Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

A Survey of Geometric Data Structures for Ray Tracing

Allen Y. Chang

Department of Computer and Information Science

Technical Report
TR-CIS-2001-06
10/16/2001

A Survey of Geometric Data Structures for Ray Tracing

Allen Y. Chang [*]

Department of Computer and Information Science
Polytechnic University

October 13, 2001

Submitted in partial fulfillment of the requirements for the Ph.D. Degree in Computer Science at Polytechnic University, Brooklyn, New York, 11201.

Keywords: Ray tracing, data structures.

Abstract

Ray tracing is a computer graphics technique for generating photo-realistic images. To determine the color at each pixel of the image, one traces the path traversed by each ray of light arriving at the pixel back through several reflections and/or refractions. The most time-consuming phase of a ray tracer is ray traversal, which determines for each of a large number of rays, the first object met by that ray. Many data structures have been proposed to accelerate this process. This survey describes and compares the construction and traversal algorithms for a variety of commonly used data structures from practitioner's point of view.

[*] `achang@cis.poly.edu`. Work on this survey has been supported by National Science Foundation under Grant ITR-0081964.

Acknowledgments

This survey is not written by the author alone; without the following people, it could not be finished. The author would like to thank my advisor, Professor Boris Aronov, not only for the inspiration of this survey, but also for his generous patience, support, and the countless discussions that have guided the author throughout the entire course of writing. The author is grateful to Professor Yi-Jen Chiang and Professor Micha Sharir. Professor Chiang's class initiated my interest in Computational Geometry, while Professor Sharir's class at New York University revealed to me the beauty of this field. Their guidance helped me to understand the core operation of ray tracing. The author is also grateful to Professor Hervé Brönnimann for helpful discussions, careful review, and many meaningful suggestions of this survey. Professor Brönnimann's class, "Programming Workshop – Algorithms libraries", forever changed my programming style, helped me look at the framework of a ray tracer in a brand new way. Professor Pankaj K. Agarwal showed the author very helpful pointers and bibliography for ray tracing. Without Professor Agarwal, this survey would only be a few pages long. The author would also like to thank Professor Sariel Har-Peled, Professor Seth Teller, and Dr. Steve Fortune for sharing their ideas and valuable suggestions. Professor Teller and Dr. Steve Fortune pointed out many resources so that our BSP-tree section does not have to be left blank.

Contents

PART I

Introduction

1 The Root of Ray Tracing

Ray tracing has interested geometers for at least four hundred years. In 1637, René Descartes published his *Discours de la méthode* [33], which contained his experimental observations on a spherical glass flask full of water. Descartes used ray tracing as a theoretical framework to explain the phenomenon of the rainbow. He used the geometrical reflection and refraction laws to trace rays through a water drop. No one could explain the colors of the rainbow at that time, until thirty years later Newton discovered that white light contained light at all wavelengths. The color of light became an interesting topic for many researchers. Watt [116, 115] describes more details about Descartes' work.

Modern research in ray tracing by means of a computer was initiated by Appel [7] in 1968. Appel presented experimental results in the automatic shading of line drawings. The goal was to generate pictures for objects bounded by flat surfaces on a digital plotter, and to evaluate the cost of generating such pictures and the resulting graphical quality.

Comparing to wireframe drawing methods, Appel's method was very time consuming; it required several thousand times as much calculation time. Therefore, this technique was not widely used at that time because of the lack of computing power in the 1960's.

Ray tracing became popular due to Whitted's work [120]. He presented a recursive global illumination model and implemented a visible surface algorithm in 1980. His model generated very realistic scenes in many cases. However, it was still very slow. For simple scenes, 75% of the time was spent on computing the intersections of rays and surfaces. His experimental results showed that ray-surface intersection test could take more than 95% of the computing time for complicated scenes.

Whitted's work indicated that a more efficient algorithm for ray-surface intersection test can dramatically increase the performance of a ray tracer. This initiated the search for more efficient ray tracing algorithms (in the 1980's). Since then, a lot of work has been done using various approaches. Glassner's often cited *An Introduction to Ray Tracing* (IRT) [51] summarizes the state of the art prior to the 1990's.

Glassner's IRT did not provide quantitative comparisons. Researchers often used their own scenes to demonstrate the advantage of their approach over others. Therefore, the information is insufficient to compare the algorithms objectively. Haines [57] proposed several scenes to use as a standard benchmark, including recursive tetrahedral pyramid, fractal mountain, tree, dodecahedral rings, gears, etc. The scenes were put together in a freely available package known as the *Standard Procedural Database* (SPD) [59]. It was used widely for quantitative comparisons in late 80's and early 90's [73,34,112,59,41,75]. However, there are two problems. First, ray tracers tend to take the same amount of time on pictures of similar nature. Parameters of the similarity are not well understood. The second problem is when the number of objects in a scene becomes very large, ray tracers tend to have constant time behavior. It is due to the fact that objects in a large scene are so densely packed that a ray can hit an object without going too deep into the scene. This implies that SPD may not be able to accurately evaluate the performance of a data structure for large data sets.

Many novel data structures were developed to make ray tracing more efficient. We would like to investigate commonly used data structures that support efficient ray tracing. This survey is organized as follows. Part I provides background information and history of ray tracing. In section 2, we define the problem and introduce the terminology used in this survey. Each data structure is also briefly mentioned there. Part II introduces some flat (i.e., non-hierarchical) data structures. Simple bounding volumes are described in section 3. They are the earliest data structures used for ray tracing. Other flat structures such as uniform grids are discussed in section 4. These structures are easy to build and very efficient to traverse. Mixing different data structures usually results in a more efficient data structure. We discuss how to combine different flat structures in section 5.

Part III is the main portion of this survey. It introduces many hierarchical data structures for ray tracing. First, we introduce object-oriented partitioning approaches in section 6.1. Then we discuss space-oriented partitioning approaches, classifying them according to the number of subregions created at each level. We discuss binary space partitioning (BSP) trees and k-D trees in section 7.1. Both of them split a region into two subregions at each level. Octrees are discussed in section 7.2. At each partitioning step, a region is divided into eight subregions. Other hierarchical data structures when a region is divided into more than eight regions are discussed in section 7.3. They include recursive grids, adaptive grids, and hierarchical uniform grids. As in Part II, we discuss some combinations of different data structures after we introduce each of them. Finally, conclusion of this survey is given in Part IV. None of the theoretical proof for ray shooting algorithms are discussed in this survey.

2 Preliminaries

Generating an image on a computer from a model involves two main steps. First, a program must produce the geometric description of the scene as a skeleton of the image. For example, the coordinate system and the position of the objects. Based on the skeleton, some colors are added to the scene. The first step is called *meshing*, the second step is called *rendering*. Rendering usually takes a long time depending on the desired quality of the rendered scenes. To render a scene is just a matter of solving the *rendering equation* to evaluate the color and intensity [37]. A long list of variants of rendering equations can be found in Dutré's *Global Illumination Compendium* [36].

There are two models to determine the color of a certain point in the image: *local illumination model* and *global illumination model*. The former calculates the intensity of a pixel by determining how much light is transmitted directly from the light source to the point of interest. Phong lighting model [94] is often used in this case. Global illumination model considers not only the transmitted light but also the light indirectly reflected from other object surfaces. Most of the light in the real-world does not come directly from the light source, therefore global illumination model is able to simulate the real world light more closely and generate photorealistic images. The color of each pixel can be obtained by solving Whitted's illumination equation [120].

Ray tracing is one of the popular techniques; it adheres to the global illumination model. It shoots a ray for each pixel of the screen and calculates the *transmitted, reflected*, and *refracted ray* recursively. There are different types of rays. The ray that comes from the screen or viewer's eye is called the *primary ray*. If the primary ray hits an object, the light may bounce from the surface of the object. We call these rays *secondary rays*. For example, for a shiny surface, we have to calculate the reflected ray. The refracted ray should be considered if the ray hits a transparent or semi-transparent object. To add the shadow effect, we also need to consider the *shadow ray*. The origin of a shadow ray is on the surface of an object and it is directed towards the light sources. If the ray hits any object before it reaches any light source, the point located at the ray origin is in the shadow and should be assigned a dark color. Different kinds of rays are depicted in Figure 1. The light source is shown in the upper-left corner. Primary ray P is the incoming ray originating from the viewpoint. N is the surface normal. L is the reflected ray of P corresponding to N. R is the refracted ray if the surface is not opaque. Shadow ray is illustrated by vector S.

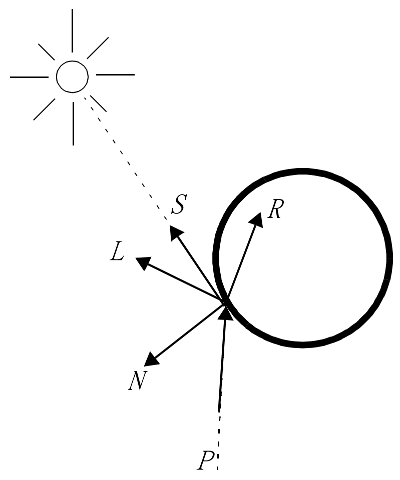


Figure 1: Illustration of different rays. P: primary ray, L: reflected ray, R: refracted ray, S: shadow ray. N is the surface normal.

In many cases we only care about which surface is visible from the viewpoint. Then, only primary rays are considered. Algorithms that only consider the primary rays are *ray casting* *algorithms*. Watt [115] points out we do not encounter highly reflective and transparent surfaces very often in the real world. By concentrating on the primary ray only, we often get some noise in the image but the rendering speed is much improved compared to considering all the rays in the scene. It is an important technique in realtime applications.

Ray tracing algorithms are *view-dependent*. A view-dependent algorithm discretizes the view plane to determine points at which to evaluate the rendering equation. Another important approach that also belongs to global illumination model is the *radiosity technique*. In contrast to ray tracing, radiosity algorithms discretize the environment to evaluate the rendering equation at any point from any viewing direction. In this survey, we only consider ray tracing methods.

The basic operation of a ray tracing algorithm is *ray shooting*. According to Pellegrini [91], a ray shooting problem can be defined as follows. Given a collection $\mathcal{P}$ of objects, we want to know, for a given point p and direction $\vec{d}$, the first object in $\mathcal{P}$ intersected by the ray defined by the pair $(p, \vec{d})$. It usually involves preprocessing a set of objects such that the first object hit by a query ray can be determined efficiently. The choice of ray shooting algorithm is important because it is the bottleneck of a ray tracer.

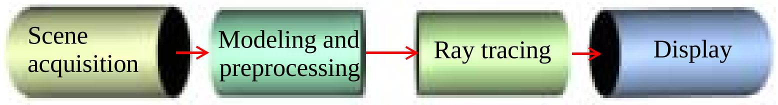


Figure 2: Ray tracing pipeline

A typical way of implementing a rendering process is the *rendering pipeline* illustrated in Figure 2. It involves several stages, one after another, to realize the image on the screen. Since ray tracing is one of the rendering techniques, it also follows this model. It consists of four main steps. The first step is to acquire data from the scene description file. A ray tracer can define its own *scene description language* (SDL) to represent the objects in the environment. Some popular SDLs include POV files from Persistence, Inc. [87], RAY files for Rayshade from Stanford University [78], and VRML file format. The NFF file format proposed by Haines [57] is also commonly used in ray tracing literature.

This survey focuses on the second and third steps in the ray tracing pipeline. During the preprocessing step, we usually construct a data structure that speeds up ray tracing. Although the preprocessing step is optional, it is often critical to the overall ray tracing performance. The third step involves using a ray traversal algorithm to search for the object hit by a given ray. We shall see many data structures and ray traversal algorithms in the subsequent sections. The last step in ray tracing pipeline is to display the image on the screen. The performance of this step is hardware dependent and is not covered in this survey.

The basic ray tracing steps can be summarized by algorithm RAYTRACE. The algorithm simply calls the RAYSHOOT function for each pixel. A *pixel* is an individual cell in two-dimensional raster image. It is a shorthand for "picture element". A three-dimensional analog of a pixel is called a *voxel*, representing an individual volume element in a scene.

We will see the term voxel many times in the following sections. Function RayShoot calls itself recursively to calculate the reflected and refracted rays. The shadow ray is handled differently. It calls function RayShootShadow to do all the work.

```
Algorithm RayTrace()
 1: Acquire the scene from scene description file;
 2: Construct a data structure for the scene;
 3: for each image pixel do
 4:     color ← RayShoot(primary);
 5: end for
 6: Display the image on the screen;
```

```
Algorithm RayShoot(ray)
Input: A ray in 3-space
Output: The color of the pixel.
 1: for each object do
 2:     Calculate intersection and store the nearest object;
 3: end for
 4: for each light source do
 5:     color ← RayShootShadow();
 6: end for
 7: if needed then
 8:     color ← RayShoot(reflected ray);
 9:     color ← RayShoot(refracted ray);
10: end if
11: Evaluate color;
12: return color;
```

Ray tracing can be treated as a process of determining the visible surface of the objects [26,19]. Unlike most other standard visible surface algorithms, ray tracing is a non-projective method. In a *projective method*, the surface elements of the objects are projected onto the image plane and a visibility calculation is performed based on a depth sort prior to projection. For all surface elements of the object, visibility calculation is based on a depth sort prior to projection of all object surface (list priority algorithm), a depth sort for every pixel (z-buffer algorithms), or a depth sort for each scan-line segment (scan-line algorithms).

For all of these visible/hidden surface algorithms, objects in the scene can be represented in different ways. Jansen [70] classifies the object representation for visible/hidden surface algorithms into two models. The first is *polygon model*. In this model, the surface of objects are approximated with a mesh of polygons. Brute-force ray tracing with the polygon model is trivial. It just searches for the candidates among the polygonal mesh to find the first hit. The problem of polygonal model object representation is there are usually a lot of polygons in a scene. Typical scenes can consist of thousands to millions of polygons. The alternative, *geometric model*, does not approximate the surface with polygons. Instead, it defines the surface with procedural representation analytically. A geometric model takes fewer primitives to describe a scene, but each ray-object intersection test itself is quite expensive. In section

3, we describe the bounding volume method, which is one way of reducing the number of expensive tests we just mentioned.

A *bounding volume* V of an object o is a solid body in space such that the surface of o is fully contained inside V. It is also known as an *extent*. The idea of enclosing an object with bounding volume is first proposed by Clark [26] to improve the performance of his hidden-surface algorithm. Whitted [120] applies this idea to ray tracing. This data structure will be discussed in Section 3.

Object extents can be clustered together to form a hierarchy. Each cluster contains two or more object extents. Each object extent can only belong to one cluster. Different clusters can also be grouped together to become a bigger cluster. Finally, the whole scene can be treated as a big bounding volume. We call this new structure a *bounding volume hierarchy*. We will define bounding volume hierarchy more formally in Section 6.1.

For the polygon model, the bottleneck is the search process. An efficient search structure is crucial. We survey several common search structures from ray tracing literature. Instead of surrounding the objects with extents, these search structures use *spatial subdivision* approaches that divide the space into several regions. Spatial subdivision techniques rely heavily upon *coherence*. Coherence is the relationship between objects in a scene. There are five types of coherence. *Object coherence* is the property that objects tend to consist of pieces which are connected, smooth and bounded. *Scene coherence* is the view-dependent version of object coherence. Object coherence carries over to 2D projections of the environment, i.e., some degree of connectivity and smoothness in the image plane as existed among the original 3D objects. Nearby rays display *ray coherence*: Two rays that have nearly the same origin and nearly the same direction are likely to trace out similar paths. *Temporal coherence* is proven to be useful for collision and visibility algorithms [4]. It assumes that if an event already happened in the near past, it is more likely that it will happen again in the near future. The last coherence property of a scene is the *frame coherence*. It is the scene coherence plus temporal dimension. Frame coherence tells us that two successive frames of an animation are likely to be similar if the difference in time is small.

The reason why we want to divide the scene into small regions is to avoid doing the expensive ray-object intersection tests. And the reason why it works is based on the observation that small regions tend to intersect relatively few objects. Thus we can usually reduce the number of ray-object intersection tests at the expense of introducing ray-region intersection tests, using a spatial subdivision. One approach to accelerating ray tracing is to partition a scene by a regular grid. The concept of uniform grid is straightforward, so is the construction of its data structure. As we will see later, it is also very easy for a ray to step through the grid voxels as well. We will discuss uniform grids in detail in section 4. As with the bounding volumes approach, constructing hierarchical structure based on uniform grid can often achieve better performance. These structures are discussed in section 7.3.

Other spatial subdivision methods that divide the scene into non-uniform regions are also discussed. These structures include the BSP-tree, the k-D tree, and the octree. Binary space partition tree (BSP-tree) in ray tracing literature is a general term. Although the name BSP-tree has been mentioned in many ray tracing applications, what it often meant is axis-aligned BSP-trees, which is a special case of the general BSP-tree. Many researchers have shown that BSP-tree provides an efficient data structure to improve the ray traversal algorithms through the use of a spatial subdivision. We will discuss BSP-trees at three different levels in this survey. Section 7.1.1 introduces the most general type of BSP-trees. It provides a general framework of binary space partitioning approach. In section 7.1.2, we take a closer look at a special case of the BSP-tree: the k-D tree. It represents a scene partitioned into axis-aligned parallelepipeds. The octree, which can be viewed as a special case of k-D tree, is discussed in Section 7.2.

The *octree* is one of the most popular data structures for ray tracing. It is a rooted tree. Each internal node in the tree has eight children. The octree is the three-dimensional version of the *quadtree* whose internal nodes have four children representing the four *quadrants* in 2-space. The internal node of an octree corresponds to a three-dimensional box. For an internal node v in the octree, the children of node v are the *octants* of v, each of which is an axis-parallel box. Each octant is one of the eight subboxes of its parent. The external nodes in the octree comprise an *octree subdivision* of the cube of the root node. Octree is well studied and understood in computational geometry [31] and computer graphics [38]. In-depth studies of various kind of octrees can be found in [100,102]. As Samet [102] points out, several researchers discovered the octree subdivision method independently in late 1970s and early 1980s. For example, Hunter's [65] Ph.D. thesis is an early treatment of the octree subdivision method.

Once a data structure is constructed, to traverse it during the ray tracing phase, we need to go from one node to the next. Several methods can help us find the neighbor node efficiently. These methods are often referred to as *neighbor finding techniques* [103]. Taking octree as an example, we can encode the position of octree boxes as octal numbers, and use these numbers to search through all nodes. We shall survey various neighbor finding techniques later in this survey.

All of the data structures mentioned above show their strength in some cases but do not behave well at all times. Researchers try to combine two or more data structures in order to benefit from the merits of both. These data structures are called *hybrids*. We classify hybrid structures into three types. The first consists of flat-flat hybrids that mix different kinds of "flat" structures such as bounding volumes and uniform grids. The second type, hierarchical-hierarchical hybrids, is based on combining several different hierarchical structures. The third type are called hierarchical-flat hybrids. They are more sophisticated structures which combine not only different hierarchical structures but flat structures as well.

A common problem of space-oriented partition schemes is sometimes an object is divided into several pieces and stored in all of the nodes that represent the regions intersected with the object. To avoid redundant ray-object intersection tests, we can associate each object primitive with a *mailbox* [9] or a *rayID* [6]. Each ray is given a unique number as the ray identifier. We can store the information of the latest ray-object examination into the mailbox. If the object is examined by a ray, the ray identifier is stored at the object's mailbox. This way we can avoid redundant tests by examining the mailbox first before the actual ray-object examination is performed.

PART II

Flat Structures

Flat structures are the simplest data structures for ray tracing. There are two approaches to construct a flat structure. One is flat object-oriented partition (FOOP) approach, the other is flat space-oriented partition (FSOP) approach. The former surrounds each object with an object extent. The extent usually has simpler shape than the enclosed object. Thus testing ray intersection with the extent is faster than testing the enclosed object. Various structures using FOOP approach are introduced in section 3. Using flat space-oriented partitioning (FSOP) approach, a scene can be divided into smaller regions. The most commonly used technique is the uniform grid method. We will discuss structures using FSOP approach in section 4.

3 Flat Object-Oriented Partitioning – Bounding Volumes

FOOP approach for ray tracing is implemented by various types of bounding volumes. As we mentioned in the introduction, the reason for using a bounding volume around the object is to reduce the number of ray-object intersection tests. During ray traversal, if the ray that passes through the scene does not hit the bounding volume, it cannot hit the enclosed object. This way, we can avoid the expensive computational cost for intersection test with the object itself. For this reason, a bounding volume should have a simpler shape than the enclosed object.

3.1 Fundamentals of Bounding Volumes

It is difficult to define an optimal bounding volume [117]. Whitted [120] chooses a sphere as the bounding volume for each object because of its simplicity of representation and ease of performing the intersection calculation. In early days, a brute-force ray tracer spent almost all of the time at computing the intersection between the ray and the objects [7]. This makes a bounding volume a good candidate for accelerating basic ray tracing. This technique is so popular that all of the contemporary ray tracers that we know use some form of bounding volumes to expedite the speed of ray traversal.

There are various types of bounding volumes. The cost of intersection tests can be reduced greatly if the bounding volumes are chosen cleverly. Figure 3 lists four commonly used bounding volumes that are described by Hanrahan [60]. They are sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), and slab. The enclosed "Dragon" [79] consists of 1,132,830 triangles. A brute-force way to determine whether a ray hit the dragon is to do intersection tests on all of the triangles. If the ray does not hit any triangle, we conclude that the ray does not hit the dragon. Running time of this approach is proportional to the number of triangles in the dragon.

The sphere is the easiest and fastest extent for testing intersections with a ray. A minimum-radius bounding sphere for an object with k vertices can be constructed simply using linear programming in $O(k)$ time. Whitted [120] chooses spherical extents for their ray tracer for this reason. The drawback of a sphere is that it usually cannot fit the enclosed object very tightly. For long and skinny objects, there is a lot of empty space between the object and its extent. Weghorst et al. [117] point out the difference in area between the orthogonal projection of the object and its extent onto a plane perpendicular to the ray is an important factor that affects the performance of a ray tracer. This empty area is called *void area*. If the void area is large, we may still have to perform the ray-object intersection test, even though the ray is relatively far from the object. Therefore, choosing a tight bounding volume becomes an important issue.

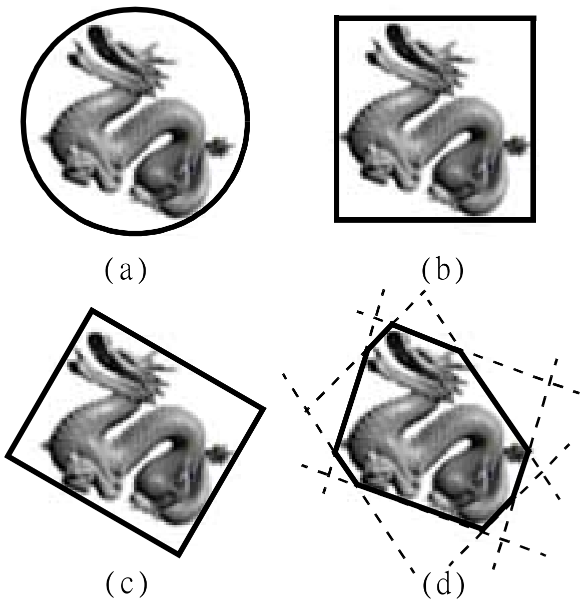


Figure 3: Commonly used bounding volumes: (a) sphere, (b) axis-aligned bounding box (AABB), (c) oriented bounding box (OBB), (d) four slabs

3.2 Slabs

To overcome the problem mentioned in the previous section, Kay and Kajiya [73] use *slab* as an extent. Figure 4 shows a teapot enclosed by a slab defined by two parallel planes. Given an object o in 3-space and a plane with unit normal $\begin{pmatrix} A \\ B \\ C \end{pmatrix}$, the slab for o is the closed region between two parallel planes defined by the implicit function $Ax + By + Cz - d = 0$, where $d = d_{min}$ or d_{max} are the signed distances of the planes from the origin.

To define a bounding volume in 3-space, we need at least three slabs. To avoid the overhead of finding the set of tightly fit slabs for each object, Kay and Kajiya [73] pre-select seven plane-set normals in advance, while Klosowski et al. [77] use 13 slab directions. The directions of pre-selected slabs are fixed independent of the objects to be bounded. The choice of slabs is to make the bounding volume tightly fit the primitive objects but also allow for efficient intersection tests between a ray and the bounding volumes. Weghorst et al. [117] discussed some criteria for choosing the slabs. Both Kay and Kajiya's [73] and Weghorst's [117] approaches produce bounding volumes that fit the enclosed objects tightly without having to compute the convex hull for each object. Although the convex hull can fit a primitive object very tightly, it is not used in ray tracing applications since the cost of intersection test between the ray and the convex hull is too high. Using a set of slabs with fixed orientation as bounding volume can fit to the enclosed object relatively well compared to other extents; see, for example, Figure 3. However, the disadvantage of the slab method is not only that we need more memory space to store plane-set normals and corresponding d_{min} and d_{max} for each object, but also that the computation for the set of slabs for each object is not trivial.

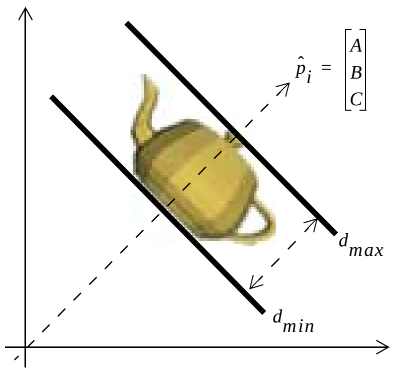


Figure 4: A slab

3.3 Bounding Boxes

An alternative approach to trade-off between the tight bound and the ease of computation is to use an axis-aligned bounding box (AABB), as described by Youssef [126] and Haine [56]. Although AABB approach cannot fit the enclosed object as tightly as slabs, the construction of AABB is cheaper than for slabs, in terms of time and space. If a tight bounding volume is the concern, the oriented bounding box (OBB) can be used. Unlike AABB, the orientation of OBB depends on the orientation of the enclosed object. OBB is widely used in the application for ray tracing [13] and collision detection [127]. For example, the OBB implemented by Gottschalk et al. [55] is aligned with the distribution of the enclosed polygon vertices using *principal component analysis* technique [83, 124]. First, we compute the covariance matrix of the data set. Then compute the eigenvalues and corresponding eigenvectors of the covariance matrix. The resulting eigenvectors can be used to define the new coordinate system of the bounding box by a linear transformation. The linear transformation matrix composed of the eigenvectors is called the principal component. Similar idea is used by Barequet et al. [14]. The difference between these two approaches is the latter uses the principal component of primitive objects for only one direction. The other directions are computed by another method.

OBB provides better fit than AABB with the trade-off of extra transformation cost for every ray-extent intersection test. To calculate the AABB of an object with k vertices, we can simply scan over the vertices of the object to find the minimum and the maximum coordinates along each axis direction in $O(k)$ time. A minimum-volume OBB, on the other hand, is usually not very easy to find. O'Rourke [88] presents an $O(k^3)$ time algorithm to compute the minimum-volume OBB for a set k points in $\mathbb{R}^3$. Barequet and Har-Peled [15] improve this result by proving that there exists an approximation algorithm that can obtain an approximation to the minimum-volume OBB in time $O(k \log^2 k)$. A randomized version of their algorithm can solve this problem in $O(k \log k)$ expected running time. Other types of bounding volumes such as cone [100], prism [14] and cheesecake [71] can also be used for special purposes. Most of these bounding volumes can be approximated in $O(k)$ time using heuristic algorithms. Simplicity of calculation is still the common criterion of selecting bounding volumes for many ray tracing applications.

4 Flat Space-Oriented Partitioning – Uniform Grids

Flat Space-Oriented Partitioning (FSOP) approach for ray tracing is most often implemented by a uniform grid. If we divide the whole scene into N_x, N_y, and N_z intervals along x-, y-, and z-axes, respectively, the three-dimensional scene is partitioned into $N_x \times N_y \times N_z$ axis-aligned grid cells. To make the analysis of time and space complexity easier, we often assume $N_x = N_y = N_z = N$ [22,23]. The universal space is then partitioned into axis-parallel cuboidal cells. Although dividing a scene into uniform grid was shown to be very simple and efficient, the choice of grid size is the major factor that can affect the ray tracing speed. In this section, we describe various ways of constructing uniform grid first. Ray traversal on a uniform grid is based on an incremental algorithm which we will describe later in this section.

