Investigating ray tracing algorithms and data structures in the context of visibility.

Kammaje, Ravi Prakash

Investigating Ray Tracing Algorithms and Data Structures in the Context of Visibility

Ravi Prakash Kammaje

Submitted to the University of Wales in fulfilment of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

September 2009

Summary

Ray tracing is a popular rendering method with built in visibility determination. However, the computational costs are significant. To reduce them, there has been extensive research leading to innovative data structures and algorithms that optimally utilize both object and image coherence. Investigating these from a visibility determination context without considering further optical effects is the main motivation of the research.

Three methods – one structure and two coherent tree traversal algorithms – are discussed. While the structure aims to increase coherence, the algorithms aim to optimise utilization of coherence provided by ray tracing structures (kd-trees, octrees).

RBSP trees – Restricted Binary Space Partitioning Trees – build upon the research in ray tracing with kd-trees. A higher degree of freedom for split plane selection increases object coherence implying a reduction in the number of node traversals and triangle intersections for most scenes. Consequently, reduced ray casting times for scenes with predominantly non-axis-aligned triangles is observed.

Coherent Rendering is a rendering method that shows improved complexity, but at an absolute performance that is much slower than packet ray tracing. However, since it led to the creation of the *Row Tracing* algorithm, it is described briefly.

Row Tracing can be considered as an adaptation of *Coherent Rendering*, scanline rendering or packet ray tracing. One row of the image is considered and its pixels are determined. Similar to *Coherent Rendering*, an adapted version of *Hierarchical Occlusion Maps* is used to identify and skip occluded nodes. To maximize utilisation of coherence, the method is extended so that several adjacent rows are traversed through the tree.

The two versions of *Row Tracing* demonstrate excellent performance, exceeding that of packet ray tracing. Further, it is shown that for larger models (2 million+ triangles), *Row Tracing* and *Packet Row Tracing* significantly outperform Z-buffer based methods (OpenGL). *Row tracing* shows scalability over scene sizes leading to a rendering method that has fast rendering times for both large and small models. In addition it has excellent parallelisation properties allowing utilisation of multiple cores with ease. Thus, the *Row Tracing* and *Packet Row Tracing* algorithms can be considered as the significant contributions of the Ph.D.

These data structures and algorithms demonstrate that ray tracing data structures and adaptations of ray tracing algorithms exhibit excellent potential in a visibility context.

DECLARATION

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed [illegible] (candidate)

Date 16/11/2009

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated. Where correction services have been used, the extent and nature of the correction is clearly marked in a footnote(s).

Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed [illegible] (candidate)

Date 16/11/2009

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed [illegible] (candidate)

Date 16/11/2009

Contents

List of Figures

List of Tables

Glossary

Notation	Description	Page List
1D	1 dimensional	82
2D	2 dimensional	14
3D	3 dimensional	8
3DDDA	3D-Digital Differential Analyser - A ray tracing algorithm that identifies voxels traversed by a 3D line (ray)	27
3DS	3DS – The file format used by Adobe 3D Studio Max previously. This format has been superceded by the .max format	164
AABB	Axis Aligned Bounding Boxes – are boxes, or cuboids, that are aligned along the coordinate axes.	29
BIH	Bounding Interval Hierarchy – A hybrid data structure that adapts concepts from the kd-tree and the BVH and uses the space median method to select splitting planes.	38
BSP Tree	Binary Space Partitioning Tree – A binary tree in which space is partitioned into two parts and each part is a node of the tree.	11
BVH	Bounding Volume Hierarchy – A ray tracing structure where objects are enclosed by bounding boxes and a hierarchy is created out of these boxes.	14
CPU	Central Processing Unit	13
FLTK	Fast Light ToolKit – A cross platform toolkit that allows development of GUIs.	175
FLUID	FL User Interface Designer – The user interface designer that allows design of interfaces using FLTK.	176
GB	Giga Byte – A measure of the amount of data. A Giga Byte equals 1024 Mega Bytes.	99
GHz	Gigahertz – A measure of the speed of a CPU.	99
GUI	Graphical User Interface – The user interface consisting mainly of windows and interacted mostly using the mouse.	175
HOM	Hierarchical Occlusion Map – An image that represents the already occluded parts of the image.	42

Notation	Description	Page List
IDE	Integrated development environment – A tool that allows easy development of applications.	164
k-DOP	k discrete oriented polytope – A polytope created by the intersection of planes aligned according to a predetermined set of k axes.	75
KB	Kilo Byte – A measure of the amount of data. A Kilo Byte equals 1024 bytes.	99
kd-tree	k Dimensional Tree – A generalised form of a binary space partitioning tree where the splitting planes are aligned according to one of the X, Y or Z axes.	11
LCTS	Longest Common Traversal Sequence – The longest sequence of nodes traversed by all the rays in a group of rays.	34
MB	Mega Byte – A measure of the amount of data. A Mega Byte equals 1024 Kilo Bytes.	139
MIP	Maximum Intensity Projection – A volume rendering method that projects the voxel with the maximum intensity onto the image plane to create the image.	83
MLRT, MLRTA	Multi Level Ray Tracing Algorithm – A high performance ray tracing algorithm that uses frustum culling and interval arithmetic to trace a group of rays.	34
NV Query	An OpenGL extension introduced by NVIDIA on their Geforce 3 graphics cards.	43
OBB	Oriented Bounding Boxes – are boxes, or cuboids, that need not be aligned along the coordinate axes. Due to the freedom allowed, these boxes more closely wrap the model.	29
Octree-R	Octree for ray tracing – A structure similar to a normal octree, i.e. one in which an internal node has eight child nodes, but where the subdivisions are adaptive. The adaptive nature means that splits need not be placed evenly along the axes. This allows the octree to use the surface area heuristic so that it is better suited for ray tracing.	26
OORC	Object Order Ray Casting – A volume rendering method that shows constant complexity according to number of primitives.	82
OpenGL	Open Graphics Library – A standard for developing graphics applications.	43
QBVH	Quad Bounding Volume Hierarchy – A bounding volume hierarchy where each internal node has four child nodes.	37
R-tree	Rectangle Tree – A tree where the nodes are rectangular and in which the internal nodes may have more than two children.	29

Notation	Description	Page List
RBSP Tree	Restricted Binary Space Partitioning Tree – A form of binary space partitioning tree that allows the splitting plane to be selected by using a predetermined set of axes.	46
RGB	Red Green Blue – A format to represent colours where a colour is given as a combination of the colours red, green and blue	170
S-kd-tree	Spatial kd-trees – A structure similar to bounding interval hierarchies. The difference is that the S-kd-tree selects the splitting planes using the surface area heuristic.	38
SAH	Surface Area Heuristic – A heuristic to select the split position popularly used to build hierarchical structures for ray tracing (e.g., kd-trees or BVHs).	19
SIMD	Single Instruction Multiple Data – Instruction sets that are popularised recently that allow operating on multiple data using a single instruction.	13
SSE	Streaming SIMD Extensions – Intel's set of instructions that allows several (four) calculations (either floating point or integer) with one instruction	8
VRML	Virtual Reality Modelling Language – A language using which 3D scenes are modelled.	164
XML	eXtensible Markup Language – A markup language that allows users to define their own parameters.	172
Z-buffer	Depth Buffer – indicated as Z-buffer as in graphics, the Z axis represents depth in a scene	38

Acknowledgements

The research would not have been possible without EPSRC's funding. The grant has provided finances with which I could work without worrying about finances. It has also provided funds for equipment and travel to conferences that has enriched my research experience.

I would like to thank my supervisor Dr. Benjamin Mora who has been a valuable influence on me for the last four years. He has supported me immensely – technically, academically and morally – during the course of the research and has provided invaluable ideas and input. His patience is also highly appreciated. I would also like to thank Prof. Min Chen, who as my second supervisor, has provided inputs and feedback that has improved the quality of my work. Dr. Robert S. Laramee's initiative in organising the Visible Lunch – a platform for valuable discussions and preparation – is also appreciated.

