# Algorithms For Accelerating Ray Tracing

David G. Hook

May 29, 1995

## Abstract

Ray tracing is a technique which probably traces its origins back to Galileo in his investigations in optics. The method has now been adapted to a variety of problems in physics, engineering and computer graphics. Central to the problem of a timely solution to using ray tracing to investigate a group of objects is being able to eliminate objects which the ray does not intersect, as, computationally, intersecting objects ranges from expensive to very expensive.

A group of algorithms and approaches which have been used in the past for accelerating ray tracing are presented, together with the lessons learned. A newer method, referred to as the bv-tree, is introduced. The bv-tree is adaptive and experimental results presented indicate that it often leads to less intersection tests being carried out than with the other methods examined.

# Declaration

This thesis does not contain any material that has been accepted for the award of any other degree or diploma in any university and to the best of my knowledge and belief, the thesis contains no material published or written by another person, except when due reference is made in the text of the thesis.

D. G. Hook

To the Echidna in us all.

# Contents

# Chapter 1

# Introduction

In the past 20 years there have been major advances in the area of computer graphics, both in terms of the objects and phenomena that can be modeled, and the stage has been reached where researchers, and the general public alike, are quite happy to apply the term photorealistic to some of the images being generated.

The two basic requirements for the generation of realistic imagery are the removal of hidden surfaces in the scene being visualised and the shading of those surfaces that are visible. Ray tracing is a technique that has been used to produce some of the most photorealistic images to date. It is conceptually simple as it involves simple vector algebra, yet it is quite powerful and allows for the simulation of lighting effects due to specular reflection, refraction and even soft shadowing. Ray tracing has also been proposed as a solution to the problem of handling diffuse illumination and caustics generated by light being focussed by glass, and although it is now felt diffuse illumination can be better handled by radiosity methods, the problem of generating photo-realistic images still relies on hybrid methods, such as discussed in the work of Chen *et al.* [19] or more recently Kok *et al.* in [54].

Ray tracing has applications in areas other than graphics and has been applied to problems such as the generation of depth information for computer modeled environments together with calculating the mass property values for complex solids such as those described using Constructive Solid Geometry (CSG) and recent work by Menon, Mavisin and Zagojac in [67] demonstrates that ray representations of solid models simplify the geometrical calculations required for solid modeling as well as extending the geometrical coverage of the solid modeling system over that of the more traditional representations. As techniques applicable to problems such as these are always also applicable to graphics, this thesis is written with computer graphics in mind. Readers wanting more information on non-graphic uses are invited to check the work of Roth [78] and the authors mentioned above.

## 1.1 The Ray Tracing Problem

Whatever the application ray tracing is being used for, the fundamental idea is that a ray is traced from some origin into the object space until it either intersects something, or travels far enough. Traveling far enough may mean either leaving the object space altogether, or in the case of computer graphics, a ray testing whether or not an object is between another object and a light source need only be traced the distance of the object from the original light source.

Figure 1 illustrates the ray tracing problem. Rays are normally classified into three different types: *primary*, *secondary*, and *shadow*. A *primary* ray is generated at the eye and traced through a pixel in the image plane into the object space. If a ray hits an object, normally, more rays are cast. *Shadow* rays, originating at the surface of the object, are cast in the direction of each light source. *Secondary* rays may also be generated if the surface properties of the object require it. Secondary rays are either *reflected* or *refracted* (or *transmitted*) rays. This process is often recursive, with each secondary ray resulting in further shadow and secondary rays being generated, until some termination criteria is met. Shadow rays are distinguished from other secondary rays as there are a range of optimisations that are specific to them, due mainly to

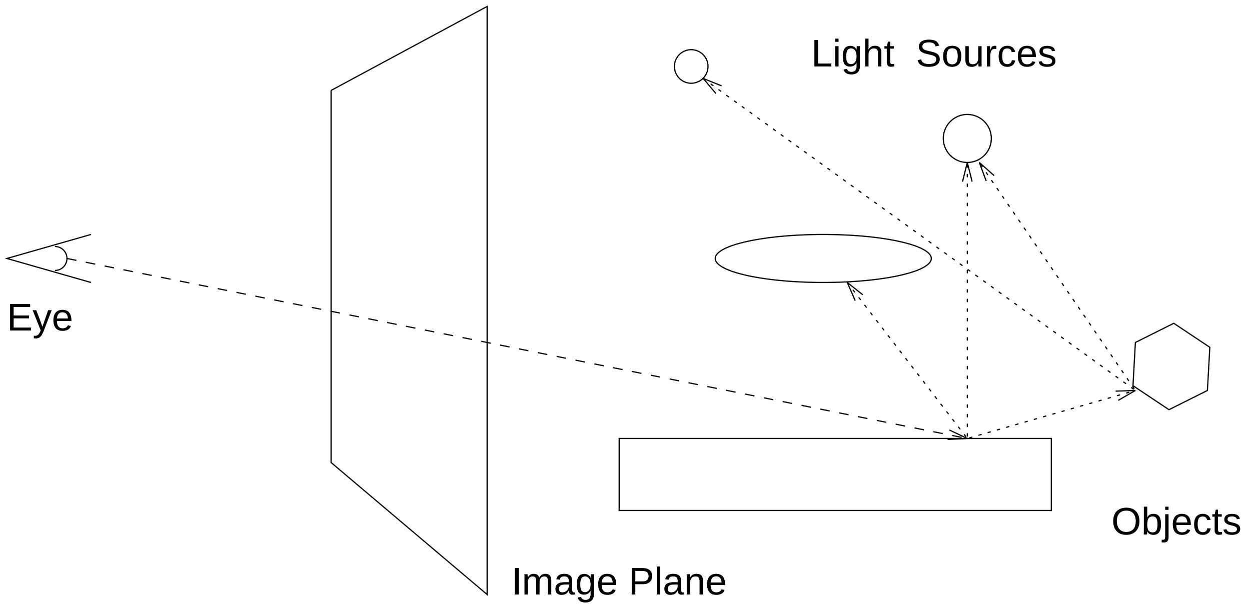



Figure 1: The path of a ray and the rays produced by it.

the fact that all that is required for a shadow is that there is a solid object between the light and the surface: the object casting the shadow need not be the closest one. Other non-primary rays are essentially random, so require general optimisation techniques to be used. Glassner *et al.* [37] provides a thorough introduction to the principles of ray tracing.

As the intersection of the ray with individual objects is often quite costly, it is desirable to intersect a ray with an object only if there is some likelihood that the ray will hit it. Since ray tracing's first appearance in computer graphics with the papers of Whitted [104] and Roth [78] schemes for accelerating ray tracing have been the subject of much research.

## 1.2 The Structure and Motivation of This Thesis

At the moment there doesn't exist any definitive answer to the problem of what is the fastest way to process a given collection of objects with ray tracing. To further complicate matters there are some algorithms which run only on particular architectures. The aim of the work described here is three fold. Firstly, to survey existing algorithms, both non machine specific and machine specific to see what can be learned. Secondly, to describe a non machine specific algorithm which appears to have fewer worst cases than existing algorithms. Thirdly, to develop a library of routines to enable easy comparison between differing acceleration techniques so that further research into acceleration methods will be easier to perform.

As the most general case in ray tracing are the secondary rays, which are essentially random, the major part of this thesis is devoted to accelerating the tracing of secondary rays. A brief survey of techniques for accelerating shadow and primary rays is given in the rest of this introduction.

Work done on accelerating ray casting in the more general case is surveyed both in reference to techniques that can be used on computers without parallel or vector architectures and techniques which have been developed to run on specialised computer architectures. This is done in the second and third chapters respectively.

Most of the techniques that are used with specialised computer architectures are based on the techniques developed for general computers, consequently, the design and use of specialised hardware has been left outside the scope of this work, and the bv-tree algorithm presented and tested at in the later chapters of this thesis was developed with general computer hardware in mind. The bv-tree algorithm is presented in the fourth chapter, and the results of its use are presented in the fifth chapter.

The results are presented using the most common acceleration techniques and the most common benchmarks for ray tracing. This was done so that there would be common ground for comparisons with work by other researchers who use these benchmarks and the results described here. In addition, the RT library, described in Appendix B, and developed during the course of the development of this thesis, is easily extendible, the addition of a new acceleration technique requiring only the addition of a couple of functions, and will also be useful to researchers wishing to do further work in the area.

Finally, in chapter six, some conclusions are presented concerning the bv-tree algorithm, algorithm development in this area in general, and some suggestions concerning future research to answer questions that this thesis has uncovered.

The work done in the development of this thesis also yielded some new insights into constructing kd-trees for ray tracing. Appendix A contains a paper which was written concerning this. Appendix B contains the manual page for the RT library which was developed for the running of the benchmarks presented in chapter 5, and appendix C contains a sample renderer written to read the files created by the benchmark generators.

## 1.3 Ray Tracing Acceleration Methods

Schemes for accelerating ray tracing have been concentrated on the following considerations: (1) reducing the amount of time spent to intersect a ray with an object, (2) techniques to reduce the number of rays traced, and (3) techniques to reduce the number of objects that need to be intersected.

While all approaches have their uses the first one has the disadvantage that it is specific to certain objects, the other approaches are applicable to all scenes although their effectiveness will vary according to the scene being generated and the conditions under which it is being done (amount of antialiasing, depth of reflection or transparency, etc). This is especially true of the second approach.

In this thesis the third approach is concentrated on while the first two approaches are briefly surveyed. Techniques that reduce the number of intersection tests can be applied across the full range of ray types that might be generated in analysing a data base of objects. The following sections provide a brief overview of methods used to optimise for specific ray types.

### 1.3.1 Intersecting Objects.

Ray casting allows for many different types of objects to be intersected, ranging from polygons to quartic or even arbitrary order surfaces. More complicated surfaces such as fractals, heightfields, CSG models can also be dealt with.

Many techniques have been published in the literature on object intersection. Roman Kuchkuda's paper [55] and the two books edited by Andrew Glassner [36] and [37] more than adequately cover the area.

One problem which has also been looked at recently is the problem of intersecting objects that are in motion during the time represented by the state of the objects in the final picture. Maurel *et al* in [65] propose an algorithm based on 4D geometry which allows for moving objects and the effects of moving light sources to be rendered at less cost than previously while still allowing the use of other acceleration techniques as well.

### 1.3.2 Accelerating the Primary Trace.

Techniques for accelerating the primary ray trace often involve reducing the number of rays generated. This can be done either by trying to eliminate rays which will not provide a major contribution to the image or trying to eliminate areas of the viewplane where rays will not hit objects.

The undersampling techniques of Akimoto [1] is an example of a method to reduce

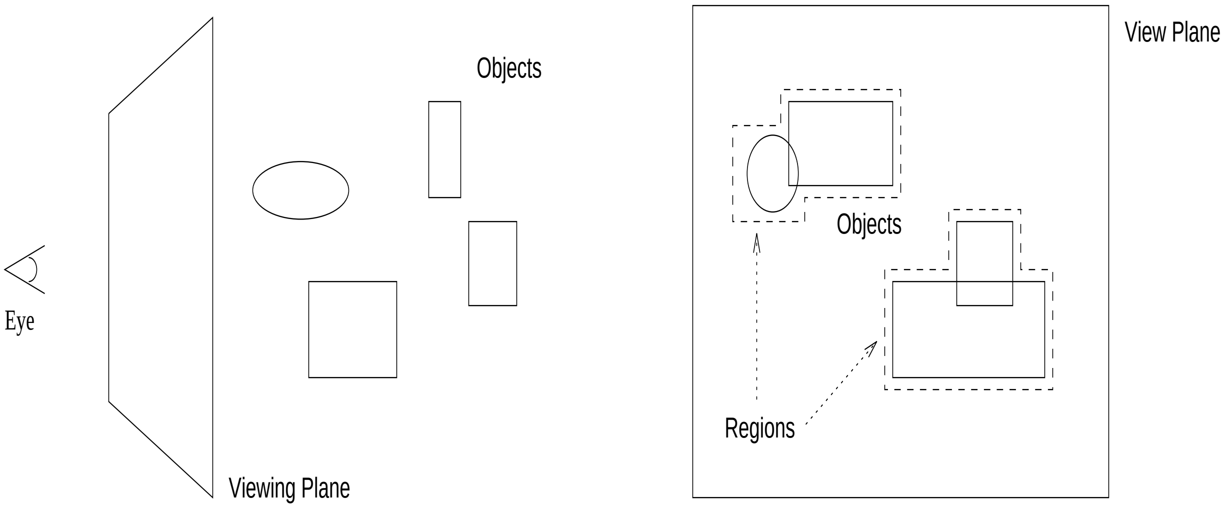



Figure 2: A sample scene and its resultant projection into screen space.

the number of primary rays cast, based on the variance in colour from subpixel to subpixel. Salesin and Stolfi [80] also discuss this as it relates to CSG modeling.

Amanatides [3] classifies primary rays into cones, and first tests an object against a given ray's corresponding cone, before testing it against the ray. The algorithm, as described, creates cones using a recursive subdivision of the screen, until either the cone is covering a single pixel, or there are few or no objects in the cone.

Ritter [77] projects the objects in the model space into the screen coordinates of the image. Only areas of the screen covered by objects are then tested. Although this algorithm may sound similar in intent to the previous one, it reduces the problem to 2D and allows the use of integer arithmetic in selecting candidate objects in addition to limiting the regions of the viewplane that need to be tested (see Figure 2). In the experience of the author this is the most effective way of dealing with primary rays.

Weghorst, Hooper, and Greenberg [98] eliminate the generation of primary rays altogether by handling the first pass using a Z-Buffer. This still constitutes an approach to ray tracing as secondary and shadow rays are then generated and traced using one of the other methods.

## 1.4 Accelerating the Shadow Trace.

Often scenes to be rendered have more than one light source. A lot of the time this is due to the absence of a proper modeling of the radiated diffuse light in the scene as illustrated by the work of Goral, *et al.* [39], so extra lights are required to give the correct effect, and also, especially in commercial applications, it is due the fact that in interiors and the like there are more than one light source. As this is the case, shadow rays tend to make up quite a high percentage of the rays generated in rendering a scene, and they often outnumber the primary rays.

Haines and Greenberg [42] describe the light buffer. An imaginary cube is placed around the light source and candidate lists of objects likely to be causing shadows in each direction are built up. The faces of the cube can be subdivided to simplify the candidate lists as required. Shadow testing is done by finding which part of which face of the cube surrounding the object the shadow ray pierces and then testing the objects in the appropriate candidate list. Ohta and Maekawa [70] discuss a similar idea. Haines and Greenberg also introduce the *shadow cache*, where the last shadow causing object is tested before any others are looked at, thus exploiting the coherence that often occurs between shadow tests on the same light.

Woo and Amantides [106] describe a shadow acceleration scheme based around uniform subdivision, in which the scene is divided into many disjoint 3D elements, or voxels. Each voxel in the subdivision has a flag allocated for each light source indicating whether or not the voxel is occluded by objects between the light and the voxel. If a voxel is totally occluded or totally nonoccluded no shadow rays need to be traced. If the voxel is partially occluded the shadow tracing proceeds as normal.

Woo [107] also presents two methods that can be done on the fly for accelerating shadow testing. Basically they rely on adaptive sampling to decrease the number of rays cast in shadow evaluation and the generation of visual effects such as penumbra.

Wilson in [103] proposed the use of an “inner tube” volume to speed up shadow

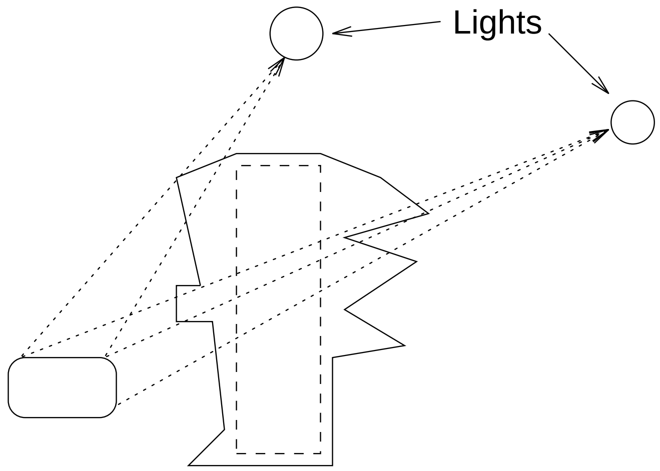



Figure 3: An example "inner tube" for a complex object, rays which pass through the rectangle do not need to be tested against the surrounding object.

testing. The idea in this case is to have a simpler volume that is completely inside an object being tested by a shadow ray. In this case if the ray pierces the inner tube the shadow ray must hit the object, and consequently the light must be occluded. As an example of how this might apply to a complicated object, see Figure 3.

Ward in [97] tries to reduce the number of shadow rays that need to be traced by ignoring light sources whose contribution to the illumination of the surface is regarded as insignificant. At the start of a shadow trace, each light is assigned a contributing value and processed from highest to lowest. When the contributing value gets below a certain threshold the remaining lights are ignored.

Woo, Perlin, and Fournier's article [106] provide a thorough survey of shadow algorithms for ray tracing as well as other image synthesis techniques.

## 1.5 Conclusions

There are many techniques for accelerating the tracing of specific types of rays, but with only a few exceptions they attempt to eliminate unnecessary rays before allowing a normal ray trace to proceed. While this elimination of unnecessary rays contributes to the speed with which a scene can be rendered, the problems of resolving reflective and refractive rays are still present, as is the problem of efficiently tracing a ray that has made it through the elimination process. For this reason the problem of handling a general ray is still significant, especially if a scene has many reflective, and refractive surfaces. The problem of finding a general acceleration scheme still remains.

# Chapter 2

# General Acceleration Schemes

Techniques suitable for all types of rays have to take into account the fact that secondary rays are essentially random, both in direction and origin. This generally involves building a data structure for the objects that aids in avoiding intersection tests against objects that are not on the path of the ray being traced, while allowing rapid isolation of objects that do need to be tested. The most common techniques involve either subdividing space either into a grid pattern or some type of hierarchy, such as a bounding volume hierarchy, or classifying the rays and building up object lists for the various classes of rays. There are many variations on each of these methods.

## 2.1 Voxel Based Methods

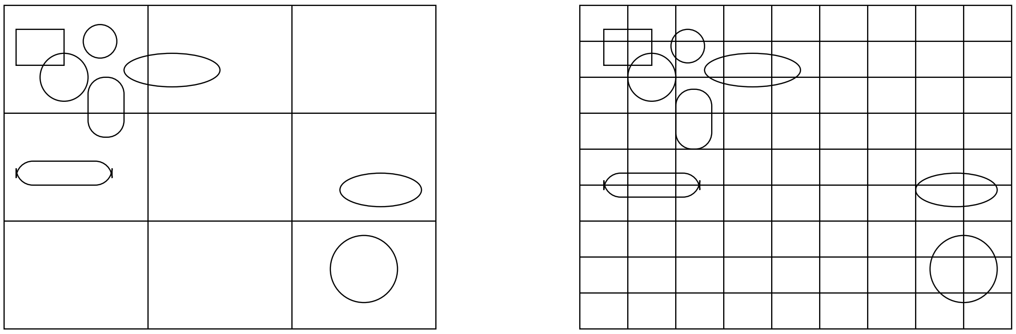
In voxel based methods the scene is divided into many disjoint 3D elements, or voxels. As a ray is traced through the scene voxels are visited along the ray's path and the objects within each voxel are intersected by the ray. Some extra heuristics which can be applied to voxel based methods are given in [63].

The major problem with voxels is that an object may exist in more than one voxel. If a ray hits an object in a voxel the intersection point of the object and the ray may be outside the voxel, and there may be an object in the next voxel along that lies on the path of the ray that is closer than the other intersection point. This can be dealt with by forcing the trace to continue until an intersection is found which lies in the current voxel. Unfortunately this then means that we may end up testing the same object against a ray many times before we in fact arrive in the voxel that the intersection of the ray and the object exists in. This is solved by associating a unique identifier with each ray traced and a "mailbox", as proposed by Glassner in [34] with each object. An object is intersected only if the ray identifier in the object's mailbox is not equal to the current ray identifier. In this case the ray identifier and the intersection information are stored in the object's mailbox, so that on later tests with the ray the ray identifier can be matched and the intersection information retrieved.

### 2.1.1 The Uniform Grid

A uniform grid is the subdivision of space into an $N \times N \times N$ grid of equally sized voxels. Uniform grids have the advantage that the grid can be traversed rapidly, however, as no account of the density the density of objects in the scene is taken in determining the grid size, many of the voxels in the grid may be empty and others may have a large number of objects in them (see Figure 4). For scenes in which objects are well distributed throughout and the value of $N$ is chosen carefully, a uniform grid tends to outperform other subdivision methods.

One major problem of the uniform grid is that, unfortunately, it is never immediately obvious what the ideal value for N should be. The larger the value of N, the more voxels you have to traverse, on the other hand, if N is too small there will be too many objects in each voxel. There is an optimal value for N and unfortunately a value of N which is either too large or too small can result in the ray tracing taking substantially more time than it would otherwise.



N = 3 N = 9

Figure 4: Differing values of N for the same scenes and the resulting grids.

In addition to this the grid has the disadvantage that it must be constructed before the ray tracing is commenced, consequently the tracing may also suffer due to the overheads of preprocessing and the memory required to store the grid. Some early attempts at using both grids and octrees were orientated around being able to calculate the indexes of the grid cells being traversed without actually having to access the grid cells in memory. Examples of this is the work of Fujimoto *et al* in [30] and Andrew Glassner in [34], which are referred to in the section on octrees below.

Amanatides and Woo [2] describe a 3D grid structure and a very simple algorithm to traverse it. A similar algorithm is also described by Cleary and Wyvill [20].

One area for accelerating the traversal of a uniform grid is to be able to eliminate empty voxels rapidly. Yagel, Cohen, and Kaufman [108] present an algorithm called Raster Ray Tracing (RRT) which is an enhanced double version of Bresenham's 2D line algorithm. This does utilise a grid but allows the empty voxels to be skipped cheaply. Some aliasing problems occur and a solution is proposed. The authors do claim that their initial tests showed an improvement on the ARTS ray tracing algorithm used by Fujimoto *et al* in [30].

### 2.1.2 Non Uniform Grids

There have been several attempts to make grids more adaptive. These have fallen into two categories, those that involve the introduction of a hierarchy, and those that attempt to speed up the traversal of empty grid cells by combining them in some fashion - thus allowing a higher value of N to be chosen.

A hierarchy of grids is described in the work of Jevans and Wyvill [51]. Firstly a root uniform grid is set up and then voxels which are still regarded as having too many objects in them have grids recursively placed in them. This subdivision is done on the fly so that areas in which rays do not pass require no processing.

The idea of combining grid cells was first looked at in Devillers [25] who generated a uniform grid and clustered the empty voxels into rectangular regions, called macro-regions. When a macro-region is entered, the exit point is determined without traversal of the grid voxels within the region. Traversal is then continued at the exit point, which may be in another macro-region. This speeds up the traversal of the grid as many empty voxels can be skipped simultaneously.

A variation on macro-regions is Gigante [33] where a grid structure is set up by the use of frequency buckets in each direction. The count in the bucket is incremented for each object that overlaps the bucket's range. Dividing planes are placed at equal frequencies in each direction. This creates a grid of voxels which can be of unequal size but all of which will be multiples of the size of the original bucket. Traversal of such a grid is slightly more complicated than that of the uniform grid, but the extra cost of traversal which the non-uniformity creates should be more than offset by the gain in speed provided by the rapid traversal of the of the sparsely populated regions.

