDA* method.

The problem which remains unsolved is the case where the space subdivision

	No. of Nodes In First Pass	*No. of Non-tight Cells*	*Percentage of Non-tight Cells*	*No. of Nodes In Second Pass*
Balls91	32	4	12.5	38
Balls820	246	76	31.0	341
Balls7381	2514	314	12.5	2956

Table 5.7: Examining the tightness of bounding octants for sphere-flake images.

is beyond optimum. Figure 5.9 is such an example where we can not get a tight bounding cell and a ray may miss all the objects in the cell.

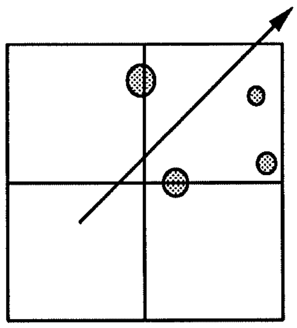


Figure 5.9: A problem of octree space subdivisions: a ray missing all objects in an octant.

5.7 Further Discussions of Space Subdivisions for Brep Models

5.7.1 The Problem

One interesting problem of octree subdivision schemes is to determine when space subdivision should stop. There are two criteria to stop further subdivision. The first one is the complexity of an octant. In our cases for indexing objects, the complexity of an octant is determined by the maximum number of primitives in the octant. If the number of objects is greater than the criterion, the octant is subdivided into eight suboctants. The process is repeated for each suboctant recursively until the number is equal to or smaller than the criterion. The second criterion is the maximum tree depth which is the maximum possible level for subdivision. Beyond this level no further subdivision is allowed.

The choice of the first criterion, that is the maximum number of primitives, is tricky. If the number is too large, there will be many ray-object intersection tests within each octant. The resulting speed up may not be optimal due to the still large number of intersection tests. It is desirable to have as few primitives in each octant as possible in order to achieve more reduction in the number of ray-object intersection tests. However, when the number of objects in an octant becomes smaller another problem arises. The tree depth must be deep enough in order to have fewer primitives within each octant. The time for tree traversal will increase tremendously for conventional tree data structures. The overheads for accessing octree nodes will become significantly large outweighing the gain from reducing the number of ray-object intersection tests. Therefore we should pay attention to both criteria. It is crucial to determine when the subdivision is profitable.

The choice of the maximum tree depth is influenced both by available memory space and by octree node access speed. With conventional octree ray tracers a deep tree will increase the memory requirement and slow down octant location. For CAM octrees the maximum depth of the tree is determined by the available CAM word width. A width of 48 or 64 trits will allow a maximum depth of ten or more. On the other hand the increasing depth will not lead to very much slowing down for accessing a node. The computation of ray traversal through octants is very efficient with our CAM octree ray tracer. Therefore, the level of space subdivisions can be relatively deep and the number of objects associated with each octant becomes smaller. When we can afford a finer subdivision, other questions arise. How fine is necessary? How fine is enough? It would be interesting to know when a subdivision is optimal for an arbitrary scene. What is the general criteria for determining that an octant can be simple enough for good performance? The algorithm should be as fast as possible and be optimal for general cases.

A common practice is to set the maximum number of primitives to six. However for some databases especially for objects defined as Breps (polygons or patches), it is very easy to have some leaves which are subdivided to a very deep level and still have seven or more primitives. We explain why this happens using the following two examples.

The first example is an object composed of polygons obtained from surface tessellation of the object. A surface tessellation is a connected mesh of pieces such as triangles which approximate the surface of an object. This technique is used in ray tracing mathematically defined surfaces such as parametric and implicit surfaces, and their boolean combinations [Snyd87]. Triangular tessellation of surfaces is implemented by sampling the parametric surfaces S(u,v) as grids and then breaking these grids into triangles. There are two ways for sampling the u,v region: uniform sampling and restricted adaptive sampling (Figure 5.10 (a) (b)). The latter is a new type of quadtree in which adjacent quadrants have restricted size relation (two neighbouring nodes differ by one level at most) [Herz87]. The adaptive sampling is more robust and efficient than uniform sampling for deformed and intersecting parametric surfaces. It is noted that with the uniform sampling technique each vertex has at most six polygons connected to it. In adaptive sampling the fineness of the subdivision over the parameter space can vary. Thus there exist certain vertices which have seven triangles converging on them. Therefore around these vertices there are always more than six primitives and the spaces are always subdivided to the deepest level.

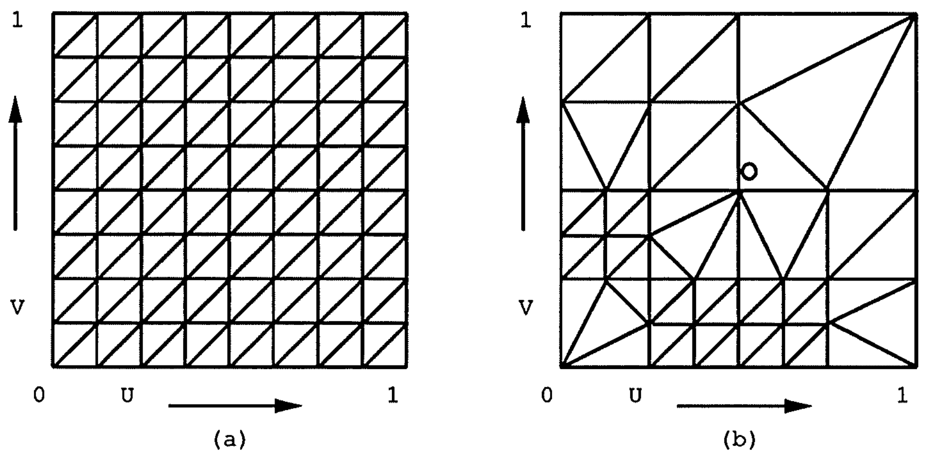


Figure 5.10: Two surface tessellation techniques: the parameter space is tessellated into triangles with (a) a uniform subdivision, (b) a restricted adaptive subdivision.

The second example is a cone of 40 polygons (Figure 5.11) from a database in the 3D object file format which is used by the graphics community for interchanging 3D objects. Among these polygons, 20 polygons share the same vertex at the apex of the cone. Here the number of primitives should be defined as 20 to avoid too fine subdivisions at the apex.

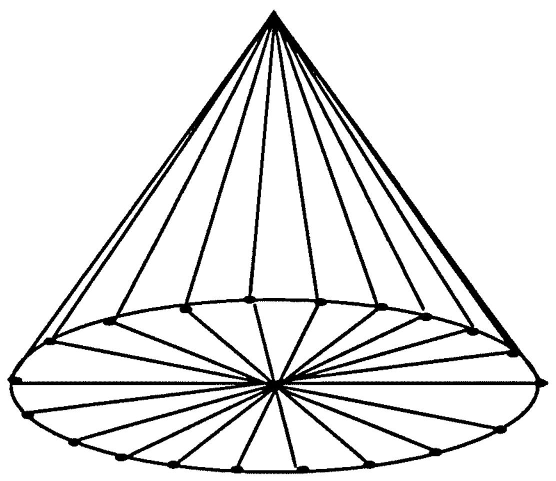


Figure 5.11: A cone defined in Brep

5.7.2 Using Integrated Vector Octrees

For a scene composed of objects defined in Breps (polygons and patches) and with a moderate complexity, we can use subdivision criteria of vector octrees. The scheme using vector octrees is slightly different from that of general object indexing. By using a vector octree, the object space is subdivided not according to the number of primitives but according to vertices, edges, faces and their connectivity relations. Similar to constructing a vector octree which has been described in Section 4.6, a subdivision stops when the space can be classified as one of v, e, f, v', e', v" and empty cells or the space corresponds to a minimal cube (a voxel). The space is then encoded as a leafcode and stored in CAM. The only difference of our index here compared with the previous vector octree is that cell types themselves are not stored, but replaced by pointers. Each pointer relates to the address of the first polygon from the group of polygons associated with the octant. The groups are organised in the same way as shown in Figure 5.2.

Once polygons are classified and organised through a CAM octree space indexing, we can use the CAM octree ray tracer presented in previous sections to generate the image. In a close examination of ray coherence and polygons' distribution within each octant in the vector octree indexing scheme, we noticed that in some common cases it is unnecessary to test all polygons in an octant before finding the nearest intersecting point. Our aim is to find the nearest intersection point as quickly as possible. It is readily conceivable that the data which is most likely to be accessed should be at the front of the sequential list. Therefore such data can be found earlier and the cost of accessing it is reduced. This idea can be used in searching the closest intersecting polygon from a group of polygons in an octant. We can keep the current intersected polygon (non-back-facing polygon) at the front in the group. When the next ray visits the same octant, it is also very likely to encounter the same polygon. So the first test may find the intersecting polygon. In such case it is not necessary to further test other polygons in the group.