4.1 Fundamentals of Uniform Grid

Uniform grid spatial subdivision approach for ray tracing was first introduced by Fujimoto and Iwata in 1985 [46]. It was proposed as a more efficient alternative to the octree. The basic idea is trying to get rid of expensive vertical movements in the octree (see Section 7.2). Each cell in the uniform grid represents a voxel. We use the terms "voxel" and "grid cell" in this section interchangeably. During ray traversal, the ray-object intersection tests are performed only on objects meeting the voxels that are penetrated by the ray. Figure 5 illustrates a uniform grid in two dimensions. The entire scene is divided into 6×6 voxels. There are 10 objects in the scene represented by ellipses. Ray R originating at point p passes through the scene without hitting any of the objects. Squares intersected by the ray are shaded in the figure. Only the objects that intersect the shaded area need to perform the intersection tests. In this example, only 3 out of 10 objects are tested against the ray. These three objects are shown as shaded ovals.

4.2 Constructing Uniform Grids

Fujimoto et al. [46,47] call their uniform grid SEADS (Spatially Enumerated Auxiliary Data Structure). It uses a three-dimensional array to map the corresponding voxels in the scene. In the preprocessing stage, the information about the objects in the scene is stored into the corresponding array element that represents the voxel intersected by one or more objects. SEADS allows very simple and fast ray traversal using 3DDDA algorithm. 3DDDA will be discussed in the next section when we discuss the ray traversal methods. This data structure is completely independent of object shape and topology. It only relies on the pre-selected resolution of the uniform grid. The work of Fujimoto et al. [46,47] was a technology breakthrough at that time. It shows good performance improvement over octree using their test scenes. There is an interesting point in Fujimoto's approach: What is the optimal grid

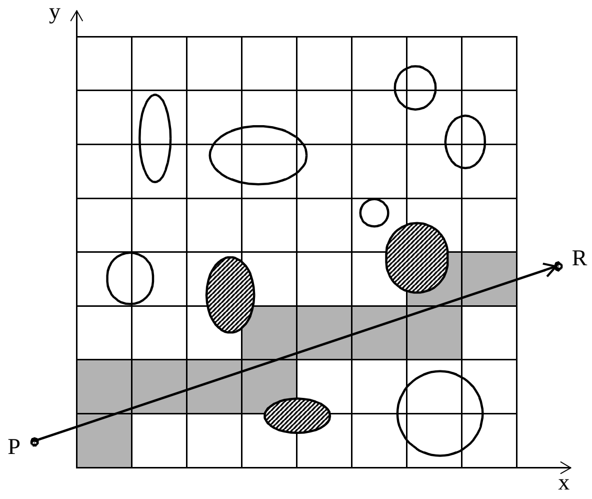


Figure 5: Illustration of uniform grid

size for a given scene and how can we find it automatically? Up to now, no one can give a definite answer to this question.

A similar data structure that also employs 3D array to store the object information is introduced by Yagel et al. [125]. Their grid size is chosen to be equal to the unit voxel, i.e., the same as the maximum scene resolution. Therefore, for a scene with a resolution of 1000 pixels on each side, Yagel's approach will require a $1000 \times 1000 \times 1000$ array. In the preprocessing phase, first scan-convert each of the geometric objects comprising the scene into a discrete voxel representation. Each array element in their data structure represents a 3D discrete raster of voxel in the same way as a 2D raster of pixels represents a 2D image. Since a voxel is very small, only a single object is allowed in each voxel. Therefore, there is no need to store a list of objects that are intersected with the voxel. In addition to object and coordinate information, an array element also stores all of the view-independent attributes that can be precomputed during the preprocessing phase. The attributes include surface normal, texture color and light source visibility and illumination. However, because so much information is stored for each voxel in the array, the resulting data structure pushes memory usage to the extreme. Yagel et al. assume memory usage will not be a problem in the future. Therefore, they only consider the ray traversal speed, not the memory space consumption. Their experimental results indicate that the data structure construction time is linear in the number of objects if the resolution is fixed.

There are two major differences between Yagel et al.'s data structure [125] and Fujimoto et al.'s SEADS [46, 47]. The latter divides a scene into voxels. Each side of the scene has the same resolution. Each voxel represented by SEADS is a box and does not have to be a cube. On the other hand, a voxel in Yagel's data structure represents the smallest unit in three-dimensional space, which implies that each side of the voxel is equal to a unit length. Thus a voxel represents a unit cube in 3-space using Yagel's approach. The other difference is a voxel in SEADS stores a list of objects that intersect this voxel, while Yagel's voxel can only store one geometric primitive, as it is assumed to be the limit of resolution.

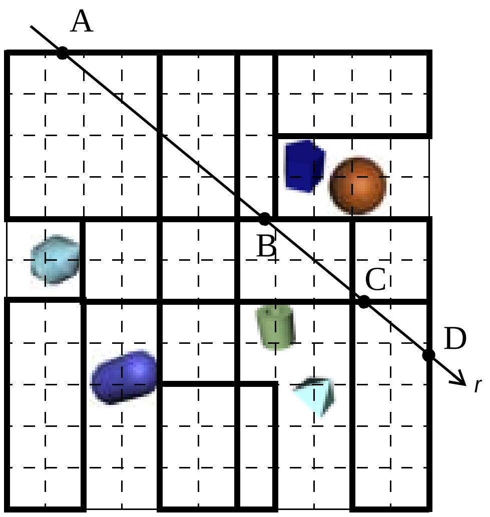


Figure 6: A r ray passes through macro-regions. (Not all of the macro-regions are drawn.)

Another interesting approach that makes use of empty voxels is proposed by Devillers [34]. During the preprocessing, a list of axis-aligned bounding boxes, called *macro-regions*, is constructed. Each macro-region is a maximal box of empty voxels, as shown in Figure 6. The thick rectangles represent some of the macro-regions for the empty voxels. Since macro-regions can overlap, each empty voxel may point to one or more macro-regions that enclose it. Ray r traversing empty voxels skips uninteresting voxels by only examining the farthest intersection point of the ray and a macro-region pointed by the current voxel. For example, in Figure 6, instead of moving the ray voxel-by-voxel incrementally, only four ray-voxel intersection tests are required for the ray to traverse the entire scene. They are marked as A, B, C, and D.

Although macro-regions can help us skip uninteresting empty voxels, the construction of this data structure is time consuming. Moreover, due to the overlapping nature of macro-regions, the data structure needs to consume additional memory space. Cohen et al. [27] use an idea similar to macro-regions and present another data structure that is easier to construct and does not require extra space. As in Devillers' macro-region approach [34], Cohen's structure also stores information in the empty voxels to assist ray traversal. Instead of putting pointers to macro-regions, empty voxels are filled with scene-dependent information that indicates the proximity to the surrounding objects in the preprocessing stage. The information stored in an empty grid cell defines a *free-zone* in which it resides. Thus it is possible to *skip* empty cells along the ray's direction without missing a possible intersection with an object. The difference between macro-regions and free-zones is the latter do not overlap. This idea is similar to Yagel's modified RRT approach mentioned in Section 4.3 which stores proximity flags in the empty cells to indicate the cells are in the object vicinity.

The approach of Cohen et al. is to construct a uniform grid first and then build a conceptually "flat" octree based on the uniform grid. The grid cells are further classified using the same philosophy as octree described in Section 7.2. The empty space is subdivided into smaller grids if it is close to an object. However, there is no tree structure constructed.

We can look at the subdivided space as a *flat pyramid*. In order to construct a flat pyramid as mentioned, each grid cell has to use two extra flags to provide the "regional" information. One of the flags is to indicate whether the grid cell is empty or not. This can be done by stealing the most significant bit of the cell word as an empty/non-empty flag so that there's no extra space needed for this flag. The second flag indicates to which region the grid cell belongs. This can be done by filling all of the empty cells with an index that indicates the region information as shown in Figure 7. A grid cell with index i means it belongs to a region that has 2^i by 2^i pixels in 2D case, or 2^i by 2^i by 2^i voxels in 3D.

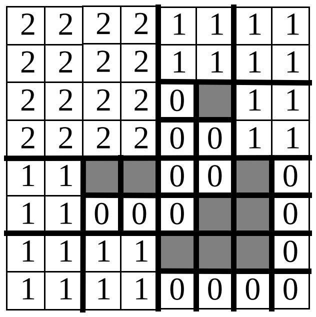


Figure 7: Illustration of flat pyramid (from [27])

In addition to the grid index, Cohen et al. also store the distance information in the empty grid cells to reduce the cost of a single step as well as the number of steps when rays traverse the scene. During the preprocessing stage, empty voxels are filled with scene-dependent information indicating the proximity to the nearest object. The voxels around an object define the zone of the same distance to the object. Cohen et al. call these zones *proximity clouds* due to the flexibility in terms of both shapes and functionality. Each cloud layer indicates a certain distance from the nearest non-empty cell. A ray enters a cloud cell can then safely skip a distance determined by the value stored in the cell.

4.3 Traversal Methods for Uniform Grids

Ray traversal on the uniform grid [46, 47, 6, 78, 125, 114, 89] is based on the incremental algorithm for line drawing on 2D raster grid. The line generating algorithm is known as *digital differential analyzer (DDA)*. More detailed description of DDA can be found in the book of Foley et al. [38]. Before we get into various ray traversal approaches, we would like to briefly explain the DDA algorithm.

Let us consider a line $y = mx + B$ entering the raster grid and reaching point (x_i, y_i), as shown in the lower left corner of Figure 8. We assume $0 \leq m \leq 1$, other slopes can be handled by suitable reflections about the axes. The actual pixel generated for the line at this point is $(x_i, Round(y_i))$, where $Round(y_i) = Floor(0.5 + y_i)$. Suppose the grid size is one. The next pixel generated for the line is based on the intersection point of the line and the vertical line $x = x_i + 1$. Since the grid size is fixed at one, the x-coordinate for the next pixel can be expressed in terms of the x-coordinate of the current pixel, i.e., $x_{i+1} = x_i + 1$.

The y-coordinate of the next pixel can be expressed as $y_{i+1} = Round(y_i + m)$. Following this method, all of the pixels can then be generated incrementally based only on the previously calculated result.

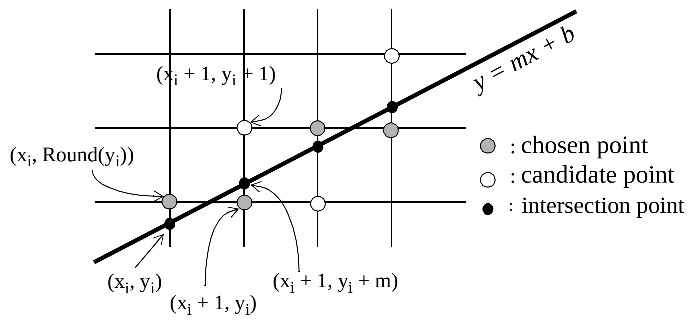


Figure 8: The basic DDA Algorithm for raster graphics.

A more efficient DDA known as the *midpoint line algorithm* was introduced by Bresenham [20] and improved by van Aken and Novak [5]. This incremental algorithm uses only integer arithmetic to calculate the coordinate of the next pixel. Consider Figure 9, the line is represented by implicit function $F(x, y) = ax + by + c = 0$. The midpoint line scan-convert algorithm relies on a *decision variable* d, defined as $d = F(M)$, where M is a point with coordinates $(x_p + 1, y_p + \frac{1}{2})$. The incremental algorithm starts at a point with integer coordinate (x_0, y_0). If we define $dy = y_1 - y_0$ and $dx = x_1 - x_0$, the slope-intercept form of the line can be written as $y = \frac{dy}{dx}x + B$. Line $F(x, y)$ can be expressed by $F(x, y) = dy \cdot x - dx \cdot y + B \cdot dx = 0$. Here $a = dy$, $b = -dx$, and $c = B \cdot dx$ in the implicit form. The decision variable can be expressed by the implicit function

$$d = a(x_p + 1) + b(y_p + \frac{1}{2}) + c \tag{1}$$

To determine whether we should go to point NE or point E (see Figure 9) in the next incremental step, we test the sign of d. Foley et al. show that the calculation can be transformed into pure integer arithmetic by multiplying all coefficients in equation (1) by 2, such that a, b, c, x_p and y_p are all integers. Since we only need know the sign for the decision variable, instead of testing the sign of d, we test the sign of $2d$ instead. If $2d > 0$ (and so is d), we increment both the current x and y coordinate by one. This means we choose the candidate point at the northeast corner (NE) of the current grid. Otherwise, only x increments by one. Thus the point at the east side (E) of current position is chosen. The algorithm works incrementally with only simple integer operations until it reaches the destination coordinate.

In the 80s when CISC machines were predominant, integer operations performed much faster than floating-point operations. Our test on a machine with AMD-K6/500 CPU running Linux operating system shows the midpoint line algorithm is 22.91 times faster than the original Bresenham's incremental algorithm using floating-point arithmetic. Even on a RISC machine such as Sun Sparc Ultra 5, although the difference of speeds between integer and floating-point operations are not as significant as on a CISC machine, the midpoint line

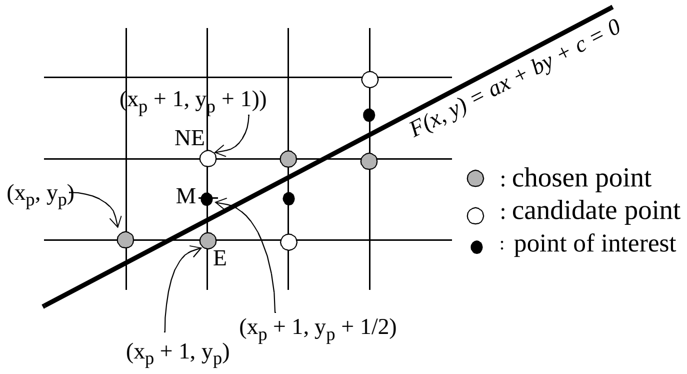


Figure 9: Illustration of midpoint line scan-convert algorithm. M is the midpoint. E and NE are the candidates points to be chosen. This algorithm can be implemented using only integer operations.

algorithm is still 6.65 times faster than the floating point version of the DDA incremental algorithm.

Now that we know how DDA works, let us look at the first ray traversal algorithm for uniform grid called 3D Digital Differential Analyzer (3DDDA), introduced by Fujimoto et al. [46,47]. 3DDDA is only a three-dimensional extension of two-dimensional DDA algorithm with minor modifications. It is a tool to enumerate the grid cells pierced by the ray in SEADS. Fujimoto et al. call the mechanism of employing 3DDDA on SEADS for ray tracing the *Accelerated Ray-Tracing System*, abbreviated as ARTS. Figure 10 uses 2D grid to explain the differences between Fujimoto's approach and Bresenham's algorithm.

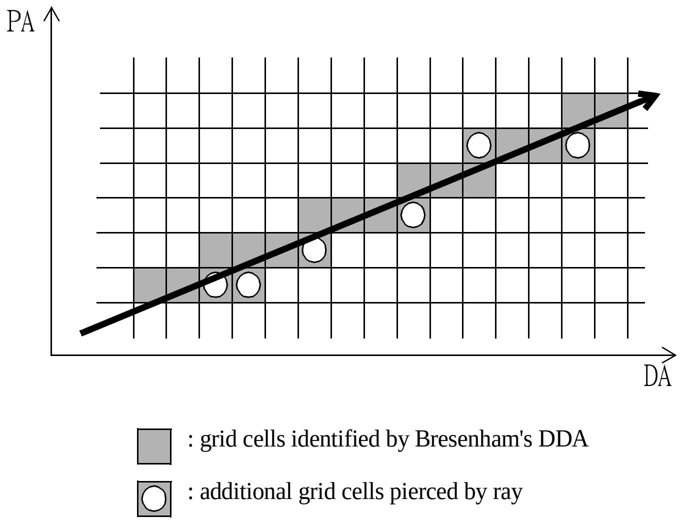

PA
DA
: grid cells identified by Bresenham's DDA
: additional grid cells pierced by ray

Figure 10: Comparison of Bresenham's algorithm and modified DDA.

First, the grid size in 3DDDA does not have to be one. Bresenham's algorithm always steps one pixel at a time. Second, Bresenham's algorithm only detects *some* of the cells met by the ray, namely those that are entered by crossing an edge (face) to the *driving axis* direction. (We call the axis of the greatest movement at each unit step the *driving axis* (DA). The other axes are *passive* (PA).) 3DDDA, on the other hand, has to check the cells that are pierced along PA direction as well. In Figure 10, the shaded cells represent the cells identified by Bresenham's algorithm. The cells that are pierced by the ray but not identified by Bresenham's algorithm are the cells with a circle. These are the additional cells identified by 3DDDA. The additional cells can be identified by checking the intersection of the ray and the planes that are parallel to the DA direction.

To implement 3DDDA, see Figure 11, the ray $z = f(x, y)$ is projected into two mutually perpendicular planes. Now we can use two synchronized DDA algorithms to track the ray $z = f(y)$ along DA-PA1 plane and the ray $z = f(x)$ along DA-PA2 plane. For each iteration of this incremental algorithm, we need to check all three directions along x-, y-, and z-axes for the grid cells pierced by the ray. If we apply midpoint line algorithm on both projected rays, 3DDDA can be implemented with only integer operations. A similar ray traversal algorithm also based on DDA algorithm is presented by Amanatides and Woo [6]. The main difference between their algorithm and Fujimoto's is Amanatides and Woo do not discriminate the driving axis and passive axis. This makes the implementation even easier than the original 3DDDA. Another difference is Amanatides and Woo use the ray coherence property (see section 2) to prevent redundant ray-object intersection tests.

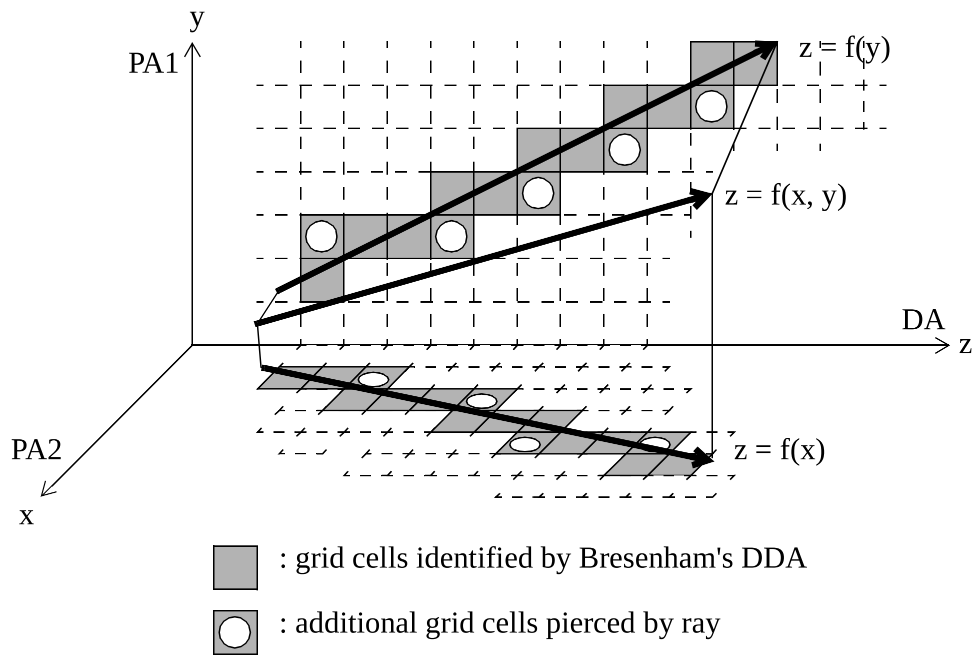
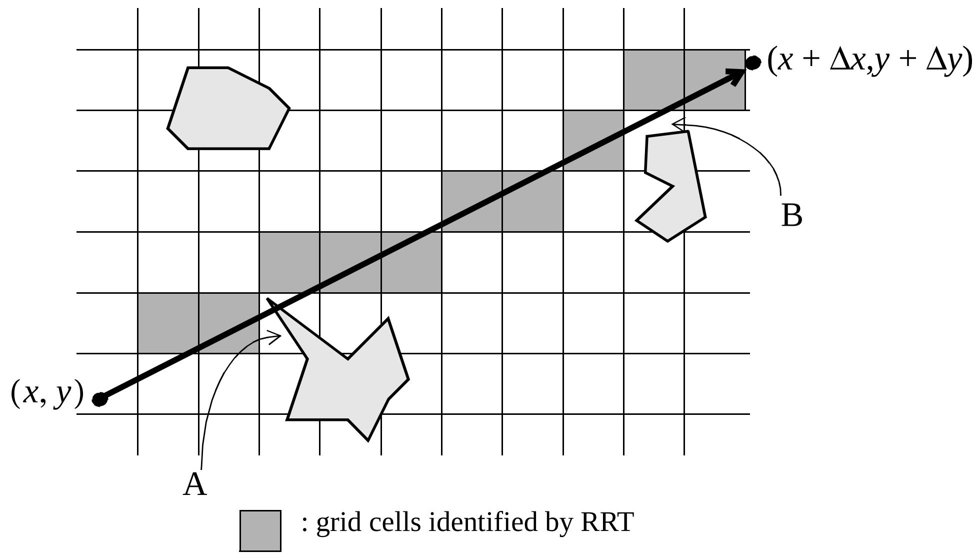

y
PA1
z = f(y)
z = f(x, y)
DA
z
PA2
x
z = f(x)
: grid cells identified by Bresenham's DDA
: additional grid cells pierced by ray

Figure 11: 3DDDA

Instead of ray traversing a geometric representation for 3D scene, Yagel et al. [125] introduce a mechanism for ray traversal that employs a 3D discrete raster of voxels for 3D scene. They call it *raster ray tracing* (RRT) method. Unlike ARTS, which intersect analytical rays with the object list to find the closest intersection, RRT employs 3D discrete rays traversed through the 3D raster to find the first voxel hit by the ray.

RRT is in fact a generalized version of Bresenham's algorithm. Following the same paradigm as 2D scan-convert algorithm, RRT is incremental and uses simple arithmetic. The only difference is RRT works on 3D scenes while Bresenham's algorithm is originally designed for 2D raster images only. Figure 12 uses 2D grid to illustrate the concept. A ray originated at point (x, y) traverses the scene and reaches the end point $(x + \Delta x, y + \Delta y)$. The three lightly shaded areas represent the objects in the scene. The dark grid cells are the voxels identified by RRT algorithm.



$(x + \Delta x, y + \Delta y)$
B
(x, y)
A
: grid cells identified by RRT

Figure 12: RRT ray traversal

Notice that at voxel A, the object is hit by the ray. However, RRT fails to identify it. The hit miss is due to the discrete nature of RRT line generator. Thus results in the lost of image quality. Also note that the ray passes through voxel B, but RRT skips this voxel without performing any intersection test. In this case, we avoid the ray-object intersection test without sacrificing the image quality. Yagel et al.'s empirical results show that there are less than 1.5 percent of hits missed with their approach. Therefore, RRT may be used if one can tolerate lower image quality. Since RRT only focuses on the ray-voxel intersection along the DA direction, the speed of moving the ray from one voxel to another is faster than 3DDDA which also has to consider the voxels hit by the PA directions. Several researchers tried to improve RRT algorithm in either software or hardware via alternative approaches. For example, Wang and Kaufman [114] present a 3D antialiasing algorithm employing volume sampling technique to resolve the hit miss problem in RRT. The idea is to employ a *filter weight function* and generate a "thick" ray such that the radius of the ray covers more than one voxel unit. The filter weight function is a weight function that specifies the magnitude of importance of each point within the filter support. Delfosse et al. [32] also point out the hit miss problem in RRT can be resolved by special graphics hardware.

Yagel et al.'s experimental results show that rendering time may decrease even though the number of objects increases when applying their RRT method to ray tracing. It is a common feature of the current widely available test scenes. When we put more and more objects into a test scene, the density of the object distribution grows. Consequently, the ray has a higher chance to hit an object without roaming too far. The running time of RRT is based on how many voxels the ray passes through. If the density of object distribution is high enough, there is a great chance for the ray to hit an object by only visiting a few voxels.

We discussed the data structure of proximity clouds in the previous section. It allows the ray to "skip" a *distance* between two arbitrary points along the ray direction. Cohen et al. [27] use L_p-*metrics* to describe the distance between two points.

An L_p-*metric* (see, e.g. [96, Definition 5.3, page 222]) is the distance between two arbitrary points $r = (r_1, r_2, ..., r_d)$ and $s = (s_1, s_2, ..., s_d)$ in Euclidean space $\mathbb{E}^d$ given by

$$\left(\sum_{i=1}^{d} |r_i - s_i|^p\right)^{1/p}, \text{ for any } p \geq 1. \tag{2}$$

We only focus on $\mathbb{E}^2$ for clarity. Let $r = (x_1, y_1)$ and $s = (x_2, y_2)$ be two points in $\mathbb{E}^2$, and let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. The distance between r and s can be expressed as $d_p(\Delta x, \Delta y) = (|\Delta x|^p + |\Delta y|^p)^{1/p}$. Some familiar examples of L_p-metrics are

1. L_1 is the City-Block (or Manhattan) distance defined as $d_1(\Delta x, \Delta y) = |\Delta x| + |\Delta y|$,
2. L_2 is the Euclidean distance defined as $d_2(\Delta x, \Delta y) = \sqrt{|\Delta x|^2 + |\Delta y|^2}$,
3. L_∞ is the Chessboard distance defined as $d_\infty(\Delta x, \Delta y) = max(|\Delta x|, |\Delta y|)$.

Cohen and Sheffer use L_p-*metrics* as follows. Consider a ray represented in the parametric form $R = R_0 + tR_d$, $t \geq 0$, where R_0 is the ray origin, $R_d = [c_x, c_y]$ is the direction vector of R. Let $R_1 = (x_1, y_1)$ be an arbitrary point on R, we want to find the coordinate of another point $R_2 = (x_2, y_2)$ on R, which is d units ahead of R_1. The coordinate of R_2 can be calculated by $x_2 = x_1 + d \cdot \frac{c_x}{d_p(c_x, c_y)}$, and $y_2 = y_1 + d \cdot \frac{c_y}{d_p(c_x, c_y)}$. The L_p-*metric* $d_p(c_x, c_y)$ is a constant, and only needs to be computed once for each ray. During ray traversal, instead of stepping through each grid cell one at a time, the ray can skip distance d at once, depending on the distance map that is pre-calculated in the preprocessing stage. If we calculate the distance map based on L_1-*metric*, a ray traversing the scene looks like the illustration on the left of Figure 13. If we calculate the distance map based on L_2-*metric*, the ray skips a Euclidean distance at each iteration as shown on the right of Figure 13. The triangles in the scene represent the objects. The black dots along the ray are the actual steps the ray will take to pass through the entire scene. Proximity cloud is useful for a sparse scene, since we can skip many intersection tests. However, for a dense scene, using proximity cloud can slow down the rendering process due to the overhead of calculating the L_p-metric.

In this section, we discussed uniform grid and its variation. Ray traversal in these structures are all based on DDA line algorithm. Ray tracer based on uniform grid (e.g., Rayshade 4.0 [78]) is very efficient because grid traversal is based on simple incremental algorithm which can be done using fast integer operations. The major drawback of these uniform grid structures is they all assume the objects are distributed uniformly. Therefore, the performance of

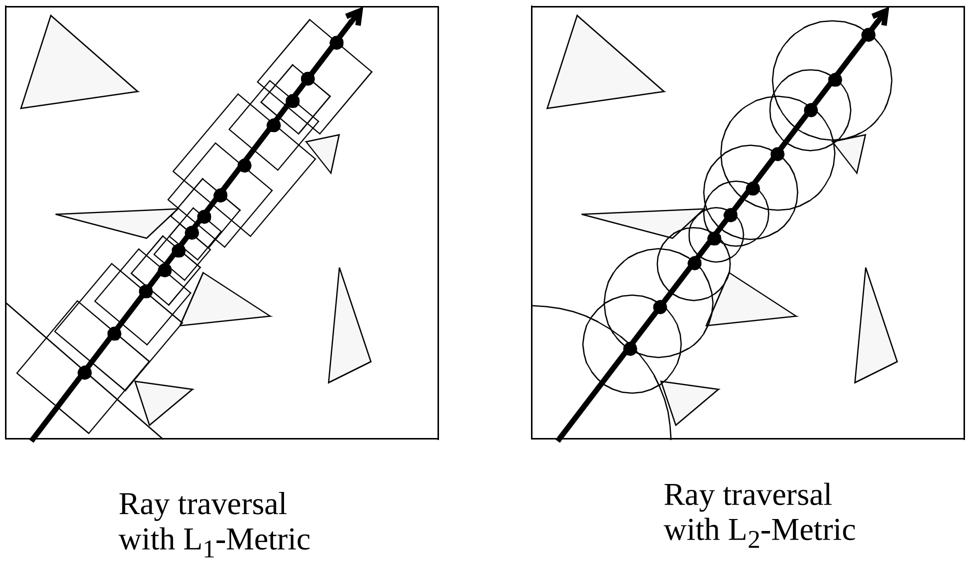

Ray traversal with L_1-Metric
Ray traversal with L_2-Metric

Figure 13: Ray traversal based on L_1 (left) and L_2 metrics (right)

ray traversal is very sensitive to the grid size, which has to be determined before constructing the space partition. M. Gigante [48] proposes a non-uniform grid structure to alleviate this problem. The advantage of this structure is we do not have to worry about picking the right grid size because it is less sensitive to the grid size. For objects that are distributed non-uniformly, non-grid structures perform better. We shall discuss those structures in later sections.

