GAZİANTEP UNIVERSITY GRADUATE
SCHOOL OF NATURAL & APPLIED SCIENCES

DEVELOPMENT OF AN ALGORITHM FOR VOXELIZATION OF CSG (CONSTRUCTIVE SOLID GEOMETRY) OBJECTS USING A 2D CSG PROCESSOR

Ph.D THESIS
IN
ELECRICAL AND ELECTRONICS ENGINEERING

BY
SEMA KOÇ
MAY 2005

Development of An Algorithm for Voxelization of CSG (Constructive Solid Geometry) Objects Using a 2D CSG Processor

PhD Thesis
in
Elecrical and Electronics Engineering
University of Gaziantep

Supervisor
Assoc. Prof. Dr. Savaş UÇKUN

Co-Supervisor
Assist. Prof. Dr. Ulus ÇEVİK

by

Sema KOÇ

May 2005

ABSTRACT

DEVELOPMENT OF AN ALGORITHM FOR VOXELIZATION OF CSG (CONSTRUCTIVE SOLID GEOMETRY) OBJECTS USING A 2D CSG PROCESSOR

KOÇ, Sema

Ph. D. In Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Savaş UÇKUN

Co-Supervisor: Assist. Prof. Dr. Ulus ÇEVİK

May 2005, 106 Pages

In this thesis we present a new approach for the voxelization of volumetric scene graph. The voxelization algorithm is based on the creation of volume slices using the depth information of front and back surfaces belonging to scene primitives. The algorithm generates slices of the each primitive intended to be voxelized using 2D CSG processor.

For the volume scene tree "Blist (Boolean list)" representation is used. In Blist formulation, a Boolean expression is represented as a list of primitives instead of a tree, and this may be evaluated in a pipeline fashion, combining at each step the result of classifying the cells against the current primitive with the result of the previous classification. The fundamental breakthrough provided here lies in the fact that the result of the previous classifications does not require the list of values of cell-primitive classification results, nor a stack of intermediate results of evaluating sub-expressions. Instead, Blist passes from one primitive to the next a simple label, which may be stored using at most log(H+1) bits, where H is the height of the CSG tree .

Where this study differs from previous works is that it provides the following: the z-buffer based voxelization algorithm can voxelize different type of objects (convex and concave objects, polygons, lines and surfaces), also the algorithm is suitable for accurately voxelizing objects with hidden cavities. Using Blist representation, the volume scene tree expression can be evaluated without using recursion or stack.

Keywords: volume scene tree, Blist, voxelization, CSG (Constructive solid geometry), z-buffer.

ÖZ

2 BOYUTLU CSG(YAPISAL KATI GEOMETRİ) İŞLEMCİSİ KULLANARAK CSG NESNELERİNİN VOKSELİZASYONU İÇİN BİR ALGORİTMANIN GELİŞTİRİLMESİ

KOÇ, Sema

Ph. D. In Electrical and Electronics Engineering

Tez Yöneticisi: Doç. Dr. Savaş UÇKUN

Yardımcı Tez Yöneticisi: Yar. Doç. Dr. Ulus ÇEVİK

May 2005, 106 sayfa

Bu tezde hacimsel sahnenin vokselizasyonu için yeni bir yaklaşım önerilmektedir. Bu vokselizasyon algoritması ön ve arka yüzeylerin derinlik bilgilerini kullanarak hacimsel kesitlerin oluşturulması ilkesine dayanmaktadır. Algoritma vokselizasyonu yapılacak olan her bir nesnenin kesitlerini 2 boyutlu CSG işlemcisi kullanarak oluşturmaktadır.

Hacimsel sahne ağacı için Blist (Boolean list) gösterimi kullanılmaktadır. Blist formülasyonunda Boolean açılımı, ağaç yerine sahne temel öğelerinin listesi olarak gösterilmektedir ki bu da her bir vokselin sahne öğelerine karşı sınıflandırmasının sonucu ile bir önceki sınıflandırmanın sonucunun birleştirilmesi ile ardışıl olarak hesaplanabilir. Burada önemli olan nokta bir önceki sınıflandırma, ne hücre-öğe sınıflandırması sonuç listesine ne de alt açılımların hesaplanması için bir ara yığına ihtiyaç duymaktadır. Bunun yerine, Blist metodu bir öğeden diğerine sadece bir etiket kullanarak geçebilmektedir ve bu etiket değerini saklamak için sadece log(H+1) bit yeterli olmaktadır. Burada H, CSG ağacının yüksekliğidir.

Bu çalışma şu yönleri ile önceki çalışmalardan ayrılmaktadır.

Z-arabelleği temelli vokselizasyon algoritması farklı cisimlerin (iç-bükey, dış-bükey cisimler, poligonlar, çizgi ve yüzeyler) vokselizasyonu için kullanılabilinir ve ayrıca algoritma saklı kovuklu cisimlerin doğru bir şekilde vokselizasyonu için uygundur. Blist gösterimi kullanarak hacimsel sahne grafiği yığın veya özyineleme kullanmadan hesaplanabilinir.

Keywords: Hacimsel sahne ağacı, Blist, vokselizasyon, CSG (yapısal katı geometri), Z-arabelleği.

ACKNOWLEDGEMENTS

I would like to express my sincerest appreciation to my supervisor Assist. Prof. Dr. Ulus Çevik for his guidance, suggestions, valuable criticisms, great help and encouragement in the preparation of this work.

I would also like to thank Assoc. Prof Dr. Savaş Uçkun for undertaking the responsibility of being my official supervisor at Gaziantep University.

I must also thank my parents, especially mum, for everything they have done for me.

Finally, special thanks go to my husband Sabit Kayhan for his love and strength throughout my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

In recent years, the ability to integrate and process volumetric information has become increasingly desirable due to rapid advances in volume graphics [1, 2] and volume visualization [3], and proliferation of 3D data acquisition techniques.

Volume data are 3D entities that may have information inside them, might not consist of surfaces and edges, or might be too voluminous to be represented geometrically. Volume data are obtained by sampling, simulation, or modeling techniques. For example, a sequence of 2D slices obtained from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) is 3D reconstructed into a volume model and visualized for diagnostic purposes or for planning of treatment or surgery. The same technology is often used with industrial CT for non-destructive inspection of composite materials or mechanical parts. Similarly, confocal microscopes produce data, which is visualized to study the morphology of biological structures. In many computational fields, such as in computational fluid dynamics, the results of simulation typically running on a supercomputer are often visualized as volume data for analysis and verification. Recently, many traditional geometric computer graphics applications, such as CAD and simulation, have been exploiting the advantages of volume techniques called *volume graphics* for modeling, manipulation, and visualization.

Volume graphics is concerned with the modeling, manipulation and display of synthetic geometric objects using volumetric representations and volume rendering techniques. This allows geometric objects to be intermixed with volume data sets under a uniform volume representation scheme. In this process, the 3D scene is pre-converted to a volume representation using a voxelization algorithm before volumetric modeling and rendering techniques are applied.

Volume visualization is a method of extracting meaningful information from volumetric data using interactive graphics and imaging, and it is concerned with volume data representation, modeling, manipulation, and rendering [3].

Over the years many techniques have been developed to visualize 3D data. Since methods for displaying geometric primitives were already well-established, most of the early methods involve approximating a surface contained within the data using geometric primitives. When volumetric data are visualized using a surface rendering technique, a dimension of information is essentially lost. In response to this, volume rendering techniques were developed that attempt to capture the entire 3D data in a single 2D image. Volume rendering convey more information than surface rendering images, but at the cost of increased algorithm complexity, and consequently increased rendering times. To improve interactivity in volume rendering, many optimization methods as well as several special-purpose volume rendering machines have been developed [4-5].

1.2 3D Data Acquisition and Volumetric Data

3D data acquisition techniques are often the main data sources in many application fields. Biomedical scanning devices, such as CT, MRI, and confocal microscopes, produces large volumes of biomedical image data that may be used by a wide range of applications such as computational biology, medical diagnosis, surgical simulation and biomedical education. High-resolution remote sensing and geospatial sensors can generate very large scale and multi-layered geospatial datasets in a variety of different forms. Many different types of 3D datasets are also generated from scientific simulations and experiments.

Volumetric data is typically a set S of samples (x, y, z, v), representing the value v of some property of the data, at a 3D location (x, y, z). If the value is simply a 0 or a 1, with a value of 0 indicating background and a value of 1 indicating the object, then the data is referred to as binary data. The data may instead be multivalued, with the value representing some measurable property of the data, including, for example, color, density, heat or pressure. The value v may even be a vector, representing, for example, velocity at each location.

In general, the samples may be taken at purely random locations in space, but in most cases the set *S* is isotropic containing samples taken at regularly spaced intervals along three orthogonal axes. When the spacing between samples along each axis is a constant, but there may be three different spacing constants for the three axes the set *S* is anisotropic. Since the set of samples is defined on a regular grid, a 3D array (called also *volume buffer, cubic frame buffer, or 3D raster*) is typically used to store the values, with the element location indicating position of the sample on the grid. For this reason, the set *S* will be referred to as the array of values $S(x, y, z)$, which is defined only at grid locations. Alternatively, rectilinear, curvilinear (structured), or unstructured grids are employed (e.g., [6]). In a *rectilinear* grid the cells are axis-aligned, but grid spacing along the axes are arbitrary. When such a grid has been non-linearly transformed while preserving the grid topology, the grid becomes *curvilinear*. Usually, the rectilinear grid defining the logical organization is called *computational space,* and the curvilinear grid is called *physical space*. Otherwise the grid is called *unstructured* or *irregular*. An unstructured or irregular volume data is a collection of cells whose connectivity has to be specified explicitly. These cells can be of an arbitrary shape such as tetrahedral, hexahedra, or prisms. The array *S* only defines the value of some measured property of the data at discrete locations in space.

A function $f\,(x, y, z)$ may be defined over $R3$ in order to describe the value at any continuous location.

The function $f\,(x, y, z) = S(x, y, z)$ if (x, y, z) is a grid location; otherwise $f\,(x, y, z)$ approximates the sample value at a location (x, y, z) by applying some interpolation function to *S*. There are many possible interpolation functions. The simplest interpolation function is known as *zero-order interpolation*, which is actually just a nearest-neighbor function. The value at any location in $R3$ is simply the value of the closest sample to that location. With this interpolation method there is a region of constant value around each sample in *S*. Since the samples in *S* are regularly spaced, each region is of uniform size and shape. The region of constant value that surrounds each sample is known as a *voxel* with each voxel being rectangular cuboids having six faces, twelve edges, and eight corners.

1.3 Volume Graphics

Volume graphics is concerned with the graphics defined by volume data. More specially, it is concerned with the synthesis, modeling, manipulation, and rendering of volumetric objects, stored in a volume buffer, which is actually an area that holds a 3D grid of point samples of a volume space. Volume visualization focuses on sampled and computed datasets in large-scale data processing applications such as biomedical, geographic and scientific applications. More importantly, volume-based graphics offers a consistent solution to the primary deficiency of the traditional surface-based graphics, including its inability to encapsulate the international description of a model and the difficulties in representing and rendering amorphous phenomena (e.g. cloud, gas, fluid, etc.).

Volume graphics also proposes to extend raster graphics into 3D by extending a 2D frame buffer [7] into a 3D volume buffer. The fundamental technique, upon which a volume graphics system built, is the real-time rendering volume data from the volume buffer to a 2D frame buffer displayed on the screen.

1.3.1 From Vector Graphics to Raster Graphics

The display of graphics in the sixties and seventies was based on vector drawing devices an object-based approach to scene representation, manipulation, and display. A geometric representation of the objects comprising the scene was stored in a display-list. Refreshing the screen was accomplished by redrawing the vectors comprising the objects in the display-list. The major advantages of vector graphics were its ability to perform object related operations on the display-list and the fact that the vectors it drew were continuous and thus exhibited no aliasing. This technology, however, offered calligraphic drawing only, while the interior shaded areas were extremely hard to render.

The alternative approach, termed *raster graphics,* has been predominant since the late seventies. Raster graphics utilizes a 2D frame-buffer (a raster) of pixels for scene representation and a point-based renderer for coloring those pixels that correspond to the discrete representation of the geometric objects. Screen refresh is performed by a video controller, which repeatedly displays the frame-buffer onto the screen [8].

Table 1 Comparison between vector graphics and raster graphics and between surface graphics and volume graphics.

2D	Vector Graphics	Raster Graphics
Scene/object complexity	-	+
Block operations	-	+
Sampled data	-	+
Interior	-	+
Memory and processing	+	-
Aliasing	+	-
Transformations	+	-
Objects	+	-
3D	Surface Graphics	Volume Graphics

Table 1 compares vector graphics with raster graphics. Unlike vector graphics, raster graphics provides the capability to present realistic, shaded, and textured surfaces in full color, as well as line drawings (row 9 in Table 1). The main disadvantages of this approach are the aliasing present in the image due to the discrete nature of the representation (row 6 and the large memory and processing power this approach requires (row 6). The latter two difficulties delayed the full acceptance of raster graphics until the late seventies when the technology was able to provide cheaper and faster memory and hardware to support the demands of the raster approach. In addition, the discrete nature of rasters makes them less suitable for geometric operations such as transformations (row 7) and accurate measurements On the other hand, a main appeal of raster graphics is that it decouples image generation from screen refresh, thus making the refresh task insensitive to the scene complexity (row 1). In addition, the raster representation lends itself to block operations, such as *bitblt* (bit block-transfer) in which a window or a rectangular block of pixels can be rapidly transferred with variety of pixel-by-pixel operations between the source and destination blocks (row 7) [8]. Raster graphics is also suitable for displaying 2D sampled digital images, and thus provides the ideal environment for mixing digital images with synthetic graphics (row 8). These advantages, coupled with advances in hardware and the development of antialiasing methods, have led raster graphics to replace vector graphics as the primary technology for computer graphics.

1.3.2 From Surface Graphics to Volume Graphics

The object-based approach of vector graphics has been adapted for 3D graphics at the expense of maintaining and manipulating a display-list of geometric objects and regenerating the frame-buffer after every change in the scene or viewing parameters. This approach, termed s*urface graphics*, combines raster technology for the display and an object-based approach for the representation, manipulation and rendering of 3D scenes. This method is supported by powerful *geometry engines*, which constitute the present hardware for polygon rendering. These have flourished in the past decade, making surface graphics the state-of-the-art in 3D graphics [8].

Surface graphics strikingly resembles vector graphics in many ways. Like vector graphics, surface graphics represents the scene as a set of geometric primitives kept in a display-list. These primitives are transformed, mapped to the screen coordinates, and converted by *scan-conversion* algorithms into a discrete set of pixels, which is stored in the frame-buffer. This digitization process is also called *rasterization* or *pixelization*. Any change to the scene, viewing parameters, or shading parameters requires the image generation system to repeat this process and re-process the complete scene description. Surface graphics generates merely the surfaces of 3D solid objects viewed from a given direction, and subject to limitations similar to those of vector graphics, it does not support the rendering of the interior of these 3D objects.

Instead of a list of geometric objects, volume graphics employs a 3D volume buffer as a medium for the representation and manipulation of 3D scenes. A 3D scene is discretized earlier in the image generation pipeline, and the resulting 3D discrete form is used as a database of the scene for manipulation and rendering purposes, which in effect decouples discretization from rendering (viewing and shading). Furthermore, all objects are converted into one unit form meta-object – the voxel. Each voxel is atomic and represents the information about at most one object that resides in that voxel.

Volume graphics offers the same benefits as surface graphics, with several advantages that are due to the decoupling and uniformity features. The rendering phase is viewpoint independent and insensitive to scene complexity and object complexity. It supports Boolean and block operations and constructive solid modeling. When 3D sampled and simulated data is available, such as that generated by medical scanners (e.g., CT, MRI) or scientific simulations (e.g., CFD), volume graphic is suitable for their representation. It is capable of representing amorphous phenomena and it can have information on both the interior and exterior of 3D objects. Several disadvantages of this approach are related to the discrete nature of the representation, namely, that transformations and shading are performed in discrete space. In addition, this approach requires substantial amounts of storage space and specialized processing.

The same appeal that drove the evolution of the computer graphics world from vector graphics to raster graphics, once the memory and processing power became available, is starting to drive a variety of applications from surface-based representation of 3D scenes to voxel-based representation. Naturally, this trend first appeared in applications involving sampled 3D data, such as medicine and scientific visualization, in which the datasets are in volumetric form. The diverse empirical imagery applications of volume visualization still provide a major driving force for advances in volume graphics.

1.3.3 Weaknesses of Volume Graphics

Discrete Form Unlike surface graphics, in volume graphics the 3D scene is represented in discrete form. This is the cause of many of the maladies of voxel-based graphics, which are similar to those of 2D rasters [9]. The finite resolution of the raster poses a limit on the accuracy of some operations, such as volume and area measurements, that are based on voxel counting (row 9 in Table 2). Manipulation and transformation of the discrete volume are difficult to achieve without degrading the image quality or losing some information (row 4). Rotation of raster by angles other than 90 degrees is especially problematic since a sequence of consecutive rotations will distort the image.

Since the continuous object is reconstructed by sampling the discrete data during rendering, a low resolution volume yields high aliasing artifacts (row 3 in Table 2). This becomes especially apparent when zooming in on the 3D raster. When naive rendering algorithms are used, the 3D discrete points may appear to be parted from each other, and may cause the appearance of holes. Nevertheless, this can be alleviated to some extent in ways similar to those adopted by 2D raster graphics, such as employing either reconstruction techniques (e.g. supersampling, filtering) or a high-resolution volume buffer.

Loss of Geometric Information In volume graphics we allow each voxel to maintain only local information pertaining to the volume unit it represents. After a surface object has been voxelized, the voxels comprising the discrete object do not retain any geometric information regarding the surface definition of the object. Thus, it is advantageous, when exact measurements are required (e.g., distance, volume, area), to employ surface-based modeling where the geometric surface definition of the object is available. A voxel-based object is only a discrete approximation of the original continuous object where the volume buffer resolution determines the precision of such measurements. On the other hand, several measurement types are more easily computed in voxel space (e.g., mass property, adjacency detection, and volume computation) (row 9 in Table 2). The lack of geometric information in the voxel may inflict other difficulties, such as those encountered when rendering discrete surfaces. An essential requirement for most shading methods is the ability to calculate the normal vector to the surfaces comprising the 3D scene [5].

In traditional surface graphics, normal vectors are either analytically calculated from the surface representation or stored as part of the surface representation. In voxel-based models discrete shading method is employed to estimate the normal from a context of voxels. A variety of image-based and object-based methods for normal estimation from volumetric data based on fitting some type of a surface primitive to a small neighborhood of voxels.

A partial integration between surface and volume graphics is conceivable as part of an object-based approach in which an auxiliary object table, consisting of the geometric definition and global attributes of each object, is maintained in addition to the volume buffer. Each voxel consists of only an index to the object table, allowing exact calculation of normal exact measurements, and intersection verification for discrete ray tracing. The auxiliary geometric information might be useful for re-voxelizing the scene in case of a change in the scene itself.

Table 2 A comparison between vector graphics and raster graphics.

	Vector Graphics	Raster Graphics
Rendering and Screen refresh	Rendering is embedded in screen refresh	scan-conversion is decoupled from screen-refresh
Rendering performance	Sensitive to scene and object complexity	Insensitive to scene and object complexity
Memory and Processing requirements	Variable depends on scene and object complexity	Large but constant
Screen space aliasing	Nonexistent	Frequent
Transformation	Continuous, performed on the geometric definition of objects	Discrete, performed on pixel blocks (windows)
Boolean block Operations	Difficult, must be performed analytically	Trivial, by employing bitblt, pixel-by-pixel operation, aggregation, quadtrees
Capable of rendering Interior	No, boundary only	Yes, colored, shaded and textured surfaces
Adequacy for sampled Digital imaes	No	Yes
Measurements(e.g. distance, area)	Analytical, but often complex	Discrete approximation, but simple

Memory and Processing A typical volume buffer occupies a large amount of memory; for example, for a moderate resolution of 512^3 the volume buffer consists of more than 10^8 voxels. Even if we allocate only one byte per voxel, 128 M bytes will be required (row 2 in Table 2). However, since computer memories are significantly decreasing in price and increasing in their compactness and speed, such large memories are becoming more and more feasible. This argument echoes similar discussion when raster graphics emerged as a technology in the mid-seventies. With the rapid progress in memory price and compactness, it is safe to predict that, as in the case of raster graphics, the memory will soon cease to be a stumbling block for volume graphics.

Yet, the extremely large throughput that has to be handled requires a special architecture and processing attention. *Volume engines*, analogues to the currently available geometry engines, are emerging. Because of the presortedness of the volume buffer and the fact that only a single type of object – *the voxel* – has to be handled, volume engines are conceptually simpler to implement than current geometry engines.

1.3.4 Advantages of Volume Graphics

Insensitivity to Scene Complexity One of the most appealing attributes of volume graphics is its insensitivity to the complexity of the scene, since all objects have been pre-converted into a finite size volume buffer (row Table 2). Although the performance of the voxelization phase is influenced by the scene complexity, rendering performance depends mainly on the constant resolution of the volume buffer and not on the number of objects in the scene. This is in contrast to representing the volume with an octree whose size varies according to the scene complexity [10]. Insensitivity to the scene complexity makes the volumetric approach especially attractive for scenes consisting of a large number of objects, such as those generated by fractal systems. Another example of such a scene is a curved surface represented by a large polygon mesh that is generated by a polyhedral smoothing or fitting algorithm. A polygon mesh can approximate a curved surface, where the approximation precision and presentation quality increase with the number of polygons in the mesh. However, using a very fine mesh in conventional surface graphics is expensive with respect to space and display time.

Insensitivity to Object Complexity In volume graphics, rendering (viewing and shading) is decoupled from digitization (voxelization) and all objects are first converted into one meta object, the voxel, which makes the rendering process insensitive to the complexity of the objects (row 5 in Table 2). Thus, volume graphics is particularly attractive for objects that are hard to render using conventional graphics systems. Examples of such objects include curved surfaces of high order and fractals, which require the expensive computation of an iterative function for each volume unit [11]. Constructive solid models are also hard to render by conventional methods but are straightforward to render in volumetric representation.

Another type of object complexity involves objects that are enhanced with a technique know as *texture-mapping*, where the realism of objects is increased by simulating surface details by texture-mapping is commonly implemented during the last stage of the rendering pipeline where the texture is extracted from a 2D texture image and mapped onto the surface to be rendered, and its complexity is proportional to the object complexity [8]. In volume graphics, texture-mapping is performed only once, during the voxelization stage, where the texture color is calculated and stored in each voxel. Solid texturing, which employs a 3D texture image, has also a high complexity similar to texture-mapping [8]. In volume graphics however, solid texturing, like texture-mapping, is performed during the voxelization stage.

Viewpoint Independence A main difference between voxel-based graphics and conventional surface graphics is that in the former the scene is discretized (voxelized) once for multiple viewing conditions, while the latter the scene is repeatedly scan-converted after every change in the viewing parameters, causing a performance bottleneck in its rendering pipeline (row 10 in Table 2). This attractive advantage of volume graphics can be attributed in part to the fact that, in the volumetric representation, a unit of memory is allocated for each unit of space, in contrast to surface graphics, where memory is assigned only to complete surface patches. This enables volume graphics to store view independent attributes at each volume unit, while surface graphics is not able to provide storage for attributes that vary across its basic surface elements.

In anticipation of repeated access to the volume buffer (such as in animation), all viewpoint independent attributes can be precomputed during the voxelization stage, stored with the voxel, and be readily accessible for speeding up the rendering. The voxelization algorithm can generate for each object voxel its color, its texture color, its normal vector (for visible voxels), and information concerning the visibility of the light sources from that voxel. Actually, the viewpoint independent parts of the illumination equation, that is, the ambient illumination and the sum of the attenuated diffuse illumination of all the visible light sources [8], [4] can also be precomputed and stored as part of the voxel value.

Once a volume buffer with precomputed view-independent attributes is available, a rendering algorithm such as a *discrete ray tracing* algorithm can be engaged. Discrete ray tracing is based on traversing 3D discrete rays through the volume buffer. The discrete ray tracing approach is especially attractive for ray tracing complex surface scenes and constructive solid models, as well as 3D sampled and computed datasets (see below). In spite of the complexity of these scenes ray tracing time was approximately the same as for much simpler scenes and significantly superior to traditional space-subdivision ray tracing methods. Moreover, in spite of the discrete nature of the volume buffer representation, images indistinguishable from the ones produced by conventional surface-based ray tracing can be generated by employing auxiliary object tables and screen supersampling techniques, which casts several rays per pixel.

Sampled and Simulated Datasets Sampled datasets and simulated datasets (such as in computational fluid dynamics) are often reconstructed from the acquired samples of simulated points into a regular grid of voxels and stored in a volume buffer. Such datasets provide for the majority of applications using the volumetric approach. Unlike surface graphics, volume graphics naturally and directly supports the representation, manipulation, and rendering of such datasets (row 7 in Table 2), as well as provides the volume buffer medium or intermixing sampled or simulated datasets with geometric objects (row 8) [12].

I*nner Information* A central feature of volumetric representation is that, unlike surface representation, it is capable of representing inner structures of the objects, which can be revealed and explored with the appropriate manipulation and rendering techniques (row 7 in Table 2). Natural objects as well as synthetic objects are likely to be solid rather than hollow. The inner structure is thus an important aspect of image complexity, which is easily, explored using volume graphics and cannot be supported by surface graphics. Moreover, while translucent object can be represented by surface methods, these methods cannot efficiently support the modeling and rendering of amorphous phenomena (e.g., clouds, fire, smoke) that are volumetric in nature and do not have any notion of tangible surfaces [13,14,15].

Block Operations An intrinsic characteristic of rasters is that adjacent objects in the scene are also represented by neighboring memory cells. Therefore, rasters lend themselves to various meaningful grouping-based operations, such as *bitblt* (bit block-transfer) operations, or its 3D counterpart, *voxblt* (voxel block-transfer) operations, which support transfer of cuboidal voxel blocks with variety of voxel-by-voxel operations between source and destination blocks (row 6 in Table 2) [16]. Such block operations add a variety of modeling capabilities, which aid in the task of image synthesis. Moreover, the volume buffer lends itself to Boolean operations that can be performed on a voxel-by-voxel basis during the voxelization stage. This property is very advantageous when Constructive Solid Geometry (CSG) is the modeling paradigm. CSG operations such as subtraction, union, and intersection between two voxelized objects are accomplished at the voxel level [10], thereby reducing the original problem of evaluating a CSG tree of such operations during rendering time down to a 1D Boolean operation between pairs of voxels during a preprocessing stage. Once a CSG model has been constructed in voxel representation, it is rendered like any other volume buffer. This makes discrete ray tracing of constructive solid models straightforward.