Swansea University's Computer Science Department, with a research lab that is highly conducive to research, has positively influenced my research, both due to the beautiful workspace as well as due to the people in the lab. My lab-mates have, during the past four years, been wonderful. They have made my stay in Swansea a very happy one. Special thanks should also go to a few lab-mates (in no particular order) – Ben Spencer, Tony McLoughlin, Chitra Acharya, Liam O'Reilly and Ed Grundy – for proofreading my thesis. In addition, for supporting and maintaining the equipment of the lab, the technicians of the lab are also appreciated.

Thanks should also go to my tennis and badminton partners – currently Salar Kasto and Ben Spencer respectively. I would also like to remember my earlier partners – Jibu Panicker, Ganesh M. and Rajmohan Subash. Regular tennis and badminton have kept me physically and mentally fit and for this I am highly thankful to them. My friends, who have enriched my life, are also due credit.

Finally, I would like to thank my parents – Yajnanarayana and Vasanthi Kammaje – without whose support and encouragement, the Ph.D. would not have been possible. I would also like to acknowledge my soon to be wife, Deepthi, for being supportive and helping me through periods of great stress.

Chapter 1

Introduction

Computer Graphics are an integral part of modern life. Computer Graphics is defined as – "Most simply, **pictures** that are generated by a computer" [Hil00]. The most visible uses of Computer Graphics are in computer games and movies. However, they are also extensively used in areas such as automobile engineering, architecture, photography, newspaper and magazines, medical industry, etc. The predominance means that accuracy and performance are very important.

Rendering is the process of generating images from a model. The model normally represents a 3D object based on a physical entity. To render this model implies generating the image of the model from a particular viewpoint. The rendering may either be a photo realistic rendering – which produces images that aim to be close to reality, a non photo realistic rendering – which aim to have more artistic or stylistic properties, or an interactive rendering – that sacrifices realism for high interactive frame rates [PH04].

Depending on the type of renderings desired, the methods used to generate the rendering vary accordingly. Photo realistic renderings use physical properties of surfaces, materials and light to produce images that are very close to reality.

Ray tracing – which is the basis for several methods that generate photo realistic renderings – utilises the principles based on the physical properties of light. In its most basic form, as given by Appel [App68], it is a technique whereby rays are generated and traced, usually backwards – from the eye / camera through the image plane to the model, to find the closest object that each ray intersects. When all the rays corresponding to all the pixels of the image undergo a similar process, the image is generated. The image thus generated does not include any shadows, reflections or refractions. These additional physical effects are computed through further rays that are generated at the point of intersection [Whi80].

Although ray tracing is physically based, it does not model indirect lighting – i.e., light that is dispersed by other objects in the scene. Global Illumination is the name given to the class of techniques that model both direct and indirect lighting – i.e., lighting that is reflected by diffuse surfaces – in order to generate images that are highly photo realistic. Images generated with global illumination look very realistic. Radiosity [GTGB84] was one of the first techniques to model diffuse indirect reflections.

Rasterisation is the other major technique used to render images. It consists of taking the set of polygons of the model and mapping them onto the pixels of the screen to generate the image. Since it is not based on physical properties, optical features like shading, reflections and refractions are generated with artificial techniques and are not as accurate as ray tracing. However, it is extremely fast, making it highly suitable for use when interactive to real time frame rates are necessary. Due to the high frame rates achieved, it is very popular and is used extensively in computer games.

Motivation and Aims

Irrespective of the rendering method used, the first major step of all rendering methods is to find the visible surface. Methods that solve this problem are known as visible surface determination methods or just visibility methods. Very simply, it can be considered as the process of finding the closest object at every pixel of the image.

There have been several methods like Z-buffer, Area subdivision, BSP tree method, etc., to solve the problem [FvDFH90]. Ray tracing / ray casting can also be considered as a visibility determination method when only primary rays are traced. Due to the fact that ray tracing is a very expensive operation, several innovative structures and algorithms have been developed to improve its performance. The underlying principle of most algorithms and data structures is to create coherence and optimally utilise this coherence.

One of the best structures for ray tracing, especially for static scenes are kd-trees. The *Surface Area Heuristic* to construct them has shown to create trees that significantly reduce the number of ray–node traversals and ray-primitive intersections than naive construction heuristics. Kd-trees, and other structures built on the scene, improve and allow utilisation of object coherence by grouping closer objects into closer nodes creating coherence by ordering / sorting the scene.

The other form of coherence, more widely used in recent times together with object coherence, is the use of image coherence by tracing groups of neighbouring rays through the structure. This method is popularly called packet ray tracing. It works on the basis that rays that are closer in the image traverse a similar path down the structure. The combination of object and image coherence has resulted in impressive results. If the visibility problem is considered as a searching problem, packet ray tracing is using a basic result from searching algorithms that searching k neighbouring elements in a tree can be achieved in $log(N) + k$ steps [Ben79]. With this, the number of steps needed to render the image are reduced significantly leading to accelerated performance.

For the visibility problem, finding only the first intersection for all the pixels is necessary. This makes it possible to maximise the utilisation of the coherence provided by the structure and the image. Investigating methods to maximise the coherence provided by ray tracing structures to the visibility problem is the motivation of the thesis.

Thus, the aims of the research are to develop and investigate new visibility / rendering algorithms that build upon and utilise the coherence of structures and algorithms currently popular for ray tracing and to investigate them in that context.

Contributions of the Research

The main contributions of the research are:

- the development and study of a structure that aims to minimise the number of ray–node traversals and ray–triangle intersections by providing a closer fitting structure – i.e., a structure that improves object coherence.

- the development and study of an algorithm that utilises the entire coherence provided by a kd-tree and investigates empirically the complexity of this algorithm.
- the development and study of a new algorithm that utilises the coherence provided by a kd-tree or octree by using a single row at a time – similar to scanline algorithms. The method is extended so that groups of rows can be traced to maximise coherence utilisation.

Thesis Outline

The thesis starts off with this chapter – Introduction – that introduces the main motivations, aims and contributions of the thesis.

Chapter 2 – describes the previous and related work in ray tracing. It also describes briefly the popular visibility methods of the other popular method of rendering – rasterisation.

Chapter 3 – introduces and describes *Restricted Binary Space Partitioning Trees (RBSP Trees)* in detail. It describes the motivation, the heuristics to construct them, the algorithms to traverse them for producing images and finally compares it to kd-trees.

Chapter 4 – Introduces the concept of *Coherent Rendering*, develops the algorithm and investigates it from the point of view of empirical complexity.

Chapter 5 – introduces the algorithm – *Row Tracing* – and its packet variant – *Packet Row Tracing* – in detail and provides the results when the algorithm is used to generate an image.

Chapter 6 – briefly summarises the contributions of the thesis. It will also provide a brief list of future work that can be attempted to realise the potential of the introduced structure and algorithms.

Publications

The following are the publications and technical reports achieved during the course of the Ph.D. The thesis describes the methods and results of these these publications in greater detail.

Kammaje, R.P.; Mora, B., "A Study of Restricted BSP Trees for Ray Tracing," IEEE Symposium on Interactive Ray Tracing, 2007. RT '07., pp.55-62, 10-12 Sept. 2007. [KM07] – Chapter 3 is based on this publication. It describes the structure and the methods described in the paper in greater detail. The chapter also provides additional traversal methods, one of which significantly improves upon the results presented in the paper.

Benjamin Mora, Ravi Kammaje and Mark W. Jones, "On the Lower Complexity of Coherent Renderings," Swansea University, Technical Report, 2008. [MKJ08] – Chapter 4 provides the algorithm in greater detail with the results pointing to the possibility of a lower complexity followed by a discussion regarding the complexity of the algorithm.

Kammaje, Ravi P.; Mora, Benjamin, "Row tracing using Hierarchical Occlusion Maps," IEEE Symposium on Interactive Ray Tracing, 2008. RT '08., pp.27-34, 9-10 Aug. 2008. [KM08] – Chapter 5 describes the *Row Tracing* algorithm presented in this paper in much greater detail. *Row Tracing* traverses a kd-tree or octree using an entire row of the image at a time to maximise coherence. A 1D version of *Hierarchical Occlusion Maps* is used to determine occluded nodes and occluded parts of triangles. *Hierarchical Occlusion Maps* are shown to be responsible for both accurate visibility determination as well as acceleration of performance. To maximise coherence and performance, an adapted version of the algorithm that traces groups of rows is also developed and investigated.

