USING PROJECTION TO ACCELERATE RAY TRACING

by

Anastasia Bezerianos

A thesis submitted in conformity with the requirements
for the degree of M.Sc.
Graduate Department of Computer Science
University of Toronto

Abstract

Using Projection to accelerate Ray Tracing

Anastasia Bezerianos

M.Sc.

Graduate Department of Computer Science

University of Toronto

2001

The high cost of Ray Tracing image rendering has driven many researchers to devise acceleration techniques like bounding volume hierarchies.

This thesis introduces new ways of building bounding volume hierarchies. We modify existing algorithms to use bounding volumes projected onto the viewing and light planes. This allows faster traversal of less tight hierarchies. First, hierarchical structures are constructed using only the information derived from the projection of the bounding volumes. Second, a view independent hierarchical structure is projected onto the viewing and light planes. Finally, we augment each of the previous hierarchies with testing rays against both the projected and the view independent volume. Thus fast traversal of the hierarchy and tight modeling are combined.

Testing demonstrates that projection hierarchies accelerate Ray Tracing when the number of objects in a scene is large and the number of possible hierarchies to choose from is limited.

Contents

List of Figures

Chapter 1

Introduction

One of the most versatile image rendering techniques introduced in literature is Ray Tracing. It is simple, elegant and easily implemented. Furthermore, it can model aspects of realistic photographs such as reflections, refractions, and shadows in an intuitive way.

1.1 Ray Tracing, a fast overview

Ray Tracing is a global illumination rendering method. The technique addresses several computer graphics problems: visibility, clipping, light propagation, and shadows. In its simplest form, ray casting, the method solves the visibility problem by identifying the closest object at each image pixel.

In ray casting, rays of light are traced from the eye, through the image plane, onto the scene. These rays are tested against the objects in the scene in order to determine possible intersection points with any objects. If no object is hit by the ray, the pixel from where the ray originated is attributed the background color. This technique is also referred to as Backward Ray Tracing, since light rays are not followed forward, from the light source to the eye, but backwards, from the eye to the light source. The color returned by the ray is set as the color of the pixel of origin.

Several issues involving realistic rendering, such as shadows, reflection, refraction, are

dealt with very simply and effectively by Ray Tracing.

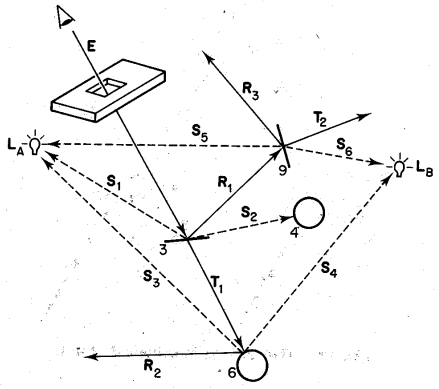


Figure 1.1: The course of a light ray from the eye E ([Gla89])

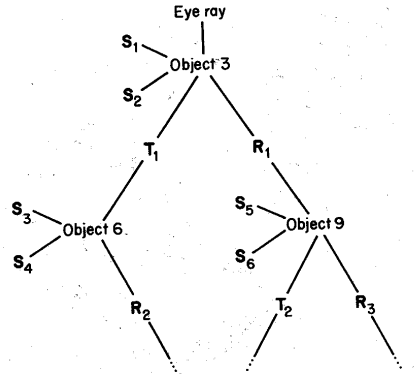


Figure 1.2: The ray tree for Fig 1.1 ([Gla89])

1.1.1 Shadow Rays

When the color of a point on an object needs to be computed it must be determined if a light source affects this surface. Thus, from any visited surface, shadow rays are sent to all the light sources in the scene. If a shadow ray hits an opaque object before reaching the light source then the surface is in shadow with respect to this light source, i.e. this light does not contribute to the color of the surface; on the other hand, if the shadow ray reaches the light source then this source contributes to the surface color. This type of ray is also called illumination ray.

The above simple check is only effective when trying to render scenes with point lights, that is light sources that illuminate in a single direction. For area lights many shadow rays must be cast from the lights to the object, in order to accurately sample the area light.

1.1.2 Reflection Rays

Shiny surfaces act like mirrors and reflect other objects. In order to find the light that is reflected on a surface in Ray Tracing, a reflection ray is shot from the reflective surface in the direction dictated by the direction of the incident ray (eg. the ray from the camera). For perfect reflection

surfaces this direction is unique, computed as seen in Figure 1.3. To determine the color of the reflection ray, the ray is traced backwards until the object from which the light originated is found. The color of this object is the color of the reflection ray and it contributes to the color of the shiny surface.

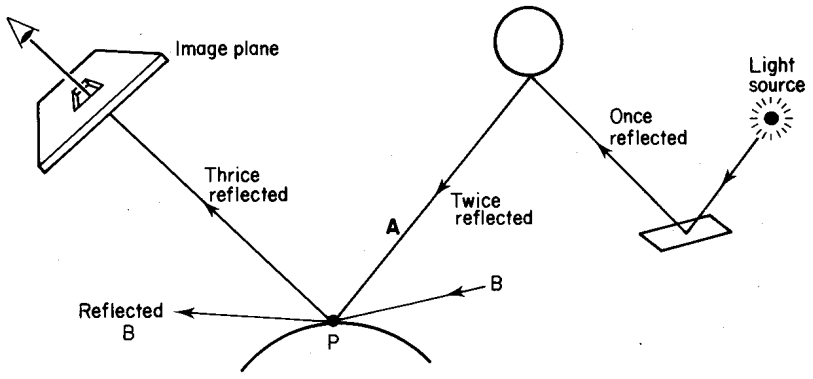


Figure 1.3: Reflection Rays, from [Gla89]

1.1.3 Refraction Rays

As in reflection, refraction angles are unique for a specific ray hitting an object (Figure 1.5) thus refraction ray casting, as in reflection, provides a way to render transparent surfaces that transmit light (medium) (Figure 1.4). By following the transmitted ray backwards the color of the ray can be determined.

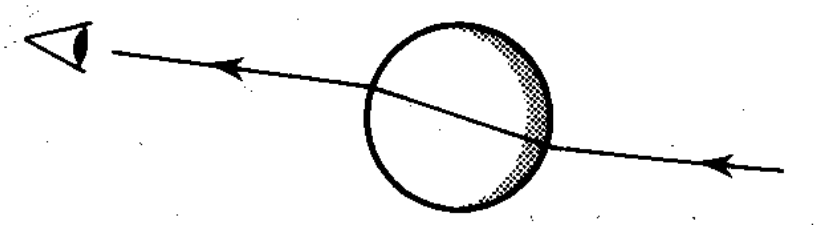


Figure 1.4: Transmitted Rays, from [Gla89]

1.1.4 Limitations

1.1.4.1 Aliasing

The Ray Tracing rendering algorithm suffers from some inherent drawbacks. First of all, since Ray Tracing is a point sampling algorithm aliasing problems can occur, which is to be expected, since it represents continuous phenomena using discrete representations. Several methods have been proposed in order to address the aliasing effect. The simplest one is supersampling. According to it, for every pixel, several rays should be cast and their average color will provide the final color for the pixel. This technique does not really eliminate the aliasing problem, it reduces it. Needless to say it can be also very expensive.

A more refined method is adaptive supersampling [Whi80]. According it, for every pixel 5 rays are shot (4 from the corners and one in the middle of the pixel). If these rays have approximately the same color, then the color of the pixel is the average of the 5 rays' colors. If one of the rays has a different color, then the pixel is viewed as 4 segments, each one sampled as a pixel (5 rays). This method is more efficient than simple supersampling; nevertheless it still doesn't address the problem fully, since 5 similar colored rays do not guarantee that no small object exists in the pixel in one of the smaller areas. Moreover, the grids visited are regular. Stochastic or distributed ray tracing [DW85] abandons the idea of a regular grid and instead samples each pixel by a number of uniformly distributed rays, thus addressing the aliasing problem that comes from regular grids (jaggies, popping edges). Nevertheless this approach introduces noise into the picture, because every pixel is an average of a number of random rays. Finally, statistical supersampling [LRU85] uses statistical methods to determine the number of rays that should be sent through each pixel, in order to achieve a desired error tolerance.

1.1.4.2 Specular Interactions

Another limitation of ray tracing is that it considers totally specular interactions, with either perfectly reflected or refracted rays (Figure 1.5).

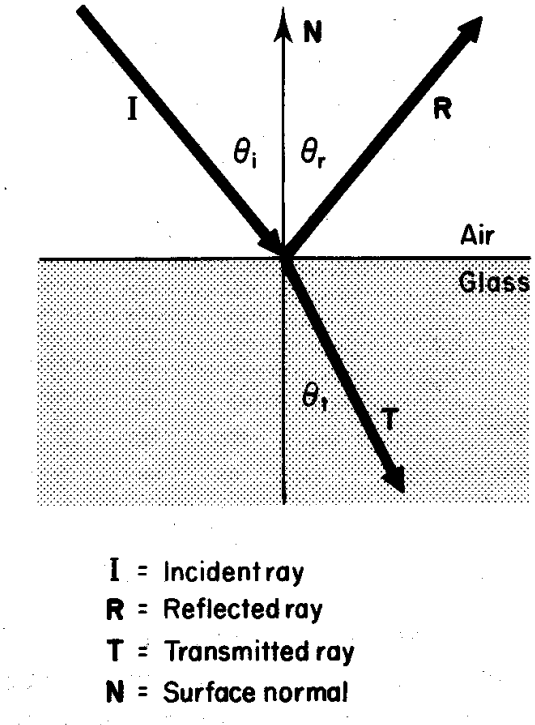


Figure 1.5: Specular reflection and refraction angles, from [Gla89]

Thus ray traced scenes don't easily show color bleeding (where a brightly colored surface's color will "bleed" onto adjacent surfaces), as seen in Figure 1.6[1] Nevertheless, diffuse reflections can be accomplished by combining ray tracing with another light simulation approach, radiosity.

[1]The "color bleeding" image was modeled by Stephen Spencer using in-house modeling and animation software and rendered with the RADIANCE global illumination package. Copyright 1992, ACCAD, The Ohio State University.

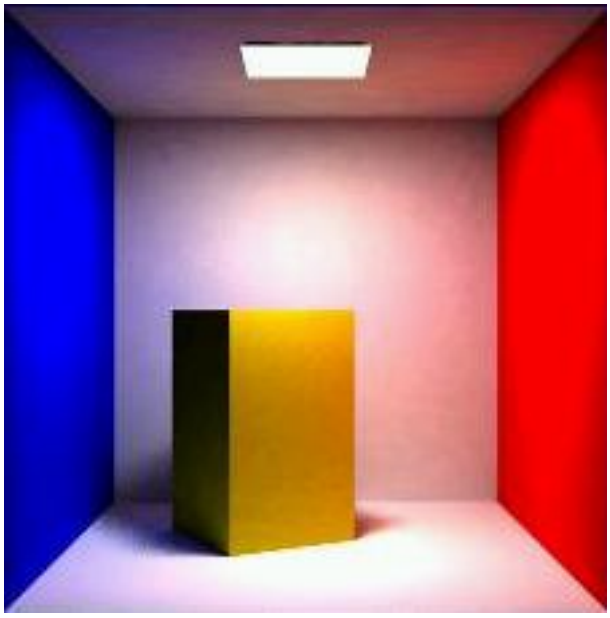


Figure 1.6: Example of color bleed. Both the red and blue walls ”bleed” their color onto the white walls, ceiling and floor.

1.1.4.3 Efficiency

Finally, the biggest and most difficult to address problem of ray tracing is its time cost.

Assume that the camera or eye is placed in world coordinates at position **E**. Then the pseudocode of a simple ray tracer is roughly as shown in Figure 1.7.

So assuming we have n objects and an $(u \times v)$ resolution image, then the time complexity of the approach is $\mathcal{O}(n \times (u \times v))$. As this formula shows, this algorithm is very inefficient. It becomes even worse if one thinks that this is only the visibility part of the algorithm, as well as the fact that the test for ray-object intersection can be extremely high (especially for complex objects).

```
FOR each scanline in image
{
    FOR each pixel in scanline
     {
          determine ray from E through pixel;
          FOR each object in scene
          {
               IF object is intersected and the closest considered
                    store intersection and object name;
          }
          set pixel color to that of closest object intersection;
     }
}
```

Figure 1.7: A simple ray tracer

1.2 Motivation

The time complexity of the Ray Tracing rendering method depends on several factors. The most computationally expensive component is the calculation of ray-object intersections, so the number of rays that are cast and fail to hit objects must be minimized.

As we mentioned before the cost of the simple ray tracer is

$$C_{RT} = n \times (u \times v) \times C_{OBJ\_INT} \tag{1.1}$$

Several solutions have been proposed over the years for improving the time requirements. All of them aim at reducing the number of objects tested for intersection, since the C_{OBJ_INT}

in Equation 1.1 is so high. Thus several structures that identify candidates for testing are used. These structures try to model as close as possible the distribution of objects in the scene and have a creation cost $C_{BUILD}(n)$, which depends strongly on the number n of objects in the scene; testing a ray against them should be cheaper than against an object. The cost for ray tracing a scene with a structure for identifying candidates is given by Equation 1.2. We denote by r_{HIT} the number of rays that hit the structure and by $r_{MISS} = (u \times v) - r_{HIT}$ those that fail to hit it, with a small intersection cost C_{MISS}.

$$C_{RT} = C_{BUILD}(n) + r_{MISS} \times C_{MISS} + r_{HIT} \times C_{TRAVERSE} \tag{1.2}$$

The cost $C_{TRAVERSE}$ of traversing the structure, as well as the number of rays missing and hitting the objects of the scene, depends on how accurately the hierarchical structure represents the spatial distribution of the real scene. Moreover, the $C_{TRAVERSE}$ cost and the cost of missing C_{MISS} the hierarchy is affected by how cheap it is to test rays against the structure. In the traversal cost is contained the cost $r_{O\_MISS} \times C_{O\_MISS}$ of rays that hit the structure and fail to hit an object, as well as the cost of intersecting the r_{SUC} successful rays that hit an object. The number of such objects, per ray, is far less than n, as the structure reduces the number of potential candidates for testing.

In our work we will attempt to further reduce the cost C_{MISS} and $C_{TRAVERSE}$, that is the calculation cost of rays hitting the structure. Furthermore, we will research and test structures, in order to find which one works best with our acceleration technique.

Although there has been a rapid improvement of hardware, ray tracing performance still hasn't speed up enough enough to cover the needs of animators and designers. Moreover people are always making complicated and expensive scenes, so even more speed is required by the ray tracing tools. This is why in this work we try to improve the current ray tracing performance.

1.3 Thesis Contributions

The unique aspects of the work presented in this thesis are:

1. 2-dimensional pre-projection of bounding volumes.

2. Hierarchy based on the 2-dimensional bounding volumes.

3. Introduction of a nearest-neighbor based bottom up hierarchy.

4. Comparative study of ray tracing bounding volume hierarchy acceleration techniques.

In brief we argue that the intersection calculations between rays and projected bounding boxes can accelerate the traversal of a hierarchy. We compare the results for several types of hierarchies and prove that the hierarchy selectivity greatly affects the performance of ray tracing. Finally, we show that the dominant cost of ray tracing is that of the intersection calculations at the leaves of a hierarchy (actual objects) and thus that the hierarchy quality matters.

1.4 Thesis Organization

Our thesis is organized as follows: After having introduced the basic concepts of ray tracing and having briefly outlined our work in this chapter, we will proceed in presenting work done so far in the field of accelerating ray tracing using techniques similar to ours in “Previous Work” (Chapter 2). We will then give a detailed description of our ideas and some implementation details in the “Overview of Our Approach” (Chapter 3). The following two chapters (Chapter 4 and 5) describe in detail the relation between bounding volumes used in literature and ray intersection cost, and attributes of the hierarchical structures used to test our approach respectively. Chapter 6, apart from describing our testing strategies and metrics, includes a detailed analysis of the results of our testing. Finally, we summarize our work and observations in the “Conclusion” of the thesis (Chapter 7).

Chapter 2

Previous Work

A lot of work has been put into accelerating the Ray Tracing engine, mainly because the results of the method are of such high quality. In [AK89] the techniques for accelerating Ray Tracing are basically divided in three categories. We will adopt this classification for our purposes of presenting previous work on the field.

As stated in [AK89], acceleration methods aim either at reducing the cost of intersecting rays, reducing the number of rays intersecting the environment, generalizing the rays as entities, or a combination of the above.

In our work we focus on faster intersections with the environment. So we will present here work done over the years dealing either with improving the time complexity of intersecting a ray with an object or with reducing the number of rays that need to be tested for intersections.

2.1 Faster Ray-Object Intersections

Most of the objects represented in rendered images and animation are quite complex. That directly translates into expensive ray-object intersection computations. To go around this problem the idea of a *bounding volume* was introduced [Whi80].

A bounding volume is a 3D object that contains the original object and is generally easier to intersect with than the original object. Given that, a ray is first tested against the bounding

volume and only if this test is successful is tested against the actual contained object.

There is a trade-off though between the simplicity of the bounding volume (cheap intersection computation) and the tightness of the bounding volume (minimizing the number of rays that hit the bounding volume but not the contained object). This trade-off is expressed in Equation 2.1 by [WHG84], where r_b is the number of rays tested on a bounding volume, B the cost of a test against the bounding volume, r_o is the number of successful hits on the bounding volume and I the cost for testing against the actual object.

$$Cost = r_b * B + r_o * I \tag{2.1}$$

Examples of "cheap" bounding volumes (Figure 2.1) are spheres, axis-aligned or non-aligned boxes and cylinders. More sophisticated (tighter), and thus more expensive, bounding volumes include intersections or unions of more than one bounding volumes or other closely fit convex volumes (Figure 2.2).

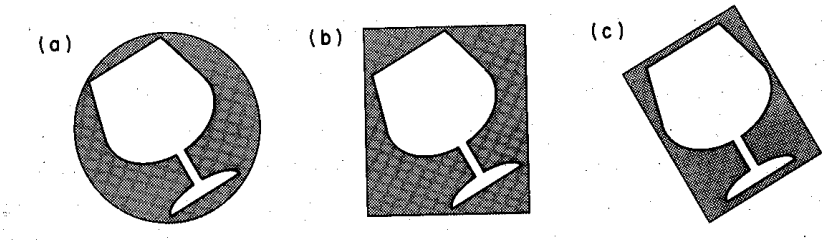


Figure 2.1: Simple Bounding Volumes, from [AK89]

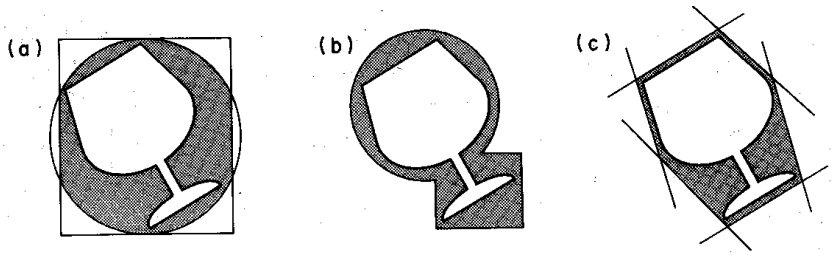


Figure 2.2: Complex Bounding Volumes, from [AK89]

The trade-off expressed in Equation 2.1 is apparent in the results of our work. We chose a scene of the ones tested at Chapter 6, the “tree” scene of size 11 and a hierarchical algorithm used, the R-tree, which we will describe in detail in Paragraph 2.2.2.1.3.

As seen in Chapter 4 the cost of intersecting the proposed 2D projected bounding box is 4 floating point operations (FLOPS), while that of an ordinary axis aligned 3D box is on average 27 FLOPS. However, as we will discuss later on, 2D projected boxes tend to be considerably less tight than 3D boxes. In Figure 2.3 we can see how the simplicity of intersecting a bounding volume B affects the number of rays r_o that succeed in hitting the bounding volume and must be tested on the simple object. These results correspond to the scene in Figure 2.4.

	Intersection Cost B	Average # Rays r_o
2D Bounding Box	4.0	513.1
3D Bounding Box	27.4	81.6

Figure 2.3: Trade between complicated/tight bounding volumes and simple/loose ones for scene in Figure 2.4.

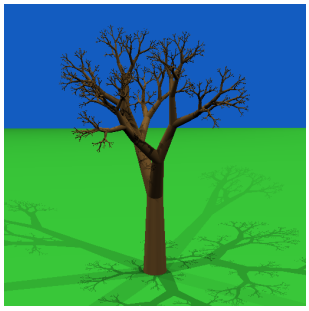
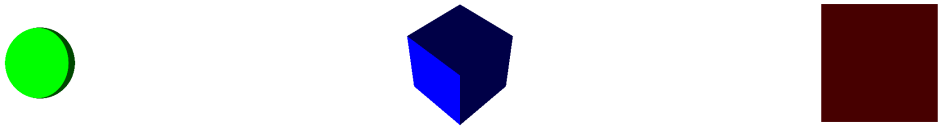


Figure 2.4: Tree scene

We must note here that the big difference between the two types of bounding box in the average number of rays that enter the bounding box r_o, in Figure 2.3, arises in part because the bounding boxes are arranged hierarchically. So the looseness of the 2D bounding boxes leads to more unsuccessful probes down the tree (for example through the intermediate levels of the

R-tree, as seen in Paragraph 2.2.2.1.3).

A more illustrative example is that of Figure 2.5. The compared tightness of the 2D and 3D bounding box is more obvious, since no hierarchical structures are used to enhance the performance of ray tracing.



	Intersection Cost B	Average # Rays r_o
2D Bounding Box	4.0	67650 out of 307200
3D Bounding Box	27.4	41596 out of 307200

Figure 2.5: The sphere is our actual object. The blue box is the 3D bounding box of the green sphere (for a particular angle of the camera). The red square is the 2D box that encloses the 3D bounding volume of the sphere. In the table we can see the number of rays that succeeded in hitting the 2D and 3D box out of all the rays in the scene. These successful rays will then be tested against the actual object enclosed in these presented bounding volumes, our sphere.

2.2 Fewer Ray-Object Intersections

The most common techniques used for reducing the number of rays that are shot in a scene are Space Subdivision techniques and Object Hierarchies. The basic distinction among them is that Space Subdivision manipulates the 3D space based on the objects in the scene, whereas Object Hierarchies group objects based on their position in the 3D space.

2.2.1 Space Subdivision

3D spatial subdivision techniques are used in order to narrow down the number of objects that are most likely to be hit by a ray. According to this set of algorithms, space is divided in a top down fashion into smaller pieces. Originally, these extends (pieces) of the scene were of equal size, generating a regular grid (Figure 2.6[1]). A common notion in all spatial subdivision technique is the *voxel*. A voxel is the smallest axis aligned 3D unit that can be created by partitioning space. Basically it is the 3D equivalent of a pixel. Each voxel is associated with a list of objects from the scene that are contained (partially or wholly) in it.

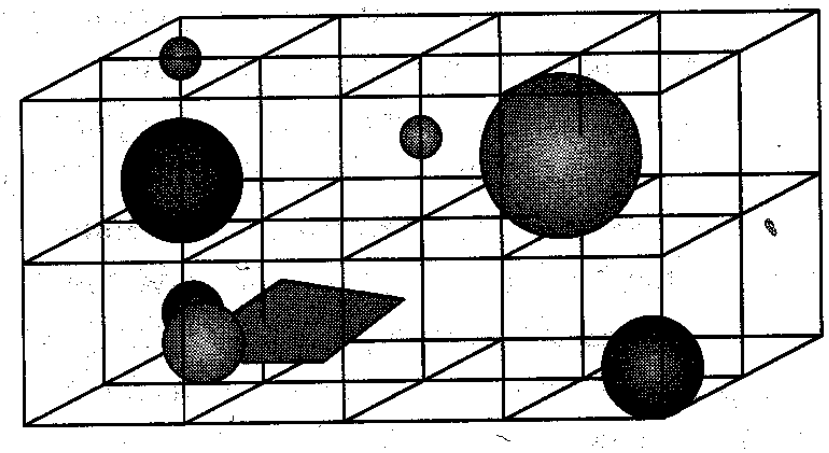


Figure 2.6: A regular grid

Traversing a 3D grid, in the order the partitions are hit by a ray, is also referred to as *walking*. A ray walking through a 3D grid needs only to be intersected with the objects contained in that grid (all of them, so that the closest intersection point can be found). Furthermore, since the grids that the ray crosses are visited in order, if a grid is determined to contain an intersection no further calculations are needed for the specific ray (Figure 2.7)[2]. Nevertheless, uniform 3D grid structures are very inefficient, both in terms of space (may have a big number of empty voxels) and time (traversing empty voxels). These shortcomings are addressed by

[1]Image from [FvDFH97]
[2]Image from [FvDFH97]

several adaptive subdivision techniques.