1.3.5 Scene Representation and Composition

In a volume space, the parts of many objects in the scene may be overlapping each other. That is, each point in a volume space can be occupied by many objects. So, such a point needs to be explicitly represented. Data transformation is the key technique in volumetric scene composition and manipulation. Volumetric scene composition can be described by scene expression, which constructs a volumetric scene from objects of heterogeneous representations using various blending and filtering functions. A popular technique in scene composition is scene graph. The scene manipulation can be implemented by volume rendering technology. As volume data is independent of specific spatial representations, volume rendering under the volume graphics paradigm is particularly suitable as unified rendering mechanism for a scene with heterogeneous representations. Using today’s technology, a simple volume graphics engine can be assembled using a standard PC with a real time volume rendering chip. However, since the volume buffer needs to be pre-computed by the application program, the rendering difficulties have shifted to the dynamic volume reconstruction (voxelization) over a complex volumetric scene representation. Thus, the two most fundamental issues in volumetric interactions using a volume graphics system are: the modeling and representation of a volumetric scene; and the scene voxelization algorithms (evaluation) that efficiently reconstruct the volume buffer over the current viewing region for each scene modification.

1.3.6 Volume Scene Graph

A popular technique in scene composition is scene graph. It organizes geometric objects and their rendering parameters in a tree-like hierarchical structure. Also, it allows sharing of substructures, which naturally forms a cyclic directed graph. Due to its flexibility of defining complex 3D scenes, scene graph has been widely used in many 3D graphics systems. Recently, this idea has been extended to volume scene graph for the composition of complex volumetric scenes from multiple volume datasets [17]. However, the evaluation of a volume scene graph is done in a brute-force manner, i.e. for each point in volume space, recursively computing the value of the scene graph starting from the root. This is a very expensive procedure, and can only be used as a preprocessing step, which is not practical for interactive applications.

A similar technique often used for 3D model composition is the Constructive Solid Geometry (CSG) representation. It allows users to define complex 3D solid objects by hierarchically combining simple geometric primitives using Boolean operations and affined transformations [18].

1.4 Constructive Solid Geometry

One of the most popular techniques often used for 3D model composition is the Constructive Solid Geometry (CSG) representation. CSG technique was originally introduced in solid modeling for computer aided design applications [19], and has also been widely used later in 3D computer graphics and animation [20-22]. A natural extension of the CSG method is to allow volume datasets to be included as primitives in the CSG construction process, as described independently in several publications [23-26].

Boolean operations are convenient for incrementally designing solids, for modeling the geometric effects of certain manufacturing operations, and for expressing interferences between parts to be assembled. A user's specification that combines simple solid primitives through a sequence of Boolean operations yields Boolean expressions, which, for further processing, may be parsed and stored as a binary tree, called the CSG representation of solid (In Figure 1.1 the text input (left) defines two rectangles, A and B of different dimensions, both centered at the origin; two other rectangles, C and D, obtained by rotating copies of B; and a Boolean expression S that combines all four rectangles. Parsing the Boolean expression according to the precedence rules of Boolean operators yields a binary tree (center), which corresponds to a 2-D region (right).)

Some modelers, for example [27-29], represent solids by N-way trees, where each node may have more than two children. The leaves of the CSG tree represent instances of simple parameterized primitive solids, such as blocks, cylinders, spheres, and cones in 3-D, or rectangles and disks in 2D. Primitives are often modeled mathematically as intersection of half spaces. The internal nodes of the CSG tree represent potentially more elaborate regular sets and are associated with regularized Boolean operators: union, intersection, and difference, respectively denoted +, ., and -.

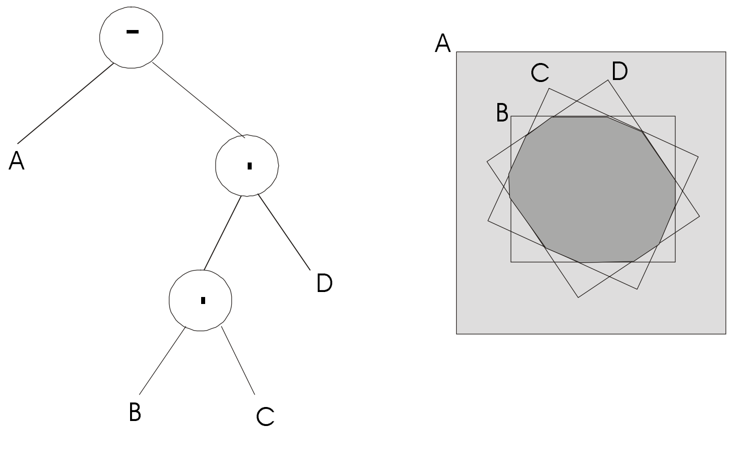


Figure 1.1 CSG representation**.**

1.4.1 General Research Issues

Guaranteed validity is one of the major advantages of CSG: It enables CSG representations to be parameterized, composed almost arbitrarily, edited interactively without danger of producing an invalid model (a representation corresponding to a “nonsense object”). CSG has also disadvantages. We list below some perceived problem areas together with examples of pertinent research:

Practical implementation of CSG restricts the represented objects to be semi algebraic sets specified as regularized Boolean combinations of algebraic inequalities. It is often inconvenient, and sometimes impossible, to specify certain well-defined operations (such as offsetting and blending) in these terms. Rossignac and Requicha added constant radius blending [30] and offsetting operations [31] to the CSG scheme. Shirma et al. [32] studied tools for modeling Boolean combinations of sweeps, and [33] developed tools for displaying them.

The design in CSG of complex 3D solids requires that the users produce sequences of rotations and translations that will correctly position the solid primitives. This task is particularly hard when it is necessary to achieve complex 3D arrangements of primitives bounded by curved surfaces. Rossignac [34] and Anderson [35] incorporated constraint-based specifications in the CSG representation, and Peterson [36] and Vossler [37] described techniques for converting 2-D contours into CSG representations of 2D regions and swept volumes.

Most CSG-based systems were developed for representing and manipulating homogeneously 2D regions or 3D solids. Certain CAD/CAM applications, however, deal with multi-dimensional collections of open geometric elements. Cameron [38] used 4D CSG representations to compute intersections of moving objects, and Rossignac [39] proposed algorithms for performing Boolean operations on nonhomogeneous, n-dimensional sets.

A CSG representation of a given solid is not unique and thus, it is expensive to determine whether two CSG trees represent the same solid and whether any particular CSG tree represents the empty set. Tilove [40, 41] dealt with efficient methods for comparing sets represented by two different CSG trees, and Cameron [42] proposed improvements to some of Tilove’s results. Woodwark [43] described a different method, which simplifies CSG representations by reasoning on geometric approximations of combinations of half spaces.

A CSG representation, although informationally complete, does not contain explicit information about the boundary of the represented solid. Some applications, such as mass-property calculation [44] and null-object detection [40, 45], do not require such information and may be said to operate directly on CSG. Other applications require that some form of partial or complete boundary evaluation be done. Requicha and Voelcker [46] described the basic techniques for boundary evaluation, which is inherently computationally intensive. The generation of shaded pictures from CSG has received a great deal of attention; many references can be found in [31].

The performance of boundary evaluation and other CSG-based algorithms can be improved by reducing redundant computation. The essential notion may be summarized as follows:

Compute only entities and over regions of space that can affect the desired final result.

Improve the Boundary Evaluation Algorithm**.** For a solid defined in CSG, boundary evaluation algorithms compute intersection of the primitive's boundaries with solid's boundary. Typically, candidate elements (faces, edges, points, that are subsets of primitives' boundary) are classified against all primitives, and the results combined according to the Boolean expression that defines the solid. However, the classification of particular candidate element with respect to certain primitives or subtrees may be irrelevant to the final result. Such redundant computation may be easily avoided in simple cases. (e.g. given an element X and two primitives A and B, if X not Є A, then there is no need to compute whether XЄB in order to determine that X ЄA.B)

Eliminate Redundant Primitives. A CSG representation may contain many redundant primitives or half spaces that can be eliminated without altering the represented solid. The detection of such redundancies is relatively expensive, but in some applications it provides a one-time processing step that enables all subsequent calculations to be speeded up. Note that the redundancy of primitives cannot be calculated by performing boundary evaluation and keeping track, for each resulting boundary element, of the primitives it lies on. The active zone algorithm [47] leads to improvements of the performance of redundancy-detection algorithms.

1.4.2 CSG Tree Conventions and Terminology

Although a given set may have infinitely many CSG representations, throughout this thesis that to each set S is attached a particular CSG tree, called “the tree of S” or “the CSG representation of S” each algebraic representation of the Boolean function associated with a CSG tree by its positive form, which uses only regularized + and . operators, but represents rigorously the same Boolean function and thus the same solid as the original tree.

The tree regularized operators, union, intersection, and complement, form a Boolean algebra over regular sets. Therefore, De Morgan’s laws [48] can be invoked to manipulate CSG expressions.

A CSG tree may be converted into its positive form by a preorder traversal of the tree applying, when appropriate, the following transformations to each node:

$$X\text{-}Y = X.\overline{Y}$$

$$\overline{X+Y} = \overline{X}.\overline{Y}$$

$$\overline{X.Y} = \overline{X} + \overline{Y}$$

$$\overline{\overline{X}} = X$$

This reformulation exchanges the . and + operators at negative internal nodes, replaces – operators by . if the corresponding node is positive and by + otherwise, and replaces negative primitives by their regularized complements. The resulting positive form contains only . and + operators. In a positive form, all nodes are positive, but may represent unbounded regular sets. However, the set represented by the entire positive form is equal to solid represented by the original CSG tree.

1.5 Voxelization

Voxelization is the process of generating volume dataset from a geometric model, and is a necessary step in the volume graphics pipeline as proposed in [1]. Thus, voxelization is used as a pre-processing step in the current volume graphics paradigm. A major drawback of this paradigm is its limited support for dynamic scenes and interactive applications. A volume representation is often not suitable for or capable of providing many sophisticated object manipulation and modeling operations needed for applications such as computer aided design, animation and surgical simulation. Each object's original representation (geometric or volumetric) is generally more convenient, accurate, and cost-effective. In a computer-aided design system, for instance, complex geometric modeling operations have to be interactively applied to the geometric representations. The voxelization process, therefore, needs to be done on the fly after each change of the model for volume rendering and other volume related applications such as layered manufacturing and finite element analysis. The ability to work with both geometric and volumetric representations also allows some part of the 3D models to be directly defined from scanned volume data to produce alternative 3D models that are often more accurate and realistic.

To support dynamic scenes and interactive applications, an interactive volume graphics paradigm can be employed. In this approach, 3D scenes are modeled and manipulated in their own representations, and the volume techniques are only applied to the desired regions (e.g. the viewing regions) of the scene when needed. This requests fast voxelization algorithms that are able to generate volume representations of regions of interest on the fly for complex 3D scenes involving both geometric and volumetric objects.

Conceptually, voxelization is a set membership classification problem for all voxels in a volume against the given 3D model. In addition, from the view of the process, voxelization is a 3D scan conversion [49] process.

1.5.1 Voxelization Techniques

Voxelization is essentially a sampling process, and therefore sampling theory rules should be taken into account. The first voxelization algorithms were binary, assigning, for example, 1 to occupied voxels and 0 to those unoccupied [3,50]. This approach totally ignored sampling theory, and consequently rendered pictures suffered from aliasing. This was predominantly related to the poor estimation of the surface normal vector. Techniques were proposed to improve the normal vector estimation by taking into account information from a larger neighborhood (e.g., contextual shading [51], context sensitive normal estimation [52], center-of-gravity shading [53]). Although the improvement was significant, none of the techniques yielded a normal vector precise enough for the simulation of such effects as reflection and refraction of light on an object surface. Better results were obtained by discrete ray tracing [54]. In this technique, in addition to estimating the normal from the discrete data, the normal was confirmed from the analytic object description, which is kept along with the voxel raster. Hohne and Bernstein [55] pointed out that shading of scanned objects could be significantly improved, if one takes advantage of "inaccuracy" of the 3D scanning device. Due to physical limitations, a point-spread function of the scanner is not the ideal dimensionless pulse, but rather a Gaussian like profile with a finite support. Therefore, it acts as a low-pass filter, suppressing high frequencies and blurring object edges. This is known as the partial volume effect (PVE). Using such data, realistic shading can be achieved when the normal is computed by means of a discrete gradient filter (e.g., by central differencing). The first "smoothed" objects were synthesized for the sake of algorithm testing. A value, proportional to the distance from a center of the test sphere, was stored in voxels near its surface [56]. Other test objects were obtained by simulating the PVE by computation of relative occupancy of each voxel, shared both by the object and background [57].

Techniques for voxelization of smooth objects, primarily aimed at visualization, were proposed later. We classify them into two categories: filtration and distance field techniques.

Filtration techniques solve the problem of aliasing by low-pass filtering of the object. Different filters were used: cone shaped Bartlett filter [58, 59], Gaussian [60] and oriented box filter (a 1D box filter perpendicular to object surface) [61- 63]. The continuous filtered function is subsequently sampled at grid locations. Assuming that the filter support is smaller than the object, its interior is represented by some "inside" density and background is assigned some "outside" density. There is, of course, a transition area, which smoothly blends the inside and outside densities in a thin layer around the surface. Thus, this volumetric representation is similar to the data obtained by a 3D scanning device.

Distance field techniques assign to all voxels of a scene their distance to the nearest surface point of the object [64-67]. Usually a certain value, typically 0, is assigned to the point on the surface. The distances are in general unbounded and the distance field of an object embodies the whole scene, which means that the traditional notion of spatially localized objects is violated. Object interior and background are usually distinguished by a different sign of the distance.

In recent years, a number of curve and surface voxelization algorithms have been proposed [50, 68- 72]. Broadly speaking, these algorithms aim to provide extensions of 2D scan conversion methods to a volumetric domain. This requires additional sampling in the third dimension for each scanline. The existing algorithms are object-type specific, i.e. different algorithms are needed to be employed for different types of objects (e.g. lines, circles, polygons, quadratic surfaces, etc.) leading to implementation difficulties for general volume graphics tools.

Solid object voxelization, on the other hand, has not been sufficiently studied. Voxelization of solid models is a much harder problem. This is because solid objects are normally represented by their boundary surfaces without explicit interior information, which would require an inside test for each voxel in the volume space. One of the few algorithms for solid objects is the one by Lee and Requicha [73] based on point classification.

Another common solid representation is the constructive solid geometry (CSG) method [18]. Graphics techniques for CSG models have been studied by many researchers. Most such efforts are focused on direct display of CSG objects [21, 22]. With volume graphics, the CSG method can also be extended to include volume data sets as CSG primitives. Such models are called volumetric CSG or (VCSG) models [24], and are potentially useful for combining geometric and volumetric objects in one common modeling environment. There are also a number of voxelization and volume rendering techniques for CSG and VCSG models. They include beam oriented voxelization algorithm [74], the volume sampling approach [23], the octree-based rendering algorithm [24], and the distance volume algorithm [75]. The common problem of these algorithms is that the volume reconstruction process is expensive and very slow.

In [17], the evaluation of volume scene graph is done in a brute-force manner, i.e. for each point in the volume space the value of the scene graph is computed recursively starting from the root. This is a very expensive procedure, and can only be used as a preprocessing step, which is not practical for interactive applications.

In [26], a hardware voxelization algorithm was described. Although this algorithm is fast for small-scale interactive applications, its performance is limited by the need of generating an intermediate object for each Boolean operation node, and a hardware restriction in blending function combination. It was improved by using a point classification map for Boolean operations based on a frame buffer color-encoding scheme [76]. But the algorithm only provides a binary volume, which may not provide complete information in volumetric space.

In [77], another volume pipeline is used in which each slice is applied to the entire CSG (Constructive Solid Geometry) tree, rather than performing volume level voxelization for each CSG node named “slice sweeping”. Here, the basic idea is to generate a slice for each object in the scene first, and than apply the blending and filtering functions on the slice in the postfix order of the volume scene tree. This algorithm needs slice data storage to store the intermediate result. A 2D texture memory was used to represent slices in a slice stack. Since the slice stack operation will occupy a large amount of memory and time, the more slice stack operations will lead to slower volume voxelization process. Thus, in order to reach a higher speed, interior operation nodes must be reduced as much as possible during the design of the volume scene tree.

In another study, Prakash [78] introduces a z-buffer based voxelization algorithm. However, it is limited to only convex objects represented as unstructured grid cells and the voxelization process is intertwined with the scan conversion process.

Karabassi [79] presents another z-buffer based voxelization algorithm, which creates volume data using depth information extracted from six different views of objects. Although this algorithm is fast and easy to implement, the algorithm misses concavities, if some area of a surface is not visible from any of the six faces. And this area will not be properly voxelized.

The voxelization of composite objects or scenes has not been sufficiently studied, though it is perhaps the only way to render and analyze complex scenes with intermixed multiple geometric and volumetric objects. Currently, it seems that there is no an efficient algorithm for the voxelization of such scene graphs.

1.6 Volume Visualization

Representing a surface contained within a volumetric data set using geometric primitives can be useful in many applications; however there are several main drawbacks to this approach. First, geometric primitives can only approximate surfaces contained within the original data. Adequate approximations may require an excessive amount of geometric primitives. Therefore, a trade-off must be made between accuracy and space requirements. Second, since only a surface representation is used, much of the information contained within the data is lost during the rendering process. For example, in CT scanned data useful information is contained not only on the surfaces, but within the data as well. Also, amorphous phenomena, such as clouds, fog, and fire cannot be adequately represented using surfaces, and therefore must have a volumetric representation, and must be displayed using volume rendering techniques.

Volume rendering is the process of creating a 2D image directly from 3D volumetric data. Volume rendering can be achieved using an *object-order*, an *image-order*, or a *domain-based* technique.

Object-order volume rendering techniques use a *forward mapping* scheme where the volume data is mapped onto the image plane. In image-order algorithms, a *backward mapping* scheme is used where rays are cast from each pixel in the image plane through the volume data to determine the final pixel value. In a domain-based technique the spatial volume data is first transformed into an alternative domain, such as compression frequency, and wavelet, and then a projection is generated directly from that domain.

For this algorithm, the strict definition of back-to-front can be relaxed to require that if two voxels project to the same pixel on the image plane, the first processed voxel must be farther away from the image plane than the second one. This can be accomplished by traversing the data plane-by-plane, and row-by-row inside each plane. For arbitrary orientations of the data in relation to the image plane, some axes may be traversed in an increasing order, while others may be considered in a decreasing order. The traversal can be accomplished with three nested loops, indexing on x, y, and z. Although the relative orientations of the data and the image plane specify whether each axis should be traversed in an increasing or decreasing manner, the ordering of the axes in the traversal is arbitrary.

An alternative to back-to-front projection is a *front-to-back* method in which the voxels are traversed in the order of increasing distance from the image plane. Although a back-to-front method is easier to implement, a front-to-back method has the advantage that once a voxel is projected onto a pixel, other voxels which project to the same pixel are ignored, since they would be hidden by the first voxel.

Another advantage of front-to-back projection methods is that if the axis which is most parallel to the viewing direction is chosen to be the outermost loop of the data traversal, meaningful partial image results can be displayed to the user. This allows the user to better interact with the data and terminate the image generation if, for example, an incorrect view direction was selected. Partial image results can be displayed to the user during a back-to-front method also, but the value of a pixel may change many times during image generation. With a front-to-back method, once a pixel value is set, its value remains unchanged.

Clipping planes orthogonal to the three major axes and clipping planes parallel to the view plane are easy to implement using either a back-to-front or a front-to-back algorithm. For orthogonal clipping planes, the traversal of the data is limited to a smaller rectangular region within the full data set. To implement clipping planes parallel to the image plane, data samples whose distance to the image plane is less than the distance between the cut plane and the image plane is ignored. This ability to explore the whole data set is a major difference between volume rendering techniques and the surface rendering techniques. In surface rendering techniques, the geometric primitive representation of the object need to be changed in order to implement cut planes, which could be a time consuming process. In a back-to-front method, cut planes can be achieved by simply modifying the bounds of the data traversal, and utilizing a condition when placing depth values in the image plane pixels.

For each voxel, its distance to the image plane could be stored in the pixel to which it maps along with the voxel value. At the end of a data traversal a 2D array of depth values, called a Z-buffer, is created, where the value at each pixel in the Z-buffer is the distance to the closest non-empty voxel. A 2D discrete shading technique can then be applied to the resulting image.

1.7 Conclusion

In this chapter, we give an overview of the volume graphics, including volume scene representation, volumetric scene model, voxelization techniques and volume visualisation.

Volume graphics is concerned with the graphic scenes defined by a volume data. A volume scene can be constructed from various types of geometric or volumetric objects such as curves, surfaces, solids and CT data sets. A popular technique in scene composition is the scene graph. It organizes geometric objects and rendering parameters in a tree-like hierarchical structure.

Evaluation of the scene graph is carried out by a voxelization algorithm. In recent years, a number of curve and surface voxelization algorithms have been proposed Solid voxelization, on the other hand, has not been sufficiently studied. Solid objects are normally represented by their boundary surfaces. Since the interior of a solid object is not explicitly represented, solid voxelization is difficult and requires an inside/outside test for each voxel involved.

The voxelization algorithm used in this study is based on the fact that surface graphics displays a surface by 2D scan conversion process. When only a slice of the object is displayed, the result is essentially a slice of the volume from a 3D scan conversion. Since 2D scan conversion is implemented in hardware in modern graphics systems, we ought to be able to take this advantage for 3D voxelization.

Using a slice-by-slice approach, our algorithm proceeds by moving a cutting plane, called *Z-plane*, parallel to the projection plane, with a constant step size in a front-to-back order. The thin space between two adjacent *Z*-planes within the *volume space* is called a slice For each new *Z*-plane, the algorithm defines the new slice as the current orthogonal viewing volume, and renders all the surface primitives using standard CSG rendering procedures. The resulting frame buffer image from the display of this slice will become one slice of the resulting volume.

This voxelization algorithm can be used for the voxelization of surfaces, solids, and polygons as well. Surface rendering effects, such as color and shading, can also be stored in the volume representation without extra computation. For the volume scene tree we used “Blist (Boolean list)” representation [4] as a different than traditional representation. In Blist formulation, a Boolean expression is represented as a list of primitives instead of a tree, and this may be evaluated in a pipeline fashion, combining at each step the result of classifying the cells against the current primitive with the result of the previous classification. The fundamental breakthrough provided here lies in the fact that the result of the previous classifications does not require the list of values of cell-primitive classification results, nor a stack of intermediate results of evaluating sub-expressions. Blist distributes the merging operation to the primitives and reduces the storage requirement for each voxel to log(H+1) bits.

CHAPTER 2

A REAL TIME CSG DISPLAY PROCESSOR FOR CONVEX AND CONCAVE OBJECTS

2.1 Introduction

In this chapter, we introduce a real time CSG (Constructive Solid Geometry) display system to render convex and concave objects.

The formation of CSG models is based on performing set operations, usually union or difference, being applied to simpler primitive volumes. Advantages of this display system include capability of rendering both convex and concave objects, elimination of sorting the surfaces according to frontness and backness, and a simpler implementation.

2.2 Previous CSG Rendering Algorithms

One of the most popular methods to model solid objects or volumes is CSG. Unbounded planes divide space into half-spaces. With these half spaces higher-level intersection objects can be defined as convex solids or as their complements, which are convex holes or voids. More complex volumes can be constructed from the union, intersection, and difference of collection of these convex/concave units. In such an object, plane half-spaces oriented away from the viewpoint are called ‘front surfaces’ and oriented towards the viewpoint are called ‘back surfaces’.

The problem of hidden surface removal arises whenever a 3-D object is wanted to be displayed without ambiguity. Hidden surface removal algorithms attempt to determine surfaces that are visible or invisible to the viewer. The z-buffer algorithm is one of the most popular hidden surface removal algorithms. The depth of the current surface, belonging to an object, is compared with those of previous surfaces stored in a buffer. The one having the smaller depth is stored in the buffer as the visible surface. But such an algorithm requires surface sorting, such that in order to constitute an image correctly all the front surfaces of an object must be processed before the back surfaces [80-82]. And this may introduce significant delays even in parallel processing applications. So we developed an algorithm to eliminate the surface-sorting problem for both convex and concave objects.

Although there are many CSG rendering algorithms in the literature, most of them are able to render only convex objects [21], [80], [83-84]. This rendering algorithm can be used for effective visualization of both convex and concave objects.

2.3 CSG Display Algorithm used and Its Implementation

Our CSG display algorithm operates on a similar basis to Çevik's display algorithm [82]. But, sorting of the surfaces according to frontness and backness was eliminated for both convex and concave objects.

The algorithm to display an object is based on the comparisons of distances, depth, between the viewpoint and surfaces in the scene.

2.3.1 The Depth Value

The display space can be considered as a box as in Figure 2.1.

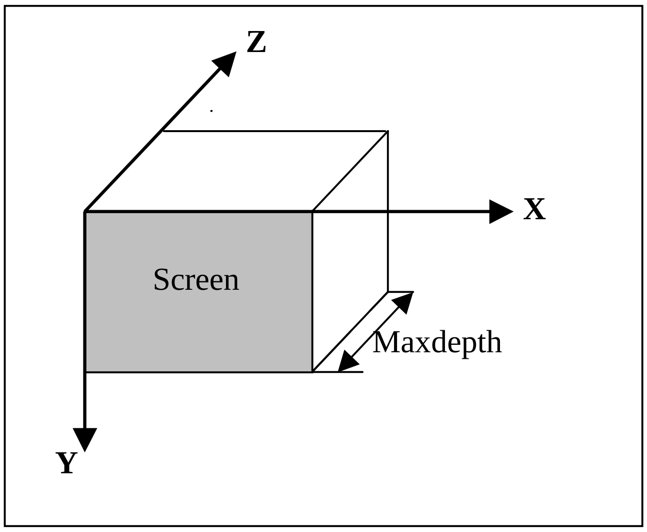


Figure 2.1 3-D Representation of the display space

The depth z, for a plane in the display space, can be written as

$$Z_{x,y} = Z_0 + xdZx + ydZy \tag{2.1}$$

$$K_{x,y} = K_0 + xdkK + ydKy \tag{2.2}$$

where Z_0 and K_0 are the Z and K values at the origin while dZx, dZy, dKx, and dKy are differential increments of Z and K along the x and y axes. K can be defined as the distance between a pixel and a plane if the plane is perpendicular to the screen.

In [81], the depth of a plane, to be displayed, is generated by means of a depth generator on every pixel simultaneously on the display window.

2.3.2 Display of convex and concave objects

The following simple algorithm, depicted in Figure 2.2, employs the plane properties (front or back planes), explains how a convex object is formed from half spaces. In diagram (a), the visible part of the planes which construct the object surface is determined by selecting the front planes which have greater distances to the viewpoint amongst the other front planes along a given ray. In diagram (b), selecting back planes having smaller distances isolates the back surface. Finally, in diagram (c), the front planes are kept along a given ray whenever ZFs<ZBs, where ZFs and ZBs are the depths of a front surface, and the depth of a back surface, respectively. Otherwise, they are clipped out by the back planes. The surface produced by this process is the visible surface of the intended object observed from the viewpoint.

A similar algorithm can be used to display concave objects. Figure 2.3 explains the algorithm. First, in diagram (a), the front planes which have smaller distances are selected amongst the others, along a given ray. Then, in diagram (b), the back planes having greater distances amongst the others are selected. Finally, the visible surface of the hole is obtained as part of the previously selected front planes for which ZFs>ZBs (Diagram (c)).