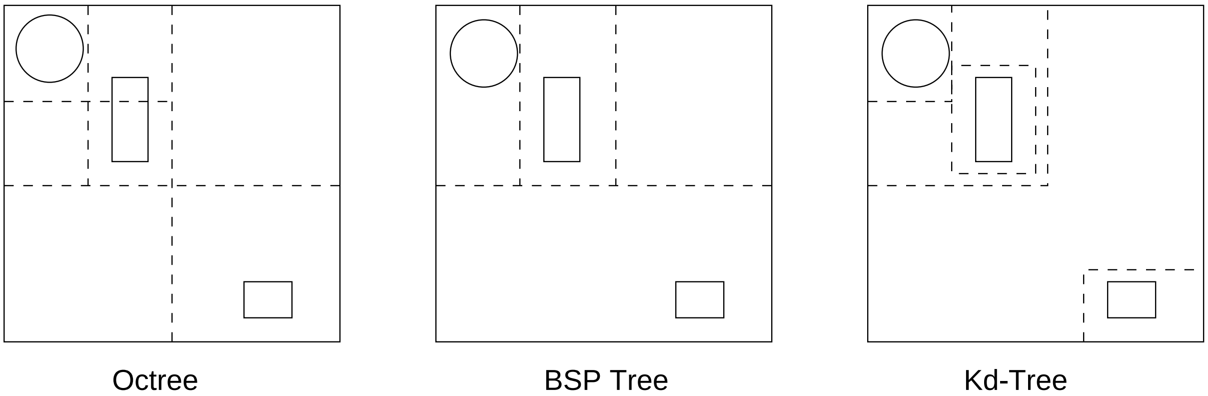



Figure 5: Space contained in the subranches of different trees for the same scene.

## 2.2 Tree based methods

Another popular method for dividing and traversing space is using a spatial partitioning tree (SPT) as the basis of the algorithm for tracing the ray. SPT-based methods can be handled iteratively but more recent research (detailed below) has show that they are very attractive as they allow the traversal to take advantage of a tree based data structure's recursive nature, which often produces a simpler, and in some cases faster algorithm. Figure 5 gives an example of the partitionings expected from the tree based methods detailed below.

McNeill, Shah, Hébert, Lister, and Grimsdale [66] performed a comparative study between BSP-trees, Octrees, and Grids and found that, overall, the octree provided the better performance. They also discovered that adaptive tree structures are in general smaller than a uniform grid of a similar level of efficiency.

### 2.2.1 The BSP-tree

BSP-trees are produced by taking the existing object space and dividing the objects into two groups according to where they lie on some side of a plane. Often the plane is calculated by dividing one of the coordinate axes in two, but not always. The reader should note that in several cases BSP-trees referred to in the literature are actually octrees in disguise, such as Kaplan in [57], and are recorded here under that title.

Arnaldi, Priol, and Bouatouch [14] describe a BSP tree, but employ an associated voxel index grid to improve traversal time. If a ray misses all objects in a voxel the index grid is used to determine which region the ray will enter next. Once the grid has been created the actual BSP tree is no longer necessary, and the traversal of the tree is eliminated at the cost of storing the grid.

In some cases gains can be made if the BSP tree is not partitioned with planes parallel to one of the axes. Such a BSP tree is described in the work of Lin [61]. As with the work of [28] a polygon is chosen, defining a dividing plane. In this case the object space is restricted to be made up of nothing but polygons and the polygons in the object space are clipped against the dividing plane and put into the appropriate half space. This technique turned out to be quite slow, apparently due to the complicated tree traversal.

Another approach to chosing a dividing plane is through the use of heuristics. MacDonald and Booth [63] describe two heuristics for constructing BSP trees, one using surface area as an estimate of probability of, intersection, and another based on the fact that the optimal dividing plane lies between the spatial median and the object median planes. They also improve the efficiency of traversing the BSP tree through better processing of interior nodes, and better computation of the ray exit point.

### 2.2.2 The Octree

An octree is constructed by dividing 3D space into eight voxels. The voxels may be formed by dividing the original region at the midpoint of each axis. The dividing point may also be chosen by some other method, such as a median cut of the object space along one dimension, but the former is more common. Each voxel is recursively subdivided until some predetermined depth is reached or the number of objects within a voxel is less than some predetermined number. The work of Samet [83] provides a complete review of octrees and their applications.

Glassner [34] uses an octree to divide up the object space. The nodes of the octree are kept in a hash table to avoid traversing the octree and also to reduce the memory requirements for storing the octree. Moving to the next voxel is achieved by finding a point that is guaranteed to be within the next voxel and using it as the basis for calculating an index into the hash table. Leaves of the octrees are assigned names that reflect their position in the hierarchy so as to avoid moving up and down the implied tree.

If space is available the octree can also be stored explicitly. Kaplan [57] describes an octree algorithm in which the tree is stored explicitly. He refers to the structure as a Binary Space Partition tree, as each direction is split in half at each level, rather than splitting one level into eight voxels at each step.

One problem of traversing an explicitly stored octree is that of having to recurse up and down the stack when a recursive traversal is used. Arvo [6] describes "recursive" traversal for an octree using an explicit stack. This allows efficient traversal for primary rays as the recursion can be abandoned when an intersection is found. Unfortunately copying the stack to allow the continued traversal for secondary rays can be computationally expensive in some cases.

Another improvement for traversal of an octree is described by Samet in [82, 83], where a traversal algorithm is described which doesn't require descending from the root each time. Traversal is done by moving up and down the tree for successive cells along the ray's path. This algorithm places some limits on the structure of the octree and the ray tracing that can be done: the cells must be cubes, rational integer arithmetic is required, and because of the rational arithmetic there is a limit on the depth of the shade tree for shadow, reflection, or transparency calculations.

Spackman [90] uses an algorithm called SMART (Spatial Measure for Accelerated

Ray Tracing) for traversing an octree. Again the traversal proceeds using integer arithmetic, but in this case the octree is traversed recursively by a series of vertical and horizontal steps with two uniformly bounded SMART vectors, an update vector, and a comparison vector. Index calculations proceed by halving the values down vertical steps and using precalculated values down horizontal steps.

In another case where the tree is not stored explicitly Fujimoto, Tanaka, and Iwata [30] use a 3DDDA (3D Digital Difference Analyser) to traverse an octree. At any non-leaf node there is a $2 \times 2 \times 2$ grid. The 3DDDA is used on this sub-grid until it exits the bounds of the grid. Normal up and down traversal is then used. It turned out this method had little advantage over techniques without the 3DDDA because the vertical traversal is the most computationally expensive function in any method.

Sung [93] also describes a variation of the 3DDDA for octree traversal called DDAOCT. In this case every voxel is guaranteed to have a size which is a multiple of the smallest voxel, thus allowing an efficient traversal algorithm to be developed. The key, in this, case is the construction of Leveled Voxel Space Indexes (LVSIs) to aid the algorithm. However if a leaf node is larger than the smallest voxel it is traversed as if a uniform subdivision grid existed.

Another approach to the storing of an octree is in the work of Peng, Zhu, and Liang [71] who describe an octree with its leaves stored in a one-dimensional array. This is called a linear octree. Internal nodes are not stored at all so vertical traversal is never done. An encoding scheme is used to access the tree via a binary search and to allow for leaf to leaf traversal. While, in their implementation, objects had to be tessellated into triangles, it is quite possible the technique would be applicable to other object types as well.

A couple of hybrid schemes have been proposed using octrees as the base. Scherson and Caspary [84] use a short octree with a hierarchy of bounding volumes inside the leaves. This hybrid technique has the following advantages: fewer cells are traversed in the octree because the voxels are large and the tree is not deep, and fewer bounding volumes are intersected because the upper levels of the hierarchy are not maintained.

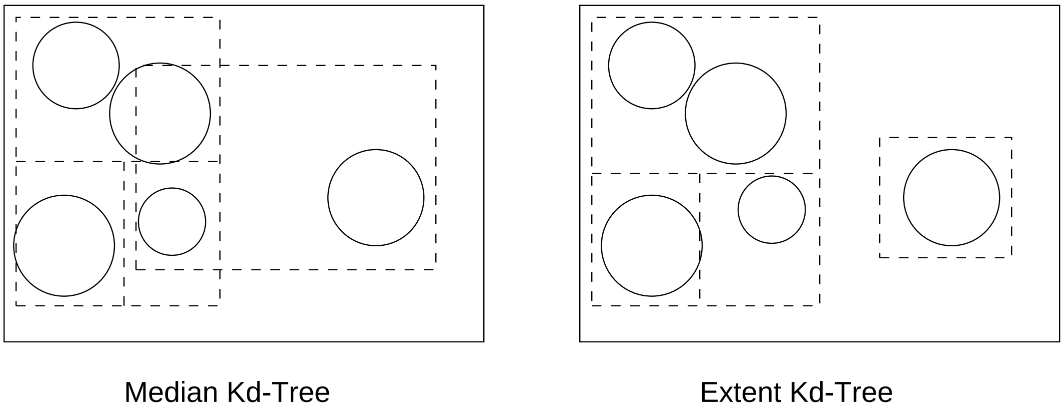



Figure 6: Sub-regions generated by the two kd-tree subdivision approaches.

Unfortunately calculating how deep to make the octree is still an open question.

Glassner [35] also proposed a hybrid scheme combining both bounding volumes and octrees. In this case a subdivision was performed using the octree, and the resulting octree was used to create the bounding volume hierarchy. As the bounding volumes are only meant to contain objects at the appropriate level of the octree some clipping had to be performed (as in [89]). Either the octree or the bounding volume hierarchy may be traversed to trace rays.

### 2.2.3 The Kd-Tree

The Kd-tree was first proposed by Bentley [13]. In a Kd-tree the object space is subdivided along the x, y, or z dimensions at each level of the tree. There is no requirement that the tree subdivide space along different dimensions at each level and two schemes have been put forward for subdividing at each level: a median cut according to the number of objects, and a straight spatial subdivision.

One approach to partitioning space in a kd-tree is to select a partition plane such that an equal number of objects lie on either side of it. Fussell and Subramanian [31] proposed a kd-tree where the subdivision was done using a median cut on the number of objects. This algorithm stored the kd-tree explicitly and allowed for recursive traversal in order to trace the rays. The algorithm was run on scenes from the SPD benchmarks, developed by Eric Haines [42], compared with an octree and found to be favourable.

An alternate strategy to doing the partitioning is to chose the partitioning plane such that it divides the longest extent in the object space. Hook and Forward [48] describe an algorithm which does this and generates a bounding volume hierarchy using a kd-tree as the basis. This partitioning method was was found to result in faster ray tracing than with basing the partitioning around a median cut on the number of objects. As can be seen in Figure 6 the median cut based subdivision produces larger bounding volumes in situations where the objects are not uniformly distributed through object space, thus slowing the algorithm down as higher branches of the kd-tree contain more empty space than a tree generated by dividing along the longest extent. These results are also in line with the work of Weghorst, Hooper, and Greenberg on bounding volume hierarchies, as published in [98].

## 2.3 Bounding Volume Hierarchies

An easy way of reducing the cost of intersecting an object is to surround it with a bounding volume which is much cheaper to intersect. This way the ray will only be tested against the object when there is some chance of success and the marginal increase in the cost of successfully intersecting the object is more than offset by the decrease in the cost of detecting a failure. Bounding volume hierarchies are often used in hybrid methods involving grids or spatial partition trees.

Ideally a bounding volume should be chosen so that it provides a tight fit around the objects it is enclosing. If the bounding volume is not tight, unnecessary intersection calculations will be done on the objects enclosed, on the other hand if it is too tight it may be as comparatively expensive to intersect as the objects it encloses and the point of having a bounding volume is missed. For this reason bounding volumes

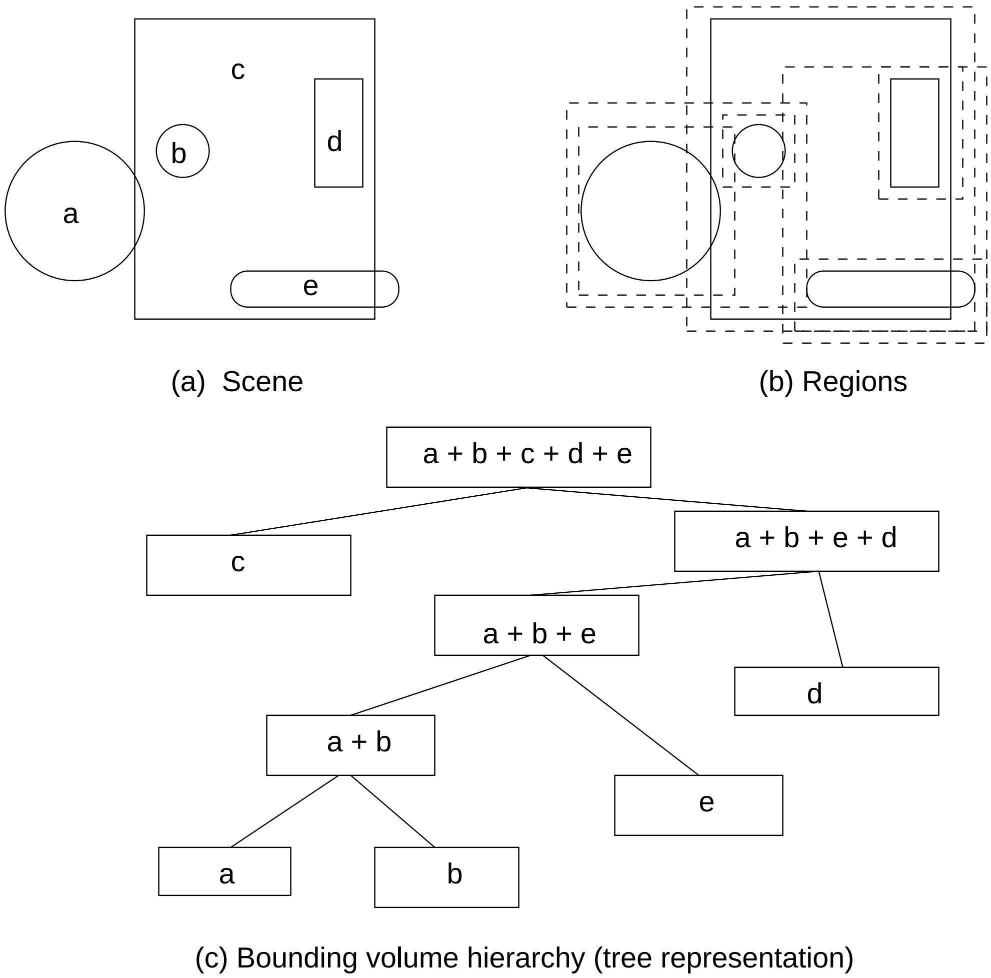



Figure 7: An example scene, bounding regions, and the resulting hierarchy.

are generally made up of spheres, parallelepipeds, or composed of slabs.

Rubin and Whitted [79] describe an object hierarchy consisting of parallelepipeds. In order to intersect a ray with the hierarchical environment, the top level is tested first and then the ray is recursively traced down the hierarchy only along those subbranches whose bounding volume the ray enters. Jansen [50] also describes an algorithm for a similar structure.

Different shaped bounding volumes can also be used. Weghorst, Hooper, and Greenberg [98] describe such a hierarchy where the shape of the bounding volumes is chosen according to the tightness of the fit and the complexity of the bounding volume. Each bounding volume is assigned a complexity factor and the product of the complexity factor with its volume is computed. The bounding volume with the smallest product is the one used to bound the hierarchy so far constructed. This technique did require some input from the modeler as to how to bound objects as well, but their results demonstrated the importance of have bounding volumes which gave a tight fit. Figure 7 shows an example scene, a two dimensional representation of the hierarchy overlaid on the scene and a tree representation of the resulting hierarchy. In this case parallelepipeds have been used for the sake of clarity, but, as mentioned, the algorithm described by Weghorst, *et al.* allows for more complicated bounding volumes to be used.

A method for creating more adaptive bounding volumes has also been described. Kay and Kajiya [53] describe an algorithm employing what they call *bounding slabs* to construct bounding volumes. Increasing the number of slabs bounding an object decreases the amount of empty space in the bounding volume. A tradeoff must be made, however as the more bounding slabs there are the more expensive it becomes to intersect the bounding volume. A hierarchy is constructed using a recursive median cut algorithm.

The idea of using grids and bounding volumes in a hybrid scheme has also been proposed. Synder and Barr [89] describe a hybrid method combining grids, bounding volume hierarchies, and lists with surfaces that are tessellated into triangles. The choice of which method to use on a scene depends on the scenes complexity and after an initial division has been made the complexity of the resulting subscenes is used to decide which method to apply to the subscene. Another key point of their approach is that it is possible to have a bounding volume for a triangle which only contains part of the triangle. This makes the culling of missing triangles faster but can result in the same triangle being compared against a ray more than once in each traversal.

## 2.4 Ray Classification Methods

Some methods of ray classification have being based around trying to find some larger 3D entity which can be used as a wrapper around a set of rays. Only the objects in the scene which lie inside the wrapper need to be checked against the rays it contains. Heckbert and Hanrahan [46] attempt to classify rays into beams. Often rays follow virtually the same path from the eye point to objects, and objects to light sources. So rather than trace individual rays a beam is traced. When different parts of the beam hit different objects the beam is subdivided. This has the advantage of allowing better antialiasing since the object space is no longer point sampled, in addition to allowing for more complicated shadowing. Unfortunately the method is not able to handle refraction very well.

Cylinders have also been used as wrappers. Speer, DeRose, and Barsky [91] detail a method to take advantage of coherence between shade trees. Cylinders are placed around rays in a shade tree such that each cylinder intersects no objects, but are otherwise as big as possible. When generating the next shade tree, the ray is compared against the corresponding cylinder of the previous ray's shade tree. If the new ray doesn't pierce the side of the cylinder then the new ray can be processed in a similar fashion to the old one. Although this technique is theoretically attractive the overhead involved tends to outweigh the savings.

The work of Ohta and Maekawa [70] is based on ray coherence theorems for spheres and convex hulls. The algorithm makes the following assumptions: each object is bounded by a sphere or a convex hull, each of the M possible ray origins is bounded by a sphere or convex hull, space around each ray origin is divided into $D$ directions, and there is a two dimensional array of candidate objects $O[M][D]$. Given the parameters $M$ and $D$, a preprocessing step is used to create $O[M][D]$. Then for each ray, the ray tracing algorithm computes the index j of the ray origin and the index d of the direction, and then intersects those objects in $O[j][d]$. A large number of directions is normally necessary to ensure the list of objects in $O[j][d]$ is small.

Horvath, Marton, Risztics, and Szirmay-Kalos [49] build on the work of Ohta Maekawa to further exploit ray coherence in the visualisation of object databases made up of spheres and convex polyhedra. While theoretically very elegant the pre-processing time required for a rendering sometimes makes the total cost of a rendering greater than before. Another problem is that the geometry currently applicable to the algorithm is restricted to just spheres and convex polyhedra.

As a ray can be described in terms of its origin and two angles giving the direction, it can also be described as a point in 5 dimensional space. Arvo and Kirk [5] describe a method which combines an octree of ray origins with direction cubes to produce an approximation to the 5-D Hypercubes that can be used to represent ray direction and origin as one point. Rays are classified according to origin into an octree which then allows a presorted set of candidate objects to be fetched for that ray's particular direction. The computational cost of intersecting a ray with the object space is still low, despite the required searching and object sorting, because there is normally a high level of coherence in ray tracing a scene and similar rays share the benefit of the initial sort. The technique also uses lazy subdivision so that subdivision is only done in the areas that need it.

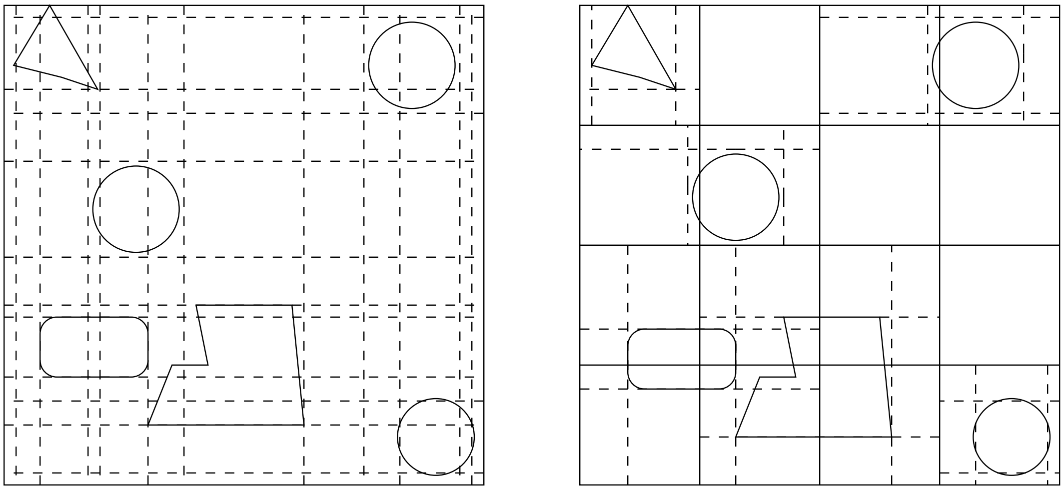
Simiakakis and Day [88] present Five-dimensional Adaptive Subdivision (FAS), a hybrid method incorporating the 5-D and the spatial subdivision methods in order to benefit from both. They calculate a heuristic designed to minimise the sum of the costs of the bounding volume intersections and the culling of the objects by the 5-D component by profiling the renderer, and subsequent renderings are done on the basis of the heuristic. The results indicate that it provides an efficient method for rendering polygonal based scenes, but has problems with non-polygonal objects and scenes with a large number of reflective/refractive objects.

## 2.5 Extent based methods

Semwal [86] describes a technique called the Slicing Extent Technique (SET) which partitions the object space with slicing planes perpendicular to the three axes. A slicing plane is created for both the minimum and maximum extent of each object in the scene. After culling out duplicate planes each plane is divided into a rectangular cell, and each cell contains a list of objects within the cells area. Cell size and the space between slices vary, and rays are traced by processing the planes in order along the ray's path and determining each cell on each plane hit by the ray. Unfortunately in a scene with many objects there will be many planes and traversal between planes becomes difficult, in addition, as described, the SET algorithm does not handle a ray origin within an extent, so only primary rays can be traced using this method.

As the way in which extents are created can effect the efficiency of the ray tracing some work has been done on calculating the approximate number of intersections that might result from a given tree of extents. Goldsmith and Salmon [38] provide a means for calculating this cost and use this to help provide a means of automatically generating more optimal trees. Some speed up was reported but the estimation method only took into account primary rays.

Wilson and Deo [100] also describe a method based on extent tracing. Arrays of object extents are constructed for each axis. A ray is then traced by visiting successive extents in each array. When an object's closest extent in an axis is passed, it is tested for intersections with the ray. To speed up the processing, some extents are chosen to form voxel boundaries, so that only local extents are processed. Figure 8 shows an example scene with the two types of extent tracing. As can be seen from the diagram, once voxels are formed from extents a ray crossing the scene will cross extents less often than without voxels, resulting in less intersection tests against their associated objects. Although attractive from a theoretical point of view, this scheme turns out to be only comparable to previous ones [102].



(a) Extent tracing (b) Extent tracing with subdivision

Figure 8: Two applications of extent tracing.

## 2.6 Using Z-buffers.

Many graphics workstations now have hardware Z-buffers and Sung [94] proposes a method for accelerating ray tracing on machines with a hardware Z-buffer. The technique is called the Area Sample Buffer and methods are put forward for using it to handle primary rays, shadow rays, and other secondary rays. While the algorithm is described as being very efficient, unfortunately, it is restricted to polygons and there are some problems handling shadow rays.

## 2.7 Conclusion

Since the early papers by Turner Whitted [104] and Roth [78] much progress has been made in algorithms to efficiently traverse and access space for the purpose of ray tracing. One of the problems that is still apparent is that the efficiency of a given technique depends a lot on the distribution of objects in the scene being traversed. The work of Weghorst *et al* does indicate that a well constructed bounding volume hierarchy should minimise the number of intersections taking place in any scene, and, providing the cost of traversing the hierarchy is not too high, minimise the time taken to do the traversal as well. An attempt at developing a low cost tight fitting structure and the results of those endevours are presented in chapters four and five.

It has also been speculated by some researchers that no new data structure will be designed to speed up ray tracing time [44], and having got to this stage many researchers felt that more sophisticated hardware was required. The next chapter deals with these developments, but, as will be seen, many of the these algorithms also have some sort of grid or hierarchy as their basis.

# Chapter 3

# Parallel Acceleration Techniques

As ray tracing involves the generation and tracing of individual rays through the scene the problem lends itself well to parallel solutions. Research in this area commenced in the mid-1980's, but early research was orientated around special purpose hardware or simulations of special purpose hardware which created some problems for doing analysis or comparisons of algorithms. More recently parallel machines have become more available and, consequently, better analysis of parallel algorithms has been possible.