However there are some special cases which violate the above rule. One of the examples concerns polygons from transparent objects or concave objects. In these situations several polygons from the group may intersect the ray and the correct intersection point must be selected from them. We can separate these cases from the general situation by marking the related octants as irregular cells. When a ray encounters any of these cells we test all polygons associated with the cell for ray-object intersection, then find the right one. The irregular cells also include octants in which polygons from two or more objects meet.

Summarising the advantages of using vector octrees to index polygon faces of objects in Breps, we conclude that,

1. By indexing Brep primitives in a vector octree, subdivisions near a congested vertex can stop earlier. A vertex is said to be congested when more than six polygon faces are connected to it.

2. The number of ray-object intersection tests in a regular node of the vector octree can be reduced by placing the most recently intersected polygon at the front of the group. This polygon is very likely to be intersected by the next ray.

5.7.3 An Example

The new algorithm has been tested using a scene of a soccer ball. The database of the ball consists of 32 polygons, each polygon has either five edges or six edges. The image of the soccer ball is in Figure 5.12. The statistics of intersection tests and comparison of the improved algorithm with the previous CAM octree ray tracer are listed in Table 5.8. It demonstrated that the number of intersection tests is reduced from 5369 to 4268 and an improvement of 20.5% is achieved.

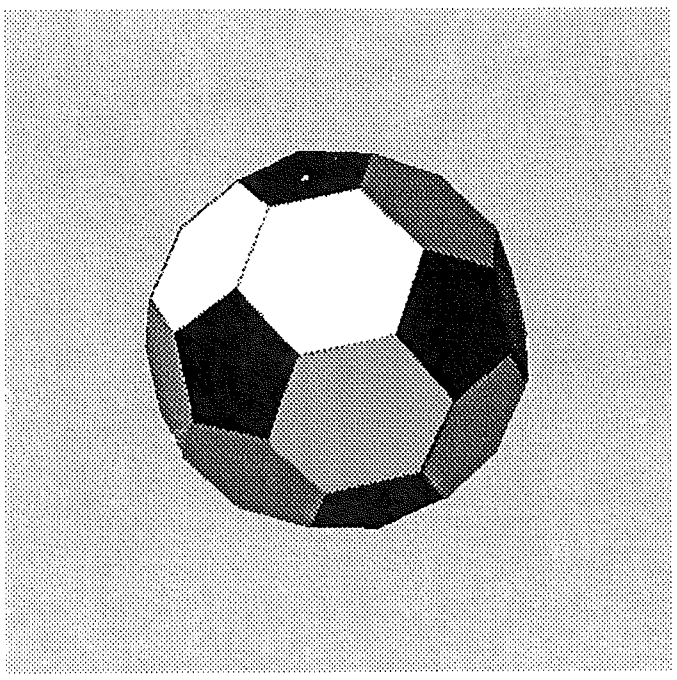


Figure 5.12: The image of a soccer ball

	No. of Inters. Tests	*Percentage of Improvement*
Old CAM Version	5369	0
Improved Version	4268	20.5

Table 5.8: Comparison of the CAM octree ray tracer and the improved algorithm for Breps.

5.8 Summary

This chapter presented a new ray tracing acceleration algorithm using CAM octrees. The new algorithm gives memory savings compared with Glassner's octree method and all other uniform subdivision methods. Several major features can be noted.

- It uses an adaptive space subdivision.
- It keeps non-empty octants only.
- Each octant binds more tightly to its associated objects.

In comparison with existing space subdivision algorithms, the new CAM algorithm has the smallest execution overheads for data structure manipulations. The new CAM approach provides fast ray traversal due to the fact that:

1. No ray-octant intersection tests are necessary.
2. The next non-empty octant to be visited for testing ray-object intersections is found by simple searches in CAM.
3. The use of *adaptive 3D-DDA* allows, in general, a small number of steps to traverse the object space. The empty spaces are skipped quickly for arbitrary rays.
4. The traversals of parallel primary rays can be handled even more efficiently. They involve only searches and a small number of integer operations.

The underlying benefits of employing CAM are that we can accommodate adaptive and fine space subdivisions without too much increase in memory or a significant decrease in speed. Therefore the Brep object classifications using vector octrees become feasible. The CAM octree scheme is also simpler than conventional octree methods. For moderately complex scenes which are often used in geometric modelling systems, the CAM approach can be the ideal choice for its algorithm simplicity and efficiency.

Chapter 6

Ordered-Retrieval of CAM Octrees

6.1 Introduction

One problem unique to CAMs concerns multiple responses. In CAM operations, a search may yield zero, one or many responders. Reading many responders into a sequential device is required in some circumstances. How to select a word among many responders is then an interesting problem. This requires the CAM system to provide a means to indicate the number of responders and to select each responder in an order. The possible orders of selection are:

1. at random;
2. based on physical locations;
3. based on contents of stored words.

The last two orders are usually used in CAM designs. The current Syracuse CAM design adopted the second order of selection. Selecting words in order of their physical locations is very straightforward. During searching, the CAM hardware detects whether there are zero, one or many responders. The responded words are examined sequentially, one at a time. When many responders exist, a multiple responder resolver (MRR) is activated to select one word to read at each cycle. The MRR selects the first word using an external priority scheme based on the physical location of the word. After the word is read out, the MRR clears the current responder and proceeds for the next selection. The process is repeated until all responders are read out.

A more complicated design is required sometimes in applications. It uses the third type of order selection to retrieve CAM words according to the values of words (or parts of words). For binary word storage, the order can be ascending (or descending) by selecting a responder which has a minimum (or maximum) numerical value. As seen later, several hardware architectures have been investigated for ordered retrieval and have been found very useful in many applications [Seeb62, Lewi62, Rama78].

The ordered retrieval of binary words from CAMs has been described by several groups of researchers. At the early stage of CAM research, most work was developed using cryogenic and superconductive elements. For example, Seeber and Lindquist [Seeb62] designed a cryogenic CAM for ordered retrievals. Lewin [Lewi62] proposed a special read circuit which has some remarkable features. It uses a "column-pair" sensing arrangement with an interrogation routine that retrieves an m-word list in exactly $2m-1$ read cycles. Details of his design will be discussed later. Advances in VLSI technology during the subsequent decade provided researchers with more economic means for CAM manufacture. In the late seventies, Ramamoorthy *et al.* [Rama78] designed a fast cellular associative memory for ordered retrieval using distributed logic. In their design, the control logic and the storage logic are designed together. This is in contrast with previous designs that place the control logic outside the storage logic. The search and read operations are parallel by word and serial by bit-slice. There are many other hardware designs which can be found in the review paper of Parhami [Parh73]. Although it is still difficult to make practical designs for CAM ordered retrieval in a large scale under current technology, it may be feasible in the future when VLSI technologies are further developed.

This chapter analyses the feature of ordered retrieval from the applications' point of view. Specifically, ordered retrieval of CAM octree nodes is considered and the importance of this feature for octree operations is assessed. Previous hardware designs of ordered retrieval are mainly for CAMs of binary word storage whereas the octree CAM contains ternary word (trit) storage. Thus we need to introduce the notion of order for trit words. To enable ordered retrieval, words must satisfy the uniqueness requirement. We will give an informal proof of the uniqueness of trit words for octree applications. Lewin's binary word ordered-retrieval theorem is described and is then extended to trit words. Some examples are presented to demonstrate the improvement using CAM ordered retrieval to octree operations such as tree traversal, back-to-front display, scan line conversion and ray tracing.

6.2 Orders of Octree Nodes

To retrieve nodes in order, the following questions should be answered:

1. What can be used to identify the order?
2. What does an order mean for octree nodes?
3. How can we find the order in CAM words which contain *don't cares*?

In answering the first question we re-examine the structure of CAM octrees. A CAM octree is a leafcode stored in CAM, and uses *don't cares* to indicate the sizes of the nodes. Each word in CAM has three fields (*id*, *location*, and *colour*) which are used to identify different trees stored in CAM at the same time, to locate a node within one tree and to indicate the colour of a node respectively. The order in the *id* field is less relevant than the other two fields for general applications. In the following discussion we only consider one tree, which means that *id* is fixed. The order in the *colour* field shows insufficient information in separation of words, since many octants may have the same colour. It is not suitable for ordered retrieval.

The locational order is the most widely used and an important feature in geometric modelling. In a CAM octree, each leaf is disjoint from other leaves. The uniqueness of a leaf in the octree implies that any two nodes of the octree must have different locations. Thus all the nodes of the octree are distinguishable by their locations. The locational field can therefore be used as a key field for ordered retrieval. The detailed comparison of the trit patterns will be discussed in the latter sections.