It can be seen, in this algorithm, there is no importance of which surface front/back is processed first. The algorithm can be implemented in identical processors, Figure 2.4, at each pixel. Although the use of the processors is going to be explained in detail later, it is important, here, to mention that our processors have an additional register, (B), for holding the depths of the back planes of the object to be displayed.

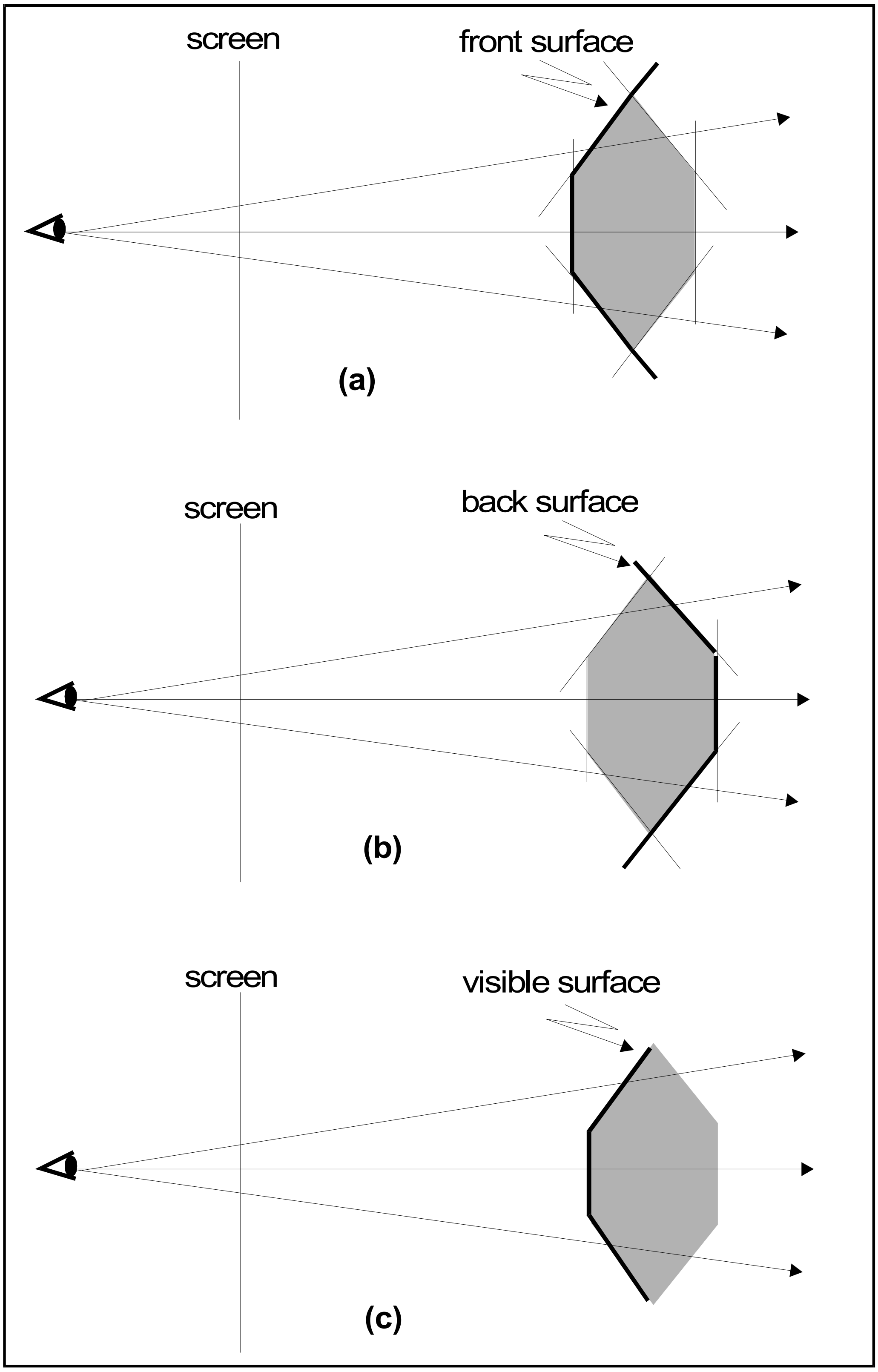


Figure 2.2 Algorithm used to display a convex object

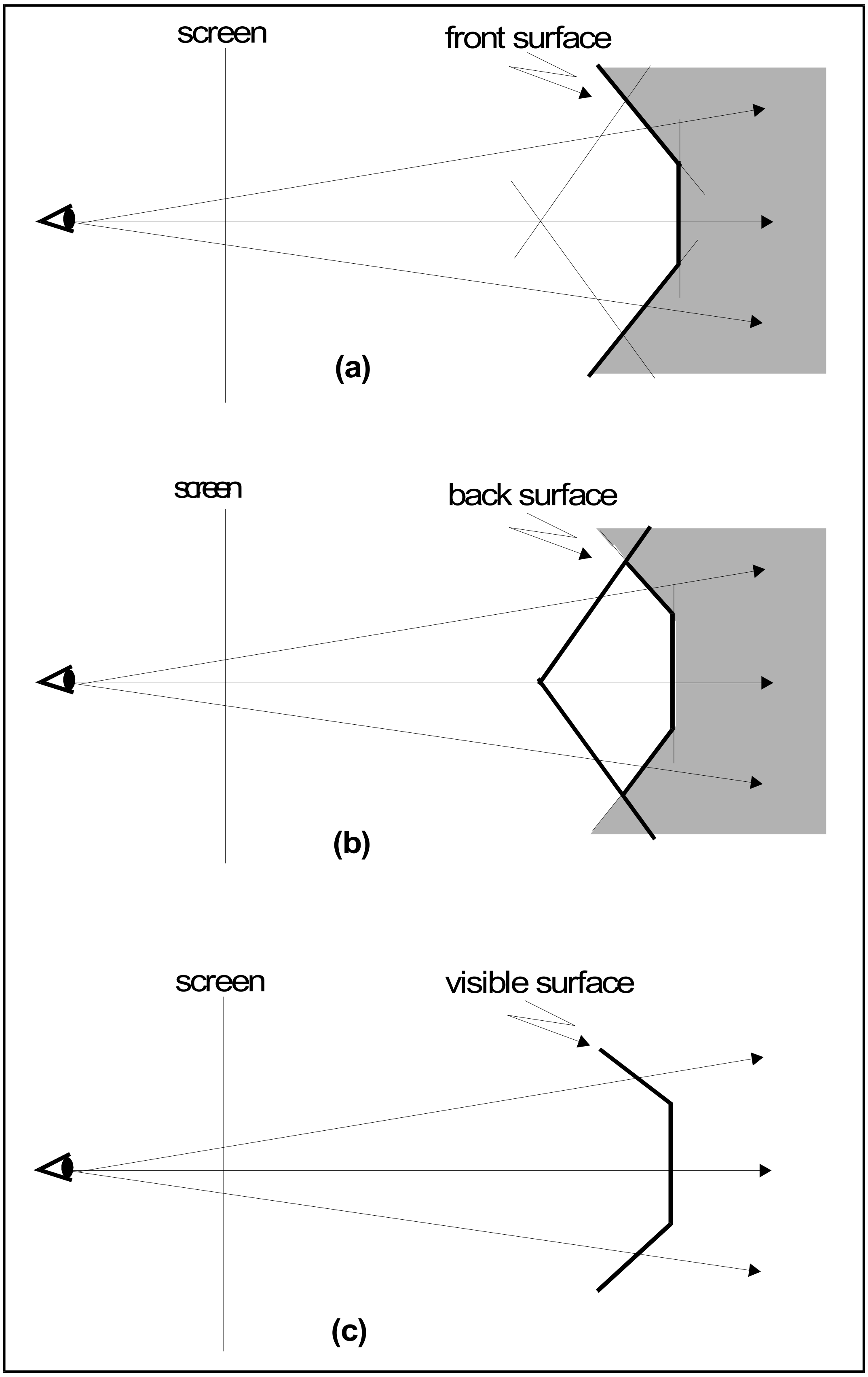


Figure 2.3 Algorithm used to display a concave object

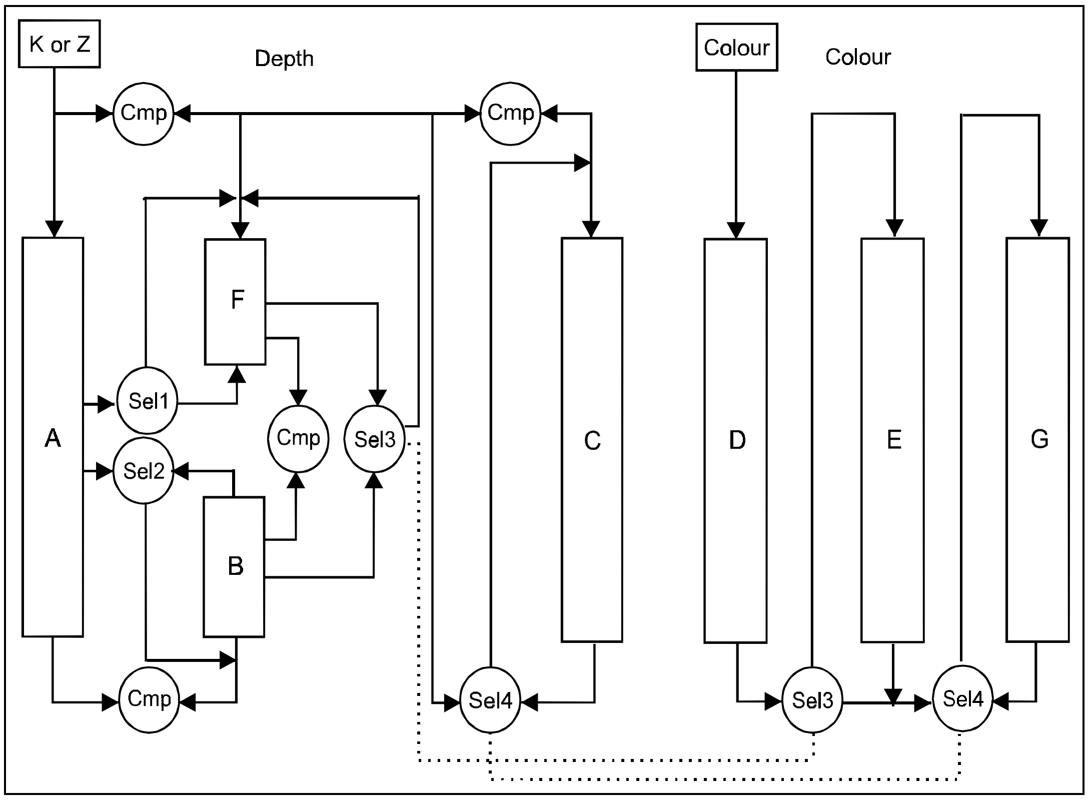


Figure 2.4 The basic pixel processor for a pixel

However, hidden surface removal algorithms without eliminating the surface sorting problem did not use a register for holding back planes. Instead, front and back planes were processed in the same register. If all the back surfaces were processed after the front surfaces the object was displayed correctly. But if one or some of the front planes was/were processed after the back planes then it/they would never be clipped even its/their depths was/were greater than the depths of the back surfaces (which were processed before front surfaces), since the back surfaces clipped all front surfaces, replacing depth values with maxdepth value (which represents infinity) at each pixel where their depths were less than those of front surfaces.

In order to build algorithms we need some information about the planes which are used to constitute an object. We can divide this information into two parts: externally supplied information; includes data related to the nature of the plane and supplied along with input data, and internally supplied information which is generated during the comparisons of the depth values, in the pixel processors. So, we can define this information in the form of Boolean variables as follows:

Externals:				Internals:			
L=0	Convex object	L=1	Concave object	N=1	$A<0$	N=0	$A>0$
S=1	First plane	S=0	Not first	J=1	$A\leq F$ or $A\leq B$	J= 0	$A>F$ or $A>B$
H=1	Front surface	H=0	Back surface	T=1	$A=F$ or $A=B$		
				T=0	$A\neq F$ or $A\neq B$		
Y=1	Perpendicular	Y=0	Not Perpend.				
M=1	Overflow	M =0	No Overflow				

2.3.3 First-Stage selection

The role of the first-stage selection is to determine the object defined by each set of planes in the display list.

In this stage we use three depth registers: A, F, B, and two colour registers: D and E (Figure 2.4). Initially all the depth registers are loaded with maxdepth value and the colour registers are loaded with the background colour.

For each plane the Z or K values are initially stored in A-registers. If the plane is a front plane or a perpendicular plane the depth values in A-registers are compared with those in F-registers, which hold the depth of the previous front surface, and greater depth values are stored in F-registers (smaller depths for concave objects). If the plane is a back plane then the depth values in A-registers are compared with those in B- registers, which hold the depth of the previous back surface, and smaller depth values are stored in B-registers (greater depths for concave objects).

After processing all the planes of the object, the depth values of F and B-registers are compared at each pixel in the scene and the following actions are taken:

Convex		Concave
If $ZF<ZB$;	Keep ZF	If $ZF>ZB$; Keep ZF
If $ZF\geq ZB$;	Put maxdepth value into F-register	If $ZF\leq ZB$; Put maxdepth value into F- register

The colour section is a slave process to that of the depth. And possible cases for the first-stage selection according to nature of the plane are:

Front/Perpendicular plane

Depth section	**Colour section**
1)Put A (Transfer the depth from A to F-register)	1)Put D (Transfer the colour from D to E-register)
2)Keep F (Keep the current depth in F-register)	2)Keep E (Keep the current colour in E-register)
3)Put Maxdepth (Put Maxdepth in to F-register)	3)Put background (Put background colour into E-register)
4)Put 0 (Put 0 into F-Register)	4) Put internalcolour (put internal colour of the object into e- register)

Back plane

1)Put A (Transfer the depth from A to B-register)

2)Keep B (Keep the current depth in B-register)

3)Put Maxdepth (Put maxdepth into B-register, surface is at infinity)

There is no need to use a colour register for the back planes, since they are invisible.

Front/Perpendicular plane (A and F registers are in use)

- S=1, Y=1 (The first plane in the display list, perpendicular to the screen)

As the plane is perpendicular to the screen, A-register is loaded with a K value. If $K<0$, the plane is clipped by the screen, and the visible surface is at infinity. If $K\geq0$, on the other hand, the visible surface is the screen itself. The result is:

If $K<0$ Put 1 ($N=1$) If $K \geq 0$ Put 0 ($N=0$)

- $S=1$, $Y=0$, $H=1$ (The first plane in the display list, not perpendicular, front surface)

The plane is visible only if $0 \leq Z<$Maxdepth. If $Z<0$, which means that the surface is between the screen and the viewpoint, the plane is clipped by the screen, so the internal colour of object is displayed. If $Z \geq$Maxdepth, then the visible surface is at infinity.

If $0 \leq Z<$Maxdepth Put A ($N=0$, $M=0$)

IF $Z<0$ PUT 0 ($N=1$)

If $Z \geq$Maxdepth Put 1 ($N=0$, $M=1$)

- $S=0$, $Y=1$, $L=0$ (Not first plane, perpendicular to the screen, convex object)

If $K \geq 0$ the visible surface is the one contained in F-register. In regions where $K<0$, however, the perpendicular plane clips the other surfaces so that the visible surface is at infinity.

If $K<0$ Put 1 ($N=1$) If $K \geq 0$ Keep F ($N=0$)

- $S=0$, $Y=1$, $L=1$ (Not first plane, perpendicular to the screen, concave object)

If $K<0$, the visible surface is the plane itself, if $K \geq 0$ then the screen is the visible surface itself.

If $K<0$ Keep F ($N=1$) If $K \geq 0$ Put 0 ($N=0$)

- $S=0$, $Y=0$, $H=1$,$L=0$ (Not first, not perpendicular, front surface, convex object)

The visible surface will be the plane which has the greater depth. But if $A=F$ then the result is up to the user, it can either be the depth of A or F.

If $F<A<$Maxdepth Put A ($M=0$, $N=0$, $J=0$)

If $0 \le A<F<$Maxdepth	Keep F	(M=0, N=0, J=1, T=0)
If A=F	Put A or F	(M=0, N=0, J=1, T=1)
If $A<0 \le F$	Keep F	(N=1)
If Maxdepth$\le$A	Put 1	(N=0, M=1)

- S=0, Y=0, H=1, L=1 (Not first, not perpendicular, front surface, concave object)

The visible surface will be the plane which has the smaller depth, if A=F then result is up to the user, it can be either A or F.

If $0\le F<A<$Maxdepth	Keep F	(M=0, N=0, J=0)
If $0 \le A<F<$Maxdepth	Put A	(M=0, N=0, J=1, T=0)
If A=F	Put A or F	(M=0, N=0, J=1, T=1)
If $A<0 \le F$	Put 0	(N=1)
If Maxdepth$\le$A	Keep F	(N=0, M=1)

Back planes (A and B registers are in use)

- S=1, Y=0, H=0 (The first plane, not perpendicular, back surface)

If the plane depths are less than 0 then the plane is clipped by the screen, and maxdepth value is put into B-register. If plane depths are greater than 0, then these depths are put into B-register.

If Z<0	Put 1	(N=1)	If $Z \ge 0$	Put A	(N=0)

- S=0, Y=0, H=0, L=0 (Not first, not perpendicular, back surface, convex object)

The plane which has the smaller depth is kept in B-register (If the depth is greater than 0).

If $0 \le B<A<$Maxdepth	Keep B	(M=0, N=0, J=0)
If $0 \le A<B<$Maxdepth	Put A	(M=0, N=0, J=1, T=0)

If $B<\text{Maxdepth}\leq A$	Keep B	(M=1, N=0)
If $A<0\leq B$	Put 1	(N=1)
If A=B	Keep B or Put A	(N=0, M=0, J=1, T=1)

- S=0, Y=0, H=0, L=1 (Not first, not perpendicular, back surface, concave object)

The plane which has the greater depth is kept in B-register (If the depth is less than maxdepth).

If $0\leq B<A<\text{Maxdepth}$	Put A	(M=0, N=0, J=0)
If $0\leq A<B<\text{Maxdepth}$	Keep B	(M=0, N=0, J=1, T=0)
If B<Maxdepth<=A	Put 1	(M=1, N=0)
If $A<0\leq B$	Keep B	(N=1)
If A=B	Keep B or Put A	(N=0, M=0, J=1, T=1)

Using the above cases we have constructed Karnaugh maps [85] for the simplification of the Boolean functions S1 and S2 which control the data selection within the pixel processors. The actions to be taken according to values of S1 and S2 are:

Front/Perpendicular plane (Sel1)			**Back plane (Sel2)**		
Put A	S1=0	S2=0	Put A	S1=0	S2=0
Put 0	S1=1	S2=0	Keep B	S1=0	S2=1
Put F	S1=0	S2=1	Put 1	S1=1	S2=1
Put 1	S1=1	S2=1			

After the simplification of the maps we obtain the following Boolean functions for S1 and S2.

For front/perpendicular surfaces

$S1' = L'Y'S'N + Y'N'M' + L'YS'N' + LY'S'N' + LYS'N$

$S2' = Y'SN + L'Y'J'N'M' + YSN' + LY'N + LYN' + SN'M' + LJN'M'T'$

For back surfaces

$S1' = L'J'S'N' + L'S'N'M + LS'N + LJS'M'K'$

$S2' = SN' + T'L'JN'M' + LJ'N'M'$

2.3.4 Clipping process

The clipping process determines whether a surface which has already been in F-registers after first-stage selection lies inside a clipping volume, at a given pixel, or not. The clipping algorithm consists of, first, expressing the clipping phrase in the postfix form. Then, for each clipping plane, a partial inside/outside test is performed, at each pixel, on the previously selected surface. At the end of this stage the visible surfaces are stored in F-registers. The partial inside/outside test is illustrated in Figure 2.5.

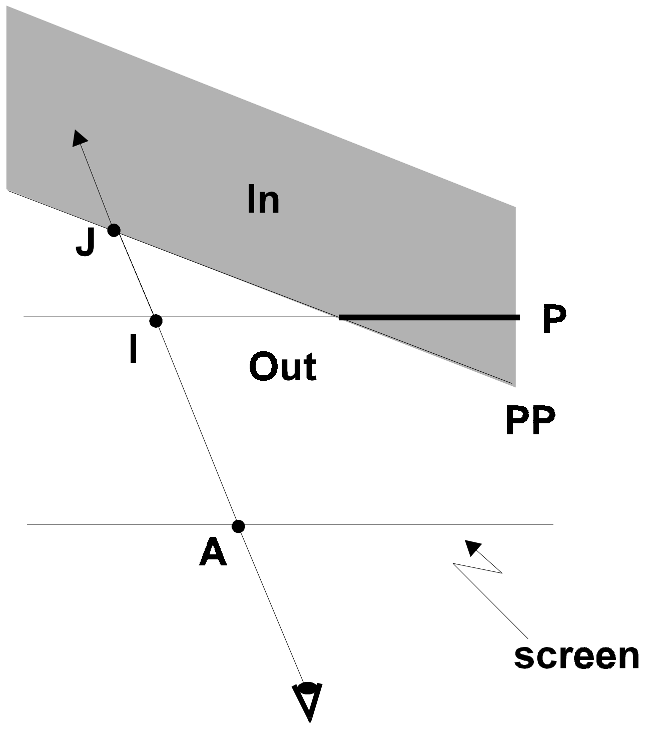


Figure 2. 5 Clipping Algorithm

Now, the details of the evaluation of the partial inside/outside test will be given.

- **Y=1** (Clipping plane is perpendicular).

If the clipping plane is perpendicular, the sign of the K value belonging to this plane at a given pixel determines if the point is going to be kept or discarded (Figure 2.6).

LL=1	if K≥0	(N=0)
LL=0	if K<0	(N=1)

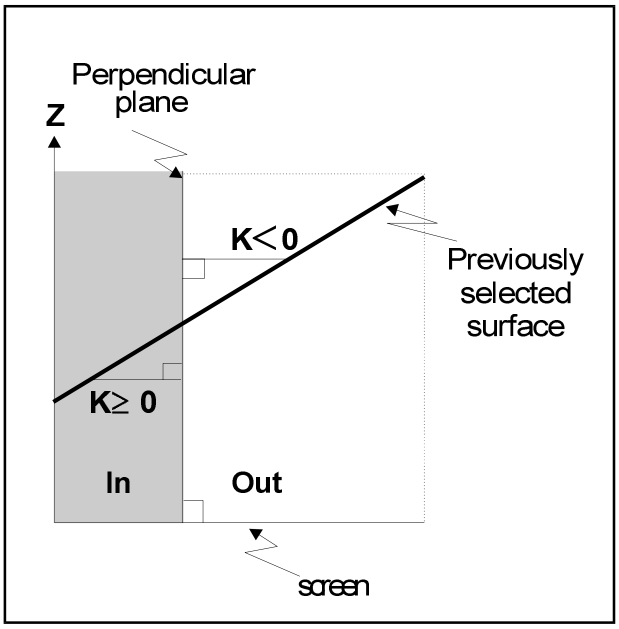


Figure 2. 6 Inside/outside test for a perpendicular clipping plane

For non-perpendicular planes, two points can be defined, at each pixel, one on the clipping plane, and one on the previously selected plane (Figure 2.7 and 2.8). By comparing the depths of A and F it can be determined whether F is inside or outside the half-space defined by the clipping plane. In the case of the equality between the depths of A and F, the result is arbitrary because there is no reason why such a point should be considered inside or outside. Therefore, it is up to the user to decide as a preliminary condition whether user wants that surface to be kept or not. This information is represented by an external variable X such as, X=0 if the surface will be kept, and X=1 if not.

- **Y=0, H=1** (Not perpendicular, front plane)

LL=0	If A>Maxdepth	(M=1, N=0)
LL=1	If A<0	(N=1)
LL=0	If A>F	(J=0)
LL=1	If A<F	(J=1)
LL=X	If A=F	

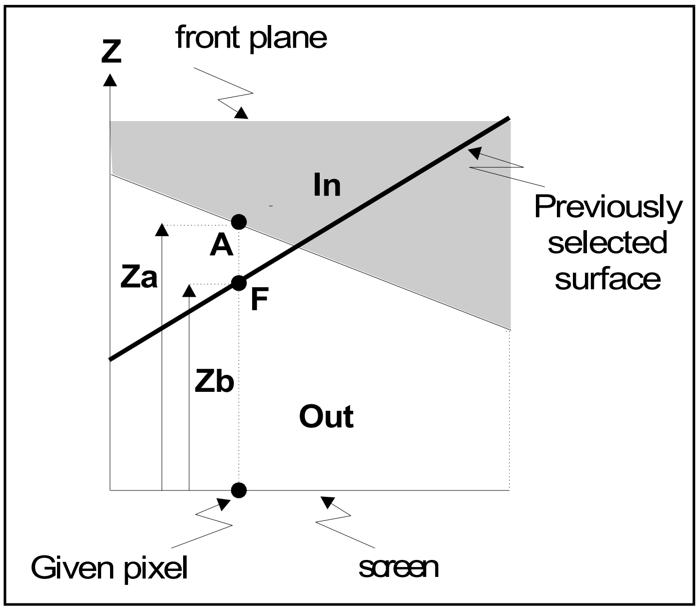


Figure 2. 7 Inside/outside test for a front clipping plane

- **Y=0, H=0** (Not perpendicular, back plane)

LL=0	If A<0	(N=1)
LL=1	If A>Maxdepth	(M=1, N=0)
LL=0	If A<F	(J=1)
LL=1	If A>F	(J=0)
LL=X	If A=F	

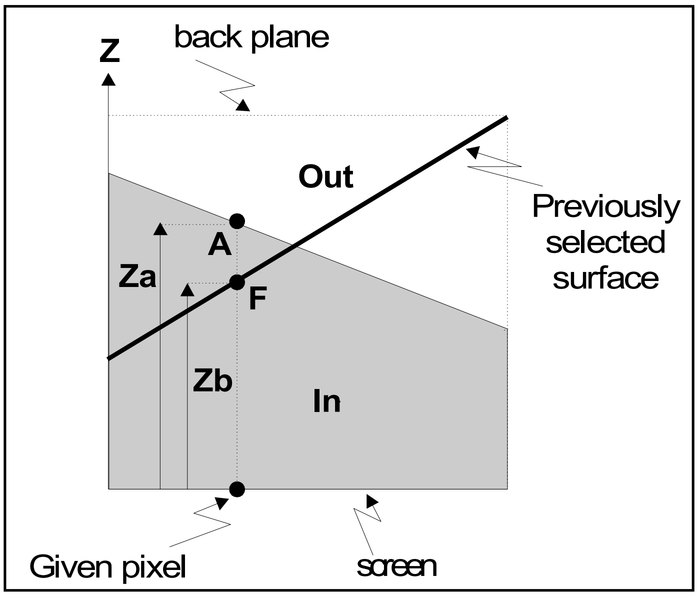


Figure 2. 8 Inside/outside test for a back clipping plane

Using these results, a Boolean function can easily be obtained for LL (partial answer for each clipping plane).