5 Flat Hybrid Structures

We have seen two types of flat data structures using FOOP and FSOP approaches in section 3 and 4. Now we would like to describe how they can be mixed together to construct a hybrid structure. First we would like to show that different "flavors" of FOOP itself can be mixed together. Then we describe how can different FSOP methods be combined. At the end of this section, we show FOOP and FSOP can also be combined to become another hybrid structure.

5.1 Flat OOP-OOP Hybrid

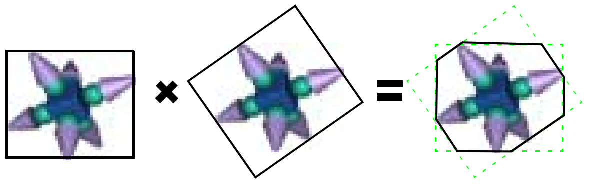


Figure 14: A hybrid structure obtained via intersecting AABB with OBB

An OOP-OOP hybrid usually fits the primitive object better than using just one type of bounding volume. Thus fewer ray-object intersection tests are needed. Kay and Kajiya [73] describe a way of combining AABB and transformed bounding box in order to fit the object more tightly than just using AABB. The object is enclosed within the *intersection* of the two bounding boxes. As shown in Figure 14, the object on the left is enclosed with an AABB, the same object in the middle is enclosed with an OBB. A new hybrid extent produced by intersecting these two bounding boxes is shown on the right hand side.

Arvo and Kirk [13] describe an alternative way of combining different types of bounding volumes. An example of this approach is shown in Figure 15. The object in this figure is the famous *Unfinished Slave 'Atlas'* statue by Michelangelo. This image is reconstructed by Stanford University Computer Graphics Laboratory [79]. Object Atlas has approximately 250 million vertices and 500 million triangles. For such a complicated object, we can cover part of the object by two or more bounding volumes. A new hybrid structure that covers the entire Atlas can be obtained by the *union* of these bounding volumes. In Figure 15, we first enclose part of Atlas object with a sphere extent. Then enclose other part of the object with an AABB. A new Sphere-AABB hybrid structure obtained by the union of the two bounding volumes is shown on the right.

The difference between the "intersection" type of hybrid (such as AABB-OBB hybrid) and the "union" type of hybrid (such as Sphere-AABB hybrid) is: for the former hybrid, ray-object intersection test will be executed only if the ray hits *all* of the bounding volumes. Thus ray-extent intersection tests must be performed on all of the extents before doing the ray-object intersection test. On the other hand, for the latter hybrid, ray-object intersection test will have to be performed if the ray hits *any* of the bounding volumes. The cost of test for a union type hybrid is more expensive than for an intersection type hybrid, if a ray hits the extent but misses the primitive object. However, union type hybrid fits the primitive object more tightly than intersection type hybrid. Therefore, the chance of a ray hitting the extent but missing the object can be greatly reduced.

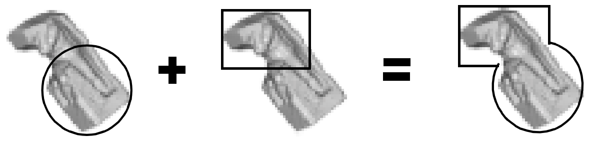


Figure 15: A hybrid structure obtained via the union of a sphere and an AABB

5.2 Flat SOP-SOP Hybrid

RRT approach introduces the hit miss problem, as we discussed in section 4. For scenes that are sensitive to the quality, Yagel et al. [125] propose a hybrid approach that can eliminate the hit miss problem. Instead of changing the underlying uniform grid structure, they use a hybrid ray traversal method on the same data structure. The solution is to combine their RRT with Fujimoto's 3DDDA [46, 47]. We call this hybrid traversal approach SRRT, meaning Semi-RRT approach. To implement an SRRT, we need to add a proximity flag for all the voxels around the object surface to indicate that we are in the vicinity of an object. This can be done similarly to the way Cohen et al. [27] construct their free-zone.

During the ray traversal phase, RRT method is used if the proximity flag is off, which indicates the voxel is in empty space. If the ray encounters a voxel with proximity flag turned on, we immediately switch to 3DDDA method instead. As a result, the speed of ray traversal using SRRT is fast in empty space and slows down when it is close to an object. The speed of SRRT is slightly slower than the original RRT but it guarantees that no hit miss will occur. Yagel et al. claim that SRRT is a *constant time* ray tracer. It is true that uniform grid subdivision is insensitive to the number of objects in the scene. The speed of ray traversal only depends on the number of voxels a ray traversed. For a scene with resolution 1000 pixels on each side, therefore, SRRT is a constant time ray tracer with respect to the number of object but with a constant factor of 2000 in the worst case.

5.3 Flat SOP-OOP Hybrid

The OOP approach can reduce the number of ray-object intersection tests, but the ray-extent intersection tests are still inevitable. Although testing intersection between ray and the extent is usually faster than testing intersection between ray and the object, we still have to perform many ray-extent intersection tests, if the number of objects is large. The total number of intersections cannot be reduced using flat bounding volumes. On the other hand, constructing SOP data structure such as uniform grid can help us reduce the number of intersection tests. The speed of *each* intersection test is still the same, i.e., OOP approach speeds up ray tracing by replacing complicated intersection test with simpler one. The total number of intersection tests cannot be reduced this way. SOP approach speeds up ray tracing by reducing the number of intersection tests. However, for each object, the cost of ray-object intersection test cannot be reduced.

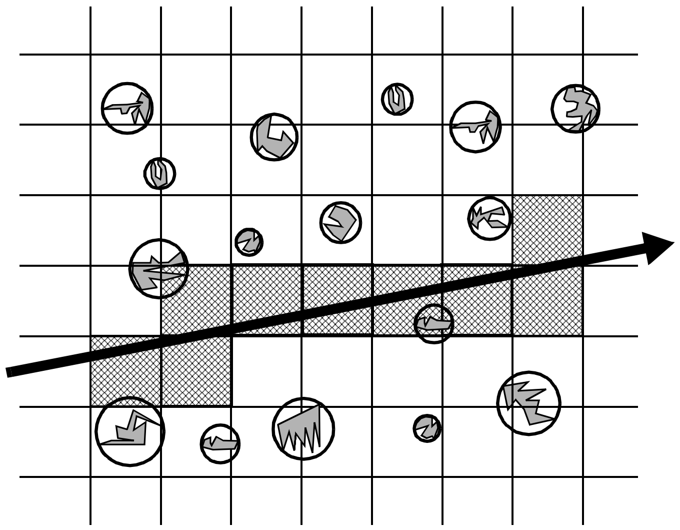


Figure 16: An SOP-OOP hybrid structure obtained by combining uniform grid and spherical extents

A number of researchers have addressed the idea of combining SOP and OOP methods to gain the benefits from each. Constructing a flat SOP-OOP hybrid structure is straightforward: we first enclose all of the primitive objects with our favorite extents, as described in Section 3, then a uniform grid can be build on top of these extents using any of the methods described in Section 4. Figure 16 shows an example of combining uniform grid and spherical extents. Only those objects whose extents meet the shaded areas of the uniform grid need to test for intersection. In the figure, only ray-sphere intersection tests will be performed because the ray does not hit any extent.

If the number of objects is small, and each object is complicated, we can even construct a flat OOP-SOP hybrid as Figure 17. Object "Lucy" on the left has approximately 116 million triangles, object "Bunny" on the right has about 725,000 triangles. There are only two objects in the scene but each object is extremely complicated. In this case, each objects can be enclosed with an AABB first. Local uniform grids can then be constructed within each of the AABB. The whole structure becomes a new hybrid with OOP on top of SOP [75].

In general, there can be unlimited number of hybrid structures. For example, we can also construct a structure, if we prefer, that combines the OOP-OOP hybrid with the SOP-SOP hybrid that we mentioned in this section. Since we have seen only flat data structures so far, our current discussion on hybrid structures only covers the flat data structures. More combinations will be described after we discuss the hierarchical data structures in Part III (see Section 8).

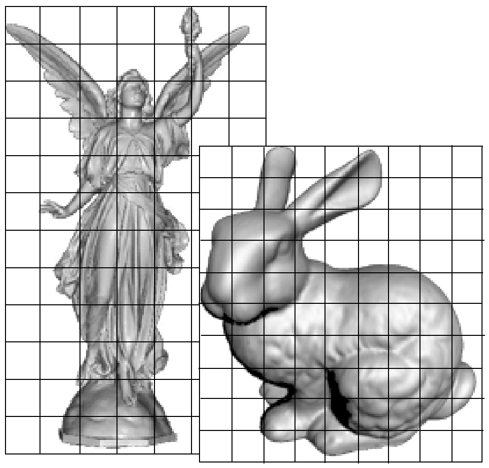


Figure 17: An OOP-SOP hybrid structure consists of AABB and uniform grid. These objects were reconstructed by Stanford University Computer Graphics Laboratory.

PART III

Hierarchical Structures

The object-oriented and space-oriented partitioning approaches can also be applied to hierarchical structures. As opposed to the flat structures, ray traversal in the hierarchical structures involves *vertical movements* in addition to *horizontal movements*. During horizontal movements, a ray only moves between neighboring regions that are at the same depth. During vertical movements, a ray may move from the higher level of the structure to the lower level or vice versa, since the neighboring regions may not have the same depth in the hierarchy. The hierarchical object-oriented structures (in section 6) are constructed using multiple levels of flat object-oriented structures. They are easier to implement than the hierarchical space-oriented structures (in section 7), which allow faster ray traversals due to their advanced features.

6 Hierarchical Object-Oriented Partitioning

Hierarchical object-oriented partitioning is realized by bounding volume hierarchies. In section 6.1, we define a bounding volume hierarchy more formally. Various criteria for constructing such a hierarchy are described in section 6.2. After it is built, we would like to know how a ray traverses it. Section 6.3 provides some answers to this question.

6.1 Bounding Volume Hierarchies

A *Bounding volume hierarchy* (BVH) is a rooted tree. Each node in the tree is a bounding volume. The internal node represents a bounding volume enclosing all the bounding volumes of its children. The leaf node is a bounding volume that encloses a primitive object. Figure 18 is an example of a two-level bounding volume hierarchy. On the left-hand side, each object is surrounded by a sphere extent. A big sphere that encloses all of the small spheres can be viewed as a parent with three small spheres as its children. The conceptual tree structure is drawn on the right-hand side of Figure 18.

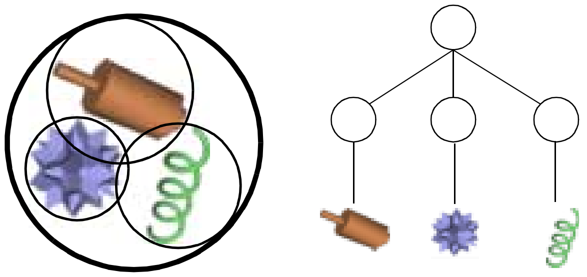


Figure 18: A two-level bounding volume hierarchy. The children of an internal node are also bounding volumes. Each leaf node points to a primitive object.

6.2 BVH Tree Construction

Although adding a bounding volume for each object can make the intersection tests faster (see section 3), the worst-case asymptotic running time for ray traversal is still $O(n)$, where n is the number of objects. It happens when the ray hits all of the bounding volumes but misses all of the enclosed objects. Creating a hierarchy tree of the bounding volumes can reduce the number of intersection tests by ignoring the uninteresting part of the tree and thus speed up the ray traversal time up to $O(\log n)$ if the resulting tree is balanced. There are two ways to build a BVH: bottom-up and top-down.

To construct a BVH from bottom up, the most straightforward way is just to enclose a fixed number of object extents into a larger extent. The number of children within each bounding volume is called the *branching factor* which indicates the maximum number of branches for each internal node. The extents in the higher level can be grouped together in a similar manner. The construction process continues until the number of extents is less that the branching factor. We then group the rest of the extents into a single extent. The last extent is the root of the hierarchy that represents the bounding volume of the whole scene. This approach is illustrated in Figure 19. Figure 19(a) is the input scene. We assume three objects are grouped together according to their order in the input. A BVH constructed by this method is shown in Figure 19(b). The preprocessing takes $O(n)$ time and space, where n is the number of objects. Although the straightforward method is easy to implement, there can be a lot of overlapped areas that make ray traversal very inefficient.

Weghorst et al. [117] suggest an alternative bottom-up approach to construct a BVH with a fixed branching factor. Before the tree construction, the objects are sorted by their x-coordinate. We then proceed as in the straightforward approach. By pre-sorting the objects before construction, objects that are close to each other can be put into the same cluster. The *object coherence* is automatically taken into account using this approach. The object coherence property expresses the fact that objects tend to consist of pieces that are connected or close to each other, and that disjoint objects tend to be largely disjoint in space [13]. A BVH constructed by Weghorst's approach [117] can reduce the overlapped areas between different groups of bounding volumes because the proximity between objects is taken into account while building the hierarchy. Although it takes $O(n \log n)$ time to construct the hierarchy with n objects, ray traversal on this structure is more efficient.

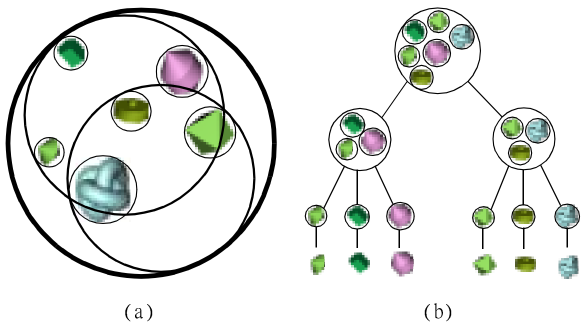


Figure 19: A straightforward bottom-up approach to construct a bounding volume hierarchy. (a) Three objects are grouped together based on the input order. (b) The corresponding tree structure of the scene.

Kay and Kajiya [73] present a similar bottom-up approach for constructing BVH that also consider the proximity between objects. The difference is the latter work uses slabs as the extents. Weghorst et al. [117] only consider sphere, AABB and cylinder as the candidates for extents.

As opposed to the bottom-up approach, Kay and Kajiya [73] introduce a BVH constructed in a top-down fashion. Branching factor is two in their approach. The key point in Kay and Kajiya's top-down approach is to find the median-cut in the object space. At each level, objects within a group are sorted by their x coordinate. These objects are then partitioned at their median. The descendant of the current node are two almost equal sized subgroups. The splitting process recurses until there is at most one object in the subtree. Since objects are split into two equal sized subgroups at each iteration, the final BVH is a balanced binary tree. Another similar top-down BVH construction is proposed by Smits [106]. One of the differences between Kay-Kajiya's and Smits' approach is the former uses slab extent while the latter chooses AABB. Another difference is that Kay and Kajiya always sort the objects along x coordinate at each level, Smits alternate different coordinates for sorting the objects at each level. The order of the sorting coordinate follows $x \rightarrow y \rightarrow z \rightarrow x$ cycle, i.e., at the top level, sort all objects along x coordinate. At the second level, sort all objects along y coordinate, and so on.

6.3 Ray Traversal in BVHs

Additional data structures are often required to assist the traversal in a BVH. A commonly used auxiliary data structure is a priority queue. Kay and Kajiya call it a heap. If we visit a node in BVH, the node is inserted into the heap. When we want to explore a node, it is extracted from the heap. The heap implemented by Kay and Kajiya is maintained dynamically for each ray and is organized by the distance of the bounding volumes along the ray. Each element in the heap is a candidate to perform ray-object intersection test. Initially, only the root bounding volume is inserted into the heap. At each iteration, a candidate closest along the ray is extracted from the heap. Ray-extent intersection tests are performed on all of the children of this node. An extent is inserted into the heap only if it is hit by the ray. Ray-object intersection tests are performed if the node extracted from the heap is a leaf. The process continues until the heap is empty. The HeapBVHTraverse algorithm is summarized below.

Algorithm HeapBVHTraverse(ray, B)
Input: A ray and the root bounding volume B.
Output: The first object hit by ray if it exists.

```
1:  Initialize heap to contain only $B$;
2:  **while** heap is not empty **do**
3:     candidate ← ExtractMin(heap);
4:     **if** candidate is leaf **then**
5:        Perform ray-object intersection test;
6:     **else**
7:        **for each** *child* of candidate **do**
8:           **if** ray hits the bounding volume of *child* **then**
9:              InsertHeap(*child*);
10:          **end if**
11:       **end for**
12:    **end if**
13: **end while**
```

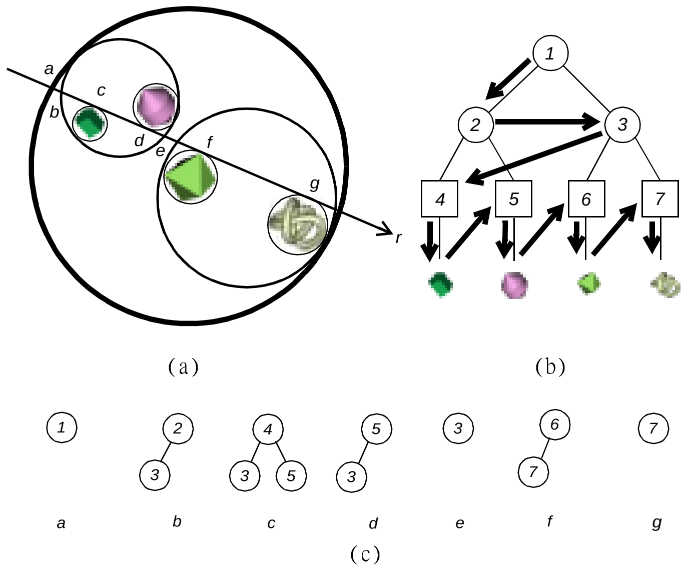


Figure 20: Ray traversal in BVH using Kay and Kajiya's approach [73] (a) A three-level BVH. (b) The corresponding tree structure. The arrows indicate the order in which the nodes were put into the heap. (c) The heap contents during ray traversal.

We now use Figure 20 as an example to illustrate how a ray traverses the BVH using Kay and Kajiya's method [73]. Figure 20(a) shows a BVH for four objects. A ray r passes through the scene, hitting every bounding volume without hitting any primitive object.

The labels from a to g represent the ray-extent intersection points. Figure 20(b) is the corresponding tree structure of this BVH. A heap is maintained during the ray traversal. The heap structure at each ray-extent intersection point is illustrated in Figure 20(c). At each intersection point, the following actions are performed on the heap:

1. At point a, ray r hits the root sphere representing the scene. The heap is initialized with node 1 as the only element in the heap, and then extracted from the heap. All of its children (in this case nodes 2 and 3) are inserted into the heap. The order of insertion depends on which child is pierced by the ray first.
2. At point b, node 2 is extracted from the heap to be examined. Since all of its children (i.e., nodes 4 and 5) are hit by the ray, they are inserted into the heap.
3. At point c, node 4 is extracted from the heap. Since it is a leaf node, ray-object intersection test is performed.
4. At point d, node 5 is extracted from the heap to perform intersection test.
5. At point e, node 3 is extracted from the heap, then both of its children are inserted into the heap.
6. At point f, extract node 6 for ray-extent test.
7. At point g, node 7 is extracted for ray-extent test.
8. The heap is empty, so traversal process stops.

In Figure 20, we arranged the scene such that the ray does not hit any object, to demonstrate the order in which the nodes are put into the heap. Now let us look at another example using the same approach. The ray hits an object this time. In Figure 21, once the ray enters the root sphere, the ray tracer performs intersection tests on all of its children. Since both bounding spheres 2 and 3 meet by the ray, they are inserted into the heap, in the order in which they are entered by it. The next step is to examine the children of sphere 2. Since only sphere 5 is intersected by the ray, only it is inserted into the heap. Next ray-object intersection test is performed on the object in sphere 5. Since no intersection is found, we move on to the next sphere in the heap, which is sphere 3. The next step is to add all the bounding spheres within sphere 3 that are intersected by the ray into the heap. Only sphere 6 is added in this case. At last, we perform ray-object intersection test on the object in sphere 6 and find an intersection. The process is then stopped because the heap is empty.

In contrast to heap assisted approach, Smits [106] employs a data structure similar to *skip list* [53, 118]. A skip list is a dictionary-like data structure that allows searching to be performed in $O(\log n)$ average running time. Unlike Kay and Kajiya's approach [73], the skip list is static. The list structure does not change for different ray directions. We do not need to maintain it during ray traversal. The way the skip list stores the nodes resembles depth-first order of the tree. Each internal node in the skip list has two pointers. One pointer

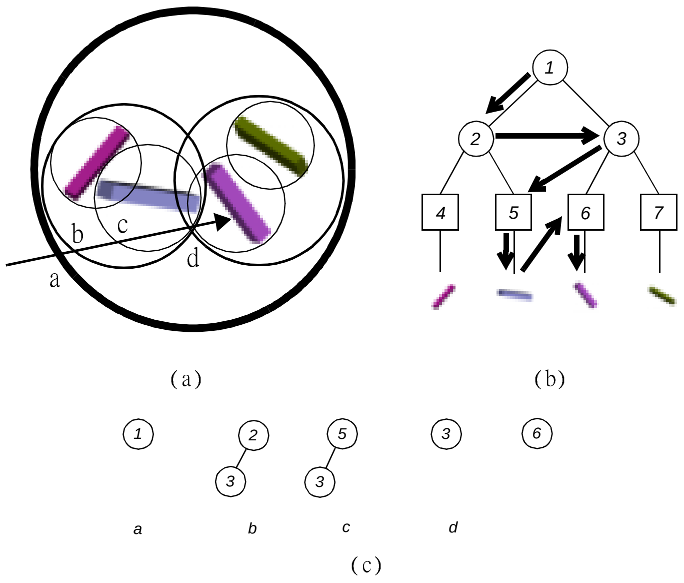


Figure 21: Ray traversal in BVH using Kay and Kajiya's approach [73]. In this case, the ray hit one of the objects in the scene. (a) A three-level BVH. (b) The corresponding tree structure. The arrows indicate the order of the nodes put into the heap. (c) The heap maintained during ray traversal.

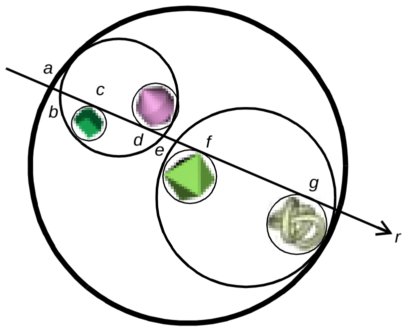
points to the next node to be visited by regular depth-first order. The other pointer points to the next skip node, usually a sibling of the current node. If a ray intersects an extent, we visit the regular next node in the list. Otherwise, we visit the skip node. The leaf node does not have the pointer to the skip node. It points to the primitive object instead. Ray-object intersection tests are performed only if the ray intersects the leaves. BVH traversal using a skip list can be implemented by the following algorithm.

Algorithm SkipBVHTraverse(ray, B)

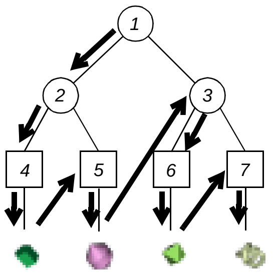
Input: A ray and the root bounding volume B.

Output: The first object hit by ray if it exist.

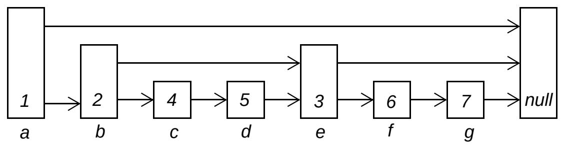
```
1:  $node \leftarrow B$;
2:  $O \leftarrow NULL$; {$O$ is the list of objects hit by the ray.}
3:  while $node \neq NULL$ do
4:    if $ray$ intersects the bounding volume of the current $node$ then
5:      if $node$ is a leaf node then
6:        Perform ray-object intersection test on object $o$ associated with $node$;
7:        if object $o$ is hit by the ray then
8:          $O \leftarrow O \cup o$;
9:        end if
10:     end if
11:     $node \leftarrow$ next node;
12:   else
13:     $node \leftarrow$ skip node;
```



(a)



(b)



(c)

Figure 22: Ray traversal in BVH using Smits' approach [106]. (a) The same BVH as Figure 20 is redrawn here for reference. (b) The corresponding tree structure and traversal path. (c) The skip list for ray traversal.

```
14:     end if
15: end while
16: if $O \neq NULL$ then
17:     return The first object in $O$ hit by the ray;
18: else
19:     return $NULL$;
20: end if
```

The same example we used for algorithm HEAPBVHTRAVERSE is drawn in Figure 22. We traverse the BVH with the help of skip list this time. Figure 22(a) is redrawn for reference. The traversal path is illustrated with thick arrows in Figure 22(b). The skip list structure is depicted in Figure 22(c). In this figure, we show a worst case example to illustrate the ray traversal path. This situation rarely happens in the real world. If a ray hits the bounding volume represented by node 1 without hitting the enclosed bounding volumes, we can avoid visiting all other nodes by following the link pointed by the skip list.

Figure 23 shows how to use Smits' approach to reduce the number of ray-object intersection tests. Here, at point a, the ray enters the congested root sphere, so we follow the *next* link of node 1 to examine node 2. The ray does not intersect node 4, so we proceed to node 5. The ray does not intersect the object in node 5 either, so we follow the link to test node 3. The ray intersects with node 3, so we go to its child node 6. There we find an object hit by the ray so we add it to the object list. The ray does not intersect node 7, so we can skip the ray-object intersection test there. At the end, the algorithm reports the object in node 6 is hit by the ray.

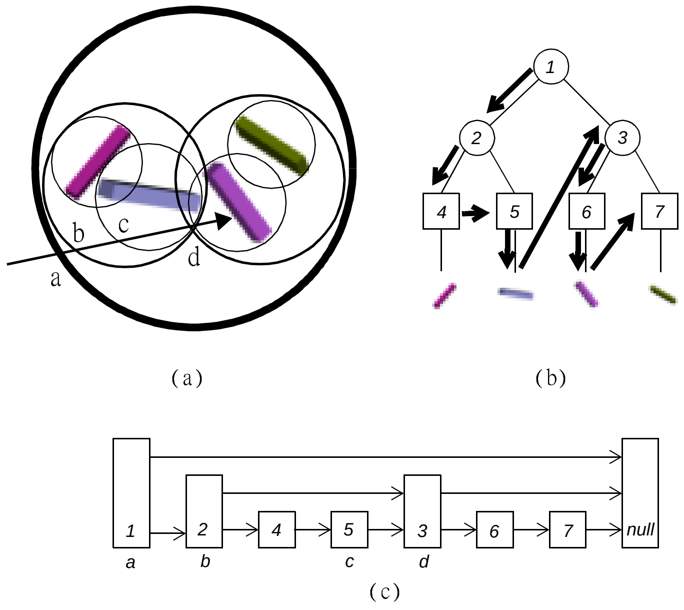


Figure 23: Ray traversal in BVH using Smits' approach [106]. (a) A ray that hits both bounding spheres. (b) The corresponding tree structure and traversal path. (c) The skip list for ray traversal.

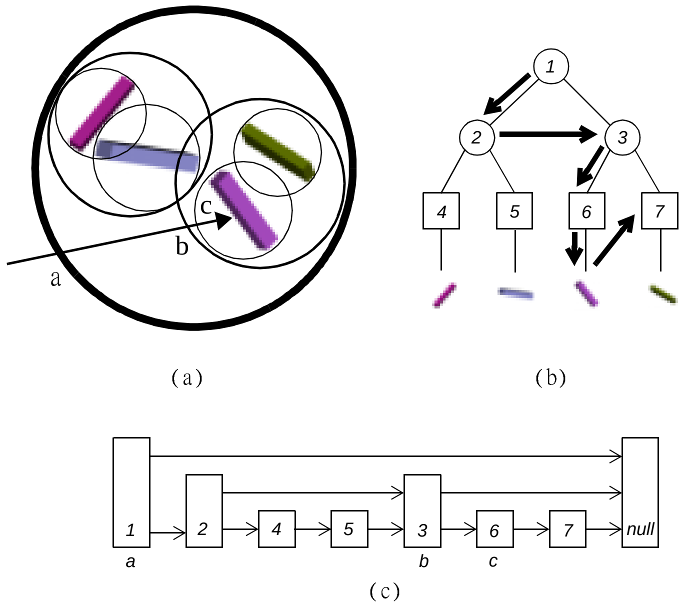


Figure 24: Ray traversal in BVH using Smits' approach [106]. (a) A ray hits only one of the bounding sphere. (b) The corresponding tree structure and traversal path. (c) The skip list for ray traversal.