Chapter 2

Ray Tracing – Algorithms and Data Structures

Contents

While rendering from a viewpoint, the first step is to determine the objects that are visible from the viewpoint. There have been several algorithms and data structures that have been used to achieve this and these are grouped into the class of visibility determination methods. Ray tracing / ray casting can be considered as a visibility determination method when only primary rays are traced to determine the closest object at each pixel. There has been extensive research in ray tracing leading to the development of many data structures and algorithms to efficiently perform ray tracing. The other main form of rendering – rasterisation, also has several visibility determination methods. In this chapter, the data structures and algorithms for ray tracing will be described followed briefly by other visibility determination methods.

2.1 Ray Tracing

Ray tracing as a method to determine visible surfaces was introduced by Appel [App68]. The method introduced by Appel is more commonly known as *ray casting*. Ray tracing is the rendering method introduced by Whitted [Whi80], in which a ray from the eye / camera for each pixel of the image is cast and the first object in the scene that intersects the ray is found. Upon finding this intersection, further rays – a shadow ray from the intersection point to the light source, a reflection ray determined by the rule that its angle is equal to the angle of incidence, and if the object is transparent a refraction ray whose direction is found based on Snell's law – are spawned and a tree of rays is formed (Figure 2.1). The shader then traverses this ray tree, finds the intersection of each secondary ray and gathers the contribution of each secondary ray to ultimately shade the pixel in consideration. Due to the method's close similarity to physical reality, the images generated are of excellent quality. However, ray tracing as described models only direct illumination – specular and refracted components. In order to be more realistic, indirect illumination also needs to be modelled and the methods to generate such images are classified under the group of methods known as *Global Illumination*.

An observation to be noted is that the rays in ray tracing, in contrast to physical reality, originate from the camera rather than the light source itself. This allows only the set of relevant rays to be considered. Also, several objects may be occluded by objects in front of them – in which case the rays are blocked by other objects in front. Thus, tracing these rays is unnecessary and wasteful. Only the rays of interest – rays initially originating from the camera and passing through the image plane – are traced.

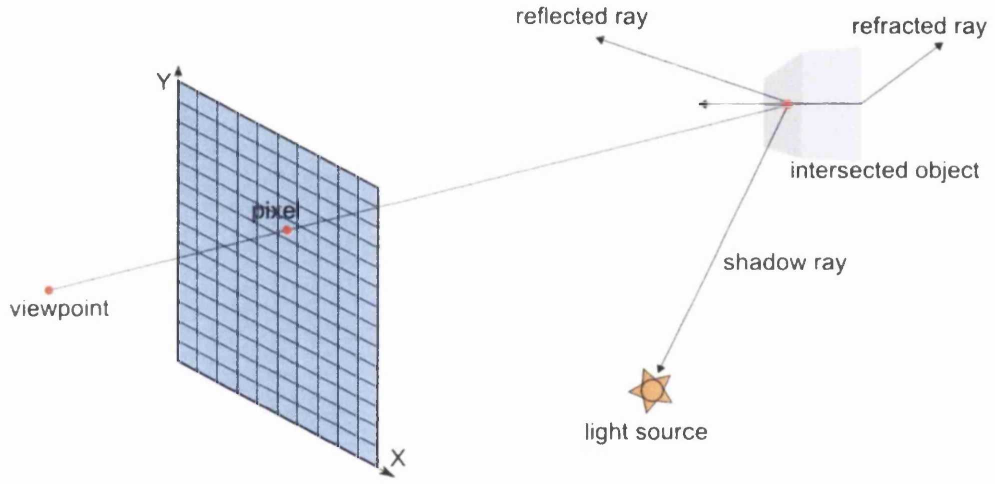


Figure 2.1: Ray tracing. The ray tracing method whereby a ray is traced from the viewpoint through the pixel to find the first intersected object. At the object of intersection, additional rays are spawned to generate reflections, refractions and shadows.

The rendering times to generate a few images, 44 mins, 74 mins and 122 mins, are specified by Whitted. The algorithm was implemented in C on a VAX-11/780 computer running UNIX and the images had a resolution of 480×640 with 9 bits per pixel. This relatively poor performance was due to the hardware limitations as also due to the fact that it was a new algorithm with a very large number of calculations performed. Appel used simple spheres as bounding volumes of the objects to simplify intersection calculations. However, it was revealed that for simple scenes, intersection calculations formed 75% of the time spent by the ray tracer. This was exacerbated for more complex scenes for which the intersection calculations formed upto 95% of the time spent, tending to 100% as the number of objects increase to a very large number. However, the images generated by the method were of a very high quality, engendering immense interest in accelerating ray tracing to generate similarly high quality images, but with a faster performance.

The identification of ray intersections as the main cause of the poor performance of ray tracing led to extensive research – both in accelerating the actual object–ray intersections and in the use of acceleration structures to reduce the number of object–ray intersections. As Section 2.3 will show, numerous acceleration structures have been invented and researched and are of varying quality with respect to ray tracing performance. For a particular scene, the number of object intersections that a ray tracer has to perform is highly dependent on the acceleration structure used. However, irrespective of the acceleration structure used, the ray has to be intersected with the objects at some stage of the algorithm. Hence, fast object–ray intersection calculations are imperative.

2.2 Ray Intersections

A ray is a semi-infinite line and can be represented by its parametric form as

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{2.1}$$

where $\mathbf{o}$ – origin of the ray
t – is the parameter and mostly $t > 0$
$\mathbf{d}$ – direction of the ray

For primary rays, the origin is the viewpoint from which the scene is to be rendered. For secondary rays, it is the intersection point of the primary ray and the object. Since the ray travels only in one direction – forwards – objects that are behind the origin are discarded. The direction is given by the vector from the source of the ray to one of the points that it passes through. It is usually normalised. In case of primary rays, the direction of the ray, $\mathbf{d}$, is given by normalizing the vector from the viewpoint to the pixel being considered.

The problem of finding the intersection reduces to finding the parameter, t, at which the ray hits an object in the scene. Several objects may intersect the ray along its path. But, only the first object that it hits is relevant. This is determined by selecting the object with the minimum positive intersection parameter. A negative intersection parameter indicates that the intersection is behind the viewpoint and hence such intersections are disregarded.

Scenes are most commonly represented using triangles. Most popular acceleration structures use boxes / rectangular cuboids. Hence, fast methods to intersect the ray with triangles and boxes are important. One of the most popular acceleration structures – the kd-tree – uses an axis-aligned plane and hence intersections between a ray and an axis-aligned plane are also considered. Finally, spheres, in addition to being a common primitive, have also been used as bounding volumes in the first ray tracers and hence ray–sphere intersections are also discussed.

2.2.1 Ray–Triangle intersections

In a majority of scenes, objects consist exclusively of triangles. They are the simplest possible polygon and can be compactly represented. Other polygons can be easily broken down into triangles. Complex objects with curved surfaces can be approximated to fine detail depending on the number of triangles used [SB87]. Thus, they are the predominant primitive in ray tracing scenes. In many ray tracers (as in ours) triangles are the only primitive supported. Pharr et al. [PKGH97] use a similar approach of rendering only triangle based scenes and found that any disadvantages of this approach was outweighed by the advantages. The predominance of triangles is both the cause and effect of extensive research in efficient ray–triangle intersections.

The most obvious ray–triangle intersection method intersects the ray with the plane of the triangle and determines whether the intersection point is within the triangle [Bad90] [ray]. Haines [Hai94] gives a few strategies to determine if a point is within a polygon.

One of the strategies to verify if a point is inside the triangle is given by Arenberg [Are88]. It computes two of the three barycentric coordinates of the intersection point by using the triangle's normal – computed either at intersection time or pre-computed and stored earlier. Due to the properties of barycentric coordinates, it is only necessary to verify if they are both positive and their sum is less than one.