LL=H'.Y.T' + H.Y'.N + X'.H.Y'.J.M' + Y.J'T'+ H.Y.J.T + H'.Y'.M. N'

+ H'.Y'.T.N' + X.H.Y'.J.T'.M'+ H'.Y'.J'.N'

As explained above, LL is an input for the local stack for each pixel. When all the planes of the clipping volume have been processed, the evaluation of the stack gives the result (LL') of the clipping for that pixel. The value of the LL' determines that how the F-register will be modified by the clipping:

Keep F	If LL'=1	The surface is inside
Put 1	If LL'=0	The surface is outside

2.3.5 Second-Stage selection

The second-stage selection is required each time a new object is to be added to an existing scene which has been kept in C-registers (depth) and G-registers (colour). An object which may have been clipped during the clipping stage is transferred from F-registers to the C-registers taking into account that which object will be visible at a given pixel. The comparison is as follows:

Put F	If	$F \leq C$	$(J=1)$
Keep C	If	$F > C$	$(J=0)$

Keep in mind the possibility of the fact that a new scene overwriting the existing one may be required. So an external variable, P, affects the second-stage selection such that:

Put 1	If	$P=1$	(initialise C-registers)
Keep C	If	$P=0$	(no change)

Therefore, there are three possibilities for the modification of C-registers:

Put 1	$S_3=0$,	$S_4=0$	put F	$S_3=0$, $S_4=1$
Keep C	$S_3=1$,	$S_4=X$ (don't care)		

After the simplification of the maps we obtain the following Boolean functions for S_3 and S_4.

$S1=J'P'$ $S2=P'$

2.4. Hardware Implementation

In this section, the display algorithms which were implemented in terms of Boolean functions will be realized using digital logic, mostly in LUT based FPGAs.

The structure of the system is shown in Figure 2.9.

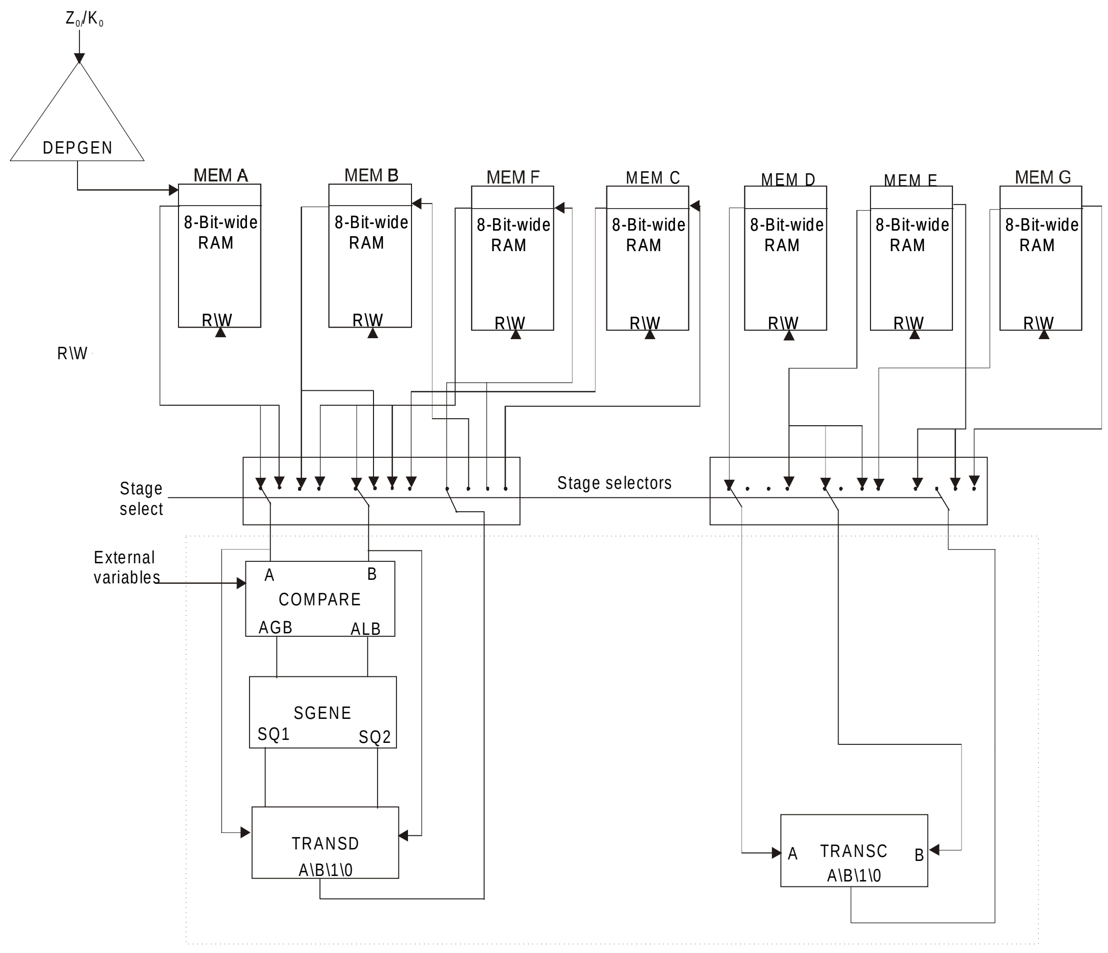


Figure 2.9 The system structure

Here, DEPGEN is the tree-structured depth generator [81-82]. Each cluster of it, making a row in the display area is connected to the MEM A memories. While the depth values are coming from the DEPGEN, the colour values corresponding to each plane are fed in the MEM D memories.

The stage selectors switch between the first-stage and second-stage selection. There are four different positions of the stage selectors. At the three of these positions the system is in the first-stage selection mode. At the last position the system is in the second-stage selection mode. When the switch is on the first position, the selector connects the mem A and mem F to the depth section of the pixel processors, and mem D and mem E to the colour section. For the second position, the selector connects mem A and mem B to the depth section, here, the usage of the colour registers is not necessary since B registers keep the depth values of the back surfaces. At the third position, mem B and mem F are connected to the depth section and mem E to the colour section. For the last position, the selectors connect the mem F and mem C to the depth section of the pixel processor and mem E and mem G to the colour sections.

The part shown in the dashed frame is the pixel processor. The compare module compares two depth values, while SGENE module determines the S1 and S2 values, which were explained in the previous section, according to the result of the comparison. TRANSD puts the required value, either the new depth or the old one or 1 or 0 on the output depending on the S1 and S2. The TRANSC is the same as TRANSD, but it deals with the colour values rather than the depth.

2.4.1 The Data Structure and Data Flow

The depth data coming from the tree-structured depth generator is downloaded to the pixel processor in a serial manner. The depth stream carrying the depth value is composed of 32-bit-long strings in 2's complement format (Figure 2.10).

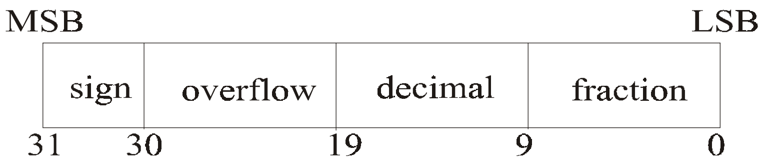


Figure 2.10 Data structure of the depth string

The first 10 LSBs (low significant bits) (0-9) are for the fractional part of the depth, the next ten (10-19) are for the decimal part, the other eleven (20-30) are for the overflow, which means that the plane is beyond the display limit of the window, and the MSB (most significant bit) is for the sign of the depth value.

The first column in Figure 2.11 explains the flow of the depth data when the selection switch is at the first position. Each pattern in the figure represents a plane. In the very first item, the selector has decided that the plane in the F register is going to be kept, while the next plane belonging to an object is ready to be downloaded from the depth generator. In the second item, the depth values of the downloading plane are compared to those of the one keep in the F register. When the plane has been downloaded, in the third item, the selector determines what is going to be transferred into the F register. When the switch is at the second position, the same action takes place, except the B register is used instead of the F register. For the third position in Figure 2.11, the depth values stored in the B and F registers are compared and the depth values of the visible part is stored in the F register. For the second stage selection the same action takes place, but in that case F and C registers are used. Then the object, in the F register, is added to the existing scene.

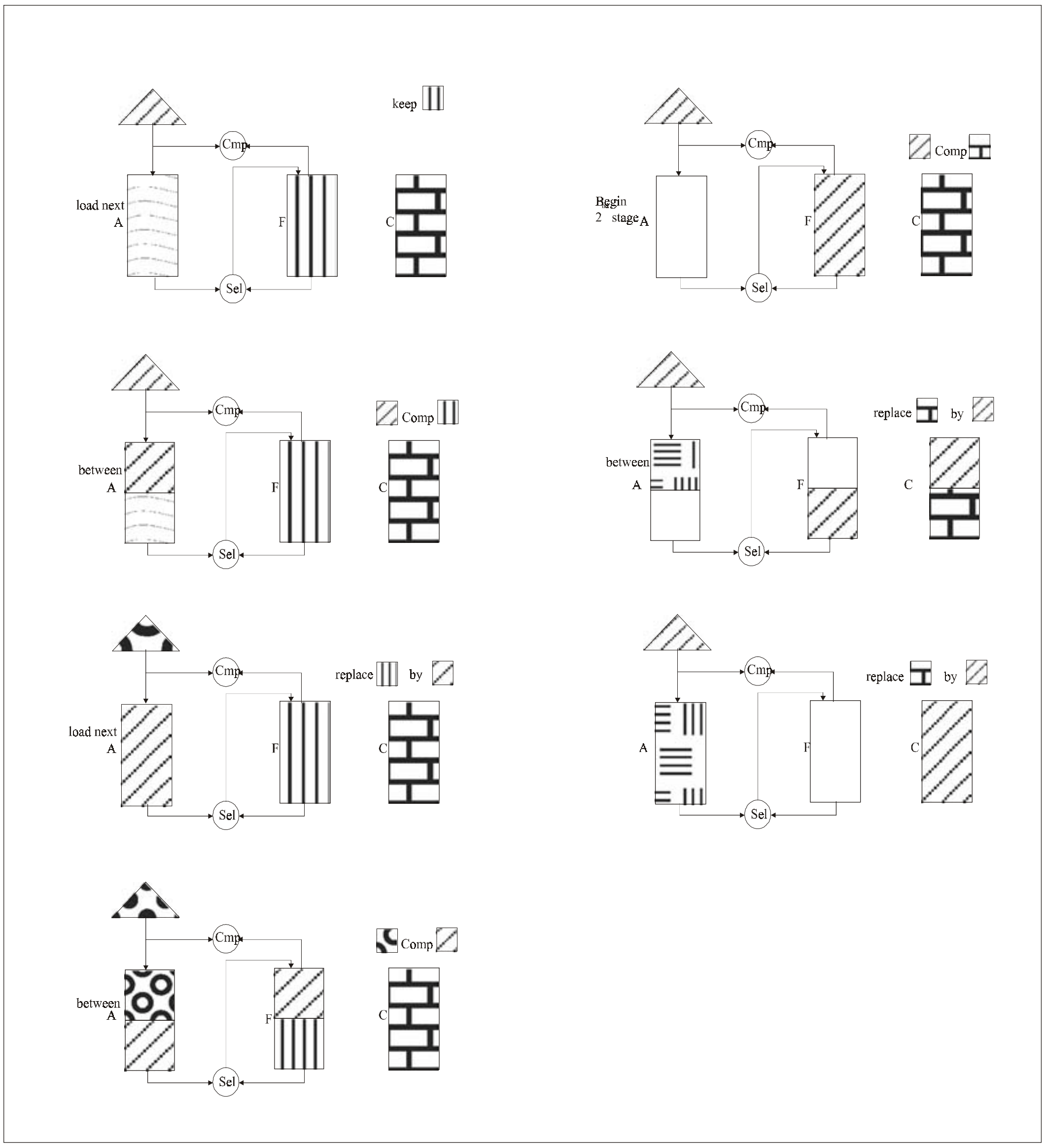


Figure 2.11 The data flow

The simulation of the display system shown in Figure 2.9 was performed using the Work View® software package. This software allows the user to arrange the schematics in a hierarchical order that brings about a compact view, and, consequently, an easy-to-follow circuit. The software can use the XILINX® supplied component and symbol libraries to save the user from a laborious behavioural determination of the components. The user, then, can simulate the schematic by means of the VIEWSIM®, which is the functional simulation program supplied by the Work View.

2.5 Performance

The performance of a pixel-based system, in terms of the number of surfaces rendered per second, can be defined by the following formula:

$$P = \frac{1}{N_D \cdot T_C} \tag{2.3}$$

where

P, performance in terms of surfaces/sec,

N_D, number of bits to store the depth (so the number of clock cycles required to render a single surface),

T_C, clock period in terms of seconds.

Using this formula, the number of surfaces that can be rendered in real-time can be calculated:

$$P_R = t \cdot P \qquad P_R = \frac{t}{N_D \cdot T_C} \tag{2.4}$$

where

P_R, real-time performance,

t, frame time for real-time display in sec. (1/30 to 1/70).

This is the real-time performance of an S_x by S_y pixel area of the display. Here, it is assumed that there is a basic pixel processor available for every pixel on the display area. As can be noticed, the real-time performance is practically dependent on the clock speed. And it is independent of the display size as long as there is a single processor array dealing with the entire area of the display.

With a 32-bit depth length and a 10 MHz clock speed, the performance of such a system would be:

$$P = \frac{1}{32.100 \cdot 10^{-9}} = 312500 \text{ surfaces/second,}$$

Or, in real-time (t=1/50 sec.)

$$P_R = \frac{1/50}{32 \cdot 100 \cdot 10^{-9}} = 6250 \text{ surfaces/frame time.}$$

Although the performance that was calculated above is enough for the real-time animation of simple objects, it must be improved for complex scenes. The basic strategy for increasing the real-time performance is to subdivide an image area into sub-areas, not necessarily continuously, and generate sub-images in parallel, using multiple panels of the pixel processor array. In this approach, many pixel processor arrays can operate on separate, small, n by m panels on the screen.

Primitives that fall into more than one region must be processed in the corresponding panels, which mean processing the same primitive more than once in separate panels. This suggests that there is not a fixed performance; instead that the performance is dependent upon how evenly the primitives fall into the panels, and how many panels are being processed in parallel. A rough estimate of the potential increase in the performance however can be made, assuming an even distribution of primitives within the panels, and having enough panels to cover the entire screen.

$$P_R = \frac{n \cdot m \cdot M_P}{S_n \cdot S_m} \cdot \frac{t}{N_D \cdot T_C} \tag{2.5}$$

where

$S_n.S_m$,	screen resolution,
n.m,	panel size.

For a 1024 by 1024 pixels resolution screen, the maximum real-time performances with respect to different panel sizes are given in Table 2.1.

Table 2.1 Maximum real-time performance with respect to panel sizes

PANEL	NUMBER OF SURFACES PER FRAME
512 X 512	25000
256 X 256	100000
128 X 128	400000
64 X 64	1600000
32 X32	6400000
16 X 16	25600000
8 X 8	102400000

The graph of the real-time performance with respect to the number of processor arrays working in parallel is shown in Figure 2.12.

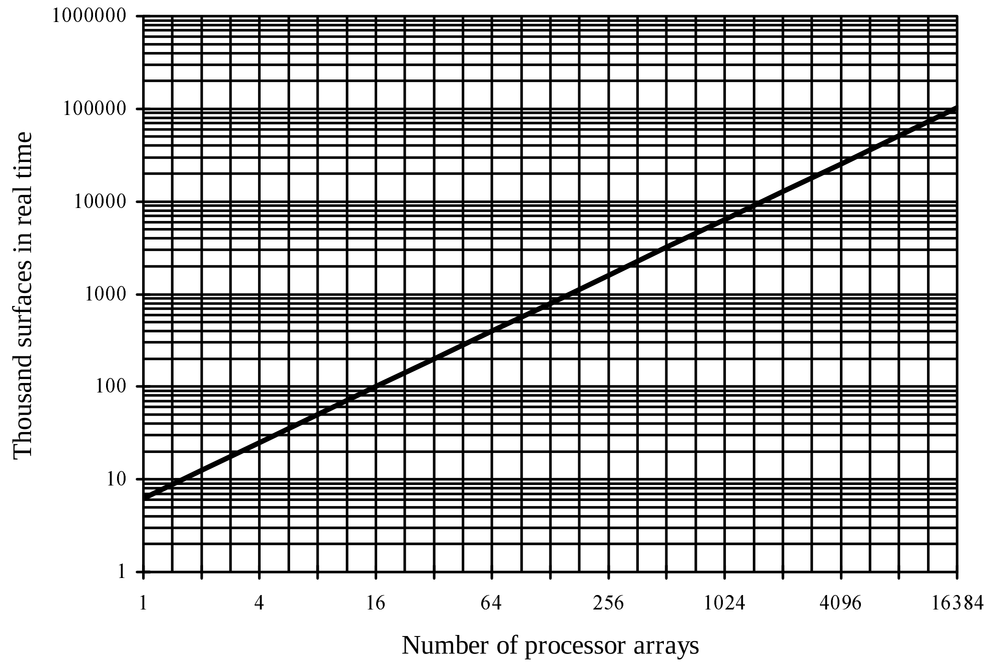


Figure 2.12 The real-time performance

CHAPTER 3

THE VOLUMETRIC SCENE GRAPH and its EVALUATION

3.1 Introduction

In this chapter we will introduce the volumetric scene graph and volume scene tree expression. Scene evaluation is carried out by the slice-sweep voxelization algorithm [26].

3.2 The Volumetric Scene Graph

Generally, a scene graph is a hierarchical organization of shapes, groups of shapes, and groups of groups that collectively define the content of a scene. Shapes and subtrees may be shared among multiple groups, creating a directed acyclic graph. Scene graphs are widely used to define complex 3D scenes with hundreds or thousands of separate shapes.

Traditionally, scene graphs contain shapes defined by surfaces, such as sets of polygons. Volume scene graphs introduced here support volumetric shapes defined procedurally or by data sets read from disk. Compositing operators in the graph specify the treatment of overlapping volumes. Transform operators translate, rotate, and scale shapes or subtrees to construct a scene.

Constructive Solid Geometry (CSG) introduces an explicit scene description expressed as a binary tree containing solid primitive shapes (box, sphere, etc.) at the leaves, and set operators (union, intersect, subtract) for interior nodes. The procedurally defined primitives and set operators of CSG trees place them on the functional side of the scene description spectrum.

Volume scene graphs extend CSG trees to support leaves containing volume data set primitives, and to support interior nodes that perform arbitrary operations.

3.3 Volumetric CSG Modeling

A volumetric CSG model provides a constructive way of designing 3D models using intermixed geometric and volumetric objects.

Volumetric Boolean operations in the volumetric CSG tree can be represented by blending operations in the scene expression tree. The standard definition of volumetric Boolean operations is straightforward. They can also be represented by blending functions in a 3D graphics. More specially, a frame buffer is applied between the incoming pixel color and the pixel color at the corresponding location in the frame buffer. Let C_s denote the incoming pixel color in the frame buffer and C_d denote the corresponding pixel color in the frame buffer. And C is the output color to be written back to the frame buffer. Thus, volumetric Boolean operations in the volumetric CSG tree can be implemented with the replacement of blending functions in the scene expression by the following combinations:

Union:

$$C=\max(C_s,C_d)$$

Or:

$$C=k_sC_s+k_dC_d$$

Where k_s and k_d are their factors respectively.

Intersection:

$$C=\min(C_s,C_d)$$

Difference:

$$C=\min(C_d-C_s,0)$$

Where, C is equal to 0 if $C_d-C_s<0$.

3.3.1 Volume Scene Tree Conversion

A volumetric scene expression can be represented as sequence of objects and operators that are carried out in a certain order. It describes the composition process of a complex volumetric scene, and can be best represented by a volume scene expression tree, or simply VST (Volumetric Scene Tree).

It is quite easy to turn a volumetric CSG tree into a volumetric scene expression tree (i.e. VST). VST is not a complete binary tree. It can be any kind of tree, which allows a parent node to have more than two children or only one child. So the VST of the volumetric CSG tree may also have more than two operands for an OR (union) blending operator. This kind of scene expression tree can reduce the time of stack operations, thus running speed of the program may be improved.

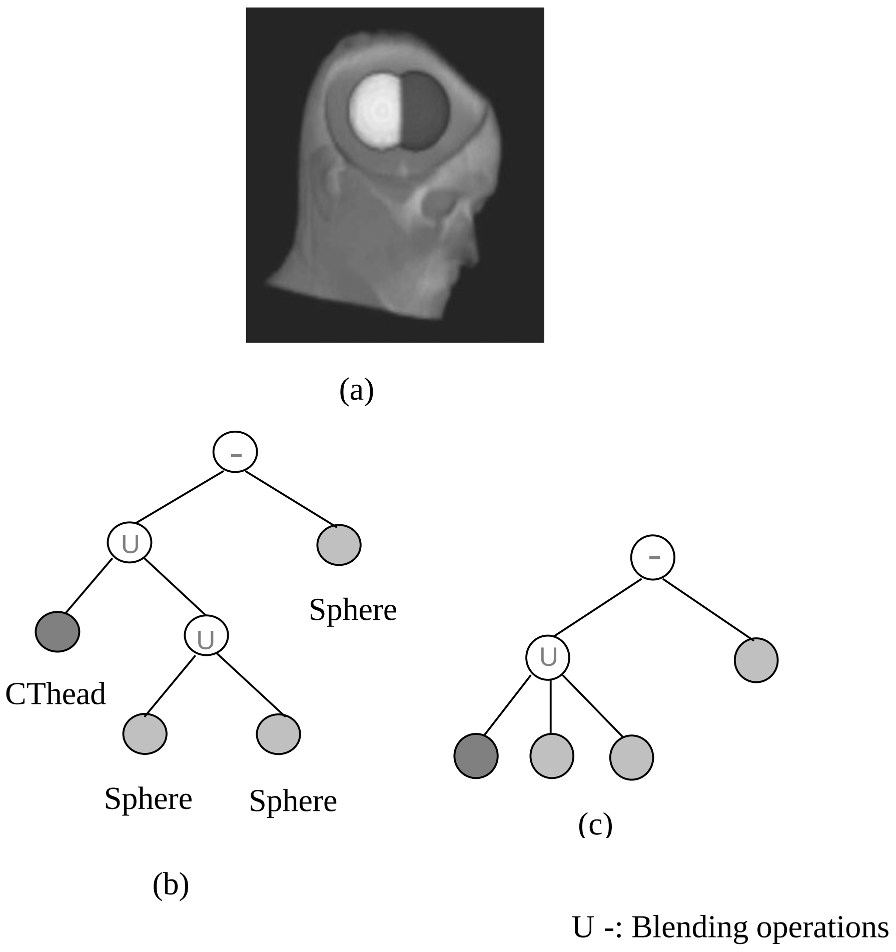


Figure 3.1 An example for general volumetric CSG modeling: (a) resulted image, (b) the volumetric CSG tree, (c) and its corresponding VST.

Figure 3.1 shows such a simple example, which converts a volumetric CSG tree into a VST. Here, the CT dataset has 256 different intensity levels for each voxel. The brain of the CT head is represented by two geometric spheres with black and white colors.

The evaluated result of a volumetric scene expression is a field defined over a volume space. Conceptually, each object represents a volumetric field, defined by its attribute values. The voxelization process will then extract volumetric information from different representations and perform operations for them according to their operators in the scene expression.

3.4 Scene Evaluation

Scene evaluation is very important for volumetric scene graph. In this chapter, scene evaluation is carried out by the slice-sweep voxelization algorithm [26].

3.4.1 Slice Sweeping

Space-sweep approach has been widely used in many computational geometry and computer graphics problem [93-94]. This approach can be applied to compute volume information of a scene expression on a 2D slice, which moves across the volume space in regular increments. So, this approach is called slice sweeping.

The basic idea of slice sweeping is to generate slices for each object in the scene first, and then apply the blending functions on the slices in the order defined by the scene expression [26]. To ensure the correct operational order, a postfix expression is generated by a standard pre-order traversal of the volume scene tree. For example, the expression of the volume scene tree

((Cylinder)U(Curve))U((Surface)U(Cube)U(Sphere))

will be rewritten into the following postfix form:

(Cylinder)(Curve))U((Surface)(Cube)(Sphere)UUU

This algorithm proceeds by moving a slice plane, which is parallel to the projection plane, with a constant step size in a front-to-back or back-to-front order in volume space. This process is shown in Figure 3.2. For each new slice, all primitives are rendered into the slice buffer (i.e., frame buffer) using a standard rendering pipeline. This algorithm needs a slice stack data structure to store the intermediate result slices. For each slice, stored as a 2D image, the algorithm reads one node at a time from the postfix expression of the VST, and pushes slices of leaf nodes (objects) into the stack. If the node is a blending operator, it can have more than one child node. For this operator node, it needs to pop *k* slices from the stack, where *k* is the number of children for this node. At the end, the last slice in the stack will be final evaluated result for this slice.

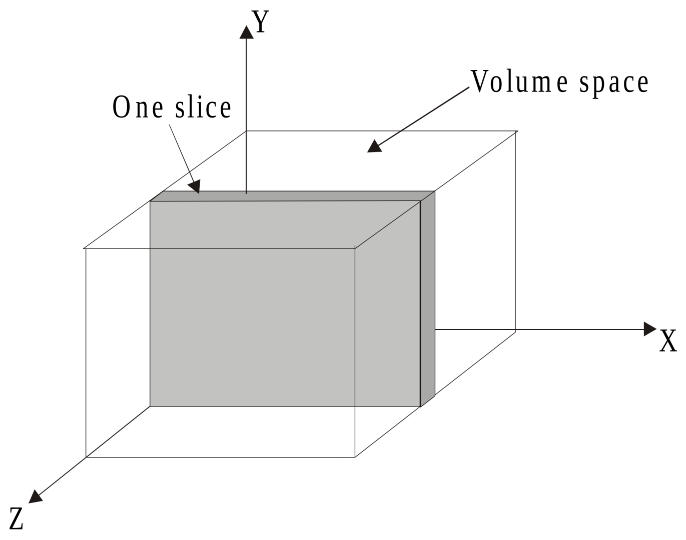


Figure 3.2 Slicing the volume space

The entire algorithm is described in the following pseudocode:

```
Slice Sweeping (VST vs_tree)
{
for (each slice S){
  read the first node P from the VST
  while (P is not NULL) {
    if (P is a leaf node){
     voxelize P into the slice S;
push(S)
}
if(P is a blending operator){
for(each operands)
T[i]=pop();
S=blending(T[0],…,T[k],P) ;
Push(S);
}
S=pop();
Output the final slice S into volume memory;
Move to the next slice;
}
}
```

3.4.2 Solid Object Slicing

Unlike volumetric objects, solid objects are normally represented by their boundary surfaces without explicit interior information. Thus, when solid objects are clipped with a geometric method, their interior voxels should be generated to get complete solid slices. In this chapter, for solid voxelization, an XOR operation based method will be introduced.