Now we move to the second and the key problem—the orders of octree nodes. We start from the simple problem of orders of point data, then go to the volume data of octants. The point data are stored as binary words whereas the volume data are stored with *don't cares.* The order in a 1-D space is very simple. A series of points can be sorted in either increasing or decreasing order along the axis of 1D space. For higher dimensions such as in 2D or in 3D spaces several possible orders exist. The user must define more precisely which specific order is required. Points in a 3D space can be organised in different ways according to the priority of the three principal axes. We need to specify the primary key, secondary key and so on, for ordering points. This means that points must be ordered on the basis of their coordinate values on each axis in turn. If several points have the same coordinate value on the primary key, then these points are compared by their values on the secondary key.

The above discussions are based on point data. The space orders of nodes of an octree are different. The main reason is that points are of the same size and differ only in their positions, but octants vary in location as well as in size. We cannot sort octants by their coordinates of position alone. The orders important to octrees are summarised here.

- **The Depth-First Order**

  A typical order of octree nodes is the preorder which is the depth-first traversal order for a pointer octree. Many octree operations like octree display with hidden surface elimination involve preorder traversal. A preorder can be described implicitly in octree leafcodes. When leaves are sorted and visited in an increasing sequence, the tree is traversed in a preorder. This is the key for the efficient use of CAM with ordered retrieval capacity. The preorder also applies to point data if locations of points are encoded like those of nodes of leafcodes.

- **The Breadth-First Order**

  The breadth-first traversal of an octree is mainly used in pointer octrees to skip over subtrees in some operations. This is seldomly used in leafcodes. Therefore it is less relevant to CAM octrees which are one form of leafcodes.

- **Other Orders**

  There are some other spatial orders for traversing an octree. They are related to the dimension of the enquiry space. The enquiry space can be a line (1D) parallel to one axis of the 3D space. An example of a scan-line in an octree space is given in Figure 6.1(a). The line can be either a complete scan or a segment. Octants which intersect the scan line are read out in an ascending sequence by the nodes' coordinate values along the axis. If the enquiry space is a slice (2D) as in Figure 6.1(b), the intersected octants follow the preorder of the output tree.

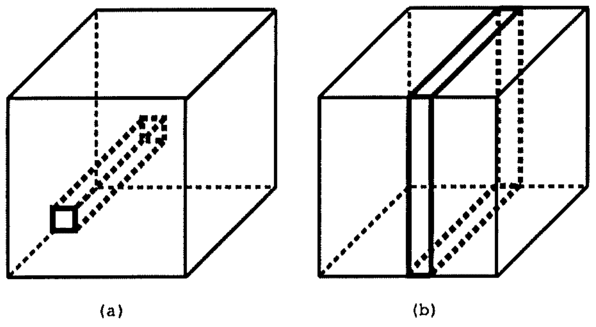


Figure 6.1: Examples of enquiry spaces: (a) a scan line, (b) a slice

6.3 Distinguishable Trit Words

To do ordered retrieval, each word among those words which will be read out must be unique. This means that we need to store distinct words in the CAM memory.

In the case of binary storage, two words are distinguishable if and only if there exists at least one digit column in which two words contain different values. With the situation of trit storage, like Syracuse CAM, we must know what separates two trit words. Since a *don't care* matches both 0 and 1, two words are indistinguishable at the column where one of them has the value *don't care*. For example, the word with the location pattern of 001 *** and the word with that of 001 100 represent two different nodes but can not be distinguished for order. Two distinct trit words must be different in at least one digit column and in that column none of them has a *don't care*. The words 001 *** and 000 100 are distinct as they are different in the third column. Only the values 0 or 1 of a trit column contribute to separate two words and are important when implementing ordered retrieval of nodes of CAM octrees.

We also note that nodes of an octree are disjoint from each other. This guarantees that each trit word in a CAM octree is unique. The case of non-distinct words 001 *** and 001 100 mentioned above will not occur since they represent two octants which partially overlap in the space. The octant corresponding to 001 100 is in fact a suboctant of the node 001 ***.

6.4 Lewin's Ordered Retrieval of Binary Words

As mentioned before, there were several proposals for designing CAMs for ordered retrieval. We chose Lewin's method as an example to demonstrate the principle of the design. Lewin's design retrieves only binary words. Extension from binary words to trit words for Lewin's algorithm will be discussed in the next section.

The method that Lewin proposed in the early 60's [Lewi62] aims at solving the problem of sequential reading out from a binary CAM. It involves generating an interrogation sequence which successively isolates each word from the list of words to be read out. The interrogation process is based on a sensing arrangement for the memory and will be summarised below. It has been demonstrated that with this design ordered retrieval of m responders requires exactly *2m-1* memory cycles, and is independent of the word width.

Lewin's system contains a memory with an array of words and a register of interrogation drivers. Each word in the array is b bits in width. The register has b sensing devices, each has a pair of sense lines coupled to the corresponding bit cell in every word. Four output states can be detected by each sensing device. These states are shown in Table 6.1.

Names	*Column-Pair Signals*	*Meaning*
Sense "0"	0,1	All words selected have "0" in this position.
Sense "1"	1,0	All words selected have "1" in this position.
Sense "X"	1,1	Some of the words selected have "0" in this position. The others have "1".
Sense "Y"	0,0	No word selected.

Table 6.1: Lewin's column-pair sense signals and the meanings

The memory access has two steps.

1. The first step activates a set of words to be retrieved by searching for all words that match the given search pattern. This set of words will be read out into a sequential device in an order. Here we assume that the read-out order is an ascending one.

2. The second step executes an interrogation routine which gives a series of interrogation patterns, each pattern subsequently isolates a subset of responders. The pattern at a given cycle depends on the pattern and sensed results of the previous cycle. The process of interrogation is described below.

At any memory cycle, the system simultaneously senses the value in all columns of the set of words using the current interrogation pattern. The sensing result indicates whether individual columns contain mixed values of 0 and 1, or only one value (either 0 or 1), or no digits. If there are some columns containing mixed values, then the interrogation pattern is changed by setting the most significant mixed-value digit column to 0. In the next memory cycle, sensing is performed with the new pattern. The result of sensing is the activation of a subset of words from the previous set. A complementary subset is discarded. The process is repeated and the size of the subset is successively decreased until there is no more mixed value in any column. Then the word is retrieved. After reading out the word, the interrogation pattern is changed in a reverse direction. The least significant driver which has a value 0, is set to 1 and all drivers on its right are changed to *don't care*. This pattern visits the previously discarded complementary subset. The process stops if no more mixed values are sensed and the last interrogation pattern has no 0 state.

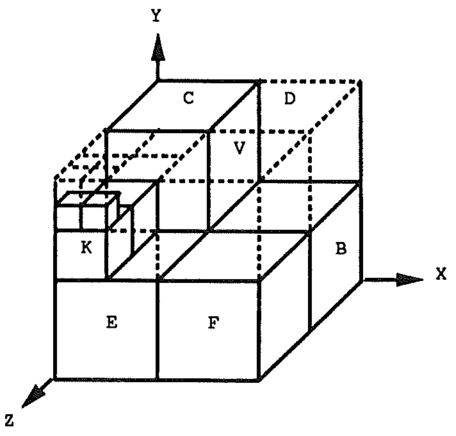
6.5 Extension of Lewin's Method to Trit Words

The above binary word ordered retrieval routine can be extended in order to retrieve trit words. The main difference between binary words and ternary words in CAM concerns *don't cares*. With ternary CAM storage, the bit cells discussed in the last section become trit cells. Each sensing device has its sensing lines coupled to a trit column in every word. The four useful output states from sensing a column of trit cells are:

1. Some cells in the column are 0 and others are *don't care*.
2. Some cells in the column are 1 and others are *don't care*.
3. Cells in the column have 0, 1, and/or *don't care*.
4. Selected words have *don't care* only in the column.

As shown in Section 6.3, the *don't care* in any column does not contribute to separate two words. For any column which contains *don't cares* and some digit value (0 or 1), the sensing result should be determined by digit 0 and 1 in the column. Like binary CAM, we name the above four states of a sensing device 0, 1, X and Y states. Here the Y state of Lewin's column-pair sense signals is used to indicate the column where all cells have *don't care* in storage.

Here we give an example of retrieving an octree in a depth-first order. The object is shown in Figure 6.2 together with the corresponding CAM contents of leaves. The solid lines denote visible edges of black nodes of the object. The dotted-lines are visible edges of white nodes which represent empty space.