Möller [MT97] provides a method that eliminates the need for the triangle's plane equations by computing the barycentric coordinates and the intersection parameter t by translating the triangle onto the YZ plane with the ray along the X axis and then transforming it to a unit triangle. As this method does not need the plane equation of the triangle, this method can be used when memory consumption is a priority.

Wald, in his thesis [Wal04], provides a similar method that is also an optimisation of the barycentric coordinate test. The method first computes the intersection between the triangle's plane and the ray. Then the barycentric coordinates are computed by projecting the triangle onto the axis-aligned plane on which the triangle projects the maximum area. By mathematically simplifying the expressions for the barycentric coordinates of the intersection point, a few per triangle constants are identified. By pre-computing these constants, the number of operations for the test are reduced to a small number (worst case: 10 multiplications, 1 division and 11 additions, best case: 4 multiplications, 5 additions and 1 division). The method is shown to be easily vectorised using SSE instructions (Intel's Streaming SIMD Extensions) [Int08] [SSE09b] to intersect four rays with one triangle.

Another method used for the intersection is the use of Plucker coordinates. It has been used extensively to determine the ray–triangle intersections [Eri97] [Sho98] [TH99] [Jon00]. Plucker coordinates are a mapping of a 3D line into a 5D coordinate system. The coordinates of the 5D system are found by a cross product of the two end points and a subtraction of the two points. It provides a simple method to determine the orientation of one line with respect to the other with an inner product of the vectors. By considering the three edges of the triangle and the ray as vectors, the intersection of the ray to the triangle is found.

Segura and Feito [SF01] provide a method that is faster than either Möller's [MT97] or Badouel's [Bad90] that has been shown to be mathematically equivalent [O'R98] [KS06] [Eri07] to the intersection test with Plucker coordinates with a better efficiency. The scalar triple product – representing the signed volumes of the parallelepipeds defined by three vectors – is used to determine if there is an intersection or not. By eliminating complex operations, the algorithm is simple, fast and robust. As provided, the method only determines if there is an intersection or not and does not calculate the actual point of intersection. However, if there is an intersection, the barycentric coordinates and the intersection points are easily determined by the values calculated as a part of the algorithm.

The scalar triple product can be defined as a function of three vectors, $sign3D$ and can be given as:

$$sign3D(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \mathbf{p}.(\mathbf{q} \times \mathbf{r}) \tag{2.2}$$

where

$\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ are vectors
The symbols . and $\times$ indicate a vector dot product and a vector cross product respectively.

If $\mathbf{o}$ and $\mathbf{d}$ represent the viewpoint and the direction of the ray being traced and $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are the three vertices of the triangle, the intersection can be determined as shown by the pseudocode below:

```
bool IntersectionRayTriangle(a, b, c, o, d, w)
{
        w[2] = sign3D(d, (b-o), (a-o));
        w[0] = sign3D(d, (c-o), (b-o));
        side3 = w[2] > 0.0;
        side1 = w[0] > 0.0;
        if(side3==side1)
        {
                w[1] = sign3D(d, (a-o), (c-o));
                side2 = w[1] > 0.0;
                if(side3==side2)
                {
                        v = w[0]+w[1]+w[2];
                        v = 1.0 / v;
                        w[0] *= v;
                        w[1] *= v;
                        w[2] *= v;
                        return true;
                }
        }
        return false;
}
```

Listing 2.1: Triangle–ray intersection using scalar triple products

If there is an intersection, the barycentric coordinates of the intersection point are given by the values in w. Using these, the actual intersection point can be easily computed as shown by the pseudocode below:

```
intersectionPoint[0]=w[0]*a[X]+w[1]*b[X]+w[2]*c[X]
intersectionPoint[1]=w[0]*a[Y]+w[1]*b[Y]+w[2]*c[Y]
intersectionPoint[2]=w[0]*a[Z]+w[1]*b[Z]+w[2]*c[Z]
```

Listing 2.2: Computing the ray–triangle intersection point using the barycentric coordinates calculated in Listing 2.1.

Due to the simplicity of the method, it is used in our implementation to intersect with triangles. The method has been vectorised using SSE instructions when packet ray tracing is used to intersect four rays with one triangle. When larger packets are used, the packets are split into groups of four rays and intersected using the SSE version.

Closer observation of the above intersection pseudocode, `IntersectionRayTriangle`, reveals that there exist a few optimisations in the process like computing the values of $(\mathbf{a} - \mathbf{o})$, $(\mathbf{b} - \mathbf{o})$, $(\mathbf{c} - \mathbf{o})$ just once in the method. Further reductions in the number of calculations are also possible using the property of scalar products and cross products given below.

- $\mathbf{p}.(\mathbf{q} \times \mathbf{r}) = \mathbf{q}.(\mathbf{r} \times \mathbf{p}) = \mathbf{r}.(\mathbf{p} \times \mathbf{q})$
- $\mathbf{p} \times \mathbf{q} = -\mathbf{q} \times \mathbf{p}$

In addition, when triangle meshes are used, the vertices and edges of a triangle are shared by other triangles pointing to the possibility of several pre-computations as Amanatides and Choi [AC97] suggest. By sharing these computations, they could reduce the worst case computations from 51 flops to 33 flops. However, in our implementation, these optimisations have not been applied as we concentrate more on reducing the number of intersections performed.

In addition to the simplicity of triangles, the availability of several efficient intersection methods has popularised the use of triangles as primitives. However, for a few other primitives like spheres and boxes, efficient intersection methods are available that make it cheaper to intersect them as whole primitives instead of splitting them into triangles.

2.2.2 Ray–Sphere intersections

Spheres are a common primitive in a few scenes. In addition, they can be used as bounding volumes to accelerate ray tracing as ray–sphere intersections are reasonably fast.

The mathematical method to intersect a sphere with a ray, as given in *Realtime Rendering* [AMH02], uses the implicit mathematical equation of the sphere:

$$f(\mathbf{p}) = ||\mathbf{p} - \mathbf{c}|| - r = 0 \tag{2.3}$$

where

$\mathbf{c}$ – is the center of the sphere
r – is the radius of the sphere
$\mathbf{p}$ – is a point on the sphere

At the intersection point, both the ray's parametric equation and the sphere's equation should hold. Substituting the ray equation into the sphere equation at this point gives:

$$f(\mathbf{r}(t)) \;=\; ||\mathbf{r}(t) - \mathbf{c}|| - r = 0$$

expanding the parametric equation of a ray in the sphere equation provides the following at the point of intersection.

$$\begin{aligned} ||\mathbf{o} + t\mathbf{d} - \mathbf{c}|| &= r \\ (\mathbf{o} + t\mathbf{d} - \mathbf{c}).(\mathbf{o} + t\mathbf{d} - \mathbf{c}) &= r^2 \\ t^2\mathbf{d}^2 + 2t(\mathbf{d}.(\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}).(\mathbf{o} - \mathbf{c}) - r^2 &= 0 \end{aligned} \tag{2.4}$$

If we consider
$b = (\mathbf{d}.(\mathbf{o} - \mathbf{c}))$
$c = (\mathbf{o} - \mathbf{c}).(\mathbf{o} - \mathbf{c}) - r^2$

and when $\mathbf{d}$ is normalised, $\mathbf{d}^2 = 1$

then, Equation 2.4 reduces to

$$t^2 + 2tb + c = 0$$

adding b^2 to both sides and simplifying, we have

$$\begin{aligned} t^2 + 2tb + b^2 + c &= b^2 \\ t^2 + 2tb + b^2 &= b^2 - c \\ (t+b)^2 &= b^2 - c \\ t + b &= \pm\sqrt{b^2 - c} \\ t &= -b \pm \sqrt{b^2 - c} \end{aligned} \tag{2.5}$$

If the solutions are real, then the ray intersects the sphere and if the solutions are imaginary, then the ray does not intersect. This can be easily computed by determining if $b^2 - c < 0$ or not. Thus, if as in the case of bounding spheres, it is only necessary to determine if the sphere is intersected or not, then determining whether $b^2 - c < 0$ is sufficient.