A cubic volume space is first defined over the CSG model. The algorithm proceeds slice by slice in a front-to-back order by moving the Z-plane, a plane parallel to the projection plane, along the viewing direction to generate slices for all primitives (Figure 3.2). For each slice, the algorithm defines the viewing volume of the system as a thin space between two adjacent Z-planes inside the volume boundary, and then renders each primitive within this viewing volume. When the algorithm moves from Z-plane to Z-plane, slices of the primitives are displayed and composited onto the frame buffer in a front-to-back order. Using a proper color-coding of the primitives, the algorithm can directly generate slices in the frame buffer. The distance between adjacent Z-planes determines the resolution of the volume in the Z direction. The resolutions in the X and Y directions are determined by the size of the display window.

If the CSG tree has *n* primitives, *n* distinct colors are assigned to different primitives so that the color code of the *j*th primitive is a binary number, with the *j*th bit set to 1 and all other bits set to 0. For a spatial point P, the color of P with respect to the *j*th primitive, $C_j(P)$, is defined as the color of the *j*th primitive if P is inside the primitive, and 0 otherwise. Now, if we combine $C_1(P)$, $C_2(P)$,…$C_n(P)$ using a logical operation OR or XOR, the result, C(P), is exactly the classification index of point P. Thus, for each Z-plane, algorithm generates a slice for each primitive, and then composites the slices from the primitives into one single slice of classification indices in the frame buffer using appropriate frame buffer pixel functions. This composition process is carried out as follows:

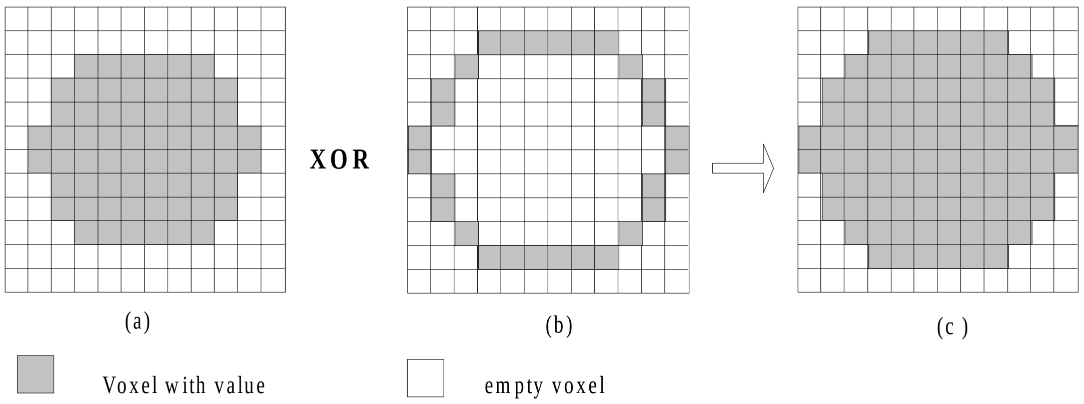


Figure 3.3 The process of solid voxelization for one slice by XOR

((a) the previous solid slice, (b) the boundary voxelization of current slice , (c) the final solid current slice)

For a primitive defined by its surface boundaries, since only the boundary surfaces are drawn, we need to have a way to determine the interior points [79]. The idea is based on the principle that when a ray is shot from a pixel towards the *j*th primitive in the viewing direction, it has to enter the primitive object first (*j*th color bit becoming 1) and stay there (keeping the *j*th bit) until it exits the object (changing the *j*th bit to 0). This can be done by drawing the boundary surfaces of each primitive with a logical XOR operation (Figure 3.3). When a slice is complete, the frame buffer will not be cleared, i.e. the frame buffer content of the slice will be used for blending operations with subsequent slices. This way, the XOR operation will automatically set the *j*th color bit to 1 for all interior points, and 0 for all outside points. Since the pixel colors on the slice generated by the *j*th primitive has 0's at all bit positions except the *j*th, the XOR operation for the *j*th primitive will have no effect to the classifications of other primitives.

In order for this algorithm to work correctly, the slicing process has to start from the outside of the object. In other words, the objects need to be completely contained by the volume space. Consequently, the volume space may need to be made very large. Since the size (resolution) of the volume is limited by the system memory, a large spatial region can lead to low resolution voxel representations of the object details. But for a correct operation, the slicing process has to start to overcome this problem; the voxelization process should be applied on the fly to only region of interest (e.g. the viewing region) of the scene. With a fixed sized volume representation, a region-based dynamic voxelization leads to a natural multi-resolution volume rendering approach with desired level of detail for different viewing regions.

This region based voxelization process, however, requires a small modification of the basic voxelization algorithm. The basic algorithm assumes that the slicing process starts from the outside of all primitives. But when the Z-plane starts from the front face of a sub region of the spatial domain, the XOR operation may not work correctly for all pixels since the parity is no longer guaranteed for every pixel. If we define the 0^{th} slice as the space between the first Z-plane of the region and the $z=-\propto$ plane, as shown in Figure3.2, the 0^{th} slice can be used to represent and store information about the starting status of each pixel. To generate the content of the 0^{th} slice, we can draw all the geometric primitives once using the 0^{th} slice as the viewing volume, with the XOR logical operation set for the frame buffer. The resulting frame buffer will not be cleared, i.e. the frame buffer content of the slice will be used for blending operations with subsequent slices. This way, the XOR operation will automatically set the jth color bit to 1 for all interior points, and 0 for all outside points. Since the pixel colors on the slice generated by the jth primitive has 0’s at all bit positions except the jth, the XOR operation for the jth primitive will have no effect to the classifications of other primitives.

CHAPTER 4

THE BLIST REPRESENTATION OF VOLUMETRIC CSG GRAPHS AND its EVALUATION

4.1 Introduction

In this chapter, the Boolean list (Blist) representation of volumetric CSG scene graphs will be given. In Blist formulation, a Boolean expression is represented as a list of primitives instead of a tree, and this may be evaluated in a pipeline fashion. Evaluation of the volume scene is carried out by z-buffer based voxelization algorithm [95]. This voxelization algorithm generates slices of objects using the 2D pixel processor, given in Chapter 2.

4.2 Binary Representation of CSG Trees

In a CSG model, primitive shapes are combined through regularized Boolean expressions. The primitive shapes define solids, which may be represented as parameterized primitives, such as cylinders or blocks, or as more general boundary or procedural representations. The Boolean expression combines the primitive shapes through union (U), regularized intersection (∩), and regularized difference (-) operations. It may be represented as a binary tree, whose interior nodes store the Boolean operators and whose leaves store integer references to the primitives in a table.

To make things more precise, we assume that the input CSG model is represented as a binary tree. With each node, *n*, is associated a structure with several fields. The field n.type specifies the type of the node: leaf or internal.

Internal nodes contain the following fields:

- n.type, which is equal to *node*
- n.operator, an operator (U, ∩, or -)

- n.leftChild, a pointer to the left child
- n.rightChild, a pointer to the right child
- n.parent, a pointer to the parent node (null for the root)

Leaf-nodes contain the following fields:

- n.type, which is equal to *leaf*
- n.parent, a pointer to the parent node
- n.primitiveReference, which identifies the corresponding primitive in a Table of Primitives, which contains a complete description of the primitive, for example: the primitive's type, parameters, position, and color.
- n.primitiveID, an integer used during the CSG-to-Blist conversion to identify the corresponding entry in the Blist table.

Many other popular CSG representations may be converted to this simple format. For instance, rooted, directed, acyclic CSG graphs may be expanded into binary CSG trees. CSG trees with transformation nodes may be converted to trees with only Boolean nodes by composing the transformations that are applied to each primitive and by storing the result in the table of primitives.

Let S be an r-set [96]. Let X be a candidate set: curve, surface, volume, edge-neighbourhood. A set membership classification process segments a candidate set, X, into three subsets: X∩iS; X-S; and X∩bS, where iS and bS stand respectively for the interior and the boundary of S. All set membership classification algorithms perform two tasks: (1) subdivide X into cells of uniform classification against primitives and (2) combine the binary results of classifying these cells against the primitives according to the corresponding Boolean expression. CSG-to-boundary conversion algorithms are often based on such set-membership classifications.

Subdivision typically involves computing intersections between the carrier of X and the surfaces that bound the primitives. Classification may in general be expressed as a combination of binary values, which represent the results of classifying the cell against the interior of the primitives. The combination uses Boolean operators (OR, AND, and AND NOT) for the corresponding CSG operators (U, ∩, and -).

The standard way of implementing the evaluation of such expressions [47] is illustrated by the pseudo-code of Procedure *ClassifyAgainstTree*, below, which is invoked using the cell and the root-node of the tree as arguments.

```
PROCEDURE ClassifyAgainstTree(cell,node)
    IF node.type = leaf
    THEN RETURN InPrimitive(node.primitiveReference, cell)
    ELSE RETURN Combine(ClassifyAgainstTree(cell,node.leftChild),
    node.operator, ClassifyAgainstTree(cell,node.right.child));
```

When the set membership classification process implements such a recursive procedure of evaluating the Boolean expression, it requires a stack of binary values, whose length equals the height, H, of the binary tree.

Note that the recursive procedure produced above may be improved by avoiding the *ClassifyAgainstTree*(cell,node.right.child)) call when its result is irrelevant. For example, if A returns FALSE, then A∩B is FALSE no matter what B returns. Similarly, when A is TRUE, then AUB returns true, no matter what B returns. The Blist technique proposed here avoids all these unnecessary calls.

The CSG tree described above could be represented in different ways. For example, using the inverse Polish notation, the CSG expression *(A∩ (BUC)) -((DU(E ∩F)) -G)* could be represented by the sequence of operations ###U∩### -U#∩-, where # pushes a new value on the stack and where the other operators combine the two top elements of the stack.. The expression could be evaluated without recursion. Nevertheless, the evaluation would use a stack, whose length would be equal to the height of the tree, at least when the CSG tree is full (i.e. when all leaves are at the same depth).

The storage required for this stack is usually not a problem when the cells are classified one at a time against the entire CSG model.

However, when the cells form a regular array of points in space or on the boundary of a primitive or when they form a pencil of rays, it may be more efficient to classify all cells against one primitive, before classifying them against the next primitive. For example, rasterization techniques may be used to classify all pixels or all voxels against a single primitive, and forward difference techniques may be used to compute the intersections of a family of rays with an algebraic surface. In such cases, either all the results of cell-primitive classification must be stored for all primitives before they are combined, or each step of the combine process may be performed as soon as its arguments are available.

The latter solution requires storing a stack of intermediate results for each cell.

Parallel implementations of algorithms that classify large amounts of cells (points, voxels, or ray segments) may assign one primitive to each processor or thread. A cell may be either classified simultaneously by all threads or may be classified against the primitives one-by-one in a pipeline fashion. The pixel-planes architecture [97] combines both approaches. In any case, these binary classification results for each cell must be combined according to the Boolean expression that defines the CSG solid. The combine process requires passing the results of the cell-primitive classifications between processors, and may be implemented as a hardware combine-tree, an array of p by log(p) processors. Because the array does not grow linearly with the number of primitives, it is difficult to extend such architectures to support larger CSG models without performing several passes or breaking the tree into subsets, for which intermediate results must be stored and recycled.

In Blist representation of Boolean expression, because it may be represented as a list of primitives, instead of a tree, and may be evaluated in a pipeline fashion, combining at each step the result of classifying the cell against the current primitive with the result of the previous classifications. The fundamental breakthrough provided here lies in the fact that the result of the previous classifications does not require the list of values of cell-primitive classification results, nor a stack of intermediate results of evaluating sub-expressions. Instead, Blist passes from one primitive to the next a simple label, which may be stored using at most log(H+1) bits, where H is the height of the CSG tree. Note that log(H) equals log(log(|P|)), where |P| is the total number of primitives [4].

Using the Blist representation, any Boolean expression can be evaluated without using recursion or a stack. In fact, the whole binary merging process is replaced by a simple comparison between the content of the label that is attached to the cell and the name of the primitive.

In the remainder of this chapter, we define the structure of the Blist table; propose a simple algorithm for converting volumetric CSG trees to a Blist form while minimizing the number of bits needed for each label; and provide an intuitive explanation of the essence of our approach. We demonstrate the conversion process and the evaluation algorithm on an example.

4.3 The Blist Representation

The Blist method transforms in some sense the CSG tree into a decision graph. A primitive classifies a candidate cell and depending on the result, forwards the cell to one or another primitive.

Blist represents a CSG expression as a table, called BL, of primitive entries. The entry BL[p] associated with primitive number *p* contains:

- BL[p].primitiveReference: The ***reference*** to the primitive's description, which includes its type, parameters.

- BL[p].sign: The ***sign*** (binary value) indicating, when set, that the result of classifying a cell against the primitive should be complemented.

- BL[p].name: The ***name*** associated with the primitive (several primitives may have the same name and many primitives have no name).

- BL[p].stamp: The ***stamp***, which contains the name of the next primitive in the list that should classify the cell that are inside the current primitive if its sign is positive, or the cells that are outside of the current primitive, if its sign is negative.

The Blist representation can be applied to any CSG tree. However, to simplify definitions and proofs, each algebraic representation of the Boolean function associated with a CSG tree is replaced by its positive form, which uses only regularized + and . operators, but represents rigorously the same Boolean function and thus the same solid as the original tree.

4.3.1 Positive Form

The three regularized operators, union, intersection, and complement, form a Boolean algebra over regular sets [86]. Therefore, De Morgan’s laws [47] can be invoked to manipulate CSG expressions.

Any CSG tree may be converted into its positive form by a preorder traversal of the tree applying, when appropriate, the following transformations to each node:

$X-Y=X.\,Y'$,

$\overline{X+Y}=\overline{X}.\overline{Y}$,

$\overline{X.Y}=\overline{X}+\overline{Y}$

$\overline{\overline{X}}=X$

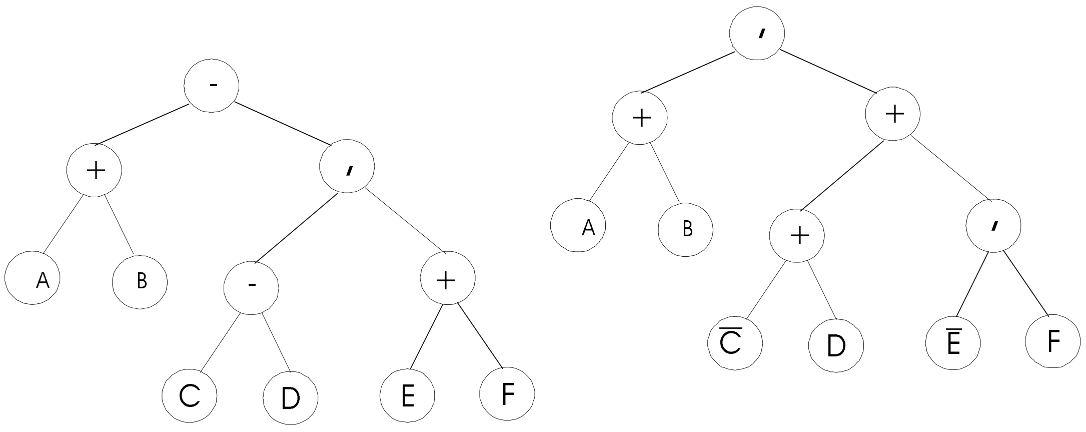


Figure 4.1 Tree reformulation: (AUB)-(CU(DUE))∩(FUG) is transformed into its positive form: (AUB)∩-(C’∩(D’UE))U(F’UG’).

This reformulation (illustrated in Figure 4.1) exchanges the . and + operators at negative internal nodes, replaces - operators by . if the corresponding node is positive and by + otherwise, and replaces negative primitives by their regularized complements. The resulting positive form contains only . and + operators. In a positive form, all nodes are positive.

The set represented by the entire positive form is equal to the solid represented by the original CSG tree, and hence is bounded (assuming the original primitives are bounded).

Any CSG tree can be expressed in a positive form, and thus, without loss of generality, we shall use a positive form throughout the rest of this thesis.

4.3.2 The Blist Formulation of Volume Scene Graph

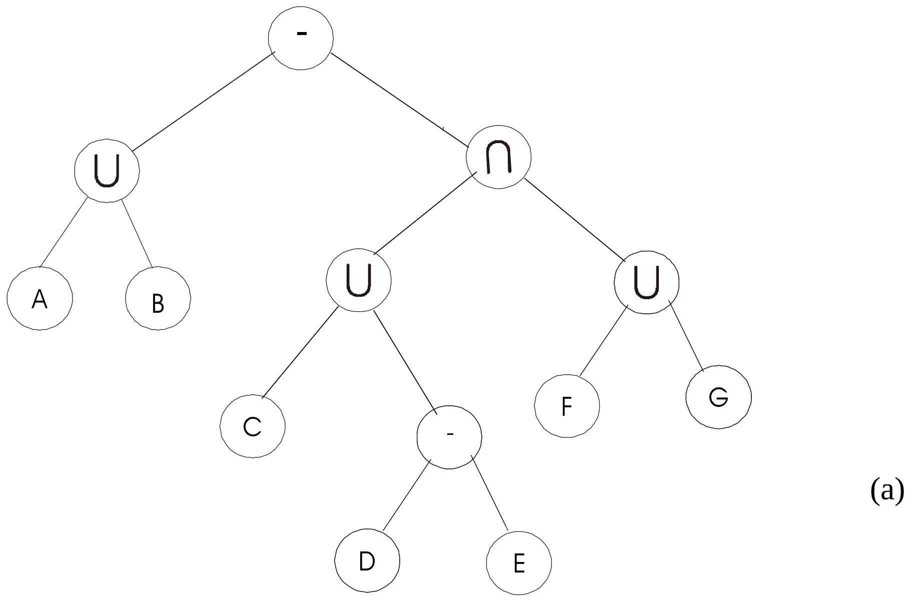
In Blist formulation of the volume scene tree each primitive represents a volumetric dataset or a geometric model. The basic idea of our algorithm is to generate a slice for each primitive in the list first, then to evaluate Blist by updating a label, when its value matches the primitive’s name. At the end if the label on the voxel is 1(IN name) the voxel is inside the volume scene otherwise, it is outside the volume. So, the Boolean expression of the volume scene tree is evaluated directly, i.e. combining steps used with the traditional recursive evaluation is not necessary. The details of Blist formulation of the volume scene tree are given below.

The CSG-to-Blist conversion process takes, as input, the root-node of the binary tree, T, and produces the corresponding BL table. Both structures have been described above. The conversion performs the following steps:

1. Convert T into a positive form by applying deMorgan’s laws and propagating complements to the leaves.
2. Rotate the tree by switching the left and right children at each node to make the tree left heavy.
3. Visit the leaves from left to right and for each leaf, p, fill in the corresponding fields of BL[p].

For unbalanced trees, Step 2 may reduce the total number of bits needed for each label to less than [log(H+1)].

We describe the details of these steps below using as example T= (AUB)-(CU(D-E))∩(FUG). The literals, A, B…G, denote integer primitive references. Parsing this expression yields a binary tree shown in Figure 4.2 (a).



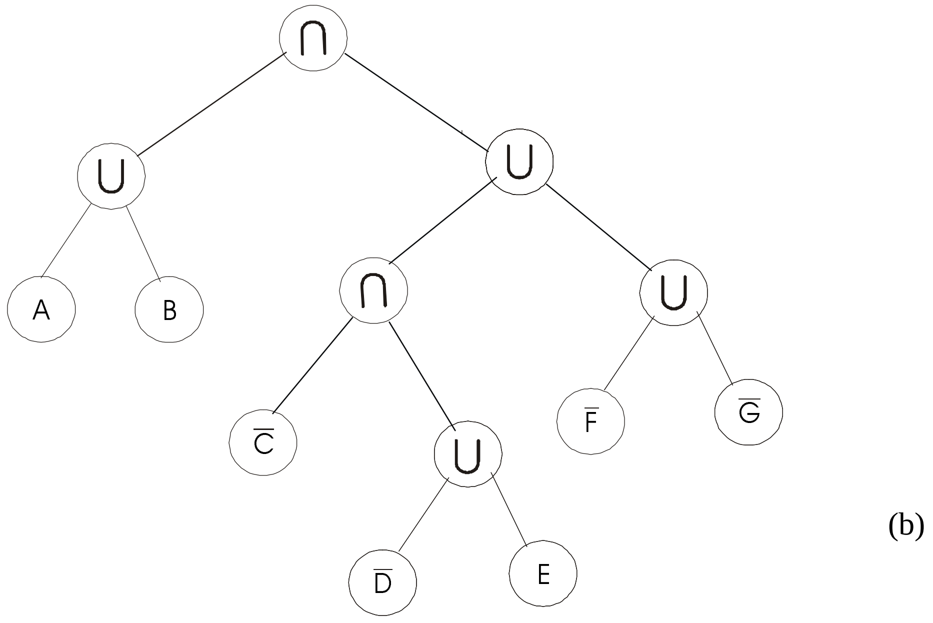
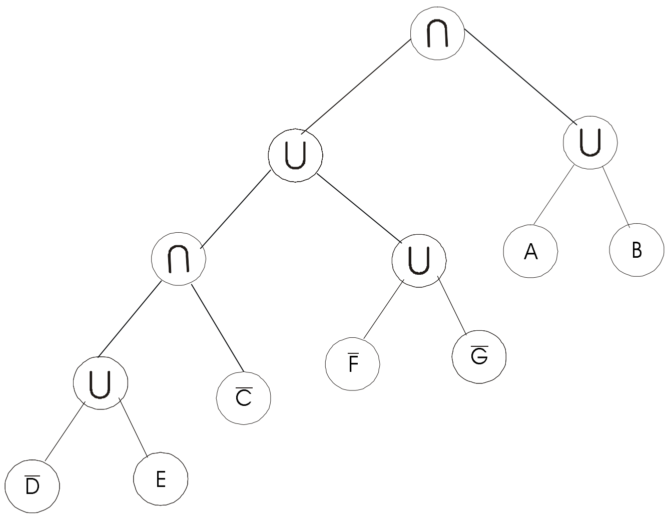
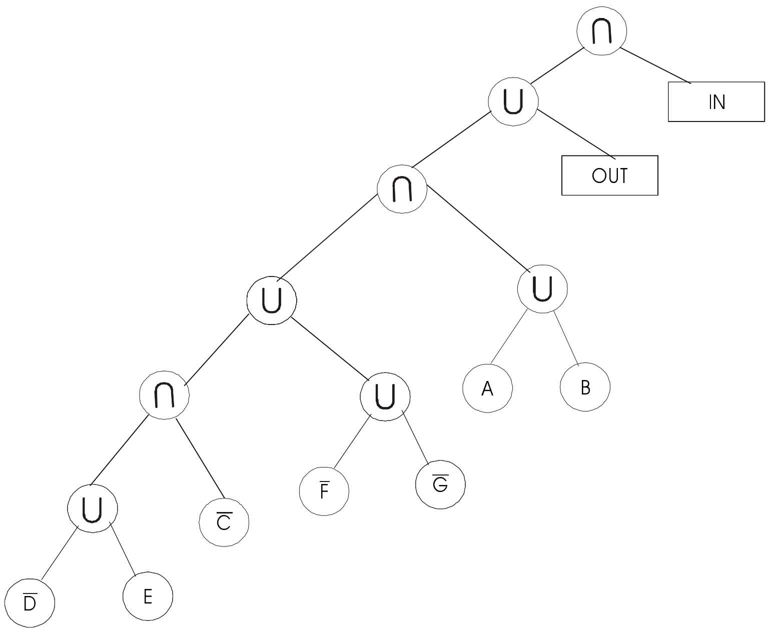


Figure 4.2 The binary tree corresponding to the Boolean expression is shown in (a). Its positive form is shown in (b). Note the complemented primitives are indicated using overscores.



(a)



(b)

Figure 4.3 (a) The binary tree of Fig. 4.2 has been rotated to make it left heavy. (b) The result is inserted as the left-most leaf into a small tree with special IN and OUT nodes marked by rectangles.

We first convert that tree to its positive form (as in [47]). This conversion process traverses the tree top-down and applies the DeMorgan transformations: A-B→A∩B**'**, (A∩B)**'**→A**'U**B**'**, (A∩B)**'**→A**'U**B**'**, and (A**'**)**'**→A, where X**'** denotes the complement of set X. The result (Figure 4.2 (b)) is a tree with the same structure and no difference operators. Note that some of its leaves (indicated by an apostrophe) have been negated, i.e., replaced by their complements.

Then, we exploit the fact that both U and ∩ are commutative (AUB-BUA and A∩B-B∩A) to switch the left and right child of nodes when the right child is a higher sub-tree (i.e. has a superior maximum path length from its root to its leaves). The switch is performed during a recursive traversal of the positive tree, switching first the lower-level nodes and reporting their height to the parent node, before we consider switching the parent node. The result is illustrated Figure 4.3 (a).

Finally, we insert the resulting tree, T, as the left-most leaf of a two level tree: (T UOUT)∩IN, as shown Figure 4.3 (b) and we traverse the new tree and assign consecutive integer ID's, P.primitiveID, to each leaf, P, in left-to-right order.

Now we are ready for the final phase, which fills in the Blist table BL. During that phase, we initialize the content of BL to zero and, once more, traverse the rotated positive version of the tree T recursively. At each leaf, P, we invoke the procedure *Match*(P,BL) illustrated by the pseudo-code below.

```
Procedure Match(P,BL)
(a) p := P.primitiveID;
(a) IF BL[p].name ≠0 THEN releaseIntegerName(BL[p].name);
(c) BL[p].sign := P.sign;
(d) BL[p].primitiveReference := P.primitiveReference;
(e) M := P;
(f) WHILE M ≠M.parent.leftChild DO M := M.parent;
(g) op := M.parent.operator;
(h) IF op = "∩" THEN BL[p].sign := NOT BL[p].sign;
(i) M := M.parent;
(j) WHILE (M≠M.parent.leftChild) OR (M.parent.operator = op) DO M := M.parent;
```

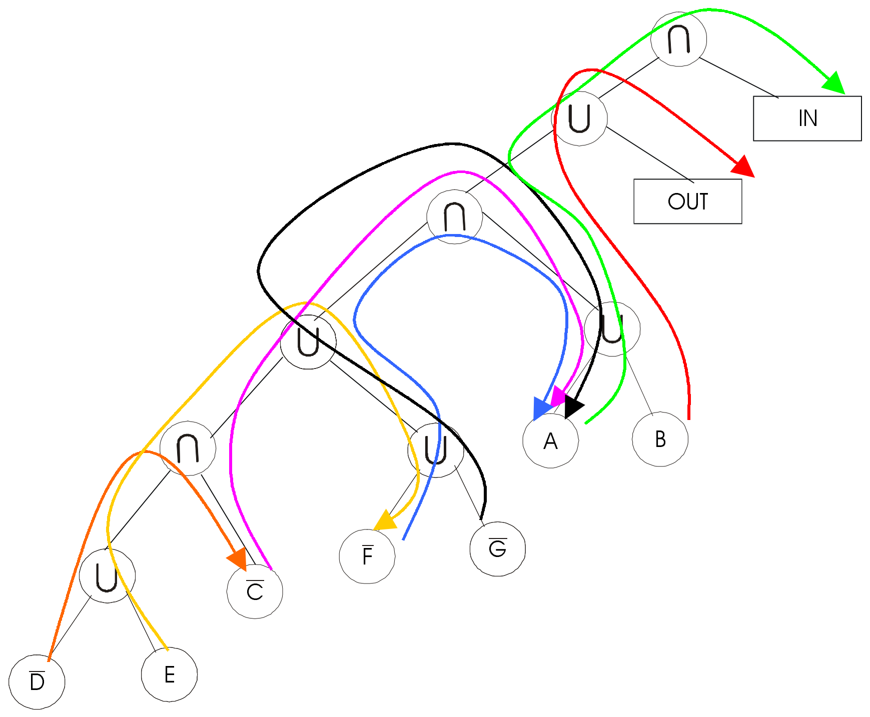
```
(k) M := M.parent.rightChild;
(l) WHILE M.type ≠leaf DO M := M.leftChild;
(m) m := M.primitiveID;
(n) IF BL[m].name = 0 THEN BL[m].name := lockLowestAvailbleIntegerName;
(o) BL[p].stamp := BL[m].name;
```

Procedure *Match* is based on the following observation. Consider a sub-tree S-((P∩B)∩C)U((MUE)∩F) in T. If we are classifying the voxel v against S and discover that it is outside of primitive P, then we must classify it against primitive B, which is the next one in the Blist representation. Therefore, marking the voxel with a zero label will indicate to the next primitive, B, that it should process the voxel. If, however, we discover that $v \subset P$, we can skip primitives (or sub-trees) B and C and go directly to M, which is the left-most leaf of the sub-tree ((MUE)∩F). Consequently, we must write on the label of v the name used by M.