(a).

	id	location	colour
A	01	000 *** ***	01
B	01	100 *** ***	01
C	01	010 *** ***	01
D	01	110 *** ***	00
E	01	001 *** ***	01
F	01	101 *** ***	01
G	01	011 000 ***	01
H	01	011 100 ***	00
I	01	011 010 ***	00
J	01	011 110 ***	00
K	01	011 001 ***	01
L	01	011 101 ***	00
M	01	011 011 000	00
N	01	011 011 100	00
O	01	011 011 010	00
P	01	011 011 110	00
Q	01	011 011 001	01
R	01	011 011 101	01
S	01	011 011 011	00
T	01	011 011 111	00
U	01	011 111 ***	00
V	01	111 *** ***	00

(b)

Figure 6.2: An object stored as a CAM octree for ordered retrieval: (a) the octree decomposition of the object and (b) its CAM contents

Figure 6.3 demonstrates the process of getting the order of octants shown in Figure 6.2. Figure 6.3(a) is the search pattern which looks for all black nodes of the octree. Figure 6.3(b) lists the words responding to the above search. These words are activated on the searching cycle. Figure 6.3(c) prints out all memory cycles of the interrogation process. The interrogation driver states are a set of descriptors which are used to select a subset of words. The descriptor at the first cycle come from the search pattern shown in Figure 6.3 (a). Each subsequent descriptor is updated from its previous one according to the sensed condition of the previous cycle. If Xs were sensed, the current descriptor is set by changing the driver which corresponds to the left most X of the previous cycle to the 0 state. For example,

(a) The Search Pattern: 01 *** *** *** 01

(b) Responders: A, B, C, E, F, G, K, Q, R

(c) Interrogation Routine:

Cycle	Interrogation Driver States	Sensed Condition	Word Selected
1	*** *** ***	XXX XXX X 0 1	A,B,C,E,F,G,K,Q,R
2	0** *** ***	0 XX XXX X 0 1	A,C,E,G,Q,R
3	00* *** ***	0 0 X YYY YYY	A,E
4	000 *** ***	0 0 0 YYY YYY	A^
5	001 *** ***	0 0 1 YYY YYY	E^
6	01* *** ***	0 1 X YYY Y 0 1	C,G,K,Q,R
7	010 *** ***	0 1 0 YYY YYY	C^
8	011 *** ***	0 1 1 0 XX X 0 1	G,K,Q,R
9	011 *0* ***	0 1 1 0 0 X YYY	G,K
10	011 *00 ***	0 1 1 0 0 0 YYY	G^
11	011 *01 ***	0 1 1 0 0 1 YYY	K^
12	011 *1* ***	0 1 1 0 1 1 X 0 1	Q,R
13	011 *1* 0**	0 1 1 0 1 1 0 0 1	Q^
14	011 *1* 1**	0 1 1 0 1 1 1 0 1	R^
15	1** *** ***	1 0 X YYY YYY	B,F
16	1*0 *** ***	1 0 0 YYY YYY	B^
17	1*1 *** ***	1 0 1 YYY YYY	F^

^ means that the current word is read out on this cycle

(d) Order of Output Nodes: A, E, C, G, K, Q, R, B, F

Figure 6.3: The interrogation process to retrieve trit words from CAM in an ascending order

the interrogation driver states in cycle 2 are set to 0** *** *** which is updated from the sensed condition XXX XXX X01 of cycle 1. If there was no X sensed in the previous cycle, one word must have been read out. The current descriptor is then set by advancing the right most 0 driver of the previous descriptor to the 1 state, and changing all 1 driver states to its right into *don't care.* One of examples is cycle 5 which follows reading out of node A in cycle 4. In cycle 17, the process stops as there are no more 0 driver states. The final order of these words is listed in Figure 6.3(d).

6.6 Applications

This section demonstrates several common applications of ordered retrieval of a CAM octree. They include algorithms of octree preorder traversal, octree scan conversion and octree acceleration of ray tracing. Comparisons of Syracuse CAM algorithms and the new ordered-retrieval-based CAM algorithms are given. It is shown that with ordered retrieval CAM we can further improve speed and simplify software.

6.6.1 Preorder Traversal

6.6.1.1 Algorithm 1

This is a depth-first traversal algorithm (see Algorithm 6.1) for displaying an octree in a back-to-front order. In the octree, background nodes are stored as nodes of other colours. The algorithm is designed based on Syracuse CAMs and is a simple recursive process. The coding schemes, that is whether a leaf is encoded by interlacing the coordinates in the x y z or z y x order, influence the order of the output. However the final images from these orders are the same.

```
PROCEDURE CAMtraverse(TRITS id, OCT octant)
/* traverse the octree in preorder, display leaf nodes */
/* background nodes are stored */
BEGIN
    Search(id, EncodeOct(octant), ANY);
    IF (MultipleResponse()) THEN
        FOR i = 0 TO 8 DO
            CAMtraverse(id, SubOctant(i, octant));
    ELSE
        /* display the leaf node */
        Read(&dummy, &dummy, &colour);
        Display(octant, DecodeColour(colour));
    ENDIF
END
```

Algorithm 6.1: Displaying a CAM octree with background nodes stored in the CAM

6.6.1.2 Algorithm 2

This algorithm (Algorithm 6.2) is slightly different from Algorithm 6.1 by storing only foreground nodes in the CAM. Therefore less nodes are visited during the traversal of the octree. There are possibilities that some searches return no responder when the enquiry space corresponds to a non-stored empty node.

```
PROCEDURE CAMtraverse(TRITS id, OCT octant)
/* traverse the octree in preorder, displaying foreground leaf nodes */
/* background nodes are not stored */
BEGIN
    IF (!Search (id, EncodeOct (octant), ANY)) THEN return;
    ELSE IF (MultipleResponse()) THEN
        FOR i = 0 TO 8 DO
            CAMtraverse(id, SubOctant(i, octant));
    ELSE
        Read(&dummy, &location, &colour);
        Display(DecodeOct(location), DecodeColour(colour));
    ENDIF
END
```

Algorithm 6.2: Displaying a CAM octree without white nodes

The above algorithms are based on CAMs without ordered retrieval. Their main sources of inefficiency are recursive procedure calls and suboctant calculations during octree traversals. A simple example that involves a traversal of a tree of 299,593 nodes has been tested to estimate how much time these procedures consumed. On a MicroVaxII, the time for 299,593 recursive calls is 16.08 seconds and the time for subdivisions is 15.46 seconds. The total time is 31.54 seconds plus the CAM access time which is approximately 0.05 seconds, assuming each CAM operation costs 100ns. The time of recursive procedure calls and subdivisions dominates the whole process, and forms the bottleneck for the system performance. When a conventional octree algorithm is dominated by the above two procedures, the Syracuse CAM has no practical advantage in terms of improving the algorithm's performance. It is thus necessary to remove recursive calls. Removing recursion is also important when non-recursive low level machine languages are used.

6.6.1.3 Algorithm 3

Algorithm 6.3 is a new algorithm using CAM with ordered retrieval hardware.

This algorithm is insensitive to background nodes. Here we assume that they are not stored. If they are stored in a CAM octree, we can use the new algorithm either to visit all nodes, or to visit foreground nodes. In the latter case, we move background nodes to a temporary *id* using a `MultipleWrite` operation before calling `CAMtraverse`.

```
PROCEDURE CAMtraverse(TRITS id, OCT octant)
/* traverse the octree in preorder, display leaf nodes */
/* background nodes can be either stored or ignored */
BEGIN
    IF (Search (id, EncodeOct(octant), ANY) THEN
        OrderedRead(&dummy, &location, &colour);
        Display(DecodeOct(location), DecodeColour(colour));
    ENDIF
END
```

Algorithm 6.3: Displaying a CAM octree using ordered-retrieval CAM

6.6.1.4 Statistics

The code of Algorithm 6.3 is simpler than those of Algorithm 6.1 and Algorithm 6.2. Firstly, all recursive calls are removed. Secondly, no subdivisions are necessary. This can be seen more clearly in Table 6.2 which gives a comparison of the above algorithms in terms of the number of nodes in CAM, procedure calls and CAM operations.

	Nodes in CAM	*Recursive Proc. Calls*	*Subdivi-sions*	*CAM Searches*	*CAM Reads*
Alg. 1	8436	9641	1205	9641	8436
Alg. 2	3916	5801	725	5801	3916
Alg. 3	3916	0	0	1	3916

Table 6.2: Comparison of octree display algorithms using CAMs without ordered-retrieval (Alg. 1 and Alg. 2) and with ordered-retrieval (Alg. 3).

6.6.2 Scan Conversion

Scan conversion of an octree can be treated either as a series of conversions from quadtrees into raster scans or a group of tests for line-octant intersections. The latter is a straightforward extension of the 2D quadtree to run-length conversion algorithm. Algorithms for raster scan conversion from quadtrees have been designed with the aid of Syracuse CAM. Two different methods exist.