The book also discusses a geometric solution as given by Haines [Hai89] that uses the geometric method rather than the algebraic method. The geometric solution considers the various cases where a ray may not intersect and tries to determine these cases with minimal calculations resulting in a more efficient method when the ray misses the sphere. However, in cases where the ray intersects the sphere and the intersection points are necessary, the computations are of similar cost to the algebraic method.

Although ray–sphere intersections are fairly fast, normally scenes in computer graphics do not consist of many spheres. Also, using them as bounding volumes is not very efficient as they do not closely fit most objects in the scene.

2.2.3 Ray–Plane intersections

The ray–plane intersection is used by ray tracers either as a part of the ray–triangle intersections or to determine if a ray intersects a node when a *Binary Space Partitioning tree*(BSP Tree) or kd-tree is used. The mathematical equations of the ray and the plane are used to achieve the intersection.

If the plane to be intersected is given by

$$f(\mathbf{p}) = \mathbf{n}.(\mathbf{p} - \mathbf{p_0}) = 0$$

where $\mathbf{n}$ – is the normal to the plane
$\mathbf{p_0}$ – is a point on the plane

then, at the intersection point, both the ray's parametric equation and the plane's equation should hold. Hence, the equation of the ray can be substituted for the value of $\mathbf{p}$ to give:

$$f(\mathbf{r}(t)) = \mathbf{n}.(\mathbf{r}(t) - \mathbf{p_0}) = 0$$

expanding the above equation gives

$$\mathbf{n}.((\mathbf{o} + t\mathbf{d}) - \mathbf{p_0}) = 0$$

solving this equation gives the solution for the parameter t at which the ray intersects the plane as

$$t = \frac{\mathbf{n}.\mathbf{p_0} - \mathbf{n}.\mathbf{o}}{\mathbf{n}.\mathbf{d}} \tag{2.6}$$

that can be factorised to

$$t = \frac{\mathbf{n}.(\mathbf{p_0} - \mathbf{o})}{\mathbf{n}.\mathbf{d}} \tag{2.7}$$

2.2.3.1 Ray–Axis-Aligned Plane Intersections

Geometrically, a dot product indicates the projection of one vector onto the other. When only axis-aligned planes are considered, the coordinate axes themselves are the normals of the plane. The dot product of a vector with the axis (the projection of a point on to an axis) is just the corresponding coordinate. For eg., the projection of point $(1, 2, 3)$ onto the X axis is 1, onto the Y axis is 2 and onto the Z axis is 3. Using this result, Equation 2.6 reduces to:

$$t_{int} = \frac{p_{axis} - o_{axis}}{d_{axis}} \tag{2.8}$$

where
t_{int} – is the parameter of intersection
p_{axis} – is the point's coordinate along $axis$
o_{axis} – is the corresponding coordinate of the ray's origin
d_{axis} – is the corresponding coordinate of the normalised direction of the ray

As the equation shows, it is a very efficient method.

This simple and efficient intersection method between a ray and plane is one of the causes for the popularity of the kd-tree – that consists entirely of axis-aligned split planes. In addition, it is also used for one of the more popular ray–box intersection methods.

2.2.4 Ray–Box Intersections

One of the most popular methods to intersect a ray with a box is the method proposed by Kay [KK86], more popularly called the slabs method. A pair of planes along an axis is defined as a slab. For each slab, the parameters, t_{entry} at which the ray enters the slab, and t_{exit} at which the ray exits the slab, is computed using the ray plane intersection method given by Equation2.7 (If the plane is axis aligned, then Equation 2.8 can be used instead). The interval given by t_{entry} and t_{exit} determines if the ray intersects the slab. If $t_{entry} < t_{exit}$, then there is an intersection between the ray and the slab in consideration. A bounding volume is the intersection of several slabs and consequently, the intersection of the ray with the bounding volume is the intersection of the intersection intervals. Hence, the ray intersects the volume if – $max(t_{entry}\, of\ all\ slabs) < min(t_{exit}\, of\ all\ slabs)$. For an axis aligned bounding box, this method is shown by the set of Equations 2.10.

It is a very simple method that calculates the intersection of the ray with the six planes of the box and determines intersection by using the values of the intersection parameters. Additionally, it also provides the intersection parameters at which the ray enters and exits the box.

The method has been improved by Williams et al. [WBMS05] by eliminating degenerate intervals caused due to floating point values of -0.0. They also propose pre-computing the results of the division operation to optimise it. However, it still has branches whose misprediction can be quite detrimental. Hence, a branch-less version using the *min* and *max* operations provided by the SIMD instruction set can result in much faster intersection with boxes [GM03] [BP04] [BP05].

The intersection between a ray and an axis-aligned plane is given by Equation 2.8. However, it is well known that division is an expensive operation [SL96], even on modern CPUs, and thus minimizing it is imperative. It may be observed that the term $\frac{1}{d}$ is a constant for a ray that can be pre-computed and stored as d_{rec} . The intersection operation can thus be rewritten as

$$t_{int} = (p_{axis} - o_{axis})d_{rec} \tag{2.9}$$

At the root node, the parameters for the six planes – one entry plane and one exit plane along each axis – of the bounding box / root node are computed as below.

$$\begin{aligned}
t_x &= (bb[x_{entry}] - o_x)d_{rec} \\
t_x &= (bb[x_{exit}] - o_x)d_{rec} \\
t_y &= (bb[y_{entry}] - o_y)d_{rec} \\
t_y &= (bb[y_{exit}] - o_y)d_{rec} \\
t_z &= (bb[z_{entry}] - o_z)d_{rec} \\
t_z &= (bb[z_{exit}] - o_z)d_{rec} \\
t_{entry} &= max(t_x \quad , t_y \quad , t_z \quad) \\
t_{exit} &= min(t_x \quad , t_y \quad , t_z \quad)
\end{aligned} \tag{2.10}$$

where
bb – is an array of the six coordinates representing the maximum and minimum coordinate along each axis ordered axis-wise, i.e., first two values are the minimum and maximum values for the X axis, the next two are values for the Y axis and the final two are values over the Z axis
$x_{entry}, x_{exit}, y_{entry}, y_{exit}, z_{entry}, z_{exit}$ – are the indices of the entry and exit coordinates along each axis.
t_{entry} and t_{exit} – t parameters at which the ray enters and exits the box

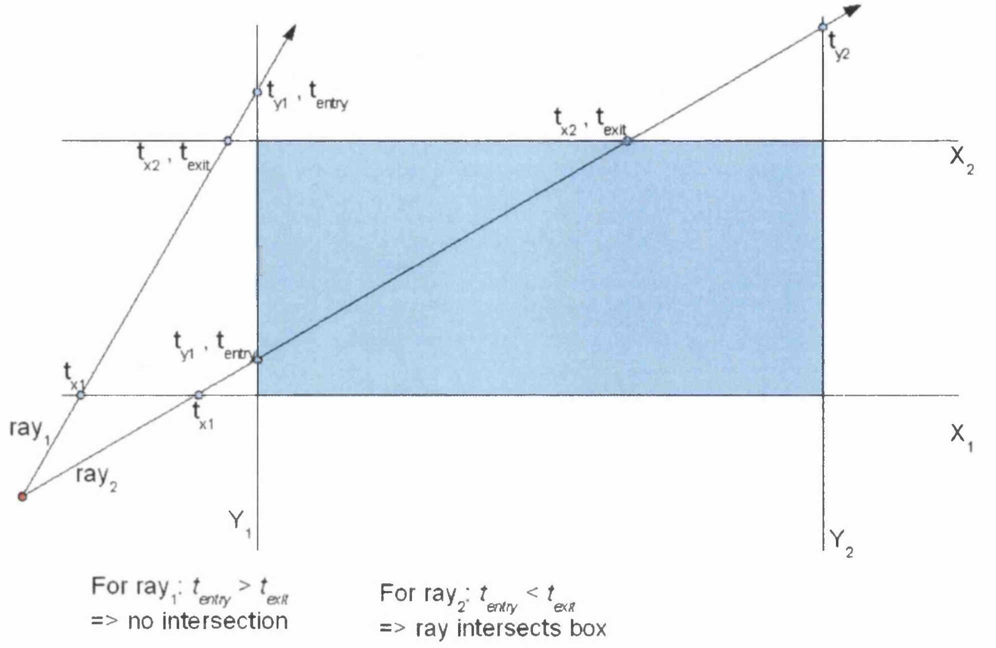


Figure 2.2: Ray–box intersection. $\mathbf{ray_1}$ does not intersect the box as $t_{entry} > t_{exit}$. $\mathbf{ray_2}$ intersects the box as $t_{entry} < t_{exit}$.