Parallel acceleration techniques fall into two classes: *ray distribution* and *object distribution*. In ray distribution methods the object database is copied to each processor and the processors are assigned sets of rays to trace. In object distribution methods, the objects are divided up using some spatial subdivision algorithm and groups of them are assigned to each processor. Rays that enter the area of space taken up by the objects are then handed to the processors during rendering.

Ray distribution techniques involve greater overhead in memory usage, but are simpler to implement than most object distribution techniques. Clusters of rays can be assigned to each processor at the start or dynamically as the previous ray is processed. Dynamic allocation of rays allows for better load balancing with only a small increase in interprocessor communication.

Object distribution techniques involve substantially more communication overhead. Memory usage is more efficient than with ray distribution but load balancing is more complex, and it is not immediately obvious how objects should be distributed between processors in a manner that will ensure that each processor is being used efficiently. Processors whose object space includes a light source are particularly prone to being swamped while other processors remain idle. This happens because light sources act as a target point for rays during shadow testing, something which is normally done at every level of shading.

It is important to note that not all the machines described in the rest of this chapter have been built. It has been indicated where this is the case.

## 3.1 Ray Tracing Machines

The idea of designing a special purpose machine is always appealing. Rather than having to consider limitations of the hardware, you can simply design them out. In some cases research has been done using simulations rather than constructing the hardware. Where this is the case it is important to bear in mind that simulations often do not take into account problems or overheads that might arise due to interprocess communication.

### 3.1.1 Simulations

The problem of dynamically balancing the assignment of clusters of objects to processors was examined by Dippé and Swensen in [26] Clusters of objects were bounded by volumes which shared common vertices with the bounding volumes of other object clusters. Load balancing involved moving boundary vertices and redistributing objects according to the space enclosed by the new bounding volumes. Several bounding volume geometries were studied: parallelepipeds, general boxes and tetrahedra. Parallelepipeds and general boxes are more complicated to intersect than tetrahedra, but allow for load balancing to be done more locally as there are only seven volumes sharing a common vertex. In the case of tetrahedra, intersection is simpler, but more volumes share a common vertex.

Hébert, McNeill, Shah, Grimsdael and Lister [47] proposed an architecture that consisted of a collection of processor clusters. Each cluster had some shared memory and a shared bus in addition to local memory for each processor. The objects in the scene to be rendered were loaded into each processor cluster and an octree was built in the shared memory with subtrees being assigned to each processor in the cluster. Pixels were then assigned in blocks to each cluster for ray tracing. This architecture showed good performance characteristics unless local memory on the processors was too small, in which case bus contention became a problem.

More recently McNeill, Shah, Hébert, Lister, and Grimsdale [66] proposed a dynamic strategy for octree construction on a parallel ray tracing machine, rather than building the octree in an initialisation stage. Initial tests on the algorithm indicated that it handled large databases of objects very effectively.

Kobayashi, Nishimura, Kubota, Nakamura, and Shigei [59] studied static load balancing on 1, 2, and 3-D mesh machines using simulation. In all cases uniform subdivision was performed on the input data and the resulting voxels were assigned to processors. It was assumed there would normally be more voxels than processors so two methods of distributing voxels was used: distributed allocation and block allocation. Figure 9 shows both these methods over a 2d slice, the letters represent individual processors. The block allocation method assigns several adjacent voxels to each processor. The distributed allocation method assigns a block of adjacent regions to adjacent processors. While block allocation provides for better object space coherence and uses less memory, load balancing is much better in the distributed allocation case, and performance is 2 to 3 times faster. The 2D mesh is shown to

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | b | c | a | b | c | a | b | c |
| d | e | f | d | e | f | d | e | f |
| g | h | i | g | h | i | g | h | i |
| a | b | c | a | b | c | a | b | c |
| d | e | f | d | e | f | d | e | f |
| g | h | i | g | h | i | g | h | i |
| a | b | c | a | b | c | a | b | c |
| d | e | f | d | e | f | d | e | f |
| g | h | i | g | h | i | g | h | i |

(a) Distributed Allocation

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | b | b | c | c | c |
| a | a | a | b | b | b | c | c | c |
| a | a | a | b | b | b | c | c | c |
| d | d | d | e | e | e | f | f | f |
| d | d | d | e | e | e | f | f | f |
| d | d | d | e | e | e | f | f | f |
| g | g | g | h | h | h | i | i | i |
| g | g | g | h | h | h | i | i | i |
| g | g | g | h | h | h | i | i | i |

(b) Block Allocation

Figure 9: Distributed and block allocation.

outperform the 1D and 3D meshes.

Kunii, Nishimura, and Noma [56] applied task graphs to the problem of designing a parallel ray tracing machine. Full details of this are beyond the scope of this document, but simulated performance was shown to be good when the number of processors was large and communication time between processors was small.

Morris and Dew [68] described a data flow machine for doing constructive solid geometry (CSG) consisting of a ray controller, a shader, and primitive and CSG combine classifiers. The ray controller casts rays into the CSG combiners which in turn pass them on to the primitive classifiers. After processing in the primitive classifiers the results are then passed back up the pipeline. All the primitive classifiers operated in parallel and some CSG combine classifiers operated in parallel also; otherwise work was assigned to the processors as needed.

### 3.1.2 LINKS-1

Nishimura, Ohno, Kawata, Shirakawa, and Omura [69] describe the LINKS-1 machine. The design consisted of 64 unit computers, a root computer, and a memory swapping unit. The ray tracing process was divided up into three functions, sorting, ray casting and shading. Three unit computers were put together to form a pipeline which carried out these tasks. A speedup of only two was reported because the ray casting unit in the pipeline required about 30 percent more time than the other units.

## 3.2 Hypercubes

Hypercube machines a machines with a number of processors which is a power of 2 (i.e. $2^n$) with each processor connected to $n$ others [76].

In [81] Salmon and Goldsmith described a algorithm that involved distributing the object database among the processors of a 128 processor NCUBE machine using a hierarchical spatial subdivision. This was done by assigning subtrees to each processor. Several pixels were then multi-tasked on each processor to avoid the idle time associated with waiting for results from another processor. Processor loads still varied by as much as 39 percent as some subtrees were accessed far more frequently than others. A proposed solution was to assign very busy subtrees to more than one processor.

An octree algorithm was described by Kobayashi, Nakamura, and Shigei in [60] which mapped octree voxels to processors. The mapping is reported to be nearly optimal as it assigns octree leaves which are siblings to adjacent processors, and assigns other leaves to processors with a communication distance proportional to the distance between the leaves' levels in the tree. If there are more voxels than processors, block or scattered allocation can be done.

Bouatouch and Priol [14] describe a subdivision algorithm based on the image's screen space. This divides the object space into a set of 3D pyramids and the resulting 3D regions representing what objects lie in a block of pixels in the image are divided up among the processors. As secondary rays may pass into another processor's region and two processors can deadlock trying to add a ray to each other's full ray queue a virtual token ring is used to detect the deadlock situation.

Static load balancing is done by subsampling the image and then recursively dividing the screen so that each half has roughly the same number of rays. An adjacency graph is constructed for determining which processor to communicate with when passing rays from volume to volume.

Priol and Bouatouch [74] extend the previous work with one difference being that voxels are orthogonal parallelepipeds. Similar to the idea of Kobayashi *et al*, the assignment of processors to regions is done in order to attempt to minimise the distance between processors in the hypercube that are adjacent in the graph. Efficiency is poor (only 30 percent for 64 processors) partly due to load imbalance, but also because some processors intersect a ray with the same object repeatedly.

A Hierarchical Tree of Extents was used by Caspary and Scherson [16] with upper levels of the tree being replicated in each processor. Subtrees below the upper levels are then assigned to individual processors. Figure 10 demonstrates the idea, with the letters representing individual processor units. The decision of how many levels should belong to the upper levels is discussed and some testing is done with this parameter. All tests were done on a simulated hypercube. Neither deadlock in communication or message bottlenecks (both of which would be a consideration here) were discussed.

Badouel, Bouatouch, and Priol [9][10] implemented a shared memory model using a distributed memory model on a 64 node hypercube. Each processor in the machine stored part of the object database in local memory, the remaining portion of memory being used as an object cache. Each processor is given a block of pixels to trace and objects not present in the processor's memory are requested and cached. When a processor finishes its subimage, work is requested from other processors.

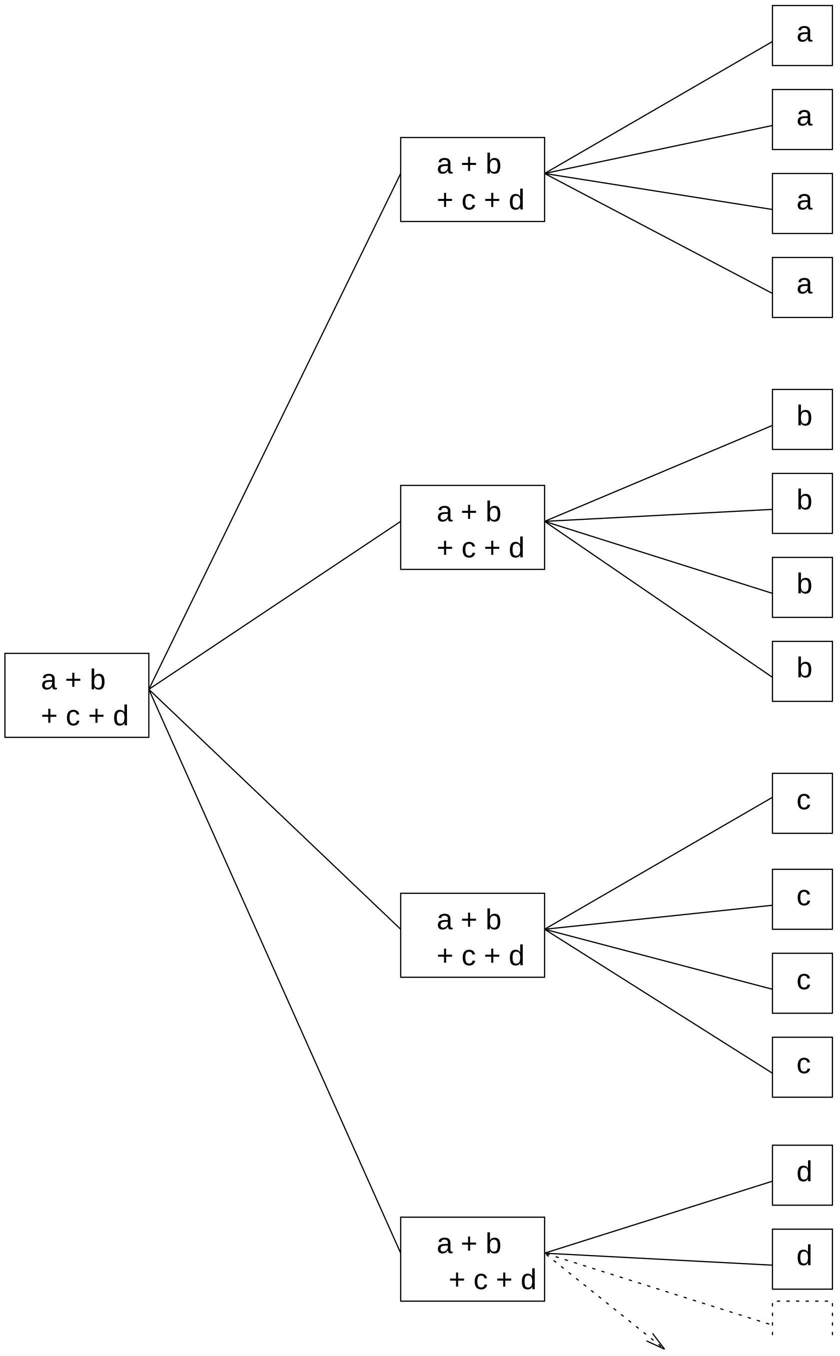



Figure 10: Distribution of octree nodes amongst CPUs - Caspary approach.

The shared memory algorithm outperforms the voxel decomposition method and has several advantages. Deadlock does not occur in shared memory because rays are not passed around. Ray-object intersections are not repeated by other processors. As the number of processors increases the size of the object cache increases as less objects need to be stored in each processor. Efficiency was often greater than 78 percent.

Object distribution was looked at by Carter and Teague [15] who distributed the object database and performed object caching. An hierarchical subdivision was constructed and replicated on each processor. In the case of an object being requested by a remote processor the subtree containing the object was transferred to minimise the number of subsequent messages.

## 3.3 Transputers

Transputers are processors provided with four communication ports that allow them to connect with other processors, thus constructing a parallel machine. It is important to note that not all transputer based machines that have appeared in the literature have actually been built.

A transputer architecture based on ray distribution was constructed by Atkin, Ghee, and Packer in [7]. The transputers were connected in a linear array with bidirectional links. Blocks of pixels were passed along the array, until a free machine was found to run it on. A speed up of 76 was noted using 80 processors, although this was for a very small scene (only four spheres). The entire object set had to be stored in each node.

A transputer based machine designed around a 3D mesh was proposed by Caubet, Duthen, and Gaildrat in [17]. One difficulty with this design is that a transputer has only four communications ports and each node in the mesh needs six links, thus two transputers would need to be used for each node. Construction/simulation of this machine is yet to be carried out.

Cosshall [21] described the NEXUS multi-processor. In a similar vein to the work of Atkin *et al.* [7] the machine relied on distribution of the rays to speed up the ray tracing. However shared memory was utilised and it was hoped that only part of the object set needed to be stored in the local memory of the processors at any one time.

Green, Paddon, and Lewis [40] described a tree computer based on transputers. Objects were distributed to the processors and cached by others when they were required. Near linear speed ups occurred if the object set was resident in the local memory of each processor, and it was noted that degradation of performance occurred when objects had to be requested and cached. Green and Paddon [41] extended their previous work further and constructed a machine using 22 processors. A linear speedup was again noted and the system was tested using some of the benchmarks described by Haines in [43].

Barton [8] and Chalmers [18] both make reference to a machine called the Meiko Computing Surface. This is referred to as having from 14 to over a hundred transputers in it, but unfortunately, little detail about performance results is given.

A group of transputers operating on a regular subdivision was used by Pitot, in [72], to carry out the ray tracing. Initial results are presented for the prototype system which look favourable, however, the system exhibits problems arising from deadlock between the transputers and the use of a regular subdivision leads to inefficiencies when objects end up clustered in one voxel. Communication between processors to synchronise the rendering were also proving costly.

## 3.4 Shared Memory Machines

Shared memory machines have a memory which is accessible by any processor in the machine [76]. Processors may also have local memory as well. Some communications and caching issues arise here as typically accessing to remote memory locations often involve a delay compared local memory accesses.

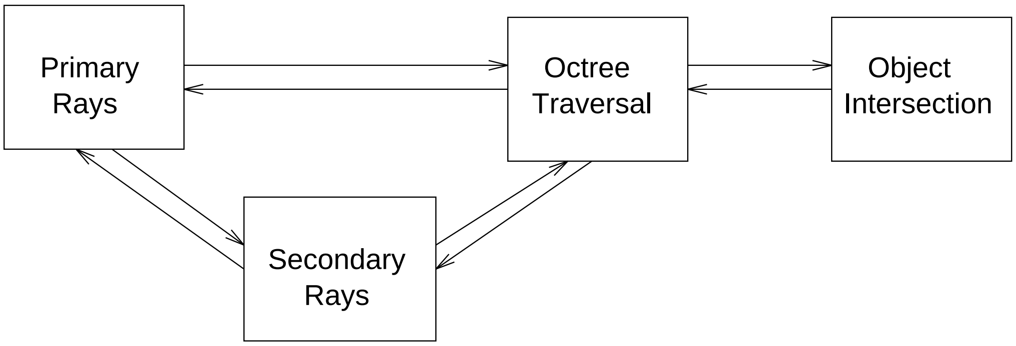



Figure 11: Communication between tasks on a Sequent.

A load balancing study was performed by Barrett [12] on a Sequent shared memory multiprocessor. The ray tracing was divided into four tasks, namely: a ray task which initiates rays and computes intensities, an octree task which traverses an octree structure with a ray, an intersection task which performs ray-object intersections, and a tree task which coordinates other tasks and generates secondary rays. Each task had an input queue. Figure 11 illustrates the connectivity between the tasks. Two forms of scheduling were investigated, static and dynamic.

Static scheduling allocates a percentage of the total processors to each of the four tasks. Two dynamic scheduling algorithms are investigated. One referred to as periodic checking, and the other referred to as event checking. In periodic checking, queues are checked periodically for an imbalance, and the tasks assigned to processors changed accordingly if an imbalance occurs. In the case of event checking, queues are checked when an event signals a possible imbalance.

In the case of static scheduling, the number of processors assigned to each task had a major impact on performance. If too many processors were assigned a particular task, that task would end up having to wait for other tasks to complete. Of the two dynamic scheduling algorithms, event checking outperformed periodic checking by a small amount. As a last test case an image space subdivision algorithm, which assigned scanlines to individual processors was run. It outperformed all three of the other scheduling algorithms.

More recently, Badouel, Bouatouch, and Priol [11] implemented a ray tracing algorithm that utilised distributed memory using the Intel iPSC/2. Two algorithms were presented, one, quite complex, based on message passing, and the second, regarded as more straight forward, using a control-orientated parallel approach and a shared virtual memory. Extensive analysis was done and the control orientated approach proved to be more successful. Indications are the method would also be applicable to areas such as radiosity.

## 3.5 Vector machines

Vector machines can be used to accelerate computation in situations where one function needs to be applied to many operands at once. To get an appreciable speed up this often requires that the function can be divided into stages and that there are a large number of operands.

Plunkett and Bailey [73] describe work they did on a Cyber 205. Several object types were intersected with 100,000 rays in vectors of 500 rays. Speed ups on the intersection testing ranged from 10 to 33, however test scenes only showed a speed up of about six times as it was found not much of the ray tracing algorithm can be vectorised.

Gaudet, Hobson, Chika, and Calvert [32] describe the simulation of the Pipelined Engine for Ray Tracing (PERT) which consisted of three processors. The ShadeProcessor, the ShellProcessor, and the PrimProcessor. These processors looked after ray spawning and shading, ray bounding volume intersections, and ray object intersections respectively. Two architectures were designed using multiple PERTS, one with local memory and broadcast buses, and the other with global memory and a shared bus. Indications were that the broadcast bus model may not keep processors busy enough, particularly when broadcast latency becomes a factor. The shared bus did not appear to be a bottleneck as much computation is performed between bus accesses.

Wilson and Deo [101] refer to the work of Defend [23] and Quek [75], both of whom attempted to introduce some spatial subdivision into the vectorised algorithm in an attempt to reduce the number of unnecessary calculations being carried out. Both tested their algorithms on an Alliant FX/8, which is an eight vector processor machine. In both cases they found the resulting vectors generated were too short to show any gain in speed.

## 3.6 SIMD Machines

SIMD machines are machines with many simple processors controlled by a single controller. As with vector machines problems SIMD machines are well suited to problems involving large amounts of data which on which a common operation needs to be performed.

A ray tracing algorithm similar to the octree was proposed by Dorband [27] based on sort computation, that is sorting with the comparison step replaced with a high-level operation. Initially a voxel is set up which contains the entire scene, and each processor is assigned a ray and an object. On processors where the ray and object are within the voxel the voxel is subdivided into eight smaller ones and the process repeated until all voxels not containing the both the ray and the assigned object are culled. This algorithm was designed for the Massively Parallel Processor, a $128 \times 128$ mesh processor, although the feasibility of this algorithm was simulated.

Delany [24] used a Connection Machine CM-1. Each processor was assigned a ray and an object and the objects were organised in the processor-address space using an octree. Some execution times were given, but no performance analysis.

Marsh [64] proposed mapping an octree onto a mesh of processors. Ideally there would be one octree node per processor, but for very complex scenes where this may not be possible the notion of virtual processors was used. Several virtual processors could be assigned to each real processor. Marsh also looks at the problem of handling CSG. Estimated performance is given, but the algorithm was untested.

Image space, rather than object space, is divided up in the work of Williams, Buxton, and Buxton [99] who describe ray tracing on a $64 \times 64$ array of processors. In this case the work is divided up by assigning one pixel to each processor. Extensive descriptions are given on object representation and ray-object intersection for the SIMD model, but no timings are given.

Crow, Demos, Hardy McLaughlin, and Sims [22] also describe two ray tracing implementations based around dividing the image amongst the processors. Two alternative implementations were tried. While performance was acceptable it was found that in one case the implementation would allow some processors to become idle during the processing of secondary rays, but the alternative would waste a lot of computation. In later work Lin and Slater [62] simulated a ray tracing algorithm which improved processor efficiency — on one scene that noted processor efficiency over 50 percent. In addition to this some work was done in the design to avoid having all the objects tested for every ray.

## 3.7 Conclusions

Parallel acceleration of ray tracing appears to be an area which is only just coming of age. Algorithms are very much tied to the architecture of the machine so comparisons are difficult. Further work is needed just to establish which architecture would be best suited to the problem. Research has shown that it is very difficult to get worthwhile acceleration of ray tracing on a vector machine, and a number of the approaches used seem to boil down to a brute force approach involving throwing more processors at a naive ray tracing algorithm.

Hypercubes and shared memory machines allow a wide range of algorithmic approaches. Issues such as dividing the object database and the different types of rays are more easily resolved, and both advances in general algorithms and parallel algorithm development could be utilised here. In addition it appears that building large enough machines of this type is beginning to become feasible, so the future of this area probably lies with these architectures.

Whatever shape future parallel algorithms take, if current experience is any indicator, they will most probably be built around some form of spatial subdivision in order to make the best use of the additional computational power that specialised architectures can make available. For this reason the rest of this thesis is devoted to examining a subdivision algorithm that can be run on standard computer hardware. It should be pointed out, however, that, by using the methods described in this chapter, the algorithm described in the following chapters, being hierarchical and thus similar to the octree in complexity, should adapt fairly smoothly to parallel architectures.

# Chapter 4

# The BV-tree

The bv-tree, or bounding-volume tree, is essentially a version of a bounding volume hierarchy with a couple of major differences from previous methods involving bounding volumes. It is built on the fly, and, as it is hierarchical, it is collapsible on the fly, which means it can be used to accommodate hierarchies that are large enough to exhaust available memory, in addition the work of McNeill *et al.* in [66] indicates that adaptive data structures generally make better use of available memory during a rendering, and the data structure is designed to take advantage of this fact. Unlike other tree hierarchies it also guarantees that any object will only be tested against the ray once, so, for this reason, it does not require ray mail boxes to be associated with the objects stored in the tree as proposed by Glassner in [34].