The first one is described in [Oldf87] and is based on an adaptive scan-line refinement approach. At the start of each scan line, a search pattern is formed by converting the y coordinate into the binary code and setting the x to *don't cares*, then interlacing the coordinates. The pattern corresponds to a rectangle of unit width and with a length of the full size of the root of the quadtree. The CAM is searched with the above pattern and a number of matching entries may respond. When the background nodes are stored in CAM, there is always at least one responder. If a single responder returns, it indicates the number of pixels to be drawn and the colour of these pixels. If multiple records respond, the search pattern is refined by forcing the most significant trit of x to 0. This is equivalent to halving the previous rectangle and selecting the first half as the new enquiry area. The CAM search is carried out for the new pattern. The above process will repeat until a single match is obtained. After the responder is read out, the search pattern is changed again to test the other half of the rectangle. If the background nodes are not stored in the CAM, a search may return no responder. In such cases the pixels corresponding to the area being searched are filled with the background (white) colour. The above algorithm is extensible to 3D by searching rectangular parallelepipeds.

The second method of scan conversion was proposed by Williams [Will88b]. This method uses the pixel locating ability of trit storage. For each scan line, the locational code of the left-most pixel is used as the initial search pattern. The search will return a single responder which is the node enclosing the pixel. By decoding the node size and colour, a run length of the line is obtained. The size of the node is added to the current pixel position to give the coordinates of the next search pixel. The process repeats until the end of the row is reached. If there are m quadrants or octants in a row, processing each row requires m searches and reads, plus some computation. There is no case which returns zero responder or multiple responders. The pixel location method requires the background nodes to be stored in the CAM. Otherwise it will be difficult to determine the next pixel when a search does not have any matching node. Many more searches will be invoked in the background areas.

We observed that a scan conversion is simply a process of reading out nodes which intersect with each scan line in the ascending order. The process of adaptive scan line refinement in the first approach above is actually aimed at isolating a subset of multiple responders step by step to get the spatial order. Since Syracuse CAM retrieves multiple responders by their physical locations, the next word which is required to be retrieved in an order can only be selected in software. The time spent on the refining procedure to resolve multiple responders costs a lot of execution time. If a tree is deep, for example fine subdivisions of a space, the scan conversion process may be slow. To reduce the number of refining steps, a cache method was used [Oldf87]. In moving from a scan line to the next one, the search patterns of the former line are taken as initial guesses for the latter one. If there is no responder, the search pattern is made less precise. If there are multiple responses, the search pattern is made more precise. However, adaptive refinement requires a buffer to hold the active search patterns and needs checking of the buffer. These checks take extra execution time.

Using Lewin's CAM of ordered retrieval, the above refinement process can be eliminated completely. For each scan line, one search operation activates all responding words to be read out. If there are m responders, only $2m - 1$ cycles is needed to read these words in the ascending order. The process is insensitive to background nodes. In the case where background nodes are not stored, their corresponding runs can be derived from the addresses and sizes of two subsequent foreground nodes.

Unlike preorder traversals of an octree, the output order of scan conversion is

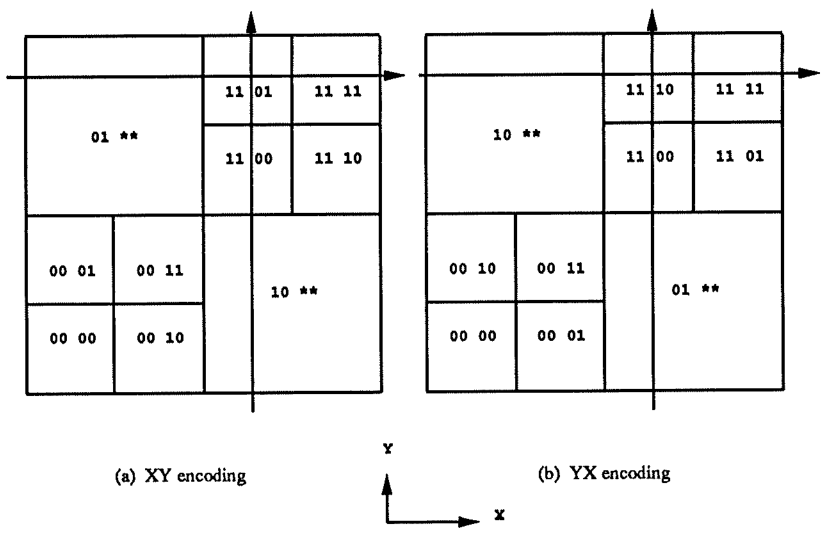


Figure 6.4: The coding scheme does not affect the order of a raster scan along any of the three axes.

not affected by the coding schemes. We demonstrate this using a 2D diagram shown in Figure 6.4.

6.6.3 Ray Tracing

The ray tracing algorithm based on CAM and *adaptive 3DDDA* in Chapter 5 involves comparison of a series of ray segments of a ray against CAM contents to select all octants intersecting the ray in an order. The order must be such that it allows the ray to visit the octant nearest to the ray origin first. A ray segment is either a complete or a partial scan line. Thus a method similar to ordered retrieval for the scan conversion algorithm can be used in ray tracing. Here octants intersecting with a ray segment are retrieved in an ascending order along the main driving axis (MDA) of the ray. The MDA, as shown in Figure 5.5, could be any of the three axes of the octree space.

Once the order can be identified in hardware, other functions closely related to the CAM operations can be pipelined to further speed up the process of ray traversal in an octree. Figure 6.5 shows an simplified example of pipelining the process of searching all octants which intersect a given ray segment. Some buffers are necessary between the stages as different stages may vary in the execution time. The input to the pipeline is the description of a ray segment. The output is a group of indices, each associated with an octant. These octants are in an increasing order along the positive direction of the MDA. If the ray is in the same direction as the axis, these octants are visited in the same order as the output. Otherwise they are visited in the reverse order.

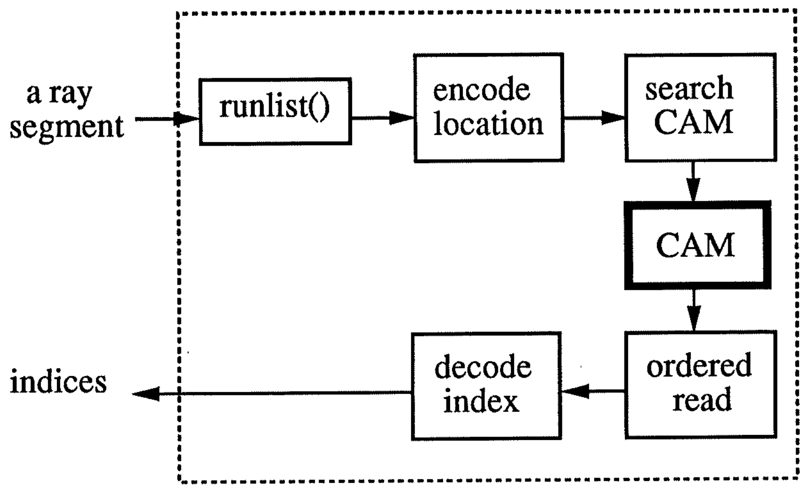


Figure 6.5: Pipelining the process of searching a CAM octree for octants intersecting with a ray.

6.7 Summary

The scheme of ordered retrieval from CAMs of trit words is discussed in this chapter. This feature is found most useful for depth first traversals and scan conversions of octrees, and ray tracing with CAM octrees. By using a special circuit to read out multiple responders in a specific order, the inefficient recursive procedure calls and repeated octant subdivision calculations in programs are eliminated. The scan line refinement process in octree scan conversion and CAM octree ray tracing algorithms is also removed. The software is therefore further simplified. The cost of extra hardware may be justified by the benefit of improved performance, simplified software and increased reliability.

Chapter 7

Concluding Remarks

This dissertation comprises analysis of 3D volumetric representations with emphasis on octree data structures, designs of new algorithms for octree manipulation and visualisation, as well as acceleration of image generation, particularly ray tracing, using CAM architectures. In this chapter we summarise the main findings of this study, and further discuss our CAM ray tracing algorithm in relation with other general purpose parallel machines. These discussions are based on analysis of general parallel ray tracing algorithms and our CAM octree ray tracer. Algorithms which are most efficient for sequential processing by exploiting coherence are not necessarily most suitable for general parallel processing. Therefore, we need to assess the applicability of the CAM octree ray tracer to general purpose multiprocessors. Finally some further studies related to this dissertation are suggested.

7.1 Summary

A volumetric object can be represented either as a large 3D uniform grid of voxels (volume elements), or as a relatively compact non-uniform collection of volumes. Objects represented in the second form are obtained by adaptive recursive decompositions. An octree is a special case of them in which each non-terminal volume is subdivided into eight sub-volumes. In conventional computer architectures, the uniform grid structure is stored as an array of data elements, while the octree structure can be stored in one of three principal forms: a pointer structure; an array of data corresponding to the preorder of nodes; or an array of encoded data. These three formats are named pointer octrees, treecodes and leafcodes respectively. The main drawbacks of uniform grid structures lie in their large memory requirements and slow voxel access. The octree approach is a partial solution to save space and improve speed. However, conventional octree algorithms are still far from the goals of real time implementation, simplicity, consistency and intuition. To achieve real time performance, a special hardware octree engine [Meag82] has been designed. The hardware has a simple architecture, involves only integer operations, and is robust. However, octrees still have some shortcomings even with these hardware assistance, for instance:

- Construction of octrees from other representations is still implemented in software. These algorithms, especially those for converting Breps to octrees, are either brute-force or complicated for pointer trees and treecodes. The process of representation conversion is thus a bottle-neck to octree-based solid modelling systems.