Once the t_{entry} and t_{exit} parameters are computed, the ray–box intersection is determined by simply comparing the two. If $t_{entry} > t_{exit}$, then there is no intersection, otherwise there is an intersection with the ray entering at t_{entry} and exiting the box at t_{exit}. Figure 2.2 shows this in 2D.

The entry and exit coordinates along each axis depends on the direction of the ray – determined by the sign of d_{rec} . If d_{rec} is positive, the ray is said to traverse in the positive direction, otherwise it is said to traverse in the negative direction. The direction of the ray determines the order in which the ray enters and exits the box. Normally in a ray tracer, these are pre-computed and stored in a variable with 1 indicating that the ray is travelling in the positive direction and 0 indicating a negative direction for the ray.

Other methods of ray–box intersections have been researched. One of the methods is to compute the intersection by Plucker coordinates [Mah05] [MW04]. The method has the advantage in that it does not need divisions when the intersection point is not needed. Most of the computations of this method are vector operations that allow optimisation using SSE instructions. The method can be used efficiently for traversing BVH trees where the intersection distance is not normally necessary. Woo [Woo90] proposed a method that identifies the back-facing planes of the box, reducing the number of planes to be tested to three. Another recently proposed method for axis-aligned bounding boxes (AABBs) projects the ray and the AABB onto the three axis-aligned planes and determines intersection using the slopes of the projected ray [EGMM07]. The method reduces the 3D problem to a 2D problem and is division free. The authors claim a performance advantage of 18% over the fastest method. However, the method does not determine the actual intersection distances that need to be determined with additional calculations if necessary.

In spite of the several methods available, the slabs method appears to be the most popular method to intersect a ray with a box. This is mostly due to it being simple. It is also a method that can be easily vectorised with SSE. In addition, the method provides the intersection distances without additional calculations.

2.3 Acceleration Structures

Although very fast primitive–ray intersection methods exist, intersecting every primitive with every ray to determine each pixel of the image is prohibitive. Thus, divide and conquer methods by which it is possible to determine the primitive at each pixel by intersecting the corresponding ray against a small subset of primitives are used. The structures that facilitate this – acceleration structures – are classified into two main classes: object subdivision structures and space subdivision structures.

Acceleration structures work on a simple principle. The structure subdivides the scene – either the space or the objects – into several groups of either uniform or varying granularity. Each subdivision, called a *node* for tree structures, usually contains (either fully or partly) a few objects. If a ray does not intersect the enclosing structure, then the objects enclosed are also not intersected by the ray. On the other hand, if the ray intersects the subdivision, then the ray may intersect one of the objects in the node. Thus, only objects in intersected subdivisions need to be tested. Acceleration structures allow reduction of the number of primitives that the ray has to intersect to a very small number.

However, tracing a ray through an acceleration structure adds additional computation to ray tracing. The cost of ray tracing is given by Weghorst et al. [WHG84] as

$$R_t = N_T * C_T + N_{PI} * C_{PI} \tag{2.11}$$

where R_t – is the rendering time
N_T – is the number of node (or bounding volume) traversals
C_T – is the computational cost of traversing a node (or bounding volume)
N_{PI} – is the number of primitive intersections
C_{PI} – is the computational cost of a primitive intersection

If acceleration structures are not used, the first term would be zero. However, the second term would be very large resulting in an impractical cost for ray tracing any scene consisting of more than a few primitives. Hence, acceleration structures for ray tracing – that increase the first term, but reduce the second term significantly – are necessary.

One class of acceleration structures – space subdivision structures – divide the space in a scene into sub-spaces and classify the primitives as being included in any of the subspaces. A few of the space subdivision structures that are used to accelerate ray tracing are BSP trees, kd-trees, octrees and grids. Some of the more popular methods to construct these structures along with the ray tracing method using them will be described briefly in the following sections.

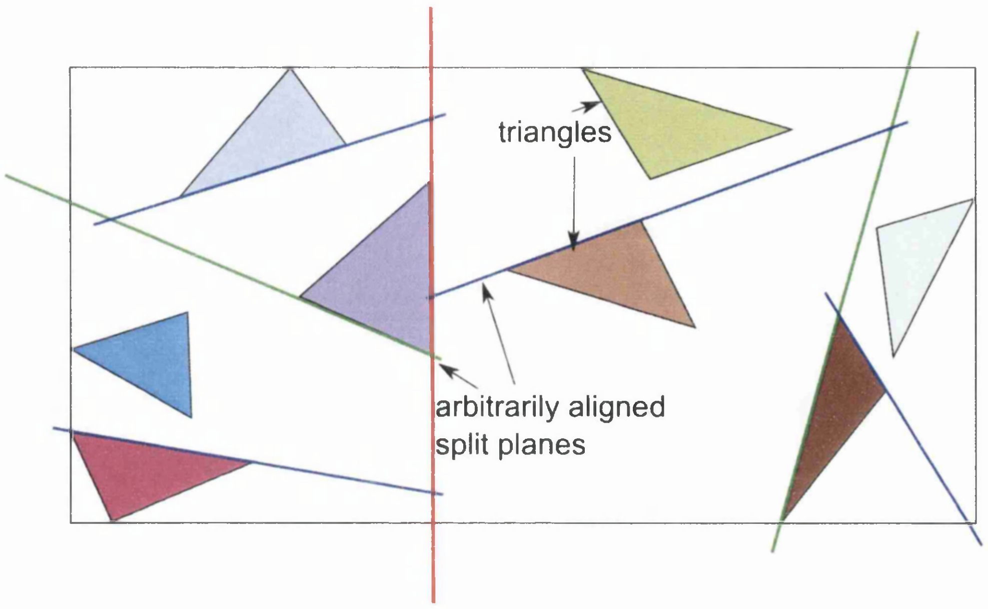


Figure 2.3: A 2D BSP Tree. The splitting planes (lines in 2D) are selected so that they are aligned according to the edges of triangles.

2.3.1 Binary Space Partitioning Trees (BSP Trees)

BSP trees – Binary Space Partitioning trees are hierarchical structures – more widely used in hidden surface algorithms [FKN80] [Thi87] [GC91]. BSP trees partition space into two parts. In their most general form, these trees can partition space along any arbitrary axis. The partitioning axis is normally selected from amongst one of the planes of the scene. The potential of BSP trees to effectively separate triangles and empty space to create a structure that closely fits the scene is quite clear. Figure 2.3 shows a BSP tree in 2D.

One of the only known early implementations of ray tracing on BSP trees [Thi87] states that it can provide improvements over ray tracing performed without partitioning. It uses a median cut scheme that generates balanced trees. However, the efficiency of the structure with respect to ray tracing is in doubt as a comparison against more popular structures is not provided. Recently, Ize et al. [IWP08] show that the use of arbitrarily aligned planes can be useful for ray tracing. Their method will be described in further detail in Chapter 3.

Other than that, BSP trees in their general form have not been used for ray tracing. Most often, the stated reason is the difficulty in constructing a tree that is good for ray tracing. In addition, the fact that the planes can be arbitrary makes storage and traversal more expensive [Cha01] [SF90]. Instead, kd-trees – a class of BSP trees that restricts the splitting planes to be axis-aligned – are used frequently. Another subset of the BSP tree, that in many ways is similar to kd-trees, is *Restricted BSP trees* investigated by us [KM07]. This structure and associated methods to ray trace with it will be detailed further in Chapter 3.