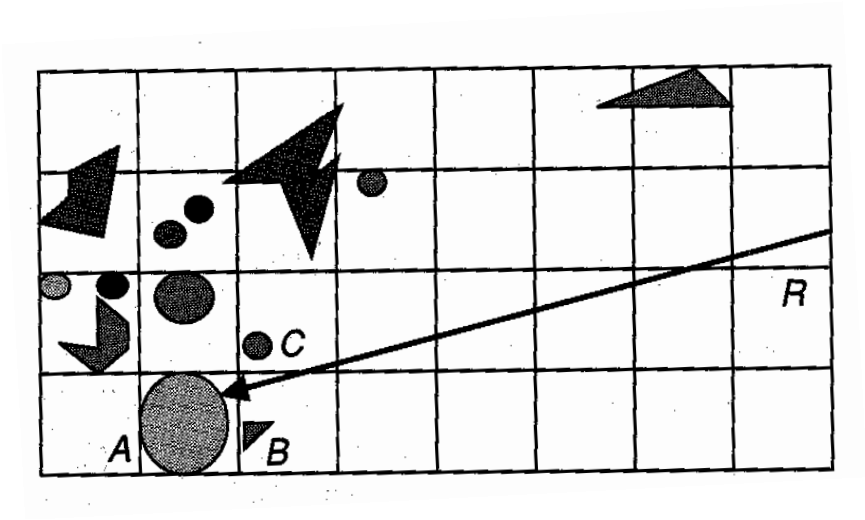


Figure 2.7: Spatial partitioning. Ray R intersects only A, B and C.

2.2.1.1 Octrees

An octree, as used in [Gla84], is a data structure containing a collection of voxels. It is generated by recursively subdividing the rectangular 3D space into decreasing volumes (voxels), until each voxel contains less objects than a predefined number. The technique is dynamic, i.e. empty voxels do not need to be subdivided, often resulting in big empty regions. If part of the surface of an object passes through a voxel it is associated with the voxel and is stored into a list.

The acceleration of ray tracing using such a structure is based on the fact that we can follow a ray through the compartments (voxels) that it traverses and test it against only the objects in that voxel. The above advantage is lost if the overhead of moving from compartment to compartment is big and this is what the data structure of the voxels tries to address.

In the octree, at each step, the non-empty voxels are subdivided into 8 equal children, using as partitioning planes the 3 coordinate axis aligned planes that go through the midpoint of each one of the axes inside the voxel volume (Figure 2.8(a)). These 8 children are labeled by the

number characterizing their father and an extra digit from 1 to 8. They are then associated with the list of objects that they contain. The voxels are stored and retrieved using a hash table on their names. By avoiding explicit child pointers, this reduces the storage cost (Figure 2.8(b) [3]).

The next voxel that a ray must visit is determined by where the ray exited the previous voxel, making a small adjustment to make sure that the point lies inside an adjacent voxel.

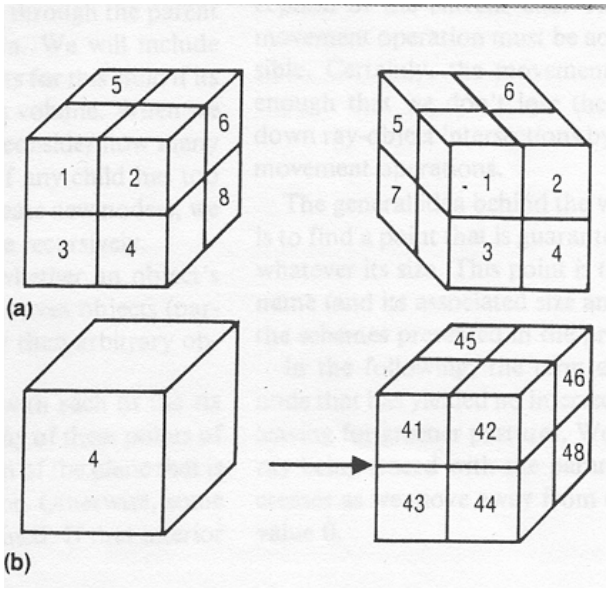


Figure 2.8: Spatial subdivision using an octree. (a) subdivision of the scene; (b) subdivision of a voxel

2.2.1.2 BSP Trees (Binary Space Partitioning) for simple objects

BSP trees have been mostly used in visibility testing of scenes containing polygons. In a BSP tree, a polygon from the scene is chosen as the tree root and all the other polygons are either added to the left child node (if they are in front of the root polygon), or to the right child (if they are at the back of the root), or to both if the root intersects the particular polygon (Figures 2.10, 2.9). The splitting is propagated until only one polygon (or a fragment of a polygon) is in each leaf.

[3] image from [Gla84]

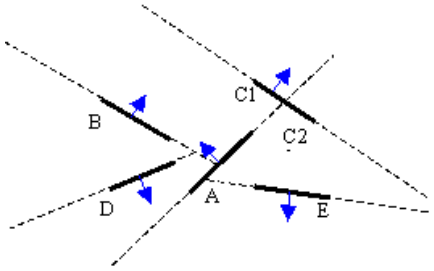


Figure 2.9: Partitioning of scene by planes

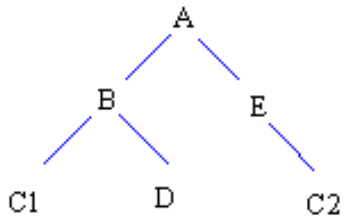


Figure 2.10: The BSP tree for Figure 2.9

2.2.1.3 *k-d* trees

A variation of BSP trees is used by [Kap85] to accommodate all objects (not only polygons) and to accelerate ray tracing. The splitting planes of the scene are now axis aligned (in the original BSP structure any orientation is possible) in order to speed up splitting. This variation, also known as *k-d* trees, is quite similar in the subdivision process to octrees. Three axis aligned planes split the scene into eight equal voxels. Each object is tested against every voxel and if it is inside the voxel the object is added to the voxel's list of objects. Voxels with a big number of objects are further subdivided using the same process. Empty voxels remain as they are, thus the structure adapts to the specific scene. Although the subdivision process is quite similar to that of the octree, the data structure is quite different.

Each time a split according to a plane is performed two new voxels are created, as children of the original voxel. The plane and the axis (orientation of the partitioning plane) are stored in the root. Voxels are assigned a list of objects that are intersected by the surface of that voxel.

The ray-object intersection procedure is a traversal of the resulting binary tree, from root to a voxel that contains objects intersecting the ray. If no objects intersect the ray in a voxel, the next voxel is calculated, as in the octree, and the new insertion point at the voxel is placed as the new beginning of the ray at the root.

The main difference between a *k-d* tree and a BSP one, as introduced in [Kap85], is that the partitioning planes are defined differently. In *k-d* trees an axis aligned plane is fit for a partitioning plane at each level of the tree, if it divides the objects into almost equal numbers

and "cuts" (intersects) as few objects as possible. In other words the partitioning planes in *k-d* trees are flexible in terms of position and choice of partitioning dimension. The subdividing procedure stops when one branch of the tree gets exactly one object completely on one side of the partitioning plane. If this node is significantly bigger than the bounding box of the included object, then the bounding box is stored at the node.

2.2.1.4 Observations concerning Space Partitioning Techniques

The *k-d* tree structure adapts to objects more than the BSP Tree and the octree and results in more balanced trees than BSP Trees, since *k-d* trees cannot have distinct voxels of large empty sections. So the traversal of the balanced *k-d* tree is faster, but on the other hand there is a loss in the intersection efficiency, since in both BSP Trees and octrees there may be large empty sections that are discarded fast [SF90].

As far as octrees are concerned, the worst case for storage space is $\mathcal{O}(n^2)$ and the construction time $\mathcal{O}(n^2)$, where n is the number of objects in the scene. On the other hand the worst case construction time for BSP and *k-d* trees is $\mathcal{O}(n \log n)$ and storage space $\mathcal{O}(n))$. The average case all the above complexities is very much dependent on the distribution of the objects in the scene, but in all cases it is close to $\mathcal{O}(n \log n)$ for construction and $\mathcal{O}(n)$ for storage [Hav00], [dBvKOS98], [LG98].

The above method that is space based (octree) is quite inefficient in terms of space requirements, since the storage of object information is not linear in n(the very worst case that can arise is $\mathcal{O}(n^2)$), where n is the number of objects. This will occur in the case where every object is intersected by a partitioning plane or is included in more than one voxel). On the other hand for the object based approach (*k-d* tree) this storage complexity is considerably less, indicating that object oriented techniques reduce space requirements.

Furthermore, the fact that each object can be stored in several nodes of a structure renders the approaches static or difficult to update. If for instance an object was to change place in the scene every voxel linked to this object would have to be updated and the voxel would have to

be tested again against all voxels of the structure. Furthermore, the structure would have to be reviewed again in order to join possible new adjacent empty voxels or to subdivide voxels that are no longer empty.

These particular issues of space requirements and updating difficulties are best addressed with bounding volume (object) hierarchies, which focus on partitioning the set of objects instead of space and more specifically with adaptive hierarchies that can accommodate changes in the scene without having upgrading excessive cost. This is why in our work we will focus on the special characteristics and traits of bounding volume (object) hierarchies.

2.2.2 Bounding Volume Hierarchies (BV)

The notion of hierarchical bounding volumes was originally mentioned in [RW80]. A number of object bounding volumes can be contained in a *parent bounding volume*. Rays that do not intersect the parent do not have to be tested against the enclosed bounding volumes. Thus the number of rays shot can be greatly reduced, by introducing a small overhead of testing the parent volume. This technique applied recursively provides a Bounding Volume Hierarchy.

We will provide here an overview of such hierarchical algorithms, organized by the type of Bounding Volume used as the lowest levels of the resulting hierarchy (bounding volumes of simple objects) as well as intermediate parent volumes.

2.2.2.1 AABB (Axis Aligned Bounding Boxes)

One of the most common types of bounding volumes for simple objects are 3D boxes with edges aligned to the scenes coordinate system. Some of the proposed hierarchies for such bounding volumes are presented here.

2.2.2.1.1 Bounding volume hierarchies In [GS87] an automatic procedure for generating object hierarchies was proposed and for the first time the notion of the probability a ray will hit an inner volume was used. In [GS87], the proposed procedure for constructing the hierarchy

tree is incremental. As each object of the scene is considered, the algorithm must decide which position in the tree is more appropriate for inserting the new object. In order to decide for the best position, the authors introduce a cost function, defined as the area by which a bounding sub-tree would be increased, were the new object placed at its root. The best position to insert the new object is the one yielding the minimum cost. The time consumed for the insertion of each object is thus logarithmic.

More specifically there are three possibilities for adding a new object.

1. Create a new root with one child being the old root and the other child being the new object.

2. Create a new root by simply adding the new object as a new child node.

3. Finally, the new object can be added as a descendant (not only child) of the root. In this case, some child has to be chosen in order to accommodate the new object most efficiently.

In order to test the quality of the created tree hierarchies, the authors compared the expected number of bounding volume intersections with the ones that actually took place. The comparisons demonstrate that the actual number of intersections is very close to the expected ones, while the time for rendering the scene is reasonably less than that of bounding hierarchies constructed by other automated procedures.

2.2.2.1.2 Top Down Median Cuts Since [GS87] several ideas have been expressed toward new ways to improve bounding volume hierarchies. Although most of these ideas were proposed to address object collision issues, some can be extended to ray tracing.

One such technique, using axis aligned bounding boxes, is that proposed in [Ber97]. According to this approach, a binary bounding volume tree is built. For each node, starting with the root node that contains all the primitives, the longest axis of the bounding volume is found and the partitioning plane will be chosen orthogonal to it. This is a top down approach, but not

an incremental one as that of [GS87]. It yields cube-like nodes, which is a good feature for collision detection, but doesn't provide any additional benefits to ray tracing. A δ coordinate is chosen on the longest axis, as the point where the partitioning will occur. According to the experimental results of the paper, the median of the AABB is the best choice. So all primitives in the AABB are characterized as either positive or negative with respect to the partitioning plane. If the projection of a primitive's midpoint on the axis is greater than δ then the primitive is labeled as positive; negative otherwise (Figure 2.11 [4]). If it so happens that all primitives in a node are classified as negative or positive, then the set of primitives in the node is split into two sets of almost equal size and the procedure continues.

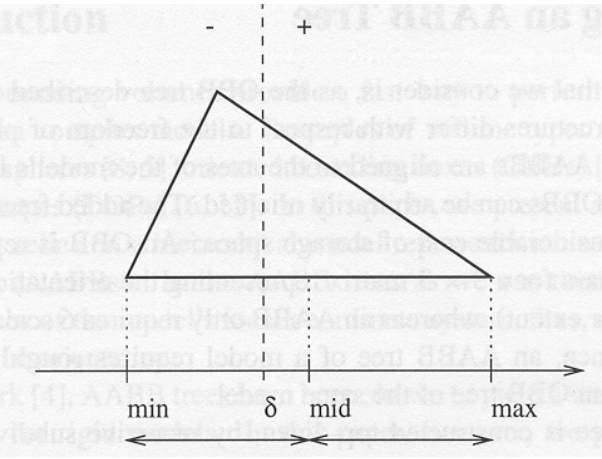


Figure 2.11: This primitive is classified as positive

2.2.2.1.3 R-tree The R-tree [Gut84] is a structure for handling multidimensional point data, that was adopted by the computer graphics society. An R-tree is a height balanced tree with its leaf nodes containing pointers to data-points and its nodes corresponding to disk pages. This structure is dynamic and designed in a way that spatial search requires visiting few nodes.

Each data-point is determined by an n-dimensional rectangle that surrounds it completely (tight bounding box). Having M as the maximum number of entries in a node and $M/2$ the

[4] image from [Ber97]

minimum, an R-tree has the following properties: firstly, every non-leaf node is the smallest rectangle that spatially contains the rectangles of the children nodes, secondly, the root node has at least two children (unless it is a leaf), and finally, all leaves appear in the same level in the tree.

When constructing the R-tree, a node is found to insert entry E, in such a way as to minimize the enlargement of the node in which the entry is inserted. If there is a tie the node with the resulting smallest area is chosen. After determining the node in which E is to be inserted, if this node has no room, it must be split and the tree must be readjusted where required. The split of an over-full node can be done in three different ways: firstly, exhaustively, by finding all possible splits and choosing the minimum area splits. It yields the best result, but has $\mathcal{O}(2^{M-1})$ complexity. Secondly, the split can be done with a quadratic algorithm in M. This split chooses the two entries that, if joined would cover the maximum area and puts them into different groups. The rest are assigned to the two new groups. This split provides a small area split, but does not guarantee the best split. Finally, a linear split algorithm can be used. Two entries are again chosen, the one with the highest low side (of the bounding box) and the one with the lowest high side, resulting in the extreme rectangle in all dimensions. The entries are normalized by dividing their dimensions by the extreme respective dimension. Then the pair with the greatest normalized separation along any dimension is chosen to form the two new groups. The remaining entities are assigned similarly to one of these groups.

In the testing we will provide, we will experiment with the linear split used in the R-tree, in order to achieve faster construction time.

When deleting an object, the entry is located and is removed from the node it was in; then, if that node has fewer nodes than it should, we eliminate the node and reallocate its entries; finally, the changes are propagated up the tree.

This adaptive technique of building index trees can be modified in order to produce object hierarchies for ray tracing. The notion of bounding box used in this paper is equivalent to the notion of bounding volume in object hierarchies. Both bounding boxes denote a upper and

lower value of an attribute (x,y,z coordinates) across an axis (dimension). Moreover, traversing the tree directly translates into casting rays and traversing object hierarchies.

2.2.2.1.4 R*-tree Several variation of the R-tree have appeared in literature. One of the most interesting is the R*-tree variation [BKSS90], created in order to address several of the drawbacks of R-trees.

The R*-tree basically follows the organization of an R-tree in all aspects but three. The first thing that differentiates the R*-tree is the algorithm for choosing a subtree for inserting an entry. Here an additional check is introduced. If the node we are visiting is one level from the leaves, then we chose the entry that needs the least overlap enlargement in order for the new entry to fit. This method is quadratic in terms of the number of elements in a node, but the authors propose to only check a subset of the elements, the largest entries, for overlapping.

Furthermore, all entries are sorted in every axis by the lower and upper value of their rectangles ($\mathcal{O}(n \log n)$ time) and when a split is needed the axis is chosen not only by the area increase (as in R-tree), but also by the margin increase (sum of length of sides) and the overlap increase. These choices, as well as the choice of subtree, provide a tree with less overlapping, and thus smaller average traversal paths.

Finally, in order to address the non-deterministic nature of the incremental R-tree (as far as order of inputs is concerned), when a split is needed on a node, a number of the entries of the node (larger ones) are reinserted. This procedure yields a better tree in performance and quality, but can result in high complexity in the creation (worst case close to n^2). Nevertheless, this is an optional part of the R*-tree implementation and we chose not to include it in our testing.

2.2.2.1.5 Comments on Axis Aligned Bounding Volume hierarchies The Top Down Median Cuts approach, although fitting for collision detection because it is fairly easy to refine the hierarchy, does not seem appropriate for ray tracing. The fact that the median of a dimension is used can result in more objects at one side of the tree. This will result in highly unbal-

anced and deep trees, which translates into multiple ray casts. A good solution instead of using the median for partitioning, is to use the point where the area of the objects (not the scene area) is balanced. This approach is used in Top-Down Binary Axis Partitioning as described in 2.2.2.1.2. Nevertheless, the above technique can be problematic in scenes where the midpoint of a bounding volume is not characteristic of the object's shape. Finally, it is not easily adaptable. In other words the hierarchy cannot be reused and updated when an object moves in the scene.

The R-tree and R*-tree adaptive indexing methods are quite fast ($\mathcal{O}(n \log n)$ average creation complexity), but do not yield optimal results. The structure of the resulting trees greatly depends on the order in which the elements were inserted, and there is a chance that the resulting trees will be poor in quality and in performance ($\mathcal{O}(n^2)$), if no re-insertion is used. Furthermore, the "empty" area in a node (area that no child node covers) is not taken into consideration. In the case of the R-tree, the overlap between parent nodes leads to trees with increased number of paths to be traversed, a situation dealt with in the R*-trees. The fact that the R-tree variants have insertion and deletion algorithms makes them easily adaptable when the scene displays relatively small changes.

2.2.2.2 OBB (Oriented Bounding Boxes)

Apart from AABB, hierarchies of OBB's have been studied because they fit the scene primitives tighter than AABB, resulting in a smaller number of lost rays r_o that have to be intersected with the actual object in the cost equation Equation 2.1. OBB's, as mentioned in [AK89], have been used for years for accelerating ray tracing. Nevertheless, when it comes to automatic bounding volume hierarchy generation several problems arise, namely how to compute a tight OBB and how fast can a ray intersection be calculated.

Several hierarchy models had been proposed, but none of them works well for large unstructured scenes, until [GLM96]. This technique, which constructs a hierarchy known as the OBB-Tree, aimed at solving collision detection issues. The choice of OBB's for a polygon

primitive according to other approaches has as follows. All polygons are triangulated and the mean and covariance matrix of the vertex coordinates are computed. The eigenvectors of the covariance matrix are mutually orthogonal and after being normalized can be used as basis vectors to bound the external vertices. The paper also proposes the use of convex hulls of the primitives instead of the triangulation. So the mean and the covariance comes from sampling uniformly the convex hull (the denser the sample the tighter the fit). We mention here that non-polygonal objects are not addressed by the technique.

As far as the hierarchy is concerned, the method adopts a top-down approach that results in a binary tree. A split of a OBB is done along the longest axis of the box with a plane orthogonal to it at the mean point, partitioning polygons according to which side of the plane their center point lies on.

In order to determine if an axis separates 2 polygons (used in tests for collision detection), a theorem is introduced. A line $\vec{L}$ is a separating axis of two OBB's if and only if the projections of the boxes' axis onto $\vec{L}$ are disjoint. The theorem also states that 15 axial projections are enough to determine if two OBB's collide.

The above approach, although introduced for collision detection, can be extended to ray tracing by viewing a ray as an OBB of infinite length and zero width and height. The separating axes theorem can then be applied by calculating only 3 of the 15 axes. This method, although it greatly reduces r_o, still introduces a big overhead into the computation of OBB's and the intersection test. Nevertheless, the fact that all polygons are triangulated makes the approach quite unsuitable for ray tracing, since the creation of the hierarchy can just as easily be replaced by rendering polygons using hardware.

2.2.2.3 Boxtrees

Another approach is the Boxtree [BCG+96], a bottom up extension of the R-Tree's structures for non axis aligned bounding boxes. The technique used in Boxtree for creating the new father bounding box is quite similar to that of OBB-Tree, but the resulting bounding boxes are aligned

only to one vector (corresponding to the largest or smallest eigenvalue of the covariance matrix of the point coordinates). The other two directions are determined by computing the exact minimum area bounding rectangle of the projection of the points coordinates onto a plane orthogonal to the original direction. Apart from bounding volumes boxes (axis aligned and oriented ones) this methodology is also used by the authors for triangular pyramids.

2.2.2.4 k-DOPs (Discrete Orientation Polytopes)

In order to gain the most of fixed orientation bounding boxes as well as OBB, in [KHM+98], a hierarchy is introduced using a k-DOP bounding volume [KK86]. This type of bounding volume is a convex polytope with faces defined by half-spaces that come from a fixed number k of orientations. For instance AABB are 6-dops with orientation vectors dictated by the coordinate axes. k-DOPs are closer to the sense of convex hulls and OBBS, but at the same time the intersection tests are bound by the number k and are tests against k fixed directions (Figure 2.12 [5]).

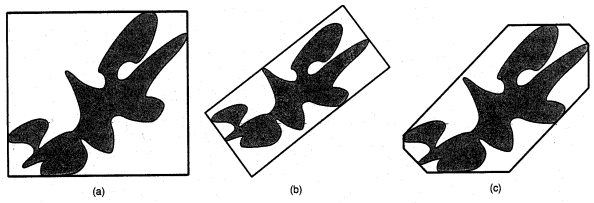


Figure 2.12: (a) axis aligned bounding box; (b) oriented bounding box; (c) k-dop (k=8)

The authors propose for the k normal vectors a set where the X,Y,Z coefficients take on the values $\{-1, 0, 1\}$, so that no multiplications are needed.

The tree construction is a top-down approach and a number of techniques for splitting are mentioned (minimum sum of the volumes for resulting children, minimum larger volume of resulting children, axis of larger variance, axis with bigger k-DOP). The focus is mainly driven toward the volume of the children (since it is important for collision detection). As it is

[5] image from [KHM+98]

mentioned in [KHM+98] in order to adapt it to ray tracing, surface and not volume should be the main consideration.

2.2.2.5 Others

Finally, there are several preprocessing methods that aim at "directing" the creation of the hierarchy. One such framework is described in [TCL99]. According to this guide, a preprocessing step is used in order to partition the scene into disjoint components that are a tighter fit for the primitives. These components will form the top levels of the bounding volume hierarchy. Then, nearby nodes are paired, in a bottom-up pass, to build a binary tree. The partitioning step of the original scene (assumed to be composed of triangles) groups together triangles that share sides, edges of these triangles that only belong to one triangle or that touch triangles of the group, as well as points that are only incident to a triangle in the group. After that, any technique can be used for the rest of the bounding volume hierarchy.

2.3 Summary

In this chapter we mentioned the most common techniques for reducing both the cost of intersecting rays and the number of rays reaching an object. Since Bounding Volumes have proven to accelerate considerably the intersection calculations, we will definitely make use of them in the work to follow.

After having reviewed Space Subdivision techniques we have concluded that as approaches they have high storage requirements and that the resulting structures are not easily reusable when the original scene is slightly altered.

Bounding Volume Hierarchies are less space demanding and present an average creation time close to that of Space Subdivision techniques. Several types of such hierarchies exist, distinguished by the criteria for splitting (or joining) bounding volumes and by the bounding volumes they use. From the examination of representatives of these techniques it is apparent

that hierarchies using Axis Aligned Bounding Volumes are easier and cheaper to compute, but are less tight in representing the scene than Oriented Bounding Boxes or Orientation Polytopes.

Chapter 3

Overview of Our Approach

3.1 Idea Behind Our Approach

In order to help speed up ray tracing, all the approaches mentioned so far aim either at reducing the number of rays tested against objects by using object hierarchies, or at reducing the ray intersection cost using bounding volumes. Nevertheless, an intersection test, even against a bounding volume, takes up time and computational effort.

We will examine an entirely new approach in order to eliminate most of the cost of intersecting a ray with a bounding volume in object hierarchies, adding a small overhead in the preprocessing steps. The main ideas in this approach are as follows:

① Before rendering the scene, *all object bounding volumes are projected on the viewing plane*. Thus instead of testing for 3D intersections between any ray and a bounding volume, a test of whether the ray origin is inside the bounding volume projected area suffices. In other words we project the scene into its 2D equivalent onto the viewing plane to eliminate 3D calculations on the bounding volumes. We hope that the 2D bounding box, derived from the 3D one, is faster in intersecting, even if it keeps less information about the actual object.