## 4.1 Initial Motivation

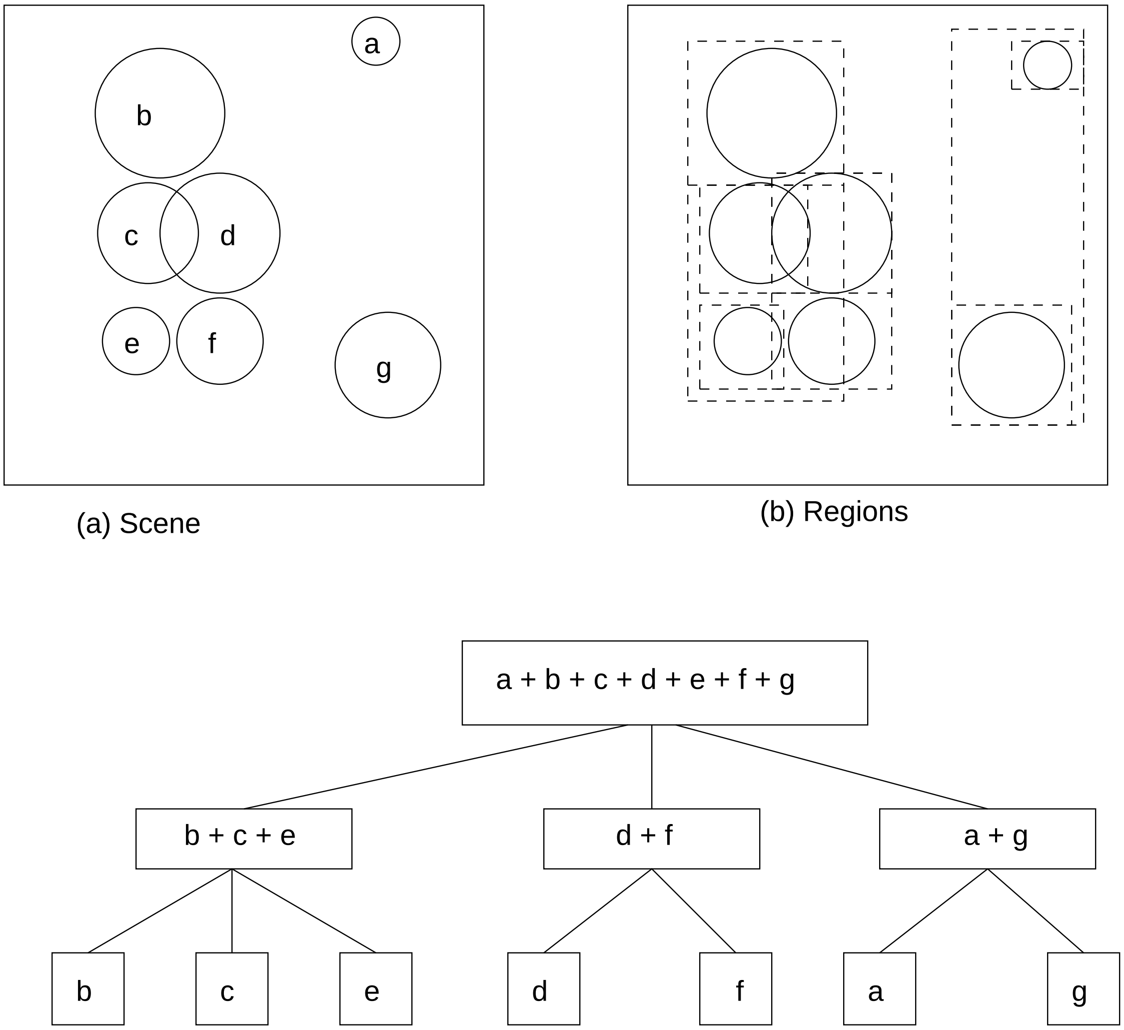
The principle motivation for the bv-tree was that research into the kd-tree as described in Hook and Forward [48], and Subramanian and Fussell [92] showed that 15 to 20 percent of the object intersection time was spent in the tree algorithm confirming that a hit was actually taking place within the confines of the voxel the ray was currently in. Other problems with the kd-tree were that large objects tended to create lots of empty space in the tree as they often crossed empty partition planes, there were storage requirements associated with duplicate pointers to objects that lived in more than one tree node, and there was the extra overhead of storing mail box information in each object as detailed above. In addition to this, performance of the kd-tree tended to deteriorate compared with that of the grid if the data was distributed through the scene in a uniform manner, whereas the performance of the grid deteriorates quite badly if the distribution of information through the scene is non-uniform.

The idea of the bv-tree was that by allowing a three way split, large objects would be filtered out of the tree nodes early, thus resulting in a tighter overall fit, that as the objects were guaranteed by the tree construction algorithm to be contained within the voxel representing the tree node, objects would only need to be tested once and there would be no need for mailboxes or duplicate object pointers. The extra storage required because each level of the tree represents a three way split is more than offset by the savings made on not having duplicate pointers and ray mail boxes present in the object structure. Figure 12 shows an example of a bv-tree structure for a sample scene, the region diagram demonstrates how empty space is eliminated rapidly with every subdivision.

## 4.2 The BV-Tree Node Structure

The bv-tree node structure is quite straight forward. It could be specified in Pascal using variant records or in C/C++ by using a structure and a union. The C language definition for the node structure can be seen in Figure 13.

The `Type` field can have one of five possible values: `X`, `Y`, `Z`, `Splitable`, and `Leaf`. The `X`, `Y`, and `Z` node types are branch nodes that have been subdivided along the X, Y, or Z axes respectively. The `Splitable` node type indicates a leaf node that is still




(a) Scene (b) Regions

(c) BV-tree composition of scene.

Figure 12: A sample scene, its decomposition into a bv-tree, and the regions it generates.

```
typedef struct bvt {
    char                Type;
    union {
        struct {
            float        SplitValue;
            struct bvt   *Left;
            struct bvt   *Mid;
            struct bvt   *Right;
        } Branch;
        struct {
            object       *ObjList;
            int          Depth;
        } Leaf;
    } U;
}
```

Figure 13: C definition of the bv-tree node structure.

a candidate for subdivision, and the `Leaf` node type indicates a leaf node that is no longer suitable for subdivision. When a ray encounters a `Leaf` type node the objects in the object list are tested in the order given to see if the ray intersects with any of them.

The `Branch` structure in the union part of the node has the value along which the axis given by the `Type` field was subdivided and the pointers to the left, mid, and right subbranches of the node.

The `Leaf` structure in the union part of the node contains the list of objects in that leaf (`ObjList`), and the depth (`Depth`) of the leaf node in the tree structure.

On most implementations the `Branch` structure will be twice the size of the `Leaf` structure, so it is likely, if a need arose to further condense the tree, a simple packing algorithm could be devised to eliminate the wasted space resulting from the creation of leaf nodes.

## 4.3 A Brief Note on the Algorithms

In some of the algorithms detailed below reference is made to various macros that are not defined. In most cases the names fully describe the fields they return but the abbreviation `BBox` is used to refer to the the bounding box of a node or object. The macros `Min`, `Mid`, and `Max` return the minimum, middle, or maximum values of the bounding box given as the first argument in the axis given as the second argument. The syntax of the code used in the algorithms below is essentially the same as that of the C programming language.

## 4.4 Initialisation

Initially the tree is a single node containing all the objects marked as a `Splitable` node. The only preprocessing required is the calculation of the bounding box for the node. The following pseudo-code demonstrates the idea.

```
/*
 * InitTree
 *
 *     set up a tree node from the list of objects passed in and
 * return a pointer to it.
 */
bvtnode *
InitTree(
    object *ObjList;
)
{
    n = New(bvtnode);
```

```
    Objects(n) = ObjList;
    Depth(n) = 0;

    CalcBBox(n);

    return(n);
}
```

## 4.5 Node Subdivision

Nodes and branches of the tree are created by dividing the space at each level into three regions along the dimension of greatest extent. The regions are created as follows: the leftmost region is the one in which all objects are less than the value of the splitting plane, the middle region is the one in which all objects intersect the splitting plane, and the rightmost region is the one in which all objects lie on the positive side of the splitting plane. The splitting plane is always axis aligned to make these calculations easy to do. Once a leaf node is split up, the bounding boxes for each of its three subregions are calculated and providing a certain depth, or a minimum number of objects in a node has not been reached the splitting process is carried out depending on the path of the ray.

At the moment a good value for the minimum number of objects is 3. Also, the splitting process is not carried out if the split will result in only one sub node containing objects. A typical algorithm for handling the splitting of the nodes is as follows:

```
/*
 * SplitNode
 *
 *      find the "optimum" splitting value and divide the node
```

```
 * along an axis.
 */
void
SplitNode(
    bvtree *node
)
{
    xsplit = Mid(BBox(node), X);
    ysplit = Mid(BBox(node), Y);
    zsplit = Mid(BBox(node), Z);

    xdbl = xright = 0;
    ydbl = yright = 0;
    zdbl = zright = 0;
    tot = 0;

    for (o = Objects(node); o != Null; o = Next(o)) {

        if (xsplit < Min(BBox(o), X))
            xright++;
        else if (xsplit < Max(BBox(o), X))
            xdbl++;

        if (ysplit < Min(BBox(o), Y))
            yright++;
        else if (ysplit < Max(BBox(o), Y))
            ydbl++;

        if (zsplit < Min(BBox(o), Z))
            zright++;
        else if (zsplit < Max(BBox(o), Z))
```

```
            zdbl++;

        tot++;
    }

    /*
     * Check for special cases
     */

    /*
     * can't split further
     */
    if (xdbl == tot && ydbl == tot && zdbl == tot) {
        Type(node) = LEAF;
        return;
    }

    /*
     * The size of the middle branch is used as a measure of
     * the relative cost of doing a split along an axis, the
     * higher it is the more costly the split. We bias this
     * value further if we have a situation we would rather avoid,
     * in this case having one of the resulting subbranches being
     * empty.
     */
    if (xright == 0 || xright == tot || xdbl == tot)
        xdbl = xdbl + tot;
    if (yright == 0 || yright == tot || ydbl == tot)
        ydbl = ydbl + tot;
    if (zright == 0 || zright == tot || zdbl == tot)
        zdbl = zdbl + tot;
```

```
if (xdbl < ydbl)
    if (xdbl < zdbl) {
        SplitValue(node) = xsplit;
        Type(node) = X;
    } else {
        SplitValue(node) = zsplit;
        Type(node) = Z;
    }
else
    if (ydbl < zdbl) {
        SplitValue(node) = ysplit;
        Type(node) = Y;
    } else {
        SplitValue(node) = zsplit;
        Type(node) = Z;
    }


left = New(bvtnode);
right = New(bvtnode);
mid = New(bvtnode);

for (o = Objects(node); o != Null; o = NextO) {

    NextO = Next(o);

    if (SplitValue(node) < Min(BBox(o), Type(node))) {
        Next(o) = Objects(right)
        Objects(right) = o;
    } else if (SplitValue(node) < Max(BBox(o), Type(node))) {
```

```
            Next(o) = Objects(mid)
            Objects(mid) = o;
        } else {
            Next(o) = Objects(left)
            Objects(left) = o;
        }

    }

    CalcBBox(right);
    CalcBBox(left);
    CalcBBox(mid);

    Depth(right) = Depth(mid) = Depth(left) = Depth(Leaf) + 1;

    if (MaxDepth(node) > MaximumDepth)
        Type(right) = Type(mid) = Type(left) = Splitable;
    else
        Type(right) = Type(mid) = Type(left) = Leaf;

    if (NumberOfObjects(left) < MinObjects)
        Type(n) = Leaf;
    if (NumberOfObjects(mid) < MinObjects)
        Type(n) = Leaf;
    if (NumberOfObjects(right) < MinObjects)
        Type(n) = Leaf;

    Left(node) = left;
    Mid(node) = mid;
    Right(node) = right;
```

```
    return;
}
```

If required, the construction algorithm for the tree could be handled in a preprocessing stage, but the experience of the author in using this algorithm is that this results in a large tree, many parts of which are never visited by rays cast during the rendering process. This is also in line with the result obtained by McNeill *et al.* in [66].

It may seem to the reader that calculating the bounding boxes at each level is an unnecessary expense. However the work of Weghorst, *et al.* in [98] shows that this is more than offset by the savings made by the reduction in the number of intersection calculations done during a ray traversal as the recalculation results in subbranches of the bv-tree being pruned from the search earlier than they would be otherwise.

## 4.6 Traversal and Ray Intersection

Traversal of the bv-tree, as with most tree based algorithms involves a relatively simple recursive algorithm. In its most basic form the traversal algorithm is similar to that laid out by Jansen in [50], excepting the fact that there are three branches to be traversed instead of two. This algorithm is as follows:

```
/*
 * TreeTrace
 *
 *      return True if the ray intersects an object in the tree structure
 * pointed to by node, False otherwise.
 */
boolean
TreeTrace(
```

```
    ray *r;
    bvtnode *root;
)
{
    if (MissedBBox(r, BBox(node))
        return(False);

    hit1 = hit2 = hit3 = False;

    if (Type(node) == Leaf)
        return(IntersectObjects(r, node));
    else if (Type(node) == Splitable)
        SplitNode(node);
        TreeTrace(node);
    } else {
        if (Direction(r, Type(node)) > 0.0) {
            hit1 = TreeTrace(Left(node));
            hit2 = TreeTrace(Mid(node));
            if (!hit1)
                hit3 = TreeTrace(Right(node));
        } else {
            hit1 = TreeTrace(Right(node));
            hit2 = TreeTrace(Mid(node));
            if (!hit1)
                hit3 = TreeTrace(Left(node));
        }
    }

    return(hit1 || hit2 || hit3);
}
```

Note: the `IntersectObjects` routine tests each object in `node` against the ray `r` returning `True` if one or more of the objects in `node` are intersected, and the `MissedBBox` routine returns `True` if the ray `r` misses the bounding box given in the second argument.

## 4.7 A More Refined Traversal Algorithm

It is noted here that simple reasoning indicates that if an intersection occurs in the left or right subbranch of the node the intersection will lie outside of the range of the bounding box for the subbranch on the other side of the node.

This is a useful feature of the tree, and by taking into account the origin of the ray it enables us to arrive at a more efficient algorithm. In this case we can use knowledge of the ray's origin and its direction to assist us in pruning branches that the ray will not enter.

```
/*
 * TreeTrace
 *
 *      return True if the ray intersects an object in the tree structure
 * pointed to by node, False otherwise.
 */
boolean
TreeTrace(
    ray *r;
    bvtnode *root;
)
{
    if (MissedBBox(r, BBox(node))
        return(False);
```

```
    hit1 = hit2 = hit3 = False;

    if (Type(node) == Leaf)
        return(IntersectObjects(r, node));
    else if (Type(node) == Splitable)
        SplitNode(node);
        TreeTrace(node);
    } else {
        if (Origin(r, Type(node)) < SplitValue(node)) {
            hit1 = TreeTrace(Left(node));
            hit2 = TreeTrace(Mid(node));
            if (Direction(r, Type(node)) > 0.0 && !hit1)
                hit3 = TreeTrace(Right(node));
        } else {
            hit1 = TreeTrace(Right(node));
            hit2 = TreeTrace(Mid(node));
            if (Direction(r, Type(node)) < 0.0 && !hit1)
                hit3 = TreeTrace(Left(node));
        }
    }

    return(hit1 || hit2 || hit3);
}
```

## 4.8 Further Refinements

There are a number of further refinements that can be made to the traversal and intersection algorithm. These involve two basic comparisons, involving either the ray origin and the bounding box of the tree node, or a current intersection of the ray and the bounding box of the tree node.

In the first case, if the ray origin for the axis along which the tree node has been created is less than the minimum value of the bounding box of the tree node along its axis of creation and the ray direction in that axis is less than zero, the ray will not enter that tree node. This is a simple test, which results in the traversal speeding up substantially. An example code fragment for this is given below:

```
if (Origin(r, Type(node)) < Min(BBox(node), Type(node))
    && Direction(r, Type(node)) < 0.0)
            don't traverse the node....
```

Similar circumstances can also be constructed for cases in which the ray direction is positive and its origin is greater than the maximum value of the node along the axis of creation.

In the second case, a current intersection of the ray may also be considered as a ray origin providing the intersection took place outside of the range of the bounding box for the leaf node we are trying to prune. We need to check for this as, if the intersection took place in the range of the leaf node (a circumstance that can arise when we are attempting to prune the middle subbranch of a node) a new intersection may lie between the current intersection and the ray origin. An example code fragment for this test is as follows:

```
if (HitLocation(r, Type(node)) < Min(BBox(node), Type(node))
    && Origin(r, Type(node)) < Min(BBox(node), Type(node))
        don't traverse the node....
```

In this case it does not matter in which direction the ray is heading, if the current intersection and the origin lie outside the bounding box for the tree node, we have already found a closer intersection and there is no need to search further (at least in this case). As before this can also be applied to ray origins and intersection locations lying above the maximum bounding box of the tree node.

Although the addition of these tests does make the code look a bit complicated, the addition of these tests, at least on the scenes attempted, cut about 10 percent off the ray casting time.

## 4.9 Conclusion

In as far as the original motivation for the design of the data structure is concerned the bv-tree meets the criteria given. It eliminates the computational overheads required to check to see if an intersection is within the bounding box of the tree node we are in, as no objects appear in any tree node more than once. For this reason the bv-tree data structure also eliminates the need to contain ray "mail box" information in the objects themselves. Despite the fact that it is a three way split instead of a two way split as with the kd-tree, the tree should not take any more space as it will be shallower and there are no duplicated object pointers.

Finally the bv-tree should provide a tight fit around the objects in the scene, so should result in less intersection tests, and hopefully greater speed, than a kd-tree, while still being more adaptive than other methods, such as the uniform grid. Overall the results in the next chapter show this to be the case.

# Chapter 5

# Comparison of the BV-tree With Other Methods

To implement all the alternative methods for the general acceleration of ray tracing would be an undertaking well beyond the scope of this work, and in any case it would be unreasonable to do so. As mentioned earlier, some techniques require specialist hardware, which is not available, others are designed for special situations. What has been done instead is to implement three of the more common ones.

## 5.1 The Library

The software the tests were built on, the rt library, is a C library of approximately 7,000 lines in length. It provides facilities for defining basic primitives as well as routines that operate on an object stack to enable different sets of primitives or other hierarchies to be included in object hierarchies. Matrix stack facilities are also provided to allow nesting of transformations and to limit the scope to which a transformation is applied. The manual page for the rt library can be found in Appendix B and, at the time of writing, the C source for the library can be found in the file pub/libRT.tar.gz on the anonymous ftp server gondwana.ecr.mu.oz.au.

Generally the implementations have been kept simple and the library has been designed for easy incorporation of other primitives and optimisation schemes. It is hoped that keeping the library simple will enable others to investigate and reproduce the results given here with the minimum of difficulty. A range of subdivision methods have also been used in the hope of providing a common basis for which these results can be compared with other researchers efforts. Some optimisations were applied via a shadow cache, such as described in Haines and Greenberg [42] and the termination of the ray trace when a shadow ray had traveled the distance to the lightsource. Both these optimisations are common and have a similar impact whatever general acceleration method is employed. In cases where there are shortcomings in the implementations attempts have been made to point them out and give reference to the appropriate literature.

### 5.1.1 The Uniform Grid

Amanatides and Woo [2] describe a 3D grid structure and a very simple algorithm to traverse it. A similar algorithm is also described by Cleary and Wyvill [20].

The implementation used here involved the construction of a $20 \times 20 \times 20$ array of voxels containing lists of the objects that are contained in or overlap the voxel. No attempt is made to eliminate empty regions and "mail box" information is stored in each object so that the intersection routine for a given object is only called once per ray. Objects are stored as a list in each voxel.

### 5.1.2 The Octree

There are many references on methods for implementing the octree starting with the work of Glassner [34] and continuing with other researchers as in the work described in [57] [30] [93] [71] [84] [35] [6] [82, 83] [90]. The definitive discussion of octrees can be found in Samet's book [83].

The implementation used here is a recursive traversal similar to that used by Arvo in [6]. The major difference being that in this case the stack is implicit, rather than explicit. This does mean that the algorithm is slower than it could be, although this is not regarded as significant since the number of intersection tests carried out on the objects stored in the tree will remain the same, with any implementation.

The maximum depth of the octree is 10, and leaf nodes cease to subdivide when there are less that 5 objects in them, the level was chosen so that the octree would be roughly the same size as an equivalent kd-tree or bv-tree. Greater subdivision would also result in better timings, however the tree would then be using much greater resources than the other methods detailed here. Objects are stored as lists in each leaf node.

### 5.1.3 The Kd-tree

In our implementation the Kd-tree is constructed on the fly and is traversed with each ray tracing. Subdivision is done according to the dimension of the longest extent. Although Wilson and Deo theorised in [100] that it was faster to use the tree to construct a grid using the method of [2] so that the leaves could be traversed, other work has shown that recursive traversal can often be quicker [50], [92], additionally as pointed out in [100] it allows the tree to be deeper as the memory used to store the grid is not required. This can result in a finer subdivision thus saving more on intersection tests. The paper by Hook and Forward [48] demonstrates that dividing along the dimension of largest extent produces faster results than using a median cut across the objects as in [92].

The maximum depth of the tree is 30 and a heuristic is used to stop subdivision when subdividing result in more than a third of the objects in the leaf node appearing in the two subnodes. Objects are stored as lists in each leaf node.

### 5.1.4 The BV-tree

The implementation of the BV-tree is the one described in the previous chapter, with all the refinements added to the traversal/intersection algorithm.

The maximum tree depth is 30 and subdivision stops if a subdivision results in two subnodes being empty. That this is possible is a feature of the BV-tree data structure, the subdivision algorithm, by virtue of ensuring every object only appears once anywhere in the tree also means that if it is possible to partition the objects, it will be better to do so. This also allows us to eliminate a lot of object bounding box tests as in a lot of cases a single object will appear in a leaf node. As with the other methods, multiple objects are stored in a leaf node as a list.

## 5.2 The Machine

All the benchmarks were run on a SPARCStation 2 running Sun Solaris 2.3 with 32 Megabytes of local memory. In order to minimise problems that may arise with network traffic due to remote serving of files both the data, the executable and the binary were stored on the machine's local disc. The machine was run with its windowing system and remote logins disabled. All programming was done in the C language.

## 5.3 The Benchmarks

Unlike computational benchmarks, such as Dhrystones, or SPECmarks, benchmarks for ray tracing have not been formally established. There has been one attempt at establishing a set of data bases for benchmarking for ray tracing to date by Eric

Haines in [43], and as these benchmarks now regularly appear in the literature they have been used here so that other researchers have a chance of comparing their results without having to reimplement the work given here.

The benchmarks fall into three categories, non-uniform, strictly-uniform, generally-uniform. There is only one benchmark which is non-uniform, the tree benchmark, there is one which is strictly-uniform, the tetra benchmark, and the remainder of the benchmarks can be considered to be generally-uniform. The figures for the tetra benchmark and the tree benchmark provide the best indication for how the bv-tree structure performs at opposite ends of the spectrum. The additional general uniform benchmarks have been included here as, while they are biased towards what is more ideal for the uniform grid, they demonstrate other advantages of the bv-tree structure as well as exposing some of the limitations and providing pointers to areas of difficulty. They also help raise suggestions as to how some of these shortcomings can be dealt with and show up some of the difficulties inherent in ray tracing varying sets of objects.

## 5.4 The Results

In the following subsection the tables are given abbreviated column headings, bv representing the bv-tree, gr representing the grid, kd representing the kd-tree, and oc representing the oc-tree. Timings are rounded down to the nearest second and intersection counts are rounded up to the nearest thousand.

### 5.4.1 The Tree Benchmark

Of all the SPD benchmarks the tree benchmark is the one that exhibits the most highly non-unform properties. It consists of a large ground polygon with a fractal tree placed in the middle of it. To make matters more interesting the tree is surrounded

Figure 14: The fractal tree

by seven light sources.

The tree benchmark demonstrates that the bv-tree performs at least as well as, and sometimes better than, the kd-tree for a scene which exhibits great non-uniformity in the distribution of objects. The rapid deterioration of the grid demonstrates how this structure suffers when objects become clustered in some voxels, while others are empty. The timings for the octree would probably be improved if the tree was made

| No. of | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| Objects | bv | gr | kd | oc | bv | gr | kd | oc |
| 4095 | 307 | 3503 | 322 | 4783 | 518 | 5209 | 2628 | 18311 |
| 511 | 259 | 1097 | 278 | 4092 | 469 | 2007 | 2350 | 19082 |
| 255 | 225 | 522 | 243 | 3249 | 430 | 1020 | 2163 | 16911 |
| 63 | 184 | 406 | 225 | 2025 | 423 | 670 | 2365 | 13875 |
| 15 | 157 | 385 | 182 | 326 | 365 | 384 | 1975 | 18514 |
| 7 | 123 | 400 | 133 | 188 | 294 | 294 | 1569 | 8158 |

Table 1: Timings and intersections for the tree benchmark

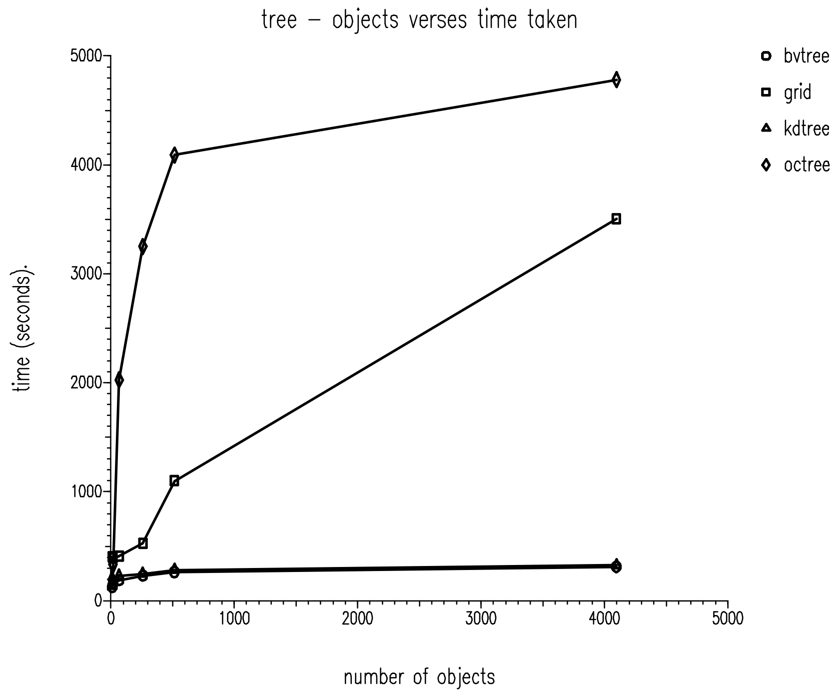



Figure 15: The timings for the tree benchmark

deeper, although in this case it represented a tree with $8^{10}$ nodes, demonstrating why Glassner in [34] and Fujimoto *et al.* in [30] found it necessary to introduce caching and node reduction schemes to make the algorithm practical. The timings and intersection counts for the tree benchmark can be found in Table 5.4.1 and a graphical comparison of the timings can be seen in Figure 15.