2.3.2 Kd-trees

The most popular space subdivision structure for ray tracing has been the k dimensional tree, more popularly known as the kd-tree. It was invented as a search structure by Bentley [Ben75]. It was adapted to ray tracing by Kaplan [Kap85] (calling them bintrees) and then by Fussel and Subramanian [FS88] who referred to it by the current name. It was adapted using the existing research on BSP trees. The first implementation preferred well balanced kd-trees to reduce the depth of the tree. The implementation used splitting planes placed either at the space median – the mid-point between the minimum and maximum points or the object median – a split with which half of the objects are on either side of the splitting plane.

2.3.2.1 Construction

The main criteria for rendering performance with the kd-tree is the quality of the trees constructed. A well constructed tree can be several times faster for ray tracing than a poorly constructed tree and hence most research has concentrated on improving the quality of kd-trees.

A kd-tree is a recursive structure where every node can be considered as the root node of the sub-tree below it. Naturally, construction of a kd-tree is undertaken in a recursive manner. It is a top down process where initially the entire space is considered. A bounding box is created for all the primitives lying in this space by taking the minimum and maximum coordinates of the primitives along each axis. One among the X, Y and Z axes is selected and space is split along that axis by placing a plane, that is normal to the axis, at the selected point. The primitives are then classified as being on one of the sides of the plane. Some primitives could lie on both sides of the plane and in this case, the triangles are generally included in both the space partitions. Another solution to this problem could be to clip the triangles at thc splitting plane. However, in our implementation, the first approach is used. Subsequently, each partition of space is a node of the tree containing the triangles lying in that part of space.

At each node, the space represented by the node is again partitioned into two by selecting an axis and a point on this axis where the splitting plane is placed. The primitives are classified again and the two space partitions created are made into child nodes. The process is recursively continued until predetermined criteria – called the termination criteria – are met. At this point, the node is made into a leaf node that is not divided further. The two termination criteria normally used are: the depth of the tree of the node, and the number of primitives contained in the node.

The pseudocode below and Figure 2.4 show the recursive construction process.

```
constructKDTree(Node node, int[]nodeTriangles)
{
  //If depth of the tree reaches the termination depth
  //or if the number of triangles in the node is less than
  //a small predetermined number, make it into a leaf node
  if(depth==MAX_DEPTH
      or nodeTriangles.size <= MIN_NODE_TRIANGLES )
  {
    //Make the node a leaf node.
    node.leafNode = true;
    node.noOfTriangles = nodeTriangles.length;
    //The index where the triangles in this node start from
    //is the last index before the triangles are added
    node.triangleIndex = leafNodeTriangles.length;
```

```
        //Add the triangles to the global leaf node triangle list
        leafNodeTriangles.add(nodeTriangles);
        return;
    }
    //Find the split axis and position to split the node
    node.splitAxis = findSplitAxis();
    node.splitPosition = findSplitPosition();

    //Classify the triangles into left part and right part
    int[] leftTriangles = findLeftTriangles(splitAxis, splitPosition);
    int[] rightTriangles = findRightTriangles(splitAxis, splitPosition);
    //Recursively construct left and right parts
    constructKDTree(node.leftNode, leftTriangles);
    constructKDTree(node.rightNode, rightTriangles);
}
```

Listing 2.3: Recursive kd-tree construction algorithm.

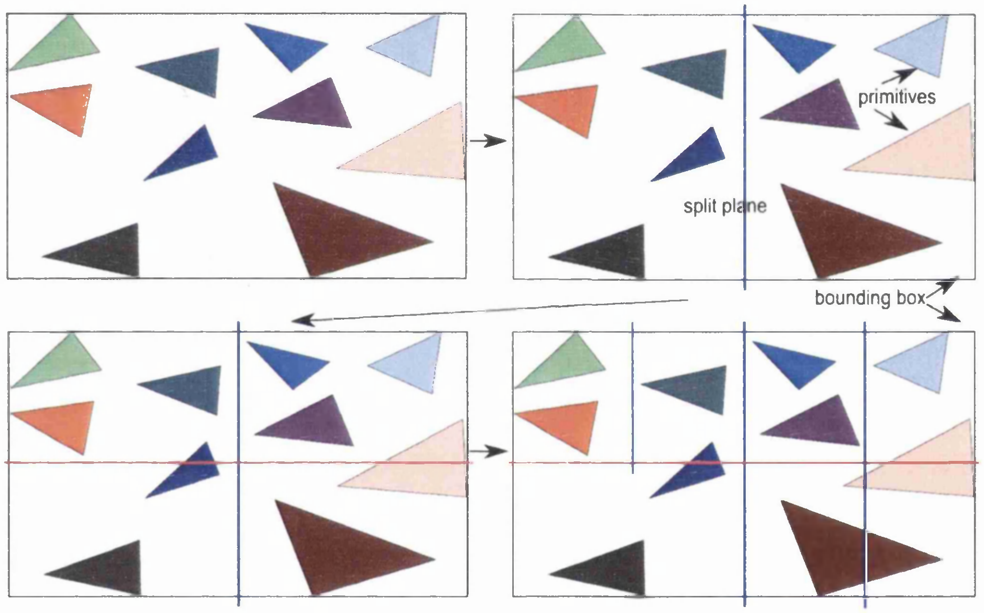


Figure 2.4: Kd-tree Construction with the Space Median heuristic and with termination criteria – maximum triangles in leaf node = 2.

The main challenge in the construction of a good kd-tree for ray tracing is finding a good splitting plane to partition the space in the given node. The position of this plane determines the quality of the tree for ray tracing. Several methods have been proposed to find this split position.

The simplest method used is to place the splitting plane in the centre of the two bounding planes of a node along one of the three axes [Kap85] [FS88]. Using the mid-point[1] as the splitting plane's position is not very efficient for ray tracing. Since a ray can skip traversing empty nodes, it is preferable to identify empty spaces and create empty nodes. In addition, ray tracing can be

[1] Mid-point here means the spatial median. Since, the splitting planes are indicated by a point and a normal (axis), a plane placed at the mid-point of the extent along an axis divides the space into two equal halves.

accelerated by effective separation and classification of triangles to create a kd-tree that closely fits the scene. The *Surface Area Heuristic* (SAH) aims to do this and was introduced by MacDonald and Booth [MB90] to kd-trees, adapted from the Bounding Volume Hierarchy construction method by Goldsmith and Salmon [GS87].

The SAH is a heuristic to determine a locally optimal split position at a node. It takes advantage of the property of kd-trees that the splitting plane can be placed arbitrarily between the minimum and maximum point along an axis. Its conception propelled kd-trees to be the preferred data structure for ray tracing. The SAH has been widely used [MB90] [SF90] [Sub91] [Wal04] [Wal05] [PH04] to accelerate ray tracing.

The SAH is based on the probability of a ray hitting a node and the cost of computing the intersections to the geometry within it [MB90]. The probability of an arbitrary ray intersecting a node is dependent on the surface area of the node itself. This is quantised into a cost that indicates the cost of ray tracing if the split in consideration is used. The cost takes into consideration the probability that the split node is hit and the number of objects contained by it. The intersection cost of each primitive in the node is considered to be a constant. Similarly, node traversal cost is also considered a constant. This is not unreasonable for ray tracers that use solely triangles as primitives. This cost to be calculated at each potential split point is given by:

$$Cost = T_c + I_c \frac{SA(leftNode)LT + SA(rightNode)RT}{SA(node)} \tag{2.12}$$

where
T_C – Cost of traversing a node
I_C – Cost of intersecting a triangle
$SA(leftNode)$ – Surface area of the left node formed by this split plane
$SA(rightNode)$ – Surface area of the right node formed by this split plane
$SA(Node)$ – Surface area of the node being split
LT – Number of triangles in the left node
RT – Number of triangles in the right node
$SA(node)$ – Surface area of the node

The cost is computed as per the above equation at each potential split point along all the axes and the point with the minimum cost – the locally optimal split position – is selected[2]. The split plane is placed at this point and the primitives are classified accordingly.