To further illustrate how we can take advantage of the skip list, let us consider the example in Figure 24. In Figure 24(a), a ray enters the scene and hits only one bounding sphere. Since the ray does not hit sphere 2, we can skip all of the intersection test within sphere 2. Following the *skip* pointer in Figure 24(c), we can jump to sphere 3 and so on. Figure 24(b) shows the ray path where intersection test has to be performed.

To summarize, we discussed the BVH construction and ray traversal in this section. Since bounding volumes can overlap, to find the first intersection point, we have to keep a list of objects hit by the ray. After we find all of the objects pierced by the ray, we then pick the closest hit from the list. Hill [44] suggests that we can keep only the first eight hits to speed up the process. According to his experience, it is enough for most of the cases. This kind of bookkeeping job is not necessary for uniform grid because each grid cell is disjoint. However, this method may be useful for shadow rays. In that case, we only want to find out if there is any object blocking the light source. Once we find a hit, we can conclude the point hit by the primary ray is in shadow. Haines [58] proposes a way to improve Kay and Kajiya's heap approach [73]. The sorting process can be eliminated by treating the primary ray (find the *first* hit) and the shadow ray (find *any* hit) differently. In general, BVH approach is easy to implement, although implementing an efficient one is more difficult. Another advantage of BVH approach is its memory requirements are much less than for space-oriented partitions because BVH does not chop up objects into pieces.

7 Hierarchical Space-Oriented Partitioning

7.1 Two-Way Subdivisions

7.1.1 General BSP-trees

The Binary Space-Partitioning Tree (BSP-tree) was originally introduced by Fuchs, Kedem and Naylor [45] to determine the visible surfaces of a scene containing a set of polygons. These polygons are referred to as *scene polygons* [8]. The idea is to sort the scene polygons into a back-to-front ordering relative to a given viewpoint. However, front-to-back ordering [54] seems more suitable for a ray shooting query. In this section, we define a BSP-tree more formally and derive the construction algorithm directly from the definition.

Any hyperplane h in $\mathbb{R}^d$ can be expressed by an implicit function $H(x_1, x_2, \cdots, x_d) = a_{d+1} + \sum_{i=1}^{d} a_i x_i = 0$. Let

$$h^+ = \{(x_1, \cdots, x_d) \,|\, H(x_1, \cdots, x_d) > 0\}$$

and

$$h^- = \{(x_1, \cdots, x_d) \,|\, H(x_1, \cdots, x_d) < 0\}$$

be the positive and negative open half-spaces bounded by h, respectively. Let δ be a fixed constant – the maximum number of objects meeting a node, we call δ the *capacity* of the node. The *general BSP-tree* for a set $\mathcal{S}$ of objects in $\mathbb{R}^d$ is defined as a binary tree $\mathcal{T}$ with the two following properties:

1. If $card(\mathcal{S}) \leq \delta$, then $\mathcal{T}$ is a single leaf. The object(s) in $\mathcal{S}$ is (are) stored in this leaf node.

2. If $card(\mathcal{S}) > \delta$, then the space is cut by a hyperplane h_v, call the *splitter* of v, which is the root of $\mathcal{T}$. The information about h_v is stored in v. The left child of v is the root of a BSP-tree $\mathcal{T}^-$ corresponding to the negative open subspace h_v^- and stores the subset $\mathcal{S}^- \subset \mathcal{S}$ of all objects intersecting h_v^-. The right child of v is the root of a BSP-tree $\mathcal{T}^+$ corresponding to the positive open subspace h_v^+ and stores the subset $\mathcal{S}^+ \subset \mathcal{S}$ of all objects intersecting h_v^+. Objects that meet both h^- and h^+ are stored in both $\mathcal{T}^-$ and $\mathcal{T}^+$.

The *size* of a BSP-tree is the number of nodes in the tree, together with the storage required to hold the information associated with each node. In the original design, each splitting plane was aligned with a scene polygon, such a partition is sometimes called an *autopartition*. Since the orientation of the polygons is arbitrary, the splitting planes of a BSP-tree are also arbitrarily oriented. The algorithm to construct a general BSP-tree can be derived directly from its definition. We describe the general BSP-tree construction algorithm and use a simple example to illustrate it. The capacity of a node is a *threshold* condition, which is the criterion to determine whether we want the node to be split further. Assuming the threshold capacity δ is pre-determined, a BSP-tree can be constructed as follows.

Algorithm BSPConstruct($\mathcal{S}$)

Input: $\mathcal{S} = \{o_1, o_2, \cdots, o_n\}$ is the set of n objects in 3-space.

Output: A BSP-tree $\mathcal{T}$.

```
1:  if threshold condition is satisfied then
2:      Create a single-node BSP-tree $\mathcal{T}$;
3:      Store the objects of $\mathcal{S}$ in $\mathcal{T}$;
4:  else
5:      Choose $h$ as the splitting plane;
6:      $\mathcal{S}^- \leftarrow$ objects of $\mathcal{S}$ that intersect $h^-$;
7:      $\mathcal{T}^- \leftarrow$ BSPConstruct ($\mathcal{S}^-$);
8:      $\mathcal{S}^+ \leftarrow$ objects of $\mathcal{S}$ that intersect $h^+$;
9:      $\mathcal{T}^+ \leftarrow$ BSPConstruct ($\mathcal{S}^+$);
10:     $\mathcal{T} \leftarrow$ Tree($h, \mathcal{T}^-, \mathcal{T}^+$);
11: end if
12: return $\mathcal{T}$;
```

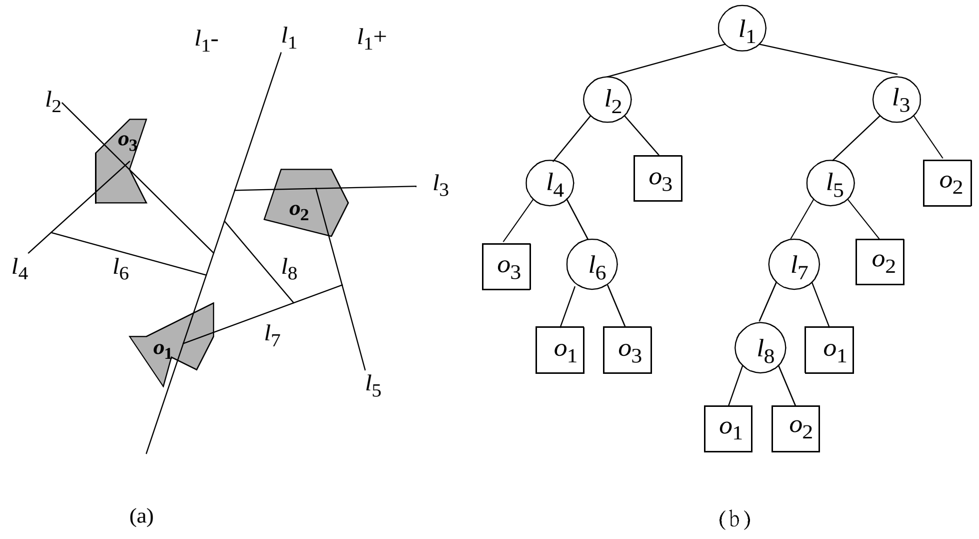


Figure 25: An example of BSP-tree in 2-dimensional space

Each region produced by algorithm BSPConstruct is a convex polyhedron. This algorithm constructs a BSP-tree with all of the objects stored in the leaf nodes. Figure 25(a) shows a scene partitioned by a BSP-tree. The original scene has three objects; the set of objects $\mathcal{S} = \{o_1, o_2, o_3\}$. Suppose at line 5 of algorithm BSPConstruct(S) picks l_1 as the first splitting line. Object o_1 is cut into two fragments. Now the left open half-plane l_1^- contains two objects: o_2 and part of object o_1. The right open half-plane l_1^+ contains object o_3 and part of object o_1. Object o_1 belongs to both of the left subset $\mathcal{S}^-$ and the right subset $\mathcal{S}^+$. So far $\mathcal{S}^- = \{o_1, o_2\}$, and $\mathcal{S}^+ = \{o_1, o_3\}$. We then call BSPConstruct($\mathcal{S}^-$) and BSPConstruct($\mathcal{S}^+$) recursively to construct the left and right subtrees. The resulting

BSP-tree is shown in Figure 25(b), if we choose δ to be one. Since an object can be cut by the splitting plane and stored in both of the subtrees, the size of BSP-tree is determined by the number of fragment of objects. Figure 25 shows a bad example of BSP-tree subdivision that produces 9 leaf nodes from 3 objects. However, it is possible to construct a three-leaf BSP-tree if we choose good splitting planes. For n non-intersecting triangles in $\mathbb{R}^3$, it has been shown that a BSP-tree (an autopartition) of size $O(n^2)$ exists [31]. A naive autopartition may even produce a BSP-tree of size $\Omega(n^3)$ [90]. In some cases, for example, $\mathcal{S}$ is a set of walls, if we view the scene from the top, each object is a line segment. We can align the splitting planes with objects, e.g. in an architectural walk through, where objects are walls. A BSP-tree constructed using this scheme may store the objects in the internal node [1].

Assume that the ray origin is always located in the negative open halfspace h^- defined by the node splitter. The ray traverses a BSP-tree $\mathcal{T}$ as follows. We start the ray shooting query from the root node of $\mathcal{T}$. If it is a leaf, we examine all of the objects stored in the node. If we find objects hit by the ray, we pick the one that is closest to the ray origin and we are done. Otherwise, we perform recursive inorder tree traversal by visiting $\mathcal{T}^-$, then (the objects stored at the root node of) $\mathcal{T}$, and then $\mathcal{T}^+$.

General BSP-trees have been used widely in many areas, for example, hidden surface removal [30,85,67], collision detection [86], point location [66], motion planning [66], ray shooting [21,8], and computer games such as DOOM and Quake [107]. Ray tracing applications often use axis-aligned BSP-trees because it enables fast ray-box intersection tests [122,84].

7.1.2 k-D trees

The k-D tree was introduced by Bentley [16] as a binary search tree for multidimensional associative searching. The symbol k in k-D tree stands for the dimensionality of the search space. k-D tree is a special case of the general BSP-tree. The difference between k-D tree and BSP-tree is the restriction on the direction for the splitting planes. For a BSP-tree, the splitting planes can have arbitrary orientations, whereas the splitting planes for a k-D tree must be axis-aligned. The "classic" k-D trees have to alternate direction of the splitting planes, e.g. in three dimensions, one splits x direction first, then y, then z, then x again and so forth. Recent applications of k-D trees do not have this restriction. This data structure has been used extensively to help solve the k-dimensional orthogonal range searching and proximity/nearest neighbor problems. An early survey of range searching was conducted by Bentley [17]. More recent surveys that deal with range searching problem for different shapes of objects can be found in [2,3]. Various approaches to construct an efficient k-D tree for ray tracing are described next.

Construction

Before we describe the k-D tree construction algorithm, let us look at an example. Figure 26(a) shows a scene with five objects in 2-space. We would like to partition the scene into regions such that within each region there is no more than a single object. We start by choosing a splitting line l_1 that is parallel to y-axis. The sub-region on the left hand side of l_1 is further divided by the line l_2 which is parallel to the x-axis. Since the sub-region below l_2 only contains part of object o_1 and nothing else, we leave that region untouched. The sub-region above l_2 contains two objects, so it is further subdivided by the vertical line l_4. For the sub-region to the right of l_1, we can apply the same method by splitting the region, alternating horizontal and vertical lines. The resulting space subdivision is shown in Figure 26(a). The k-D tree corresponding to the subdivision is shown in Figure 26(b).

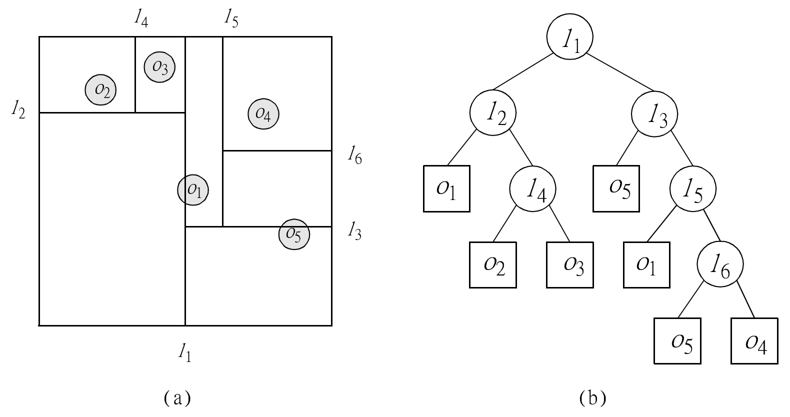


Figure 26: (a) A subdivision in 2-space. (b) The k-D tree created corresponding to the subdivision on the left.

In the previous example, we pre-select the termination threshold value to be one. If we use a more general termination condition, a more general k-D tree can be constructed by the following algorithm.

Algorithm KDConstruct(S)
Input: $\mathcal{S} = \{o_1, o_2, \cdots, o_n\}$ is the set of n objects in k-dimension.
Output: A k-D tree $\mathcal{T}$.

```
1:  if the threshold condition is satisfied then
2:      Create a single-node k-D tree $\mathcal{T}$;
3:      Store the objects of $\mathcal{S}$ in $\mathcal{T}$;
4:  else
5:      Choose a splitting plane $h_i$ that is parallel to the $i$-th axis, $1 \le i \le k$;
6:      $\mathcal{S}^- \leftarrow$ objects of $\mathcal{S}$ that meet $h_i^-$;
7:      $\mathcal{T}^- \leftarrow$ KDConstruct($\mathcal{S}^-$);
8:      $\mathcal{S}^+ \leftarrow$ objects of $\mathcal{S}$ that meet $h_i^+$;
9:      $\mathcal{T}^+ \leftarrow$ KDConstruct($\mathcal{S}^+$);
10:     $\mathcal{T} \leftarrow$ Tree($h_i, \mathcal{T}^-, \mathcal{T}^+$);
11: end if
12: return $\mathcal{T}$;
```

Line 1 of algorithm KDConstruct is the termination criterion. First, it can be a preset limit on the number of objects that may stored in a single k-D tree node. If the number of objects are equal to or below the threshold, we stop further splitting of the current node and form a single node k-D tree that stores all of the given objects. Kaplan [72] suggests using one as the threshold number of objects. We use Kaplan-BSP in the following context to refer to the k-D tree obtained using this criterion. It is the same as BSPConstruct on page 40 (with $\delta = 1$) except for the direction of the splitting planes. Subramanian and Fussell [112] also implement a k-D tree that is similar to Kaplan-BSP. The only difference is Kaplan-BSP will still split, say along x-direction, current cell into two cells even it is empty, while Subramanian and Fussell's k-D tree will skip splitting empty cell itself. One can visualize a level of octree (see section 7.2) as a three-level Kaplan-BSP. Cassen [21] implements an algorithm for constructing a k-D tree using evolutionary technique. The automatic termination criterion is based on their cost function of the evolution process. During Cassen's k-D tree construction, the cost of k-D tree is monitored. If at some point, even if the region is subdivided but the overall cost function does not decrease over a certain percentage, their algorithm concludes that it is not worthy to do any further subdivision and the entire construction process stops at that point.

The second possible threshold condition in line 1 is the height of the k-D tree which one may want to limit. Once the maximum tree height is reached, we stop dividing the regions and store all of the objects within the regions in the corresponding nodes. In Kaplan's BSP-tree construction, the maximum tree height is set to 30. As in all of the spatial subdivision methods, if the height of the hierarchy is too high, we may end up with a lot of expensive vertical movements in the hierarchy. On the contrary, if the tree height is too low, many ray-object intersection tests may have to be performed. After all, reducing the number of ray-object intersection tests is the primary goal of constructing a spatial subdivision. One can also choose the threshold value after the entire scene is given in order to optimize the structure for ray tracing.

Line 5 of algorithm KDConstruct picks an axis-aligned splitting plane h_i and separates the region into two open half-spaces h_i^- and h_i^+. The choice of the plane is another factor that affects the performance of a k-D tree. The most straightforward way is to split the scene at the spatial median, i.e. exactly halving the length, width or height of the region. This approach is implemented by Kaplan-BSP [72] and Samet's PR k-D tree [100]. The advantage of this method is we don't have to spend extra time in finding where to split during the tree construction. Another convenient way is to pick an axis-aligned splitting plane arbitrarily as suggested by Arnaldi et al. [9]. A k-D tree can be constructed easily using either Kaplan's or Arnaldi's approach. Both methods perform well if the objects are uniformly distributed. A more sophisticated way suggested by de Berg et al. [31] is to split at the object median. This way we can ensure the resulting k-D tree is better balanced even if the objects are not uniformly distributed. MacDonald and Booth [82] implement several k-D trees with different position of splitting planes to compare the performance. Their experimental results show that if we choose the splitting plane somewhere between the spatial median and the object median, we can get a better performance and spend less time in ray traversal. Subramanian's [110] and Whang's [119] experimental results confirm this point.

Suppose a near optimum splitting plane can be found for each dimension, using a specific optimality criterion. In k-dimensional space, there are k different axis-aligned splitting plane candidates. Each is the best splitting plane along an axis direction. The question is which one should we choose first? Choosing the splitting planes in different order can also affect the performance of ray tracing. One approach is to cyclically divide the space starting from the first dimension, then the second dimension, and so on. For example, in three-dimensional case, we can construct a k-D tree by choosing the splitting plane that is perpendicular to x-axis. We then divide each of the resulting subspaces by a splitting plane perpendicular to y-axis, and then the same rule is applied to the z-direction. De Berg et al. [31] provide an algorithm that uses this cyclic approach to construct a 2-D tree. The same approach is also used by Kaplan [72] to build the Kaplan-BSP. Choosing the splitting plane cycling through the axes is easy to implement and results in faster construction of the k-D tree data structure due to inexpensive determination of the splitting plane. However, several experimental results [82, 111, 110] show that there are other approaches that may save more time at the ray traversal stage.

Arnaldi et al. [9] introduce a semi-cyclic way to choose the splitting planes. Their approach consists of two steps. The first step only considers two-dimensional subdivision. This step results in cells that are long along the third axis. At the second step, the leaf nodes are further subdivided along the third dimension. One advantage of this approach is it makes the neighbor-finding task easier by focusing on the two-dimensional neighbor first and then worry about the neighbor in the third dimension later.

We can also find a better splitting plane by examining the best splitting plane along each dimension first, and then picking the best one among those candidates for each dimension. This assumes we have a way to measure "goodness" of a plane. Using this approach, we have to spend more time on finding the best of the best splitting planes at each iteration of the k-D tree construction phase. This approach behaves well even in very bad situations. Consider an extreme case in the plane shown in Figure 27(a) with $n = 6$ thin rectangles.

In this case, we will find that all the best splitting planes have the same orientation. The regions of the resulting space subdivision allow long and skinny sub-regions as shown in Figure 27(a). The corresponding k-D tree using this acyclic approach is shown in Figure 27(b); it is balanced. If we are restricted to split the region along x- and y-axis in turn, the scene may be divided by the way shown in Figure 27(c). This example shows a k-D tree subdivision with a lot more excessive splits than the acyclic version. Removing the restriction of the order for cutting is shown to be a better way to construct a more adaptive k-D tree and can dramatically improve the ray traversal speed in most cases [82, 111, 110].

Lines 6-9 of algorithm KDCONSTRUCT build the left and right subtrees for the current node. The objects in $\mathcal{S}$ are separated into two groups. Objects that do not meet h_i and fully contained within the half-space h_i^- are put into subset $\mathcal{S}^-$ at line 6. We then call KDCONSTRUCT recursively on $\mathcal{S}^-$ at line 7. The right subtree is handled similarly at lines 8-9. The objects that intersect the splitting plane h_i are traditionally stored in both $\mathcal{S}^-$ and $\mathcal{S}^+$. This approach is implemented by Kaplan [72] and Arnaldi et al. [9]. The resulting k-D tree can end up having many excessive nodes due to fragmentation of the objects. Arnaldi et al.'s trick is to use the extreme point of the chosen object as the base of the splitting plane in order to reduce the number of object fragments. This approach is illustrated in Figure

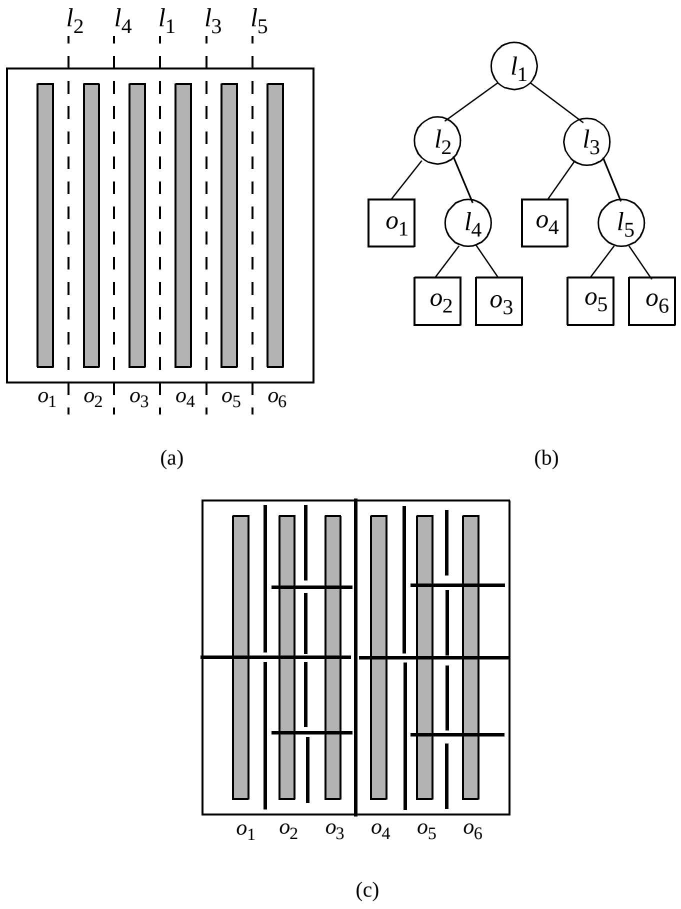

l_2 l_4 l_1 l_3 l_5
o_1 o_2 o_3 o_4 o_5 o_6
(a)
l_1
l_2 l_3
o_1 l_4 o_4 l_5
o_2 o_3 o_5 o_6
(b)
o_1 o_2 o_3 o_4 o_5 o_6
(c)

Figure 27: The worst case k-D tree structure. (a) No restriction on splitting direction. (b) The corresponding tree structure on the left. (c) Choosing the splitting plane along $x \to y \to x \to y$ order.

28.

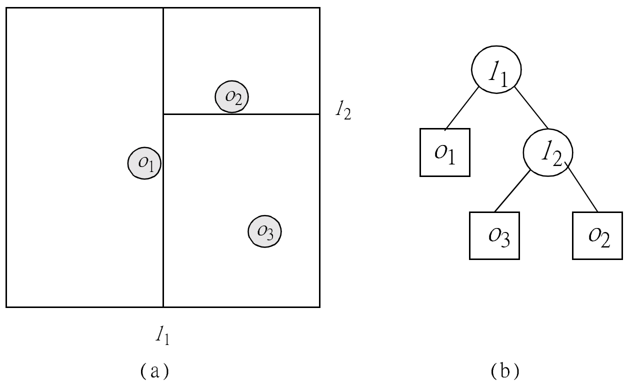


Figure 28: Arnaldi's k-D tree construction

Consider the scene with three objects o_1, o_2 and o_3 shown in Figure 28(a), Arnaldi et al.'s approach is to pick arbitrarily an object as the base of the splitting plane. Suppose object o_1 is chosen. The space can be divided by a plane that passes through the rightmost point of o_1. We can put object o_1 into the left subtree without cutting into two pieces. This way we can save some memory space by reducing the number of object fragments. The k-D tree constructed by Arnaldi's approach is shown in Figure 28(b). Another way to reduce the number of excessive object fragments that was suggested by Bentley [16], Samet [100] and de Berg et al. [31] is just simply to store the objects that intersect the splitting plane in the right subtree. To test the intersection between a ray and these objects, the right subtree has to be checked. In this case, ray traversal is very different. The k-D tree is no longer a space partition because the first object hit by the ray is not always the first one we find. Although this approach was originally used for multidimensional range search problem, it provides easy mechanisms to reduce the size of k-D trees.

The construction of the k-D tree often result in an unbalanced tree. Friedman et al. [43] proposed an *adaptive k-D tree* to overcome this problem. However, both the original and improved version of k-D trees are only suitable for handling data sets that reside in the main memory. To account for the external memory issue, Robinson [99] suggests using k-D-B-tree, a hybrid tree that combines both Friedman's adaptive k-D tree [43] and Comer's B-tree [28], to overcome this weakness.

Ray Traversal

The basic steps of traversing a k-D tree are as follows.

1. Find the leaf node at which the ray origin is located.
2. Test for ray-object intersections within the leaf node. If the ray hits any object, report the first object hit by the ray and stop.

3. Find the next neighbor of current node, i.e., the leaf node of the tree entered by the ray after it leaves the current node.
4. Repeat step 2-3 until the ray hits an object or out of scope.

Performance of step 1 is determined by the height of the k-D tree. For a balanced k-D tree with ℓ leaves, this step can be done in time $O(\log \ell)$. Performance of the second step is determined by the ray-object intersection algorithm. A thorough survey of efficient ray-surface intersection algorithms can be found in Hanrahan's article [60]. Without lost of generality, we can assume the intersection test can be performed in $O(1)$ amortized time per object – this assumes that on average an object is not very complicated. The most important factor that affects the performance of a ray traversal algorithm is step 3, where we need to advance the ray from one region to another.

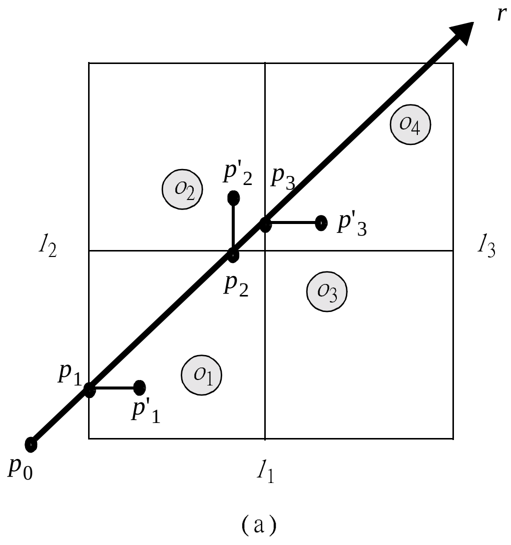
The traditional way to traverse a ray through a k-D tree was introduced by Kaplan [72] which utilizes the spatial coherence property of a ray. We assume there is a simple function RayExtend(r, p, v). This function takes three parameters. The first parameter is the ray r. The second parameter p is the intersection point of the ray and (the axis-oriented box corresponding to) the current node. The third parameter is the node v representing the bounding box. RayExtend pushes p a small amount away from the ray origin and perpendicular to the face of the bounding box that contains p. The resulting artificial point p' is used to determine which leaf node needs to be examined next. Kaplan's algorithm works as follows.