- With leafcodes, there exist some simple bottom-up construction algorithms [Atki86, Tang88] which make use of locational information and spatial coherence of objects. The bottom-up algorithms are also more flexible, for example it can easily be extended to convert objects with curved surfaces to octrees. Furthermore, these simple construction algorithms may be accelerated using hardware. However, leafcodes impose searching problems for implementing most octree algorithms. Searching operations on conventional computers are slow.

Each of the three principal octree formats has some advantages over other formats in terms of their efficiency in either time or storage. One format can be best in some individual experiments, but is worse for other operations. Therefore it is difficult to build one integrated solid modelling system using only one data format and yet to keep all operations at their most efficient. It is also unrealistic to use all three octree formats in one system for efficiency or other reasons. This will inevitably introduce extra maintenance cost. One solution to the above problems may lie in exploiting different machine architectures and finding a better design which is efficient for most octree operations.

Leafcodes are very close to such a goal except for the problem concerning searches. We thus tried to find a way of solving the searching problem. It has been known that the special architecture of content addressable memory (CAM) is well suited to searching operations. A CAM is an architecture that consists of a collection of elements which have data storage capabilities and can be accessed simultaneously on the basis of data contents instead of addresses. The main advantages of CAMs are their parallel searching, pattern matching and masked parallel updating capabilities. We need to examine what can and can not be expected of a particular CAM architecture for our specific problems of octree manipulations.

We have explored and evaluated the possibilities of using CAM in the frame of 3D volumetric modelling and image generation. We developed a novel approach toward using CAM to improve the performance and software design of octree related problems. It should be noted that CAM is not a replacement of conventional computing but is likely to form a complementary technology. CAM and conventional architecture should exist in the same box, with a flexible interface to applications.

CAM is suitable for implementation of octree leafcodes. By using CAM the system performance is improved. We also reduce both the conceptual and programming complexities invoked in implementing octree algorithms. The CAM algorithms are more intuitive and straightforward compared with their conventional counterparts. A new ray tracing acceleration algorithm using CAM and octrees has also been designed and tested. The experiments showed that the new algorithm gives more space saving than conventional octree approaches and is much faster than the current best accelerated ray tracer using space subdivisions. Based on new algorithms developed in this thesis, we could design a CAM octree system to improve the overall 3D operations in solid modelling, 3D image processing and image rendering.

Computing environment is a significant factor for designing algorithms. CAM octrees can provide a base for unifying and simplifying modelling, rendering, geometric searching, property calculation and set operation evaluation in one integrated system. We can also unify the goals of efficiency, simplicity and versatility on the basic CAM octree architecture. It is also interesting to note that CAMs are very flexible. CAMs can support several different tasks: 2D geographical information systems, 3D image processing, 3D solid modelling systems, 4D animation systems, and so on.

7.2 Parallel Ray Tracing with Shared CAM

Since the ray tracing technique has inherent parallelism, it fits massively parallel architectures very well. Parallel and concurrent execution of ray tracers is the only solution for interactive or real time ray tracing under today's computing power. With current VLSI techniques, we hope to achieve real time ray tracing in the near future. Before this can be realised, a lot of research needs to be done to understand parallel implementations of ray tracing.

7.2.1 Classifications

Several approaches are applicable to achieve parallelism of ray tracing. They are closely related to machine or hardware architectures.

- The first approach vectorises the algorithm on Single Instruction Multiple Data (SIMD) machines [Plun85]. These machines include CRAY supercomputers, Connection machines and so on. In this approach, the rays are vectorised straightforwardly and each ray is tested against all objects in the scene. Neither space coherence nor ray coherence is used. It is easy to implement but is inefficient for scenes with a large number of objects.

- The second approach executes ray tracing on a collection of general-purpose computers. Harrison [Harr89] experimented with this approach on the Cambridge processor bank [Need82]. However, this architecture is not suitable for real time implementation of ray tracing because communication is too slow.

- The third one is implemented on general-purpose multiprocessors of Multiple Instruction Multiple Data (MIMD) structure, for instance [Goh90]. A group of processors are connected and all node processors execute their own program in parallel. There exist two categories of MIMD architectures—tightly coupled and loosely coupled processors. In the former, all processors share the same main memory and each have a small local cache. In the latter, each node processor has its own local memory and processing power. Examples of loosely coupled multiprocessors are Hypercubes and transputers.

- The last approach is to build special-purpose hardware in which either pipelining and parallelism are mixed, for instance [Ulln83], or 2D/3D processor grids are used [Clea86].

7.2.2 Multiprocessor Parallel Ray Tracing

We shall concentrate on the third approach—parallel ray tracing on loosely coupled and tightly coupled multiprocessors.

7.2.2.1 Loosely Coupled Multiprocessors

The algorithms on a loosely coupled multiprocessor grid are similar to those on a multicomputer in the sense that both involve local storage on each processor and communication between processors. The aims are to make efficient use of all processors (load balance), keep storage minimal and reduce communications. Two basic problems for loosely coupled multiprocessor parallelism are:

1. How does the program organise the scene database on different processors? Each processor has either the entire scene database or a part of it.
2. How does the program distribute rays? The ray distribution may be related to how the database is organised.

In the case of copying the entire scene to each processor, data management on each processor is simple. Each processor works independently when it is allocated a group of rays. The processor needs only to communicate with the host computer and the amount of communication is minimal. However, this scheme ignores the spatial coherence information in object space. To use spatial coherence we can presort objects using space subdivision (uniform grids or octrees). Here, extra memory space is required to store a grid description or an octree description. The amount of local memory on each processor needs to be large enough to keep the whole scene and the grid description. Naturally, we come to the second solution. Each processor is assigned with one or more regions, each region has a part of the database.

When objects are distributed partly onto each processor, a single processor can not always fulfill the task of ray-scene intersection for a ray, because one processor may have only a small portion of objects from those which need to be tested against the ray. Two methods have been attempted: one moves objects around processors and the other moves rays around processors.

- **Moving objects:** The algorithm based on moving objects around processors is proposed by Green *et al.* [Gree88] for a transputer network. The processors are organised as a tree. An octree is used for subdividing the space. The root processor contains the whole database and the octree description. Each node processor keeps a copy of the octree description and a part of the scene database. A dynamic allocation of objects to processors is implemented. Each processor is assigned a group of rays and tests each ray in turn. When a local request for an object fails, the processor sends its parent processor a request for the object. All processors are well loaded and used efficiently. However, global communication is high. Firstly, when more transputers are used, the processor tree becomes deep. An object needs to pass many node transputers before reaching the destination. Secondly, if a scene database is complex and the octree is deep, the number of requests for failed objects will increase. Intersecting one ray with a scene give rise to a lot of communication. The communication cost would degrade the performance of parallel computation.

- **Moving rays:** Another approach is based on moving rays, [Clea86, Dipp84, Nemo86]. Every processor has a part of the database. Each part contains a group of objects inside a region of the space. Neighbouring processors have adjacent regions. Each processor tests a ray with objects in its region. If intersecting nothing the ray is passed to an adjacent processor and is tested again. The advantage is that objects are copied to each processor only once. This saves the time used to move a large amount of data around processors. The obvious problem is that the load for each processor may not be balanced. Some processors may always be busy because of having a region which contains a lot of objects, while many other processors are far less loaded. There are also network congestion problems at regions near a ray origin or a light source. There were attempts to solve these problems through dynamic load redistribution by introducing more expensive boundary intersection [Dipp84]. Others [Prio89] used static load balancing methods. The parallel ray tracer based on moving rays have another problem of repeated ray-object intersections (if one object is crossing several processors).

In implementing our CAM octree ray tracer for parallel processing on loosely coupled multiprocessors, the above two approaches are applicable. The approach of moving objects can be improved on the following bases:

- Our CAM octree ray tracer has fast ray traversals in scene spaces.
- CAM octrees are more compact in comparison with the conventional octrees by having less nodes and tight cells for indexing spaces.
- By storing octrees in the CAM, each processor can hold more objects hence the number of failed object requests can be reduced and communication cost is lower.

With the above architecture of loosely coupled multiprocessors, the CAM can be connected to one processor which manages high level CAM interface. Other processors access CAM through messages. This approach requires communication between CAM processor and other processors. The bottle-neck will lie in communication and CAM contention.