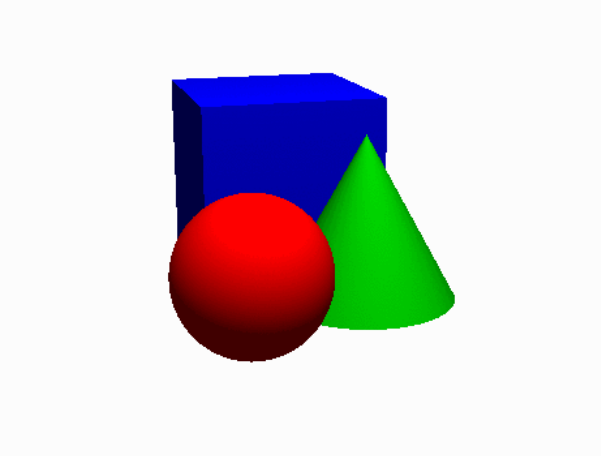


Figure 3.1: A 3D scene

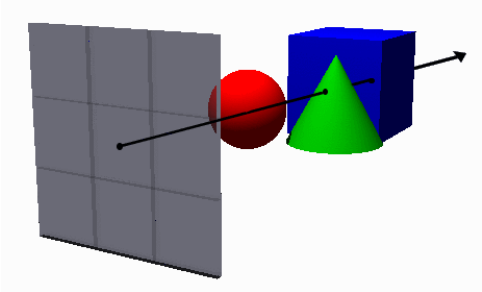


Figure 3.2: Ray cast into the scene (3D approach), for scene in Figure 3.1

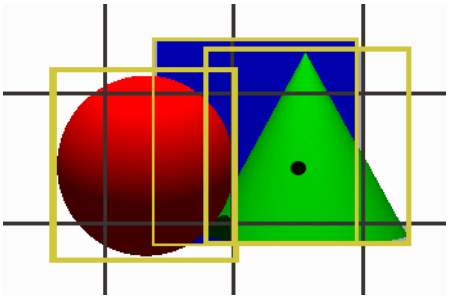


Figure 3.3: Point in 2D box check, our 2D approach, for scene in Figure 3.1

② Furthermore, *we create a 2D based hierarchy, using the projections discussed above, so as to further accelerate the approach*. Thus objects are grouped according to the density of the scene, as perceived from the viewing plane. Again, since the 2D boxes are not as tight as the 3D ones (we use the word "tight" to indicate that they model the object more closely), we want to see if the gains of 2D intersections on traversing the hierarchy can compensate for the "loose" fit.

③ We also *create view independent hierarchies, which we project on the viewing plane*. In other words we keep the structure of the view independent hierarchy to form a projected one, where all internal nodes are the projected equivalents of the view independent one.

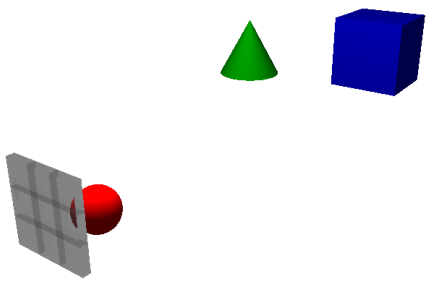


Figure 3.4: A simple scene with 3 objects and it's viewing plane.

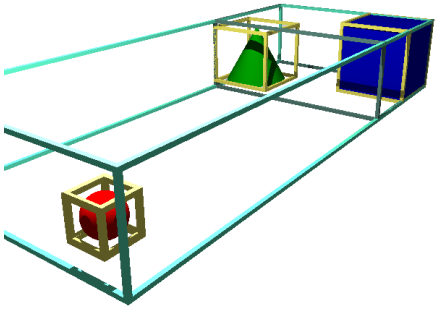


Figure 3.5: A 3D hierarchy for the scene in Figure 3.4.

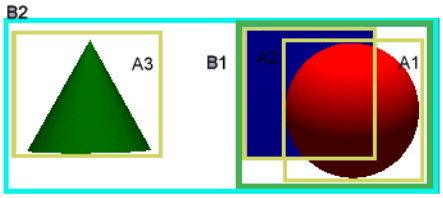


Figure 3.6: A 2D hierarchy based on 2D criteria.

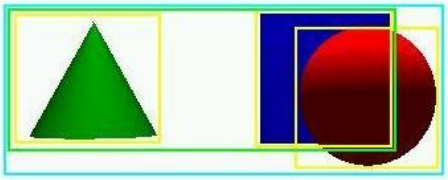


Figure 3.7: A 2D hierarchy derived from the 3D hierachy of Figure 3.5.

Thus we get the information of the $z - axis$ (third dimension) as well as the benefits of the 2D hierarchy intersection and traversal.

④ In order to acquire a detailed picture of the performance of the above hierarchies, we will try different intersection schemes when traversing the 2D projection hierarchies. We will use 2D intersections on the bounding boxes, 3D, as well as combination of the above.

⑤ Finally, we will test 2D and 3D hierarchies with several hierarchy building algorithms.

We expect that the 2D hierarchy intersection approach reduces the weight of ray casting in the first level rays (visibility rays), which in some scenes tend to be the vast majority of rays shot. Furthermore, we want to verify if the cost of the projecting bounding volumes, building the 2D hierarchy, and testing if ray origins are inside the projected areas, is trivial compared to that of ray casting into a 3D scene.

We believe that the main gain of having a 2D hierarchy comes from using a simple point-in-rectangle containment tests on the bounding boxes of the hierarchy, rather than ray-box intersection tests. So it remains to see if the gains of a 2D test on a 2D bounding box can compensate for the fact that the 2D bounding box contains less information about the 3D object than a 3D bounding box.

3.2 Implementation Issues and Decisions

We will now explore some details concerning the architecture of the approach. As we mentioned, based on the projection of the bounding volumes a hierarchy is built (the case of the 2D hierarchy). Although the criteria for building this hierarchy are all calculated in 2D, we keep the original 3D bounding volumes. So if we want to animate the viewing camera and the scene remains static, for small angles, the hierarchy does not need to change, only the components should be re-projected.

For the 3D hierarchy that uses 2D traversal methods, we first build a view independent hierarchy and we project every level of the hierarchy to the viewing plane. To be more precise, the original structure of the view independent hierarchies is preserved (ie if two objects are joined in the view independent hierarchy they are also joined in the projected ones). Nevertheless, the intermediate levels of the hierarchy, although following the structure of the view independent hierarchies, are constructed based on the 2D bounding volumes of their children, not on projecting their view independent counterpart (Figure 3.8).

The 3D view independent hierarchies tested follow the conventional method and are implemented in order to compare their performance with that of the 2 above forms of projected hierarchies.

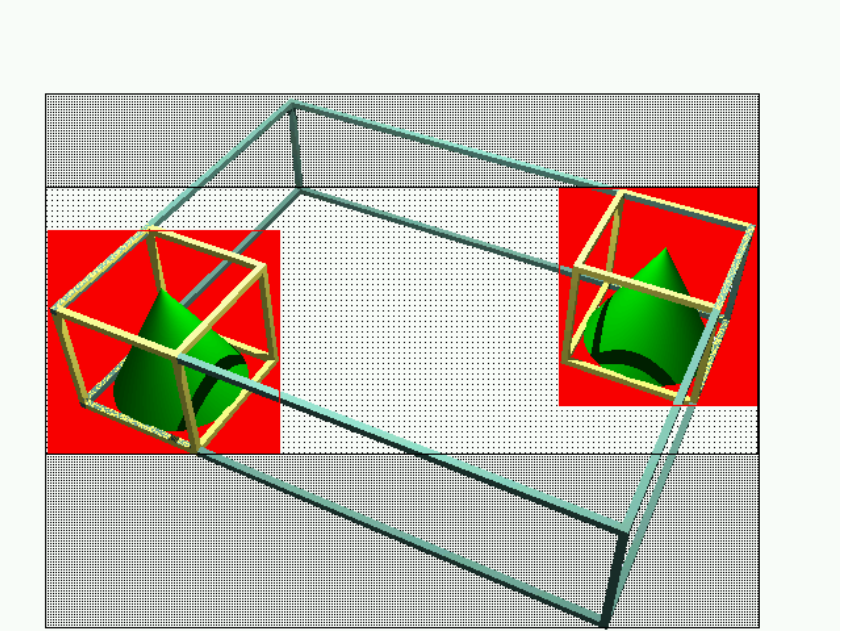


Figure 3.8: The inner 2D rectangle represents the joining of the 2D bounding boxes, following the 3D joining structure. The outer rectangle is what the projection of the intermediate level of the hierarchy would be like. Our 3D hierarchies using 2D intersections use the intermediate nodes based on joining the 2D boxes (inner rectangle).

3.2.1 Lights and Shadow rays

The projection onto the viewing plane aims at accelerating first level rays, that is visibility rays. In a similar way we deal with shadow rays. We view lights as cameras and decide on planes surrounding the light for creating hierarchies. Rays originating from lights are viewed in the same way as visibility rays, in order to maintain a uniform approach for all rays. Shadow rays too must be tested to identify the appropriate hierarchy. In other words, a shadow ray is first created from a specific light and then tested against all lights in order to find the correct hierarchy to use. This approach costs $\mathcal{O}(\log l)$ to choose the correct light from which to get a hierarchy, where l the number of lights, whereas if the shadow ray was only considering the light hierarchies of the light it originated from it would cost a constant time. The logarithmic factor above comes from the structure that the lights are stored in. Although this way

of treating shadow hierarchies is redundant, it provides us with a uniform method of treating rays, irrespective of their origin. As it turns out (Section 6.3), this redundant procedure actually degrades the performance of our approach. Thus apart from the data concerning the uniform approach we also present some from cases where shadow rays are considered to have "knowledge" of their light origin.

If we are dealing with point lights inside the scene, then we apply a light cube, that is we define 6 planes completely surrounding the light. Onto them we project and create 6 projection hierarchies. When the actual ray tracing is done the hierarchy that is closer to the object tested is chosen.

If the light is a directional light, or a light outside the scene volume and far away, one plane is enough for creating a 2D hierarchy for the ray tracing procedure. The cost of building a hierarchy, as we will demonstrate in our experiments, is extremely cheap compared to the ray casting procedure in small scenes. So the overhead of building hierarchies for each light is small, but it becomes noticeably large as the number of lights and objects in the scene increase. Even small area lights can be accommodated by a single 2D hierarchy (it's like moving the camera a little bit) and reprojection of the bounding volumes. Nevertheless, if the area of the light becomes relatively big, then a single hierarchy will probably not be as efficient as several more specific ones.

So shadow rays also profit from using point-in-rectangle containment tests and hierarchies created specifically for the casting light, as far as traversal and intersection on bounding boxes is concerned. Of course, rays must first be translated (projected) to the light plane (an operation that costs up to 21 FLOPS, as we will see in Chapter 5).

Given that lights are usually static in a scene, this preprocessing cost for the light hierarchies can become insignificant when a scene is rendered for an animation. On the one hand, if the camera changes position in the scene the light hierarchies remain unaffected; on the other hand, if a few objects change position, then an adaptive light hierarchy (such as one built using an R-tree variant) only needs a small update.

3.2.2 Other rays

Of course the projection hierarchies do not speed-up secondary rays, that is reflection and refraction rays. In order to deal with them in the 2D hierarchy approach we propose to also built a global, view independent 3D hierarchy. This hierarchy needs only to be built once and works as all other view independent hierarchies: it aims at reducing the number of objects a secondary ray tests for intersection. So we also get an acceleration benefit for the secondary rays by adding a preprocessing cost. This once built hierarchy is a standard acceleration technique for animations. As for the 3D hierarchies that are later projected on the viewing plane and lights, the 3D element is the one used for secondary rays.

In an animation, the view independent hierarchy, just like the light hierarchies, needs to be updated only when an object moves through the scene. This 3D hierarchy is used only when objects in the scene have specular surfaces, and as the number of specular surfaces increases in a scene, and with them the number of secondary rays, the acceleration provided by the 2D intersection methodology tends to degrade (although primary rays will always be cast faster).

3.3 Choice of Bounding Volumes and Hierarchies

We will explain here our choices of bounding volume hierarchies (projection and global), as well as bounding boxes and the projection shapes we used. Although our idea is quite general, we have implemented it using approaches that we believe are quite representative and efficient.

3.3.1 Bounding Volumes and Projected Shapes

In our choice of bounding volumes for our primitives, we have come to the conclusion that Axis Aligned Bounding Boxes (AABB) best serve our purposes, because of the following properties:

- AABB are faster and simpler to intersect with rays than Oriented Bounding Boxes (for

all comparisons see Chapter 4).

- Although they are more expensive to intersect than spheres they are easier to join or split.

  Assume two spheres, one with center $\mathbf{S}_{c1}$ and radius S_{r1} and the other with center $\mathbf{S}_{c2}$ and radius S_{r2}. In order to join them into one bounding sphere we have to first calculate the distance between their centers (an operation requiring a square root evaluation, which is costly) in order to calculate the diameter of the new sphere ($dist(\mathbf{S}_{c1} + \mathbf{S}_{c2}) + S_{r1} + S_{r2}$) and the center of the new sphere.

  On the other hand, the surrounding box of two AABB's is determined by the extreme values of the two boxes' vertices.

Given the fact that we will not be performing the ray-bounding volume tests in our 2D hierarchy, we do not seek bounding volumes with cheap ray-BV tests (such as spheres) but rather bounding volumes that may be merged cheaply (thus accelerating the generation of the 3D hierarchy based on the 2D joining).

As we will see in Chapter 4 spheres project to circles and bounding boxes to hexagons (in the general case). For both these 2D shapes the cost of testing if a point is inside (the ray origin) is somewhat high (there are square root used for the tests on the circle and in-polygon tests for the hexagon). This is why we propose the use of a 2D box enclosing the projection of a bounding volume. This box is aligned to the viewing plane, thus the inside calculation is simply a matter of comparing 2D coordinates. Nevertheless, this 2D box is less accurate than the actual projection, but given the fact that the testing is so cheap, we will here try to determine if it is affordable. Besides, some of the hierarchies based on the 2D projection that we will be testing, require axis aligned (in 2D) bounding boxes.

3.3.2 Intersection Schemes

As we mentioned when briefly describing our work, we will use several intersection schemes for traversal in our 2D hierarchies:

2D intersection traversal: A simple point containment in polygon test. Every ray is projected on the plane of interest (unless it is a visibility ray, so the starting point of the ray is sufficient) and than tested against the 2D bounding box at the root of the hierarchy (a simple in-polygon test). If the test is true, then the ray is tested against the 2D bounding boxes of the children.

3D intersection traversal: The normal 3D intersection against a 3D bounding box. For the 2D hierarchies it should be noted that although the hierarchies are built using 2D (projection) criteria, the test of the rays is done on the 3D bounding boxes in the levels of the hierarchy, thus taking advantage of the tighter 3D boxes.

Combination intersection traversal: A first fast test is performed on the 2D bounding box in the hierarchy. If the test is true, a test is also performed on the tighter 3D bounding box, to eliminate rays that were successful on the loose 2D bounding box.

All the above tests have meaning in the 2D hierarchies built using 2D criteria. Nevertheless, only the 3D intersection and traversal can be performed in the case of the 3D independent hierarchies (since no projection information is kept). Finally, in the case of the 2D hierarchies that come as a direct translation and projection of an independent 3D hierarchy, the 3D intersection test makes them degenerate to the independent 3D hierarchies. So only the 2D and combination intersection traversal are of interest.

3.3.3 Hierarchies

For our testing purposes we have implemented and tested both view independent and projection-based approaches for comparison. The projected based hierarchies are as we mentioned of two forms, which basically have to do with the criteria used to build the 2D hierarchies. The one form uses the 2D information of the actual objects of the scene. The other takes an existing view dependent hierarchy and translates it into a 2D plane (viewing plane or plane of a light cube).

In order to best review the results of the 2D intersection approach we chose some representative hierarchy building approaches: top-down, bottom-up and adaptive.

R-tree: The R-tree is a classical top-down adaptive method. It divides objects using an axis aligned partition and minimizes the surface area of the joint bounding volumes.

R*-tree: The R*-tree is also an adaptive top-down hierarchical approach. It is different to the R-tree in that it can join bounding volumes so as to minimize overlap, something very useful in ray tracing (less subtrees traversed in the hierarchy).

Top-Down Binary Axis Partitioning: This is a static top-down approach (also referred to as Top-Down Binary Splitting TDBS). The tree is binary and each time the node is split on one of the 3 axes, so as to derive the best linear partitioning in terms of surface area (children with approximately the same surface area for the bounding volume). It is linear in the sense that not all combinations of objects are tried at each axis which would cost $\mathcal{O}(2^n)$ time. Instead, the bounding volumes are sorted according to their center on the given axis and linear combinations are tested (the first bounding volume and the rest, the first two and the rest, etc), which is a $\mathcal{O}(n)$ operation.

Bottom-Up Binary Matching: We introduce this approach in order to cover an optimal binary matching among the objects in the scene. It is a static bottom-up approach. At each step two bounding volumes are joined, those with the smallest surface area. This resembles a Nearest Neighbor search, but we define the distance metric as the combined surface area. The generated tree is a binary tree. As we designed and implemented this approach, it takes $\mathcal{O}(n^2)$ time. We first sort all bounding volumes, according to the shortest distance with their closest neighbor. The first pass for the closest pair matching is the one that takes up $\mathcal{O}(n^2)$ time, but the actual cost is very small, since the pairing is based on a few coordinate comparisons. The sorting is a $\mathcal{O}(n \log n)$ calculation. At each step the two objects that are closest are joined and their bounding box replaces them in the sorted structure (after having found its nearest object in $\mathcal{O}(\log n)$ time, or $\mathcal{O}(n \log n)$ for all the

merging operations). The most costly operation (determining originally the closest object to every other object), can be reduced in complexity if we use a slightly modified approximation algorithm for Nearest Neighbor finding ($\mathcal{O}(n \log n)$ [KOR98, IM98, Tsa99]).

The above hierarchies are implemented both in a view independent matter (as seen in the bibliography so far), as well as projection hierarchies based on the 3D hierarchies and on 2D criteria. The fact that we use AABB allows the implementation of the R-tree variants in the view independent hierarchies (where the splitting and joining criteria require AABB), but it also facilitates the projection hierarchies, because the joining of axis aligned bounding boxes (in 3D) is much faster than the joining of spheres or oriented boxes.

3.4 Algorithm

A description of the ray tracing algorithm can be seen in Figure 3.9), with the preprocessing steps we propose (Figure 3.10), including secondary and shadow ray testing. Note that the `Find_Project_Intersection` test on a projection of a bounding volume is much cheaper than the actual intersection test on a bounding volume, as we will demonstrate.

The `Build_Project_Hierarchy` function described in Figure 3.9 is the one for creating the 2D projected hierarchy. When we use a 3D hierarchy that we later project, the function is that presented in Figure 3.11.

Also, in this description of ray tracing (Figure 3.10) we express `Intersect_Object` as a 2D operation, that is a ray is tested on the 2D projected box of the derived 2D hierarchy. Nevertheless, as we have mentioned, in order to better understand the performance of all hierarchies involved and to enhance their performance, we use other intersection schemes on the projected hierarchies as well. So for the projected hierarchies which have been created on 2D criteria we also try intersections on the 3D bounding boxes of the objects, as well as on a combination of 2D and 3D bounding boxes; whereas for the projection hierarchies derived from view independent ones we use 2D and combination intersection schemes.

```
Initialization:
     Build_Global_Hierarchy H_G
     forall lights l
        Build_Light_Hierarchies  H_l for l
     Build_Project_Hierarchy H_C for camera C

Build_Light_Hierarchies H_l for light l
     if l inside scene
        for the 6 directions d of the cube around l
                forall objects n Project on d
                        Build_Project_Hierarchy  H_{l 1 to 6}  for d
     else
        forall objects n Project on d closest to scene
                Build_Project_Hierarchy H_{l 1}  for d

Build_Project_Hierarchy H for C
     forall objects n in scene
        Project n on C
        Add n to H for C
```

Figure 3.9: Initialization Procedure

```
Ray_Tracing:
     forall pixels of viewing screen
           shoot ray and Find_Project_Intersection using H for C

Find_Project_Intersection using  Project Hierarchy H for C
     if node is a leaf in H
           Intersect_Object n
     else
           if ray origin in projection of bounding volume
                 for each child of node in H
                       Find_Project_Intersection of child using H for C

Find_Intersection using Global Hierarchy H_G
     if node is a leaf in  H_G
           Intersect_Object n
     else
           if ray intersects bounding volume
                 for each child of node  in H_G
                       Find_Intersection of child using H_G

Intersect_Object n
     if intersection
          forall lights l
                 shoot ray from light l to intersection on n
                 using Find_Project_Intersection using Project Hierar-
chy H_l) closer to scene
          if reflection or refraction ray
                 Find_Intersection using Global Hierarchy H_G
          calculate color
```

Figure 3.10: Ray Tracing

```
Build_Project_Hierarchy H for C
     Get Global hierarchy H_G of scene
     forall objects and levels of H_G
           Project object bounding boxes of on C
           Add projected box at the same level as the H_G in H
```

Figure 3.11: Building a 2D hierarchy using a 3D view independent one

Chapter 4

Intersection Cost Analysis

The basic cost for ray tracing comes, as we have already mentioned, from the number of ray-object intersections needed in a scene, as well as the actual cost of these intersections. In order to determine in what ways our technique will affect the ray tracing rendering time, we will first try to give an approximate cost for the ray tracing techniques commonly used. We must mention here that bounding volume hierarchies are not useful in all scenes. They definitely improve ray tracing efficiency when the objects of the scene are quite simple to intersect (low cost) and the number of objects is relatively big. But this is not the case for simple scenes (small number of objects) with expensive primitives (complicated shapes). In these cases (for example a scene with a very complicated object that covers the entire scene) hierarchy traversals and even bounding volumes will not affect the overall performance.

4.1 Bounding Volumes Intersection Cost

As we have already pointed out, the main factor that affects ray tracing efficiency is the ray-environment intersection calculations. In order to demonstrate this high cost we will review the intersection calculations on bounding volumes, which are always as cheap or cheaper than those on the actual objects.

In order to give a cost of the intersection calculations we will use as a basic unit the time

taken by the different floating point operations. Thus we will roughly compute the time needed to intersect a bounding volume.

We will assume that a floating point addition and multiplication take a fixed amount of time (which we will use as our base meter) and a division approximately 5 times as an addition. A comparison operation is assumed to take as much time as an addition, since typically floating point comparisons are implemented with subtractions.

A ray is defined as:

$$\begin{array}{rcrcl} \mathbf{R}_{origin} & \equiv & \mathbf{R}_0 & \equiv & [X_0\ Y_0\ Z_0] \\ \mathbf{R}_{direction} & \equiv & \mathbf{R}_d & \equiv & [X_d\ Y_d\ Z_d] \end{array} \quad \text{where } X_d^2 + Y_d^2 + Z_d^2 = 1 \text{ (normalized).}$$

This defines a ray as a set of points on line

$$\mathbf{R}(t) = \mathbf{R}_0 + \mathbf{R}_d * t,\ where\ t > 0. \tag{4.1}$$

4.1.1 Ray-Sphere Intersection

The simplest bounding volume to intersect with in a 3D environment is a sphere, as we will see by the calculations needed.

The equation representing a sphere is given by a vector $\mathbf{S}_c$ denoting the center of the sphere and a number S_r denoting the radius.

We will describe the geometric solution, as mentioned in [Hai89], which is significantly faster than the typical algebraic solution. We note that we have a slightly different cost associated with the algorithm, since we count comparisons and we assume that the square of the radius is a precomputed factor for all sphere bounding volumes.

1. **Ray Origin Inside or Outside of Sphere** Find the square of the distance L_{oc} between $\mathbf{R}_0$ and center $\mathbf{S}_c$ and compare with square of radius length S_r (avoid calculating square roots) (Fig 4.1).

   We have 3 subtractions from calculating the vector $\mathbf{OC} = \overrightarrow{\mathbf{R}_0\mathbf{S}_c}$, 3 multiplications and 2 additions for the square length of $\mathbf{OC}$ and a subtraction for the comparison

(we assume that the square of the sphere radius is precomputed). The total cost is $3 + 3 + 2 + 1 = 9$.

2. **Ray Direction with respect to Sphere** Calculate ray distance closest to center t_{ca} and see if it is greater than 0. If $t_{ca} > 0$ then the ray points toward the sphere, otherwise away from it (Figure 4.2).

   Essentially the calculation of the distance t_{ca} is equivalent to finding the intersection of the ray with a plane perpendicular to it, which passes through the center $\mathbf{S}_c$ of the sphere. This step of the algorithm will take 3 multiplications and 2 additions for the ray-plane intersection ($\mathbf{OC} \cdot \mathbf{R}_d$) and the comparison here does not need a subtraction. So the cost is $2 + 3 = 5$.

3. **Does ray miss the sphere?** If ray origin is outside the sphere and points away then the ray does not intersect the sphere and the total cost is $9 + 5 = 14$. Else the algorithm continues. This step has no computation cost since all comparisons have already been made.

4. **Does ray hit the sphere?** Find the square of the distance h_{hc}^2 between the closest distance of ray to center and the center. If this distance is less than 0 then the ray misses the sphere (Figure 4.3).

   This step takes up 1 addition, 1 subtraction and 1 multiplication (for t_{ca}^2), as computed in Figure 4.3 using the Pythagorean theorem (we have already computed L_{oc}^2 and we know S_r^2). The comparisons again take no time. The cost is then 3. So the total cost can be $14 + 3 = 17$ or the algorithm continues. This check for h_{hc}^2 makes sense only if the origin is outside the sphere.

5. **Intersection Point** We have determined if a ray hits an object and we need to calculate the intersection point. t from Equation 4.1 is $t = t_{ca} - \sqrt{h_{hc}^2}$ if the ray originates outside and $t = t_{ca} + \sqrt{h_{hc}^2}$ if it originates inside. Note that we only know the

square of h_{hc}^2, so we need to calculate the square root. This calculation takes as much as 15-30 times as a multiplication (we will assume 15).