Algorithm KaplanKDTraverse$(\mathcal{T}, r)$
Input: A k-D tree $\mathcal{T}$ and a ray r.
Output: The first object o hit by the ray, or $NULL$ if the ray does not hit any object.

```
1: $o \leftarrow NULL$;
2: $v \leftarrow$ root node of $\mathcal{T}$, representing the outermost bounding box;
3: $p \leftarrow$ entry point of the ray to the root box, or ray origin if it is inside the root box;
4: $p' \leftarrow$ RayExtend$(r, p, v)$, or $p$ if $p$ is ray origin;
5: repeat
6:    $v \leftarrow$ root node of $\mathcal{T}$;
7:    while $v$ is not a leaf node do
8:       if ( $p' \in l_v^-$ ) then
9:          $v \leftarrow$ left child of $v$;
10:      else
11:         $v \leftarrow$ right child of $v$;
12:      end if
13:   end while
14:   $o \leftarrow$ TestIntersect$(r, v)$;
15:   if $(o \neq NULL)$ then
16:      return $o$;
17:   end if
18:   $p \leftarrow$ exit point of current node;
```

```
19:     $p' \leftarrow$ RayExtend$(r, p, v)$
20: **until** $(o \neq NULL$ or $p'$ is out of scope)
21: **return** $o$;
```

The exit point in line 18 is determined by testing the intersection point between the ray and the six faces of the box corresponding to the current node. Once the exit point is found, the function call to RayExtend in lines 4 and 19 creates an artificial point p' by pushing it a small amount from point p into the next region and perpendicular to the face hit by the ray. The distance between p and p' has to be small enough so that we can guarantee p' is within the next region. Once the coordinates of p' are determined, lines 7-13 perform a top-down search to find the leaf node where the point p' is located. A k-D tree traversed by Kaplan's method is illustrated in Figure 29.



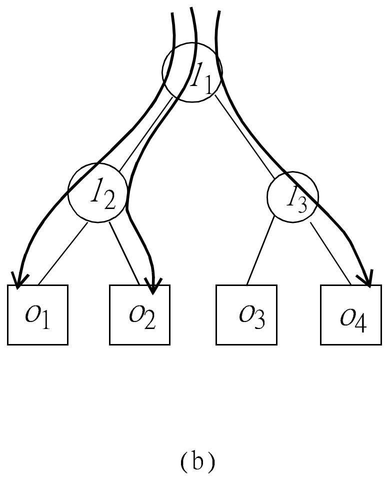


Figure 29: Example of Kaplan's ray traversal method

In Figure 29(a), a ray r enters the scene at point p_1. Point p'_1 is obtained by pushing p_1 as described above. We then search for the region that contains the point p'_1 from the root node as shown in Figure 29(b). To find the next neighbor along the ray path, point p_2 is calculated and pushed to artificial point p'_2. The same step is repeated until the ray goes out of scope. In this example, three out of four regions are examined.

Function TestIntersect(r, v) at line 14 of algorithm KaplanKDTraverse performs ray-object intersection tests on all of the objects stored in the leaf node v. It returns the first object that is hit by the ray or *NULL* if none is.

Another k-D tree traversal algorithm using the *ray clipping* trick is proposed in Subramanian's Ph.D. thesis [110]. During the ray traversal stage, a ray is "clipped" into several line segments when it passes through the regions. Subramanian's k-D tree traversal is essentially a depth-first walk over a binary search tree. We first look at the outline of his algorithm and then examine each step.

Algorithm RCKDTraverse$(\mathcal{T}, r)$

Input: A k-D tree $\mathcal{T}$ rooted at v, a ray r.
Output: First object o that is hit by the ray, or $NULL$ if the ray does not hit any object.

```
 1: $o \leftarrow NULL$;
 2: if ($v$ is a leaf node) then
 3:    $o \leftarrow$ TestIntersect($r$, $v$);
 4:    if ($o \neq NULL$) then
 5:       return $o$;
 6:    end if
 7:    return NULL; {No intersection was found.}
 8: else {$v$ is an internal node}
 9:    $p \leftarrow$ the intersection point of ray $r$ and the splitting plane corresponding to node $v$;
10:    $p_1 \leftarrow$ the entry point of $r$ corresponding to the bounding box of node $v$, or the origin
       of $r$ if it starts inside the region.;
11:    $p_2 \leftarrow$ the exit point of $r$ corresponding to the bounding box of node $v$;
12:    if ($p_1 < p$ and $p_2 < p$) then
13:       $o \leftarrow$ RCKDTraverse($\mathcal{T}^-, r$); {Case 1}
14:    else if ($p_1 \geq p$ and $p_2 \geq p$) then
15:       $o \leftarrow$ RCKDTraverse($\mathcal{T}^+, r$); {Case 2}
16:    else if ($p_1 < p < p_2$) then
17:       $o \leftarrow$ RCKDTraverse($\mathcal{T}^-, r$); {Case 3}
18:       if ($o == NULL$) then
19:          $o \leftarrow$ RCKDTraverse($\mathcal{T}^+, r$);
20:       end if
21:    else if ($p_1 > p \geq p_2$) then
22:       $o \leftarrow$ RCKDTraverse($\mathcal{T}^+, r$); {Case 4}
23:       if ($o == NULL$) then
24:          $o \leftarrow$ RCKDTraverse($\mathcal{T}^-, r$);
25:       end if
26:    end if
27: end if
28: return $o$;
```

The first two letters of algorithm RCKDTraverse stand for the abbreviation of "ray clipping". The algorithm starts by examining the current node at lines 2-9. If it is a leaf node, we perform intersection test on all of the objects that are stored in the node. If there are any objects hit by the ray, the first one is reported. If the current node is an internal node, we perform lines 10-29. The relationship between the ray and the bounding box associated with the splitting plane can be classified into four categories. The first two cases happen when the ray penetrates only one of the two subboxes that is divided by the splitting plane. The last two cases take care of the situation when the ray passes across the splitting plane, in either direction.

Figure 30 shows four types of rays passing through a box in k-D tree. The thick lines r_1, r_2, r_3 and r_4 represent the rays. The vertical dotted line ℓ is the splitting plane that cuts the box into left and right subboxes. For each ray r_i, $i = 1, 2, 3, 4$, point p_1 is the entry point of

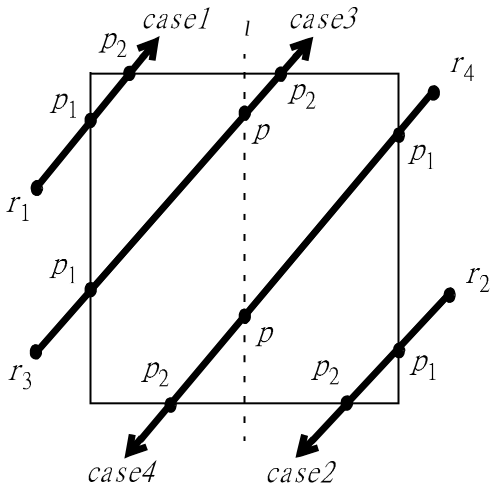


Figure 30: Four possible ways for a ray to pass through a k-D tree node.

the box; the case where p_1 is the origin of r is entirely analogous. Point p_2 is the exit point. The intersection point of a ray r_i and the splitting plane ℓ is p.

Line 13 of RCKDTraverse takes care of case 1, where only the left subtree $\mathcal{T}^-$ will be traversed. Line 15 is the opposite of case 1. Only the right subtree $\mathcal{T}^+$ will be traversed at case 2. Lines 17-20 deal with Case 3, the ray visits left subtree of the current node first. If there is no intersection detected, then go down to the right subtree. The case of the ray entering the box from the opposite side is taken care of by lines 22-25.

Arvo [11] also proposes a nearly identical ray clipping algorithm. The difference is the underlying k-D tree structure. Arvo applies the ray clipping algorithm to the structure that is the same as Kaplan-BSP, while Subramanian deals with the splitting planes independently. If we apply Arvo's algorithm and Subramanian's algorithm on the same structure, the ray traversal paths are exactly the same.

Consider the following example shown in Figure 31(a). A ray r originating at point p_0 passes through the scene with four objects o_i, $1 \leq i \leq 4$. The path of ray traversal on the corresponding k-D tree is shown in Figure 31(b). The search for intersection starts from the root node represented by l_1. Since $p_1 < p_3 < p_5$ (case 3), we first visit the left subtree of node l_1. The next step is to examine the subbox represented by node l_2. Since $p_1 < p_2 < p_3$, we go down to the left child of l_2 and reach the leaf node o_1 where the ray-object intersection test is performed. The ray does not hit object o_1, RCKDTraverse returns $NULL$ to its parent which then visits the right child of l_2.

The process continues until we reach point p_5, where the ray goes out of the scope. In the example given here, the thick line represents the search path for each algorithm. Ray-object intersection tests are performed on all of the objects. None of them are hit by the ray. If we compare Figure 31(b) and Figure 29(b), we can notice the difference between these two methods; the search path of KaplanKDTraverse always starts from the root while RCKDTraverse search for the next node starting from the current node.

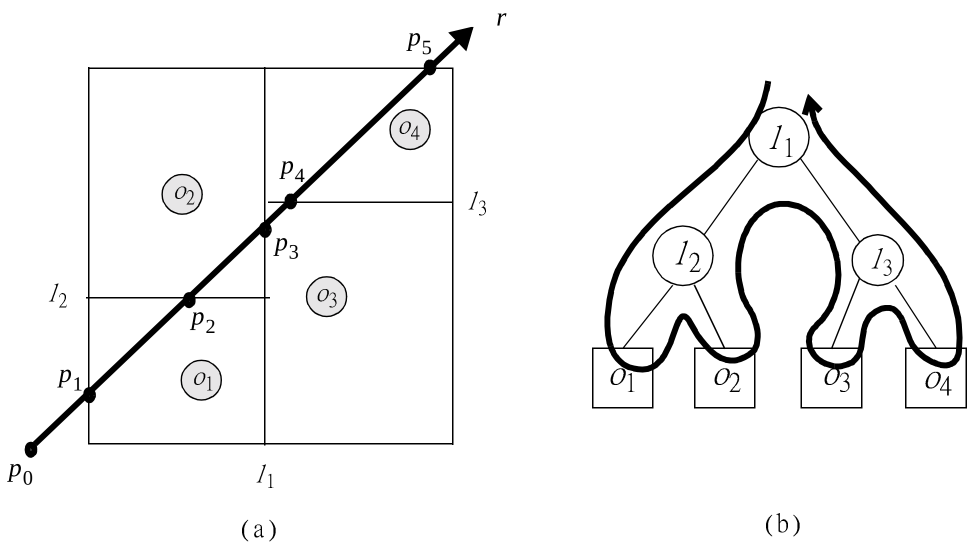


Figure 31: Example of a Subramanian's ray traversal method. (a) Ray traversal on a k-D tree subdivision. (b) The corresponding tree representation and traversal path on the left.

Arnaldi et al. [9] use a *corner stitching* technique to assist their ray traversal algorithm. The method was originally used to represent 2D VLSI layout. For each k-D tree cell, they use a fixed set of pointers associated with the corners of each cell. These pointers are used to link the neighbors together through their corners. Havran et al. [61] present a *rope tree* as an alternative of corner-stitched tree. The function of a rope is similar to a corner stitch. The difference is Havran et al.'s neighbor node does not have to be a leaf while a corner stitch always points to a leaf node. Arnaldi's ray traversal algorithm works as follows.

Algorithm CSKDTraverse($\mathcal{T}, r$)

Input: A k-D tree $\mathcal{T}$ rooted at v, a ray r originated at point p.

Output: The first object o hit by the ray, or $NULL$ if the ray does not hit any object.

```
1: $o \leftarrow NULL$;
2: $v \leftarrow$ root node of $\mathcal{T}$;
3: $p \leftarrow$ entry point of ray $r$ into $v$, or the origin of $r$ if it starts inside $v$;
4: while $v$ is not a leaf node do
5:    if ( $p \in l_v^-$ ) then
6:       $v \leftarrow$ left child of $v$;
7:    else
8:       $v \leftarrow$ right child of $v$;
9:    end if
10: end while
11: repeat
12:    $o \leftarrow$ TestIntersect($r$, $v$);
13:    if ($o \neq NULL$) then
14:       return $o$;
15:    end if
```

```
16:     Find the face through which the ray exits;
17:     $v \leftarrow$ use corner stitch pointers to go to the neighbor;
18: until ($o \neq NULL$ or $v = NULL$)
19: return $o$;
```

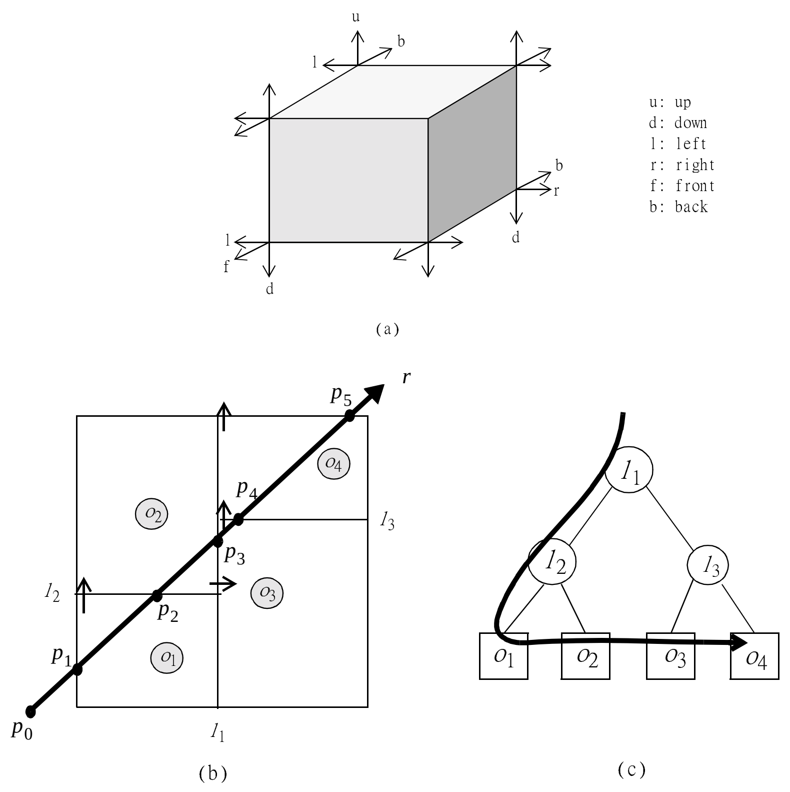


Figure 32: (a) The corner stitches associated with a node. (b) A simple planar subdivision with 4 objects. (c) The k-D tree corresponding to this space subdivision.

Following our naming convention, the first two letters of algorithm CSKDTRAVERSE stand for the abbreviation of "corner stitch". Lines 4-9 search the k-D tree from the root node to the leaf. Lines 12-17 perform ray-object intersection test within the region represented by the current node. Line 18 advances the current node to the next neighbor according to the exit point of the ray. The corner stitches associated with a single node are shown in Figure 32(a). To illustrate how the method works, we use the same 4-object example as before so that we can easily distinguish the difference between different ray traversal algorithms. In Figure 32(b), the small arrows at the corners of the regions indicate the active pointers that match with the path of a given ray. For example, if the ray enters the region from the left and exits from the top, we follow the upper-left pointer to the next region.

Figure 32(c) shows the k-D tree structure along with the ray traversal path. As usual, we start from the root node, search through the subtree until we reach the leaf node o_1, which is the location of ray origin or entry point to the scene. Ray-object intersection test is performed on all of the objects stored in the current node. In this example, the ray does not hit any object. So we need to find the next neighbor node to be examined. Since the ray exits from the top of the region represented by current node, we use the top pointer that is closer to the entry point to find the next neighbor o_2. The ray then exits from the right side of node o_2. We use the right pointer to identify node o_3.

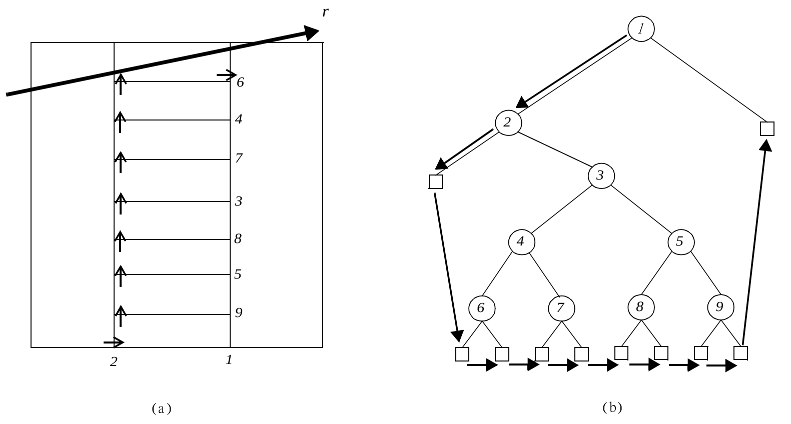


Figure 33: An example where corner stitch method does not work well.

As we can see in Figure 32(c), Arnaldi's ray traversal algorithm only walks through the leaf nodes except for the initial search from the root. With the help of corner stitches, the vertical movement in the k-D tree can eliminated. However, we were lucky in this case in that following a corner stitch always led us to the right leaf node. Sometimes it may require more time to determine which link to follow. Figure 33 illustrates an extreme situation. Suppose ray r enters the scene that is partitioned as indicated in Figure 33(a). The region is first divided by line 1, then line 2, and so on. The ray only pierces three regions in the scene. It is trivial to find the first leaf node penetrated by the ray. The problem arises when we want to calculate the next region to visit. Traversing this particular scene using corner stitch ends up visiting all of the leaf nodes shown in Figure 33(b). One can argue that this example is highly degenerate and does not represent the typical situations. Arnaldi implements the algorithm with corner stitches and mailboxes (see section 2). The experimental result shows this approach is up to 24.55 times faster than without mailboxes and without corner stitches. Nearly 80% of redundant intersection tests can be avoided by using mailboxes.

7.2 Eight-Way Subdivisions – Octrees

In this section, we define an octree more formally. An octree construction algorithm can then be derived from the formal definition directly. Given a set $\mathcal{S}$ of objects in 3-space, the corresponding octree subdivision can be defined recursively as follows. Let σ be an axis-aligned box that encloses the set $\mathcal{S}$, $\sigma := [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma] \times [z_\sigma, z'_\sigma]$.

1. If $card(S) \leq m$, where m is a pre-selected constant, then the octree consists of a single leaf node storing all of the objects in set $\mathcal{S}$. Other termination criteria are also possible.

2. If $card(S) > m$, let σ_{LUF}, σ_{LUB}, σ_{LDF}, σ_{LDB}, σ_{RUF}, σ_{RUB} , σ_{RDF}, and σ_{RDB} denote the eight octants of σ, where the subscript symbols distinguish between left (L) and right (R), up (U) and down (D), front (F) and back (B) octants. Let $x_\sigma \leq x_{med} \leq x'_\sigma$, $y_\sigma \leq y_{med} \leq y'_\sigma$, $z_\sigma \leq z_{med} \leq z'_\sigma$. $(x_{med}, y_{med}, z_{med})$ is a point inside σ. The sets of objects and the bounding box of the 8 children are $S_{ijk} = \{\ s \in S \mid s \text{ intersects } \sigma_{ijk}\}$, for $i \in \{L, R\}$, $j \in \{U, D\}$, $k \in \{F, B\}$, and $\sigma_{ijk} := X_i \times Y_j \times Z_k$, where $X_L = [x_\sigma, x_{med}]$, $X_R = [x_{med}, x'_\sigma]$, $Y_U = [y_\sigma, y_{med}]$, $Y_D = [y_{med}, y'_\sigma]$, $Z_F = [z_\sigma, z_{med}]$ and $Z_B = [z_{med}, z'_\sigma]$. In this case, the tree is comprised of an internal node with 8 children, each of which is an octree with root bounding box σ_{ijk} for the set of objects S_{ijk}.

Figure 34 is an example of a two-level octree, each octant is named as described in the above definition. Portions of objects are stored in the leaf nodes. The *leaf node* in an octree has many names. It is also known as *obel* [42], *prism* [50], *voxel* [119], *cell* [97], *cube* [10], or *octree box* [25]. We use the name "cell" and "box" for the leaf node interchangeably, whichever is more appropriate in the context. A traditional octree only stores the objects in the leaf nodes [49], but objects can also be stored in both internal and external nodes [46]. If an object intersects more than one node, pieces of the object are stored in each of them.

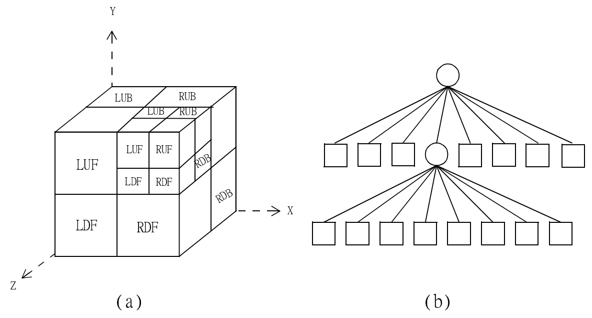


Figure 34: Octree Illustration

7.2.1 Construction of an Octree

An algorithm for constructing an octree can be derived directly from its recursive definition as follows.

Algorithm OctreeConstruct($\mathcal{S}$, B)

Input: A set $\mathcal{S}$ of objects in 3-space and the bounding box B that intersects all of the objects in $\mathcal{S}$.
Output: Octree rooted at $\mathcal{T}$

```
1: if The threshold condition is satisfied then
2:     Create a single-node octree $\mathcal{T}$;
3:     Store the objects of $\mathcal{S}$ in $\mathcal{T}$;
4: else
5:     Choose three splitting planes $h_x, h_y, h_z$ orthogonal to $x$-, $y$-, and $z$-axis, respectively;
6:     Partition $B$ into eight octants $\sigma_{ijk}$ using $h_x, h_y$, and $h_z$;
7:     for each $(\sigma_{ijk})$ do
8:         $\mathcal{S}_{ijk} \leftarrow$ subset of $\mathcal{S}$ that intersect with $\sigma_{ijk}$;
9:         $\mathcal{T}_{ijk} \leftarrow$ OctreeConstruct$(\mathcal{S}_{ijk}, \sigma_{ijk})$;
10:        $\mathcal{T} \leftarrow \mathcal{T}$.AddChild$(\mathcal{T}_{ijk})$;
11:    end for
12: end if
13: return $\mathcal{T}$;
```

The threshold condition at line 1 determines when the algorithm OctreeConstruct should stop. According to the definition, the bounding box of set $\mathcal{S}$ is recursively subdivided until each subbox contains at most m objects. In practice, the threshold can be modified such that the recursion stops when other criteria are satisfied, e.g. when the box size becomes small enough [10], or when the octree reaches the preset maximum depth [103, 81]. Once the threshold is reached, lines 2-3 construct a single node octree satisfying property 1 of the octree definition.

Lines 5-11 construct octree satisfying property 2 of the definition. At line 5, we choose three splitting planes orthogonal to the x-, y- and z-axes. In a traditional octree, each node represents a cube in 3-space. A non-leaf node is split at the *spatial median* p which is the central point of the current node [49, 103, 92, 101, 81, 108, 10, 98]. The resulting eight octants are equal-sized cubes. This approach has the advantage of easy implementation and fast construction. It also assumes the objects are uniformly distributed. If the objects are distributed unevenly, splitting at the *object median* is more efficient for ray traversal. If we split the octree box at object median, the children of current octree node may no longer represent cuboidal subspaces. In this case, each child represent an axis-aligned box. We call each octree node a *cell* to include both cube and axis-aligned box. Splitting at the object median creates a balanced octree even when the objects are not distributed uniformly (assuming few objects are met by the splitting plane). Each octant stores an equal number of objects that intersect it. Objects that meet the partitioning planes can be stored in all of the octants that intersect the objects. A balanced octree improves the worst-case performance compared to an unbalanced one. However, during the octree construction, we have to spend more time on searching for the object median at each iteration.

MacDonald and Booth [82] point out that the best splitting plane that can minimize the ray traversal time is located somewhere between the space median and the object median. Based on MacDonald and Booth's heuristic observation, Whang et al. [119] introduce a greedy approach of constructing an octree in order to find the splitting plane that can minimize their cost function. Namely, instead of choosing the best splitting plane along each axis direction, several candidate splitting planes are chosen. The final splitting plane is obtained by picking the candidate that minimizes the cost function. The same process proceeds for each axis direction at each iteration.

7.2.2 Ray Traversal in Octrees

There are two ways to traverse an octree: non-recursive and recursive. For the non-recursive approach, we can traverse the octree in three different ways: top-down vertical traversal, horizontal traversal, and bottom-up traversal. For recursive approach, we usually use top-down recursive methods. We will describe these methods in this section. Since octree can be viewed as a special case of a k-D tree, the basic ray traversal steps for an octree are similar to that of k-D tree as we described in Section 7.1.2. Except this time, the branching factor of a node is eight instead of two.

Non-recursive Octree Traversal

The first octree traversal algorithm applied to ray tracing was introduced by Glassner [49]. It is very similar to the algorithm KaplanKDTraverse (Section 7.1.2). The ray starting point in a leaf node can be located by starting at the root and then descending all the way down to a leaf. Once the starting point is found, we can advance the current ray position to the next cell using a technique similar as KaplanKDTraverse. Each iteration only involves vertical movements from root towards to the leaf. There are two differences between Glassner's vertical ray traversal algorithm and KaplanKDTraverse. First, lines 8-11 of KaplanKDTraverse is replaced by $v \leftarrow$ FindOctant(v, p'). This function compares the position of the "pseudo point" p' with the three splitting planes h_x, h_y and h_z in order to find which octant contains p'. To illustrate how FindOctant works, we let the octant containing p' be denoted by σ_{ijk}. If p' is on the left of h_x, $i = L$, else $i = R$. If p' is above h_y, $j = U$, else $j = D$. If p' is in the front of h_z, $k = F$, else $k = B$. Thus we know the octant containing p' is, say, σ_{LUF} and thus move from v to its child corresponds to σ_{LUF}. The same process continues until a leaf node is found. This leaf node is then used for finding the next cell visited by the ray.

The two RayExtend functions in KaplanKDTraverse are used to find the next cell. Since the number of neighbors for a k-D tree and an octree is different, the underlying operations are different, although they share the same interface. For the octree, we need to examine the six faces of the current cell [49]. Once the exit point is determined, the pseudo-point guaranteed to be within the next cell can be found by utilizing the space coherence property. Figure 35 shows three different situations. If the ray exits the current node from one of its six faces (Figure 35(a)), the pseudo-point can be constructed by extending a small distance from the exit point, orthogonally to the exit face. If the ray exits from one of the 12 edges, the same process has to be done twice so that the pseudo-point is shifted away from both faces that share this edge (Figure 35(b)). If the ray exits from one of the eight vertices of the current cell, we repeat the same process as in (a) three times.

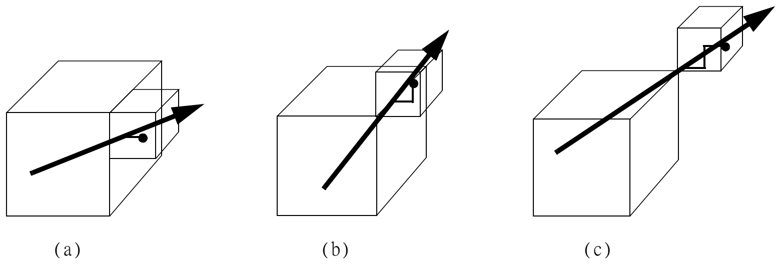


Figure 35: Three possible cases of octree pseudo point depend on the exit point of the ray and the current octree cell: (a) exit from a face, (b) exit from an edge, (c) exit from a corner.

When the octree is extremely unbalanced, this vertical traversal approach becomes inefficient. To overcome this problem, Peng et al. [92] introduce a *linear octree*. The octree traversal is only performed on the leaf nodes. The search for the next cell hit by the ray only involves horizontal movements among the leaves. Using the fact that the octree cannot be too deep, the external octree nodes are represented by a limited length sequence of octal integers and stored in a one-dimensional array. To find the leaf node containing a given point, we turn the coordinates of the point into an octal number and perform binary search on the array. The cell containing this point can be located in $O(\log \ell)$ time in worst case, if there are l leaves. Once we know which cell contains the point in which we are interested, ray-object intersection tests are performed on all of the objects that intersect this cell. If there is an intersection, we are done. Otherwise, we have to move on to the next cell hit by the ray. As in all other approaches, first we need to find the exit point of the current cell. Unlike Glassner's approach [49] that needs to test all of the six faces of the current cell, ray coherence property is used by Peng et al. [92] to reduce the number of tests. The idea is, if the ray goes upward, it cannot hit the face at the bottom. If the ray goes towards the right, it cannot hit the face on the left, and so on. Therefore, if we take the direction of the ray into account, only three faces need to be examined in order to find the exit point. Once we have the exit point, the array is searched again to find the cell that contains this point.