7.2.2.2 Tightly Coupled Multiprocessors

For tightly coupled multiprocessors which have a shared memory, processors communicate to each other through the main memory. Each processor may also have a local cache working as the high speed buffer. In such systems, the CAM can be connected in the same way as the main memory. There is a complete connectivity between CAM and processors. The problem of passing messages around processors is resolved. On the other hand, CAM access consumes very little time, thus the contention for CAM may be minimal. Our CAM ray tracer developed in this study fits the architecture of tightly coupled processors well.

7.3 Suggestions for Further Study

In this study we have developed a number of new octree algorithms based on the CAM architecture. With the success of experiments on a CAM simulator, further tests and full scale implementation of related algorithms should be possible. Experiments with complete system integration should be investigated. Wider applications of Syracuse CAM in computer graphics, for instance 4D space-time applications, robot development systems and so on, need to be developed further. Finally comparison of our CAM octree algorithms with other parallel processing methods for leafcode octrees on general purpose multi-processors would be worthwhile.

Bibliography

[Akel88] K Akeley and T Jermoluk. High-performance polygon rendering. *Computer Graphics (SIGGRAPH'88)*, **22**(4):239–246, 1988.

[Akma89] V Akman and W R Franklin. Representing objects as rays, or how to pile up an octree. *Computer and Graphics*, **13**(3):373–379, 1989.

[Aman87] J Amanatides and A Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of EUROGRAPHICS'87*, 3–10, 1987.

[Apga88] B Apgar, B Bersack and A Mammen. A display system for the Stellar TM graphics supercomputer model GS1000 TM. *Computer Graphics (SIGGRAPH'88)*, **22**(4):255–262, 1988.

[Arvo87] J Arvo and D Kirk. Fast ray tracing by ray classification. *Computer Graphics (SIGGRAPH'87)*, **21**(4):55–64, 1987.

[Atki86] H H Atkinson, I Gargantini and T R S Walsh. Filling by quadrants or octants. *Computer Vision, Graphics, and Image Processing*, **33**(2):138–155, 1986.

[Barr86] A H Barr. Ray tracing deformed surfaces. *Computer Graphics (SIGGRAPH'86)*, **20**(4):287–296, 1986.

[Boua87] K Bouatouch and M O Madani. A new algorithm of space tracing using a CSG model. In *Proceedings of EUROGRAPHICS'87*, 65–78, 1987.

[Brul88] M R Brule, J V Oldfield, J C Ribeiro and C D Stormon. An architecture based on content-addressable memory for the parallel execution of logic programs. Technical Report 8801, CASE Center, Syracuse University, 1988.

[Brun85] P Brunet and I Navazo. Geometric modelling using exact octree representation of polyhedral objects. In *Proceedings of EUROGRAPHICS'85*, 159–169, 1985.

[Carl85] I Carlbom, I Chakravarty and D Vanderschel. A hierarchical data structure for representing the spatial decomposition of 3-D objects. *IEEE Computer Graphics and Applications*, **5**(4):24–30, 1985.

[Chen88] H H Chen and T S Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, **43**(3):409–431, 1988.

[Chie86a] C H Chien and J K Aggarwal. Identification of 3D objects from multiple silhouettes using quadtrees / octrees. *Computer Vision, Graphics, and Image Processing*, **36**(2):256–273, 1986.

[Chie86b] C H Chien and J K Aggarwal. Volume/surface octrees for the representation of three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, **36**(1):100–113, 1986.

[Clea86] J G Cleary, B M Wyvill, G M Birtwistle and R Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, **5**(1):3–12, 1986.

[Clea88] J G Cleary and G Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, **4**(2):65–83, 1988.

[Cook84] R L Cook, T Porter and L Carpenter. Distributed ray tracing. *Computer Graphics (SIGGRAPH'84)*, **18**(3):137–144, 1984.

[Dipp84] M Dippe and J Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics (SIGGRAPH'84)*, **18**(3):149–158, 1984.

[Doct81] L J Doctor and J G Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, **1**(3):29–38, 1981.

[Dreb88] R A Drebin, L Carpenter and P Hanrahan. Volume rendering. *Computer Graphics (SIGGRAPH'88)*, **22**(4):65–74, 1988.

[Durs89] M J Durst and T L Kunii. Integrated polytrees: a generalized model for the integration of spatial decomposition and boundary representation. In *Theory and Practice of Geometric Modelling*, Strasser, W and Seidel, H P (Eds), 329–348, 1989.

[Fole90] J D Foley, A Van Dam, S K Feiner and J F Hughes. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, second edition, 1990.

[Fost71] C C Foster and F D Stockton. Counting responders in an associative memory. *IEEE Transactions on Computers*, **C-20**:1580–1583, 1971.

[Fran85] V R Franklin and V Akman. Building an octree from a set of parallelepipeds. *IEEE Computer Graphics and Applications*, **5**(10):58–64, 1985.

[Frie85] G Frieder, D Gordon and R A Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, **5**(1):52–60, 1985.

[Fuch89] H Fuchs, M Levoy and S M Pizer. Interactive visualization of 3D medical data. *Computer*, pages 46–51, August 1989.

[Fuji86] A Fujimoto, T Tanaka and K Iwata. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, **6**(4):17–26, 1986.

[Garg82] I Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, **20**(44):365–374, 1982.

[Garg86] I Gargantini, T R Walsh and O L Wu. Viewing transformations of voxel-based objects via linear octrees. *IEEE Computer Graphics and Applications*, **6**(10):12–21, 1986.

[Glas84] A S Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, **4**(10):15–22, 1984.

[Glas89] A S Glassner. *An Introduction to Ray Tracing.* Academic Press, London, 1989.

[Goh90] E L Goh. *Parallel Architectures with Fast Ray-Tracing Image Synthesis.* PhD thesis, University of Camridge, Cambridge, 1990.

[Gold71] R A Goldstein and R Nagel. 3-D visiual simulation. *Simulation*, **16**(1):25-31, 1971.

[Gold85] S M Goldwasser. Physician's workstation with real-time performance. *IEEE Computer Graphics and Applications*, **5**(12):44–57, 1985.

[Gree88] S A Green, D J Paddon and E Lewis. A parallel algorithm and tree-based computer architecture for ray traced computer graphics. In *Proc Conf Parallel Processing for Computer Vision and Display*, Leeds, 431–442, 1988.

[Gree89] S A Green and D J Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, **9**(6):12–26, 1989.

[Hain86] E A Haines and D P Greenberg. The light buffer: a shadow-testing accelerator. *IEEE Computer Graphics and Applications*, **6**(9):6–28, 1986.

[Hain87] E A Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, **7**(11):3–5, 1987.

[Hanl66] A G Hanlon. Content-addressable and associative memory systems: A survey. *IEEE Transactions on Electronic Computers*, **EC-15**(4):509–521, 1966.

[Harr89] S P J Harrison. *Realistic Image Synthesis.* PhD thesis, University of Camridge, Cambridge, 1989.

[Herz87] B V Herzen and A H Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics (SIGGRAPH'87)*, **21**(4):103–110, 1987.

[Hunt79a] G M Hunter and K Steiglitz. Linear transformation of pictures represented by quad trees. *Computer Graphics and Image Processing*, **10**(3):289–296, 1979.

[Hunt79b] G M Hunter and K Steiglitz. Operations on images using quadtrees. *Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-1**(2):145–153, 1979.

[Jack80] C L Jackins and S L Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, **14**(3):249–270, 1980.

[Jack85] D Jackel. The graphic PARCUM system: a 3D memory based computer architecture for processing and display of solid models. *Computer Graphics Forum*, **4**(4):21–32, 1985.

[Joy86] K I Joy and N B Murthy. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. *Computer Graphics (SIGGRAPH'86)*, **20**(4):279–286, 1986.

[Kapl85] M R Kaplan. Space-tracing, a constant time ray tracer. *SIGGRAPH'85 State of the Art in Image Synthesis Seminar Notes*, 1985.

[Kauf87a] A Kaufman. An algorithm for 3D scan-conversion of polygons. In *Proceedings of EUROGRAPHICS'87*, 197–208, 1987.

[Kauf87b] A Kaufman. Efficient algorithms for 3D scan-conversion of parametric curves, surfaces, and volumes. *Computer Graphics (SIGGRAPH'87)*, **21**(4):171–179, 1987.

[Kauf88] A Kaufman and R Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics and Applications*, **8**(6):10–23, 1988.

[Kay79] D S Kay and D P Greenberg. Transparency for computer synthesized images. *Computer Graphics (SIGGRAPH'79)*, **13**(2):158–164, 1979.

[Kay86] T L Kay and J T Kajiya. Ray tracing complex scenes. *Computer Graphics (SIGGRAPH'86)*, **20**(4):269–278, 1986.

[Koho87] T Kohonen. *Content-Addressable Memories.* Springer-Verlag, New York, second edition, 1987.