### 5.4.2 The Tetra Benchmark

The tetra benchmark consists of a fractal pyramid composed of diffuse reflecting tetrahedron (see Figure 16). There is one light source. It provides the opposite extreme to the tree benchmark, objects are uniformly distributed through space.

The tetra benchmark demonstrates that in a uniform environment the bv-tree performs closer to the grid than to the kd-tree. The uniform environment is the most

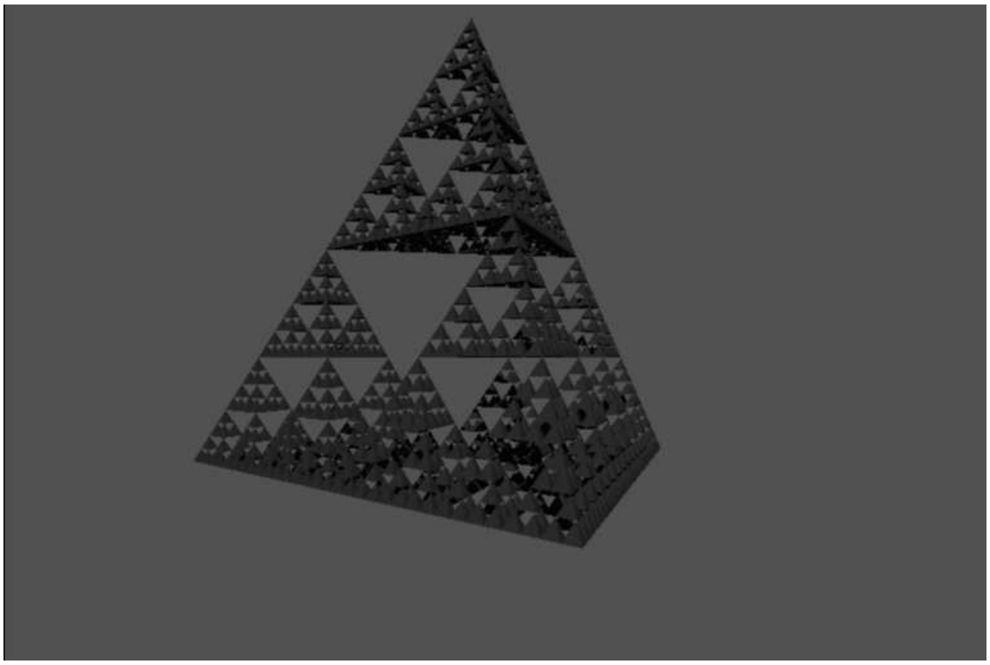


Figure 16: The fractal pyramid

| No. of Objects | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| | bv | gr | kd | oc | bv | gr | kd | oc |
| 4096 | 116 | 73 | 238 | 519 | 524 | 824 | 6061 | 19368 |
| 1024 | 96 | 64 | 201 | 381 | 558 | 658 | 6585 | 19564 |
| 256 | 79 | 63 | 168 | 256 | 600 | 626 | 7635 | 18462 |
| 64 | 65 | 66 | 115 | 187 | 657 | 658 | 8976 | 20472 |
| 16 | 51 | 72 | 62 | 62 | 748 | 748 | 5203 | 5203 |
| 4 | 35 | 90 | 31 | 31 | 864 | 864 | 1279 | 1279 |

Table 2: Timings and intersections for the tetra benchmark

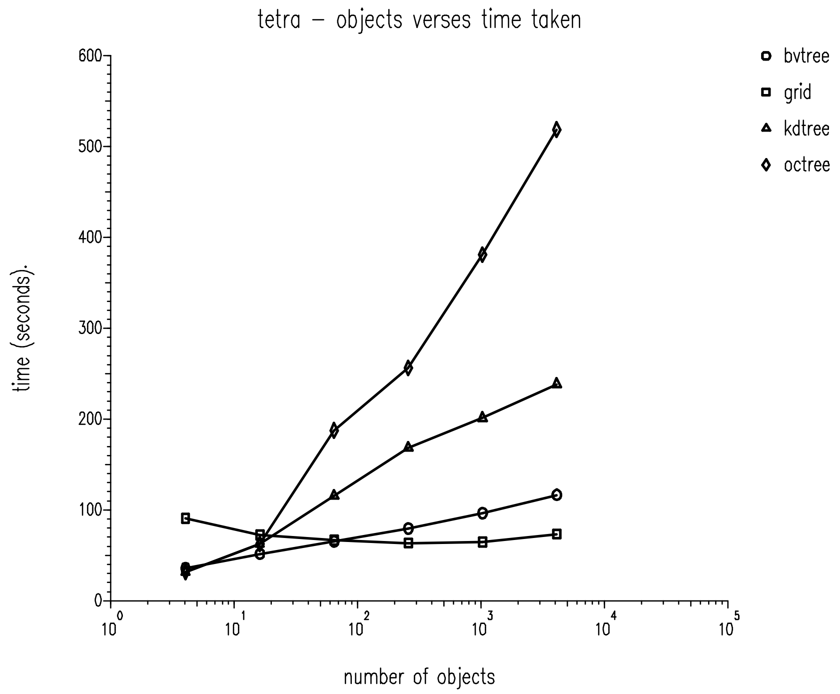



Figure 17: The timings for the tetra benchmark

ideal for the grid and the most unideal for the kd-tree. The performance improvement with the bv-tree is largely due to the way it splits three ways at each level, thus providing a tighter bounding volume at each split. The improvement in the split can also be seen in the lower number of intersection tests carried out with the bv-tree, indicating that a faster traversal algorithm would result in a significant drop in the time taken for the rendering. Table 2 shows the timings and intersection counts for scenes of varying object count. A graphical comparison of the timings can be seen in Figure 17.

### 5.4.3 The Teapot benchmark

The teapot benchmark consists of a phong shaded polygonal teapot sitting on a checkerboard. There are two light sources and both the teapot and the checkerboard are reflective. The image for 9264 objects can be seen in Figure 18.

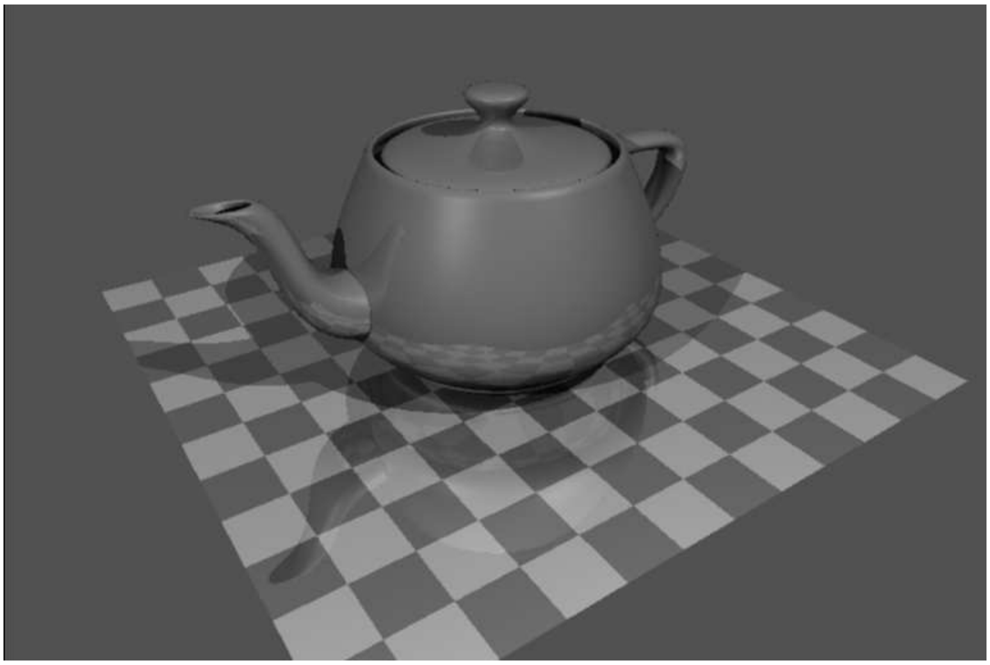


Figure 18: The teapot.

| No. of Objects | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| | bv | gr | kd | oc | bv | gr | kd | oc |
| 9264 | 632 | 335 | 637 | 1342 | 1581 | 3113 | 9649 | 17072 |
| 6420 | 618 | 311 | 615 | 1253 | 1565 | 2684 | 9971 | 19832 |
| 4096 | 584 | 284 | 592 | 1173 | 1544 | 2366 | 13502 | 20489 |
| 2292 | 490 | 299 | 586 | 1101 | 1544 | 2057 | 12520 | 16082 |
| 1008 | 459 | 260 | 602 | 1005 | 1573 | 1804 | 27419 | 14518 |
| 244 | 331 | 256 | 519 | 2094 | 1856 | 1693 | 31072 | 209485 |

Table 3: Timings and intersections for the teapot benchmark

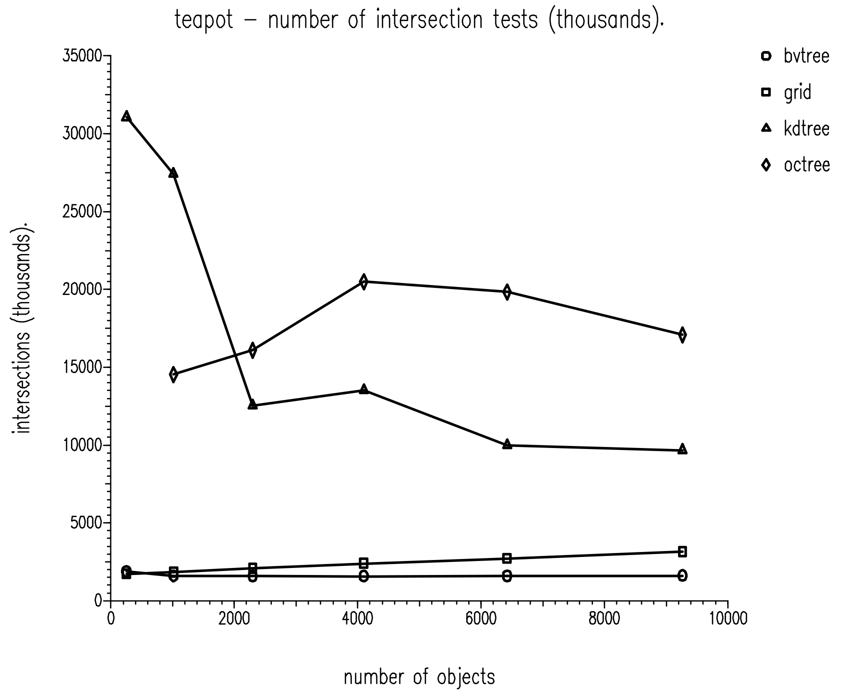



Figure 19: The ray-object intersections for the teapot benchmark

The teapot benchmark is also uniform, and, consequently, favours the grid in the timings. However as the number of objects in the scene increases the size of the polygons making up the teapot decreases. In cases where there are few objects, next to each other, and occupying large amounts of space, both the kd-tree and the octree are less efficient as there is more empty space in the upper branches of the tree, thus the tree has to be searched deeper with each ray, resulting in a longer traversal time and more intersection tests, especially at the start where the objects in the scene, though fewer, result in more empty space being in the trees. The large number of intersection tests for the octree with 244 objects is an example of how degenerate this case can become. The bv-tree's three way split reduces the likelihood of this occurring, and also results in less empty space in the tree, resulting in a consistently lower intersection count for more than 1000 objects, and only a marginally higher one for 244 when compared to the grid. Table 5.4.3 shows the timing and intersection test for the teapot, a graphical representation of the intersection counts can be seen in Figure 19. Note the first intersection count for the octree with 244 objects has

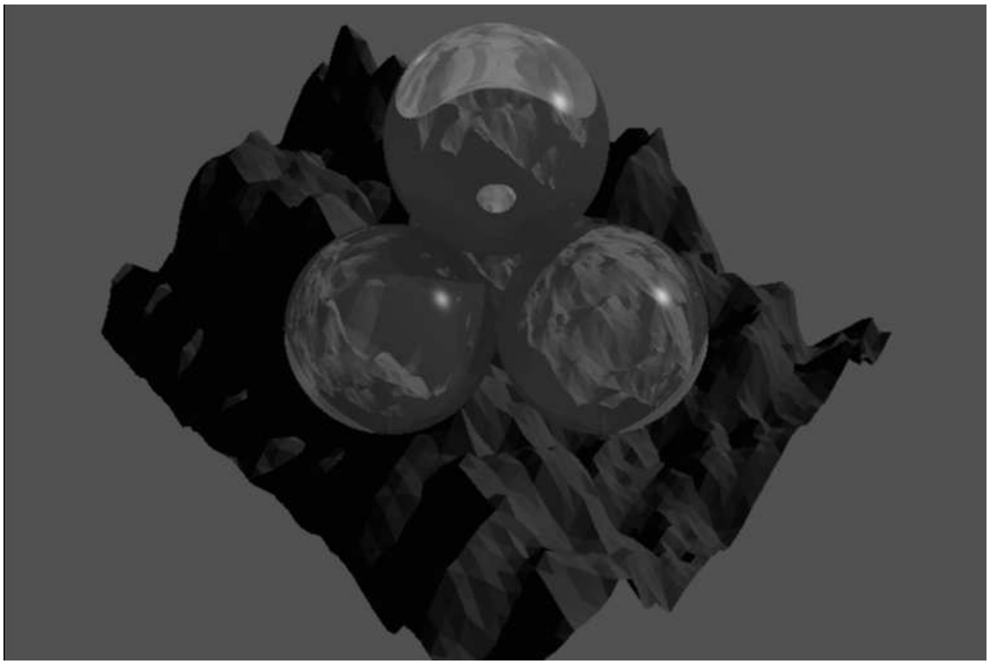


Figure 20: The fractal mountain.

been left out of the graph to prevent scaling from masking the rest of the differences.

### 5.4.4 The Mount benchmark

The mount benchmark consists of a pyramid of 4 glass spheres with a fractal mountain behind it. There is one light source. This scene is generally uniform in the density of objects so it favours the grid. The image produced for the benchmark with 8196 objects can be seen in Figure 20.

In this case the intersection count for the bv-tree comes up higher than the intersection count for the for the grid. The problem here is that the pyramid of four glass spheres does not subdivide nicely as all the spheres are touching, and this results in a tree node with all four spheres in it that also occupies a large portion of the space defined by the mountain and the pyramid. This benchmark introduces the main limitation of the bv-tree structure, in the event where there are a few objects

| No. of | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| Objects | bv | gr | kd | oc | bv | gr | kd | oc |
| 8196 | 473 | 324 | 464 | 1038 | 4485 | 3978 | 9037 | 15232 |
| 2052 | 407 | 278 | 404 | 878 | 4420 | 2956 | 8218 | 14838 |
| 516 | 371 | 267 | 366 | 743 | 4367 | 2540 | 8608 | 17007 |
| 132 | 316 | 264 | 314 | 549 | 4335 | 2295 | 8400 | 22388 |
| 36 | 262 | 265 | 279 | 421 | 4269 | 2141 | 8434 | 36492 |
| 12 | 221 | 288 | 236 | 223 | 4202 | 2073 | 5449 | 12475 |

Table 4: Timings and intersections for the mount benchmark

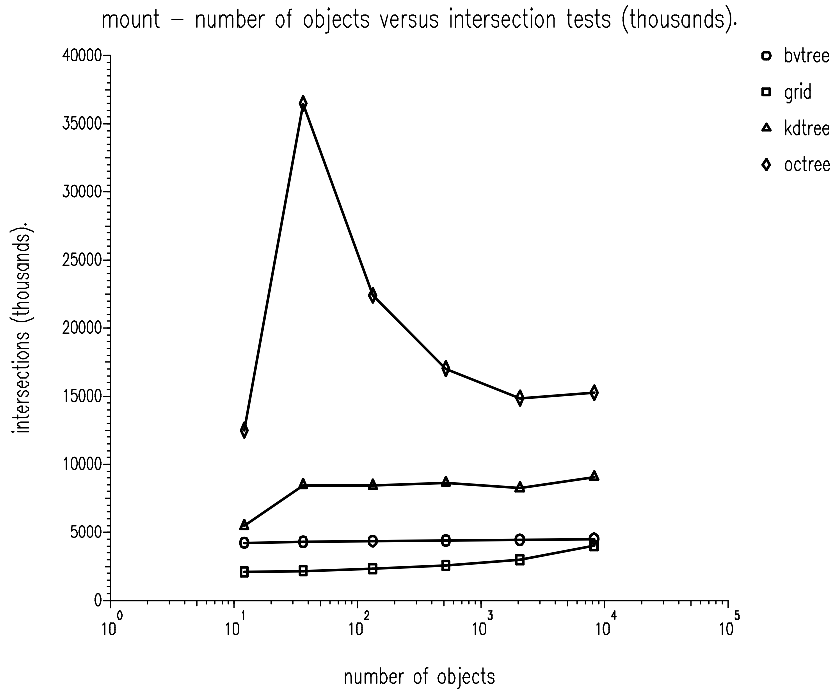



Figure 21: The ray-object intersections for the mount benchmark

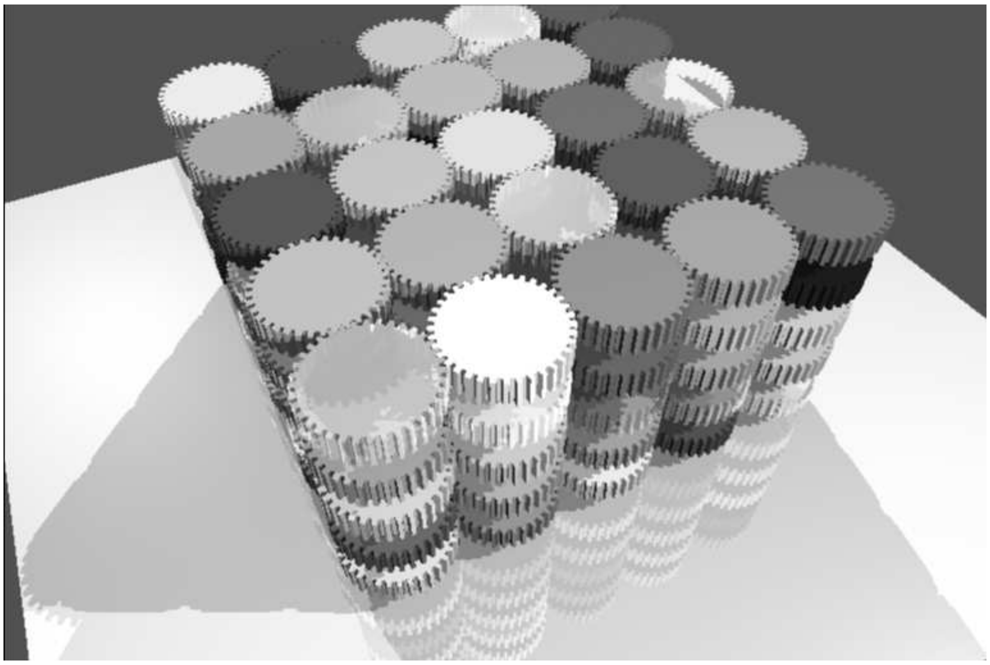


Figure 22: The gears scene.

that occupy a large amount of space in the scene, which are all touching, they end up grouped in a node so that all of them need to be intersected if one of them is. This could be solved by introducing another subdivision method at levels where the node cannot be split and it contains more than one object and occupies more than a certain proportion of the object space. This is a subject for further work but a broad outline of how this can be done can be found in the discussion of *meta-hierarchies* found in the work of Kirk and Arvo in [58]. The timings and intersection counts for this benchmark can be seen in Table 4 and a graphical comparison of the intersection counts can be seen in Figure 21.

### 5.4.5 The Gears benchmark

The gears benchmark consists of a set of gears stacked in columns over a reflecting plane. Each gear face has 144 vertices and some of the gears are transparent. The floor is a reflective polygon and there are five light sources. The image for the gears

| No. of | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| Objects | bv | gr | kd | oc | bv | gr | kd | oc |
| 18251 | 1763 | 1195 | 1698 | 9375 | 4370 | 6353 | 17135 | 718407 |
| 9345 | 1870 | 1128 | 1985 | 4793 | 5631 | 5716 | 15763 | 303569 |
| 3943 | 1525 | 1081 | 1468 | 5085 | 3612 | 5055 | 12009 | 326227 |
| 1169 | 1794 | 1157 | 2735 | 3008 | 11578 | 4829 | 72324 | 89598 |
| 147 | 1633 | 1609 | 1532 | 2797 | 4818 | 5165 | 12351 | 34183 |

Table 5: The timings for the gears benchmark

benchmark with 18251 objects can be seen in Figure 22.

The gears benchmark, like the teapot benchmark, has the objects in it getting smaller as the number of them goes up, in addition as the gears are much larger than the polygons making up the teeth, it also provides occasions where the subdivision breaks down, as in the mount benchmark. Problems with the break down of the subdivision can be dealt with as above, and, in spite of such problems, the bv-tree records a lower intersection count than all the other methods in all cases but one, and in that case (the one for 1169 objects) it is still well ahead of the kd-tree and the octree. The result for the gears benchmark can be seen in Table 5 and a graphical comparison of the intersection counts can be found in the graph in Figure 23.

## 5.5 Conclusions

The bv-tree data structure generally performs at least as well as its nearest hierarchical rival the kd-tree and in some cases better without having the clustering problems associated with the grid data structure. It also exhibits better properties than the kd-tree when confronted with a scene in which the objects are uniformly sized and distributed.

In addition to this, in scenes which are generally uniform with differently sized objects the number of intersection tests performed when the bv-tree structure is used is usually considerably lower than the number performed using other data structures.