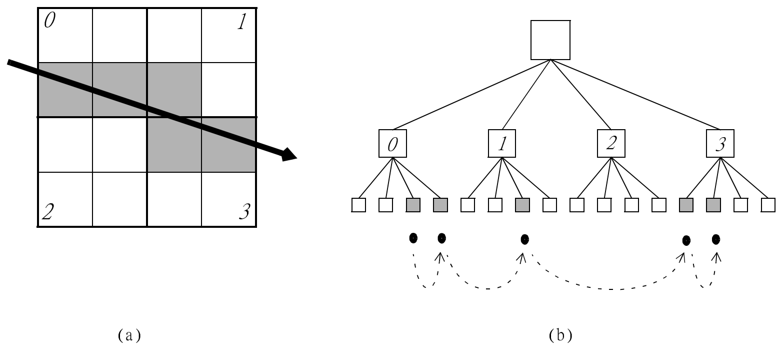


Figure 36: octree peng

Figure 36(a) shows a 2D example of a ray passing through a quadtree subdivision (the planar analogue of an octree). The corresponding tree structure is drawn in Figure 36(b). The dotted lines under the leaf nodes represent the sequence of the nodes visited by the ray that need to perform ray-object intersection test. Using the approach introduced by Peng et al. [92], each dotted line between two leaf nodes takes $O(\log l)$ time because a binary search on the array has to be performed in order to move the ray from one cell to another.

To eliminate the $O(\log l)$ factor spent on finding the next cell, Sandor [103] introduces a more sophisticated approach that searches for the next cell from bottom-up. This approach requires more calculation of the next cell than other methods. Sandor's ray traversal approach performs three basic steps. First, it uses the point location method to find the first leaf node that contains the entry point. Second, it finds the exit point of the current cell and locates the next cell on the ray path. In order to do this we need to ascend from the current node to find the octree node that is entered next by the ray, and whose size is at least the size of the cell we started with. The third step is to descend the octree to the leaf node as in Glassner's approach [49] except that we don't have to start from the root.

An improvement based on Sandor's bottom-up approach is introduced by Samet [101, 100, 102]. Instead of testing all six faces of the current cell for an exit point as proposed by Sandor, Samet only tests three faces by taking the ray direction into account. Samet's bottom-up ray traversal algorithm proceeds as follows.

```
Algorithm BUOctreeTraverse($\mathcal{T}, r$)
Input: An octree $\mathcal{T}$ rooted at $v$, a ray $r$.
Output: The first object $o$ hit by the ray, or $NULL$ if the ray does not hit any object.
 1: $o \leftarrow NULL$;
 2: $v \leftarrow$ root node of $\mathcal{T}$, representing the outermost bounding box;
 3: $p \leftarrow$ entry point of $r$ to the outermost bounding box, or the origin of $v$;
 4: $p' \leftarrow$ RayExtend($r, p, v$) or $p$ if it is the origin of $v$;
 5: while $v$ is not a leaf node do
 6:    $v \leftarrow$ FindOctant($v$, $p'$);
 7: end while
 8: $o \leftarrow$ TestIntersect($r$, $v$);
 9: if ($o \neq NULL$) then
10:    return $o$;
11: end if
12: repeat
13:    $p \leftarrow$ exit point of current node;
14:    $p' \leftarrow$ RayExtend($r, p, v$)
15:    if ($p'$ is out of scope) then
16:       return $NULL$;
17:    end if
18:    $v \leftarrow$ the node of $\mathcal{T}$ adjacent to $v$, containing $p'$, and having size greater than or equal
       to the size of $v$ (see page 60 for explanation);
19:    while $v$ is not a leaf node do
20:       $v \leftarrow$ FindOctant($v$, $p'$);
21:    end while
22:    $o \leftarrow$ TestIntersect($r$, $v$);
23: until ($o \neq NULL$)
```

```
24: return $o$;
```

Lines 1-4 in BUOctreeTraverse initialize the global variables. Lines 5-11 locate the first leaf node pierced by the ray. The main loop, from line 12 to line 24, repeats the bottom-up steps until it finds an object hit by the ray or the ray goes out of scope. The real work is done in line 18. The goal is to find the node containing the pseudo-point p', given that it has greater or equal size than the current node. Samet uses four intricate tables to encode the octants such that the task in line 18 is mainly table look-up. Each table has a corresponding table look-up function which serves as a function to return the desired information. Before we explain how these functions work, we need to know which neighbor we are looking for.

As explained in Figure 35, a ray can exit the current node in three different ways, i.e., through a face, edge, or vertex. If the ray exits from the left face, then we are looking for the L-neighbor. Similarly, if the ray exits from the right face, we look for R-neighbor. An octree node can have six *face neighbors*. They are denoted by L-, R-, U-, D-, F-, and B-neighbors. If the ray exits from the edge lying at the intersection of the left face and up face, we call that neighbor in that direction an LU-neighbor. Similar notations can be applied to the 12 edge neighbors. Finally, if the ray exits from the vertex located at the left-upper-front corner of the current node, the neighbor we are looking for is the LUF-neighbor. Same rule is used to encode the 8 vertex neighbors. The notation of face, edge, and vertex neighbors of an octree node is illustrated in Figure 37. He call that in this context a neighbor of a cell is the (possibly interior) node of the tree that lies on the correct side of the cell and is not smaller than it.

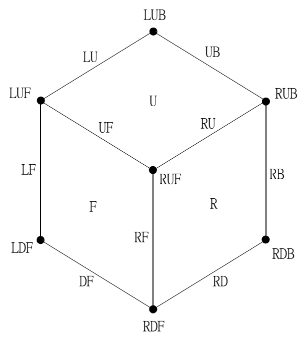


Figure 37: An octree cell has 26 neighbors (6 face-neighbors, 12 edge-neighbors, and 8 vertex-neighbors). If the ray exits from the U-face, we look for the U-neighbor. If the ray exits from the LF-edge, we look for the LF-neighbor. If the ray exits from the LUF-vertex, we look for the LUF-neighbor, and so on.

With this notation, the functions are defined below, followed by the corresponding tables. Here, symbol I represents the neighbor type, and symbol O represents the octant type of the current node (recall that an octant is one of the eight subboxes of its parent defined in page 11), so the node has octant type LUB, for example, if it is the LUB child of its parent.

1. Adj(I, O) returns true iff octant O is adjacent to its *parent's* I-neighbor, i.e., O is adjacent to the I^{th} face, edge, or vertex of its containing box. For example, Adj(L,

LUF) = *true*, ADJ(LD, LUF) = *false*, and ADJ(LDB, LUF) = *false* according to table 1.

2. REFLECT(I, O) returns the octant type of I-neighbor for current node O. For example, REFLECT(LU, LUF) is RDF according to table 2. It means if the current node is an LUF octant, its LU-neighbor is a RDF octant.

3. COMMONFACE(I, O) returns the face of O's containing box that shares with O's I-neighbor. From Table 3, COMMONFACE(LU, LDF) = L means if current node O is an LDF octant, then O's LU-neighbor shares the L-face of O's parent. COMMONFACE(LU, LUF) = NIL means if current node is an LUF octant, then O's parent does not share any common face with O's LU-neighbor.

4. COMMONEDGE(I, O) returns the edge of O's containing box that shares with O's I-neighbor. For example, COMMONEDGE(LUB, LUF) = LU, as shown in Table 4.

I(neighbor)	O(octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
L	T	T	T	T	F	F	F	F
R	F	F	F	F	T	T	T	T
...	...	...	...	...	...	...	...	...
RU	F	F	F	F	F	F	T	T
...	...	...	...	...	...	...	...	...
LDB	T	F	F	F	F	F	F	F
...	...	...	...	...	...	...	...	...

Table 1: Part of ADJ(I, O) table from Samet [102]

I(neighbor)	O(octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
R	RDB	RDF	RUB	RUF	LDB	LDF	LUB	LUF
...	...	...	...	...	...	...	...	...
RU	RUB	RUF	RDB	RDF	LUB	LUF	LDB	LDF
...	...	...	...	...	...	...	...	...
LUB	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
...	...	...	...	...	...	...	...	...

Table 2: Part of REFLECT(I, O) table from Samet [102]

Line 18 in BUOCTREETRAVERSE performs two tasks. The first task is to locate the nearest common ancestor of the current node and its neighbor containing p'. This step ascends the octree and stops at the first node such that ADJ(I, O) is false. In addition, we need to check whether the parent of current node shares the common face, edge, or vertex with the desired neighbor, using functions COMMONFACE(I, O) and COMMONEDGE(I, O).

I(neighbor)	O(octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
LU	L	L	NIL	NIL	NIL	NIL	NIL	U
...	...	...	...	...	...	...	...	...
LUB	NIL	L	NIL	NIL	B	NIL	NIL	U
...	...	...	...	...	...	...	...	...

Table 3: Part of COMMONFACE(I, O) table from Samet [102]

I(neighbor)	O(octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
LUB	LB	NIL	NIL	LU	NIL	NIL	UB	NIL
...	...	...	...	...	...	...	...	...

Table 4: Part of COMMONEDGE(I, O) table from Samet [102]

Since most of the work is done in this ascending step, that's why we classify this method as a bottom-up approach. The second task is relatively easy; it just retraces the path from the previous step, moving down the tree now, and makes mirror image moves using REFLECT(I, O) function. The table look-up step is quite complicated and requires more elaboration. Figure 38 shows an octree subdivision. Suppose we are only interested in a segment of ray that goes from octant A, passes through its *RU*-neighbor B, and reaches B's *R*-neighbor C.

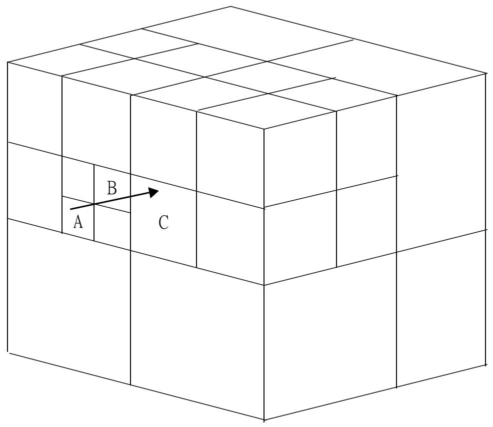
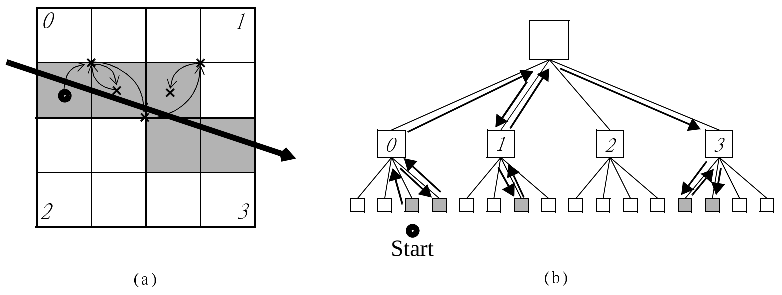


Figure 38: Example of Samet's table look-up

The first step is to locate the nearest common ancestor of octant A and B. Since A is an LDF octant and B is its *RU*-neighbor, predicate ADJ(RU, LDF) = *false* because A is not adjacent to its parent's *RU*-edge, i.e., the parent of A is the nearest common ancestor of A and B. We stop ascending the tree and make mirror image move by using function REFLECT(RU, LDF) = RUF, which is the octant type of B. We have succeeded in finding the *RU*-neighbor of *A*. The ray then goes from B to its *R*-neighbor C. As before, we ascend the tree by using ADJ(I, O) table. Since ADJ(R, RUF) = *true*, we continue ascending from B's parent, an RDF octant. ADJ(R, RDF) is *true* again. We go on to its parent which is an LUF octant. Since ADJ(R, LUF) is *false*, we stop ascending and retrace the ascending path. Because we ascend twice this time, we will have to look-up REFLECT(I, O) twice to retrace the path. We use REFLECT(R, LUF) = RUF to get to the parent of C, and then REFLECT(R, RDF) = LDF to get to octant C which is an LDF octant of its parent.

Levoy [81] also introduces an alternative bottom-up approach to traverse the octree. The difference between Samet's approach and Levoy's approach is the latter does not use table look-up for neighbor finding, but adds extra pointers that link the siblings together. To locate the next neighbor, we first advance the point along the ray to the next cell on the same level by following the sibling link. The bottom-up step is performed only if the parent of the new cell is different from the parent of the old cell, or the current cell has no sibling in that direction. We can save some time to reduce the number of vertical movements this way.



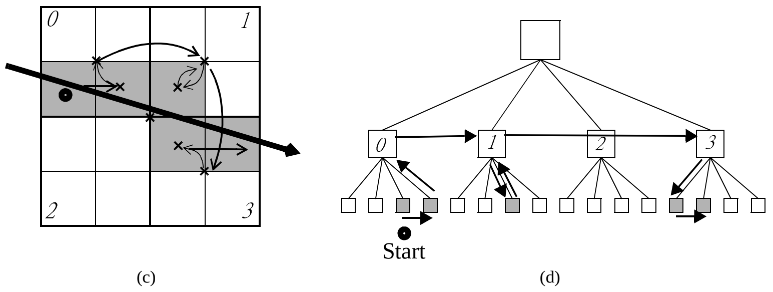


Figure 39: Bottom-up approaches. (a) A ray traverses the octree subdivision. The small arrows indicate the sequence of octants examined using Sandor's [103] and Samet's [101,102] approaches. (b) The corresponding tree and search path. (c) A ray traverses the same octree subdivision using Levoy's approach [81]. (d) The corresponding tree and search path.

A comparison of the two bottom-up approaches is shown in Figure 39. Figure 39(a) shows a ray passing through an octree subdivision and first several steps of its traversal path. Figure 39(b) shows the path of ray traversal on the octree using Sandor's [103] and Samet's [101,102] approaches. Figures 39 (c) and (d) shows the same process using Levoy's approach [81].

Recursive Top-Down Octree Traversal

Spackman and Willis [108] propose a sophisticated top-down recursive algorithm for ray traversal. The next cell visited by the ray is determined by two decision variables, one comparison variable, and increments from an update vector. The two decision variables H_{SMART} and V_{SMART} control the horizontal and vertical movements, respectively. The comparison variable $V_{compare}$ uses special encoding to find the correct octant. The update vector is scaled by child width at each iteration. The entire ray navigation can be performed with only integer operations. The top-down recursive approach is depicted in Figure 40 to compare with other approaches. Chen [24] proposes a method almost identical to that of Spackman and Willis [108]. The only difference is that the latter do not care about the exact exit point of the current voxel. Chen [24] maintains the exact coordinate of the exit point during the ray traversal process.

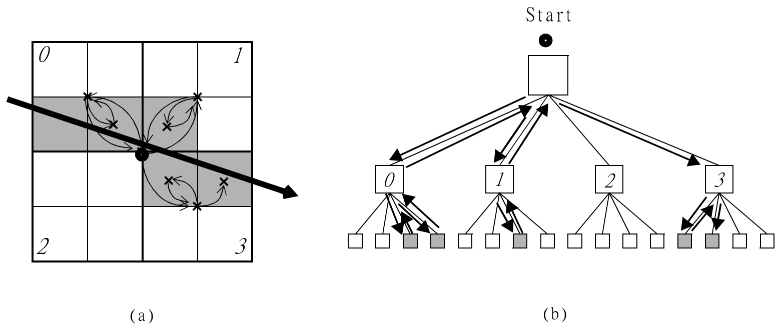


Figure 40: Top-down recursive approach

The problem of Spackman and Willis' approach [108] is that their mechanism is hard to understand. Revelles et al. [98] propose an alternative top-down method that is easier to understand. Their algorithm is based on the fact that for each octree node, at most four octants can be pierced by a ray. The first step is to select the first sub-node hit by the ray. Then select the next sub-node until the current parent node is exited.

To illustrate how to find the first sub-node hit by a ray, consider a quadtree cell as shown in Figure 41. Suppose the lower-left coordinate of the cell is (x_0, y_0), the upper-right coordinate of the cell is (x_1, y_1), and the coordinate of median point is (x_m, y_m). A ray r that oriented left-to-right on a line of positive slope can be parameterized by $t_r > 0$. The ray r intersects the planes $x = x_0$, $y = y_0$, $x = x_m$, $y = y_m$, $x = x_1$, and $y = y_1$ at point t_{x_0}, t_{y_0}, t_{x_m}, t_{y_m}, t_{x_1}, and t_{y_1} respectively. If $t_{x_0} > t_{y_0}$, we know the ray enters the cell from the left, rather than from the bottom. To determine which sub-node the ray enters, we simply check whether t_{x_0} is greater than t_{y_m}. If it is, the ray enters sub-node 2 as r_1 shown in Figure 41. Otherwise, the ray enters sub-node 0 as r_2. If $t_{y_0} > t_{x_0}$, the ray enters the cell from the down side. Similarly, if t_{y_0} is greater than t_{x_m}, the ray enters sub-node 1 as shown in r_4. Otherwise, the ray enters sub-node 0 as shown in r_3. The algorithm for finding which sub-node the ray enters is summarized as follows.

Algorithm FindEntryNode($t_{x_0}, t_{y_0}, t_{x_m}, t_{y_m}$)

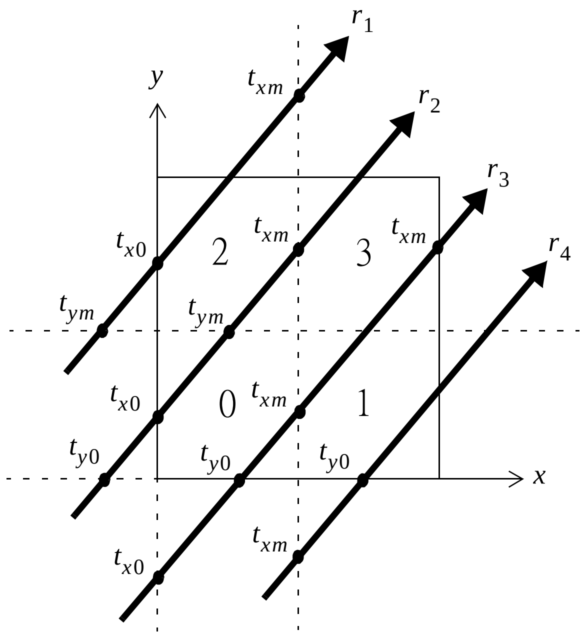


Figure 41: Determining the entry node and the next node using Revelles et al.'s approach [98]

Input: Four reference points on the ray.
Output: The sub-node first hit by the ray.

```
1: if ($t_{x_0} > t_{y_0}$) then
2:    if ($t_{x_0} > t_{y_m}$) then
3:       return sub-node 2;
4:    else
5:       return sub-node 0;
6:    end if
7: end if
8: if ($t_{y_0} > t_{x_0}$) then
9:    if ($t_{y_0} > t_{x_m}$) then
10:      return sub-node 1;
11:   else
12:      return sub-node 0;
13:   end if
14: end if
```

To determine the next sub-node visited by the ray, all we need are the reference points t_{x_1} and t_{y_1}. We use a quadtree to illustrate the idea. Three-dimensional case can be handled by also considering the z-coordinate. The main idea is to determine which hyperplane the ray intersects first. The next sub-node visited by the ray depends on the current sub-node, t_{x_1} and t_{y_1}. The process is illustrated in FindNextNode below. If none of the cases is true, the ray is out of the scope of the current node and we have to trace from the parent node.

Algorithm FindNextNode(t_{x_1}, t_{y_1})

Input: Two reference points on the ray.
Output: The next sub-node hit by the ray.

```
if ($t_{x_1} < t_{y_1}$) then
    if current sub-node is 0 then
        return sub-node 1;
    else
        if current sub-node is 2 then
            return sub-node 3;
        end if
    end if
end if
if ($t_{y_1} < t_{x_1}$) then
    if current sub-node is 0 then
        return sub-node 2;
    else
        if current sub-node is 1 then
            return sub-node 3;
        end if
    end if
end if
return $NULL$;
```

As we mentioned before, object duplication is a common problem of all space-oriented partitioning methods, and the octree is no exception. In addition, the vertical movements within an octree are expensive because they often involve following pointers between different levels. Unfortunately, vertical movements often incurred during ray traversal (over one-half of the total movements [64]). Especially when the distribution of scene objects is highly biased, we may created an octree with large depth. This makes the problem even worse. Despite of the these problems, octrees are still used for ray tracing frequently because they can naturally adapt to geometric complexity of a scene. One can easily adjust the parameters of an octree to optimize its performance, such as choose better splitting planes [82, 119] or create a balanced octree [10].

7.3 Hierarchical Multiway Subdivisions

Hierarchical multiway subdivision method is most commonly implemented by layered uniform grids. The basic concept and various ways of constructing it are described in section 7.3.1. The calculation of a ray stepping through the grid is fast and simple in general, however, there are minor differences depending on how the grid is constructed. We discuss these variations in section 7.3.2. The hierarchical multiway subdivision approach is concluded in section 7.3.3.

7.3.1 Construction

The problem of conventional uniform grid subdivision method is twofold. The first problem is, as we have seen in section 4, the use of three-dimensional array leads to a cubic growth of the memory requirement. Second, although finer-space subdivision gives better object selection resolution and fewer ray-object tests, however, as the subdivision increases, the improvement may be offset by a linear degradation caused by the increase in the number of ray-grid intersection tests. To solve the first problem, Hsiung and Thibadeau [64] introduce

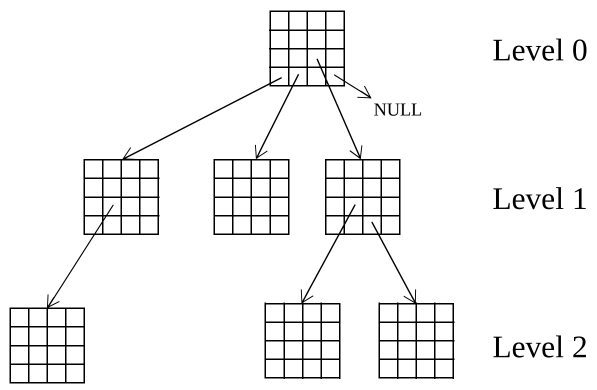


Figure 42: EN-tree: octree with enlarged nodes

a data structure called *EN-tree* (EN stands for ENlarged). Later in Section 7.3.2 we will discuss how a ray traverses an EN-tree. But first let us take a look at how an EN-tree is constructed. The EN-tree is a hybrid tree that integrates the 3D array into a "octree-like" data structure. Figure 42 shows an EN-tree in 2D. At each internal node of the tree, instead of dividing each side in half for a total of eight children as a typical octree, each side is divided into four or eight parts. This creates 4^3 to 8^3 subnodes for each internal node.

EN-tree may look like a non-uniform space subdivision such as octree. In fact, it is different from any of the octree spatial subdivision discussed in Section 7.2. It is a hybrid data structure that combines an "octree"-like data structure and SEADS. The differences between an EN-tree and an octree is not only limited to the number of subnodes – the number of subnodes in octree is always 8, while the number of subnodes in EN-tree can be either 4^3 or 8^3. Furthermore, the object space in octree is hierarchically subdivided. The splitting plane can be located at the space median, object median, or somewhere in between. On the other hand, object space in EN-tree is regularly divided into voxels. Objects may be allowed to exist at any level of an octree [46], EN-tree only stores objects at the bottom level which always have the same spatial resolution. The tree traversal in octree involves complicated neighbor finding techniques. In EN-tree data structure, the regularly subdivided space is traversed in the same way as Fujimoto's SEADS. Vertical traversal can be eliminated by using a hash table to hash grid cells to their storage. The philosophy behind Hsiung's approach is to save some memory space by dropping empty subspace. Only occupied subspaces are considered to be useful and are stored in EN-tree data structure.

Cazals and Puech [22,23] present two kinds of adaptive data structures based on uniform grid: the recursive grid and the hierarchical uniform grid. The first step of constructing both of these data structures is the same; the basic uniform grid has to be constructed. They construct the uniform grid by dividing the scene into $\alpha^3 n$ voxels, where α is a pre-selected positive constant and n is the number of objects. To keep the description simple, we will assume $\alpha = 1$. Each side along x-, y-, and z-axis is divided into $\sqrt[3]{n}$ intervals.

Cazals et al. use the number of objects in a grid cell as the termination threshold. Their recursive grid partitions the grid cell into subspaces recursively, as long as the grid cell contains more than a fixed number of objects. The recursive grid structure is similar to Hsiung's EN-tree. However, there are two differences between them. EN-tree always partitions the space into a fixed number of grid cells, while recursive grid divides the space based on the number of objects within the current grid cell. The other difference is the termination criterion. EN-tree stops splitting into subnodes, if the cell size is less than or equal to a pre-selected value. Recursive grid, on the other hand, terminate the recursive step when the number of objects within the grid cell is less than or equal to a pre-selected value. Therefore, recursive grid is more adaptive than EN-tree.

Hierarchical uniform grid (HUG) is more sophisticated than other grid structures that we discussed above. The idea behind HUG is to group together nearby objects of the same size. After the basic uniform grid is constructed, further "filtering" and "clustering" steps need to be taken before building the hierarchy structure. The algorithm for constructing HUG is shown below.

```
Algorithm HUGConstruct($S, B, m, \delta$)
Input: $S$ = a set of objects, $B$ = bounding box, $m$ = number of levels (filter level), $\delta$ = the
    maximum distance between objects that are within the same cluster;
Output: A HUG structure with $B$ as its top level node.
    {Bottom-up construction phase}
    {Filtering step}
 1: Split $S$ into $m$ subsets such that each subset $S_k$, $1 \leq k \leq m$, contains objects of similar
    size;
    {Clustering step}
 2: Within each subset $S_k$, partition the objects into subgroups such that the distance be-
    tween any two objects within a subgroup is less than $\delta$, i.e., objects are close to each
    other;
    {Top-down construction phase}
```

```
3: create the highest level cluster grid and store its objects;
4: for all other filter levels, in decreasing order do
5:    for each cluster of the level do
6:       create cluster grid and store its objects;
7:       recursively insert this grid in the hierarchy;
8:    end for
9: end for
```

At the filtering step (line 1), objects with similar length are put into the same *level* based on the pre-selected *filter*. A filter $\mathcal{F}$ is a strictly increasing sequence of positive real numbers $\{f_1, f_2, \ldots f_m\}$ such that $d_1 \in [f_1, f_2)$ and $d_{m-1} \in [f_{m-1}, f_m)$, where d_1 is the maximum length allowed in level l_1, and d_{m-1} is the minimum length allowed in level l_{m-1}. A *level* l_k of the filter $\mathcal{F}$ is an interval $l_k = [f_k, f_{k+1})$. We now collect into set S_k, $1 \leq k \leq m$, all objects with diameters in l_k. This step can be done in a manner similar to bucketsort [29]. The sorting time is then linear in the number of objects.

For the clustering step (line 2), within each subset of the same filtering level, find those objects that are close to each other. We can pre-select a threshold distance δ first. Then pick a direction along one of the x-, y-, or z-axes and find the objects that are close to each other by checking them one against each other to see if all of them are within the threshold distance. The qualified objects are the potential candidates to form a cluster. The process goes on by checking the next axis direction on those candidates, and so on. A bucket-like cluster will be formed such that if any objects o_i, o_j are in the same cluster, then their distance $d(o_i, o_j) < \delta$ in all of the x, y, and z directions.

In lines 3-9, the HUG structure is constructed in a top-down fashion according to the filter levels. Using this approach big objects are stored in the grid cells that belong to higher level of the structure. HUG is not a tree, unlike recursive grid or octree, but a “layered” structure similar to a DAG. The recursive grid and octree are constructed in a top-down fashion. The bounding box hierarchy can be constructed in either top-down or bottom-up way, but not both. HUG is built by a bottom-up and a top-down pass.

We now use a small example to explain how HUG is constructed. On the left hand side of Figure 43, a scene is subdivided into three levels of uniform grids. In the construction phase of HUGConstruct(S, B, m, δ), the following steps take place, after the objects have been classified into three groups, according to size and the clustered according to their location.

1. The whole scene is subdivided as top level grid 3. Large objects A, B, and C are stored in the grid cells that intersect the objects as shown in Figure 43.

2. The next step is to create grid 2. For each cell in grid 3, that intersects with grid 2, insert a pointer to grid 2. Medium size objects D, E, and F are stored into the corresponding grid cells.