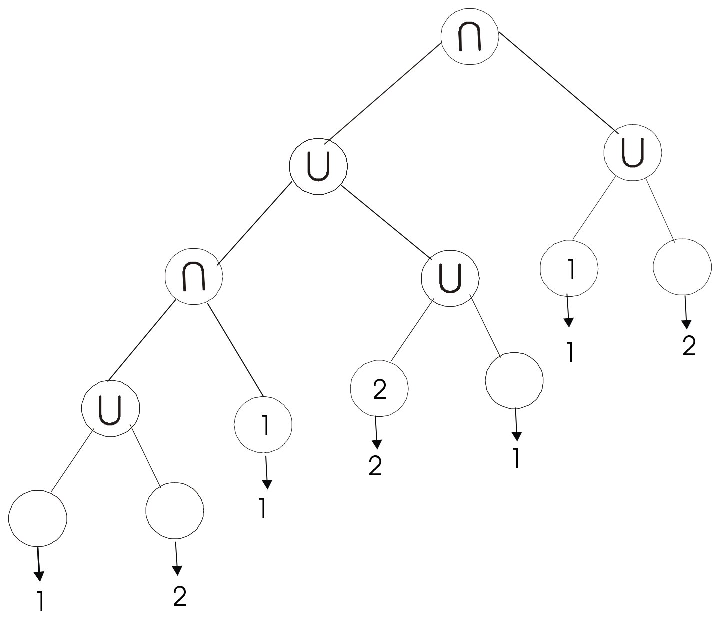
If M does not yet have a name, we use the procedure *lockLowestAvailbleIntegerName* (line n) to obtain the lowest strictly positive integer that is not yet in use as the name of any primitive coming after P in the Blist table. Note that when later we reach M, that integer is released using procedure *releaseIntegerName* (line b), so that it can be used as a name for another primitive that comes after M. The same name is often used multiple times. This strategy helps reduce the maximum number of bits needed for labels.

Given the current primitive, P, we locate its match, M, by moving up the tree (line f), until we reach the left child of a node N1. We detect this situation because it is the first time that M is the left child of its parent. We save the operator of N1 in the variable: op. In the above example of the sub-tree S, the operator for N1 is U.

However, if it was ∩, as for example in the sub-tree S-((P∩B)∩C)U((MUE)∩F), we would jump to M, only if $v \not\subset P$, or would continue to the next primitive, if $v \subset P$. To distinguish between these two situations, we toggle the sign associated with P (line h), when the variable op is "∩".



(a)



(b)

Figure 4.4 (a) blist conversion process of volume scene tree (b) converted tree

Then, we keep walking up the tree, until we reach a node, N2, which is the left-child of a node, whose operator is different from op (line j). The desired "match" leaf, M, is the left-most leaf in the sub-tree that is the right child of N2. We move to the right child first (line k), and then walk down the tree, always turning left (line l). M is the leaf where this journey ends.

If the name attributed to the IN node was not 1, but x, we simply switch all uses of x'es and 1's in the names and stamps stored in BL, so as to follow the convention that all voxels marked 1 at the end of the classification process are in.

Figure 4.4 (b) shows the modified tree to be converted to Blist representation.

In Figure 4.4 (a) the leaves of tree of are visited in the left to right order. We number the leaves with increasing positive integers. For this tree there are seven primitives (p 1:7). For each leaf, numbered p, we compute its match. Given the current primitive, D (p=1 in the table), we locate its match, M, by moving up to the tree until we reach the left child of node N1 (in this tree N1 is U). We detect this situation because it is the first time that D is the left child of its parents and we save the operator of N1 as op (in this case op=U). If N1.op=∩, the sign of p, Blist.sign is inverted. For the primitive D, since N1.op=U, in the table Blist.sign= +. Then we keep walking up to the tree until we reach a node N2, that is the left-child node whose operator is different from op (for the given primitive N2=∩). The desired match, leaf, M, is the left-most leaf in the subtree that is the right child of N2 (for the given primitive match leaf is C). Then if M (C leaf) does not yet have a name, it grabs the lowest available positive integer and use it as its name (in the table C.name=1). We also store that name as the stamp, Blist[p].stamp of p (in the table for primitive D, Blist[1].stamp=1). In (Figure 4.4 (b)) we showed the resulting names for leaves and their stamp.

Blist represents a CSG expression in the form of a table, called BL, of primitive entries. Here *BL[p].primitive reference* is the reference to the primitive's description, which includes its type, parameters, color. *BL[p].sign* is the binary sign value, when set, the result of classifying a voxel against the primitive should be complemented. *BL[p].name* is the name associated with the primitive. And *BL[p].stamp* contains the name of the next primitive in the list that should classify a voxel inside the current primitive, if its sign is positive, or outside, if its sign is negative.

The Blist table resulting from the conversion process for the example of Figure 4.4 is given in Table 1.

Table 1 Blist table resulting from tree conversion process of Figure 4.4

P	Bl[p].name	BL[p].sign	BL[p].Primitive Reference	Bl[p].stamp
1	0	-	D	1
2	0	-	E	2
3	1	-	C	1
4	2	+	F	2
5	0	+	G	2
6	1	+	A	1
7	0	+	B	1

4.3.3 Voxel classification using Blist

During set membership classification, a ***label*** is attached to each voxel and passed to the successive primitives in the Blist. When the label matches the primitive's name, the voxel is classified against the primitive. If the result of this classification matches the primitive's sign, the name on the primitive's stamp is put on the label—if not, a zero name is put on the label of the voxel.

Table 2 Blist table after labels are evaluated

P	Bl[p]. name	BL[p]. sign	BL[p].Primitive Reference	Bl[p]. stamp	Bl[p]. Class	Bl[p]. label
1	0	-	D	1	0	1
2	0	-	E	2	1	1
3	1	-	C	1	0	1
4	2	+	F	2	1	1
5	0	+	G	2	0	1
6	1	+	A	1	0	0
7	0	+	B	1	1	1

The following procedure describes how a single voxel, v, is classified against a volumetric CSG tree represented by a Blist array BL. |P| defines the total number of primitives in BL.

```
PROCEDURE ClassifyAgainstBlist(v,BL)
v.label := 0;
FOR p := 1 TO |P| DO
IF (v.label = 0) OR (v.label = BL[p].name)
THEN IF InPrimitive(BL[p].primitiveReference , v) = BL[p].sign
THEN v.label := BL[p].stamp
ELSE v.label := 0;
```

At the end of this process, if the label on the voxel is 1 the voxel is inside the CSG solid. Otherwise, it is out.

To classify a set V of voxels, we propose a simple extension of the above procedure:

```
PROCEDURE ClassifyvoxelsAgainstBlist(V,BL)
FOREACH voxel v in V DO v.label := 0;
FOR p := 1 TO |P| DO
FOREACH voxel v in V DO
IF (v.name = 0) OR (v.label = BL[p].name)
THEN IF InPrimitive(BL[p].primitiveReference , v) = BL[p].sign
THEN v.label = BL[p].stamp
ELSE v.label = 0;
```

To illustrate how this classification works, consider for example a voxel whose classification against the sequence of primitives, A,B,C,D,E,F,G, yields the following position of bits: 0100110.

Originally the label is 0 (Table 2), the voxel is out of the first primitive D, and since the primitive is inverted BL[1].sign= -1, we set the label to contain BL[1].stamp which is 1. We skip the primitive 2, F, because its name does not match the label. At primitive 3, we have match between the label and its name, the classification of the voxel against the C yields zero and since the primitive is inverted we write content of BL.stamp into the label which is again set to 1, we skip primitive 4 and 5, because their names are not 1. Then in primitive 6, the label matches its name, but classification of the primitive does not match its sign so we put 0 to label. At primitive 7, B, the label matches its name and since classification matches its sign we put its stamp to the label which is again 1. So, at the end of this process, the label has a value 1, which indicates the voxel is inside the CSG volume.

4.4 Volume Slice Generation

Voxelization algorithm introduced in this chapter is based on creation of volume slices using depth information of front and back surfaces and used for voxelization of line, polygon and solid objects.

The voxelization algorithm generates the volume representation for a given region in a 3D scene. We define a region of interest, a regular 3D box with side faces parallel to the axis planes of the viewing coordinate system, as seen in Figure 4.5.

The algorithm generates slices of the object using the 2D CSG processor, given in Chapter 2, to form the final volume representation. Using this display algorithm both convex and concave objects can be voxelized correctly. Also clipping capability of display algorithm provides us to construct more complex volumes and objects using simpler ones.

Using a slice-by-slice approach, the voxelization algorithm proceeds moving Z-plane parallel to the projection plane with a constant step size in a front to back order. The thin space between two adjacent Z-planes within the volume space is called a slice. Since the Z planes (Z-plane(i)-Z-plane(i+1)) are used as clipping planes, the clipping capability of the CSG processor provide that only the part of surfaces within planes are displayed. The resulting frame buffer image from the display of this slice will be the one of the slice of the resulting volume.

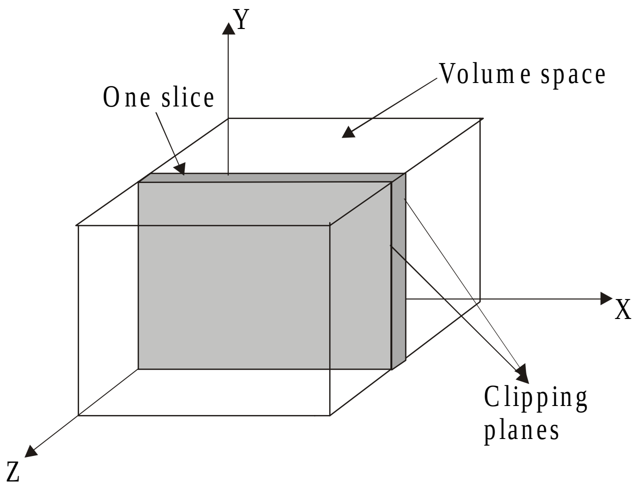


Figure 4.5 Volume space slicing

Since we have both front and back surfaces, depth/color values of the object the processor used in the slice generation puts the color of the front clipping plane into the object interior voxels, they can be displayed directly. Thus, there is no need for extra computations to fill the interior voxels of solid objects like in [1,7,9]. In these algorithms, to generate interior voxels, algorithms employ a frame buffer blending function with a logical XOR operation to carry boundary information to the interior of the solid object. And the parity rule used in solid voxelization may not be correct, if vertex or some point on edge of surface primitive happens to project to the exact center of a pixel.

4.5 Blist Evaluation of Volume Scene Tree

To evaluate volume scene tree, after generating a slice for each primitive in the list, we evaluate Blist by updating a label, when its value matches the primitive's name. At the end, if the label on the voxel matches the IN's name the voxel is inside the volume scene otherwise, it is outside the volume. So, the Boolean expression of the volume scene tree is evaluated directly, i.e. combining steps used with the traditional recursive evaluation is not necessary.

After filling Blist table (explained in the previous section), for the set membership classification a label is attached to each voxel and is passed to the successive primitive in the Blist. When the label matches the primitive's name, the voxel is classified against the primitive. If the result of this classification matches the primitive's sign, the name on the primitive's stamp is put in the label. If not, 0 is put in the label of the voxel. At the end of this process, if the label on the voxel matches the IN's name, the voxel is inside the volume scene tree, otherwise it is out. In this representation, each voxel has a classification status with respect to all primitives. If the voxel lies in several primitives that overlap, in this case the result will be order dependent.

```
Blist Evaluation(VST_tree) {
 for(each slice S) {
 v.label=0; v.color=0;
 for(each primitive P in Blist table) {
 voxelize P into the slice S;
 If ((v.name=0) or (v.label=BL[p].name))
 If (Inprimitive(BL[p].primitive reference=BL[p].sign){
 v.label=BL[p].stamp;
 If (Inprimitive(BL[p].primitive reference && v.label=IN.name)
 v.color=BL[p].color;  }
 else v.label=0;
 move to the next primitive P in Blist table; }
 put v.color to the slice of volume memory;

 move to the next slice; } }
```

CHAPTER 5

SOFTWARE SIMULATION and RESULTS

5.1 Introduction

In this chapter, the software simulation of the developed algorithms will be presented. For the simulation, a program was written in the C programming language. Also, the performance analysis is done, and voxelization and Blist evaluation methods are compared with the previous algorithms.

5.2 Data Format for the 2D CSG Display Processor

The data of the object primitives used in the simulation program are stored in text files. The following example contains the required data for drawing a cube on the screen.

```
46.06     -1.0      1.18    2

1         0         0

38.28     1.0       0       1

0          0        0

-131.42  1.0        0       1

1         0         0

328.24    -1.0      -1.68   3

1         0         0

570.25    -1.0      -1.68   6

0         0         0

193.75  -1 1.18    4

0         0         0

0         0         0       1234
```

The data set per plane, belonging to object primitive intended, consists of two rows:

The first one comprises Z0, dZx, dZy, and the colour of the plane. The second one is for external variables (H, Y, L) required to complete the definition of the plane. In the example, the first defines a plane, such that its depth, at the origin of the display window, is 46.06, the depth increment in X direction is –1.0, the depth increment in Y direction is 1.18, and its colour is green (2 corresponds to green). The plane is a front plane (H=1), is not perpendicular to the screen (Y=0), belongs to a convex object (L=0). There are 6 sets in the display list as it defines a cube. The last line in the list is a dummy entry since the program senses the end of the display list when the colour value is greater than 1000.

5.2.1 Constructing an Image

Before beginning to draw an image, the user must be aware of the fact that the size of the display window is 250x250, and the origin of the display space is at the top left corner of the window. As an example to image construction, think of two boxes, box2 being removed from box1, i.e. box1 is clipped by the complement of box2.

The display data for box1 shown in Figure 5.1(a) is:

58.57	-1.0	1.41	5
1	0	0	
42.42	1.0	0	6
0	0	0	
-127.27	1.0	0	3
1	0	0	
281.42	-1.0	-1.41	4
1	0	0	
501.42	-1.0	-1.41	1
1	0	0	
218.57	-1.0	1.41	2

0	0	0	
0	0	0	1234

and for box2 shown in Figure 5.1(b)is:

18.57	-1.0	1.41	2
1	0	0	
14.14	1.0	0	5
0	0	0	
-98.99	1.0	0	6
1	0	0	
321.42	-1.0	-1.41	1
1	0	0	
461.42	-1.0	-1.41	3
0	0	0	
258.57	-1.0	1.41	5
0	0	0	
0	0	0	1234

The mathematical expression of this is $@(box1.\overline{box2})=@(box1).\overline{box2}+@(\overline{box1}).box2$. This means that the image is constructed by adding (second-stage selection) the object firstly obtained by clipping box1 with the complement of box2 to the object then obtained by clipping the complement of box2 with box1. To perform this operation user selects 'Clipping' option in the menu. Then the program asks for the name of the files containing the clipping display data (in this case first the filename of box1 and that of box2). Then enter the clipping function. The clipping function for this case is A.!B (A stands for box1 and B stands for box2). Then type the name of the file, which will be storing the final image. The resulting picture of clipping box1 with the complement of box2 is shown in Figure 5.1 (e). User can also display the resulted image step by step. The Figure 5.1 (c) shows box1, which has been clipped by the complement of box2 (A.!B). And Figure 5.1 (d) shows the complement of box2, which has been clipped by box1 (B'.A). With the

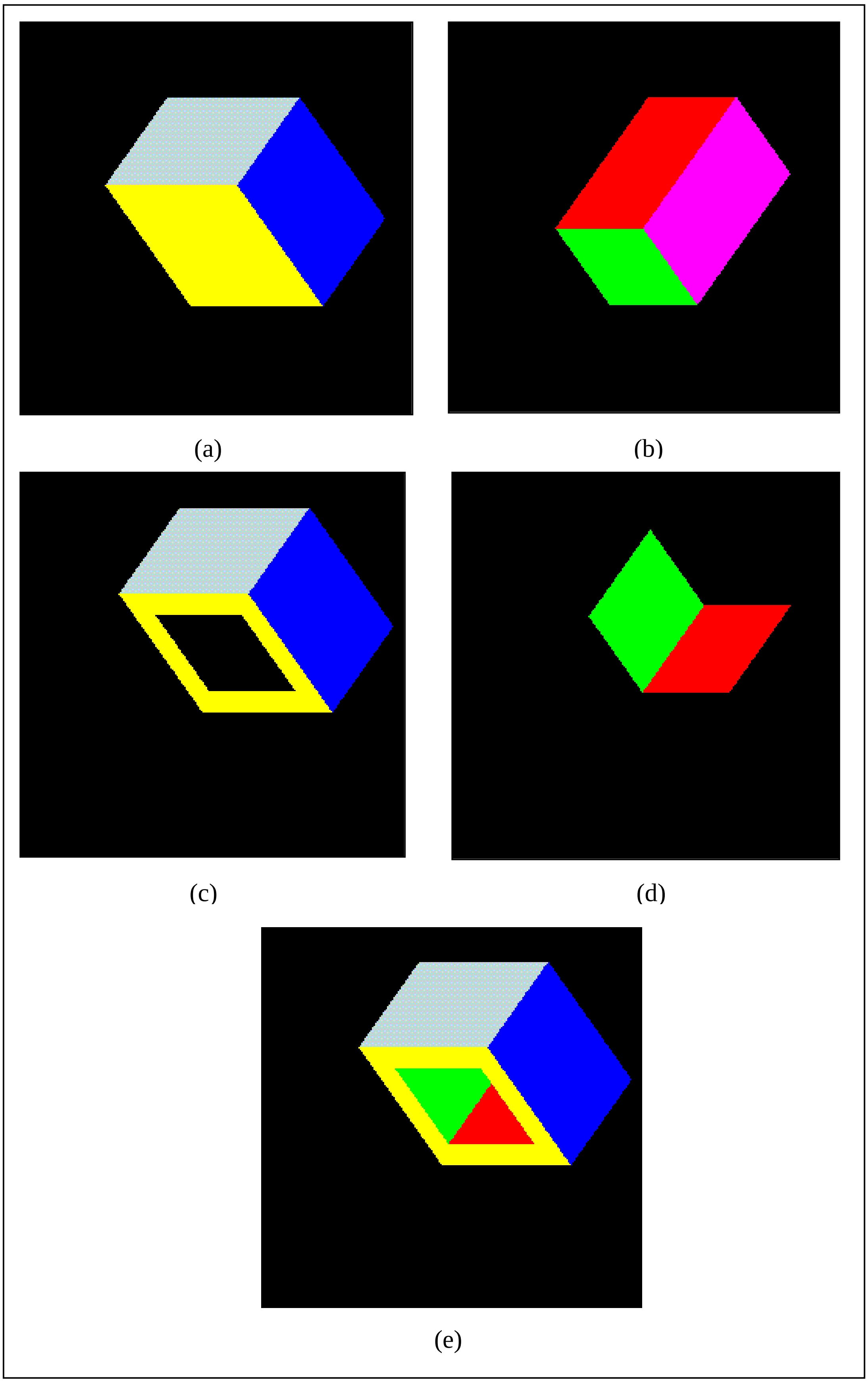


Figure 5.1 Clipping box1 with the complement of box2.

‘Clipping’ operation more complex images can be constructed using a desired number of objects.

5.3 Volume Slice Generation Using the 2D CSG Processor

Using the 2D CSG processor it is possible to clip CSG objects with CSG volumes and surfaces.

Using a slice-by-slice approach, the algorithm moves a Z-plane, parallel to the projection plane with a constant step size, in a front-to-back order. Since the Z planes (Z-plane(i)-Z-plane(i+1)) are used as clipping planes, the clipping facility of the CSG processor will ensure that only the part of the volume within the planes is displayed. The resulting frame buffer image from the display of this slice will be the one of the slices of the resulting volume.

For example, in Figure 5.2, the expression of the CSG tree can be written as:

((Cube1)U(Cube2)) – ((Cylinder1)U(Cylinder2))

It becomes the following, after converting it into the postfix form:

(Cube1) (Cube2) U (Cylinder1) (Cylinder2) U –

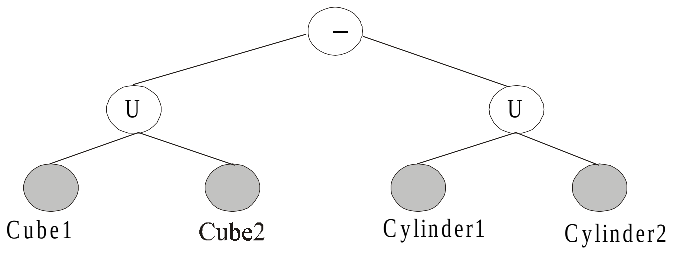


Figure 5.2 A CSG tree example

For each object in the CSG tree the display processor generates slices of the volume memory. For example, to generate volume slices of Cube1 the program uses the following clipping function: Cube1.Vz. Here the Vz is a clipping volume constituted between Z-plane(i)-Z-plane(i+1). The Z value changes between 1 and the maximum depth value of the object and its up to user to determine the number of slices. The clipping function, for each different Z values (that means for each different volumes), displays the part of Cube1, which stays in volume Vz. Since the processor puts the color of the front clipping plane into the object interior voxels, the

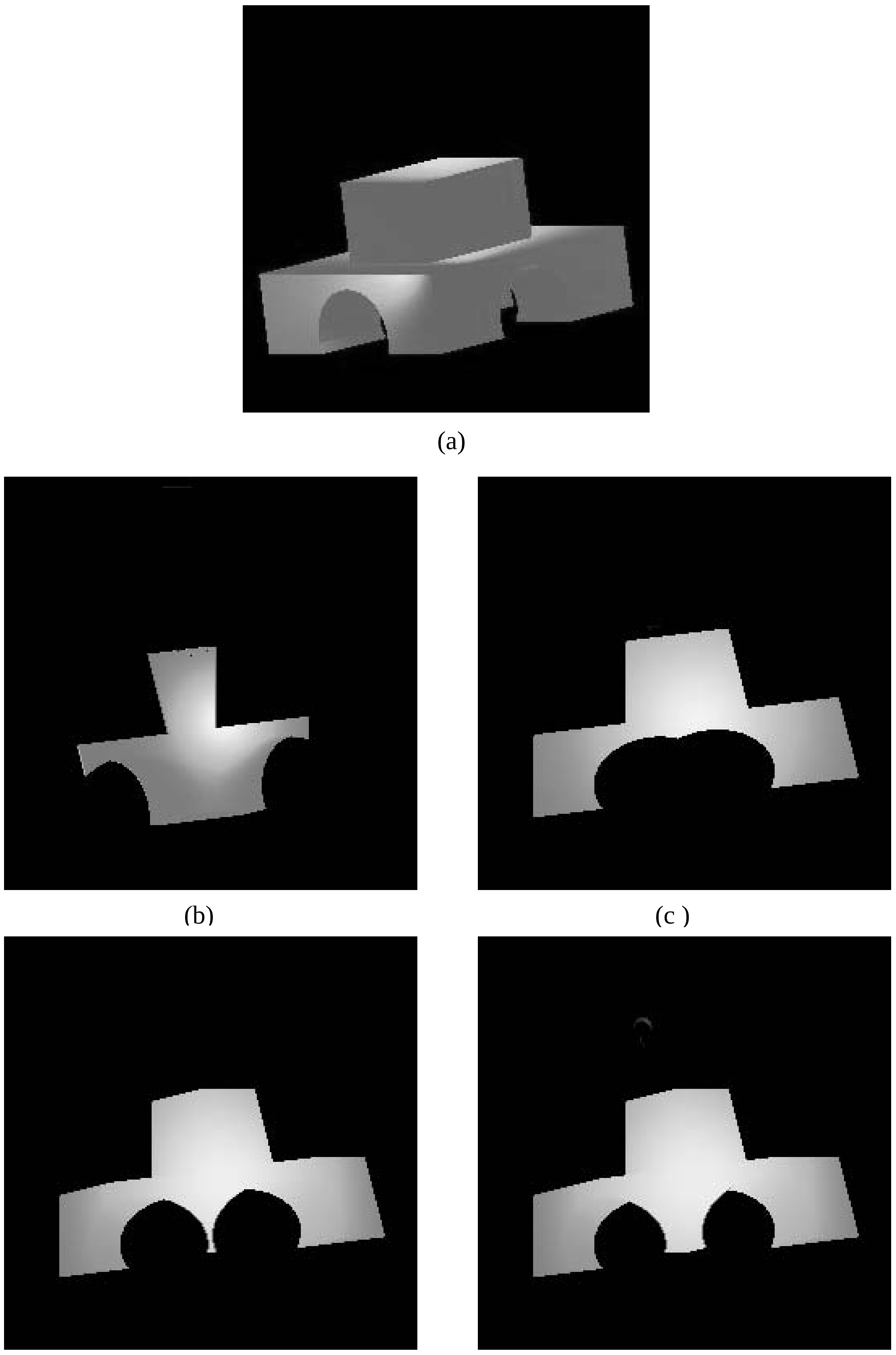


Figure 5.3 (a) Voxelization result of a CSG object. (b), (c), (d), (e) Several slices in the volume memory.

color of each object can be displayed directly. There is no need to make extra computations, like to fill in the interior voxels or shading of solid objects.