The actual point can be found by Equation 4.1 with 3 additions and 3 multiplications for the 3 dimensions. Finally the normal on the intersection point normalized takes 3 subtractions ($\mathbf{P} - \mathbf{S}_c$) and 3 multiplication (assuming $\frac{1}{S_r}$ is precomputed).

The entire step costs $1 + 15 + 3 + 3 = 22$.

So the total cost for intersecting a sphere can be either 14, 17 or 22.

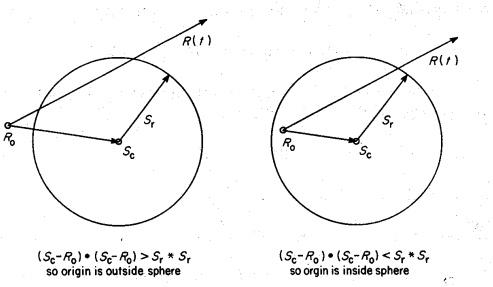


Figure 4.1: Ray origin inside or outside sphere ([Hai89])

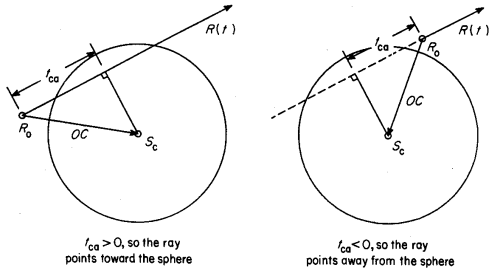


Figure 4.2: Ray direction with respect to sphere ([Hai89])

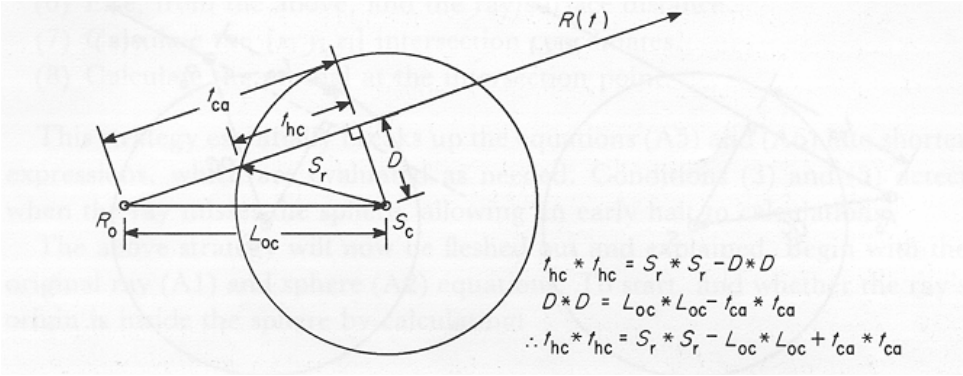


Figure 4.3: Sphere intersection ([Hai89])

4.1.2 Ray-Box Intersection

In [KK86] the ray-box calculation is determined using slabs. A slab is the space between two parallel planes. So the intersection of a set of slabs defines a bounding volume.

An orthogonal box can be defined by 2 coordinates, one describing the maximum $\mathbf{B}_h$ and the other the minimum $\mathbf{B}_l$ extend of the box.

$$\begin{aligned} \mathbf{B}_l &= [X_l\ Y_l\ Z_l] \\ \mathbf{B}_h &= [X_h\ Y_h\ Z_h] \end{aligned}$$

and a ray is, as we mentioned, given by Equation 4.1.

4.1.2.1 Axis Aligned Box

We will describe the algorithm for an axis aligned box.

1. We first set $t_{near} = -\infty$ and $t_{far} = \infty$.
2. **For each** of the 3 plane directions (we will describe the X parallel planes), the following take place.
   (a) If $X_d = 0$ then the ray is parallel to the box. If X_0 is not between the slabs ($X_0 < X_l$ or $X_0 > X_h$), then the ray does not hit.

   This step takes up one comparison with 0 that does not affect the time of the algorithm and 2 comparisons of real numbers. So the cost is 2.

   (b) If ray not parallel to the plane

   i. Intersection distances of planes are calculated:

   $t_1 = (X_l - X_0) * \frac{1}{X_d}$

   $t_2 = (X_h - X_0) * \frac{1}{X_d}$

   This step takes up to 2 subtractions, 2 multiplications and one division (we store $\frac{1}{X_d}$). So in total $2 + 2 + 5 = 9$.

ii. If $t_1 > t_2$ swap them (cost 1)

If $t_1 > t_{near}$ set $t_1 = t_{near}$ (cost 1)

If $t_2 < t_{far}$ set $t_2 = t_{far}$ (cost 1)

iii. If $t_{near} > t_{far}$ box is missed (this step has cost 1 and if the algorithm stops here the total cost is 2+9+1+1+1+1=15).

iv. If $t_{far} < 0$ the box is behind the ray so it is missed. This comparison takes no time, so if the algorithm ends here the cost is 15 (Figure 4.4).

v. Else continue with next axis.

End for

3. If all the planes are checked and no break in the algorithm is found, we have gone through $3 * 15 = 45$ steps of cost 1.

4. We still need to determine the intersection. The intersection distance is t_{near} and the ray's exit point is t_{far}. The actual intersection point can be calculated by Equation 4.1, costing an extra 3 additions and 3 multiplications.

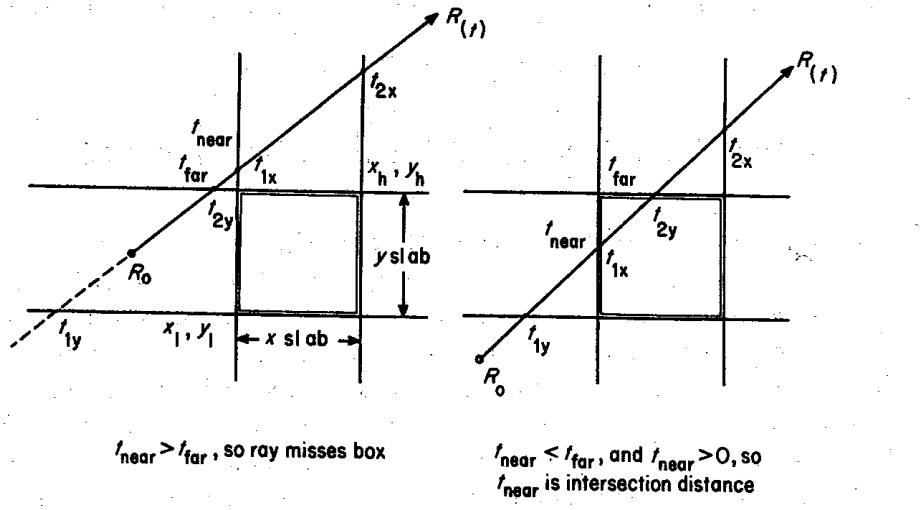


Figure 4.4: Sphere intersection ([Hai89]).

So for an axis aligned box the cost of intersecting can be 2 or 15 (if we exit on first plane), 15+2=17 or 15+15=30 (if we exit on second plane), 30+2=32 or 30+15=45 (if we exit on third plane), or 45+6=51 (if there is an intersection).

4.2 Projection and Comparisons

The simplest of the bounding boxes have costs that range from 14-22 or 2-51. We will now present the cost of ray intersection with the projection of a bounding volume. We note here that this calculation takes place when traversing the hierarchy. When we determine the actual object hit at the bottom of the tree, we will perform all the calculations.

4.2.1 Circle

The projection of a sphere is always a circle of center C_c and radius C_r. Both C_c and C_r are precomputed and are the projection of the center S_c of the sphere and the projection of the radius vector parallel to the image plane S_r respectively (Figure 4.5).

In order to test if there is an intersection of a ray as described by Equation 4.1 with the sphere, we only need to test if the origin R_0 of the ray lies inside the circle. This is performed by checking if $(X_c - X_0)^2 + (Y_c - Y_0)^2 > C_r^2$. If the length of the 2D distance between R_0 and C_c is bigger than the radius than the ray does not hit the sphere (since the sphere is translated on the viewing plane in a position that the ray is not coming from).

This test has a constant factor of 1 addition, 2 subtractions and 2 multiplications (the square of the projected ray radius is precomputed). So the cost is 5.

4.2.2 Projected Axis Aligned Bounding Box

An axis aligned bounding box will have 8 projected vertices (Figure 4.6) that have a convex hull of at most 6 vertices (4 can occur in the case of a viewing plane parallel to one of the axis of the box). Using the Graham algorithm we can find the convex hull in $\mathcal{O}(n \log n)$ time. In

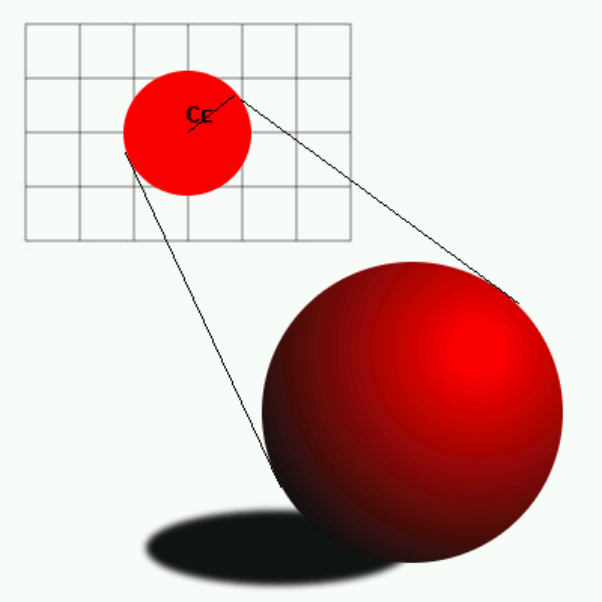


Figure 4.5: Sphere projection on viewing plane.

order to check if a ray hits the 3D box we have to check if the origin of the ray is inside the convex hull, using for instance the Jordan curve theorem.

Because we are dealing with axis aligned 3D bounding boxes, the cost of finding the 6 edges that form the convex hull of the projected nodes can be improved and computed in constant time (not $\mathcal{O}(n \log n)$ using the Graham algorithm). In order to do so we must pick an octant for the ray direction and determine thus which 3D box edges will project to 2D box edges, as shown in Figure 4.7.

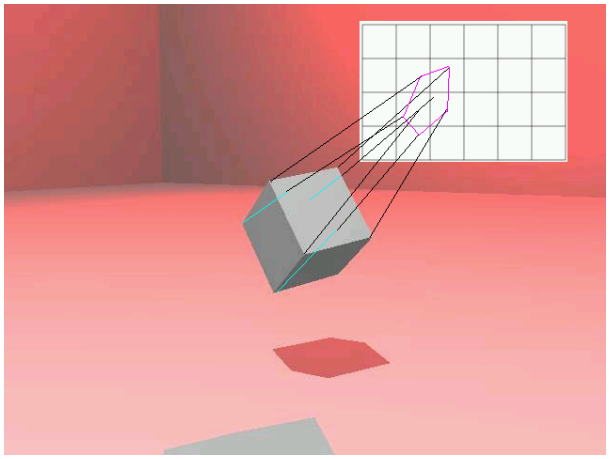


Figure 4.6: 3D Box projection on viewing plane.

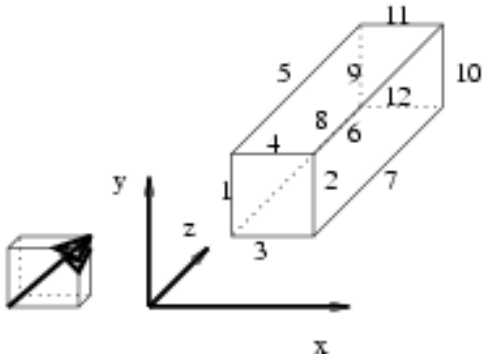


Figure 4.7: Assume that the ray direction defines the positive axes x, y, z of the octant. In our example the edges 2,4,5,6,12,7 bound the 2D hexagon. Or if we look at the $-x, y, -z$ directions the BV edges 1,3,7,10,11,5 bound the 2D hexagon.

4.2.3 2D axis aligned box

Every projected bounding volume can be enclosed in a 2D box, aligned to the viewing plane. This box can act as a fast intersection test, since checking if a ray originates from within takes only 4 comparisons.

Essentially by projecting the bounding volumes in the scene we reverse the process of ray shooting. Instead of casting rays from the viewing plane to the scene, we transform the scene

onto the viewing plane and check if rays originate in places where object projections lie.

4.3 Intersection Cost Summary

In this Chapter we have discussed several bounding volume intersection cost, more specifically those considered to be the cheapest. We also discussed some projected bounding "areas" (not volumes). The average intersection costs for all the above are summarized in Figure 4.8.

Bounding Volume	Avg Intersection Cost per ray
Sphere	17.6
3D Bounding Box	27.4
Circle	5.0
2D Bounding Box	4.0

Figure 4.8: The cost of intersecting a ray with the Bounding Volumes examined in this chapter.

It is thus apparent that the 2D box, although looser than the 3D bounding boxes, is cheaper to intersect. Circles, although easy to intersect, are harder to form hierarchical structures with and do not adapt at all to accommodate the object's shape. Instead, they are usually used in a similar way to space subdivision techniques [Hub95].

We must note here that the proposed 2D bounding volumes carry with them an inherent shortcoming. The fact that we discard the $z-$dimension leads us to keep less information about the actual simple object we are enclosing. In the case of the 2D axis aligned boxes even bigger loss of information takes place, since it is less tight than the 2D projected hexagon. In other words by using 2D bounding volumes we affect two parts of ray tracing: we improve the cost of intersecting a bounding volume and worsen the tightness of the bounding volumes, leading to more rays at the lowest level of the ray tracing, the intersection with the actual object (as seen in Figure 2.5).

Chapter 5

Preprocessing Cost Analysis

We will roughly review here the preprocessing cost described in the algorithm for building a hierarchy, as seen in Figure 3.9. We will also attempt to give an estimate of the cost for building and traversing the hierarchies we used in our testing.

5.1 Projection

The equation for calculating the intersection of a ray cast from the camera with parameters $\vec{Location}, \vec{Up}, \vec{Right}, \vec{Dir}$ provides a point P, as seen in Equation 5.1 (same as Equation 4.1), where $dist$ is calculated according to the object hit (as seen in Chapter 4).

$$\mathbf{P} = \mathbf{R}_0 + dist * \mathbf{R}_d \tag{5.1}$$

By essentially inverting the equation we get the projection of point P onto the image plane. So we treat projection as the inverse of ray casting. We know that $\mathbf{R}_0 = \vec{Location}$ and we can express $\mathbf{R}_d$ in terms of the camera and the viewing plane, as in Equation 5.2.

$$\mathbf{R}_d = \vec{Dir} + u * \vec{Up} + v * \vec{Right} \tag{5.2}$$

where (u, v) are the 2D coordinates of the ray onto the image plane. In order to derive u, v

from the given camera parameters and the 3D point P, we combine Equations 5.1 and 5.2.

Thus Equation 5.1 can be transformed into

$$\mathbf{P} = \vec{Location} + dist * (\vec{Dir} + u * \vec{Up} + v * \vec{Right}) \tag{5.3}$$

In Equation 5.3, given that we know the point $\mathbf{P}$ we want to project, we have the following three unknown factors: $dist,\ dist * u$ and $dist * v$, where u, v is essentially the projection of $\mathbf{P}$ onto the image plane. Since we are dealing with vectors, Equation 5.3 can be expressed as:

$$\begin{aligned}
\mathbf{P}.x &= \vec{Location}.x + dist * \vec{Dir}.x + u * \vec{Up}.x + v * \vec{Right}.x \\
\mathbf{P}.y &= \vec{Location}.y + dist * \vec{Dir}.y + u * \vec{Up}.y + v * \vec{Right}.y \\
\mathbf{P}.z &= \vec{Location}.z + dist * \vec{Dir}.z + u * \vec{Up}.z + v * \vec{Right}.z
\end{aligned}$$

from which we can derive u, v.

This projection operation, assuming we have precomputed all parameters related to the viewing plane, includes 6 additions, 11 multiplications and 1 division, so in total has a cost of 21 (as we have defined it in Chapter 4). We must note here that in the actual implementation, a small constant factor to the above cost is added for retrieving the camera parameters.

For a scene with n objects the cost of projection is n times the cost of projecting a bounding volume. As we have already mentioned, the cost of projecting a bounding volume greatly depends on the type of the bounding volume used. As seen in Chapter 4, projecting a sphere is equivalent to projecting two points, the center of the sphere and a vector representing the radius. We must be careful though to choose for the radius a vector that is perpendicular to the camera plane, so as to get an accurate projection of the sphere. On the other hand, projecting a 3D box is equivalent to projecting eight points, the vertices of the box.

As we mentioned in Chapter 4, intersecting the projection of a 3D bounding box takes up a constant time factor, which is nevertheless bigger than for instance intersecting a circle. This

is why we proposed the use of a 2D view aligned box that encloses the projected 3D box. The cost of determining a 2D surrounding box of the projected 3D box is trivial for 3D projected boxes (as well as for circles). For a circle it is basically 4 additions/subtractions $C_c.x \pm C_r$ and $C_c.y \pm C_r$, where C_c, C_r are the projected center and radius respectively. For a 2D box it is a 16 comparison computation, done in order to find the extreme x, y values of the projected 8 vertices.

Summarizing we have:

Bounding Volume	Projection Cost	Enclosing in 2D box cost
Sphere	2*21= 41	4
3D Bounding Box	8*21=168	16

It is apparent that spheres are cheaper to project and to enclose in a 2D box. Nevertheless, as we have mentioned before, spheres are not as adaptive to the actual object they enclose and are thus less tight than 3D boxes. Given the fact that 2D boxes are even less tight than the projection they enclose, 2D boxes enclosing projected spheres are cheaper to produce but less tight than those derived from 3D boxes (Figure 5.1). Given the fact that our goal is not only faster intersections in the hierarchy, but also fewer rays reaching the last level of the hierarchy, we will choose for our testing the 3D axis aligned box, improving the quality of the hierarchy by increasing the preprocessing cost.

So for our choice of axis aligned bounding boxes the total projection cost in the preprocessing step is $n * 8 * 21$ and an additional $n * 16$ for determining the 2D bounding box. In total the precomputing cost is $n * 184$, a constant cost, which will be added on the preprocessing cost to accelerate the intersection costs of bounding boxes for all rays in the scene.

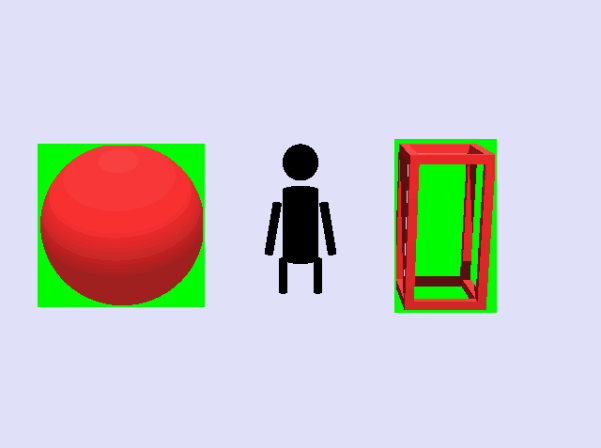


Figure 5.1: A 3D man in the middle and left and right its sphere and 3D axis aligned box bounding volumes respectively. From the green section that represents the 2D box surrounding the projected versions of the bounding volumes, it is apparent that the 2D box derived from the sphere is less tight than that of the 3D box.

5.2 Building

The cost of building a hierarchy also greatly depends on the number of objects represented in the scene, as well as the subdivision or joining criteria (if the hierarchy is built top-down or bottom-up). Refer to section 3.3.3 for a detailed description of the hierarchies used.

When considering a technique using the best join over all objects that minimizes a certain parameter (ex. minimizing volume, or overlapping), every subset of objects must be tested in order to get the best tree. This, in the most naive implementation (exhaustive) yields exponential time over the number of objects in the scene $\mathcal{O}(2^n)$.

By sorting before applying the desired criteria and then splitting or joining and using this specific order (first and last sorted object are always in different groups), we get an $\mathcal{O}(n \log n)$ time operation. This is due to the fact that the sorting takes up $\mathcal{O}(n \log n)$ time and the splitting tests $\mathcal{O}(n)$ time. This is essentially the approach followed by the Top-Down Binary Partitioning.

Keeping all instances (objects or joined objects) sorted by the criterion of splitting, we can

approximate the results of the exponential class in quadratic time ($\mathcal{O}(n^2)$), as it is the case in NN based Bottom-Up Binary Matching.

In incremental algorithms such as the R-tree and the R*-tree that aim to approximate the results of the mentioned exhaustive algorithms, the construction cost is close to $\mathcal{O}(n \log n)$ time ([dBvKOS98], [Gut84]).

The behavior described above can be easily seen from the results of our testing, more specifically in the graphs of the preprocessing time. There we can see the quadratic factor affecting the NN based Bottom-Up Binary Matching and the lower complexity of all the other approaches. In these graphs we also see the effect that building all the projected hierarchies has on the overall preprocessing time (Figure 5.2).

	Building time for 1 hierarchy			Building time for All hierarchies		
	SF = 6	SF = 9	SF = 11	SF = 6	SF = 9	SF = 11
T-D Bin Partitioning	0.01	0.19	1.11	0.01	0.19	1.11
TDBP translated 3D	0.001	0.09	0.33	0.071	0.65	3.16
R-tree	0.01	0.08	0.34	0.01	0.08	0.34
R-tree translated 3D	0.01	0.07	0.33	0.07	0.54	2.23
R-tree 2D criteria	0.01	0.12	0.42	0.06	0.59	2.34
NN 2D criteria	0.1	7.22	207.6	0.27	26.12	770.85

Figure 5.2: The complexity of different hierarchical techniques and the effect of building many hierarchies for the projected approaches. These time statistics are taken from the “tree” scene (Figure 6.1) for size factors 6, 9 and 11, when the floor is split (Section 6.3.6).

All the above operations are done over all 3 dimensions when creating a 3D independent hierarchy (that is all criteria must be checked and evaluated in all dimensions). Thus, our view dependent hierarchies, which are based on 2D criteria, will be of the same complexity, but considerably faster, since we are eliminating tests on one dimension, as it is seen in Figure 5.2.

5.3 Performance

Based on the work done in [GS87], one can determine the expected number of ray intersections on a given hierarchy, based on the number of children each node has, and the relative area it covers compared to the root node.

In [AK89] the approach of determining the performance of a bounding volume hierarchy is viewed as an attempt to see how a bounding volume affects the distribution of rays for its children volumes. Following the notation of [AK89], we denote as EC the external cost of a bounding volume, i.e. the fixed cost of a ray intersection test with the volume, and as IC the internal cost, i.e. the average cost of a ray intersection test with the contents of the volume, given that the ray did hit the volume.

So for a bounding volume A enclosing $B_1, B_2, ...B_n$ children nodes we have:

$$IC(A) = \sum_{i=1}^{n} \left\{ EC(B_i) + \frac{< P(B_i, d) >}{< P(A, d) >} * IC(B_i) \right\} \tag{5.4}$$

where $< P(V, d) >$ is the average of the projected areas of V along directions d. The above applied recursively can give an average cost of intersecting a ray with a given bounding volume hierarchy.

The above method for predicting the performance of a hierarchy is nevertheless an approximation of the actual situation. The resulting quality measure reflects the actual hierarchy performance only in cases where the probabilities of intersecting any child of a node are independent. If for instance one of the children in a node is bigger than the rest, this is not true. So the approach is actually only an accurate approximation when bounding boxes are much smaller than their parents.

That is why, in order to determine the quality of the hierarchies used in our testing, we will actually count their performance at run time. The most important measure to demonstrate the performance (and moreover the quality of a hierarchy) is the number of objects tested for intersection by a ray. In an ideal hierarchy a ray checks one object or none and in a scene with

no hierarchy n objects are checked by every ray (where n is the total number of objects in the scene). So the average number of objects hit by a ray is a good indication of the quality of the hierarchy.

In our results we will evaluate this measure only for the visibility rays, since all the other rays also use hierarchies of the same quality.

5.4 Traversing

Our hierarchies represent a given scene as a tree structure. Assuming a tree is balanced and no overlapping nodes occur (optimal), the average traversal cost is $\mathcal{O}(\log n)$, whereas in the worst case, without these assumptions, it is $\mathcal{O}(n)$. Of course object hierarchies are not always balanced (our Nearest-Neighbor-like algorithm yields unbalanced trees) or non-overlapping (all the hierarchies used may have object bounding boxes, or more likely bounding boxes from joining, that overlap). Thus the average traversal time increases, assuming that all paths to a leaf are equally probable. This of course is not always the case (we can have unbalanced optimal trees [AK89]), but it is almost always applicable in the case of approximation hierarchical algorithms.

The traversal cost deteriorates with the use of a 2D hierarchy (either built on 2D criteria, or by projecting an existing view independent hierarchy) as we have implemented it. This is a result of the fact that we eliminate tests on one dimension (the z coordinate). In other words a line query in a 3D hierarchy is not as fast and as well represented as a point query in a 2D hierarchy.