3. The next step is to create grid 1a. It is fully contained within cell (2,2) of grid 3, and intersects with cell (1,2) of grid 2 but not fully contained within grid 2. A pointer to

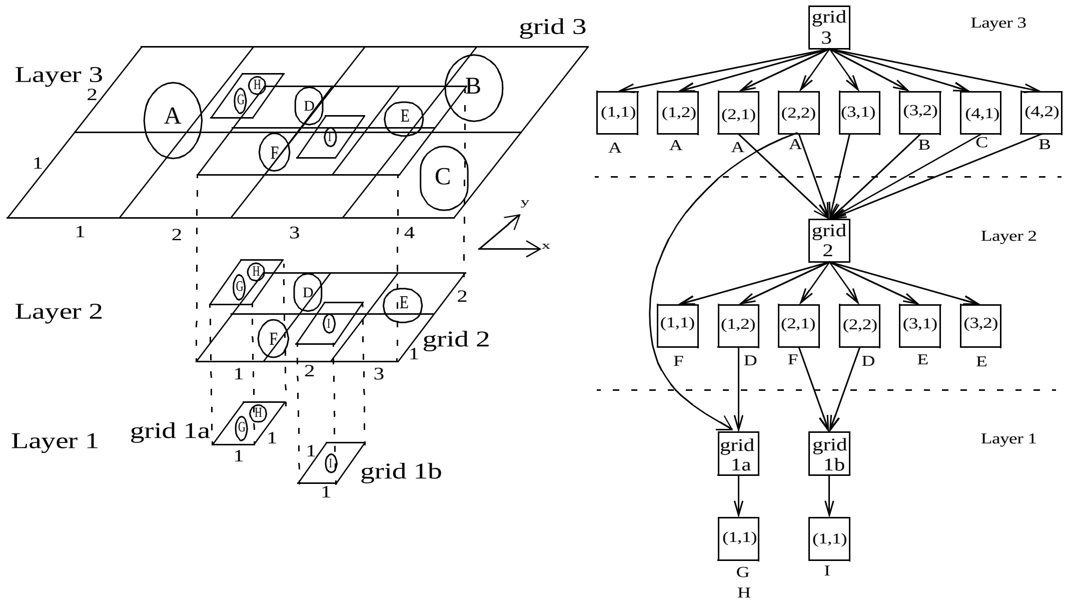


Figure 43: HUG layer view (left) and hierarchical view (right)

grid 1a is then inserted into cell (1,2) of grid 2 as well as cell (2,2) of grid 3. Small objects G and H are stored into grid 1a.

4. The final step is to create grid 1b. It is fully contained within grid 2, we insert a pointer to it into (2,1) and (2,2). Finally, object I is stored into grid 1b. STOP.

The resulting structure stores larger objects in the higher levels and smaller objects in the lower levels. In this example, large objects A, B, and C belong to top level grid 3. The medium sized objects D, E, and F are clustered and belong to grid 2. The small objects G, H, and I have the same or similar size, however, because object I is far from others, it is not clustered with other objects in the same layer. Thus, objects G and H are clustered and belong to grid 1a, whereas object I is in grid 1b by itself.

7.3.2 Ray Traversal

Incremental algorithms are used in traditional ray traversal for uniform grid. The disadvantage is, if there are many empty regions, passing through these empty regions is unnecessary and inefficient. Hsiung and Thibadeau [64] use a ray traversal method that is an adaptive, multiple step-size 3DDDA which skips empty regions in larger than unit step size. Figure 44 is a conceptual diagram of how it works. Each node of the tree represents a $1/4^3$ subspace of its parent.

If a ray traverses uniform grid with ARTS approach, the path of the ray is represented by the arrows from the leftmost point A, stepping through the subdivision space in unit step-size, until it reaches the rightmost point B at the bottom in Figure 44. Empty grid cells

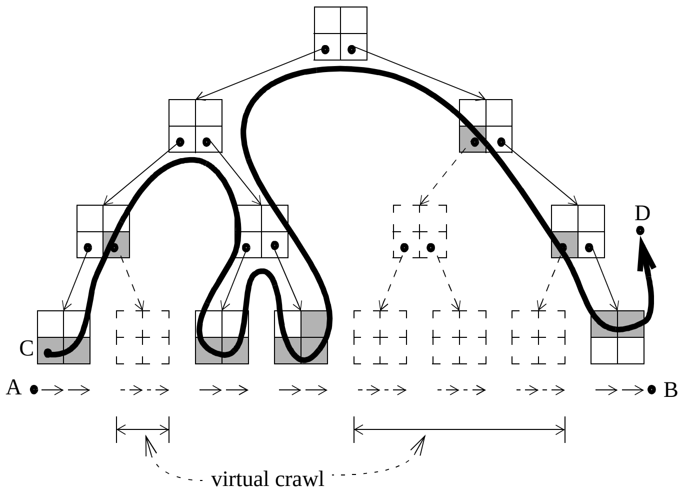


Figure 44: FINE-ARTS's EN-tree traversal and virtual crawl.

are drawn dashed. In FINE-ARTS approach, empty grid cells are absent from the EN-tree. In order for 3DDDA to step through from point A to point B, the ray has to conceptually "crawl" the grid cells that do not even exist in EN-tree. In other words, 3DDDA "virtually" steps through the subspace. Hsiung calls this stepping mechanism *virtual crawl*.

Hsiung's FINE-ARTS approach crawls the EN-tree at multiple levels and only traverses the *existing* nodes in the EN-tree. The path of the ray is depicted as a thick line from C to D. Hsiung's traversing algorithm is in effect a depth-first traversal algorithm of the portion of the tree met by the ray. The differences between EN-tree and octree traversal is that EN-tree has larger branching factor than octree. Vertical traversal in octree is more frequent (over one-half of all linear stepping) and costly. By subdividing each node into 4^3 or 8^3 subspaces and thus increasing the arity of the tree to 64 or 4096, the height of EN-tree is reduced significantly compared to the conventional octree. Therefore, the number of vertical traversal steps in an EN-tree is much lower than in an octree. Hsiung's experimental result shows that in SPD's "rings" test scene, ARTS' performance is $O(N)$ in the worst case, while FINE-ARTS' performance is $O(\lg N)$, where N is the grid resolution.

Cazals and Puech's recursive grid is traversed similarly to Hsiung's FINE-ARTS approach, except that their algorithm spends more time on the horizontal movement because they split the grid cells into more subcells than Hsiung's EN-tree. Traversal of Cazals and Puech's HUG structure [22,23] is a little different. The ray first enters the top layer grid the same way as traversing other uniform grid structures. All of the objects stored in the current grid cell have to be tested one-by-one. The subgrids which are one layer lower have to be tested by following the pointer to the subgrid, and then visiting each subgrid recursively by following the pointer to the next layer. If no intersection is found, the recursion step returns and goes on to the next object or subgrid on the ray path. If no hit is found within the current grid cell, we then step to the next grid cell using 3DDDA. The intersection tests are performed on all objects and subgrids in order along the ray until a hit is detected or the ray leaves the scene. One advantage of HUG over recursive grid is that the recursive step does not have to step through all of the layers in order to reach the bottom of the hierarchy.

As shown on the right hand side of Figure 43, a pointer from grid cell (2,2) in layer 3 allows us to jump directly to grid 1a without going through grid 2. Thus fewer steps may be taken for vertical traversal through a hierarchy in an HUG than in a recursive grid in some cases.

7.3.3 Discussion

The data structure of uniform grid is simple and easy to construct. The basic structure is a 3D array. To partition the space into uniform grid, all we have to do is to map the coordinates from object space into grid space. There is no sorting needed in the preprocessing stage if uniform grid is used as proposed by Fujimoto et al. [47] and Yagel et al. [125]. Assuming that a grid cell can only intersect at most one object, the worst case time complexity for constructing a uniform grid is $O(n + N^3)$, where n is the number of objects and N is the resolution of the grid along each direction.

The resolution of a uniform grid is independent of the object distribution. It relies on a pre-selected value, which is a positive integer between 1 and the maximum resolution along x-, y- and z-axes. Without lost of generality, we assume the maximum resolution along each axis direction is the same. If we divide the scene into N intervals along each axis, the memory space complexity will then be $O(N^3)$ in three-dimensional space.

Uniform grid subdivision method seems to be the easiest data structure if we look at it superficially. However, the procedure to determine the right value for N is not well understood. If the chosen grid resolution N is too small, for example $N = 1$, the whole scene is one big grid cell. The ray-object intersection tests need to be performed against all of the objects in the scene in the worst case. If we pick the wrong grid size, the chance of the worst case to happen can increase significantly. This happens if some grid cells meet a large fraction of the objects. It is just like the brute-force approach that does not apply spatial subdivision method at all. At the other extreme, Yagel et al. [125] “voxelize” the scene into unit voxels. The value of N in their approach is the maximum resolution along x, y or z-axes. This approach will need too much memory to be practical.

Another problem when applying uniform grids to ray tracing in a sparse scene is: A lot of memory is allocated to the empty grid cells that simply waste space. Cohen and Sheffer [27] try to use the empty grid cells by applying a proximity technique. Hsiung and Thibadeau [64] try to reduce the memory usage of uniform grid approach by applying grid structures recursively. Cazals et al.'s experimental results [22,23] confirm that using recursive grid can save some memory space.

The philosophy behind the uniform grid approach is that many small simple steps are better than one big complicated step. Moving from one grid cell to another grid cell involves only simple arithmetic. Usually integer operations are preferred. Experimental results [47, 27, 22, 23, 76, 62] show that uniform grid or its variants can be the most efficient scheme in some cases, if we choose the right grid size. However, the efficiency depends on the scene. Uniform grid structures outperform other data structures when the objects are uniformly distributed; a set of objects with uniform distribution may not arise very frequently in the real world, however.

To guess the distribution of objects, computational statistics methods may have to be applied. Cazals et al. [22,23] try to explore this direction by combining filtering and clustering techniques with statistical analysis of the scene. Their research focuses on the statistical properties of the scene rather than the local properties of a particular object. However, the result of their works proves that it is a difficult problem. After applying all these filtering, clustering, and statistic scene analysis methods, their hierarchy of uniform grids still cannot beat the speed of simply using recursive uniform grids in many cases [76].

Uniform grid method subdivides the scene by the pre-determined grid size. Thus it fails to take the advantage of object coherency. It can easily fill up the memory in a complex scene at high resolutions using 3D array to store the grid information. Further study of how to apply efficient external I/O algorithms to explore the memory coherent properties can also be pursued. An attempt towards this direction can be found in Pharr et al.'s paper [93]. Finding the optimal grid size is still a mystery. So far no one knows how to determine the grid size that is efficient in terms of both memory consumption and ray traversal.

8 Hierarchical Hybrid Structures

In section 5, we have seen that flat structures can be combined to take the advantage of the benefits of each of the participating structures. We continue our survey of hybrid structures after investigating hierarchical structures. First, combinations of two hierarchical structures are discussed in section 8.1. Then we discuss the approaches that combine a flat structure and a hierarchical structure in section 8.2.

8.1 Hierarchical-Hierarchical Hybrids

The hierarchical-hierarchical hybrid structures are constructed by building a hierarchical structure on top of another hierarchical structure to gain the benefits of each. Theoretically, there can be any number of hierarchical structures built on top of each other as proposed by Kirk and Arvo [74]. However, we have only found references in the literature to two-level hierarchical-hierarchical hybrid structures for ray tracing. The general approach starts from constructing the upper-level hierarchical data structure as we mentioned earlier. When certain termination criteria for the upper-level structure are met, we switch to building another, lower-level hierarchical structure. Each data structure within a hybrid can be constructed individually. The trick here is when to terminate the upper-level structure and switch to the lower-level structure. Ray traversals with each data structure are similar to that we discussed in the previous sections. Therefore, we will not elaborate upon the traversal methods in this section.

We found three main criteria in the literature that can help us making the decision for the switch: the object count in a cell, the density ratio of the total volume enclosed by the objects to the total volume of the cell, and the amount of projected void area.

Scherson and Caspary's [104] use the first termination criteria to construct an octree on top of BVH. Their implementation of octree-BVH hybrid is based on two observations. An octree is more efficient when the cells are large, and is less efficient when the cells are small, as revealed by their results. If we divide the space into large chunks using octree, the number of fragmented objects can be reduced. Their results also show that BVH is good for a high-resolution scene with small number of objects. The construction starts from the top level with a traditional octree; see Section 7.2. The space is subdivided recursively until it reaches the termination threshold. Scherson and Caspary suggest the octree phase should stop when the number of objects within the cell is equal to 100. The number 100 is also based on their observations that this minimizes the execution time when tracing their sample scenes. When the number of objects is less than one hundred, they build BVH within the octree cell; see section 6.1.

Glassner [50] introduces a similar hybrid structure that also builds an octree on top of BVH using 3 as the threshold size for switching over to BVH. Glassner's test scenes and Scherson's test scenes [104] are different. The selection of object count threshold depends on the test scenes. Glassner also implements the density ratio criterion in addition to the object count. If an octree cell contains at most three objects, Glassner checks the "density ratio" of the current cell. If the ratio of the volume of the objects and the volume of the cell is less than 0.3, the octree is further subdivided even though the cell contains few objects. This technique is useful if the objects themselves represent bounding volumes of smaller structures.

Another hybrid structure that also uses the object count criterion is introduced by Formella and Gill [39]. Their hybrid structure is constructed by building a modified BSP tree on top of BVH. The problem of traditional BSP tree as we described in section 7.1.1 is that it can store an object in several cells if the object is cut by the splitting plane. This increases the space requirements and the depth of the tree. To eliminate this problem, Formella and Gill sort the objects into one of the 27 possible categories according to which side of each splitting plane they lie on and which of the splitting planes they meet. Figure 45 lists all 27 subspaces created using Formella-Gill approach. The first row represents the space with no splitting plane. We add to this class all objects that meet all three splitting planes. The second row shows a space can be cut by a splitting plane aligned with x-, y-, or z-axis. Each cut creates two subspaces. If we cut the space with two splitting planes, each cut creates four subspaces as shown in the third row. Finally, a space cut by three axis-aligned splitting planes introduces eight subspaces as shown in the fourth row of Figure 45.

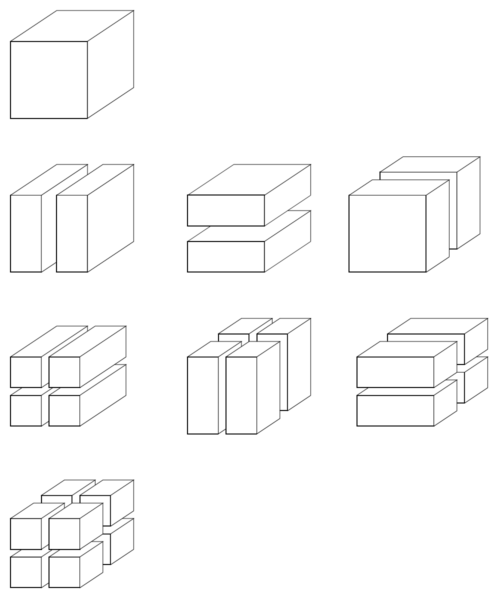


Figure 45: List of all the category of subspaces created by Formella and Gill [39]

The modified BSP structure is no longer a space-oriented partition. Each object only belongs to one of the categories so that there is no duplication. This approach guarantees linear space requirement of their modified BSP tree structure. The construction starts from the top level bounding box of the entire scene and repeats recursively until fewer than 27 objects remain in a subspace. At this point, the BVH construction is applied. Ray traversal starts from the root of the tree recursively searching for the subspace or object hit by the ray. Once the ray enters the bottom level of the modified BSP tree, we switch to BVH-style traversal. If there is no intersection found within the current subspace, we switch back to the previous tree traversal method to find the next neighbor.

The third approach, using the amount of projected void area as the switching criterion, is implemented by Subramanian and Fussell [110, 111]. They choose k-D tree as the upper level structure and BVH as the lower level structure. Although k-D tree space subdivision is adaptive, the axis-aligned partitioning planes can produce large void spaces. They are the potential sources of inefficiency during the ray traversal. BVH is good for culling away large void spaces and provide a compact representation for the objects. We have seen Subramanian and Fussell's k-D tree construction and ray traversal method in section 7.1.1. They apply MacDonald and Booth's *surface area heuristic* [82] as their termination criteria. Surface area heuristic is based on the assumption that the probability of a ray entering a region is proportional to the surface area of this region. Using this assumption, they predict the cost (time) needed for ray tracing as follows.

$$T = T_{inner} \cdot \sum_i \frac{SA(i)}{SA(root)} + T_{leaf} \cdot \sum_l \frac{SA(l)}{SA(root)} + T_{obj} \cdot \sum_o \frac{SA(o) \cdot N(o)}{SA(root)} \tag{3}$$

where T_{inner} is the cost of traversing internal node i of a hierarchy , $SA(i)$ is the surface area of internal node i, $SA(root)$ is the surface area of the root node, T_{leaf} is the cost of traversing leaf node l, $SA(l)$ is the surface area of leaf node l, T_{obj} is the cost of intersection test for object o, $SA(o)$ is the surface area of object o, $N(o)$ is the number of leaves where the object resides. The first summation is over all interior nodes i, the second over the leaves ℓ, and the third over all objects o. For object oriented partitioning methods, $N(o)$ is always one.

Subramanian-Fussell's k-D tree [110, 111] stops further dividing the space when equation (3) reaches a minimum value. After then, they start building BVH within each leaf node. The construction and ray traversal of their lower level BVH structure is similar to that of Goldsmith and Salmon's Automatic Bounding Volume Hierarchy (ABVH) [52]. Their method is based on the conditional probability of a ray hitting an inner volume given that it has hit the surrounding volume. If the ray has less chance to hit a bounding volume, it has less chance to perform the ray-object intersection test. Therefore, the time spent on ray tracing can be reduced.

8.2 Hierarchical-Flat Hybrids

The simplest form to build a hierarchical-flat hybrid is to mix different types of bounding volumes and construct a hierarchy for them [117]. It results in a more flexible BVH than using just a single type of BVH, as we described in section 6.1. The hybrid BVH structure is good for scenes that have various forms of objects. For each object, we can choose the volume that encloses it the tightest. The benefits of mixing different types of bounding volumes are described in section 5. Constructing a hierarchy over these hybrid bounding volumes can take further advantage. The hierarchical structure, if balanced, can reduce the expected ray traversal time from $O(n)$ to $O(\log n)$, where n is the number of objects.

Despite this improvement, research shows that BVH by itself is not good enough for ray tracing [75]. One solution is to integrate uniform grid with BVH. Section 4 demonstrates a flat uniform grid-like structure that is conceptually easy to implement, although it produces excessive void regions. We also describe several ways to alleviate the weaknesses of conventional uniform grid by using multi-level grids in section 7.3. Here we would like to describe another variant of multi-level grids that is conceptually different from the ones we discussed there. The structure is called adaptive grid, introduced by Klimaszewski and Sederberg [75]. This hybrid structure not only alleviates the weaknesses of uniform grid but also tries to capitalize on its strength.

Unlike other hybrid structures, the construction of adaptive grid starts from the upper level BVH. Klimaszewski and Sederberg choose axis-aligned boxes due to their simplicity. For those bounding boxes that are close to each other, they merge the boxes together if the new bounding box's surface area is smaller than the sum of the surface areas of the two boxes before merging. At the bottom level, they create local uniform grids for all of the remaining bounding boxes. Their algorithm is listed as follows.

```
Algorithm AdaptiveGridConstruct($\mathcal{S}$)
Input: A set $\mathcal{S}$ of objects.
Output: The adaptive grid.
 1: for all objects do
 2:    surround the object with a bounding box;
 3: end for
 4: for all bounding boxes do
 5:    merge nearby boxes;
 6: end for
 7: for all remaining bounding boxes do
 8:    insert box into BVH tree using surface area criterion [52];
 9:    if box surface area is too large or the box is under-populated then
10:       merge box with its parent;
11:    end if
12: end for
13: for all bounding boxes in hierarchy do
14:    construct a local uniform grid for each box;
15: end for
```

In the "teapot in a stadium" problem, the test scene has a very small object (the teapot) inside a very large object (the stadium). If we use traditional octree subdivision for this scene, the result is a very deep tree which makes ray traversal very inefficient (Figure 46(a)). If we use traditional uniform grid subdivision as described in section 4, we will create many empty grids that waste space (Figure 46(b)). The adaptive grid structure is designed to solve this problem (Figure 46(c)). In fact, Havran and Sixta's experimental result [62] shows it is only good for this kind of problem. For scattered scenes, adaptive grid does not perform well. Another problem of adaptive grid is it is harder to implement than other data structures. The uniform grid, if the grid size is set up properly, performs better in many cases in SPD

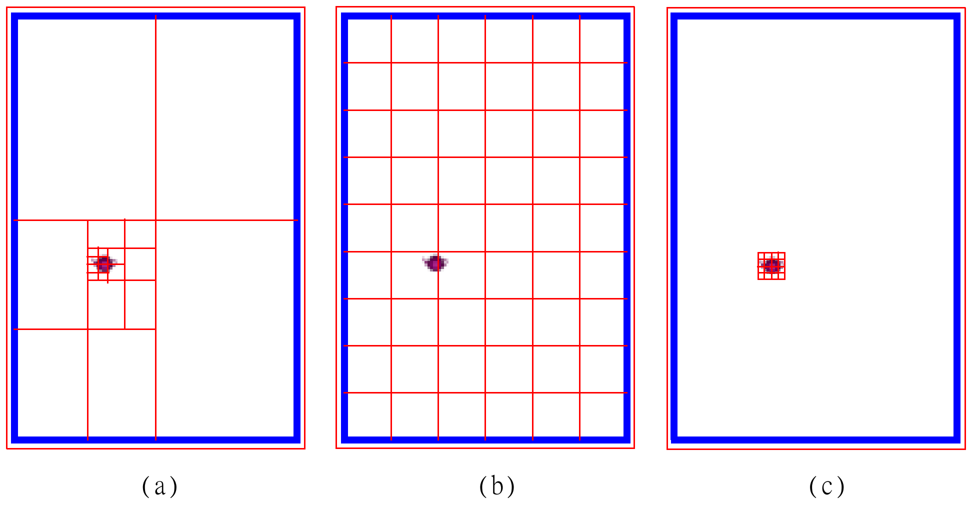


Figure 46: Comparision of different approaches for the teapot-in-a-stadium scene. (a) octree, (b) uniform grid, (c) adaptive grid.

scenes [59]. This is something that one has to be careful about when designing a hybrid structure; one can always end up with a structure that is more difficult to implement but cannot speedup the ray traversal time. This leads to an interesting article debating the performance of various grid structures in Ray Tracing News [76]. The conclusion of this debate is that the most efficient scheme for ray tracing is really scene dependent. For the "teapot in a stadium" problem, a simple solution is implemented by Kolb and Bogart [78] in RayShade 4.0. They provide a two-level grid method: one for the entire scene and the other for complex objects. The method performs better than one level uniform grid and recursive BSP tree according to Jansen and de Leeuw's test [69].

The same idea of using a uniform grid as a sub-structure can be applied to k-D tree as well. In Pradhan and Mukhopadhyay's adaptive cell subdivision [95], the upper-level structure is a k-D tree. The leaf nodes of the k-D tree are further subdivided by uniform grid. The trick here is to place a virtual grid on the scene before doing any work. The space subdivision step is illustrated in Figure 47. The first step is to create a "virtual" grid on the scene (Figure 47(a)). Then construct a k-D tree subdivision as shown in Figure 47(b). The k-D tree subdivision always picks the splitting plane that is the space median. The last refinement step is to snap the splitting plane to the boundary of the line of the "virtual" grid, as shown in Figure 47(c). This way we can be sure the size of each k-D tree sub-region is divisible by the size of a grid cell.

The termination criterion of the upper level structure is the number of objects within the node. As usual, this constant is pre-selected before the structure is built. The difference between adaptive grid [75] and adaptive cell [95] is the size of each uniform grid can be different in the former structure. Adaptive cell structure uses fixed size uniform grid for all leaf nodes. When a ray enters the structure, we first find the first leaf node that contains the intersection point along the ray path. Once the leaf node is identified, the ray pushes forward using DDA traversal method, as in section 4. The adaptive grid, on the other hand,

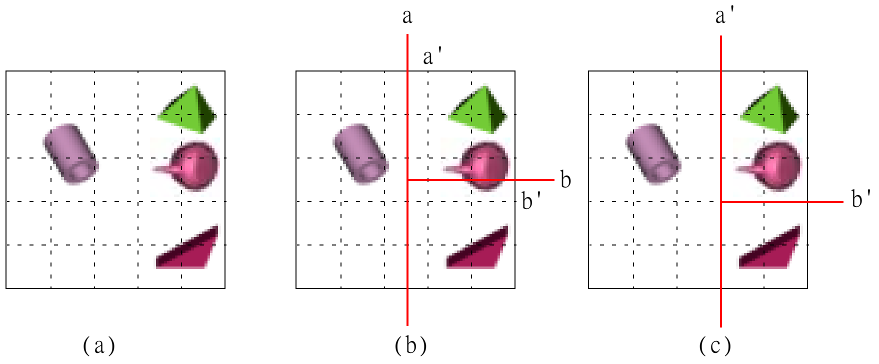


Figure 47: Dividing the space into adaptive cell. (a) A scene with virtual grid. (b) A k-D tree subdivision using space median policy. (c) A k-D tree subdivision *snaps* to the grid boundary

uses local grid with each subregion.

All of the hybrid structures that we have seen so far are two-level hybrids that mix two kinds of data structures. In principle, a hybrid structure can be the combination of more than two structures. Indeed, Kirk and Arvo [74] propose a three-level hybrid structure for ray tracing. The top level is a coarse uniform grid around the entire scene. The middle level can be either a refined uniform grid or octree around each cluster of objects. In their sample scene, a cluster of objects is an individual ride in the amusement park. For detailed elements of each ride, BVH is used as the low level structure.

If we skip the top level coarse uniform grid and implement a uniform grid plus BVH hybrid, the amount of memory requirement is huge. Using a coarse uniform grid on the top level can cut down the consumed memory by grouping primitive objects into larger aggregate objects. Another advantage of adding one more level to the hybrid structure is the increase in flexibility. The parameters, e.g. resolution, of each level can be adjusted independently. The resulting structure is therefore more adaptive to a scene than a two-level hybrid. Despite of these advantages, the termination threshold for each level still has to be adjusted manually. Choosing the best parameter for all scenes is still a problem. We usually don't know whether the threshold value is good or not until the actual ray traversal step is done. A shortcut is to run several tests and pick the threshold value that produces the correct result in the shortest time.

Hybrid structures, in general, perform better than using a single data structure. That is why many researchers choose the hybrid approach. Table 5 illustrates what data structures are used in the hybrids we found in the literature. A check mark under each column indicates the underlying data structure for the hybrid. The problem is, when we want to implement a hybrid structure, which combination is most efficient? Unfortunately, there is no specific answer. Even for a single structure, how to select the method which is the fastest is still unknown, not to mention the combination of them. We also have to be careful when choosing the combination, as not all data structures work well together [12, 121]. Another problem of hybrids is the interface between different underlying data structures. It has to be consistent

Authors	BV	BVH	UG	kD(BSP)	Octree
Duncan et al. [35]				$\surd$	$\surd$
Formella and Gill [39]		$\surd$		$\surd$	
Fujimoto et al. [47]			$\surd$		$\surd$
Glassner [50]	$\surd$				$\surd$
Havran et al. [62]			$\surd$	$\surd$	
Jansen and Leeuw [69]			$\surd$	$\surd$	
Kirk and Arvo [74]		$\surd$	$\surd$		$\surd$
Klimaszewski and Sederberg [75]		$\surd$	$\surd$		
Pradhan and Mukhopadhyay [95]			$\surd$	$\surd$	
Scherson and Caspary [104]		$\surd$			$\surd$
Stolte and Caubet [109]			$\surd$		$\surd$
Subramanian [110]		$\surd$		$\surd$	
Sung [113]			$\surd$		$\surd$
Weghorst et al. [117]	$\surd$	$\surd$			
Woo [123]	$\surd$		$\surd$		

Table 5: A list of hybrid structures

so that we can easily switch from one structure to the other. Shirley et al. [105] and Heckbert [63] suggest several good ways to implement it. There is one more problem about constructing a hybrid structure. How do we determine when to switch from one structure to the other? Can we let the program find the optimum setting automatically? We do not know the answer yet, but Jansen [68] believes the parameters cannot be adjusted fully automatically.

PART IV

Conclusion

In this survey, we studied the data structures commonly used for ray tracing for the past two decades. Object-Oriented Partitioning (OOP) structures implemented by bounding volumes can speed up the intersection tests. Bounding Volume Hierarchy (BVH) can further reduce the number of such tests. The idea is to replace the time-consuming ray-object intersection tests by simpler and faster ray-extent intersection tests. They are suitable for scenes with small number of objects and where the shape of each object is complicated.