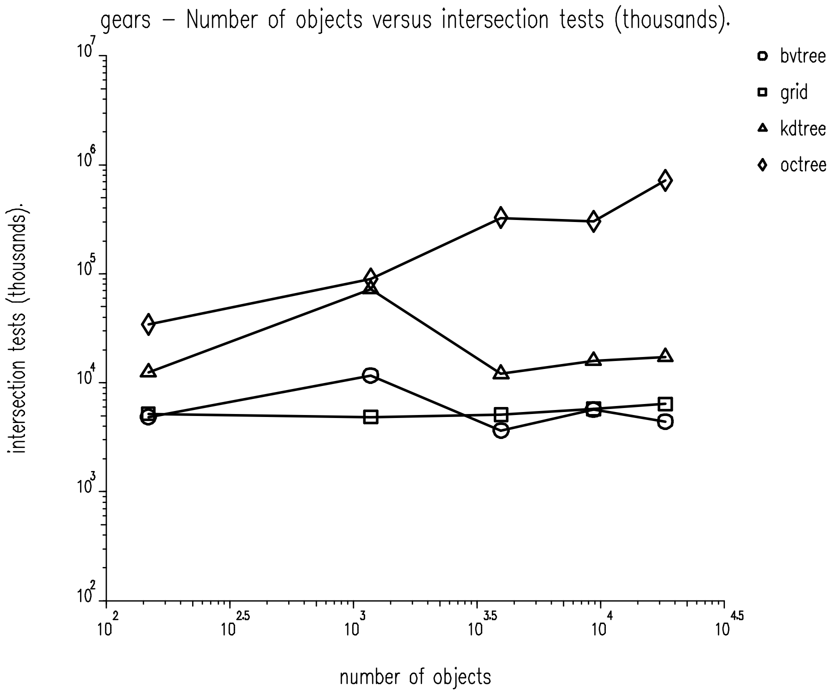



Figure 23: The ray-object intersections for the gears benchmark

The above indicates that the bv-tree is a data structure with some promise, and reducing the cost of traversing it will yield further decreases in the amount of time required to process the object space. The results also indicate that while the performance of the bv-tree does not fluctuate to the same extent as the kd-tree, special cases still exist for which another structure, such as the uniform grid, would be more appropriate.

# Chapter 6

# Conclusion

Two surveys have been provided here concerning methods for the acceleration of ray tracing and both have stories to tell. The survey of acceleration methods used on parallel computers shows three things. First, that using and designing machines for ray tracing with these methods is only just beginning to become economical. Second, that ray tracing is not very effectively accelerated by using vector based hardware. Third, that methods which attain higher efficiency on parallel architectures usually have a general purpose space partitioning algorithm as their basis. The survey of general purpose algorithms which has been carried out shows that for an algorithm to be useful in accelerating ray tracing it should be able to deal efficiently both with scenes in which objects are uniformly sized and distributed and scenes in which objects have wildly different sizes and distributions. In addition to this, if the space partitioning is based on a hierarchy, the tighter it fits the objects in the scene the more efficient it tends to be.

The survey of the literature shows that a lot of work had been done on data structures, and their associated algorithms, for space partitioning but it was not a totally closed issue. To start with, this research provides new insights into the application of the kd-tree which were discovered during the course of this work and are detailed in Appendix A. In addition to this, the work done here in the design

| No. of Benchmark | Timing (seconds) | | | | Intersections (thousands) | | | |
|---|---|---|---|---|---|---|---|---|
| | bv | gr | kd | oc | bv | gr | kd | oc |
| Tree | 307 | 3503 | 322 | 4783 | 518 | 5209 | 2628 | 18311 |
| Tetra | 116 | 73 | 238 | 519 | 524 | 824 | 6061 | 19368 |
| Teapot | 632 | 335 | 637 | 1342 | 1581 | 3113 | 9649 | 17072 |
| Mount | 473 | 324 | 464 | 1038 | 4485 | 3978 | 9037 | 15232 |
| Gears | 1763 | 1195 | 1698 | 9375 | 4370 | 6353 | 17135 | 718407 |

Table 6: Timings and intersections for the largest data sets in each benchmark

and development of algorithms and the data structure required for constructing and traversing the bv-tree shows that it is possible to construct a data structure that provides a tight fit around the objects in a world space being investigated which is not prohibitively expensive to traverse, and, finally, can be generated without assistance of a human operator.

Prior to developing and testing the bv-tree, one major deficiency in prior ray tracing research had to be addressed. The lack of a generally available library for developing and experimenting with new techniques. The ray tracing library in Appendix B was developed during the course of this research to fill this gap. While this idea has been mooted in the literature before (for example Kirk and Arvo [58]), this is the first time software which combines both a NFF parser and a general purpose library containing a variety of acceleration methods such as, the grid, the bv-tree and kd-tree, has been made available to the research community. The software is now publicly available through the ftp gateway of the Department of Engineering Computer Resources at University of Melbourne (currently gondwana.ecr.mu.oz.au) and anyone can use it to either replicate the work given here, or further investigate other ray tracing acceleration techniques.

The data structure and the associated algorithms for the bv-tree presented in chapter 4 can be seen from the summary of the results in chapter 5 presented in Table 6 to provide an efficient adaptive data structure for bounding collections of objects and tracing rays through the space they occupy. In terms of adaptive data structures Table 6 shows the bv-tree to be generally faster than its nearest rival, the kd-tree, except in the case of the Mount and Gears benchmark in which case the difference is still less than ten percent. In addition to this, all cases of the bv-tree show substantially lower intersection counts than the other adaptive structures, indicating that further work on the reducing the cost of traversing the tree structure will yield noticeable improvements in the speed of rendering.

The bv-tree tails the grid in some of the benchmarks used here, but it has the advantage that, unlike the grid, it can be created on the fly, rather than having to be done in a preprocessing phase. A grid must be assembled before any rays are traced in the volume it occupies, and it must remain static for the duration of the ray tracing. Due to this the use of a grid introduces extra initial memory overheads and these overheads are present even if the rays generated during a ray tracing traverse only a small part of the grid. The hierarchy formed by the bv-tree also guarantees that a given object will appear in only one leaf node, consequently, unlike all the other methods presented in this thesis, it does not require memory to be used replicating object pointers and it does not require the memory to store "mail box" ray information to be allocated with each object.

## 6.1 Areas For Future Research

As improvements are constantly being made in processor technology, it is likely that more and more applications will try utilising parallel processing as well as more conventional methods to decrease the amount of real time taken to perform a ray tracing. As other algorithms based on other hierarchical data structures have been transferred onto parallel architectures, and as the bv-tree is similar to these in kind, it appears likely that the bv-tree algorithm should be portable as well. This is an area for further research.

In terms of examining the timing differences between the grid and the bv-tree we can see that the bv-tree still tails in benchmarks where the objects in the data set are, more or less, uniformly distributed. In the case of the Mount benchmark we can also see from the intersection count that a situation has arisen in which the hierarchy created by the bv-tree is less than ideal. While the results here indicate that the bv-tree represents an improved data structure for handling non uniformly distributed sets and can also generally handle very uniform data better than the other hierarchical data structures given here, they also indicate that it is unlikely that a general "best case always" algorithm currently exists, and that an area of research that would enable greater throughput in rendering would be in investigating methods which analyse areas of a scene and determine, prior to rendering, the data structure, or the combination of data structures, which should be used to efficiently subdivide them for ray tracing. In the experience of this author, with architectural models, "real life" scenes tend to be a mixture of non uniform and uniform distributions of objects so it is likely that best results would be obtained if a combination of subdivision methods could be employed.

The idea of using a combination of methods for accelerating ray tracing has already been put forward in the work of Kirk and Arvo [58], and some initial work has already been done trying to develop a cost analysis of using a particular data structure to subdivide a scene, as in the work of Subramanian and Fussell [92], or Snyder and Barr [89], however the problems involved in developing a formal method for adaptively building a meta-hierarchy based on an analysis of the cost of ray tracing a given set of objects, especially in terms of secondary rays, remains largely untouched. For example, with the Mount benchmark the intersection count with the bv-tree goes up as it fails to subdivide a region containing four touching spheres which also happen to occupy more than a quarter of the scene space. While this is an extreme case, such a situation in a meta-hierarchy based on the bv-tree could be handled using the following heuristic: if a leaf node occupies more than 10 percent of the world space and contains more than one object and cannot be subdivided as a bv-tree use some other subdivision method.

The above heuristic is a bit simplistic. If there are only two objects in the leaf node it would probably be more efficient to intersect both of them than to introduce the overheads involved in another subdivision method, but it does indicate the kinds of decisions and circumstances that are involved in changing subdivision methods. Further research is required, using a wider variety of data sets than presented here, to adequately test and develop the heuristics required to construct an efficient meta-hierarchy and it is likely that the understanding resulting from such research would probably yield the biggest increases in efficiency of ray tracing.

## 6.2 Peroration

This research was undertaken as a means of investigating the new algorithms involved with the bv-tree and with a view to finding out the kind of answers it provided to the problem of accelerating ray tracing. While it is shown that the bv-tree provides some answers to some cases, some other useful things gained from the research done in this thesis are the insights into the problem of accelerating ray tracing which have opened up areas for further research.

Last night we owned, with looks forlorn,
"Too well the scholar knows
There is no rose without a thorn"-
But peace is made! We sing, this morn,
"No thorn without a rose!"

A Lesson In Latin - Lewis Carroll.

# Appendix A

# The kd-tree paper

# Using Kd-trees to Guide Bounding Volume Hierarchies for Ray Tracing

David Hook and Kevin Forward
Dept. Of Electrical Engineering
University of Melbourne

May 25, 1995

## Abstract

Ray tracing is a simple yet powerful technique for investigating complex environments and a standard method of accelerating the ray casting process has involved constructing bounding volume hierarchies which have required either the use of precalculated costs or creating them as part of the model construction. We propose a method using the kd-tree which will guide the construction of a bounding volume hierarchy as well as providing a method for its traversal by individual rays. By taking advantage of the tree structure's recursive nature we show how to construct an algorithm with very good performance characteristics which in typical real life cases is much more computationally efficient than previous methods. Results are presented using some standard ray tracing benchmarks.

Keywords: graphics, ray tracing, kd-tree, bounding volume hierarchy.

CR categories: E.1, I.3.7.

## 1 Introduction

Use of ray tracing as a technique probably goes back to early work on optics (the earliest use of ray tracing the authors are aware of is Galileo's work on optics in 1609). Since the advent of computers it has been used in seismology (density checking along the ray), engineering, robotics, and plant design (moments of inertia, interference checking), modeling of plant growth in botany, medicine (radiation treatment planning), and image generation computer graphics. Since its appearance in computer graphics [15] ray tracing has established itself as an important means for generating realistic imagery.

Central to finding a solution to a problem using ray tracing is the ease with which the entities lying on the path of the ray can be found. The brute force method, which involves checking every entity in the space being investigated, is often prohibitively expensive and as a result, various data structures and algorithms have been suggested so that most of the objects which a ray is not going to intersect can be quickly discarded and only those objects which are likely to lie on the path of the ray require the expense of doing a ray-object intersection test. Often these are based around space partitioning or bounding volume hierarchies and examples of both approaches can be found in [1] [6] [7] [8] [10] [11] [16].

In this paper, we present a space partitioning scheme based around the kd-tree, a structure first proposed for range searching by Bentley [3]. Object space is partitioned in the X, Y, and Z, dimensions but unlike previous investigations [7] we have placed an emphasis on shrinking the space occupied by each level of the tree rather than attempting to divide the object space so that a similar number of objects lie on each side of the partitioning plane. We present an adaptive algorithm for traversing and building up this structure as a ray travels through it. Recently the work of McNeill *et al.* [13] has shown that adaptive structures make much better use of the available memory and we present an adaptive algorithm for traversing and building up the tree structure as the ray travels through it, rather than using the tree to generate a grid-like structure that must be preallocated at the start of the rendering. Results obtained by using this approach show that it is one of the more efficient algorithms currently known and adapts very well to worlds in which there is a non-uniform distribution of objects, very often the case for many of the models people wish to visualise such as office interiors and building projects.

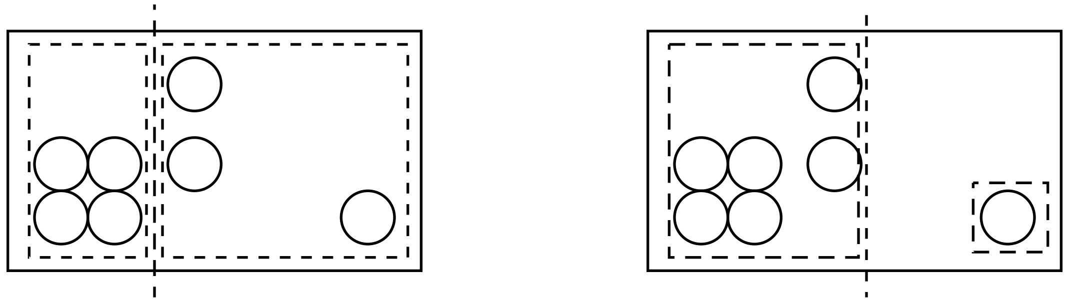
## 2 Previous Work

Early work on binary space partitioning (BSP) trees for ray tracing was done by Kaplan [11] with space tracing. Unlike the original BSP trees used by Fuchs *et al.* [4] the partitioning plane was restricted to be axis aligned and a voxel with too many objects in it was partitioned into eight equally sized subvoxels. This gave a similar partioning scheme to that of an octree as in [8] but saved on the requirement of hashing and caching schemes that were required for the location of voxels or lists of candidate objects in schemes such as [8] [6] and [1]. Like the earlier schemes tracking a ray through the tree involved pushing the ray into the next voxel, thus it was iterative in nature.

More recently Jansen [10] adapted this algorithm further so that rather than iterating through the tree, the tree was traversed recursively by clipping the ray according to the near and far side of the partitioning plane. This simplifies the algorithm making it easier to prune subtrees the ray does not intersect and using the stack to handle the backtracking required when the neighbour of a voxel must be checked if nothing in the current voxel has been intersected.

Fussell and Subramanian [7] built further on this work by developing recursive algorithms for constructing and traversing kd-trees. The kd-tree is attractive as it doesn't require the partitioning plane to lie mid way between the boundaries of the space to be partitioned so it adapts better to different environments. In the case of [7] the partitioning plane was calculated using a median cut so that each partition plane had approximately the same number of objects at each level. This algorithm performs quite favourably [14] but for environments where objects appear in clusters scattered non-uniformly throughout the object space it can result in large amounts of empty space appearing in the tree, which can lead to branches of the tree not being pruned until quite late (see Figure 1). Ideally the branches of the tree should shrink around the object space as each leaf node is subdivided.

Consideration of this led us to two conclusions, firstly it is not necessarily the case that all or even any branches of the tree should have equal number of objects on either side, secondly that it may be worth the computational expense of recalculating the bounding volume of each tree branch as a leaf node is subdivided.



a. The median kd-tree b. The space-split kd-tree

Figure 1: The results of the two kd-tree algorithms in splitting an identical scene

## 3 Terminology

The algorithms in this paper have been written in a functional style for ease of expression. The fundamental operators `Type`, `BBox`, `SplittingPlane`, `Left`, and `Right` return the following corresponding items respectively: the type of the node (which may be `Leaf`, `SplittableLeaf`, or a branch node, denoted by having a type of `X`, `Y`, or `Z`), the bounding box of the node, the plane along which space is divided in the node, the left subbranch of the node and the right subbranch of the node. The ray operators `Origin` and `Direction` return the origin point and the direction vector of the ray. Where operators are further indexed as in

```
Origin(r)(X)
```

the value of the result comes from applying the operator and then the index, in the example, the X coordinate of the point of origin.

Finally, the use of the words left and right when applied in the context of spatial subdivision are used respectively to denote towards negative infinity and towards positive infinity relative to the point in space being mentioned.

## 4 Tree Construction and Traversal

Unlike previous bounding volume hierarchies such as Weghorst *et al.* [16] where the hierarchy is built up by the user with the model containing the entire set of objects, this algorithm is adaptive and is driven by the ray casting. This has two advantages. Unlike the methods proposed in [11] and [7], calculating the the bounding box of the environment is the only preprocessing done before ray tracing is commenced and we recalculate bounding boxes as the tree is constructed and if a ray enters the region. This is computationally efficient and, as splitting a leaf node normally results in some objects appearing on both sides of the splitting plane it avoids unnecessary duplication of object pointers. This reduces the amount of memory required, allowing bigger and more complicated object spaces to be processed in a given memory space.

We start tree construction in a similar fashion to Kaplan [11], with a leaf node containing all the objects in the environment. The leaf node is marked as splittable and its bounding volume is a box representing the extent of the environment. It should be noted that as raised in [16] boxes aren't necessarily the best bounding volumes nor is it a requirement for the tree structure and its associated algorithms that boxes be used as bounding volumes. We chose boxes for their simplicity, and the optimisation of the shape of the bounding volumes at each level of the tree is an area for further research.

### 4.1 Traversal of the Tree.

When a ray enters a node of the tree, one of three things can happen.

1. The node is a leaf node, in which case the objects in it are tested against the ray. In the algorithm given below the function `Checkleaf` is meant to perform the testing of the objects in the leaf node with the ray.
2. The node is a splittable leaf node, in which case the node is split and the ray recurses to the subbranches that were created.
3. The node is a branch node (ie. it's type represents one of the three axes) in which case the origin of ray in the axis of the node's splitting plane is checked against the subbranch on the nearest side first and, if no intersection results the ray is then checked against the other subbranch.

The basic algorithm for this is as follows:

```
Hit
Traverse(r, root)
  Ray    r;
  Node   root;
{
  type := Type(root);

  if (type is Leaf)
    return(CheckLeaf(r, root));
  else if (type is SplittableLeaf) {
    Split(root);
    return(Traverse(r, root));
  } else {
    org := Origin(r)(type);
    dir := Direction(r)(type);

    left := Left(root);
    right := Right(root);

    if (org < SplittingPlane(root)) {

      if (Missed(r, BBox(left)))
        hit := NullHit;
      else
        hit := Traverse(r, left);

      if (hit <> NullHit)
        return(hit);

      if (Missed(r, BBox(right)))
        hit := NullHit;
      else
        hit := Traverse(r, right);

    } else {

      if (Missed(r, BBox(right)))
        hit := NullHit;
      else
        hit := Traverse(r, right);

      if (hit <> NullHit)
        return(hit);

      if (Missed(r, BBox(left)))
        hit := NullHit;
      else
        hit := Traverse(r, left);

    }

    return(hit);
  }
}
```

Note that this algorithm is a little naive in some respects; several improvements can be made so that branches of the tree can be pruned without having to check the bounding boxes of the nodes. For example, when traversing the tree from left to right, if we fail to find an intersection in the left subbranch and the direction of the ray in the axis of the current node is less than or equal to zero, the ray will never enter the right subbranch. A similar rule applies when going from right to left, and it is possible to use this knowledge about the ray's direction to further simplify the bounding box checks on the second subbranch to be traversed.

### 4.2 Splitting a Leaf Node.

When a ray enters a splittable leaf node the basic problem is choosing a splitting plane. We do this by calculating the spatial median for each axis, that is, the point midway between the maximum and minimum values of the bounding volume on that axis. This gives us three axis aligned planes which divide that bounding box of the leaf equally. We then choose the plane that cuts through the smallest number of objects. This minimises the number of objects which must be referenced on both sides of the branch we are creating. We do one further test where, in the event that two planes intersect the same number of objects, we choose the plane that most closely approximates a median division of the objects. In the event that more than a certain percentage of objects are intersected by all of the dividing planes the subdivision is terminated. Currently we have found that a starting figure of thirty percent works well. As the tree is progressively built and the volume of the leaf nodes gets smaller and smaller this figure is eventually reduced. At higher levels of the tree some inefficiency can be tolerated but as the leaf nodes become progressively smaller a more conservative splitting strategy is required. The reason for this is that the cost of traversing a subbranch should always be less than directly intersecting the objects that are contained in its leaves.

Another approach to choosing a dividing plane is through the use of heuristics. MacDonald *et al.* [12] describe two heuristics for constructing trees, one using surface area as an estimate of probability of intersection, and another based on the fact that the optimal dividing plane lies between the spatial median and the object median planes. Our approach was chosen as it is simple to implement, and, as will be seen later, in conjunction with continually shrinking the bounding volume of the leaf nodes as the tree is created, it still provides acceptable performance.

If a node has been selected as a suitable candidate for splitting, the objects are sorted into three lists: objects to the left of the splitting plane, objects to the right and objects divided by the splitting plane. For simplicity we currently do all these tests relative to the object's bounding box. Once the objects are divided up the left and right lists are assigned to the newly created left and right subbranches of the node being split and the divided objects are added to both the left and right subbranches. This need not be as big a strain on memory as it might sound. In our case we use lists of pointers to objects in the kd-tree, consequently adding a divided object to both sides of the splitting plane only incurs an overhead of around eight bytes.

Finally if this puts the newly created subbranches at greater than the maximum level we desire for the tree or there are fewer than some minimum number of objects in a subbranch we mark the qualifying subbranches as unsplittable leaf nodes, otherwise we mark them as splittable, and then, after calculating

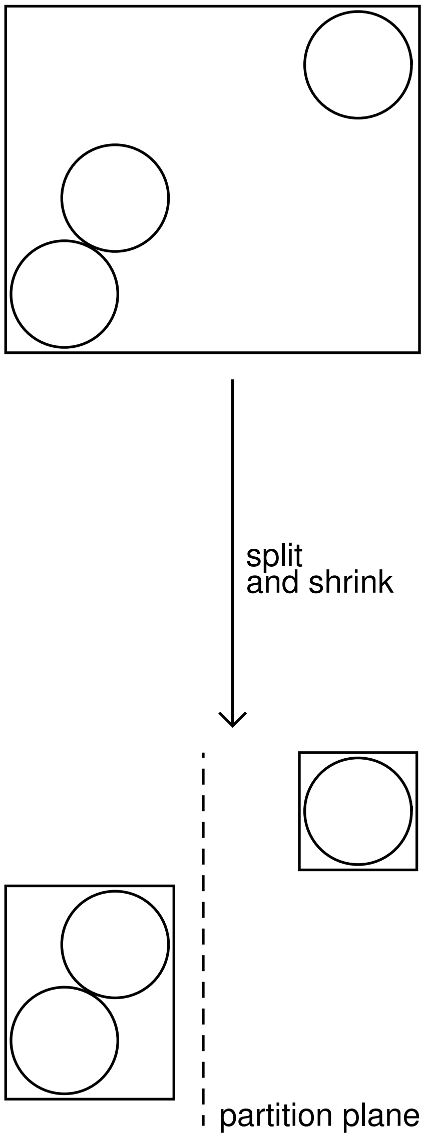



Figure 2: The splitting process.

the bounding boxes for the two subbranches, we return to the traversal routine.

The calculation of the bounding box dimensions for a leaf node as it is created has shown to result in major savings in the computations required to traverse the tree the next time through. In addition to this several of the calculations described above can be done together, rather than in several steps, which further reduces the computational cost involved in constructing and traversing the tree. Readers interested in a detailed evaluation of costing strategies should refer to K. Subramanian and Fussell [14].

## 4.3 Checking a Leaf Node.

All the objects in the leaf node are tested against the ray, and the closest object intersected by the ray is returned if the hit is valid. As objects can lie in more than one leaf node it is necessary to ensure that the hit is also in the node. This is a common problem with tree structures and unnecessary intersection tests can be avoided by tagging each object tested with a unique ray identifier, and storing the information pertaining to the hit in the object as mentioned in the work of Glassner [8]. Then instead of testing a ray against an object for the second time the hit data from the previous test of the ray with the object is used in order to avoid another, potentially expensive, intersection test.

Another problem is determining whether a hit is valid or not. This routine should only check that the point of intersection doesn't result in a distance which pushes the ray passed the bounding volume of the node. Occasionally hits occur that appear to be slightly before the bounding volume of the node, these hits should be treated as valid, as this problem arises due to the limited precision of floating point operations, not because the intersection is outside the bounding volume.

## 5 Results

We have used the nff (neutral file format) input files made available by Eric Haines and described in [9]. While it is often said that all benchmarks are suspect these files are readily available from a large number of ftp sites on the internet and provide some of the classic worst and best case scenarios that can be encountered in rendering.

In this case a general purpose rendering library was written and each different subdivision technique laid on top of it. The rendering library uses a shadow cache and no object that has been intersected once will be tested when its shadows or reflections are being calculated. If an intersection was detected for a shadow ray and the point hit was closer than the light source the intersection was automatically accepted. The number of intersections are in thousands rounded up and the second number, where given, is the number of bounding box tests (see Table 1). The CPU time figure is in seconds. All the models were generated with their default parameters.

Finally, both the kd-tree described in this paper and the median kd-tree described by Fussell and Subramanian [7] were allowed to extend to a depth of 30. In order to provide a reference point for others to compare our figures, timings for two common methods of subdivision, an octree and a grid, are provided as well. The uniform grid divided the object space into a 20 x 20 x 20 grid, and the octree was allowed to proceed to a depth of 10. It should be noted here that the traversal of octree was not as efficient as it could be (the implementation used here is recursive), but for an octree of similar size the intersection counts will remain the same, and, as these are substantially larger than for the two kd-tree implementations, reducing the cost of traversal of the octree is not going to make it competitive with a kd-tree. Readers interested in more efficient octree traversal are referred to the work of Glassner [8] and Fujimoto *et al.* in [6].