The above claim can be seen when comparing the average cost of hitting an intermediate object in a hierarchy, as seen in Figure 5.3.

It is apparent that traversing a 2D hierarchy is much cheaper if we use a projection hierarchy when the scenes are fairly complicated. When scenes are small (like the “tree” scene of size factor 6), the benefits of traversing the hierarchy fast are overshadowed by the fact that the 2D

	Traversing Cost for Intermediate objects (FLOPS)		
	SF = 6	SF = 9	SF = 11
T-D Bin Partitioning	1,069,180	1,450,650	6,921
TDBP projected 3D	421,547	14,158	4,056
R-tree	2,181,550	1,547,200	184,897
R-tree projected 3D	131,478	81,430	10,632
R-tree 2D criteria	1,162,410	106,928	1,322
NN 2D criteria	203,590	52,764	10,468

Figure 5.3: These cost statistics are again taken from the “tree” scene for size factors 6, 9 and 11. It is interesting to note how much cheaper are the intersection costs in the cases of the projected hierarchies compared to their view independent counterparts. In this scene the floor is viewed as a set of smaller parts, in order to avoid any artifacts caused by the big size of the floor object.

bounding boxes are loose and several rays reach lower levels of the hierarchy needlessly. This is also the reason behind the bigger traversal cost in the smallest scenes.

We must mention here that in the way we have implemented the projected hierarchies, more than one structure is usually needed for a single scene. Thus we introduce an extra cost in the ray tracing procedure associated with the choice of structure to be used. For example visibility rays use the visibility hierarchy, shadow rays the light hierarchy and reflection/refraction rays a view independent hierarchy. As we will show in our results, this decision in our approach greatly affects the resulting performance, especially in scenes that require many hierarchies (such as scenes with a large number of lights). This cost of choosing a hierarchy is a constant factor per ray, so we could say it is in a way scene-size independent. Moreover, as the number of objects in a scene becomes larger, the cost of intersecting the objects and traversing the hierarchy overshadows that of the choosing a hierarchy, since the number of rays that are newly introduced is relatively small. The above is apparent in our results, because when scenes

get larger, the projected hierarchies time for ray tracing converges the same way the view independent hierarchies time does, implying that after a point the cost of choosing a hierarchy is no longer significant.

Finally, we need to also associate another cost with the traversing process in the 2D hierarchies. Since we accelerate ray intersection on the 2D boxes by performing a point-in-rectangle check, we must make sure that the origin of the ray is indeed on the plane of the hierarchy. This is always true for visibility rays, but not so for shadow rays. So each shadow ray needs first to be projected on the plane of the light hierarchy. This cost is small and is performed once per hierarchy (for each ray). Nevertheless it adds to the overall ray tracing time. Its effects are included in the choice process, since the projection of the ray needs to be done prior to the choice of hierarchy.

5.5 Summary

As we have discussed in this chapter the preprocessing cost for building a 2D hierarchy is very small compared to building an independent hierarchy, given the fact that although independent hierarchies do not include projection cost, they have to incorporate the cost of considering an extra dimension. Nevertheless, this cost becomes a big factor when many lights are introduced in the scene, due to the fact that when using a view dependent approach many hierarchies must be built in order to accommodate shadow rays and other secondary rays.

The cost of traversing a 2D hierarchy is quite small compared to traversing a 3D hierarchy, specifically in the case of large scenes. The benefit from this fast traversal nevertheless is weakened by the fact that a cost in choosing the appropriate hierarchy for each ray is introduced.

Chapter 6

Testing and Results

As promised, we will demonstrate the performance of our approach testing several scenes. In order to provide testing results that are generally accepted and can be easily reproduced by any interested party, we will use test scenes from the SPD set [Hai87], as well as several other scenes following the SPD specifications.

Our aim in this section is firstly to demonstrate how the technique works in general compared to the conventional view independent hierarchy approach, and secondly to examine how its performance scales when the scene becomes more complicated and lights are added.

6.1 Comparison Metrics Used

Several metrics will be calculated, so as to give a detailed and meaningful description of the behavior of the hierarchy types examined. These metrics are:

- **Time**

  We will give two measures of time. The first, T_P, is the Preprocessing Time for the scene. This time essentially covers the hierarchy building. In the view independent hierarchies this time is equal to building the one needed hierarchy. In the project hierarchies we can view this time as a combination of the time spent to build the visibility hierarchy (T_V),

the hierarchies for the lights (T_L), as well as the 3D hierarchy needed for reflection and refraction (T_G).

The other time metric that will use for both view independent and dependent hierarchies, T_R, is the total time spent on ray tracing the scene, after the preprocessing step.

T_P and T_R are two distinct and non-overlapping metrics and they are expressed as the CPU time in seconds. Although several other metrics are also counted, time is the one we will focus on in our explanations and performance graphs.

- **The Cost of All tests Per Hierarchy Object (traversal cost)**

  The Cost of All tests in a Hierarchy (C_{AH}) expresses the number of flops (as we defined them on chapter 5) used to traverse the hierarchy, and Cost of all tests per Hierarchy Object (C_{HO}) indicates the average cost of traversing one hierarchy object (not actual scene object).

  These cost measures give a very clear idea of the calculations done when traversing a hierarchy and greatly affect the ray tracing time. This is a measure that the use of 2D hierarchies aims at minimizing.

  In the case of the 2D Hierarchies, these two measures will express the sum of costs for visibility and light hierarchies. In the 3D hierarchies the cost will also be calculated for primary (visibility) and shadow rays. The reason why we discard secondary rays is that the 2D version of a hierarchy uses a 3D view independent hierarchy, which is the same as the 3D version of the hierarchy. Since the hierarchies used by both approaches for secondary rays are the same, and given the fact that the number of secondary rays cast are the same (depends on the number of successful intersections in the previous level, which is independent of hierarchy), the cost will be the same and thus we do not take it into account.

  In our graphs we will present the Cost per Hierarchy Object (in other words the average cost of intersecting an intermediate object of the hierarchy). We chose this of the two

cost metrics, since it is more intuitive.

- **The Cost of all tests Per actual Object**

  As the above measures (C_{HA}, C_{HO}), these two costs express the calculation expense of intersecting an object at the last level of the hierarchy (all calculations C_{OA}, as well as an average cost per object C_O). Of course this cost is always calculated in a 3D approach and it demonstrates how accurate a hierarchy representation is. If the cost is small, a small number of rays reached the last level. Again, we do not count secondary rays' cost, because we want to show how the 2D approach affected the last level intersections. It is only natural that since the 2D bounding boxes are not as "tight" as the equivalent 3D, the cost will be higher. We want to see if this higher cost, together with the higher preprocessing time can be offset by the gains from traversing the hierarchy faster.

  These two measures, as well as the above, do not aim to test which hierarchy algorithm works better (this can be seen from the Number of Rays Tested). Rather, they aim to compare the 2D and the 3D version of the same algorithm.

  Again we will display only the Cost per actual Object, in other words the average cost for intersecting an object of the scene.

- **The average number of objects tested by a ray**

  As we mentioned in chapter 5, in an ideal hierarchy each ray would test one or no objects. In a scene with no hierarchical structure a ray would be tested against each object to determine the closest intersection (if any). In order to provide a measure of the quality of our hierarchies we will thus compute the average number of objects hit by a ray. The closer this number is to 1 the better the quality. This test will be performed on visibility rays only, since they use one hierarchy in all cases. But the results can be viewed as accurate for all hierarchies in a given scene, since they all follow the same heuristics.

For a fast and intuitive review of the approaches we will give here information concerning time, object cost and average number of objects hit by a ray displayed in graphs. We focus on

these measures since they are good indicators of the performance and quality of the hierarchy schemes that we compare.

Each scene is run for several size factors (SF). As far as hierarchical building is concerned we will try 2D and 3D hierarchical schemes. Moreover, for all the hierarchies we will examine several combinations of ray-intersection schemes. We will try 2D intersection tests when traversing the hierarchy, 3D intersection test and finally a 2D first pass for intersection and then a 3D pass. We believe that these combinations of hierarchies and testing approaches will provide us a general and accurate image of the aspects of each approach.

So we have the following graphs for each scene:

- Using 2D intersection scheme in 3D and 2D hierarchies
    - Ray tracing Time vs Size Factor
    - Preprocessing Time vs Size Factor
    - Average number of objects hit by a ray vs Size Factor
    - Average cost for hitting an object vs Size Factor
    - Average cost for traversing an intermediate object vs Size Factor
- We will also refer to 3D intersection scheme in 3D and 2D hierarchies, as well as a combination of 2D and 3D testing scheme in 2D and 3D hierarchies for the same graphs.

Because the size factor of the scenes is related in a different way to the actual number of objects in each scene, in some cases we will normalize the measures, in order to better compare the performance of the algorithms.

6.2 Test Scenes

6.2.1 Tree

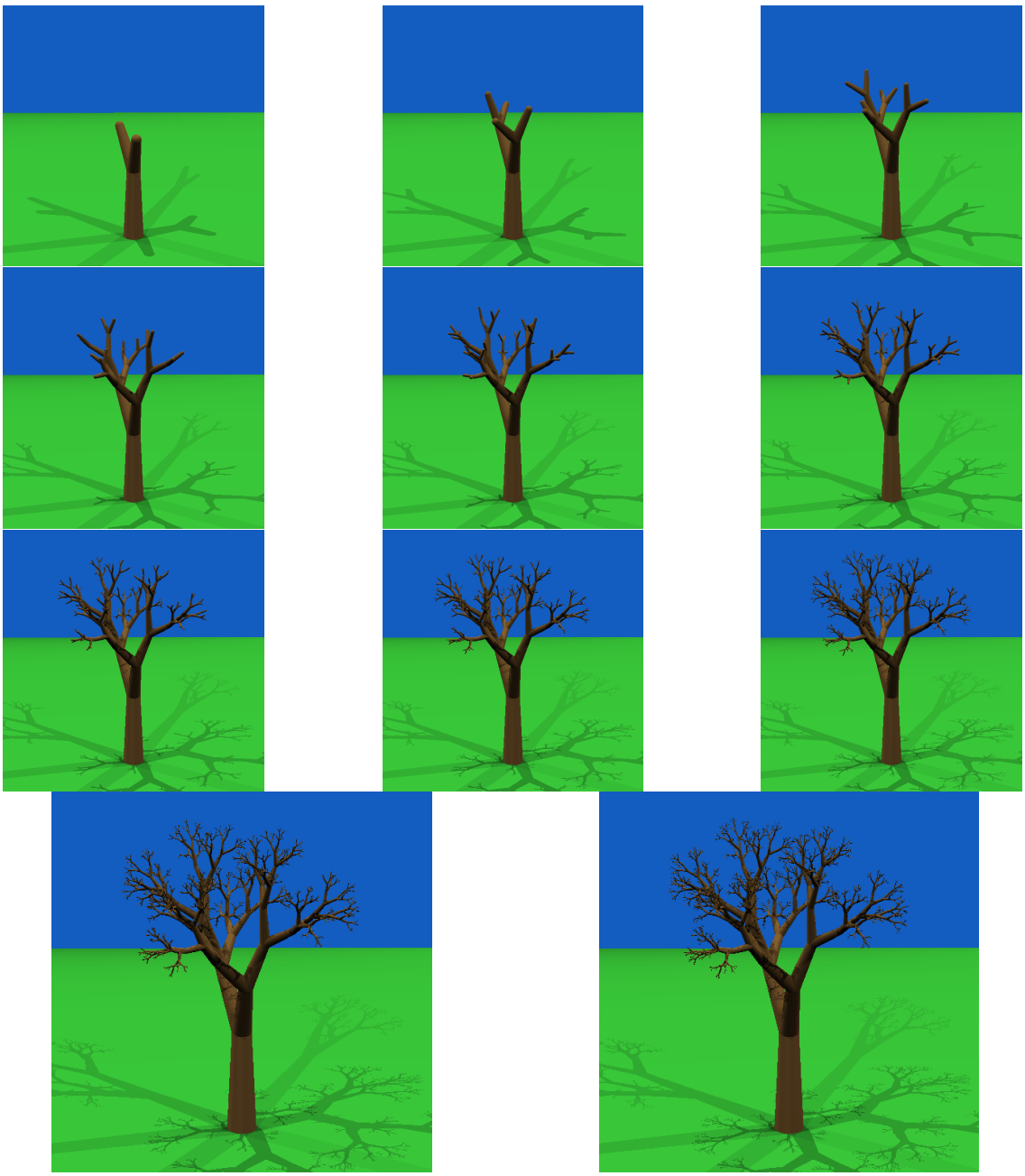


Figure 6.1: Tree scene from SF 1 to 11

This scene is indeed very challenging, for the projected hierarchies since it includes 7

lights inside the scene volume and indeed for all hierarchies since it displays a highly irregular distribution of the objects (Fig. 6.1).

The scene is run under 11 sizes (size factors or SF) containing $2^{(SF+1)}-1$ cones and spheres + 2 triangles and 7 lights. In other words the number of objects ranges between 5 and 4097. All surfaces of the particular scene are matte. This facilitates the comparison between the different approaches, since there are no reflection or refraction rays to dominate the number of rays shot in the scene.

6.2.2 Sphereflakes

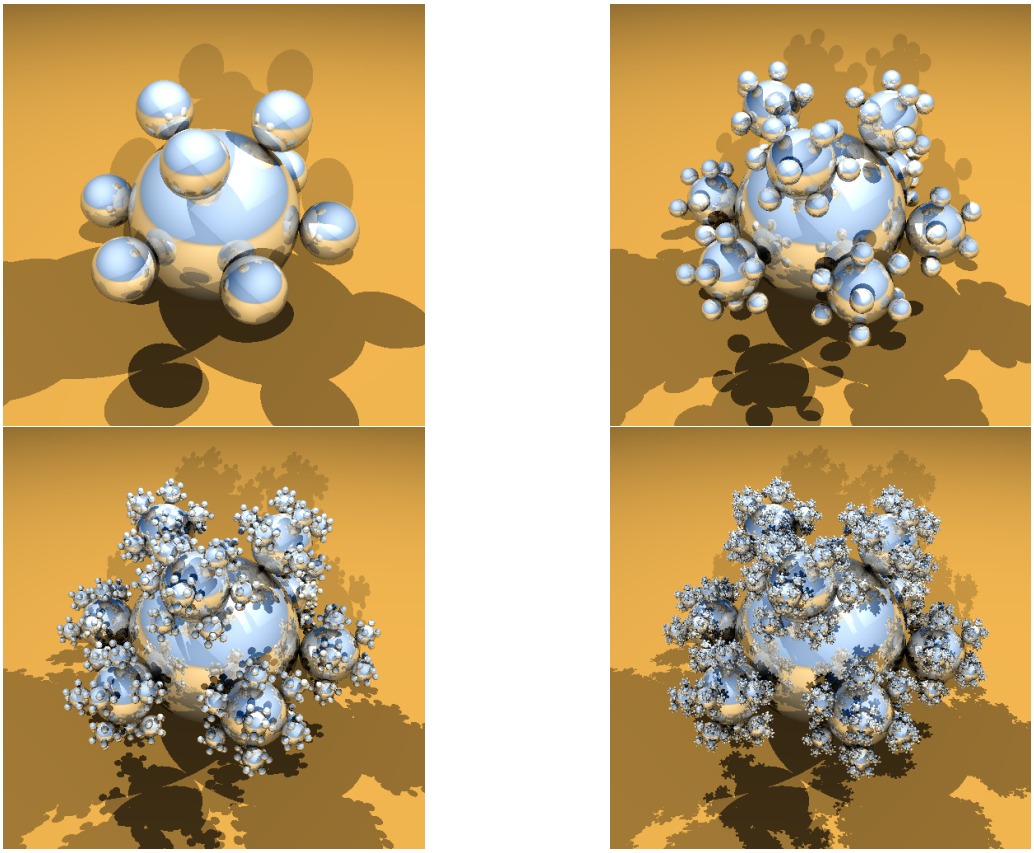


Figure 6.2: Sphereflakes scene from size 1 to 4

One of the most common scenes from the SPD library is the “sphereflakes” scene (Figure 6.2). The scene is a set of shiny spheres, with each sphere blooming a set of 9 more spheres

with 1/3rd radius. Two matte triangles for a floor are also added. There are three light sources. The size factor of the scene determines the number of objects outputed. The total number of spheres is $\sum_{sf=0}^{SF} 9^{sf} = \frac{9^{(SF+1)}-1}{8}$, plus the two triangles.

We will test the scene for four discrete SF and get scenes with 10, 91, 820 or 7381 reflective spheres. For our testing we will use all 4 sizes of the scene, beginning from the small scene and gradually increasing the size, so as to see the effects of bigger and more complex scenes on the hierarchical algorithms and on the new 2D approach. We expect that bigger number of objects will add a bigger overhead of preprocessing time in the 2D approaches, and we want to see if this overhead is small compared to the benefits of fast hierarchy traversal of the 2D approach.

6.2.3 Gears

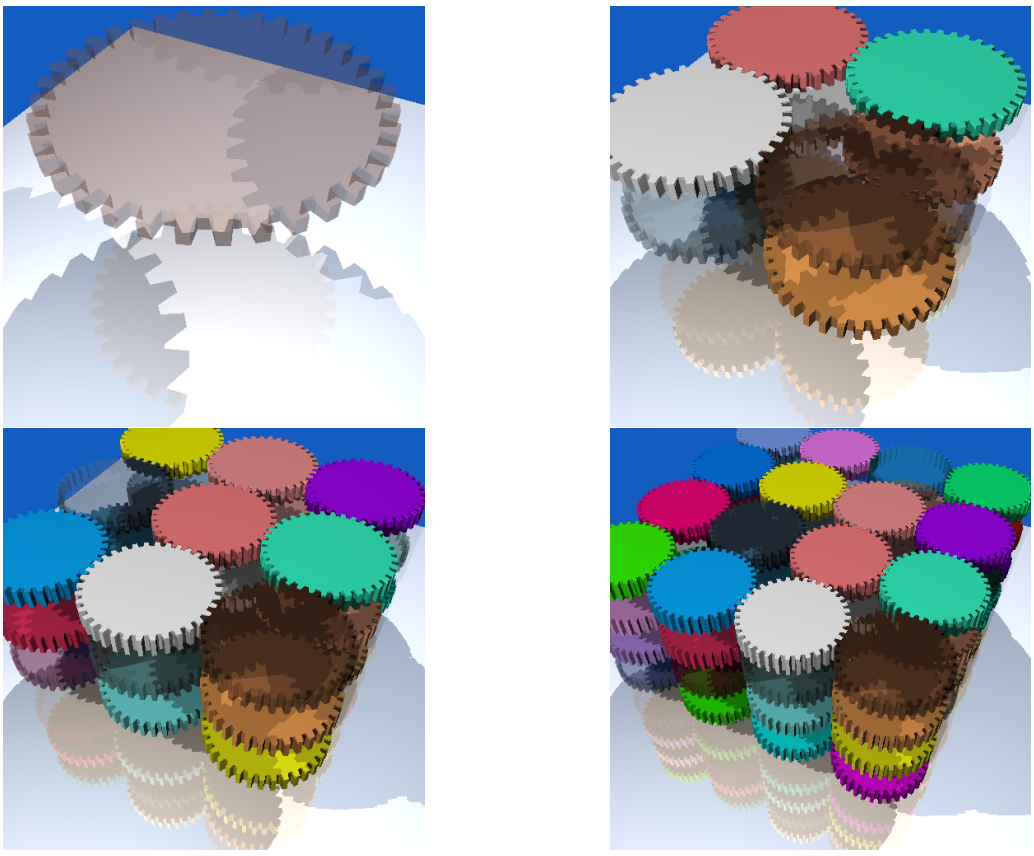


Figure 6.3: Gears scene from size 1 to 4

The final scene we will test is the gears scene (Figure 6.3). This is probably the most challenging of the three types of scenes since it exhibits reflection and refraction. Two reflective triangles are used for the floor plus the overall $4 \times SF^3$ triangles for the gears themselves.

6.3 Results and Observations

Apart from graphs demonstrating the actual values of the metrics described in Section 6.1, we will also give some relative values as described below.

Since the size factor of the Tree scene is the $\log_2(n)$, where n is the number of objects in the scene, it is not intuitive how the image scales in respect to the number of objects. That is why we believe that measures that are linked to the size of the scene, such as ray tracing time, preprocessing time, intersection cost per object, and traversal cost, should also be expressed in a $\log_2$ scale, in order to derive the growth rate of the various processes. The same applies to the Sphereflakes scene, although here we use the $\log_{9/8}$ scale. Finally, in the Gears scene the number of objects is related to the SF via the expression $4 * SF^3$. That is why in the Gears scene we will give the same graphs in a $\sqrt[3]{}$ scale. We believe that the average growth rate (the slop of our graphs) will be a good measure for comparing the behavior of the tested scenes in terms of the actual number of objects present.

6.3.1 View Independent Hierarchies

We will attempt here to explain the behavior of the several hierarchical approaches, as they are applied to the test scenes mentioned above.

The first thing to observe is that from the three view independent algorithm's performance (Top-Down Binary, R-tree and R*-tree), the Top-Down Binary Splitting (TDBS) is the fastest regardless of the number of objects and the scene characteristics (Figure 6.4). Both the R-tree and the Top-Down Binary Splitting aim at minimizing the area (volume) covered by the levels of the hierarchy. On the other hand, the R*-tree minimizes overlap between bounding volumes.

The R-tree variants are built by inserting objects one at a time, whereas the TDBS algorithm assumes knowledge of the scene, and a sort according to the midpoints of the bounding volume is applied prior to the hierarchy construction. Since the TDBS outperforms the other

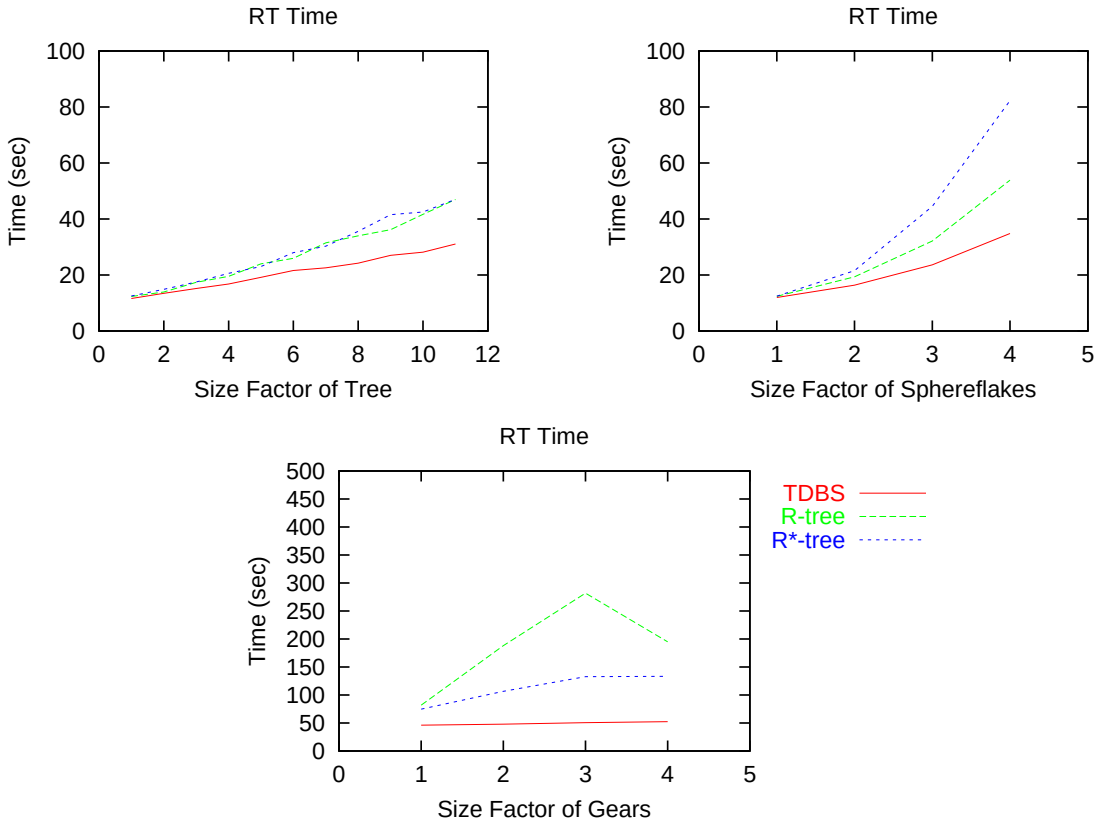


Figure 6.4: Times for view independent hierarchies for all scenes

two hierarchies, we observe that the order of inserting the objects in the R-tree and R*-tree algorithms affects their quality greatly. So the step of sorting the objects by their bounding volume center according to each dimension in the Top-Down Binary Splitting provides a more balanced hierarchy than the unordered insertion as it is done in the R-tree variants. Nevertheless, this “a priori” knowledge of the scene, as it is used by the Top-Down Binary Splitting, results in a non-dynamic hierarchy that cannot be cheaply reused.