Using our program it is possible to voxelize the complete CSG tree using the tree expression given above. In this case our clipping expression becomes:

((Cube1)U(Cube2)U – (Cylinder1)U(Cylinder2)).Vz

Figure 5.3 shows some slices of the given CSG tree.

5.3.1 Handling of Polygon and Line Primitives

Besides CSG objects, the system can handle objects made of polygons and lines. The edges of each polygon making up the objects, is encoded in linear equations of the form Ax+By+D=0, which is a perpendicular plane to the screen. These planes cut another plane of the form Ax+By+Cz+D=0 so that resultant clipped plane is the required N-sided polygon. Thus, to render an N-sided polygon N+1 planes are required. The method is illustrated in Figure 5.4. Line primitives are constructed in the same way as seen at the bottom of the figure.

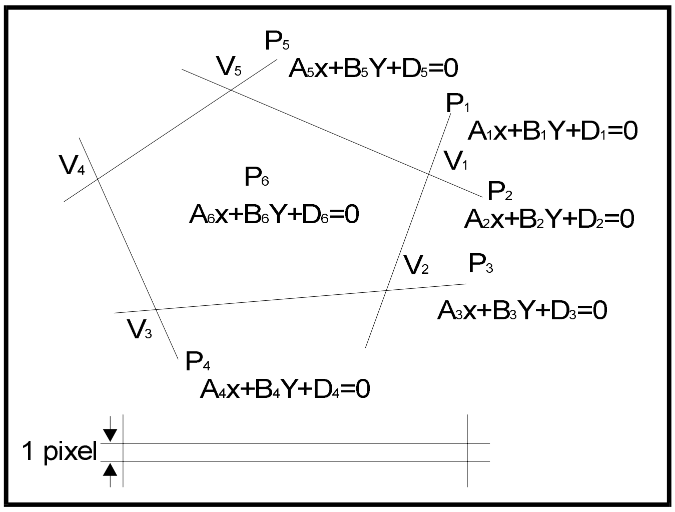
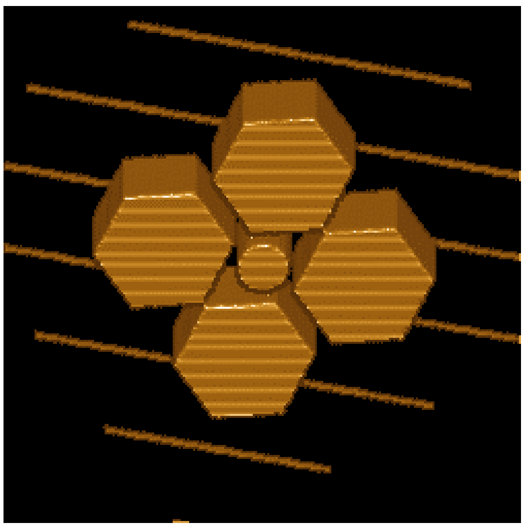


Figure 5.4 Polygon and line handling

After constructing line and polygon primitives the CSG processor can voxelize them succesfully. Figure 5.5 shows voxelized polygon and line primitives. In (a) the line thickness is more than one pixel and in (b) the line has one pixel thickness.



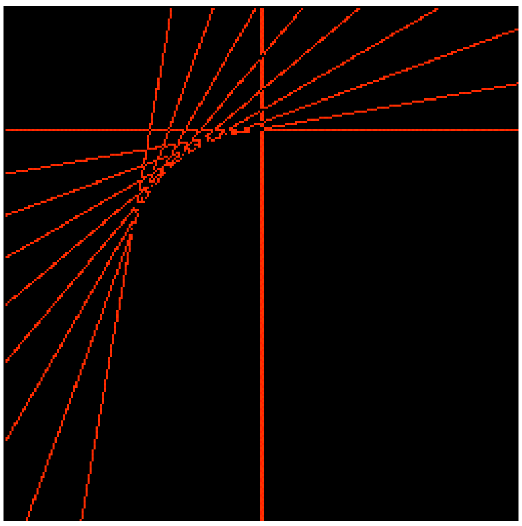


Figure 5.5 Polygon, and line voxelization examples

5.4 Blist Evaluation

In Blist formulation of the volume scene tree each primitive represents a volumetric dataset or a geometric model. The Blist method transforms the CSG tree into a decision graph. A primitive classifies a candidate voxel and depending on the result, forwards the voxel to one or another primitive.

Table 5.1 Blist table of volumetric scene tree in Figure 5.6.

P	BL[p].name	BL[p].sign	BL[p].Primitive Reference (or color)	Bl[p].stamp
1	0	+	A	1
2	0	+	B	1
3	0	-	C	2
4	1	+	D	2
5	0	-	E	1

The Blist evaluation program, firstly, transforms the tree into a positive form, then rotates the tree by switching the left and right children at each node to make the tree left heavy, and then inserts the resulting tree, T, as the left-most leaf of a two level tree: (T U OUT)∩IN. Then the leaves are visited from left to right for each leaf, p, and finally, corresponding fields of a table BL[p] are filled. Figure 5.6 (a) shows the volume scene tree. Here, cube1, cube2, cylinder1 and cylinder2 represent a volumetric CSG object produced with the CSG processor and CT head is a volumetric dataset obtained by computed tomography. Figure 5.6 (b) shows conversion process and Figure 5.6 (c) shows the modified tree to be converted to Blist representation.

Figure 5.7 shows the resulting image of the given volumetric tree, and Figure 5.8 shows another example whose volume data is constructed with the Blist evaluation method.

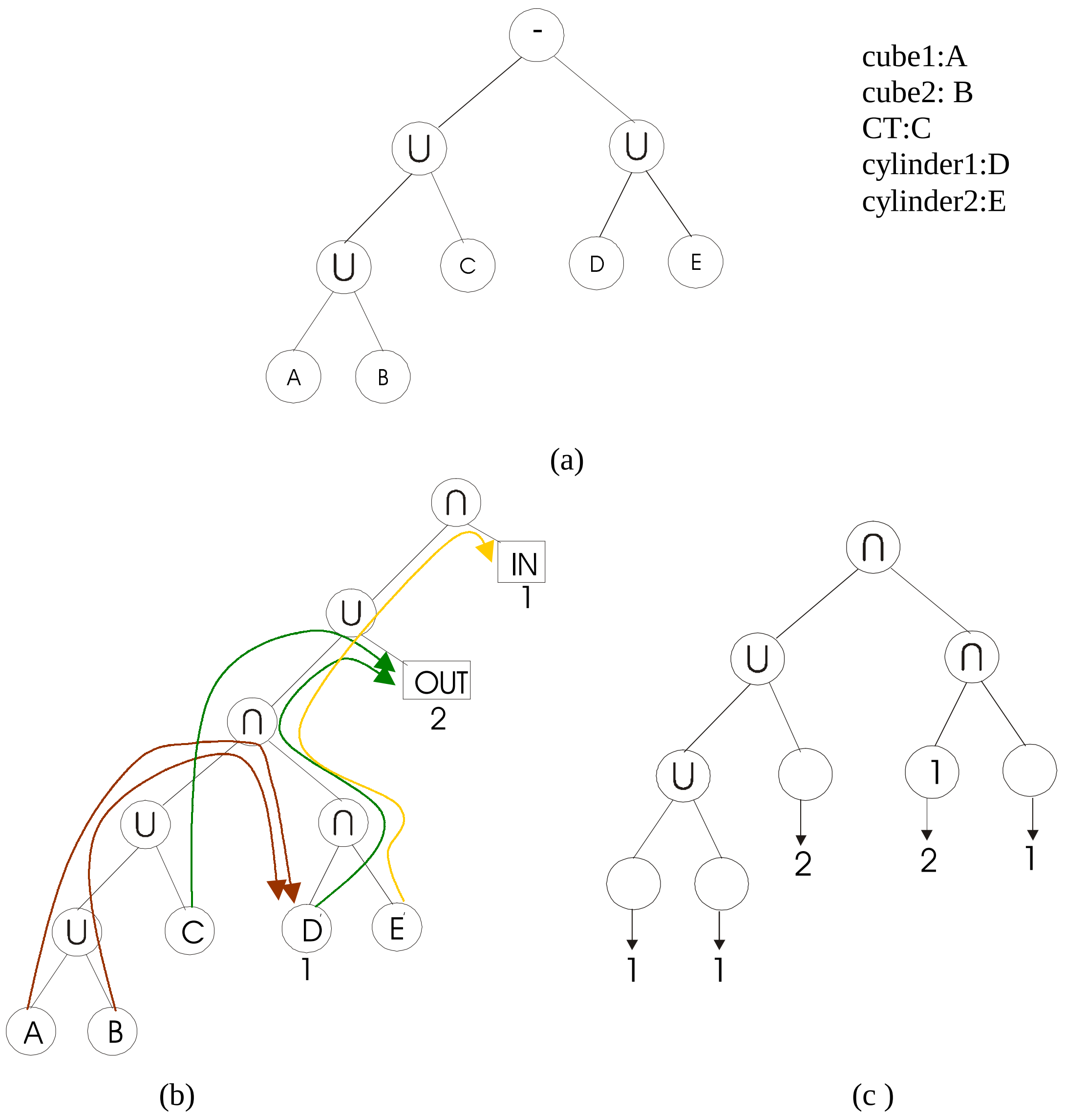


Figure 5.6 (a) Volume scene tree
(b) Blist conversion process of the volume scene tree
(c) Converted tree

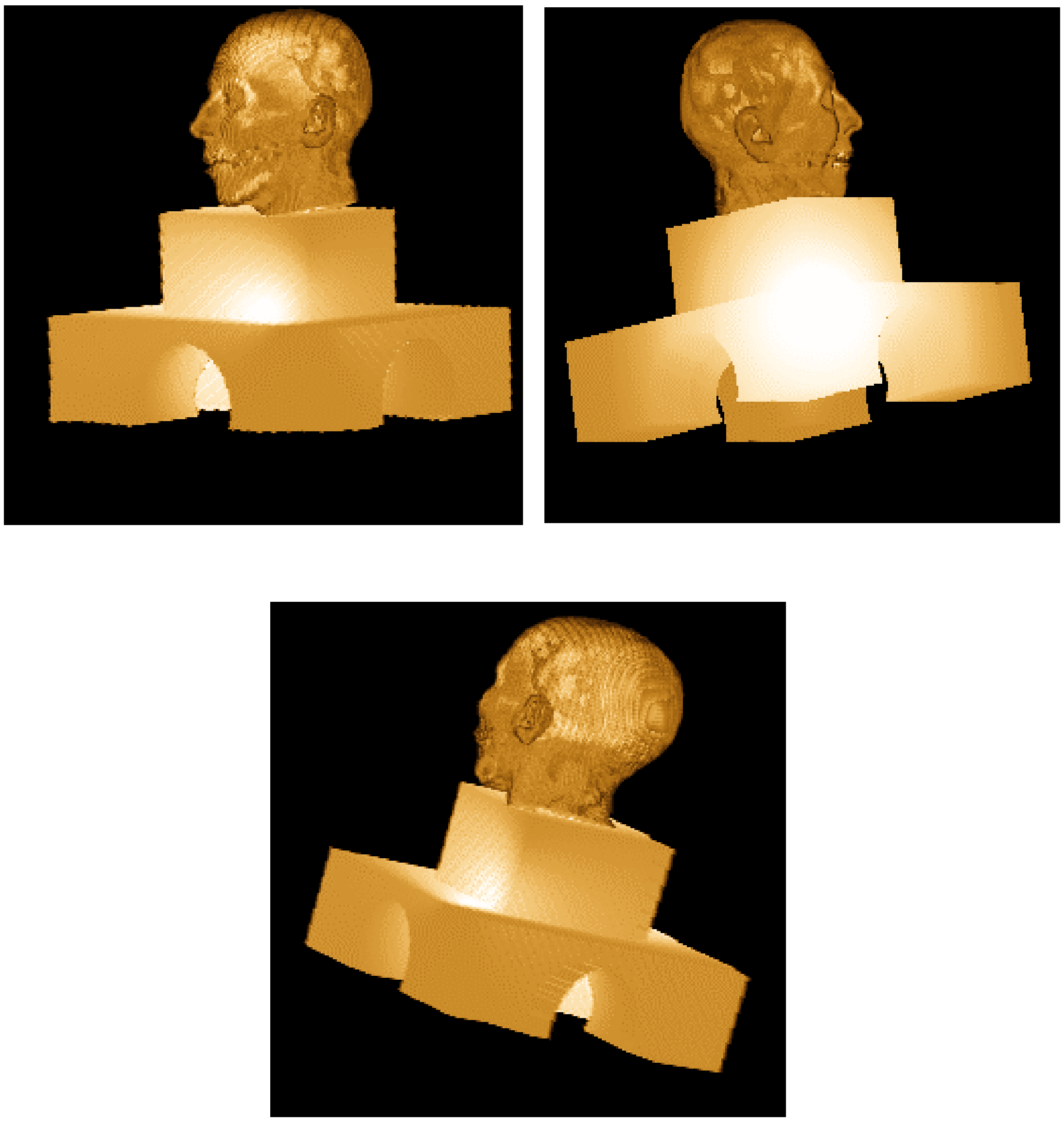


Figure 5.7 Blist evaluation result of the volumetric scene tree seen in Figure 5.6

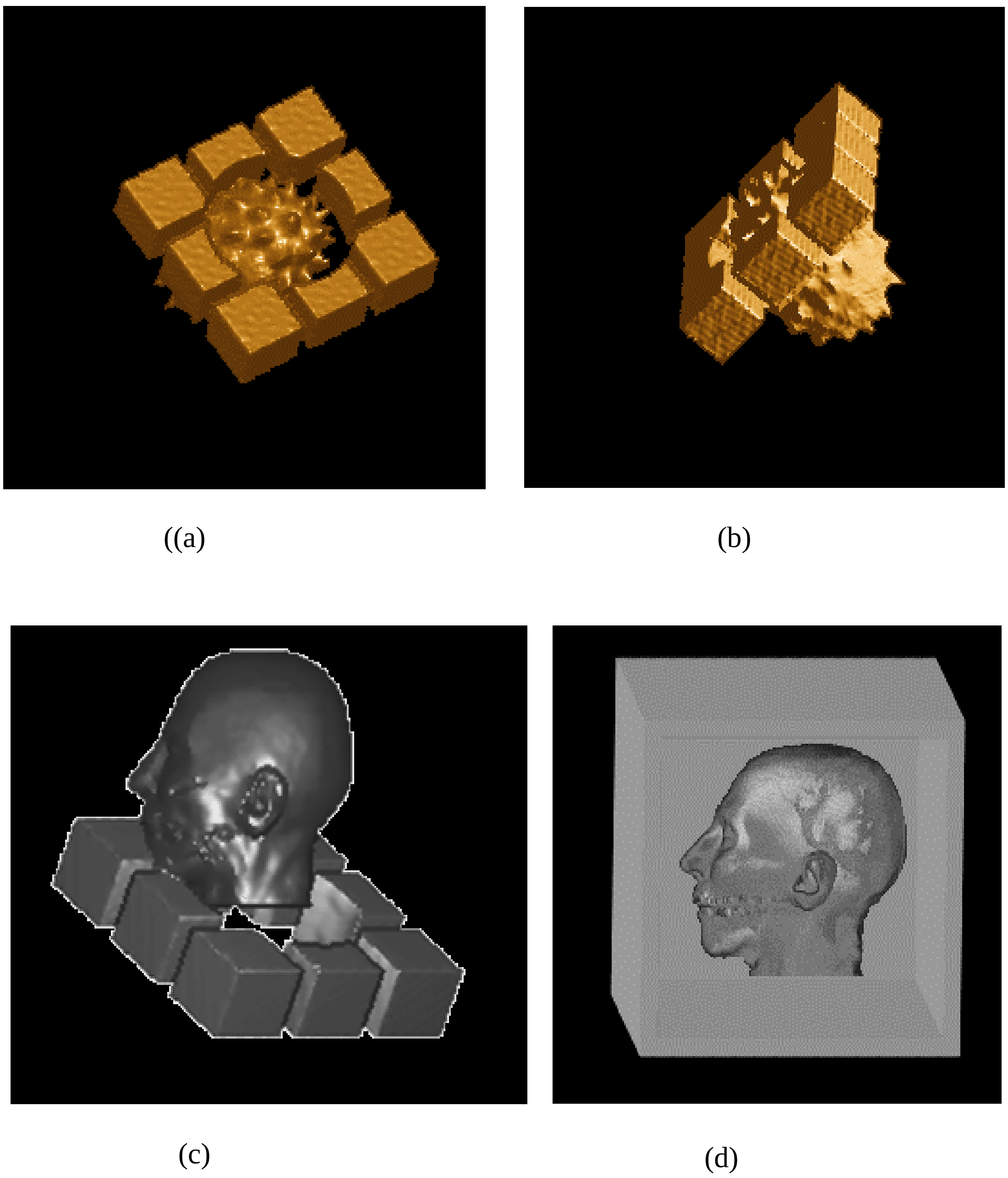


Figure 5.8 Different images whose volume data are obtained with the Blist evaluation method.

Other ability of our voxelization algorithm is that concave objects with hidden cavities can be voxelized accurately. The algorithm given in [79] was not suitable for accurately voxelizing objects with hidden cavities which means if some area of surface was not visible from any of the six faces then this area would not be properly voxelized (see Chapter 1). In Figure 5.9 (b) there is an error in voxelization, Figure

5.9 (a) shows a hollow cube, which is erroneously voxelized as a cube with a hole (Figure 5.9 (b). The Figure 5.9 (c) shows the result of our voxelization algorithm applied to the same kind of hollow cube. As it is seen, hidden cavities are kept after voxelization.

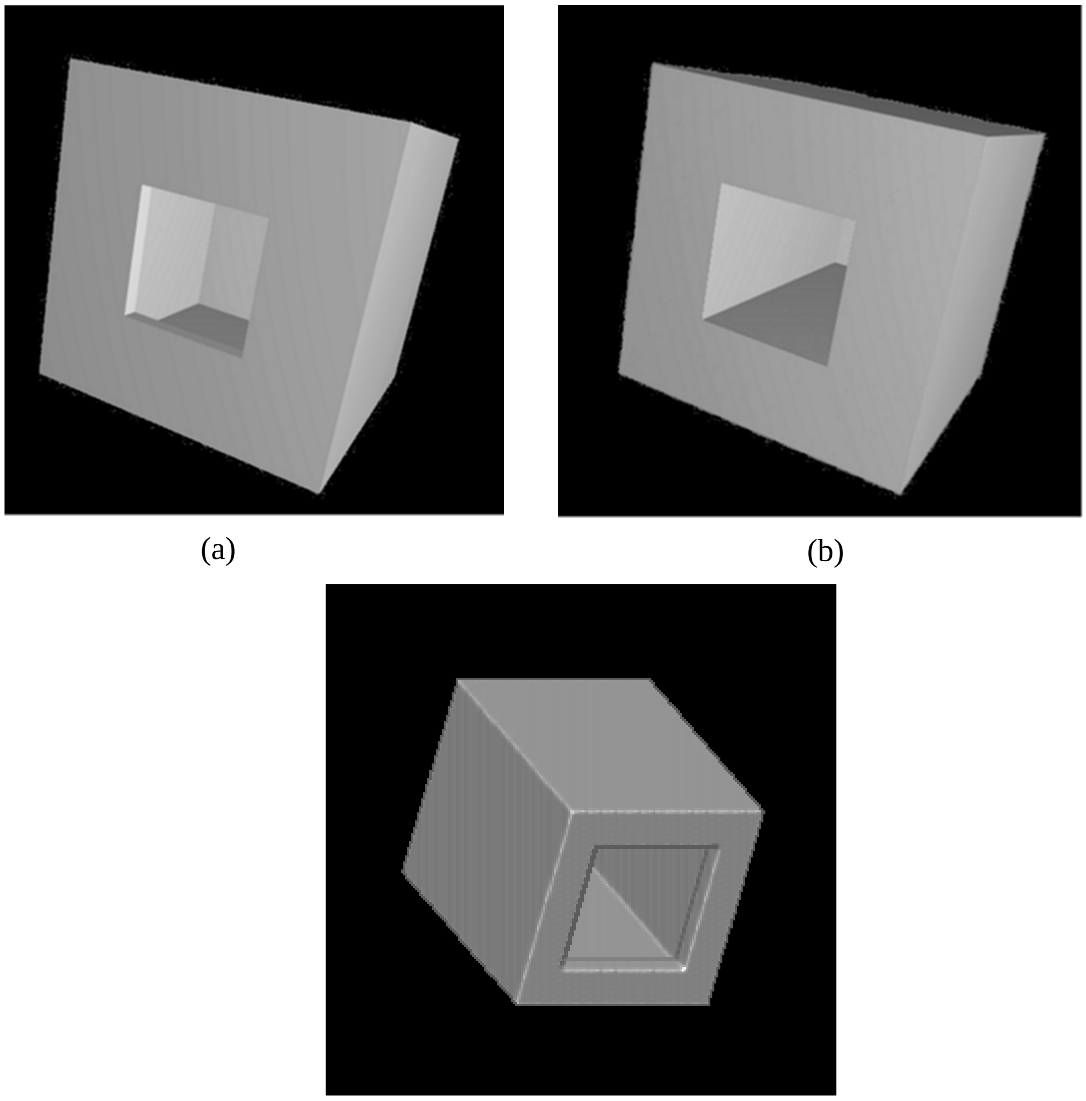


Figure 5.9 (a) Image shows a hollow cube, which is erroneously voxelized as a cube with a hole (b). (c) shows our algorithm result which is correctly voxelized as the hollow cube.

Figure 5.10 shows other examples obtained with voxelization algorithm.

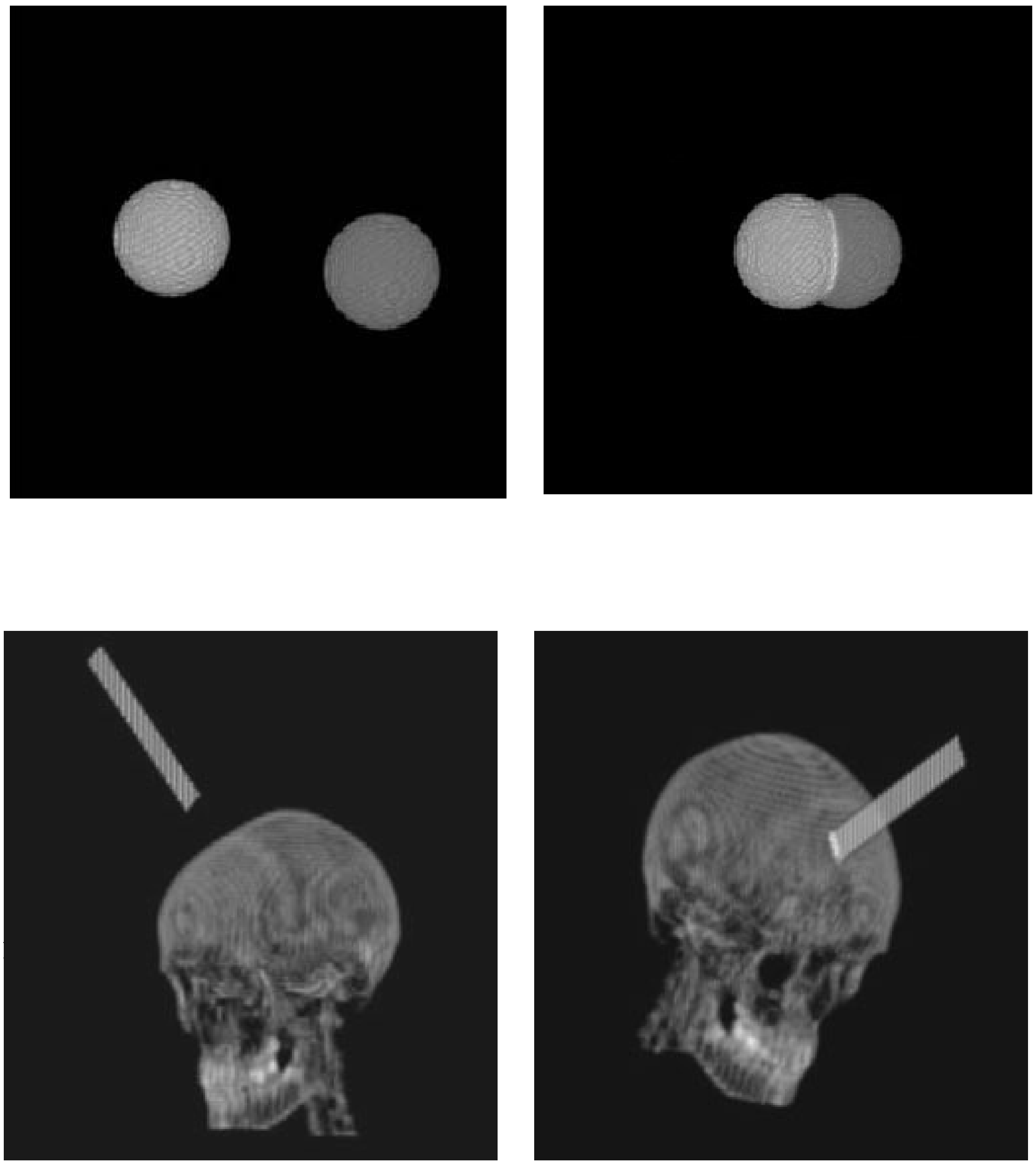


Figure 5.10 Other examples of voxelization algorithm.

5.5 Performance Analysis

One approach for the comparison of our algorithm with the other algorithms in the literature is to compare the voxelization times. But comparing the execution times on different machines with different system configurations is not very meaningful. Nevertheless, theoretically we can compare our algorithm with the other methods.

In the previous algorithms given in [26], [77], and [93] a slice is generated for each object in the scene first, and then the blending functions are applied on the slice in the order defined by the scene expression. From the point of view of time spent, since the 2D slice stack operations occupy a large proportion in practice, the running times of previous algorithms are longer than the Blist algorithm.

For a binary tree, if there are T operands, there will be T-1 operators. Then for the slice sweeping algorithm [26], the number of required push operations is 2T-1 and the number of required pop operations is 2(T-1), and the number of required comparisons is 2T-1. For the Blist approach, on the other hand, only 3T comparisons at most required, there is no need to use push and pop operations. For the volume scene in Figure 5.6, this algorithm requires a comparison for each leaf to understand whether the leaf is an operator or operand (9 comparisons). And to store the intermediate result 17 pop, push operations are needed. But our algorithm requires only 10 comparisons between the contents of the label and the name of the primitive to update a label; there is no need for push and pop instructions since Blist can be evaluated efficiently only updating a label.

This comparison is only for a small tree. If the tree becomes larger, the number of operands will increase. This also increases the 2D slice stack operations (pop, push), and comparison operations of the slice sweeping algorithm. So the difference between the processing times of the two algorithms will also become larger.

From the point of view of occupied memory, our algorithm reduces the storage requirement for each voxel to at most log(H+1), where H is height of the tree, bits, since there are no slice stack operations. But in [26], since the algorithm uses slice stack operations, M slice buffers are needed, and in [93], MN slice buffers are needed, where M and N are the number of nodes in the VST and the number of slices respectively.