The timings were done on an otherwise idle Sun SPARCstation 2 with 32 megabytes of memory.

### 5.1 Balls.nff

This model is of a fractal "sphereflake". While the objects in it are distributed in a semi-uniform fashion it has a large base polygon which causes some clustering in the grid algorithm. In the median-kd tree case the partitioning according to the number of objects results in large amounts of empty space in the top of the tree, so the cost of using the data structure is more expensive than the kd-tree which divides up according to space.

### 5.2 Tree.nff

This database contains a fractal tree grown using spheres and cones as the basis, together with a base polygon that extends well beyond the range of the tree. This results in severe clustering in the grid and as grids suffer badly from clustering even the octree comes out ahead compared with the grid method. Once again the median-kd tree suffers because there is more empty space in the upper levels of the tree.

### 5.3 Gears.nff

This model has a collection of gears stacked one on top of the other, with a slightly extended backing polygon. Unlike the scene contained in Tree.nff, this is a model which greatly favours the uniform grid as there is very little empty space in the world space and object density is uniform. For this reason the grid is faster. Owing to the distribution of objects in the scene we would also expect that the timings for the two kd-trees would be similar

| Benchmark | Structure | Polygons | Spheres | Cones | CPU Time |
|---|---|---|---|---|---|
| balls.nff | kd-tree | 240 | 1036/5128 | —- | 984.50 |
| | median kd-tree | 211 | 1028/8471 | —- | 1218.32 |
| | uniform grid | 202 | 1074/4311 | —- | 4137.27 |
| | octree | 203 | 1255/31225 | —- | 7369.62 |
| tree.nff | kd-tree | 165 | 17/298 | 258/1897 | 414.00 |
| | median kd-tree | 187 | 17/673 | 259/1709 | 970.77 |
| | uniform grid | 146 | 19/3892 | 267/4880 | 7454.25 |
| | octree | 175 | 250/5746 | 401/11971 | 6642.17 |
| gears.nff | kd-tree | 15763 | —– | —– | 2503.07 |
| | median kd-tree | 22918 | —– | —– | 2465.93 |
| | uniform grid | 5733 | —– | —– | 1481.43 |
| | octree | 303576 | —– | —- | 6492.08 |
| mount.nff | kd-tree | 5475 | 760/2802 | —– | 597.97 |
| | median kd-tree | 6884 | 724/1738 | —– | 1116.42 |
| | uniform grid | 2563 | 688/728 | —– | 436.53 |
| | octree | 11838 | 775/2620 | —– | 1390.17 |

Table 1: Intersection and timing comparisons for the four algorithms.

(within five percent), which they are. The two kd-tree algorithms will create different trees, hence the variation in the intersection counts. Owing to the uniformity of the object space, however, as the intersection count goes down the cost of traversing the tree must go up, and as the cost of intersecting polygons is comparable to the cost of traversing a branch node the timings are also comparable. The variation in intersection counts also indicate that reducing the cost of a traversing a tree based on the spatial median could lead to a much faster rendering of the scene than reducing the traversal cost of a tree based on the median of the object distribution. While this is an area for further work one approach being investigated is making the traversal of the kd-tree iterative by using an explicit stack, an approach to rendering first suggested by Jim Arvo [2].

### 5.4 Mount.nff

This model contains a fractal mountain range with three large transparent spheres in front of it. This gives a scene with three very big objects at the front and a uniform distribution of little ones at the back. The grid comes out ahead here due to the lesser number of sphere intersection tests and the uniformity of the distribution of polygons in the fractal mountain in the background. The median-kd tree fairs badly here as dividing up according to the number of objects on either side of the splitting plane results in a much higher cost in traversal because a lot of voxels in the tree will have a large amount of empty space in them. Once again the kd-tree algorithm shows the worth of being more adaptive.

## 6 Conclusions

While no algorithm is likely to be ideal for all circumstances in ray tracing, a bounding volume hierarchy guided by a kd-tree provides an efficient algorithm with which to render a scene. This algorithm is especially suited to scenes with a non-uniform distribution of objects, a situation which in our experience is normal for most models. Our results show that the algorithm we have described here is sometimes an order of magnitude faster than the one based on the uniform grid and up to two times faster than an algorithm based on a median kd-tree. In the one case where the median kd-tree shows better performance the results are so close that the difference may be ignored since timings on a Sun SPARCstation 2 can vary by up to five percent on individual runs.

When compared with the octree a kd-tree based on the spatial median always results in a much lower intersection counts and in the case where both trees are implemented using recursion the result is substantially faster timings as well. In the case of the uniform grid, it does outperform the kd-tree structure in situations where there is a large degree of uniformity in most, or all, of the object space, however, the worst case performance of the kd-tree versus the grid for the same object set is substantially closer than the worst case performance for the grid versus the kd-tree. Additionally, as more non-uniformity is introduced into the distribution of objects in the scene being rendered the timings of the kd-tree based on the spatial median improves at a greater rate than the kd-tree based on the object median.

A C version of the kd-tree intersection algorithm is available for anonymous ftp from gondwana.ecr.mu.oz.au in the file pub/vort.tar.Z. The nff file benchmarks can also be found on gondwana in the file pub/SPD.3.0.shar.Z. People interested in obtaining the rendering library can find it in pub/libRT.tar.gz on the same machine.

## 7 Acknowledgements

The authors wish to acknowledge the valuable comments made by the reviewers of this paper. They greatly helped clarify the original work.

## References

[1] James Arvo and David Kirk. "Fast ray tracing by ray classification." Computer Graphics, 21(4):269-278, July 1987.

[2] James Arvo. "Linear-Time Voxel Walking for Octrees.", Ray Tracing News, electronic newsl etter edited by Eric Haines, Vol. 1, no. 2, 1988.

[3] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching." Communications of the ACM, 18(9), September 1975.

[4] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. "On visible surface generation by a priori tree structures." Computer Graphics, 14(3):124-133, July 1980.

[5] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. "Near Real time shaded display of rigid objects." Computer Graphics, 17(3):65-72, July 1983.

[6] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. "ARTS: accelerated raytracing system." IEEE Computer Graphics and Applications, 6(4):16-26, April 1986.

[7] Donald Fussell and K. R. Subramanian. "Fast ray Tracing using k-d trees." Technical Report TR-88-07, Dept. of Computer Sciences, University of Texas at Austin, 1988.

[8] Andrew S. Glassner. "Space subdivision for fast ray tracing." IEEE Computer

Graphics and Applications, 4(10), October 1984, p. 15-22.

[9] Eric A. Haines. "A proposal for standard graphics environments." IEEE Computer Graphics and Applications, November 1987, p 3-5.

[10] Frederick W. Jansen. "Data Structures for Ray Tracing." Data Structures For Raster Graphics, Springer Verlag, 1986, p. 57-73.

[11] Michael R. Kaplan. "The Use of Spatial Coherence in Ray Tracing." Techniques for Computer Graphics Graphics, Ed. Rogers and Earnshaw, Springer-Verlag, 1987, p. 173-193.

[12] MacDonald, J. David, and Kellog S. Booth. "Heuristics for Ray Tracing Using Space Subdivision.", The Visual Computer, Vol 6, 1990, p. 153-166.

[13] McNeill, M. D. J., B. C. Shah, M. P. Hébert, P. F. Lister, and R. L. Grimsdale. "Performance of Space Subdivision Techniques in Ray Tracing.", Computer Graphics Forum, The Eurographics Association, October 1992, p. 213-220.

[14] K. R. Subramanian and Donald S. Fussell. "Automatic Termination Criteria for Ray Tracing Hierarchies." Proceedings of Graphics Interface '91, Calgary, Alberta, 1991.

[15] Turner Whitted. "An Improved Illumination Model for Shaded Display." Communications of the ACM, 23(6), June 1980, p. 343-349.

[16] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing." ACM Transactions on Graphics, 3(1), January 1984, p. 52-69.

# Appendix B

# The libRT manual page

**NAME**

RT − A Ray Tracing library for producing raster images.

**SYNOPSIS**

**#include <rt.h>**

**DESCRIPTION**

*RT (3)* is a library of C routines for defining and rendering scenes composed of 3-dimensional objects. Currently the library produces images in the *VORT (3)* format.

### Special Requirements.

The *rt* library requires the routine *fatal* to be defined so that it can flag errors that are unrecoverable, from the renderer's point of view. In the simplest case this routine can be defined as follows:

```
/*
 * fatal
 *        print out the error message and exit.
 */
fatal(s)
        char *s;
{
        printf("%s\n", s);
        exit(1);
}
```

The *rt* library also has facilities for logging statistics, to allow for the testing and debugging for new space partitioning schemes to accelerate the rendering process. To this end it also needs to have the variable *logfp* defined. If this variable is non-NULL it is assumed to point to a valid file and the logging results are dumped in there at the end of the rendering.

### Initialisation

A single call is required to set up the necessary function pointers to allow the rendering to take place, as well as setting up defaults for details such as the background colour, field of view, etc...

*rendinit()*

sets up the necessary structures to allow the rendering to take place. It should be the first call made.

```
rendinit()
```

*imagesize(w, h)*

sets the width and height (w and h) of the raster image generated by the renderer. This call can be made anywhere in a program before the call to render (see below) is made. The default image size is 512 by 512.

```
imagesize(w, h)
        int        w, h;
```

### Functions to Define Basic Objects.

*sphere(x, y, z, r)*

adds a sphere with the centre (x, y, y) and a radius r to the current object list.

```
void
sphere(x, y, z, r)
        float    x, y, z, r;
```

*polygon(count, xs, ys, zs)*

adds a polygon with count vertices to the object list. The vertices are provided in the arrays of floats pointed to by xs, ys, and zs.

```
void
polygon(count, xs, ys, zs)
        int      count;
        float    *xs, *ys, *zs;
```

*polygonpatch(count, xs, ys, zs, nxs, nys, nzs)*

adds a polygon patch with count vertices to the object list. Polygon patches differ from polygons in that they are phong-shaded, with the surface normal of the polygon being calculated by interpolating normal values given in the arrays nxs, nys, nzs. As with the polygon the x, y, and z coordinates of the vertices are provided in the arrays of floats pointed to by xs, ys, and zs.

```
void
polygonpatch(count, xs, ys, zs, nxs, nys, nzs)
        int      count;
        float    *xs, *ys, *zs;
        float    *nxs, *nys, *nzs;
```

*cone(bx, by, bz, br, ax, ay, az, ar)*

adds a cone to the object list. The base x, y, and z coordinates for the cone are given in bx, by, bz, and the base radius is given in br. The apex x, y, and z coordinates are given in ax, ay, and az, and the apex radius is given in ar.

```
void
cone(bx, by, bz, br, ax, ay, az, ar)
        float    bx, by, bz, br;
        float    ax, ay, az, ar;
```

*cylinder(bx, by, bz, tx, ty, tz, r)*

adds a cylinder to the object list. The base x, y, and z coordinates for the cylinder are given in bx, by, bz, and the top x, y, and z coordinates are given in tx, ty, tz. The radius of the cylinder is given by the value of r.

```
void
cylinder(bx, by, bz, tx, ty, tz, r)
        float    bx, by, bz;
        float    tx, ty, tz, r;
```

### Setting up Hierarchies and Grids for Faster Raytracing

By default objects are simply added to a list as they generated. In a situation with a large number of objects simply ray tracing a list of objects will be too time consuming so there are a number of partition schemes that can be used. Open and closing a partition results in a single object, so in cases where a couple of different methods are required over a seen partitions may be opened and closed as neccessary, even within other partitions.

*openbvtree()*

opens a bounding volume tree, every object defined after a call to openbvtree will be placed in the bounding volume tree's list.

```
void
openbvtree()
```

*closebvtree()*

closes a bounding volume tree. At this point every object defined since the last call to openbvtree is amalgamated into a single object representing a bounding volume tree.

```
void
closebvtree()
```

*gridsize(xd, yd, zd)*

sets the dimensions for a uniform grid. The dimensions of the grid in the x, y, and z directions are defined by xd, yd, and zd respectively. By default the dimensions are xd = 20, yd = 20, and zd = 20.

```
void
gridsize(xd, yd, zd)
        int     xd, yd, zd;
```

*opengrid()*

opens a uniform grid, every object defined after a call to opengrid will be placed in the uniform grid's list.

```
void
opengrid()
```

*closegrid()*

closes a uniform grid. At this point every object defined since the last call to opengrid is amalgamated into a single object representing a uniform grid.

```
void
closegrid()
```

*openkdtree()*

```
void
openkdtree()
```

opens a kd-tree, every object defined after a call to openkdtree will be placed in the kd-tree's list.

*closekdtree()*

closes a kd-tree. At this point every object defined since the last call to openkdtree is amalgamated into a single object representing a kd-tree.

```
void
```

closekdtree()

*openoctree()*

opens a oc-tree, every object defined after a call to openoctree will be placed in the oc-tree's list.

```
void
openoctree()
```

*closeoctree()*

closes a oc-tree. At this point every object defined since the last call to openoctree is amalgamated into a single object representing a oc-tree.

```
void
closeoctree()
```

**Viewing parameters**

*viewup(x, y, z)*

sets the view up vector. The three arguments represent the x, y, and z components respectively. The up vector components do not need to be normalised.

```
viewup(x, y, z)
        float   x, y, z;
```

*viewlookat(ex, ey, ez, rx, ry, rz)*

sets the eye point and reference point (ie. the point been looked at) by the viewer. The arguments ey, ex, and ez, are the x, y, and z components of the eye point respectively. The arguments rx, ry, and rz are the x, y, and z components of the reference point respectively. The direction vector (rx - ex, ry - ey, rz - ez) should not be parrallel with the up vector.

```
viewlookat(ex, ey, ez, rx, ry, rz)
        float           ex, ey, ez;
        float           rx, ry, rz;
```

*viewfov(fov)*

sets the field of view for the current view. The default field of view is 45 degrees.

```
viewfov(fov)
        float   fov;
```

**Starting the Render.**

Once all the objects and viewing parameters have been defined it is possible to render the scene.

*render(imname)*

renders the current object space and places a raster image of it in the file named imname. The size of this file is set using the imagesize call.

```
render(imname)
        char      *imname;
```

**Miscellaneous Routines.**

The following routines are provided for their general usefulness.

*smalloc(size)*

same as unix malloc, only it flags an error the moment NULL is returned.

```
char *
smalloc(size)
        unsigned        size;
```

*scalloc(size, n)*

same as unix calloc, only it flags an error the moment NULL is returned.

```
char *
scalloc(size, n)
        unsigned        size;
        unsigned        n;
```

**Matrix Stack Operators**

The matrix stack allows users to define a couple of transformations that only affect the objects defined between the transformation and the next popmatrix.

*pushmatrix()*

pushes the current transformation matrix onto the matrix stack.

```
pushmatrix()
```

*popmatrix()*

pops the matrix stack, restoring the current transformation matrix to what it was before the last pushmatrix.

```
popmatrix()
```

**Transformation Routines**

The following transformation routines all affect the current transformation matrix. All tranformations are premultiplied, and in all cases the variables x, y, and z represent values for the x, y, and z axes respectively.

*translate(x, y, z)*

translates the current coordinate system by the values of x, y, and z.

```
void
```

```
translate(x, y, z)
        float   x, y, z;
```

*scale(x, y, z)*

scales the current coordinate system by the values of x, y, and z.

```
void
scale(x, y, z)
        float   x, y, z;
```

*rotate(r, x, y, z)*

rotate the current coordinate system around the vector (x, y, z) by the angle r.

```
void
rotate(r, x, y, z)
        float   r, x, y, z;
```

*alignzaxis(x, y, z)*

rotate the current coordinate system so that the z axis now points along the vector (x, y, z).

```
void
alignzaxis(x, y, z)
        float   x, y, z;
```

**Setting the Lighting and Background Colour.**

The red, green, and blue intenstities passed to these routines are presumed to be between zero and one.

*background(r, g, b)*

sets the red (r), green (g), and blue (b) values of the background.

```
background(r, g, b)
        float   r, g, b;
```

*ambient(r, g, b)*

sets the red (r), green (g), and blue (b) values of the ambient lighting to be applied to objects defined after this call.

```
ambient(r, g, b)
        float   r, g, b;
```

*light(x, y, z, r, g, b)*

set up a point light source at the world location (x, y, z) and with the red, green, and blue intensities given in r, g, and b. Lights defined this way illuminate all objects in the scene that are not in shadow, irrespective of whether they are defined before or after the light-source.

```
light(x, y, z, r, g, b)
        float    x, y, z;
        float    r, g, b;
```

**Setting the Material Properties.**

*material(r, g, b, kd, ks, ksexp, ri, T, R)*

defines the colour (r, g, b), diffuse component (kd), the specular component (ks), the specular exponent (ksexp), the refractive index (ri), the transparency (T) and the reflectiveness (R) for the current surface. The new surface will be applied to all objects between this material call and the next one, and, with the exceptions of ksexp and ri, the values passed to it should be within zero and one.

```
material(r, g, b, kd, ks, ksexp, ri, T, R)
        float    r, g, b;
        float    kd, ks, ksexp;
        float    ri, T, R;
```

**Performing Logging and Getting Timing Information**

The rt library supports a number fo routines for doing logging. Note for any these routines to produce results the variable *logfp* must be non-NULL.

*rendsettime()*

sets the logging clock to start from the time it is called.

```
rendsettime()
```

*rendprinttime(s)*

prints out the amount of time that has passed since the last call to rendsettime, together with the message contained in the string s.

```
rendprinttime(s)
        char    *s;
```

**BUGS**

Polygonpatches are only phong shaded if they have 3 sides.

**SEE ALSO**

*vort*(3).

# Appendix C

# Source for a sample renderer using librt

**(Makefile)**

```
CFLAGS = -fsingle -fast -I/opt/local/include -L/opt/local/lib
RANLIB = echo           # = "ar ts" for SYS5

OBJS = gram.o lex.o main.o error.o

rend: $(OBJS)
	cc $(CFLAGS) -o rend $(OBJS) -lrt -lvort -ll -lm

clean:
	rm -f *.o core

clobber:
	rm -f rend *.o core

gram.o: gram.c
	cc -c -fast -I/opt/local/include gram.c

gram.c gram.h: gram.y

.y.c:
	yacc -d $*.y
	mv y.tab.c gram.c
	if cmp -s y.tab.h $*.h; then rm y.tab.h; else mv y.tab.h $*.h; fi

$(OBJS): rend.h
```

```
/*
 * typedef for the vector and vector/normal lists used
 * by the parser.
 */
typedef struct vn {
        vector          v;
        struct vn       *nxt;
} vnode;

typedef struct vnn {
        vector          v, n;
        struct vnn      *nxt;
} vnnode;
```

**(gram.y)**

```
/*
 * parser for NFF grammer
 */

%{
#include <stdio.h>
#include <rt.h>
#include "rend.h"

static vector	up, eye, ref;

%}

%union {
	int	y_int;
	float	y_flt;
	vnode	*y_vnd;
	vnnode	*y_vnn;
}

%token VIEWING FROM AT UP ANGLE HITHER RESOLUTION BACKGROUND
%token LIGHT MATERIAL POLYGON CONE SPHERE POLYGONPATCH

%token <y_flt>	FLOAT

%type	<y_flt>	expr

%type	<y_vnd> vertlist
%type	<y_vnn> vertnormlist

%%

input	: /* NULL */
	| inputitem input
	;

inputitem: VIEWING viewinfo
	  {
		viewup(up[X], up[Y], up[Z]);
		viewlookat(eye[X], eye[Y], eye[Z], ref[X], ref[Y], ref[Z]);
	  }
	| BACKGROUND expr expr expr
	  {
		background($2, $3, $4);
	  }
	| LIGHT expr expr expr
	  {
		light($2, $3, $4, 0.5, 0.5, 0.5);
	  }
	| MATERIAL expr expr expr expr expr expr expr expr
	  {
			/* following tradition, calculate reflectance... */
```

```
		material($2, $3, $4, $5, $6, $7, $9, $8, ((1.0 - $8) < $6) ? 1.0 - $8 : $6);
	}
	| CONE expr expr expr expr expr expr expr expr
	{
		cone($2, $3, $4, $5, $6, $7, $8, $9);
	}
	| SPHERE expr expr expr expr
	{
		sphere($2, $3, $4, $5);
	}
	| POLYGON expr vertlist
	{
		int	i;
		float	*xs, *ys, *zs;
		vnode	*v, *vnxt;

		xs = (float *)smalloc((int)$2 * sizeof(float));
		ys = (float *)smalloc((int)$2 * sizeof(float));
		zs = (float *)smalloc((int)$2 * sizeof(float));

		for (i = 0, v = $3; i != (int)$2; i++, v = vnxt) {
			xs[i] = v->v[X];
			ys[i] = v->v[Y];
			zs[i] = v->v[Z];
			vnxt = v->nxt;
			free(v);
		}

		polygon((int)$2, xs, ys, zs);

		free(xs);
		free(ys);
		free(zs);
	}
	| POLYGONPATCH expr vertnormlist
	{
		int	i;
		float	*xs, *ys, *zs;
		float	*nxs, *nys, *nzs;
		vnnode	*v, *vnxt;

		xs = (float *)smalloc((int)$2 * sizeof(float));
		ys = (float *)smalloc((int)$2 * sizeof(float));
		zs = (float *)smalloc((int)$2 * sizeof(float));
		nxs = (float *)smalloc((int)$2 * sizeof(float));
		nys = (float *)smalloc((int)$2 * sizeof(float));
		nzs = (float *)smalloc((int)$2 * sizeof(float));

		for (i = 0, v = $3; i != (int)$2; i++, v = vnxt) {
			xs[i] = v->v[X];
			ys[i] = v->v[Y];
			zs[i] = v->v[Z];
			nxs[i] = v->n[X];
```

```
                        nys[i] = v->n[Y];
                        nzs[i] = v->n[Z];
                        vnxt = v->nxt;
                        free(v);
                }

                polygonpatch((int)$2, xs, ys, zs, nxs, nys, nzs);

                free(xs);
                free(ys);
                free(zs);
                free(nxs);
                free(nys);
                free(nzs);
        }
        ;

viewinfo: /* NULL */
        | viewitem viewinfo
        ;

viewitem: FROM expr expr expr
        {
                eye[X] = $2;
                eye[Y] = $3;
                eye[Z] = $4;
        }
        | AT expr expr expr
        {
                ref[X] = $2;
                ref[Y] = $3;
                ref[Z] = $4;
        }
        | UP expr expr expr
        {
                up[X] = $2;
                up[Y] = $3;
                up[Z] = $4;
        }
        | ANGLE expr
        {
                viewfov($2);
        }
        | HITHER expr
        {
                /* IGNORED - at the momemt!
                printf("hi %f\n", $2);
                */
        }
        | RESOLUTION expr expr
        {
                imagesize((int)$2, (int)$3);
        }
```

```
	;

vertlist: /* NULL */
		{
			$$ = (vnode *)NULL;
		}
	| vertlist expr expr expr
		{
			$$ = (vnode *)smalloc(sizeof(vnode));

			$$->v[X] = $2;
			$$->v[Y] = $3;
			$$->v[Z] = $4;

			$$->nxt = $1;
		}
	;

vertnormlist: /* NULL */
		{
			$$ = (vnnode *)NULL;
		}
	| vertnormlist expr expr expr expr expr expr
		{
			$$ = (vnnode *)smalloc(sizeof(vnnode));

			$$->v[X] = $2;
			$$->v[Y] = $3;
			$$->v[Z] = $4;

			$$->n[X] = $5;
			$$->n[Y] = $6;
			$$->n[Z] = $7;

			$$->nxt = $1;
		}
	;

expr	: FLOAT
		{
			$$ = $1;
		}
	;
```