6.3.1.1 As far as the the two R-tree variations are concerned in terms of time, we observe that they behave differently according to each scene. As we have mentioned the R-tree criterion for joining objects (bounding boxes) is the minimum resulting area, whereas the R*-tree aims at minimizing the overlap between bounding volumes. These two criteria seem to converge in terms of time in the tree scene (the two global hierarchies have almost identical behavior

for all sizes of the tree). This behavior is observed only in the tree scene. Each branch of the tree is formed by a cylinder and a sphere and from the sphere sprout another two cylinders. It seems that in this particular arrangement the minimum overlap as well as the minimum average area are both achieved by joining the cylinder with the equivalent sphere. That is why the two hierarchies that come from the R-tree and the R*-tree are very similar and thus yield a similar performance. These two criteria work very differently in the other two scenes.

6.3.1.2 In the gears scene the R*-tree algorithm joins together the small “teeth” of a gear with its main body, since this yields the minimum overlap of bounding volumes. On the other hand, the R-tree tends to join the teeth of each gear with those of the gear bellow or above, since that yields a small increase in the volume of the bounding boxes. The result is that the R*-tree tree better models each gear as an entity and provides a more balanced tree in the upper level of the hierarchy.

6.3.1.3 The above situation is reversed in the balls scene. The main reason behind the better performance of the R-tree is the fact that the actual spheres do not overlap, but their bounding volumes do, and it is the bounding volumes which drive the hierarchy building algorithms. So the R*-tree tends to join spheres from two or more levels apart, since their bounding boxes overlap and the spheres of the upper levels are smaller and thus the overlap is smaller. This problem does not appear when using a volume measure to build the hierarchy (like the R-tree does). So in the balls scene, the R*-tree results in bigger bounding volumes in the intermediate levels of the hierarchy, slowing down the ray tracing procedure.

From the above we can derive that pre-sorting the objects really improves the quality of the hierarchy, but it is not always desired since it implies a static scene (no insertion or deletion without rebuilding the structure). When using one of the two R-tree variants we must keep in mind that scenes with complex objects that are constructed with overlapping smaller parts are favored by the R*-tree approach, whereas scenes with limited overlapping perform better with

the R-tree approach.

It is also interesting to mention that the average number of simple objects hit by a ray, as well as the cost for each object is almost the same in all these global view independent hierarchies. This means that the number of rays that reach the objects is almost the same in all the hierarchies. What is different is the traversing of the produced hierarchies (how soon rays are pruned, what is the cost of traversing each intermediate node, etc) and how closely they model the scene (later seen in Figures 6.15 and 6.17).

By examining the normalized ray tracing times of the View Independent Hierarchies (Figure 6.5), we observe that the gears scene is the one demonstrating the smallest growth ratio for all hierarchical types. In other words the time rising per added object in the scene is small. This implies that the reflection and refraction tests that take place in the gears scene combined with the transparent layers of the gears make the cost of adding a new layer quite small (compared to the rest of the calculations). Moreover, the fact that we do not use a threshold for pruning secondary rays, rather calculate rays until they exit the scene, causes in part the non-linear behavior of the gears scene (more objects ensure faster intersection of rays and objects, thus earlier pruning). On the other hand, in the Spherflakes scene adding a new sphere introduces new intersections and new reflection rays that have to be computed. Finally, in the Tree scene adding objects translated directly to more intersections with objects and thus the growth ratio is smaller than that of the balls scene, but greater than the gears scene.

6.3.2 View Dependent Hierarchies built on 2D criteria

Unlike their view independent version, the R-tree and R*-tree projected hierarchies perform quite similarly in respect to time in all the given scenes and sizes, but still the R-tree projected version is a bit better than the R*-tree one (as seen in Figure 6.6 where projected hierarchies are labeled “*name* P”). This is to be expected in fact. Since we are not taking into account the third dimension, the R*-tree aiming at minimizing 2D overlap, often joins together bounding

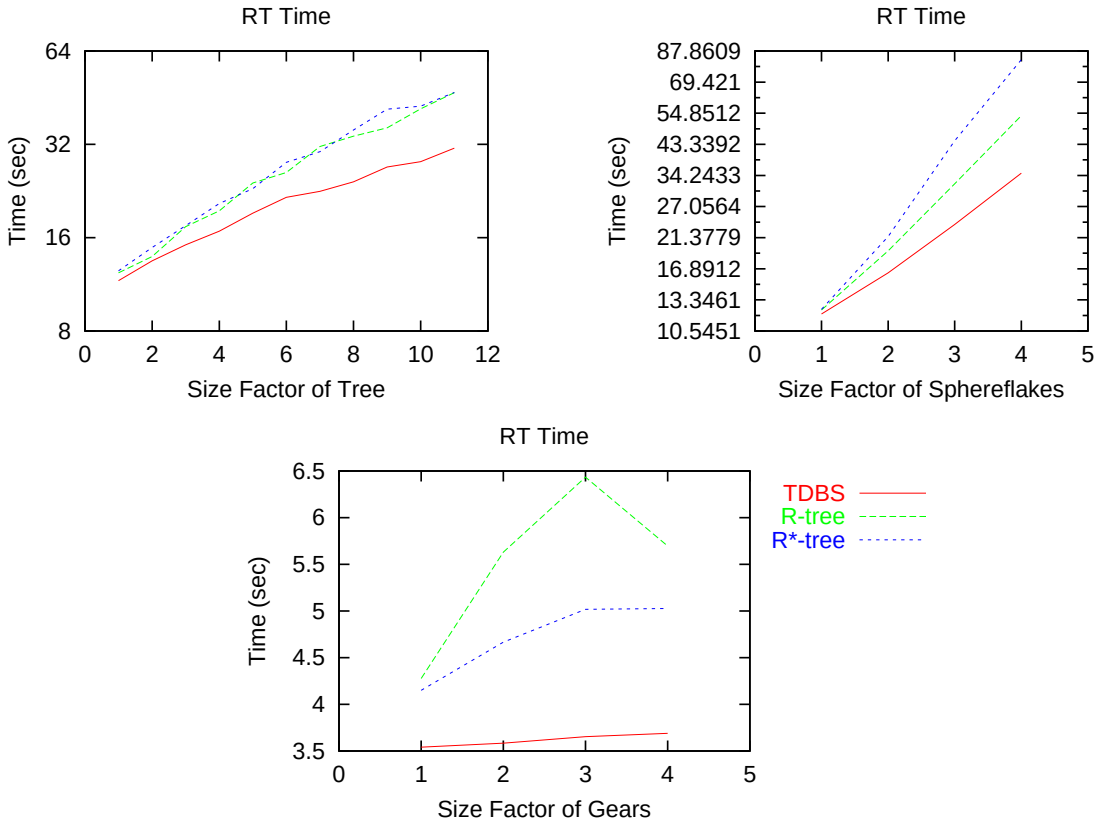


Figure 6.5: Normalized ray tracing times for view independent hierarchies for all scenes

volumes that result in more empty space in the interior nodes. This could lead us to expect a much worse performance by the R*-tree compared to that of the R-tree. This is not the case because the fact that there is small overlap prunes rays earlier into the hierarchy and forces them to chose fewer possible paths.

These two projected (2D criteria) approaches are dynamically updated (as can their respective view independent hierarchies), but the NN projected hierarchy outperforms them in terms of time in all cases (even though it is not dynamic). This indicates that the traversal cost of a 2D hierarchy (which is deeper and wider in the NN case) is insignificant compared to the benefits of a hierarchy that closely represents the scene. The above is true for all intersection types, which indicates that the use of a particular intersection scheme affects all the projected hierarchies similarly.

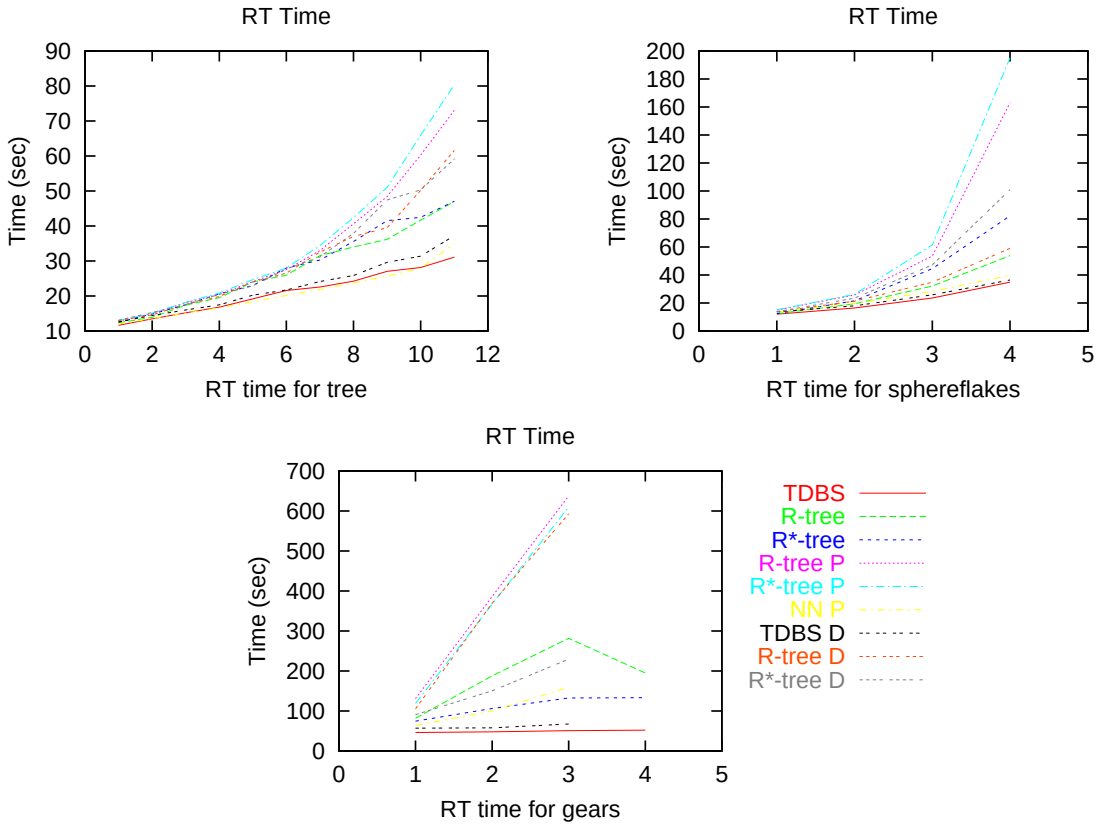


Figure 6.6: Ray tracing times for all hierarchies and scenes

6.3.3 View Dependent Hierarchies built on 3D criteria

These hierarchies, also referred as “dummies”, are original 3D hierarchies that are then projected to the viewing and lights planes. As do their view independent counterparts, the Top-Down Binary Splitting one is always the best in terms of time, and the R*-tree and R-tree ones follow the same principles and behavior as their independent counterparts in all the scenes.

It is interesting to observe that the dummy hierarchies always outperform their counterparts that use 2D building criteria. This is an indication that the third dimension should be taken into consideration in order to model a scene more closely (as seen in Figure 6.6, where dummy hierarchies are referred to as “*name* D”).

6.3.4 Intersection Schemes in View Dependent Hierarchies

The intersection schemes tested favor different types of scenes and the rendering cost depends in part on the camera and light positions. Intersections and traversal through the hierarchical structure is obviously faster when using a 2D test. Nevertheless, when the 2D boxes model the objects of the scene poorly, then a combination test filters the number of rays the reach the object level, adding an overhead at the traversal process. In all the examined cases the use of a 3D test fails to give good results, since there is no actual gain in the traversal time.

6.3.4.1 The 2D intersection scheme is better for both the 2D and dummy hierarchies in the Tree scene, the combination test is best for all projected hierarchies in the gear scene and in the balls scene the combination scheme is better (Figure 6.8). The reason the 2D test is better in the Tree scene lies in the fact that the angle of lights and of the viewing plane is such that when joining 2D boxes a tighter “area” is produced than by projecting the 3D box of the joined 3D boxes (Figure 6.7). Thus a smaller number of rays reaches the lowest level of the hierarchy with a low cost. When the 3D test is added it only contributes into the traversing time, thus rendering the ray tracing slower.

6.3.4.2 In the gears scene the angle of the camera and the lights are in positions that produce loose 2D boxes in the lowest level. This is propagated when joining the 2D boxes together. A big percentage of the area in the 2D boxes across the hierarchy is empty and so more rays reach the lowest level. On the other hand, when a 3D test is added, it refines the number of rays by also testing the tighter 3D boxes.

When it comes to choosing an intersection scheme the angle of the lights or the viewing plane is not always obvious. But as a criterion a large degree of 2D overlapping (hidden objects) amplifies the effects of loose 2D boxes (and the combination scheme should be used),

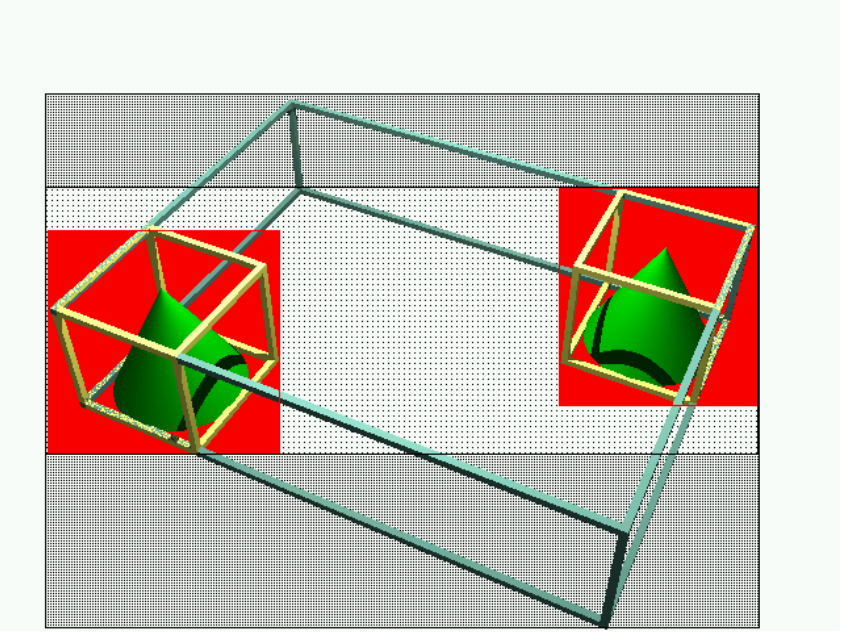


Figure 6.7: A simple case where the joining of two 2D boxes is better than projecting the 3D bounding volume derived from the joining of their 3D counterparts.

whereas in scenes with small 2D overlapping a 2D intersection scheme is faster and is affected little by the looseness of the 2D boxes.

6.3.5 Overall Time Comparisons

From the time data we have gathered it seems that in the scenes where secondary rays dominate in number (balls, gears), the projected (2D and dummy) hierarchies come very close but never quite outperform the view independent hierarchies (Figure 6.9). This leads us to believe that the cost of selecting a hierarchy is in fact the biggest slowdown of our method. This choice cost also affects the tree scene since there are 7 lights (7*6 = 42 hierarchies), but the impact is lessened because the number of rays that need to chose hierarchies is smaller.

In order to prove our point we tested the Tree and the balls scene with a single light. The

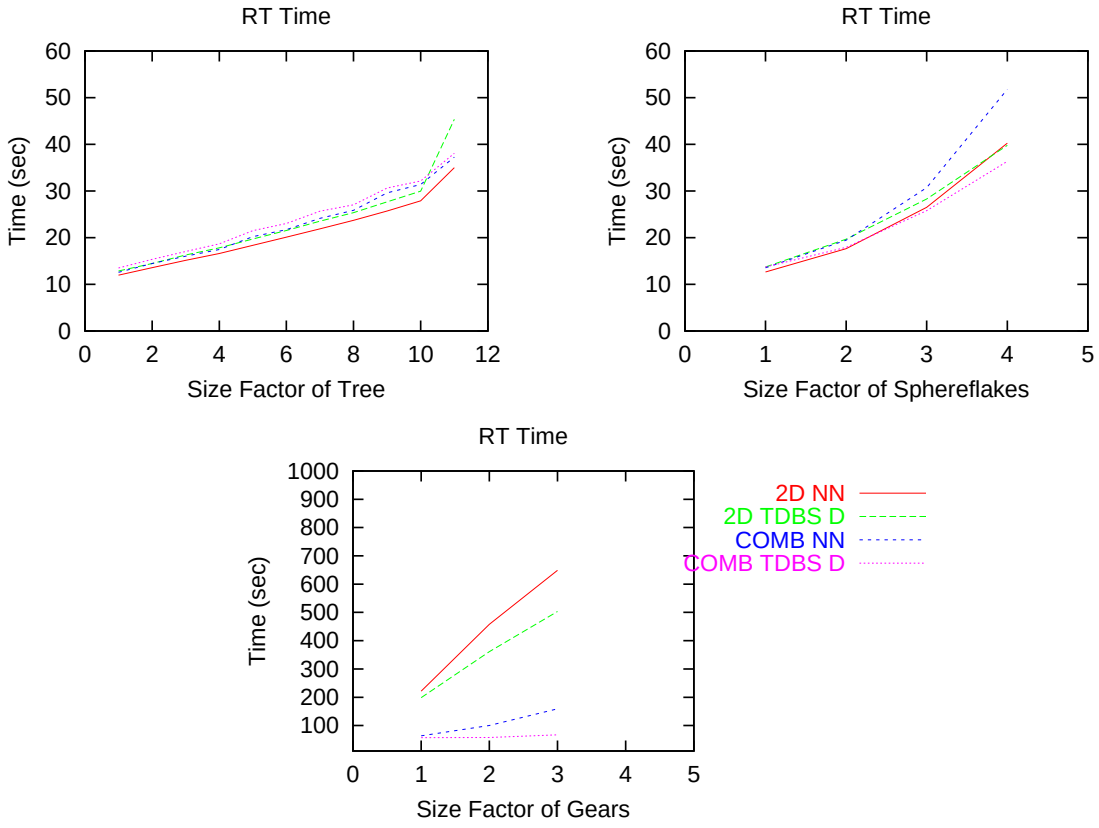


Figure 6.8: Some of the times using different intersection schemes in projected hierarchies

results show (Figure 6.10) that indeed the projected hierarchies improve their performance and are quite faster than the view independent one in the tree scene and the difference is amplified as the scene gets larger. The projection hierarchies are almost as fast in the balls scene, but because there are reflection and refraction rays as well, the overhead of choosing a hierarchy is more apparent than in the tree scene.

Although it is apparent that less lights reduce the cost of choosing a hierarchy, it is not immediately clear if the cost mentioned above comes from determining the light of origin for a shadow ray or from identifying one of the 6 hierarchies associated with the light. This is why, as we mentioned in Chapter 3, we also tested our scenes with shadow rays that have "knowledge" of the light they originated from. In these cases we eliminated the choice of light of origin (Section 3.2.1). Our results show (Figure 6.11) that indeed the projected versions are

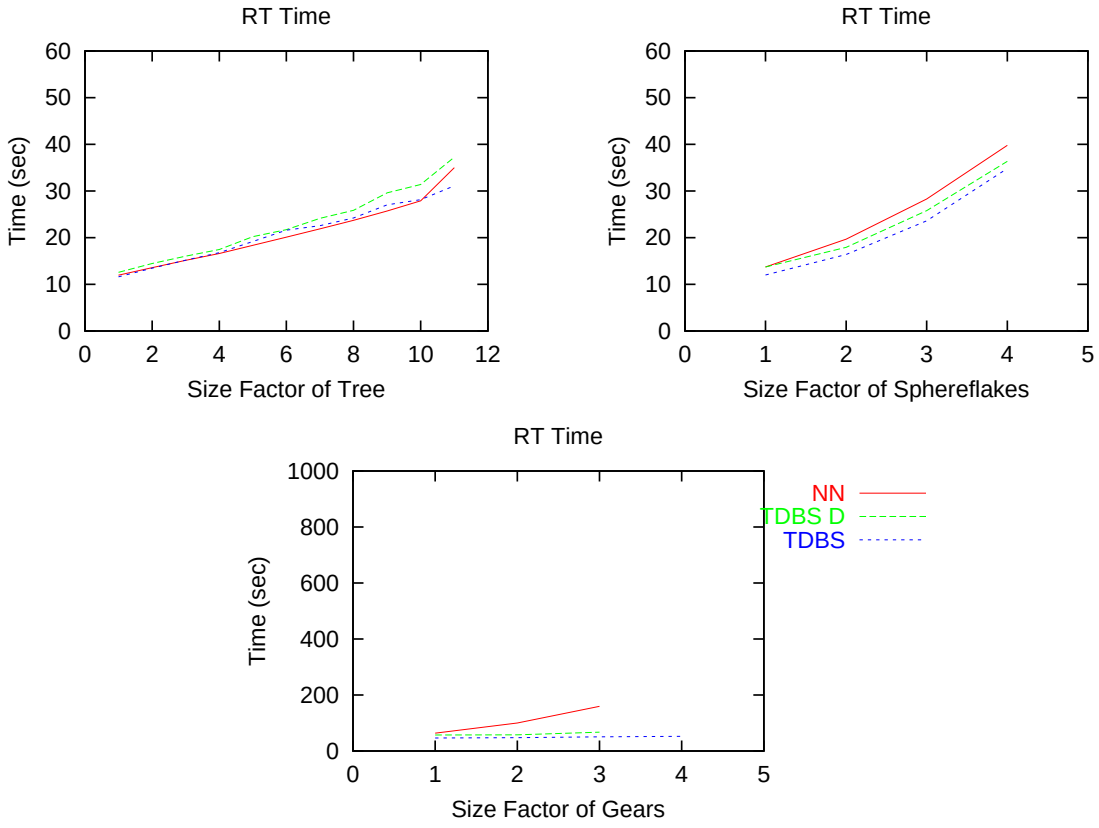


Figure 6.9: Some of the times for all types of hierarchies

accelerated, but not greatly. So we have come to the conclusion that choosing the hierarchy inside each light (one of 6 possible hierarchies) is costing more than determining the light of origin.

The above lead us to conclude that projection hierarchies can indeed improve the time complexity of ray tracing in complex scenes, provided that the number of secondary rays that need to be tested against the hierarchies and the number of hierarchies themselves is low.

6.3.6 Other Observations

Number of objects hit by ray: In Section 6.1 we mentioned that the number of objects hit by a ray gives a good indication of the quality of a hierarchy. As seen in Figure 6.12, the average number of objects hit in the view independent hierarchies is less than in all the projected

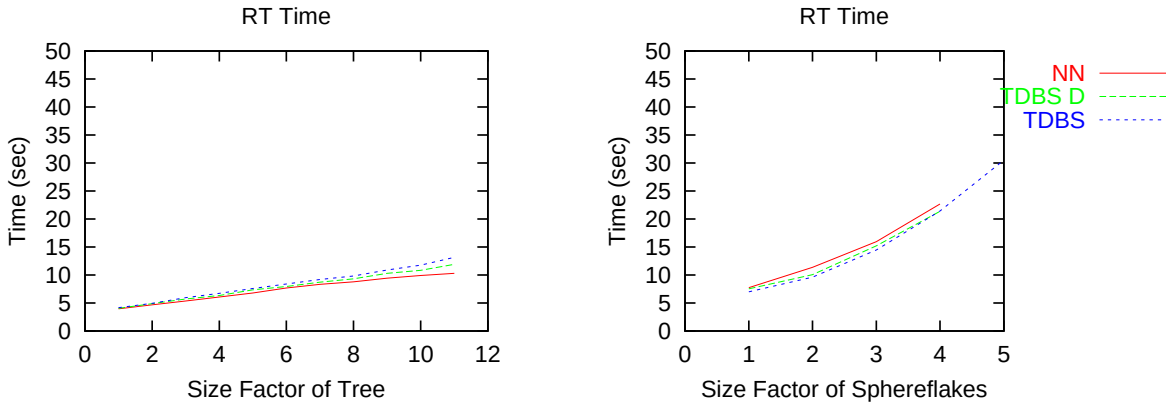


Figure 6.10: Some of the times for all types of hierarchies with less lights

approaches. As we suspected, view independent approaches model the scene more tightly. We observe that the different types of hierarchies tend to teem together: view independent ones, projected ones based on 2D criteria, projected based on 3D criteria. The similarities observed between the hierarchical approaches is a good indication that although the hierarchical algorithms of each approach are different, the fact that the criteria used are similar (2D or 3D) yields similar qualities. In this testing 2D intersection scheme is used for projection hierarchies in order to give an accurate image of the tightness of the hierarchies; the combination scheme tends to prune more rays and distort the results. This 2D intersection scheme is accountable for the fact that the projected hierarchies based on 3D criteria are less tight than the actual projected ones.

The peculiar behavior of all hierarchies around small SF values can be partially viewed as artifact of small scenes, where the cost of using a hierarchy can degrade the ray tracing procedure.

Preprocessing time: As we have mentioned in Chapter 5 the preprocessing cost of the approaches tested varies 6.13.