[Kuni85] T L Kunii, T Satoh and K Yamaguchi. Generation of topological boundary representations from octree encoding. *IEEE Computer Graphics and Applications*, **5**(3):29–38, 1985.

[Lee82] Y T Lee and A A G Requicha. Algorithms for computing the volume and other integral properties of solids. I: known methods and open issues, II: a family of algorithms based on representation conversion and cellular approximation. *Communications of the ACM*, **25**(9):635–650, 1982.

[Lewi62] M H Lewin. Retrieval of ordered lists from content-addressable memory. *RCA Review*, **23**(2):215–229, 1962.

[Mark86] D M Mark. Construction of quadtrees and octtrees from raster data: a new algorithm based on run-encoding. *The Australian Computer Journal*, **18**(3):115–119, 1986.

[Meag82] D Meagher. Geometric modeling using octree encoding. *Computer Vision, Graphics, and Image Processing*, **19**(2):129–147, 1982.

[Meag84] D Meagher. Interactive solids processing for medical analysis and planning. In *Proc. of NCGA 84*, pages 96–106, 1984.

[Nava86] I Navazo, D Ayala and P Brunet. A geometric modeller based on the exact octtree representation of polyhedra. *Computer Graphics Forum*, **5**:91–104, 1986.

[Nava89] I Navazo. Extended octtree representation of general solids with plane faces: model structure and algorithms. *Computer and Graphics*, **13**(1):5–16, 1989.

[Need82] R M Needham and A J Herbert. *The Cambridge Distributed Computing System.* International Computer Science Series. Addison-Wesley Publishing Company, Reading, MA, 1982.

[Nemo86] K Nemoto and T Omachi. An adaptive subdivision by sliding boundary surfaces for fast ray tracing. In *Proceedings of Graphics Interface*, 43–48, 1986.

[Oldf87] J V Oldfield, R D Williams and N E Wiseman. Content-addressable memories for storing and processing recursively subdivided images and trees. *Electronics Letters*, **23**(6):262–263, 1987.

[Oldf88] J V Oldfield, R D Williams, N E Wiseman and M R Brule. Content-addressable memories for quadtree-based images. In *Proceedings of The Third EUROGRAPHICS Workshop on Graphics Hardware*, 1988.

[Oliv83a] M A Oliver and N E Wiseman. Operations on quadtree encoded images. *The Computer Journal*, **26**(1):83–91, 1983.

[Oliv83b] M A Oliver and N E Wiseman. Operations on quadtree leaves and related image areas. *The Computer Journal*, **26**(4):375–380, 1983.

[Oliv84] M A Oliver. Two display algorithms for octrees. In *Proceedings of EUROGRAPHICS'84*, 251-264, 1984.

[Oliv86] M A Oliver. Display algorithms for quadtrees and octtrees and their hardware realisation. In *Data Structures for Raster Graphics*, Kessener L R A, Peters F J and van Lierop M L P (Eds.), 9–37, 1986.

[Parh73] B Parhami. Associative memories and processors: An overview and selected bibliography. *Proceedings of the IEEE*, **61**(6):722–730, 1973.

[Peng87] Qunsheng Peng, Yining Zhu and Youdong Liang. A fast ray tracing algorithm using space indexing techniques. In *Proceedings of EUROGRAPHICS'87*, 11–23, 1987.

[Plun85] D J Plunkett and M J Bailey. The vectorization of a ray tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, **5**(8):52–60, 1985.

[Prio88] T Priol and K Bouatouch. Experimenting with a pallel ray-tracing algorithm on a hypercube machine. In *Proceedings of EUROGRAPHICS'88*, North-Holland, 243–259, 1988.

[Prio89] T Priol and K Bouatouch. Static load balancing for a parallel ray tracing on a MIMD hypercube. *The Visual Computer*, **5**(1):109–119, 1989.

[Rama78] C V Ramamoorthy, J L Turner and W W Benjamin. A design of a fast cellular associative memory for ordered retrieval. *IEEE Transactions on Computers*, **c-27**(9):800–815, 1978.

[Requ80] A G Requicha. Representation for rigid solids: theory, methods, and systems. *ACM Computing Surveys*, **12**(4):437–464, 1980.

[Roth82] S D Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, **18**(2):109–144, 1982.

[Rubi80] S M Rubin and T Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics (SIGGRAPH'80)*, **14**(3):110–116, 1980.

[Same80] H Samet. Region representation: quadtrees from binary arrays. *Computer Graphics and Image Processing*, **13**(1):88–93, 1980.

[Same81] H Samet. An algorithm for converting rasters to quadtrees. *Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-3**(1):93–95, 1981.

[Same88a] H Samet and R E Webber. Hierarchical data structures and algorithms for computer graphics, Part I: Fundamentals. *IEEE Computer Graphics and Applications*, **8**(3):48–68, 1988.

[Same88b] H Samet and R E Webber. Hierarchical data structures and algorithms for computer graphics, Part II: Applications. *IEEE Computer Graphics and Applications*, **8**(4):59–75, 1988.

[Same89a] H Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, **46**(2):367–386, 1989.

[Same89b] H Samet and R E Webber. A comparison of the space requirements of mult-dimensional quadtree-based file structures. *The Visual Computer*, **5**(6):349–359, 1989.

[Sedg88] R Sedgewick. *Algorithms.* Addison-Wesley, Massachusetts, second edition, 1988.

[Seeb62] R R Seeber and A B Lindquist. Associative memory with ordered retrieval. *IBM J. Res. Dev.*, **6**(1):126–136, 1962.

[Shaf87] C A Shaffer and H Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, **37**(3):402–419, 1987.

[Slad56] A Slade and H O McMahon. A cryotron catalog memory system. In *Proc. EJCC*, 115–120, 1956.

[Snyd87] J M Snyder and A H Barr. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH'87)*, **21**(4):119–128, 1987.

[Srih81] S N Srihari. Representation of three-dimensional digital images. *ACM Computing Surveys*, **13**(4):399–424, 1981.

[Stew86] I P Stewart. Quadtrees: storage and scan conversion. *The Computer Journal*, **29**(1):60–75, 1986.

[Tamm81] M Tamminen. *The EXELL Method for Efficient Geometric Access to Data.* PhD thesis, Helsinki University of Technology, ACTA Polytechnica Scandinavica, 1981.

[Tamm84a] M Tamminen, O Karonen and M Mantyla. Ray-casting and block model conversion using a spacial index. *Computer-Aided Design*, **16**(4):203–208, 1984.

[Tamm84b] M Tamminen and H Samet. Efficient octree conversion by connectivity labeling. *Computer Graphics (SIGGRAPH'84)*, **18**(3):43–51, 1984.

[Tang88] Zesheng Tang and Lu Shengkai. A new algorithm for converting boundary representation to octrees. In *Proceedings of EUROGRAPHICS'88*, North-Holland, 105–116, 1988.

[Toth85] D L Toth. On ray tracing parametric surfaces. *Computer Graphics (SIGGRAPH'85)*, **19**(3):171–179, 1985.

[Ulln83] M K Ullner. *Parallel Machines for Computer Graphics.* PhD thesis, California Institute of Technology, USA, 1983.

[Veen85] J Veenstra and N Ahuja. Deriving object octree from images. In *Lecture Notes in Computer Science 206*, pages 196–211, 1985.

[Voel88] H B Voelcker, A A G Requicha and R W Conway. Computer applications in manifacturing. *Annual Review of Computer Science*, **3**:349–387, 1988.

[Wade87] J P Wade and C G Sodini. Dynamic cross-coupled bitline content addressable memory cell for high density arrays. *IEEE Journal of Solid-State Circuits*, **SC-22**(1):119–121, 1987.

[Wegh84] H Weghorst, G Hooper and D P Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, **3**(1):52–69, 1984.

[Whit80] T Whitted. An improved illumination model for shaded display. *Communications of the ACM*, **23**(6):343–349, 1980.

[Will88a] R D Williams. The Goblin quadtree. *The Computer Journal*, **31**(4):358–363, 1988.

[Will88b] R D Williams. *Organization and Analysis of Spatial Data.* PhD thesis, University of Camridge, Cambridge, 1988.

[Wyvi86] G Wyvill, T L Kunii and Y Shirai. Space division for ray tracing in CSG. *IEEE Computer Graphics and Applications*, **6**(4):28–34, 1986.

[Yama84] K Yamaguchi, T L Kunii, K Fujimura and H Toriya. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, **4**(1):53–59, 1984.

[Yau83] M Yau and S N Shihari. A hierarchical data structure for multidimensional digital images. *Communications of the ACM*, **26**(7):504–515, 1983.

[Yau84] M Yau. Generating quadtrees of cross sections from octrees. *Computer Vision, Graphics, and Image Processing*, **27**(2):211–283, 1984.