**(lex.l)**

```
%{
/*
 * lexical analyser for the NFF grammer parser.
 */

#include <stdio.h>
#include <math.h>
#include <rt.h>
#include "rend.h"
#include "gram.h"

%}

%%

v               return(VIEWING);

from            return(FROM);

at              return(AT);

up              return(UP);

angle           return(ANGLE);

hither          return(HITHER);

resolution      return(RESOLUTION);

b               return(BACKGROUND);

l               return(LIGHT);

f               return(MATERIAL);

pp              return(POLYGONPATCH);

p               return(POLYGON);

s               return(SPHERE);

c               return(CONE);

"#"             {               /* a comment */
                        while (fgetc(yyin) != '\n' && !feof(yyin))
                                ;
                }

[-0-9.e+]*      {
                        yylval.y_flt = atof(yytext);
                        return(FLOAT);
                }
```

```
[ \t\n]        ;
```

**main(main.c)**

```
#include <stdio.h>
#include <math.h>
#include <vort.h>
#include <sys/times.h>

extern FILE      *yyin;

/*
 * the log file.
 */
FILE              *logfp;

/*
 * main driver for the NFF renderer.
 */
main(ac, av)                                                        main
        int       ac;
        char      *av[];
{
        image             *im;
        int               i;
        char              imname[100], logname[100], buf[100];

        if (av[1] != (char *)NULL) {
                if (strcmp(av[1], "bvtree") == 0) {
                        strcpy(imname, av[2]);
                        strcpy(strrchr(imname, '.'), ".bv.pix");

                        strcpy(logname, av[2]);
                        strcpy(strrchr(logname, '.'), ".bv.log");
                } else if (strcmp(av[1], "grid") == 0) {
                        strcpy(imname, av[2]);
                        strcpy(strrchr(imname, '.'), ".gr.pix");

                        strcpy(logname, av[2]);
                        strcpy(strrchr(logname, '.'), ".gr.log");

                } else if (strcmp(av[1], "kdtree") == 0) {
                        strcpy(imname, av[2]);
                        strcpy(strrchr(imname, '.'), ".kd.pix");

                        strcpy(logname, av[2]);
                        strcpy(strrchr(logname, '.'), ".kd.log");
                } else if (strcmp(av[1], "octree") == 0) {
                        strcpy(imname, av[2]);
                        strcpy(strrchr(imname, '.'), ".oc.pix");

                        strcpy(logname, av[2]);
                        strcpy(strrchr(logname, '.'), ".oc.log");
                } else {
                        fprintf(stderr, "unknown tree type.\n");
                        exit(1);
                }
```

```
} else {
        fprintf(stderr, "don't know the structure type.\n");
        exit(1);
}

if ((logfp = fopen(logname, "w")) == (FILE *)NULL) {
        fprintf(stderr, "can't open log file %s\n", logname);
        exit(1);
}

if ((yyin = fopen(av[2], "r")) == (FILE *)NULL) {
        fprintf(stderr, "can't open input file %s\n", av[2]);
        exit(1);
}

rendsettime();

rendinit();

if (strcmp(av[1], "bvtree") == 0) {
        openbvtree();
                yyparse();
        closebvtree();
} else if (strcmp(av[1], "grid") == 0) {
        if (av[3] == (char *)NULL) {
                opengrid();
                        yyparse();
                closegrid();
        } else if (ac != 6) {
                fprintf(stderr, "grid requires x, y, and z resolution.\n");
                exit(1);
        } else {
                gridsize(atoi(av[3]), atoi(av[4]), atoi(av[5]));
                opengrid();
                        yyparse();
                closegrid();
        }
} else if (strcmp(av[1], "kdtree") == 0) {
        openkdtree();
                yyparse();
        closekdtree();
} else if (strcmp(av[1], "octree") == 0) {
        openoctree();
                yyparse();
        closeoctree();
} else {
        fprintf(stderr, "unknown tree type.\n");
        exit(1);
}

rendprinttime("Preprocessing time taken: ");

render(imname);
```

**main(main.c)**

```
}
```

```
#include <stdio.h>

/*
 * yyerror
 *
 *      print the error message from yacc
 */
yyerror(s)
        char    *s;
{
        fprintf(stderr, "%s\n", s);
        exit(1);
}
```

yyerror

```
/*
 * fatal
 *
 *      print out a fatal error and exit.
 */
fatal(s)
        char    *s;
{
        fprintf(stderr, "%s\n", s);
        exit(1);
}
```

fatal

# Bibliography


[1] Akimoto, Taka-akito, Kenji Mase, and Yasuhitto Suenaga. "Pixel Selected Ray Tracing.", IEEE Computer Graphics and Applications, July 1991, p 14-22.

[2] Amanatides, John, and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing.", Proceedings of Eurographics '87, p. 3-10.

[3] Amanatides, John. "Ray Tracing With Cones.", SIGGRAPH '84, ACM SigGraph, July 1984, p. 129-135.

[4] Arnaldi, Bruno, Thierry Priol, and Kadi Bouatouch. "A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes", The Visual Computer, Vol. 3, 1987, p 98-108.

[5] Arvo, James and David Kirk. "Fast Ray Tracing by Ray Classification.", SIGGRAPH '87, ACM SigGraph, July 1987, p. 269-278.

[6] Arvo, James. "Linear-Time Voxel Walking for Octrees.", Ray Tracing News, electronic newsletter edited by Eric Haines, Vol. 1, no. 2, 1988.

[7] Atkin, Phil, Steve Ghee, and Jamie Parker. "Transputer Architectures for Ray Tracing.", Proceedings of Computer Graphics '87, Online Publications, 1987, p. 157-172

[8] Barton, E. "Data Concurrency on the Meiko Computing Surface.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 353-377.

[9] Badouel, D., T. Priol. "An efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures.", Fifth Eurographics Workshop on Graphics Hardware, 1990.

[10] Badouel, D., K. Bouatouch, and T, Priol. "Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data.", SIGGRAPH '90 Parallel Algorithms and Architecture for 3D Image Generation Course Notes.

[11] Badouel, D., K. Bouatouch, and T, Priol. "Distributed Data and Control for Ray Tracing in Parallel.", IEEE Computer Graphics and Applications, July 1994, p. 69-77.

[12] Barrett, Martin. "A Load Balancing Experiment for Parallel Ray Tracing.", Proceedings of Ausgraph '90, p. 145-155.

[13] Bentley, Jon Louis. "Multidimensional Binary Search Trees Used for Associative Searching.", Communications of the ACM, September 1975, p. 509-517.

[14] Bouatouch, K., and T. Priol. "Experimenting with a Parallel Ray-Tracing Algorithm on a Hypercube Machine", in Proceedings of Eurographics '88, David A. Duce and Pierre Jancene (eds.), p. 243-259.

[15] Carter, Michael, and Keith Tegue. "Distributed Object Database Ray Tracing on the Intel iPSC/2 Hypercube.", Proceedings of the 5th Distributed Memory Computing Conference, IEEE Society Press, 1990, p. 217-222.

[16] Caspary, Elisha, and I. D. Scherson. "A Self-Balanced Parallel Ray-Tracing Algorithm", Parallel Processing for Computer Vision and Display, P. Dew, R. Earnshaw, and T. Heywood, Addison-Wesley Publishing Co., Reading Massachusetts, 1989, p. 408-419.

[17] Caubet, R., Y. Duthen, and V. Gaildrat. "VOXAR: A Tridimensional Architecture for Fast Realistic Image Synthesis.", New Trends in Computer Graphics, Proceeding of CG International '88, p. 135-149

[18] Chalmers, M. "On the Design and Implementation of a Multiprocessor Ray Tracer.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 420-430.

[19] Chen, E. C., H. E. Rushmeier, G. Miller, and D. Turner. "A Progressive Multi-Pass Method for Global Illumination.", SIGGRAPH 1991, ACM Press, 1991, p. 165-174.

[20] Cleary, John G., and Geoff Wyvill. "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision.", The Visual Computer, Vol. 4, 1988, p. 65-83.

[21] Cosshall, Wayne J. "Description of the NEXUS multi-processor for computer graphics.", Proceedings of AusGraph '89, p. 259-267.

[22] Crow, F. C, G. Demos, J. Hardy, J. McLaughlin, and K. Sims. "3D Image Synthesis on the Connection Machine.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 254-269.

[23] Defend, Daniel Evan. "Parallel Ray Tracing in a Vector-Multiprocessing Environment.", CSRD Rpt. No. 677, Center for Supercomputing Research and Development, University of Illinois, September, 1987.

[24] Delany, Hubert C. "Ray Tracing on a Connection Machine.", Proceedings of the 1988 International Conference on Supercomputing, July, 1988, St. Malo, France, p. 659-667.

[25] Devillers, Olivier. "The Macro-regions: an Efficient Space Subdivision Structure for Ray Tracing.", Proceedings of Eurographics '89. p 27-39.

[26] Dippé, Mark and John Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis.", Proceedings of SIGGRAPH '84, p. 149-158.

[27] Dorband, John E. "Ray Tracing on the MPP.", Frontiers of Massively parallel Scientific Computation, Proceedings of the Ist Symposium, NASA Conference Publication 2478, 1986, p. 211-215, 316.

[28] Fuchs, Henry, Zvi M. Kedem, and Bruce F. Naylor. "On visible surface generation by a priori tree structures.", SIGGRAPH '80, ACM SigGraph, July, 1980, p. 124-133.

[29] Fuchs, Henry, Gregory D. Abram, and Eric D. Grant. "Near Real time shaded display of rigid objects.", SIGGRAPH '83, ACM SigGraph, July 1983, p. 65-72.

[30] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata. "ARTS: accelerated ray-tracing system.", IEEE Computer Graphics and Applications, April 1986, p. 16-26.

[31] Fussell, Donald and K. R. Subramanian. "Fast ray Tracing using k-d trees.", Technical Report TR-88-07, Dept. of Computer Sciences, University of Texas at Austin, 1988.

[32] Gaudet, Severin, Richard Hobson, Pradeep Chilka, and Thomas Calvert "Multiprocessor Experiments for High-Speed Ray Tracing.", ACM TOG, Vol. 7, No. 3, July 1988, p. 151-179.

[33] Gigante, Michael. "Accelerated Ray Tracing using Non-Uniform Grids.", Proceedings of Ausgraph '90, p. 157-163.

[34] Glassner, Andrew S. "Space subdivision for fast ray tracing.", IEEE Computer Graphics and Applications, October 1984, p. 15-22.

[35] Glassner, Andrew S. "Spacetime Ray Tracing for Animation.", IEEE Computer Graphics and Applications, March 1988, p. 60-70.

[36] Glassner, Andrew S., editor. "Graphics Gems.", Academic Press. 1990.

[37] Glassner, Andrew S., editor. "An Introduction To Ray Tracing.", Academic Press. 1989.

[38] Goldsmith, J. and J. Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing.", IEEE Computer Graphics and Applications, May 1987, p. 14-20.

[39] Goral, C. M., K. E. Torrance, D. P. Greenberg, and B. Battaile. "Modelling the Interaction of Light Between Diffuse Surfaces.", SIGGRAPH '84, ACM Press, 1984, p. 213-222.

[40] Green, Stuart A., Derek J. Paddon, and E. Lewis. "A Parallel Algorithm and Tree-Based Computer Architecture for Ray-Traced Computer Graphics.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 353-377.

[41] Green, Stuart A., and Derek J. Paddon. "Exploiting Coherence for Multiprocessor Ray Tracing.", IEEE Computer Graphics and Applications, November 1989, p. 12-26.

[42] Haines, Eric A., and Donald P. Greenberg. "The Light Buffer: A Shadow-Testing Accelerator.", IEEE Computer Graphics and Applications, September 1986, p. 6-16.

[43] Haines, Eric A. "A proposal for standard graphics environments.", IEEE Computer Graphics and Applications, November 1987, p. 3-5.

[44] Haines, Eric A., *et al.* "Any Future for Efficiency Research?", Ray Tracing News, electronic newsletter ed. by Eric Haines, Vol. 5, #2, 1992.

[45] Hall, Roy, and Donald P. Greenberg. "A Testbed for Realistic Image Sythesis.", IEEE Computer Graphics and Applications, November 1983, p. 10-20.

[46] Heckbert, P. S. and P. Hanrahan. "Beam tracing polygonal objects", SIGGRAPH '84, ACM SigGraph, p. 119-127.

[47] Hébert, M. P., M. D. McNeill, B. Shah, R. L. Grimsdale, and P. F. Lister "MARTI - A Multiprocessor Architecture for Ray Tracing Images.", Proceedings of the Eurographics Hardware Workshop, September 1990.

[48] Hook, David and Kevin Forward "Using Kd-trees to Guide Bounding volume Hierarchies for Ray Tracing.", To be published the Australian Computer Journal, Australian Computer Society.

[49] Horvath, Tamas, Gabor Narton, Peter Risztics, and Lasxlo Szirmay-Kalos. "Ray Coherence Between a Sphere and a Convex Polyhedron.", Computer Graphics Forum, The Eurographics Association, June 1992, p. 163-172.

[50] Jansen, Frederick W.. "Data Structures for Ray Tracing.", Data Structures For Raster Graphics, Springer Verlag, 1986, p. 57-73.

[51] Jevans, David, and Brian Wyvill. "Adaptive Voxel Subdivision for Ray Tracing.", Graphics Interface '89. p. 164-172.

[52] Kajiya, James T., Brian P. Von Hersen "Ray Tracing Volume Densities", SIGGRAPH '84, ACM SigGraph, 1984, p. 165-174.

[53] Kay, Timothy L., and James T. Kajiya. "Ray Tracing Complex Scenes.", SIGGRAPH '86, ACM SigGraph, 1986, p. 269-278.

[54] Kok, A. J. F., and F. W. Jansen, "Adaptive Sampling of Area Light Sources in Ray Tracing Including Diffuse Reflection.", Eurographics '92, Eurographics Association, 1992, p. 289-298.

[55] Kuchkuda, Roman "An Introduction To Ray Tracing", Theoretical Foundations of Computer Graphics and CAD, NATO ASI, Vol. 40, Series F, Springer-Verlag, 1988, p. 1039-1060.

[56] Kunii, T. L., S Nishimura, and T. Noma. "The Design of a Parallel Processing System for Computer Graphics.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 353-377.

[57] Kaplan, Michael R.. "The Use of Spatial Coherence in Ray Tracing.", Techniques for Computer Graphics Graphics, Ed. Rogers and Earnshaw, Springer-Verlag, 1987, p. 173-193.

[58] Kirk, David, and James Arvo "The Ray Tracing Kernel", Proceedings of AusGraph '88, Melbourne, Australia, Juy 1988, p. 75-82.

[59] Kobayashi, Hiroaki, Satoshi Nishimura, Hideyuki Kubota, Tadao Nakamura, and Yoshiharu Shigei. "Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision.", The Visual Computer, Vol. 4, 1988, p. 97-209.

[60] Kobayashi, Hiroaki, Tadao Nakamura, and Yoshiharu Shigei. "A Strategy for Mapping Parallel Ray-Tracing into a Hypercube Multiprocessor System.", New Trends in Computer Graphics, Proceedings of CG International '88, Nadia Magnenat-Thalmann and Daniel Thalmann (eds.), Springer-Verlag, p. 160-169.

[61] Lin, Jimmy Jih-Ming. "Fast Ray Tracing on BSP-Trees", Masters Thesis, Computer Science Department, University of Florida.

[62] Lin, Tony, and Mel Slater. "Stochastic Ray Tracing Using SIMD Processor Arrays.", The Visual Computer, Vol 7, 1991, p 187-199.

[63] MacDonald, J. David, and Kellog S. Booth. "Heuristics for Ray Tracing Using Space Subdivision.", The Visual Computer, Vol 6, 1990, p. 153-166.

[64] Marsh, S. C. "Fine Grain Parallel Architectures and the Creation of High-Quality Images.", Theoretical Foundations of Computer Graphics and CAD, NATO ASI Series, Vol. F40, Springer-Verlag, 1988, p. 727-754.

[65] Maurel, Hervé, Yves Duthen, and René Caubet. "A 4D Ray Tracing.", Eurographics '93, The Eurographics Association, September 1993, p. 285-294.

[66] McNeill, M. D. J., B. C. Shah, M. P. Hébert, P. F. Lister, and R. L. Grimsdale. "Performance of Space Subdivision Techniques in Ray Tracing.", Computer Graphics Forum, The Eurographics Association, October 1992, p. 213-220.

[67] Menon, Jai, Richard J. Mavisu, and Jovan Zagojac. "More Powerful Solid Modelling Through Ray Representation.", IEEE Computer Graphics and Applications, May 1994, p. 22-35.

[68] Morris, D. T., and P. M. Dew "Dynamic Dataflow Algorithms for Ray Tracing CSG Objects.", Parallel Processing for Computer Vision and Display, edited by P. M. Dew, R. A. Earnshaw, and T. R. Heywood, Addison-Wesley Publishing Company, 1989, p. 353-377.

[69] Nishimura, Hitoshi, Hiroshi Ohno, Toru Kawata, Isao Shirakawa, and Koichi Omura. "LINKS-1: A Parallel Pipelined Multimicrocompute System for Image Creation.", Proceedings of the 10th Annual International Symposium on Computer Architecture, 1983, p. 387-394.

[70] Ohta, Masataka, and Mamoru Maekawa. "Ray Coherence Theorem and Constant Time Ray Tracing Algorithm.", Computer Graphics 1987 - Proceedings of CG International '87, Tosiyasu L. Kunii (ed.), Springer-Verlag, p. 303-314.

[71] Peng, Qunsheng, Yining Zhu, and Youdong Liang. "A Fast Ray Tracing Algorithm Using Space Indexing Techniques.", Proceedings of Eurographics '87, Guy Marechal (ed.), p. 11-23.

[72] Pitot, Paul "The Voxar Project.", IEEE Computer Graphics and Applications, January 1993, p. 27-33.

[73] Plunkett, David J., and Michael J. Bailey. "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed.", IEEE Computer Graphics and Applications, August 1985, p. 52-60.

[74] Priol, Thierry, and Kadi Bouatouch. "Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube.", The Visual Computer, Vol. 5, 1989, p. 109-119.

[75] Quek, Lee Hian. "Ray Tracing with Space Subdivision on a Vector-Multiprocessor System", Masters Thesis, Report No. UIUCDCS-R-89-1491, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1989.

[76] Quinn, Michael J. “Designing Efficient Algorithms for Parallel Computers.”, McGraw-Hill Book Company, New York, 1987.

[77] Ritter, J. “A Simple Ray Rejection Test.”, Graphics Gems, ed. by Andrew Glassner, Academic Press, 1990, p. 385-386.

[78] Roth, S. D. “Ray Casting for Modeling Solids.”, Computer Graphics and Image Processing, 18(2), Feb, 1982, p. 109-144.

[79] Rubin, Steven M., and Turner Whitted. “A 3-Dimensional Representation for Fast Rendering of Complex Scenes.”, SIGGRAPH ’80, ACM Press, 1980, p. 110-116.

[80] Salesin, D. and J. Stolfi. “Rendering CSG Models with a ZZ-Buffer.”, SIGGRAPH ’90, ACM Press, 1990, p. 67-76.

[81] Salmon, John, and Jeff Goldsmith. “A Hypercube Ray-tracer.”, Third Conference on Hypercube Concurrent Computers and Applications., Geoffrey Fox (ed.), Vol. 2, 1988, p. 1194-1206.

[82] Samet, Hanen. “Implementing Ray Tracing with Octrees and Neighbour Finding.”, Computers and Graphics, Vol. 13, no. 4, 1989, p. 445-460.

[83] Samet, Hanen. “Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS”, Addison-Wesley, 1990.

[84] Scherson, I. D, and E. Caspary. “Data Structures and the Complexity of Ray Tracing.”, The Visual Computer, Vol. 3, 1987, p. 201-213.

[85] Sederberg, T. W., David C. Anderson “Ray Tracing of Steiner Patches”, SIGGRAPH ’84, ACM SigGraph, 1984, p. 159-164.

[86] Semwal, S. K. “The Slicing Extent Technique for Fast Ray Tracing.”, Ph. D. Dissertation, Computer Science Department, University of Central Florida, 1987.

[87] Shinya, Mikio, Tokiichiro Takahashi, and Seiichiro Naito. “Principles and Applications of Pencil Tracing”, SIGGRAPH ’87, ACM SigGraph, 1987, p. 45-54.

[88] Simiakakis, G. and A. M. Day. “Five-dimensional Adaptive Subdivision for Ray Tracing”, Computer Graphics Forum, Vol 13, number 2, 1994, p. 133-140.

[89] Snyder, John M., and Alan H. Barr. “Ray Tracing Complex Models Containing Surface Tessellations.”, SIGGRAPH ’87, p. 119-128.

[90] Spackman, John. “Scene Decompositions for Accelerated Ray Tracing”, Bath Computer Science Technical Report 90/33, The University of Bath, UK, 1990.

[91] Speer, Richard, Tony DeRose, and Brian Barsky. “A Theoretical and Empirical Analysis of Coherent Ray-Tracing.”, Computer-Generated Images: The State of the Art, Proceedings of Graphics Interface ’85, ed. by N. Magenat-Thalmann and D. Thalmann, p. 11-25.

[92] Subramanian, K. R. and Donald S. Fussell. “Automatic Termination Criteria for Ray Tracing Hierarchies.”, Proceedings of Graphics Interface ’91, Calgary, Alberta, 1991.

[93] Sung, Kelvin. “A DDA Octree Traversal Algorithm for Ray Tracing.”, Eurographics ’91, p. 73-85.

[94] Sung, Kelvin. “Area Sampling Buffer: Tracing Rays with Z buffer Hardware.”, Eurographics ’92, The Eurographics Association, September 1992, p. 299-310.

[95] Thibault, W. C., B. F. Naylor. “Set operations on Polyhedra using Binary Space partioning trees”, SIGGRAPH ’87, ACM SigGraph, 1987, p. 153-162.

[96] Wallace R., M. F. Cohen, and D. P. Greenberg, “A Two-pass solution to the Rendering Equation - a synthesis of Ray Tracing and Radiosity Methods”, SIGGRAPH ’87, ACM SigGraph, 1987, p. 311-320.

[97] Ward, Gregory. “Adaptive Shadow Testing for Ray Tracing.”, Second Eurographics Workshop on Rendering, May 1991.

[98] Weghorst, Hank, Gary Hooper, and Donald P. Greenberg. “Improved Computational Methods for Ray Tracing.”, ACM Transactions on Graphics, Vol 3, No. 1, January 1984, p. 52-69.

[99] Williams, N. S., B. F. Buxton, and H. Buxton. "Distributed Ray Tracing using a SIMD Processor Array.", Theoretical Foundations of Computer Graphics and CAD, NATO ASI Series, Vol. F40, Springer-Verlag, 1988, p. 703-725.

[100] Wilson, Tom and Narsingh Deo. "Comparing Extent Tracing to Other Ray Tracing Accelerators.", Technical Report CS-TR-92-21, Department of Computer Science, University of Central Florida, Orlando, September 1992.

[101] Wilson, Tom and Narsingh Deo. "Acceleration Schemes for Ray Tracing.", Technical Report CS-TR-92-22, Department of Computer Science, University of Central Florida, Orlando, September 1992.

[102] Wilson, Tom and Narsingh Deo. "Extent Tracing: Algorithm and Implementation.", Technical Report CS-TR-92-34, Department of Computer Science, University of Florida, Orlando, December 1992.

[103] Wilson, Tom. "Shadow Testing Simplification.", Ray Tracing News, electronic newsletter ed. by Eric Haines, Vol. 4, #1, 1991.

[104] Whitted, Turner. "An Improved Illumination Model for Shaded Display.", Communications of the ACM, Vol 23, No. 6, June 1980, p. 343-349.

[105] Whitted, Turner, and David M. Wimer. "A Software test-bed for the Development of 3-D Raster Graphics Systems", SIGGRAPH '81, ACM SigGraph, 1981, p. 271-278.

[106] Woo, Andrew, Pierre Poulin, and Alain Fournier. "A Survey of Shadow Algorithms.", IEEE Computer Graphics and Applications, November 1990, p. 13-32.

[107] Woo, Andrew "Efficient Shadow Computations in Ray Tracing.", IEEE Computer Graphics and Applications, September 1993, p. 78-93.

[108] Yagel, Roni, Daniel Cohen, and Arie Kaufman. "Discrete Ray Tracing." IEEE Computer Graphics and Applications, September 1992, p. 19-28.