As far as the hierarchical approach used, view independent hierarchies have smaller pre-

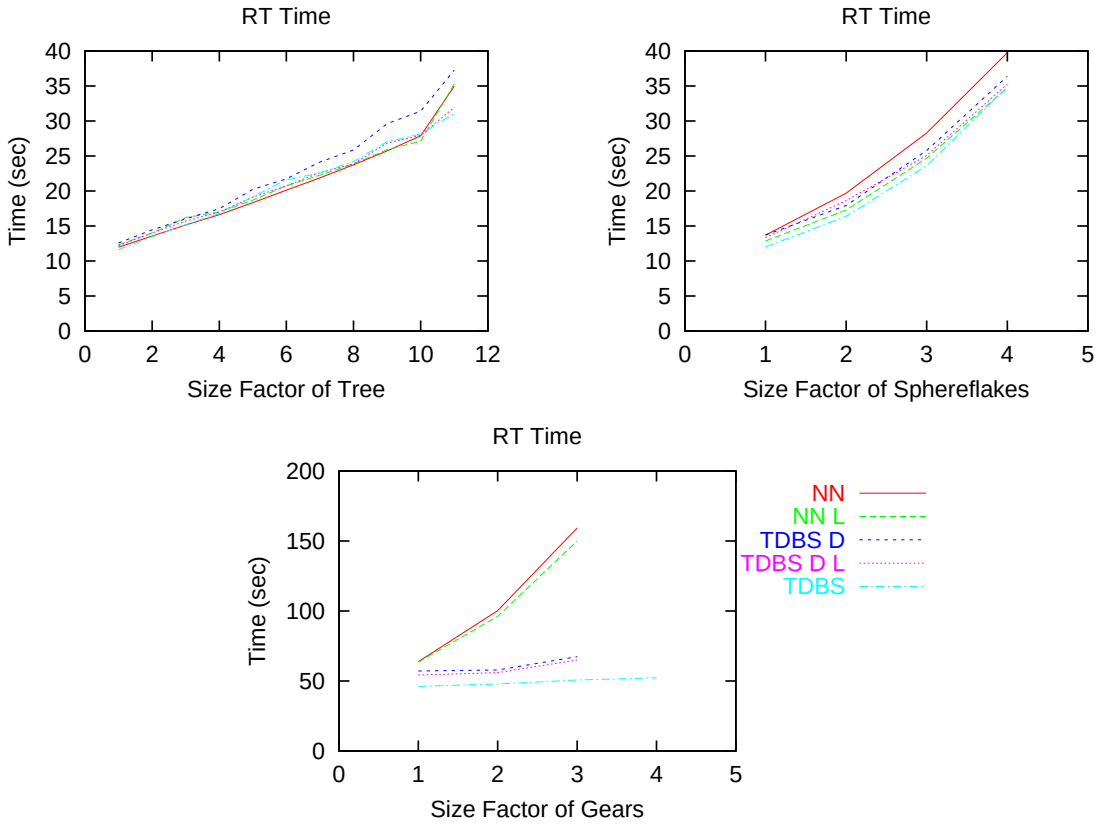


Figure 6.11: Some of the times for the original tree, balls and gears scene. All projected versions followed by L indicate that they are rendered so that shadow rays have knowledge of their light of origin.

processing time than their projected counterparts, since they only create a single hierarchy. When it comes the projected approaches, the ones built on 3D criteria are created a bit faster than those built on 2D criteria, because projecting bounding boxes is cheaper than running the hierarchical algorithm from scratch.

When it comes to preprocessing time of different hierarchical algorithms, the fastest are the R-tree and R*-tree algorithms, since they are approximation approaches of $\mathcal{O}(n\log(n))$. Although the TDBS building is also of $\mathcal{O}(\log(n))$ complexity, the sorting applied beforehand is an extra factor introduced in the preprocessing time. So the TDBS is slightly slower in terms of preprocessing time than the R-tree variations. Finally the NN approach, since it is of quadratic complexity ($\mathcal{O}(n^2)$, is the worst in terms of preprocessing time. The above observations are

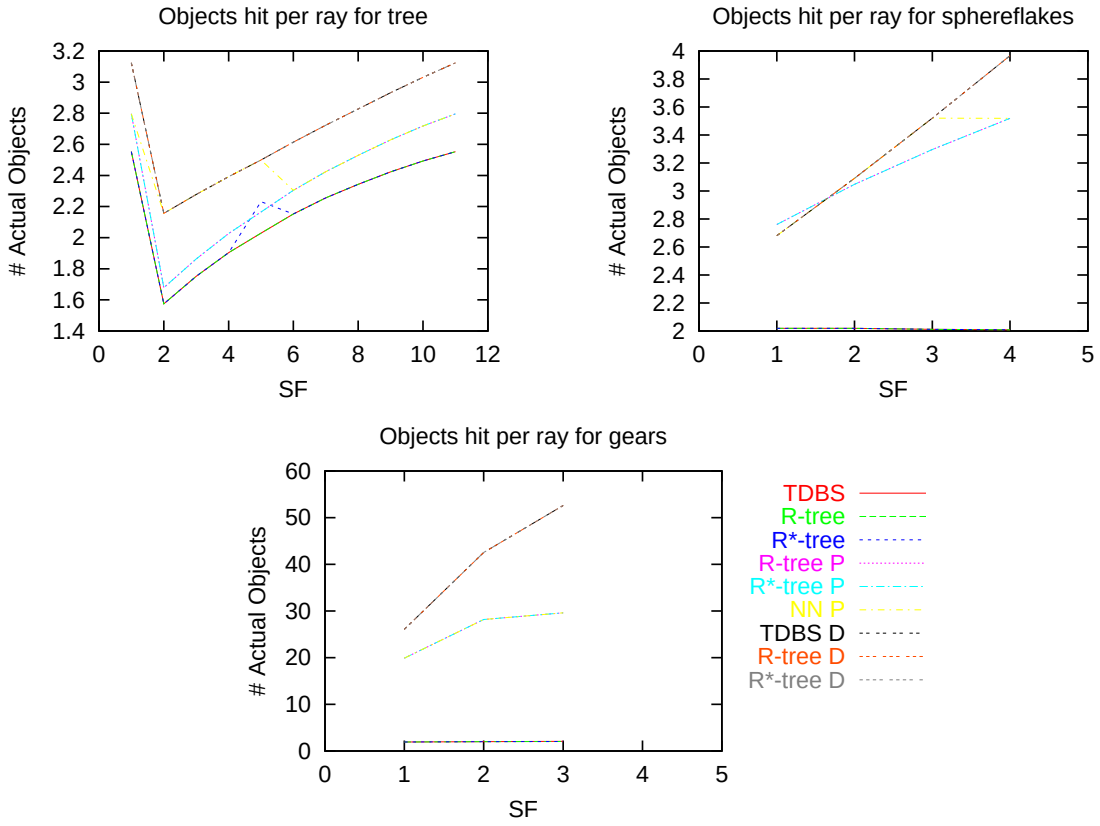


Figure 6.12: Avg number of actual objects hit by a ray for all scenes and hierarchical algorithms.

best demonstrated in the normalized graphs (Figure 6.14).

Hierarchy Traversal Cost: One of the benefits of using a projection hierarchy is the small traversal cost, since in our implementation it consists of a point-in-rectangle check. What we are essentially counting in an effort to demonstrate this fast traversal, is the ratio of traversal cost over the number of intermediate bounding volumes ($cost/\#BV$) vs the size factor. In other words we give the average cost of intersecting an intermediate node of the hierarchy, without considering the cost of intersecting a simple object at the lowest level of our structure.

Our testing demonstrates how much faster the projected hierarchies are traversed compared to their view independent counterparts (Figure 6.15).

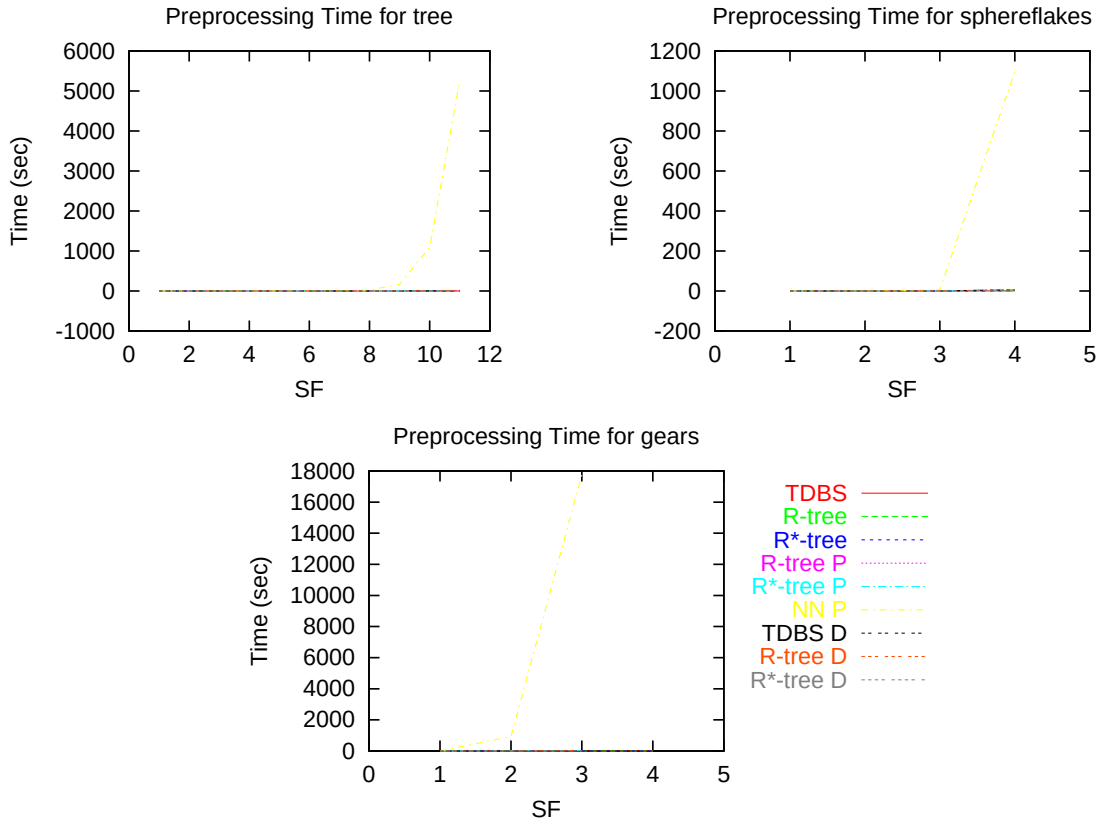


Figure 6.13: Preprocessing time for all scenes and hierarchical algorithms.

We note that the intersection cost in all approaches is considerably smaller in scenes of high SF and in smaller scenes there is a big slope. This is closely related to the fact that in the smaller scenes the average size of a bounding volume is bigger and thus the average number of rays tested against it is larger. Moreover it is an indication that smaller scenes do not really benefit from the use of bounding volume hierarchies. The overhead of creating (especially in projected approaches) and traversing (especially in the view independent approaches) these hierarchies costs in fact more than the gain of pruning rays, as clearly seen in the normalized graph of Figure 6.16.

Per Actual Object Cost: As we have already mentioned, projected hierarchies are less tight than view independent ones and result into bigger number of rays reaching the last level of

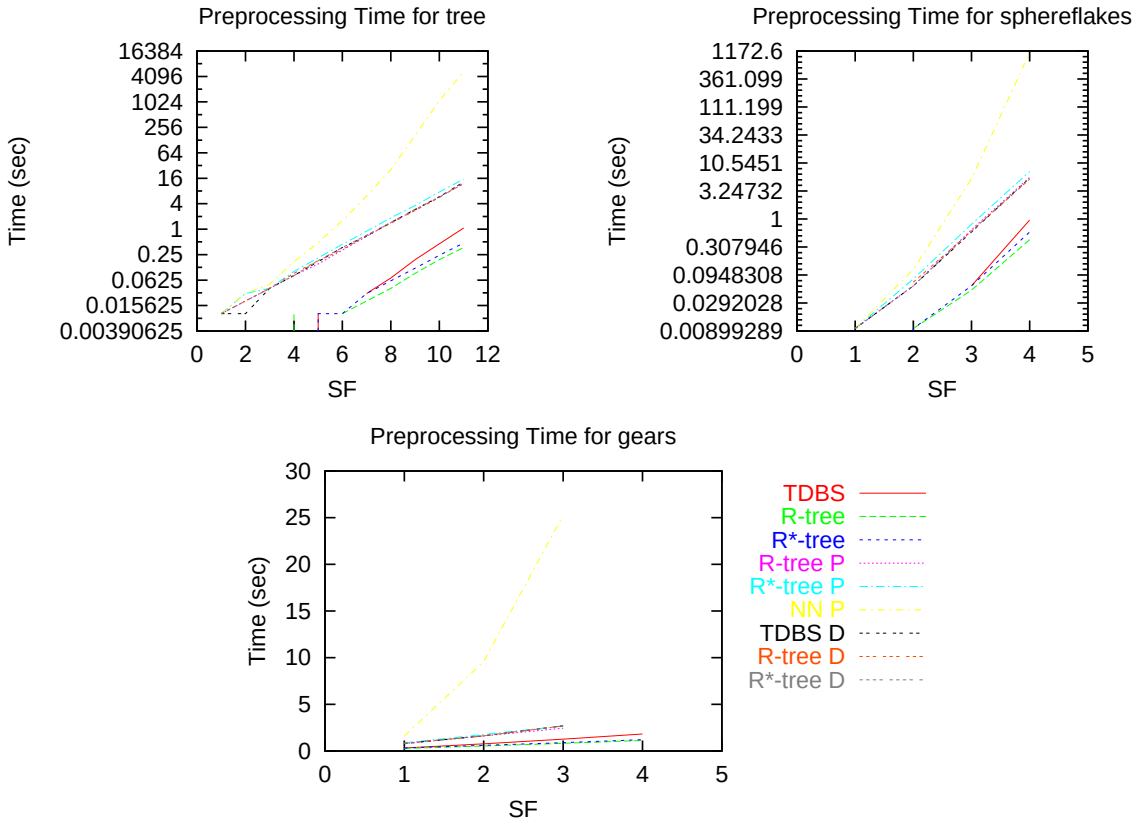


Figure 6.14: Normalized preprocessing time for all scenes and hierarchical algorithms.

the hierarchies, the actual objects. The average cost of intersecting an actual object is thus a good indication of the tightness of a hierarchy (Figure 6.17). The pattern set by the number of objects hit by a ray is also apparent here. View independent hierarchies model the scene more closely and thus prune more rays in the traversal process. We can also identify again the fact that some scenes are in reality more costly to render when using a hierarchy.

As it is apparent in Figure 6.17, the view independent hierarchies model the scene more closely then their projected counterparts, since a smaller number of rays reaches the lowest level and thus a smaller cost per intersecting an object is introduced. Among the view independent hierarchies the R-tree and R*-tree do not model the scene as closely as TDBS, since they introduce an approximation error.

Finally, from the normalized graph of the average object cost (Figure 6.18), it is obvious

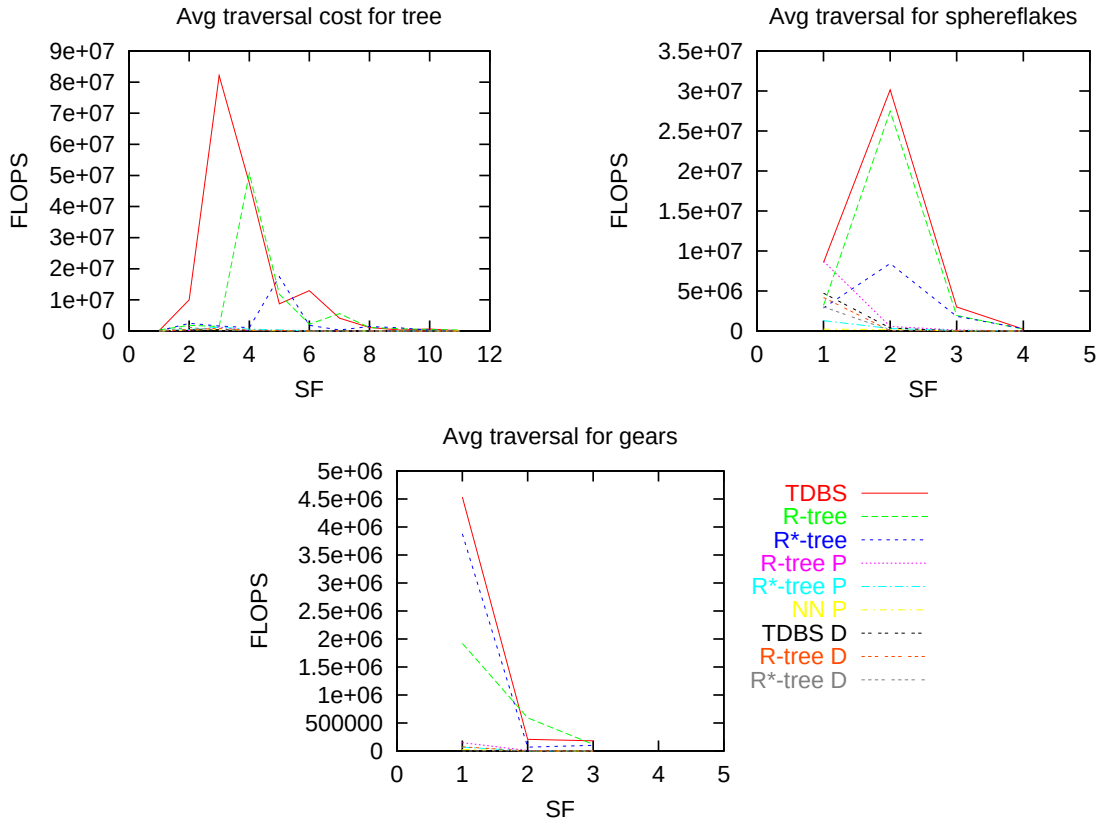


Figure 6.15: Traversal cost of intermediate objects. Projected hierarchies have considerably less traversal cost than view dependent ones.

that the average cost of intersecting an object is lower in bigger scenes. In the scenes tested here, when new objects are introduced they are always smaller than the objects already present. This leads to smaller number of rays hitting the new objects than the old, thus lowering the average cost of intersecting a simple object in the scene.

Scaling: What seems unexpected in all the presented scenes is the fact that although for the Top-Down Binary Splitting (and its dummy) and the NN algorithms the time cost scales linearly with the size factor, the R-tree variant algorithms tend to degrade as the size of the scene gets larger (Figure 6.4).

This linear scaling is the expected behavior since the size factor increments the number of

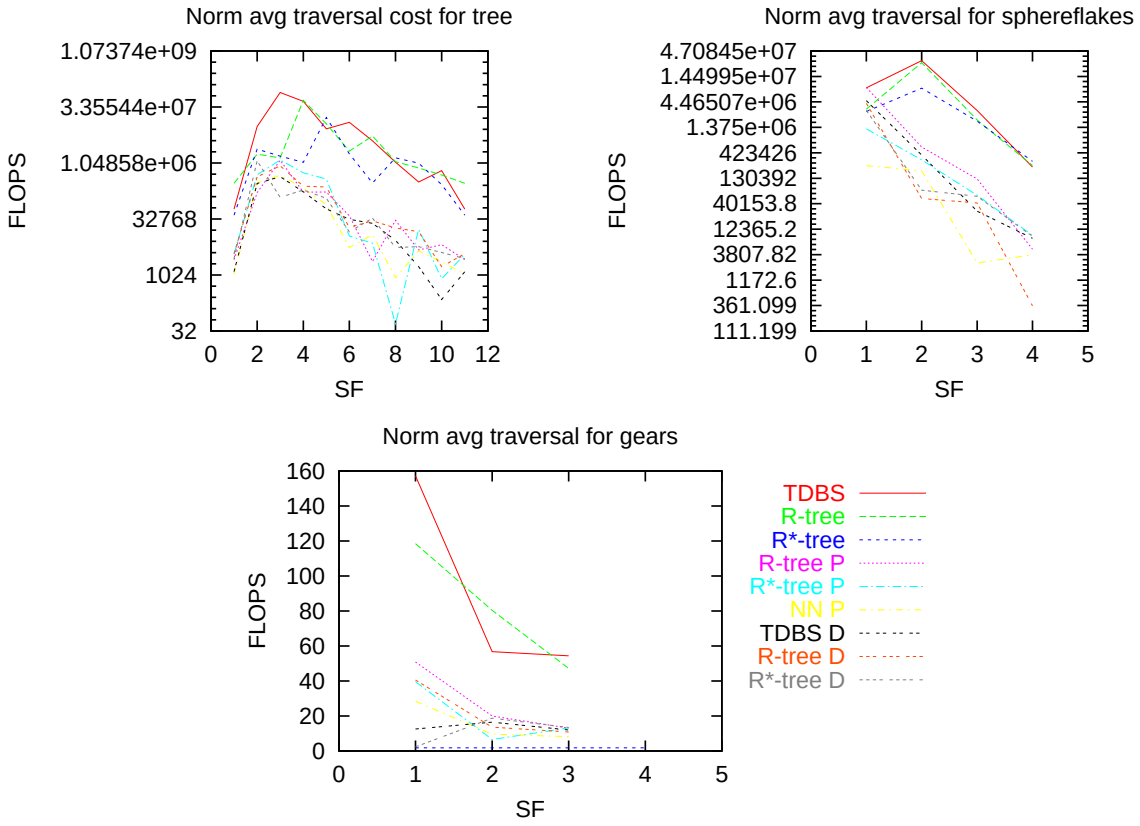


Figure 6.16: Normalized traversal cost of intermediate objects. The traversal cost is larger in small scenes and is reduced in larger scenes.

objects in a scene exponentially (for both the balls and tree scene). By using a hierarchy, for these two scenes, we expect an $\mathcal{O}(logn)$ improvement, where n is the number of objects, as we have already mentioned. Therefor, if s is the size factor, $\mathcal{O}(\log n) = \mathcal{O}(\log e^s) = \mathcal{O}(s)$. For the gears scene the equivalent scaling that we expect is $\mathcal{O}(\sqrt[3]{n}) = \mathcal{O}(\sqrt[3]{s^3}) = \mathcal{O}(s)$. But this is not the case as seen in our overall normalized times (Figure 6.19).

The unexpected behavior of the R-tree variants in almost all the cases can be attributed to the absence of a-priori knowledge of the scene as in the Top-Down Binary Splitting. More specifically, it is well known ([BKSS90]) that approximation algorithms degrade in quality when very large instances (large in respect to the rest of the scene) are inserted in the hierarchy.

The objects that constitute the floor cover a very big part of the scene without being an

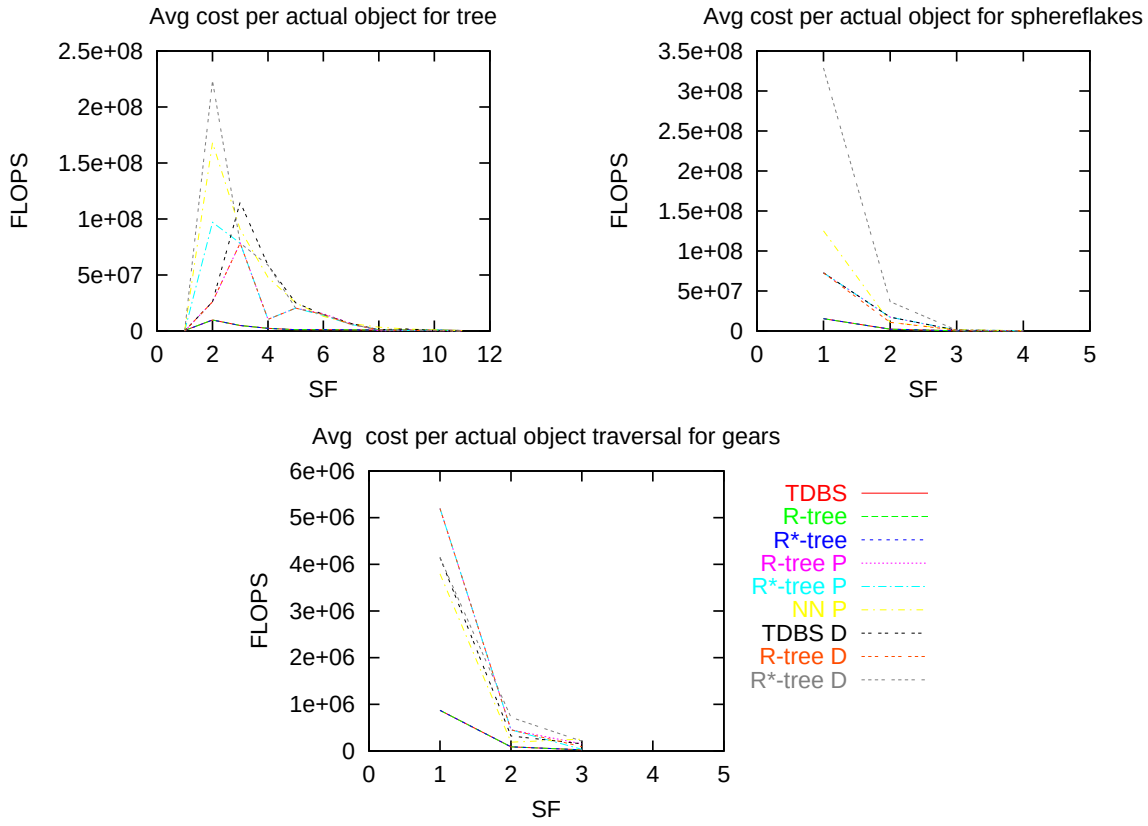


Figure 6.17: Avg cost for intersecting actual objects. View independent hierarchies model the scene more closely.

actual part of the scene. The result in both 3D and 2D hierarchies is unbalanced children and subtrees. Especially in the 2D hierarchies big floors tend to cover a big part of the viewing plane and enlarge the total area of the scene (thus there is no fast test to eliminate directions for light hierarchies).