Type	Tightness	Intersection	Hierarchy	Reference
Sphere	loose	very fast	hard	[120]
Slabs	very tight	slow	medium	[73]
AABB	medium	fast	very easy	[56]
OBB	good	medium	easy	[60]

Table 6: Comparison of bounding volumes

Table 6 lists the comparison of four different types of bounding volumes. The ray-extent intersection test for a sphere is very fast, however, because using a sphere as a bounding volume usually leaves larger void area within the extent, as column 4 of Table 6 shows, it is not easy to construct a good hierarchy such that the extents do not overlap. It is very easy to construct a hierarchy using AABB, that is why AABB hierarchy is used very frequently. The extent constructed by a set of slabs can fit the primitive object very well, but it suffers from slow ray-extent intersection test. AABB and OBB are used widely because they are easy to construct and easy to perform intersection test. OBB requires additional coordinate transformation compared to AABB. A reasonable OBB is easy to construct using a heuristic method. However, constructing an optimal (minimum volume) OBB is very time consuming and is an interesting topic in computational geometry [88, 128, 15, 18], as we discussed in section 3.

If the number of objects is large, Space-Oriented Partitioning (SOP) approaches perform better than OOP approaches because SOP approaches significantly reduce the amount of ray-object intersection tests. A comparison of different SOP approaches is given in Table 7. Uniform grid is easy to construct. Ray traversal on a uniform grid is performed incrementally. The next cell calculation usually only involves integer arithmetic. However, uniform grid assumes objects are distributed in the scene uniformly. It may not perform very well if the objects are congested at some part of the scene and are sparse in the remainder of the scene. Furthermore, the grid size is always preset manually. It does not adapt to a specific scene. If we select the wrong grid size, the performance of uniform grid will degrade. Another problem of uniform grid is it creates many empty grid cells if the scene is not dense. This results in waste of memory and slows down ray traversal.

Method	Construction	Traverse	Adaptive	Reference
Uniform grid	very simple	simple	no	[46]
BSP-tree	hard	moderate	very adaptive	[8]
Octree	simple	hard	moderate	[49]
k-D tree	moderate	moderate	adaptive	[110]

Table 7: Comparison of SOP approaches

The most adaptive space subdivision structure is a BSP-tree. The orientation of splitting planes can be arbitrary. BSP-tree is an elegant data structure. However, it is hard to create a good or optimum BSP-tree. The splitting plane of a general BSP-tree can have any orientation, which makes it difficult to choose a good one from all the candidates. Another problem of BSP-tree pointed out by Steve Fortune [40] is the splitting planes of the BSP-tree can explode the storage. It is due to large number of duplicated links to the objects. Suppose we have constructed a good BSP-tree and we have enough memory to store the tree. Tracing the rays across BSP-tree can be slow because the splitting planes are in arbitrary direction, testing intersections between the rays and the splitting planes need more calculation than other data structures.

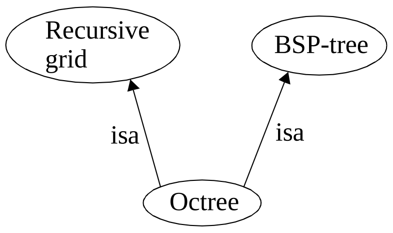


Figure 48: Relation between octree, recursive grid, and BSP-tree.

An octree has the advantages of both uniform grid and BSP-tree. In fact, we can say that octree “inherits” from both recursive uniform grid and BSP-tree. Figure 48 illustrates the relation between the three structures. We use the notation of Lakos [80] to represent the “IS-A” relation between these entities. Logical entity A “IS-A” B if and only if A *is a kind of* B. Constructing an octree is no harder than constructing a recursive grid. The subgrid of a recursive grid is depended on the pre-selected grid size. Octree, on the other hand, always restrict the number of subgrid to eight. If we restrict the splitting planes of a BSP-tree to be axis-aligned, and the splitting planes for x-, y-, and z-direction have to cut the space at the same time, the resulting structure becomes an octree. Therefore, octree is a special case of a BSP-tree.

The definition of an octree makes it less flexible than the general BSP-tree. A serious problem of octrees inherited from BSP-tree is the requirement of large memory. It also stores lots of pointers to objects that intersect many octree cells. Another problem of octree inherited from uniform grid is that octree is not good for sparse scenes [104], due to the fact that it is adaptive only to a certain degree. The most serious problem of octree is: There is no trivial way to traverse an octree. Tracing rays across an octree usually involves complicated neighbor finding techniques, as the number of neighbors of an octree cell is more than other data structures.

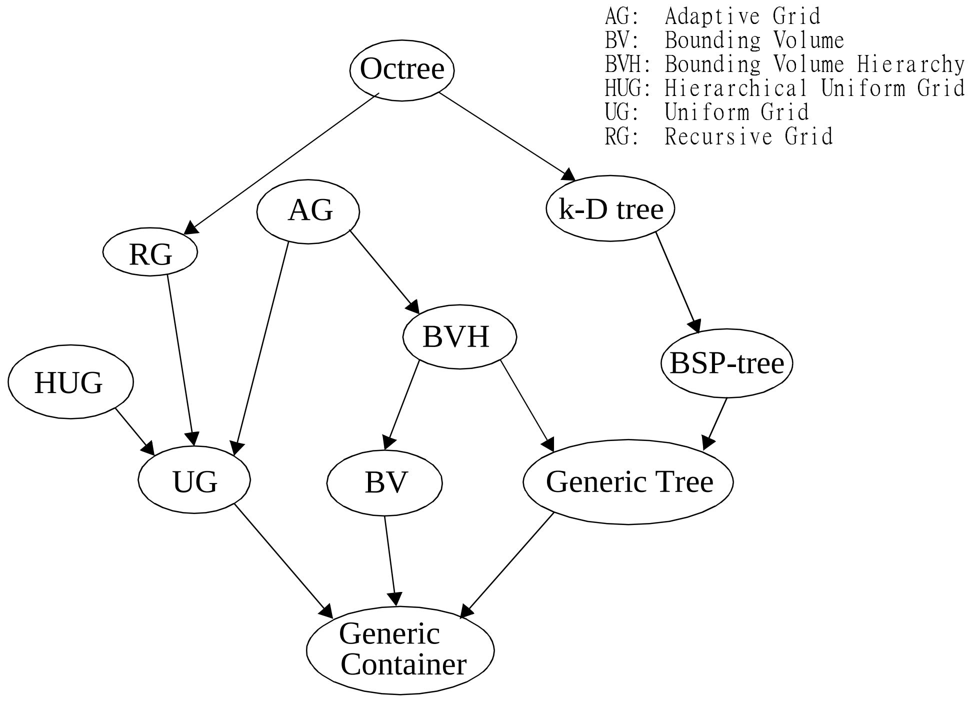
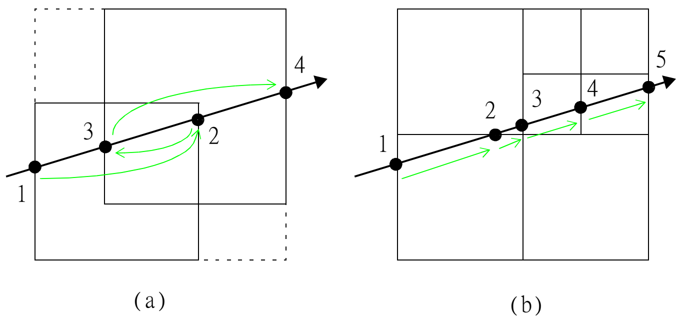


Figure 49: Relations between the data structures for ray tracing discussed in this survey.

A compromise between a general BSP-tree and an octree is the k-D tree. It is moderately simple to build and the data structure is moderately complex. Ray traversal in a k-D tree is less efficient than in uniform grid because floating-point arithmetic is involved. However, it is good for the scenes with non-uniform distribution of objects. A k-D tree is less adaptive than the general BSP-tree but is more adaptive than an octree. This makes k-D trees perform better than octrees in sparse scenes. The relationship between all of the data structures discussed in this survey is summarized in Figure 49. Each arc in the graph represents an "IS-A" relation between the logical entities. The generic container, generic tree are conceptual structures depicted to clarify the big picture. The structures in the higher levels can be viewed as special cases of the lower level structures. In this figure, we can easily identify that octree is actually a special case of a k-D tree and is also a special case of recursive grid, depending on how we look at it. k-D tree itself is a special case of a BSP-tree, and so on.

We also discussed many ray traversal algorithms for different data structures. Some of them are similar and can be used interchangeably. Jansen [70] classifies all of the ray traversal algorithms into two categories: sequential or recursive. Although he only considers bounding box and k-D tree structures, this classification can be generalized to all of the data structures that we have discussed.



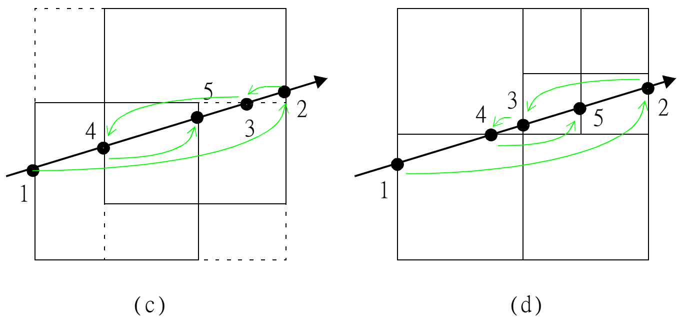


Figure 50: (a) Sequential algorithm traversal on OOP structure. (b) Sequential algorithm traversal on SOP structure. (c) Recursive algorithm traversal on OOP structure. (d) Recursive algorithm traversal on SOP structure. (from [70])

Conceptually, there are only two data structures in our survey, i.e., OOP structure and SOP structure. Applying Jansen's two traversal methods to our two data structures, we obtain four different ray traversal methods as in Figure 50. Figure 50(a) shows sequential algorithm working on OOP structure. We examine each bounding volume starting from the one that is closest to the ray origin. For each bounding volume, we need to check the entry point and the export of the ray. First, ray-object intersection test are performed within the bounding volume between points 1 and 2. We then proceed to the bounding volume between points 3 and 4. Sequential traversal used in a SOP structure is shown in Figure 50(b). As in the previous example, we start examining each region from the one that is closest to the ray origin, except this time we don't have to worry about the overlapping problem.

Recursive traversal can also be used in both OOP structure and SOP structure, as depicted in Figure 50(c) and (d). The idea of recursive method is to *zoom* in and out within a region. Therefore, it is only suitable for hierarchical structures. To traverse a bounding volume hierarchy, we need to examine the outer bounding volume first. If the ray intersects the outer bounding volume, we zoom in to the inner bounding volume and continue our intersection tests there. Similar idea can be applied to SOP structures, as illustrated in Figure 50(d).

References

[1] M. Abrash. *Graphics Programming Black Book*. Goriolis Group Inc, Scottsdale, Arizona, 1997. 7.1.1

[2] P.K. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 575–598. CRC Press LLC, NY, 1997. 7.1.2

[3] P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. *Advances in Discrete and Comput. Geom.*, 1998. 7.1.2

[4] P.K. Agarwal, L.J. Guibas, T.M. Murali, and J.S. Vitter. Cylindrical static and kinetic binary space partitions. *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 39–48, 1997. 2

[5] J.R. Van Aken and M. Novak. Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics*, 4(2):147–169, April 1985. 4.3

[6] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *EUROGRAPHICS '87, Conference Proceedings*, pages 3–10, 1987. 2, 4.3, 4.3

[7] A. Appel. Some techniques for shading machine renderings for solids. In *AFIPS Joint Computer Conference Proceedings*, volume 32, pages 37–45, Spring 1968. 1, 3.1

[8] S. Ar, B. Chazelle, and A. Tal. Self-customized BSP trees for collision detection. *Computational Geometry - Theory and Applications*, pages 23–29, 2000. Special issue on computational geometry in virtual reality. 7.1.1, 7.1.1, IV

[9] B. Arnaldi, T. Priol, and K. Bouatough. A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer*, 3:98–108, 1987. 2, 7.1.2, 7.1.2, 7.1.2

[10] B. Aronov and S. Fortune. Approximating minimum weight triangulations in three dimensions. *Discrete Comput. Geom.*, 021(04):527–549, March 1999. 7.2, 7.2.1, 7.2.2

[11] J. Arvo. Linear-time voxel walking for octrees. *Ray Tracing News*, 1(2), March 1988. `http://www.acm.org/tog/resources/RTNews/html/rtnnews2d.html##art5`. 7.1.2

[12] J. Arvo. Ray tracing with meta-hierarchies. *SIGGRAPH '90 Advanced Topics in Ray Tracing course notes*, August 1990. 8.2

[13] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Morgan Kaufmann Publishers, Inc., 1989. 3.3, 5.1, 6.2

[14] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal. BOXTREE: a hierarchical representation for surfaces in 3D. In J. Rossignac and F. Sillion, editors, *EUROGRAPHICS '96*, volume 15(3), pages C387–C396. EuroGraphics Association, 1996. 3.3

[15] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 82–91, 1999. 3.3, IV

[16] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. 7.1.2, 7.1.2

[17] J.L. Bentley. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979. 7.1.2

[18] S. Bespamyatnikh and M. Segal. Covering a set of points by two axis-parallel boxes. *Information Processing Letters*, 75(3):95–100, 2000. IV

[19] J. Bittner. Hierarchical techniques for visibility determination. Postgraduate study report DS-005, Dept. of Computer Science and Engineering, CTU Prague, March 1999. 2

[20] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. 4.3

[21] T. Cassen, K.R. Subramanian, and Z. Michalewicz. Near-optimal construction of partitioning trees using evolutionary techniques. In *Proc. of Graphics Interface '95*, May 16–19, 1995. 7.1.1, 7.1.2

[22] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: a new solution for ray tracing complex scenes. *Computer Graphics Forum*, 14(3):C–371, 1995. 4, 7.3.1, 7.3.2, 7.3.3

[23] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray-tracing. In *In Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 11–20, 1997. 4, 7.3.1, 7.3.2, 7.3.3

[24] Pai-Lan Chen. Ray tracing octrees via interpolating artificial normals on boundary surfaces. Master's thesis, National Tsing Hua University, Hsinchu, Taiwan, June 1992. 7.2.2

[25] S. W. Cheng and T. K. Dey. Approximate minimum weight Steiner triangulation in three dimensions. *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 205–214, 1999. 7.2

[26] J.H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976. 2

[27] D. Cohen and Z. Sheffer. Proximity clouds - an acceleration technique for 3d grid traversal. *The Visual Computer*, 11:27–38, 1994. 4.2, 7, 4.3, 5.2, 7.3.3

[28] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–138, 1979. 7.1.2

[29] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1992. Sixth printing. 7.3.1

[30] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994. 7.1.1

[31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin Heidelberg, Germany, 1997. 2, 7.1.1, 7.1.2, 7.1.2

[32] J. Delfosse, W.T. Hewitt, and M. Mériaux. An investigation of discrete ray-tracing. *4th Discrete Geometry in Computer Imagery Conference*, pages 65–76, 1994. 4.3

[33] R. Descartes. Discours de la méthode. in Oeuvres I-XII, C. Adam and P. Tannery and L. Cerf (eds.), 1897-1910. 1

[34] O. Devillers. The macro-regions: an efficient space subdivision structure for ray tracing. *Proc. EUROGRAPHICS '89*, pages 27–38, 1989. 1, 4.2

[35] C.A. Duncan, M.T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: combining the advantages of k-d trees and octrees. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, 1999. 8.2

[36] P. Dutré. Global illumination compendium, July 14 2000. `http://www.graphics.cornell.edu/~phil/GI`. 2

[37] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Adisson-Wesley Publishing, Inc., 2nd edition, 1996. 2

[38] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, and R.L. Phillips. *Introduction to Computer Graphics*. Adisson-Wesley Publishing, Inc., 1994. 2, 4.3

[39] A. Formella and C. Gill. Ray tracing: a quantitative analysis and a new practical algorithm. *The Visual Computer*, 11(9):465–474, 1995. 8.1, 45, 8.2

[40] Steve Fortune. Personal communication. NYU Geometry Day, November 17, 2000. IV

[41] A. Fournier and P. Poulin. A ray tracing accelerator based on a hierarchy of 1D sorted lists. In *Proceedings of Graphics Interface '93*, pages 53–61, Toronto, Ontario, May 1993. Canadian Information Processing Society. 1

[42] W.R. Franklin and V. Akman. Octree data structures and creation by stacking. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Generated Images, State of the Art*, pages 176–185. Springer-Verlag, Toykyo, 1985. 7.2

[43] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977. 7.1.2

[44] F.S. Hill, Jr. *Computer Graphics Using OpenGL*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2000. 6.3

[45] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):124–133, July 1980. 7.1.1

[46] A. Fujimoto and K. Iwata. Accelerated ray tracing. In T.L. Kunii, editor, *Computer Graphics: Visual Technology and Art: Proceedings of Computer Graphics Tokyo '85*, pages 41–65. Springer-Verlag, New York, 1985. 4.1, 4.2, 4.3, 4.3, 5.2, 7.2, 7.3.1, IV

[47] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6:16–26, 1986. 4.2, 4.3, 4.3, 5.2, 7.3.3, 8.2

[48] M. Gigante. Accelerated ray tracing using non-uniform grids. *Proceedings of Ausgraph '90*, pages 157–163, September 1990. 4.3

[49] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 15–22, October 1984. 7.2, 7.2.1, 7.2.2, 7.2.2, 7.2.2, IV

[50] A.S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988. 7.2, 8.1, 8.2

[51] A.S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann Publishers, Inc., 1989. 1

[52] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987. 8.1, 8.2

[53] M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in JAVA*. John Wiley & Sons, Inc., 1998. 6.3

[54] D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Animation*, 11(9):79–85, September 1991. 7.1.1

[55] S. Gottschalk, M. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 171–180, 1996. 3.3

[56] E. Haine. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics & Applications*, 6(9):6–16, September 1986. IV

[57] E. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987. 1, 2

[58] E. Haines. Efficiency improvements for hierarchy traversal in ray tracing. In J. Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991. 6.3

[59] E. Haines. *Standard procedural database*. 3D/Eye, 1992. version 3.13, `http://www.acm.org/tog/resources/SPD/overview.html`. 1, 8.2

[60] P. Hanrahan. A survey of ray-surface intersection algorithms. In A.S. Glassner, editor, *An Introduction to Ray Tracing*. Morgan Kaufmann Publishers, Inc., 1989. 3.1, 7.1.2, IV

[61] V. Havran, J. Bittner, and J. Žára. Ray tracing with rope trees. *14th Spring Conference on Computer Graphics*, pages 130–140, April 1998. ISBN 80-223-0837-4. 7.1.2

[62] V. Havran and F. Sixta. Comparison of hierarchical grids. *Ray Tracing News*, 12(1), June 1999. `http://www.acm.org/tog/resources/RTNews/html/rtnv12n1.html##art3`. 7.3.3, 8.2, 8.2

[63] P.S. Heckbert. Writing a ray tracer. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 263–294. Morgan Kaufmann Publishers, Inc., 1989. 8.2

[64] P.K. Hsiung and R. Thibadeau. Accelerating ARTS. *The Visual Computer*, 8:181–190, 1992. 7.2.2, 7.3.1, 7.3.2, 7.3.3

[65] G.M. Hunter. *Efficient Computation and Data Structures for Graphics.* Ph.D dissertation, Princeton University, 1978. 2

[66] Silicon Graphics Inc. BSP tree frequently asked questions. `http://reality.sgi.com/bspfaq/`. 7.1.1

[67] A. James. *Binary Space Partitioning for Accelerated Hidden Surface Removal and Rendering of Static Environments.* Ph.D dissertation., University of East Anglia, August 1999. 7.1.1

[68] E. Jansen. Comparison of ray traversal methods. *Ray Tracing News*, 7(2), Febuary 1994. `http://www.acm.org/tog/resources/RTNews/html/rtnv7n2.html##art6`. 8.2

[69] E. Jansen and W. de Leeuw. Recursive ray traversal. *Ray Tracing News*, 5(1), July 1992. `http://www.acm.org/tog/resources/RTNews/html/rtnv5n1.html##art5`. 8.2

[70] F.W. Jansen. Data structures for ray tracing. *Data Structures for Raster Graphics, EUROGRAPHICS seminar*, pages 57–73, 1986. 2, IV, 50

[71] J. T. Kajiya. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, July 1983. 3.3

[72] M.R. Kaplan. The use of spatial coherence in ray tracing. *Techniques for Computer Graphics*, pages 173–193, 1987. 7.1.2, 7.1.2, 7.1.2

[73] T.L. Kay and J.T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, November 1986. 1, 3.2, 5.1, 6.2, 20, 6.3, 6.3, 21, 6.3, IV

[74] D. Kirk and J. Arvo. The ray tracing kernel. *Proceedings of Ausgraph '88*, pages 75–82, 1988. 8.1, 8.2

[75] K. Klimaszewski and T.W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, Jan.-Feb. 1997. 1, 5.3, 8.2, 8.2

[76] K. Klimaszewski, A. Woo, F. Cazals, and E. Haines. Additional notes on nested grids. *Ray Tracing News*, 10(3), 1997. `http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html##art8`. 7.3.3, 8.2

[77] J. Klosowski, M. Held, J.S.B. Mitchell, K. Zikan, and H. Sowizral. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, 1998. 3.2

[78] C. Kolb and R. Bogart. Rayshade 4.0, 91. `http://graphics.stanford.edu/~cek/rayshade/rayshade.html`. 2, 4.3, 4.3, 8.2

[79] Stanford University Computer Graphics Laboratory. Dragon, 2000. `http://www-graphics.stanford.edu/`. 3.1, 5.1

[80] John Lakos. *Large-Scale C++ Software Design.* Addison Wesley, 1996. ISBN 0-201-63362-0. IV

[81] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990. 7.2.1, 7.2.2, 39, 7.2.2

[82] J.D. MacDonald and K.S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990. 7.1.2, 7.1.2, 7.2.1, 7.2.2, 8.1

[83] B.F.J. Manly. Multivariate statistical methods. *Chapman and Hall*, 1986. 3.3

[84] T. Moller and E. Haines. *Real-time rendering.* A K Peters, Natick, MA, 1999. 7.1.1

[85] T.M. Murali. *Efficient Hidden-Surface Removal in Theory and in Practice.* Ph.D dissertation., Brown University, Providence, RI, May 1999. 7.1.1

[86] B.F. Naylor. Interactive solid geometry via partitioning trees. *Proc. of Graphics Interface '92*, pages 11–18, June 1992. 7.1.1

[87] Persistence of Vision. POV-Ray 3.1, 1999. http://www.povray.org/. 2

[88] J. O'Rourke. Finding minimal enclosing boxes. *International Journal of Computer Information Science*, 14:183–199, June 1985. 3.3, IV

[89] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, and C. Hansen. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3), July-September 1999. 4.3

[90] M.S. Paterson and F.F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5:485–503, 1990. 7.1.1

[91] M. Pellegrini. Ray shooting and lines in space. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 599–614. CRC Press LLC, NY, 1997. 2

[92] Q. Peng, Y. Zhu, and Y. Liang. A fast ray tracing algorithm using space indexing techniques. In G. Maréchal, editor, *EUROGRAPHICS '87*, pages 11–23. Elsevier Science Publishers B. V., North-Holland, 1987. 7.2.1, 7.2.2, 7.2.2

[93] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques*, pages 101–108, Los Angeles, August 3–8 1997. ACM. 7.3.3

[94] B. Phong. Illumination for computer-generated pictures. *Communications of the ACM*, 18(6):311–317, 1975. 2

[95] B.S.S. Pradhan and A. Mukhopadkhyay. Adaptive cell division for ray tracing. *Computers & Graphics*, 15(4):549–552, 1991. 8.2, 8.2

[96] F.P. Preparata and M.L. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, 1985. 4.3

[97] E. Reinhard, A.J.F. Kok, and F.W. Jansen. Cost prediction in ray tracing. In P. Hanrahan and W. Purgathofer et. al., editors, *Rendering Techniques '97*, pages 42–51. Porto, Portugal, 1996. 7.2

[98] J. Revelles, C. Urena, and M. Lastra. An efficient parametric algorithm for octree traversal. *The 8-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media'2000*, pages 212–219, February 2000. 7.2.1, 7.2.2, 41

[99] J.T. Robinson. The k-D-B-tree: a search structure for large multidimensional dynamic indexes. *ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981. 7.1.2

[100] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989. 2, 3.3, 7.1.2, 7.1.2, 7.2.2

[101] H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics*, 13(4):445–460, 1989. 7.2.1, 7.2.2, 39, 7.2.2

[102] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1990. 2, 7.2.2, 1, 2, 3, 4, 39, 7.2.2

[103] J. Sandor. Octree data structures and perspective imagery. *Computers & Graphics*, 9(4):393–405, 1985. 2, 7.2.1, 7.2.2, 39, 7.2.2

[104] I. Scherson and E. Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, December 1987. 8.1, 8.2, IV

[105] P. Shirley, K. Sung, and W. Brown. A ray tracing framework for global illumination. *Proc. of Graphics Interface '91*, pages 117–128, June 1991. 8.2

[106] B. Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998. 6.2, 6.3, 22, 23, 24

[107] ID Software. DOOM, 2000. http://www.idsoftware.com. 7.1.1

[108] J. Spackman and P. Willis. The SMART navigation of a ray through an oct-tree. *Computers & Graphics*, 15(2):185–194, 1991. 7.2.1, 7.2.2, 7.2.2

[109] N. Stolte and R. Caubet. Discrete ray-tracing high resolution 3d grids. *WSCG '95*, pages 300–312, 1995. 8.2

[110] K.R. Subramanian. *Adapting Search Structures to Scene Characteristics for Ray Tracing.* Ph.D dissertation., University of Texas at Austin, December 1990. 7.1.2, 7.1.2, 7.1.2, 8.1, 8.1, 8.2, IV

[111] K.R. Subramanian and D.S. Fussell. Factors affecting performance of ray tracing hierarchies. Tr-90-21, University of Texas at Austin, August 1990. 7.1.2, 7.1.2, 8.1, 8.1

[112] K.R. Subramanian and D.S. Fussell. Automatic termination criteria for ray tracing hierarchies. In *Proc. of Graphics Interface '91*, June 3-7 1991. 1, 7.1.2

[113] K. Sung. A DDA octree traversal algorithm for ray tracing. In *Eurographics'91*, pages 73–85, North Holland-Elsevier, September 1991. Morgan Kaufmann Publishers, Inc. ISBN 0444-89096-3. 8.2

[114] S.W. Wang and A.E. Kaufman. Volume sampled voxelization of geometric primitives. *Proc. IEEE Conference on Visualization*, pages 78–84, 1993. 4.3, 4.3

[115] A. Watt. *3D Computer Graphics.* Addison-Wesley, 1993. 1, 2

[116] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice.* Addison-Wesley, 1992. 1

[117] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984. 3.1, 3.2, 6.2, 8.2, 8.2

[118] M. A. Weiss. *Data Structures & Algorithm Analysis in JAVA*. Addison Wesley Longman, Inc., 1999. 6.3

[119] K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Choand, C. M. Park, and I. Y. Song. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Trans. Visual and Comp. Graphics*, 1:343–349, 1995. 7.1.2, 7.2, 7.2.1, 7.2.2

[120] T. Whitted. An improved illumination model for shading display. *Communications of the ACM*, 23(6):343–349, 1980. 1, 2, 2, 3.1, IV

[121] N. Wilt and E. Haines. Oort - object oriented ray tracer. *Ray Tracing News*, 7(2), Febuary 1994. `http://www.acm.org/tog/resources/RTNews/html/rtnv7n2.html##art4`. 8.2

[122] A. Woo. Fast ray-box intersection. In A.S. Glassner, editor, *Graphics Gems*, pages 395–396. Academic Press, 1990. 7.1.1

[123] A. Woo. Recursive grids and ray bounding box comments and timings. *Ray Tracing News*, 10(3), December 2 1997. `http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html##art9`. 8.2

[124] X. Wu. A linear-time simple bounding volume algorithm. In D. Kirk, editor, *Graphics Gems III*, pages 301–306. Academic Press, 1992. 3.3

[125] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer graphics and applications*, 12(5):19–28, September 1992. 4.2, 4.3, 4.3, 5.2, 7.3.3

[126] S. Youssef. A new algorithm for object oriented ray tracing. *Computer Vision, Graphics, and Image Processing*, 34:125–137, 1986. 3.3

[127] G. Zachmann and W. Felger. The BoxTree: Enabling real-time and exact collision detection of arbitrary polyhedra. *First Workshop on Simulation and Interaction in Virtual Environments*, pages 104–113, July 1995. 3.3

[128] Y. Zhou and S. Suri. Analysis of a bounding boxes heuristic for object intersection. *Journal of the ACM*, 46(6):833–857, November 1999. IV