In this study, production of each slice of an object is performed using the 2D pixel processor. The display system is simulated WorkView and ORCAD software packages.

The performance of slice generation, in terms of number of slices produced per second, can be defined by the following formula:

$$P_{sp} = \frac{P_R}{N_S}. \tag{5.1}$$

Where,

P_{sp} the performance in terms of slice/sec ,

P_R performance of a 2D processor in terms of surface/sec,

N_s the number of surfaces that constitute the primitive,

The performance of a pixel-based system, in terms of the number of surfaces rendered per second, can be defined by the following formula:

$$P = \frac{1}{N_D \cdot T_C} \tag{5.2}$$

Where,

P,	performance in terms of surfaces/sec,
N_D,	Number of bits to store the depth (so the number of clock cycles required to render a single surface),
T_C,	clock period in terms of seconds.

Using this formula, the number of surfaces that can be rendered in real-time can be calculated:

$$P_R = \frac{t}{N_D \cdot T_C} \tag{5.3}$$

Where,

P_R,	real-time performance,
t,	frame time for real-time display in sec. (1/30 to 1/70).

This is the real-time performance of an S_X by S_Y pixel area of the display. Here, it is assumed that there is a basic pixel processor available for every pixel on the display area. As can be noticed, the real-time performance is practically dependent on the clock speed. And it is independent of the display size as long as there is a single processor array dealing with the entire area of the display.

With a 32-bit depth length and a 10 MHz clock speed, the performance of such a system would be:

$$P = \frac{1}{32.100 \cdot 10^{-9}} = 312500 \text{ surfaces/second,}$$

or, in real-time (t=1/50 sec.)

$$P_R = \frac{1/50}{32 \cdot 100 \cdot 10^{-9}} = 6250 \text{ surfaces/frame time.}$$

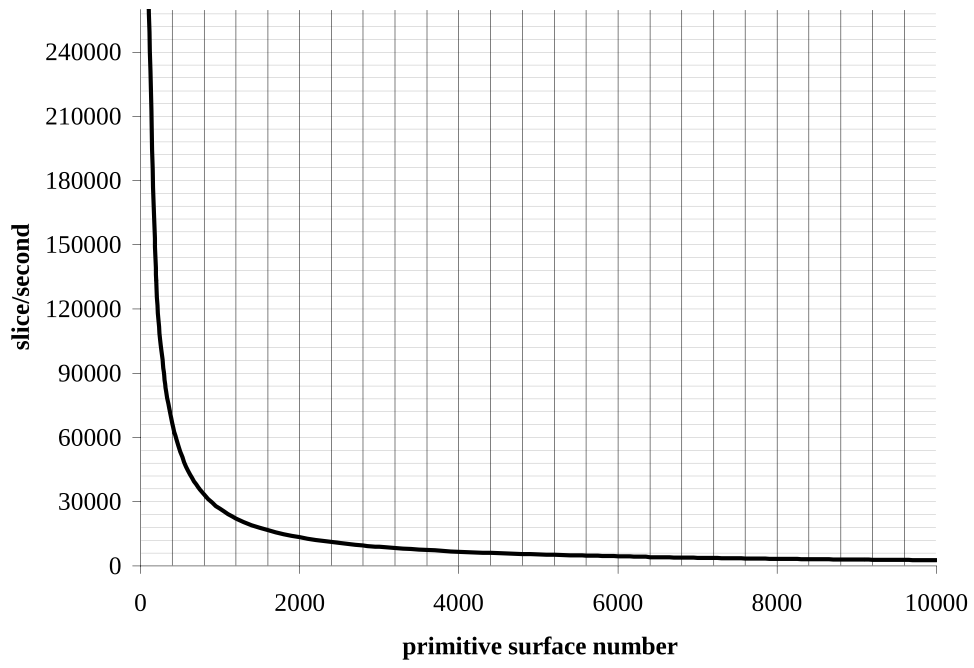


Figure 5.11 The performance of slice generating

With a 32-bit depth length, and a 10 MHz clock speed, for a 128x128 pixel resolution screen, the maximum performance of the 2D processor would be 20,000,000 surface/sec, and the performance of slice generation, $P_{sp,}$ with respect to different primitive surface numbers are given in Figure 5.11. As can be noticed, the real-time performance is dependent on the number of the surfaces, which construct the scene primitive.

CHAPTER 6

CONCLUSIONS AND FUTURE WORKS

In this study, a z-buffer based voxelization algorithm using Boolean list evaluation of volumetric CSG graph is presented.

The voxelization algorithm used in this study is implemented by a CSG surface graphics rendering system that displays surfaces by a 2D scan conversion process. When only a slice of an object is displayed, the result is essentially a slice of the volume from a 3D scan conversion. And using the depth information of front and back surfaces belonging to scene primitives our 2D CSG processor creates volume slices successfully.

In Blist formulation of the volume scene tree each primitive represents a volumetric dataset or a geometric model. The Blist method transforms the CSG tree into a decision graph. The basic idea of our algorithm is to generate a slice for each primitive in the list first, then to evaluate Blist by updating a label, when its value matches the primitive's name. At the end if the label on the voxel matches the IN's name the voxel is inside the volume scene otherwise, it is outside the volume. So, the Boolean expression of the volume scene tree is evaluated directly.

The algorithms introduced in this thesis differ from the traditional ones as follows:

- In Blist formulation, a Boolean expression is represented as a list of primitives instead of a tree, and this may be evaluated in a pipeline fashion. The fundamental breakthrough provided here lies in the fact that the result of the previous classifications does not require the list of values of cell-primitive classification results, nor a stack of intermediate results of evaluating sub-expressions. Instead, Blist passes from one primitive to the next a simple label, which may be stored using at most log(H+1) bits, where H is the height of the CSG tree [14].

- The voxelization algorithm is based on the creation of volume slices using the depth information of front and back surfaces belonging to scene primitives, and can be used for the voxelization of surfaces and solids .
- Besides CSG objects, the system can voxelize objects made of polygons and lines.
- Surface rendering effects, such as color and shading, can also be stored in the volume representation without extra computation.
- Other ability of our voxelization algorithm is that concave objects with hidden cavities can be voxelized accurately.

Future Works

As a future work, the following investigations and modifications can be considered:

- This algorithm can be realised in hardware.
- An error measure technique to quantify the quality of voxel representation of 3D solid objects can be developed. Also to increase the quality of the voxel data some filtering techniques to the resulting volume data can be applied.

REFERENCES

[1] Kaufman, A., Cohen, D., and Yagel, R. (1993). Volume Graphics.*IEEE Computer Graphics and Applications*.13, pp.51-64.

[2] Kaufman, A. (2000). State-of-art in Volume Graphics. *Volume Graphics, Springer Verlag*, pp.3-28.

[3] Kaufman, A. (1991). *Volume Visualizatio*n. *IEEE Computer Society Press Tutorial*, Los Alamitos, CA.

[4] Rossignac, J. (1998). Blist: A Boolean list formulation of CSG trees. *Technical Report GIT-GVU*-94-04.

[5] Çelebi, Ö., C., *A Template-Based Discrete Ray Tracing System for Volume Rendering*. MSc. Thesis, Gaziantep University, 2003.

[6] Speray, D., and Kennon, S. (November 1990). Volume Probes: Interactive Data Exploration on Arbitrary Grids. *Computer Graphics*, 24, 5, pp. 5-12.

[7] Neider, J., Davis, T. (1993). *Open GL Programming Guide*, Addison-Wesley Press.

[8] Cohen-Or, D. and Kaufman, A. (November 1995). Fundamentals of Surface Voxelization, *CVGIP: Graphics Models and Image Processing,* 56, 6, pp. 453-461.

[9] Coquillart, S. (August 1990). Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling. *Computer Graphics,* 24, 4, pp. 187-196.

[10] Avila, R., He, T., Hong, L., Kaufman, A., Pfister, H., Silva, C., Sobierajski, L. and Wang, S. (October 1994). VolVis: A Diversified Volume Visualization System. *Visualization '94 Proceeding*s, Washington, DC, pp. 31-38.

[11] Bakalash, R., Kaufman, A., Pacheco, R. and Pfister, H. (September 1992). An Extended Volume Visualization System for Arbitrary Parallel Projection, *Proceedings of the 1992 Eurographics Workshop on Graphics Hardwar*e, Cambridge, UK,.

[12] Cohen, M. F., Chen, S. E., Wallace, J. R. and Greenberg, D. P. (1988). A Progressive Refinement Approach to Fast Radiosity Image Generation.

Computer Graphics (Proc SIGGRAPH), pp. 75-84.

[13] Barillot, C., Gibaud, B., Luo, L. M. and Scarabin, I. M. (1985). 3D Representation of Anatomic Structures From CT Examinations. *Proceedings SPIE,* 602, pp. 307-314.

[14] Chen, L. S., Herman, G. T., Reynolds, R. A. and Udupa, J. K. (December 1985). Surface Shading in the Cuberille Environment. *IEEE Computer Graphics & Applications*, 5, 12, pp. 33-43.

[15] Chiueh, T., He, T., Kaufman, A. and Pfister, H.. (January 1994). Compression Domain Volume Rendering. *Technical Report 94.01.04, Computer Science,* SUNY at Stony Brook,

[16] Danielsson, P. E., (1970). Incremental Curve Generation. *IEEE Transactions on Computer*s. C-19, pp. 783-793.

[17] Nadeau, D. (Oct. 2000). Volume Scene Graphs. In *Proc. IEEE/ACM Symposium on Volume Visualization*, pp. 49-56.

[18] Requicha, A. A. G. (December 1980). Representation for Rigid Solids: Theory, Methods and Systems. *Computing Surveys*, 12(4), pp. 437-464.

[19] Hoffmann, Christoph M. (1989). *Geometric and Solid Modeling*. Morgan Kaufmann Publishers.

[20] Goldfeather, J., Molnar, J., Turk, S., and Fuchs, H. (1989). Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications,* 9, pp.20-28.

[21] Wiegand, T. F. Interactive Rendering of CSG Models. (1996). *Computer Graphics Forum, 15(4),* pp.249-261.

[22] Rappoport, A. and Spitz S. (August 1997). Interactive Boolean Operations for Conceptual Design of 3-D Solids. *Computer Graphics, SIGGRAPH 97,* pp. 269-278 .

[23] Wang, S. and Kaufman, A. (September 1994). Volume-sampled 3D Modeling. *IEEE Computer Graphics and Applications,* 14, pp.26-32.

[24] Fang, S. and Srinivasan, R. (1998). Volumetric CSG -- A Model-Based

Volume Visualization Approach. In Proc. *Sixth International Conference in Central Europe on Computer Graphics and Visualization*, pp. 88-95.

[25] Chen, M., Tucker, J.V., and Leu, A. (March 2000). Constructive Representations of Volumetric Environments. *Volume Graphics,* Springer-Verlag. pp. 97-117.

[26] Fang, S. and Chen, H. (March 2000). Hardware Accelerated Voxelization. In *Volume Graphics*, Springer-Verlag. Chapter 20, pp. 301-315.

[27] Okino, N., et al. (1978). *TIPS-l. Institute of Precision Engineering,* Hokkaido University, Sapporo, Japan.

[28] Hillyard, R. C. (Mar.1982). The BUILD group of solid modellers. *IEEE Comput. Graph. Appl*. 2, 2, pp. 43-52.

[29] Wolfe, R. N., Wesley, M. A., Kyle, J. C., Gracer, F., and Fitzgerald. (May 1987). Solid modelling for production design. *IBM J. Res. Den* 31, 3, pp. 277-295.

[30] Rossignac, J. R., And Reqwicha, A. A. G. (Aug. 1986). Offsetting operations in solid modelling. *Comput. Aided Geom*. Des. 3,2, pp. 129-148.

[31] Rossignac, J. R., And Reqwiicha, A. A. G. (Sept. 1986). Depth buffering display techniques for constructive solid geometry. *IEEE Cornput. Graph. Appl.* 6,9, pp. 29-39.

[32] Shirma, Y., Okino, N., Aped Kakazu, Y. (1982). Research on 3-D geometric modelling by sweep primitives. *In Proceedings of CAD '82,* pp. 671-680.

[33] Van Wijk, J. J. (1984). Ray tracing objects defined by sweeping a sphere. *In Proceedings of Eurographics '84.* Elseviers Science Publishers, Amsterdam, pp. 73-82.

[34] Rossignac, J. R. (Sept. 1986). Constraints in constructive solid geometry. *In Proceedings of the 1986 Workshop on Interactive 30 Graphics* (Chapel Hill, N.C., October 23-24). ACM Press, New York, 1986, pp. 93-110.

[35] Anderson, D. C. (May 1986). Closing the gap: A workstation-mainframe connection. *Comput. Mech. Eng*. 4,6 pp. 16-24.

[36] Peterson, D. P. (Jan.-Feb. 1986). Boundary and constructive solid geometry mapping: A focus on 2-D issues. *Comput. Aided Des*. 18, pp. 1 3-14.

[37] Vossler, D. L. (Aug. 1985). Sweep-to-CSG conversion using pattern recognition techniques. *IEEE Comput. Graph. Appl*. 5,8, pp. 61-68.

[38] Cameron, S. A. (1985). A study of the clash detection problem in robotics. *In Proceedings of the International Conference on Robotics and Automation (St. Louis, Mar.). IEEE*, pp. 488-493.

[39] Rossignac, J. R., And O'connor, M. A. *Selective geometric complexes: Representations and algorithms for processing and combining mixed dimensional geometric objects.* IBM Research Division, T. J. Watson Research Center, Yorktown Heights, N.Y.

[40] Tilove, R. B. (Oct. 1981). *Exploiting spatial and structural locality in geometric modelling*. Tech. Memo. 38, Production Automation Project, University of Rochester, Rochester, N.

[41] Tilove, R. B. (July 1984). A null-object detection algorithm for constructive solid geometry. *Commun. ACM* 27,7, pp. 684-694.

[42] Cameron, S. A. (1989). Efficient intersection tests for objects defined constructively. *Int. J. Rob. Res*. Volume 8,1, pp.3-25.

[43] Woodwark, J. R. (May 1988). Eliminating redundant primitives from set-theoretic solid models by a consideration of constituents. *IEEE Comput. Graph. Appl*. 8,3, pp. 38-47.

[44] Tilove, R. B. (Oct. 1980). Set membership classification: A unified approach to geometric intersection problems. *IEEE Trans. Comput.* C-29,10, pp. 874-883.

[45] Tilove, R. B., Requicha, Pl. A. G., And Hopkins, M. R. (May 1984*). Efficient editing of solid models by exploiting structural and spatial locality.* Tech. Memo. 46, Production Automation Project, Univ.of Rochester, Rochester, N.Y.

[46] Requicha, A. A. G., And Voelcker, H. B. (Jan. 1985). Boolean operations in solid modelling: Boundary evaluation and merging algorithms. *In Proc. IEEE 73*, 1, pp. 30-44.

[47] Rossignac J., Voelcker H. (1989). Active Zones in CSG for Accelerating

Boundary Evaluation, Redundacy Elimination, Interference Detection and Shading Algorithms. *ACM Transactions on Graphics*, Vol.8, p.51-87.

[48] Rudeanu, S. (1974). *Boolean Functions and Equutions*. North-Holland, Amsterdam.

[49] Kaufman A. (1990). *Introduction to Volume Graphics*. State University of NewYork at Stony Brook.

[50] Kaufman, A. (1991). Introduction to Volume Synthesis, *Scientific Visualization of Physical Phenomena (Proceedings of CG International '91)*, Springer-Verlag, pp. 27-35.

[51] Kaufman, A. (July 1987). Efficient Algorithms for 3D Scan-conversion of Parametric Curves, Surfaces, and Volumes''. *Computer Graphics*, 21, 4, pp. 171-179.

[52] Chen, L.-S., Herman, G.T., Reynolds, R.A., and Udupa, J.K.(December 1985). Surface shading in the cuberille environment. *IEEE Computer Graphics and Applications*, 5(12) pp.33–43.

[53] Yagel, R., Cohen, D., and Kaufman, A. (June 1992). Normal estimation in 3D discrete space. *The Visual Computer*, 8(5–6), pp.278–291.

[54] Sramek, M. (1998). Visualization of Volumetric Data by Ray Tracing. *Austrian Computer Society*, Austria,. ISBN: 3-85403, pp. 112-114.

[55] Yagel, R. Cohen, D. and Kaufman, A. (September 1992). Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5), pp. 19–28.

[56] Hohne, K.H.and Bernstein, R. (March 1986). Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, MI-5(1), pp. 45–47.

[57] Magnusson, M., Lenz, R. and Danielsson, P.E. (1990). Evaluation of methods for shaded surface display of CTvolumes. *Computerized Medical Imaging and Graphics*, 15(4), pp. 247–256.

[58] Tiede, U., Hohne, K.H., Bomans, M., Pommert, A., Riemer, M. and Wiebecke,G. (1990). Investigation of medical 3D-rendering algorithms. *IEEE*

Computer Graphics and Applications, 10(3), pp. 41–53.

[59] Wang, S.W. and Kaufman, A. (October 1993). Volume sampled voxelization of geometric primitives. *In Visualization '93, San Jose, CA*, pp. 78–84.

[60] Wang, S.W and Kaufman, A. (September 1994). Volume-sampled 3DModelling. *IEEE Computer Graphics and Applications*, 14(5), pp. 26–32.

[61] Oomes, S., Snoeren, P., and Dijkstra, T. (1997). Transforming polygons into voxels. *In Scale-Space Theory in Computer Vision, Lecture Notes in Computer Science*, Springer-Verlag, Vol. 1252.

[62] Sramek, M. (1994). Gray level voxelization: A tool for simultaneous rendering of scanned and analytical data. In *Proceedings of the 10th Spring School on Computer Graphics and its Applications*, pp. 159–168.

[63] Sramek, M. and Kaufman, A. (1998). Object voxelization by filtering. In *IEEE Symposium on Volume Visualization*, pp.111–118.

[64] Sramek, M., and Kaufman, A. (1999). Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, (3)5, pp.251-266.

[65] Payne, B.A., and Toga, A.W. (January 1992). Distance field manipulation of surface models. *IEEE Computer Graphics and Applications*, 12(1), pp. 65–71.

[66] Jones, M.W. (December 1996). The production of volume data from triangularmeshes using voxelisation. *ComputerGraphics Forum*, 15(5), pp. 311–318.

[67] Gibson, S.F.F. (1998). Using distance maps for accurate surface reconstruction in sampled volumes. In *IEEE Symposium on Volume Visualization*, pp. 23–30.

[68] Breen, D.E., Mauch, S., and Whitaker, R.T. (1998). 3D scan conversion of CSG models into distance volume. In *IEEE Symposium on Volume Visualization*, pp. 7–14.

[69] Kaufman, A. and Shimony, E. (Oct. 1986). 3D Scan-conversion Algorithms for Voxelbased Graphics. In *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pp. 45-75.

[70] Kaufman, A. (1988). Efficient Algorithms for Scan-converting 3D Polygons. *Computers and Graphics,* 12(2), pp. 213-219.

[71] Cohen, D. and Kaufman, A. (1991). Scan-conversion Algorithms for Linear and Quadratic Objects. In A. Kaufman, editor, *Volume Visualization*, pp. 280-301.

[72] Cohen, D. and Kaufman, A. (1997). 3D Line Voxelization and Connectivity Control. *IEEE Computer Graphics and Applications*. 17(6), pp. 80-87.

[73] Huang, J., Yagel, R., Filippov, V.,and Kurzion, Y. (1998). An Accurate Method for Voxelizing Polygon Meshes. In *Porc. IEEE/ACM Symposium on Volume Visualization*, pp. 119-126.

[74] Lee, Y.T. and Requicha, A.A.G. (1992). Algorithms for Computing the Volume and Other Integral Properties of Solids. *Communications of the ACM,* 25(9), pp. 635-650.

[75] Shareef, N., and Yagel, R. (May 1995). Rapid Previewing Via Volume-based Solid Modeling. *Solid Modeling'95* , pp. 281-292.

[76] Breen, D.E., Mauch, S., and Whitaker, R.T. (1998). 3D Scan Conversion of CSG Models into Distance Volumes. In *Porc. IEEE/ACM Symposium on Volume Visualization*, pp. 7-14.

[77] Fang, S., and Liao, D. (Oct. 2000). Fast CSG Voxelization by Frame Buffer Pixel Mapping. In *Porc. IEEE/ACM Symposium on Volume Visualization*, pp. 43-48.

[78] Prakash, C.E., and Manohar, S. (1995). Volume rendering of unstructured grids-a voxelization approach. *Computer Graphics*, 19(5), pp. 711-726.

[79] Karabassi, E. A, Papaioannou, G., Theoharis, T. (1999). A fast depth-buffer based voxelization algorithm, *Journal of Graphics Tools,* ACM, Vol.4, No.4, pp. 5-10.

[80] Epstein, D., Jansen, F., Rossignac, J. (November 1989). Z-Buffer Rendering From CSG: The Trickle Algorithm. *IBM Research Report Rc 15182.*

[81] Çevik, U., And Thomas, A.L. (1994). Design & Implementation Of An Intelligent Frame Buffer In A Traditional Display Pipeline System. *Melecon*

94, 7th Mediterranean Electrotechnical Conference, Antalya, Turkey, pp. 347-350.

[82] Çevik, U. (June 1996). *Design Of An FPGA Based Parallel Architecture Processor Displaying CSG Volumes And Surfaces*. Phd. Thesis. University Of Sussex, Brighton.

[83] Su, C.J., Lin, F.H., (July 1989). A New Collision Detection Method For CSG-Represented Objects in Virtual Manufacturing. *Computers in Industry*, 40 (1), pp. 79-88.

[84] Stewart, N., Leach, G., John, S. (2000). A Z-Buffer Rendering Algorithm For Convex Objects. *Proceedings Of The 8-Th International Conference İn Central Europe On Computer Graphics, Visualisation AND Interactive Digital Media' 2000-WSCG*, Volume II, pp. 369-372.

[85] Koç, S. (June 1999). *Development Of An Algorithm For The Elimination Of Surface Sorting İn The Stage Of Hidden Surface Removal İn Displaying Constructive Solid Geometry (CSG) Volumes And Surfaces*, M.Sc. Thesis.

[86] Ming, C. L., and Canny, J.F. (September 1992). Efficient Collision Detection for Animation, *Third Eurographics Workshop on Animation and Simulation*, Cambridge, United Kingdom.

[87] Michael, K., and Poston, T., and Bricken, W. (August 1994). Efficient Virtual Collision Detection for Multiple Users in Large Virtual Spaces, *Virtual Reality Software and Technology Proceedings of VRST'94*, pp. 271-286.

[88] Bez, H.E., and Bricis, A.M., and Ascough, J. (1996). A collision Detection method with applications in CAD systems for the Apparel Industry, *Computer-aided Design*, 28(1), pp.27-32.

[89] He, T., and Kaufman, A. (October 2000). Collision Detection for Volumetric Object Visualization, *Proc. IEEE Visualization'97.*

[90] Boyles, M., and Fang, S. (2000). Slicing based Volumetric Collision Detection, *ACM Journal of Graphics Tools*, pp.23-32.

[91] Gagvani, N., and Silver, D. (Oct. 2000). Shape based Volumetric Collision detection. *In proc IEEE/ACM Symposium on Volume Visualization*, pp.57-61.

[92] Lin, M.C., and Manocha, D. (1995). Fast Interference Detection between Geometric Models. *The Visual Computer,* 11(10), pp. 542-551.

[93] Duoduo, L., Fang, S. (June 2002). Fast Volumetric CSG Modelling Using Standart Graphics System. *7th ACM symposium on Solid modeling and Applications*, Saarbrucken, Germany, pp.204-211.

[94] Preparata, F.P. and Shamos, M.I. (1985). *Computational Geometry*. Springer-Verlag, New York.

[95] Koç, S., Çevik U. (Jun 2004). A New Approach For The Voxelization Of Volumetric CSG Graphs, *Computers & Electrical Engineering* 30 (4), pp. 245-255.

[96] Requicha, A. A. G., And Tilove, R. B. (June 1978). *Mathematical foundation of constructive solid geometry: General topology of closed regular sets*. Tech. Memo. 27a, Production Automation Project, Univ. of Rochester, Rochester, N.Y.

[97] Fuchs, H., Pulton, J. (1982). Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System, *Proc. 82 Conf. On Advanced VLSI, M.I.T.*

VITA

Sema KOÇ received her BSc and Msc degrees in Electrical and Electronic Engineering from the University of Gaziantep, Turkey, in 1996 and 1999, respectively. She has worked on her doctoral research in volume graphics. She has the following publications:

Journals

1. Koç, S., Çevik, U. A New Approach For The Voxelization Of Volumetric CSG Graphs, *Computers & Electrical Engineering* 30 (4), Jun 2004, pp. 245-255.
2. Erçelebi, E., Koç, S. Lifting-Based Wavelet Domain Adaptive Wiener Filter For Image Enhancement. IEE Proc. Vision, Image & Signal Processing (Accepted for publication).

International Conferences

1. Koç, S., and Çevik, U. Development of an Algorithm for the Elimination of Surface Sorting in the Stage of Hidden Surface Removal in Displaying Constructive Solid Geometry (CSG) Volumes and Surfaces. CAR&FOF' 98 14th International Conference on CAD/CAM, Robotics and Factories of the Future, Coimbatore, India. 1998., Conf. Proc. v. 1, pp. 37-43.
2. Koç, S., and Çevik, U., Volumetric CSG Graph and Its Voxelization. Eleco03 International Conferences on Electrical and Electronics Engineering, Bursa 2003.
3. Koç, S., and Çevik, U., A CSG Clipping Algortihm. JCI Proceedings of Intl. XII. Turkish Symposium on Artificial Intelligence and Neural Networks, Vol.1, No.1, July 2003, pp.359-361.
4. Koç, S., and Çevik, U. A Z-Buffer CSG Display Algorıthm With Illumination. International Conference On Informatics on September, 2004, in Çesme, İzmir.

National Conferences

1. Koç, S., and Çevik, U. Üç Boyutlu Cisimlerin Görüntü Algoritmaları Için Bir Aydınlatma Modeli. Elektrik-Elektronik Bilgisayar Mühendisliği 8. Ulusal Kongresi 1999, Gaziantep, pp.13-16.
2. Koç, S., and Çevik, U., Konveks ve Konkav Cisimlerin Görüntülenmesi İçin Kırpma Algoritması. Eleco 2002 Elektrik-Elektronik Bilgisayar Mühendisliği Sempozyumu Ve Fuarı.2002 Bursa, pp. 295-297.

3. Koç, S., and Çevik, U., Hacimsel Görüntü Grafiğinin Blist Gösterimi. Elektrik, Elektronik ve Bilgisayar Mühendisliği 10. Ulusal Kongresi, Eylül 2003,İTÜ, İstanbul, pp.496-498.

Diğer yayınlar

Sema Koç. Development of an Algorithm for the Elimination of Surface Sorting in the Stage of Hidden Surface Removal in Displaying Constructive Solid Geometry Volumes and Surfaces. A Master's Thesis, June 1999.