From these considerations we conclude that the floor of the scenes is indeed a very “particular” object and should not be treated like the rest of the objects that are contained in the scene. To prove our point we divided the floor in the tree and balls scene and observed the scaling of the algorithms (Figures 6.20,6.21). Indeed, after dividing the floor, all the tested algorithms and intersection schemes scaled in the same way and all the normalized times where close to linear. This indicates that objects that are considered to be much larger than the average for a scene should be dealt with in a different way.

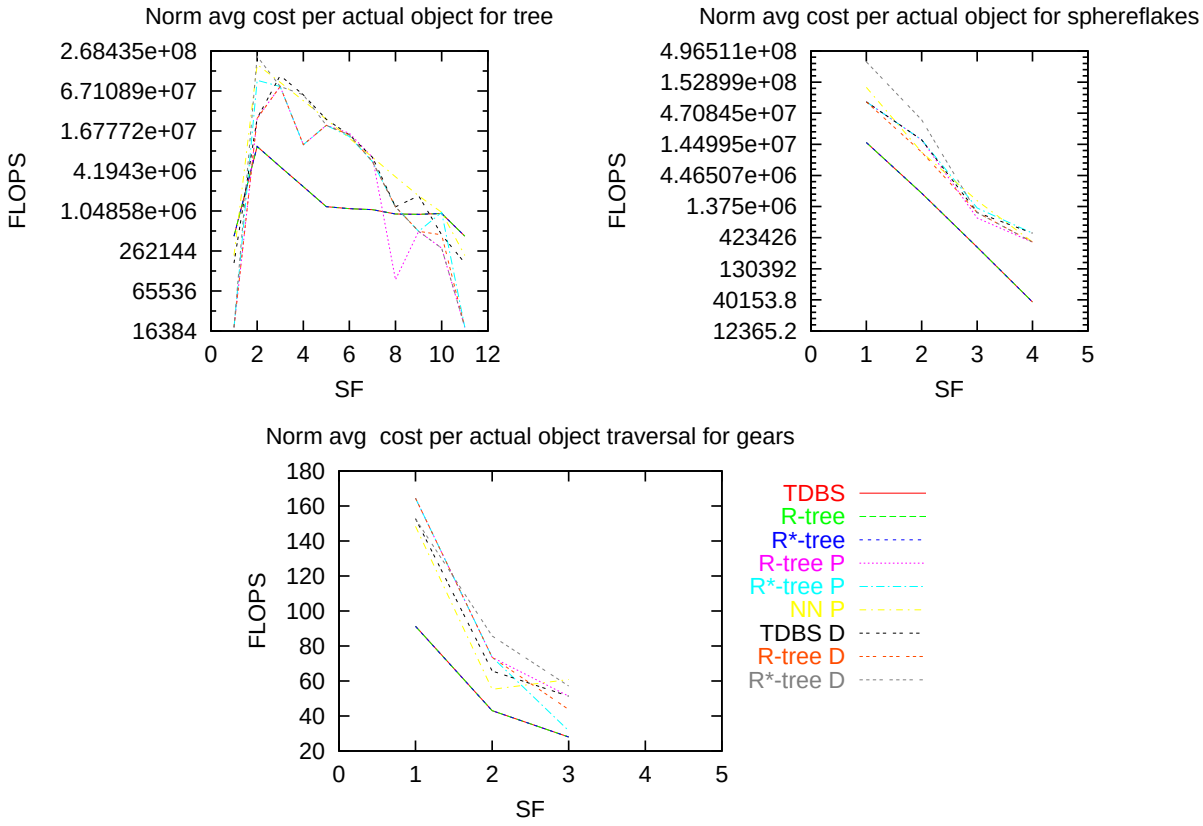


Figure 6.18: Normalized avg cost for intersecting actual objects. Larger scenes cost less in a per-object basis.

Furthermore, we can observe that when the split floor is used the projected versions of the algorithms (both based on 2D and 3D criteria) perform considerably better in the tree scene and slightly better, even in the balls scene. Finally the preprocessing cost for all the hierarchical approaches is ameliorated compared to the scenes where the floor is considered as a single entity.

This is indeed an indication that projection hierarchies are quite faster than their view independent ones when objects are of similar size, even when the overhead of too many reflection and refraction rays is so big as in the balls scene. Finally the preprocessing cost for all the hierarchical approaches is ameliorated compared to the scenes where the floor is viewed as a single entity.

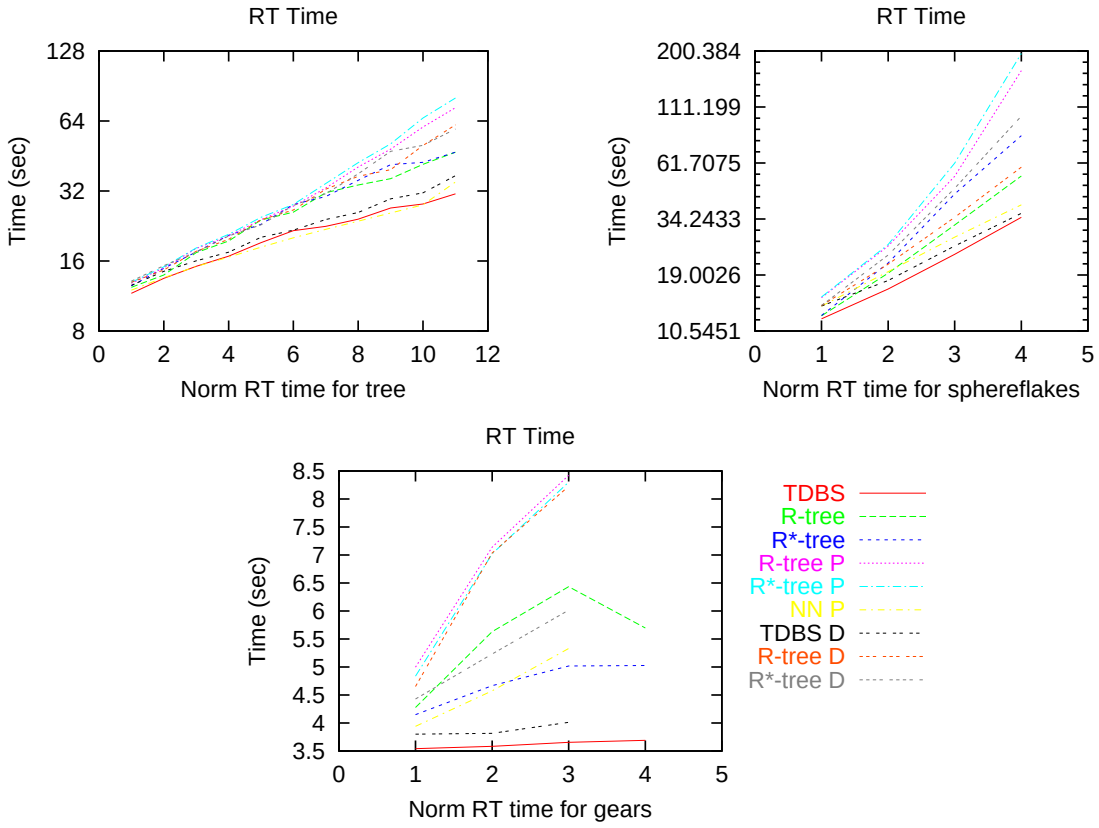


Figure 6.19: Normalized ray tracing time for all scenes and hierarchical algorithms.

6.4 Summary

From the thorough testing we have conducted we have come to these conclusions:

- Sorting the objects of a scene before building a hierarchy yields more efficient structures, but requires “a priori” knowledge of the scene.

- The traversal cost of a 2D hierarchy is insignificant compared to the benefits of a hierarchy that closely represents the scene.

- Average area criteria are better for scenes with limited overlapping (hidden objects).

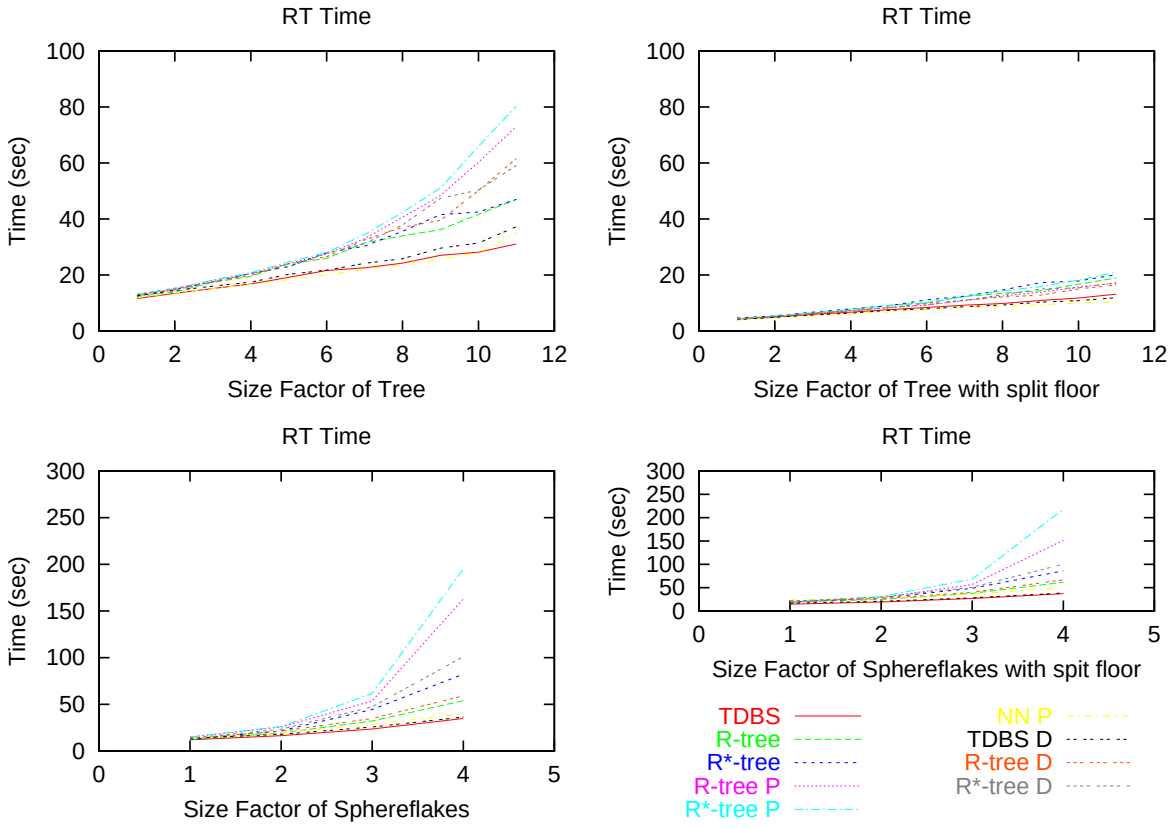


Figure 6.20: Comparing scaling of hierarchical approaches before and after splitting

- Overlapping criteria are better used with scenes having large overlapping (hidden objects).
- A large degree of 2D overlapping amplifies the effects of loose 2D boxes, so combination intersections are preferred.
- In scenes with small 2D overlapping a 2D intersection hierarchy is faster.
- The cost of choosing a hierarchy becomes significant when there is a big number of secondary rays.
- Objects that are considerably larger than the average size of objects negatively affect the quality of the produced hierarchy and slow down the ray tracing procedure.
- Projected hierarchies are faster in large and complicated scenes with a small number of

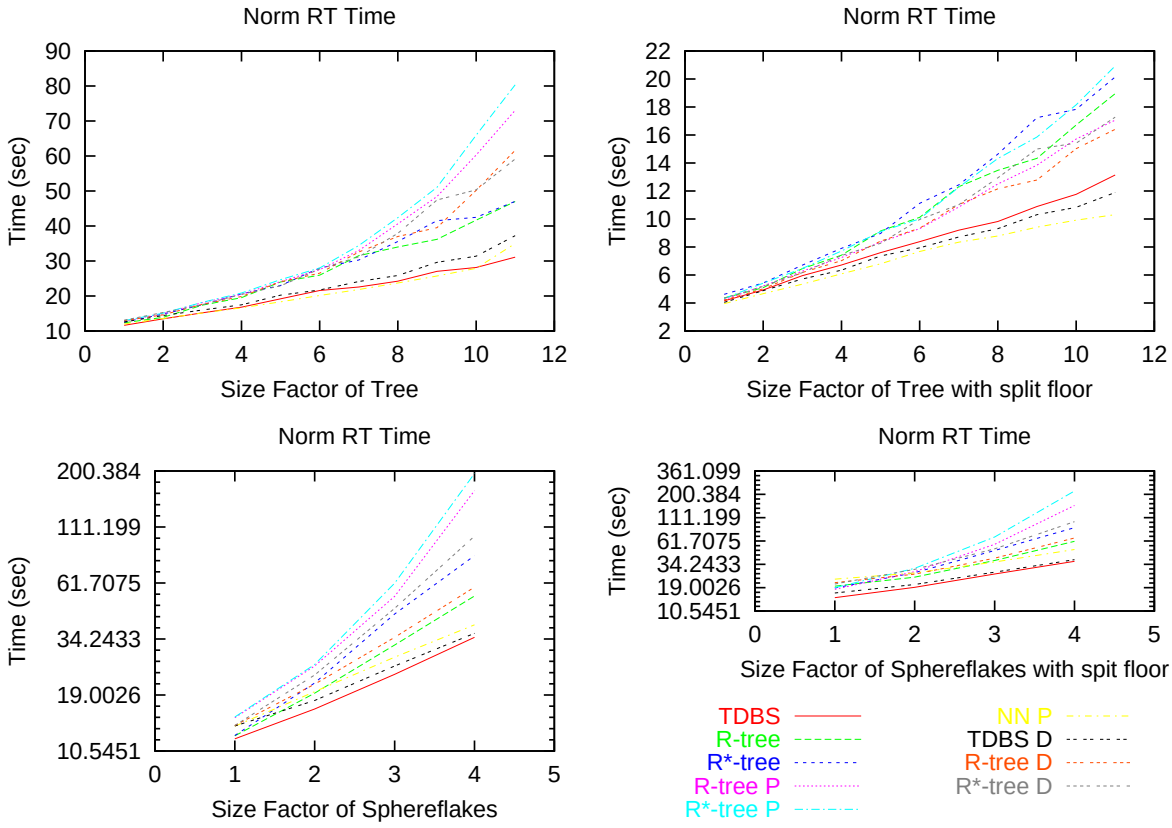


Figure 6.21: Comparing normalized scaling of hierarchical approaches before and after splitting

secondary rays.

Chapter 7

Conclusions

7.1 Summary

In this thesis we present a new way of building bounding volume hierarchies, using the projection of the bounding volumes onto the image plane and the light planes. We propose and test two alternative uses of existing hierarchical algorithms. Firstly, we construct hierarchical structures using only the information derived from the projection of the bounding volumes. Secondly, we construct the view independent hierarchy first and then we project the entire structure onto the viewing plane and the light planes. We show that the traversal of such projected hierarchies is much faster than that of their view independent counterpart, since we discard the tests in one dimension, at the cost of extra preprocessing time.

Although the traversal of a projected hierarchy is faster than a view independent one, the intermediate nodes of the hierarchy do not model the scene as closely as those of the view independent one. This is why we also propose an alternative intersection scheme for traversing the projected hierarchy. The fast traversal on two dimensions is refined and if successful the given ray is also tested against the view independent volume equivalent to the projected box tested. This way the number of rays that propagates inside the hierarchy is limited and fewer rays reach the level of the actual objects of the scene, which are usually expensive to intersect.

With the testing we performed, we demonstrated that indeed projection hierarchies may accelerate the ray tracing procedure. This acceleration is evident when the number of objects in the scene is big and thus the cost of intersecting objects and bounding volumes overshadows the cost of choosing hierarchies. Thus the faster traversal makes a difference in the overall ray tracing time. The acceleration from projection is also apparent in scenes where the number of possible hierarchies to choose from is limited, that is scenes with a small number of lights or lights outside the main body of the scene.

7.2 Discussion

This thesis has provided insight on the tradeoffs in simplifying and accelerating ray tracing. Many factors are at play and by modifying one part of the design surely another is affected.

By transforming a 3D problem to a 2D one we discard the z information of the scene. This is not crucial at visibility testing, but z information needs to be incorporated again in our approach when secondary rays come into play. In other words the fast tests of rays on 2D boxes (compared to 3D boxes) are traded against the loss of z information and of view independence in the approach. The loss of view independence accounts for the big number of generated hierarchies as well as the added traversal cost of choosing the appropriate hierarchy. The tradeoff between faster traversal and view independence has proven to be beneficial in some cases (specifically when there are few secondary rays) but has not in others.

Another tradeoff apparent in our work is the tightness of bounding volumes versus intersecting cost. Tight bounding volumes account for less ray-object intersections but take up more storage space, are harder to compute and expensive to test. Bounding volumes progress from spheres to AABB to OBB to k-dops to convex hulls to unions of convex hulls to objects themselves; intersection cost grows with each refinement. In our work this tradeoff is demonstrated by 2D and 3D bounding boxes. 2D bounding boxes form looser hierarchies that are nevertheless easier to intersect. In some case, for instance when the ray-object test is too expensive, this

tradeoff works in favor of tighter hierarchies (3D ones). This is the reason behind combination tests.

Most of the hierarchical algorithms used today are designed under the assumption that objects in a scene tend to have similar size and so aim at best accommodating an average object. Scenes often do not comply with this assumption and very large objects are introduced into the structure (for example the floors). As seen in our work, larger objects are better accommodated by the hierarchical algorithms when they are split into smaller pieces. Again a tradeoff is detected between performance and the uniform treatment of all objects.

Finally, there is a tradeoff between better dynamic structures and speed. Projected hierarchies are fast to traverse, but they need to be rebuilt when the camera viewpoint changes position (or the light position). This rebuilding process can be put off somewhat when the viewpoint movement is small.

Tradeoffs are indeed present in many aspects of our work and of ray tracing in general: tension between preprocessing time and ray tracing time, time and space, simplicity and speed.

7.3 Future Work

As we observed in our result section, projected hierarchies are really fast to traverse compared to their view independent counterparts. We can furthermore accelerate this traversal. All the projected hierarchies presented so far are oblivious of any information concerning the third dimension. This fact surely costs in terms of getting some fast visibility tests when inside the hierarchy. We can compromise the projection approach by adding some information concerning the z-axis. More specifically, we can alter the splitting criteria in such a way that the z distance of the children from the plane will affect their positioning. So the farthest children from the viewing plane can be placed for instance in the left children and the closest in the right. Thus the right children will be visited first and if there is a hit there (and there is no z overlap with the left child), we can immediately discard a set of objects. At this point the only

z information kept is the fact the children are stored ordered by their z distance.

Furthermore, we observed that the 2-dimensional bounding boxes, derived from enclosing the projections of the 3-dimensional bounding boxes, are not as tight as the original 3-dimensional bounding boxes, thus the need in some cases of a refined combination intersection scheme. We can get over this particular problem by using for each projected hierarchy bounding boxes that are axis aligned in the hierarchy plane instead of in world coordinates. This way the projections would become tighter, but an additional cost would be introduced in the preprocessing time.

Moreover, we observed that objects bigger than the average object size in the scene, like floors, tend to deteriorate the quality of dynamically built hierarchies, especially the projected ones. If floors were to be defined as special objects (with infinite bounding boxes for instance), then the form of the scene would greatly change. This solutions would help all hierarchies, especially the 2D ones, since some of the lights that are now considered inside the scene volume would cease to be so and the hierarchies would not "stretch" to accommodate such big and skewed objects.

The problem with the above solution is the fact that floors (walls, etc) would have to be viewed as a different class of object and special techniques would have to be devised for their ray tracing. The simplest way to deal with such a case thus is to split large polygons. This will help the hierarchies to better adjust and accommodate the smaller pieces in a more efficient way. As we demonstrated in our results, indeed such a splitting ameliorated the performance of all hierarchies greatly, and the projected ones outperformed their view independent counterparts.

The biggest slow down of our approach, as seen in our results, is the fact that rays need to be tested against a large number of hierarchies in order to choose the appropriate one to use. Nevertheless, in many cases they are faster than their view independent counterparts. This leads us to conclude that the projected hierarchies can improve the ray tracing performance even more if we move the selection of the hierarchy to use to the point where a ray is created

when that knowledge is available at constant time complexity, instead of $\mathcal{O}(\log l)$, like in the case of shadow rays (where l is the number of lights). At this point, we have implemented two cases for dealing with shadow rays. According to the first, rays are created and then tested against all lights, in order to preserve a uniform approach for all rays. According to the second, each shadow ray is only tested against the hierarchies of the light it was cast from. It is apparent from our results that this a priori knowledge of light origin speeds up ray tracing in the projected hierarchies. Apart from the cost of choosing a light origin there is also the cost of choosing a hierarchy inside a light. This cost can be reduced if we replace the use of the light cube (6 axis aligned hierarchies) with the use of the exact number of planes that can accommodate a given scene. For example if a light is on a partitioning plane of the scene 2 hierarchies are enough.

In general projected hierarchies are as upgradeable as their view independent counterparts when objects are moved in the scene. This is not the case however when the camera viewpoint or the light position changes. In these cases the projected hierarchies corresponding to the camera or light need to be rebuilt. This rebuild can be postponed somewhat for small camera movements, but not indefinitely. In the future we also plan to look into 2D structures that could be dynamically updated (instead of rebuilt), in order to provide incremental visibility preprocessing.

Finally, projection hierarchies can be easily enhanced to accommodate other commonly used processes in ray tracing, such as supersampling, radiosity, etc.

Bibliography

[AK89] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, pages 201–262. Academic Press, London, 1989.

[BCG+96] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. Boxtree: A hierarchical representation of surfaces in 3d. In *In Proc. of Eurographics'96*, 1996.

[Ber97] Gino Van Der Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[dBvKOS98] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 1998.

[DW85] M.A.Z. Dippé and E.H. Wold. Antialising through stochastic sampling. *Proceedings of ACM SIGGRAPH '85*, 19(3), July 1985.

[FvDFH97] J. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics, Principles and Practice*, chapter 15. Addison-Wesley, 1997.

[Gla84] Andrew S. Glassner. Space subdivision for ray tracing. In *IEEE Computer Graphics & Applications, October 1984*, pages 15–22, 1984.

[Gla89] Andrew S. Glassner. An overview of ray tracing. In *An Introduction to Ray Tracing*, pages 1–31. Academic Press, London, 1989.

[GLM96] S. Gottschalk, M.C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Proceedings of ACM SIGGRAPH '96*, pages 171–180, August 1996.

[GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. In *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD Conference. ACM, New York*, pages 47–57. ACM Press, 1984.

[Hai87] Eric Haines. "a proposal for standard graphics environments" *http://www.acm.org/tog/resources/SPD/overview.html*. In *IEEE Computer Graphics & Applications, November 1987*, pages 3–5, 1987.

[Hai89] Eric Haines. Essential ray tracing algorithms. In *An Introduction to Ray Tracing*, pages 33–77. Academic Press, London, 1989.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, November 2000.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[Kap85] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *ACM SIGGRAPH '85, Course Notes 11*, July 1985.

[KHM+98] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of *k*-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January-March 1998.

[KK86] T.L. Kay and J.T. Kajiya. Ray tracing complex scenes. *Proceedings of ACM SIGGRAPH '86*, 20:269–278, August 1986.

[KOR98] E. Kushilevitz, R. Ostrovsky, , and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th ACM Symposium on Theory of Computing*, pages 614–623, 1998.

[LG98] Szirmay-Kalos Lszl and Mrton Gbor. Worst-case versus average case complexity of ray-shooting. *Journal of Computing*, 1998.

[LRU85] M.E. Lee, R.A. Redner, and S.P. Uselton. Statistically optimized sampling for distributed ray tracing. *Proceedings of ACM SIGGRAPH '85*, 19(3), July 1985.

[RW80] S. Rubin and T. Whitted. A three-dimentional representation for fast rendering of complex scenes. *Comput. Graph*, 14(3):110–116, July 1980.

[SF90] K. Subramanian and D. Fussell. Factors affecting performance of ray tracing hierarchies. Technical report, Department of Computer Sciences, The University of Texas at Austin, July 1990.

[TCL99] Tiow-Seng Tan, Ket-Fah Chong, and Kok-Lim Low. Computing bounding volume hierarchies using model simplification. In *1999 Symposium on Interactive 3D Graphics, ACM*, 1999.

[Tsa99] Panayiotis Tsaparas. Nearest neighbor search in multidimensional spaces. In *http://www.cs.toronto.edu/~tsap/publications/depth.ps*, July 1999.

[WHG84] H. Weghorst, G. Hopper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph*, 3(1):52–69, January 1984.

[Whi80] T. Whitted. An improved illumination model fro shaded display. *Commun. ACM*, 23(6):343–349, June 1980.