It may be observed that there can potentially be an infinite number of split points along an axis. MacDonald and Booth derive the property that the split point with the minimum cost has to be between the object median and the space median. In addition, they observe that the SAH cost can only differ significantly at the limits of each primitive (Figure 2.5). Considering only these points reduces the number of SAH costs to be calculated to a manageable number.

The most expensive part of computing the SAH cost is the process of classifying the primitives as being on either side or both sides of the splitting planes and counting them. The naive method to do this is to scan through the entire list of N primitives at each potential split point and classify them. This is a very expensive method leading to a cost in the order of $O(N^2)$. An improved version of this method is to sort the primitives along the splitting axis at each node SAH step [PH04] [Sze03].

T and I are constants in our implementation where only triangles are used as primitives. Hence, using them does not affect the selection of the split plane position using the SAH.

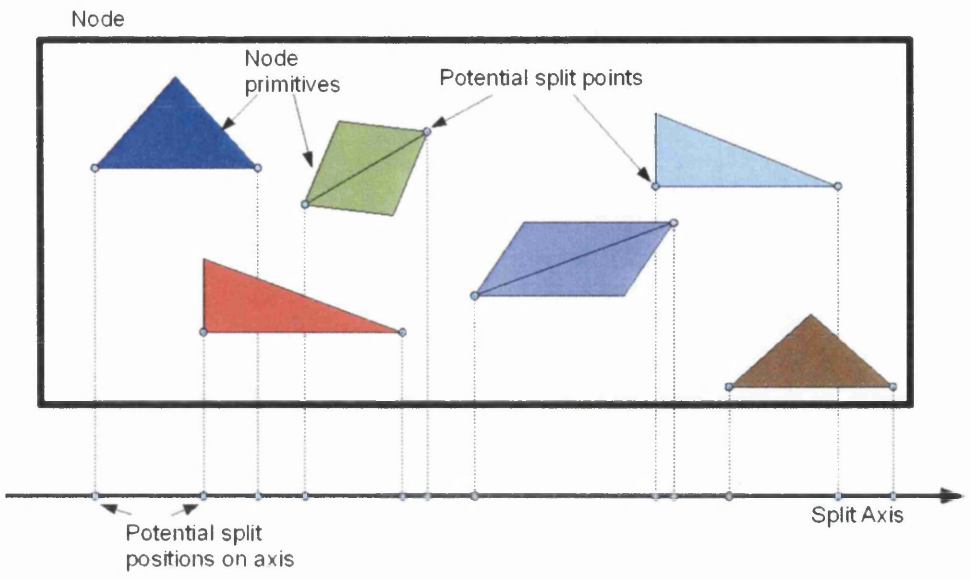


Figure 2.5: SAH potential split positions. The potential split positions are shown as blue points. For every triangle, the two extremeties of the triangle with respect to an axis are considered as potential SAH split positions.

Since the sorting is done at each step, and the number of primitives can be assumed to reduce logarithmically with the depth of the tree, the order of this computation is, on average, $O(Nlog^2(N))$. This has been refined so that it can be achieved with a complexity of $O(Nlog(N))$ [WH06] by sorting once at the beginning and maintaining this order through the construction process.

Classification of primitives is also complicated by the kd-trees's property that primitives can lie on both sides of a split. The property causes problems in counting the number of primitives as well as in the selection of a split point. Using the end points of the primitives leads to incorrect split points as Figure 2.6 shows. To alleviate this problem, clipping the primitives has been proposed [HB02] [HKRS02] so that only parts of primitives inside the node are included – as shown in Figure 2.6.

Clipping also eliminates the counting problem – shown in Figure 2.7 – that occurs when the end points of primitives are used. As can be seen from the figure, if large triangles are not clipped, they can be incorrectly included in a node leading to erroneous counts. Clipping ensures that primitives outside the node are excluded so that the SAH cost is more accurate. The operation is simplified – as shown in Figure 2.7 – where only the position of the potential split points with respect to the split plane are counted to get an accurate count of the number of primitives. It is also possible to ensure that the count is accurate by using a fast triangle–AABB overlap method [AM01].

The cost function can be also be used to favour certain conditions such as empty voxels / nodes [HKRS02] [WH06]. Empty voxels are favourable as rays intersecting them can immediately stop the traversal of the empty node allowing it to skip large portions of space. Thus, if a certain split results in one of the two child nodes to be empty, then the condition is favoured in the SAH cost by biasing it so that the cost is more probable to be less than if the split resulted in two non-empty

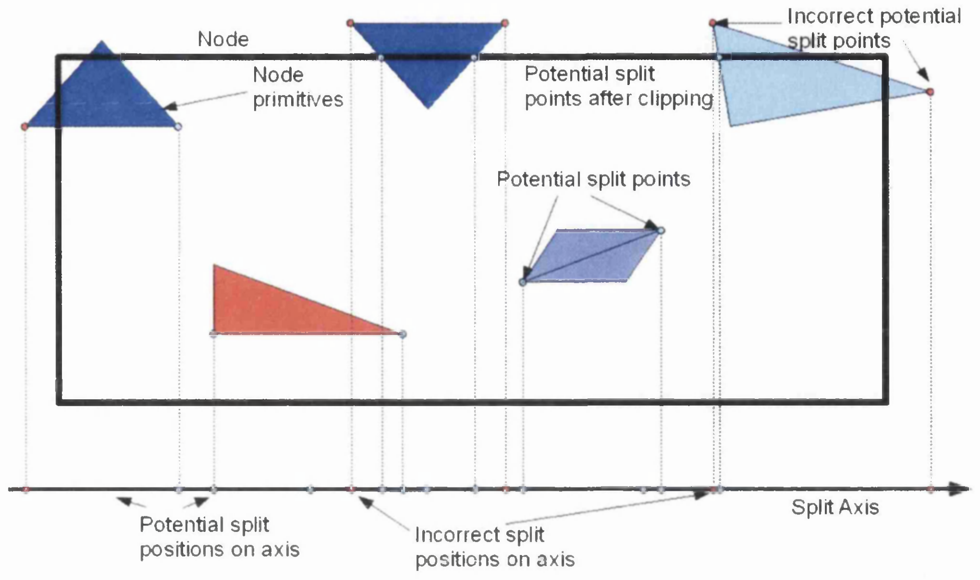

Node
Node primitives
Potential split points after clipping
Incorrect potential split points
Potential split points
Split Axis
Potential split positions on axis
Incorrect split positions on axis

Figure 2.6: SAH potential split positions, showing possible incorrect split positions (in red) if triangles are not clipped.

nodes. In [HKRS02] [WH06] the cost is reduced to 80% of the original cost to add in this bias, resulting in just a 5% improvement in performance.

Termination criteria – One of the factors in the construction of a kd-tree is to determine when a node is made into a leaf node and is not split further. It is the point at which the construction of the tree is stopped as further subdivision would not be advantageous. The two criteria that are usually used are the depth of the tree at a node and the number of triangles in the node at which the node is considered a leaf node. A primitive count of two or fewer is a reasonable point at which the node should be considered a leaf node. The depth of the tree at the node in consideration is also used in conjunction. This criterion makes every node at this depth into a leaf node irrespective of the number of primitives it contains. The depth is normally related to the number of primitives in the scene. One heuristic suggested by Havran [HB02] is to use a termination depth of $1.2log_2N + 2$. This heuristic is found to work well with most scenes with our implementation as well.

Automatic Termination Criteria – Although these termination criteria work very well, they require some prior knowledge of the models and user input. Using the SAH cost to determine an automated termination point was analysed [SF91] and the termination point was determined as the point at which the cost begins to increase. The automatic termination method suggested by Havran [HB02] achieves this by using the SAH cost to determine instances where further splits may not actually help. The SAH cost of the parent is compared to the SAH cost when the parent node is split. When this cost is above a certain percentage, the split is determined as not helpful and thus construction stops here. Another criterion suggested is when the cost indicates a very small probability of the node being hit. At this point, the probability of the node in consideration being hit by a ray is very small and hence splitting the node would be pointless.

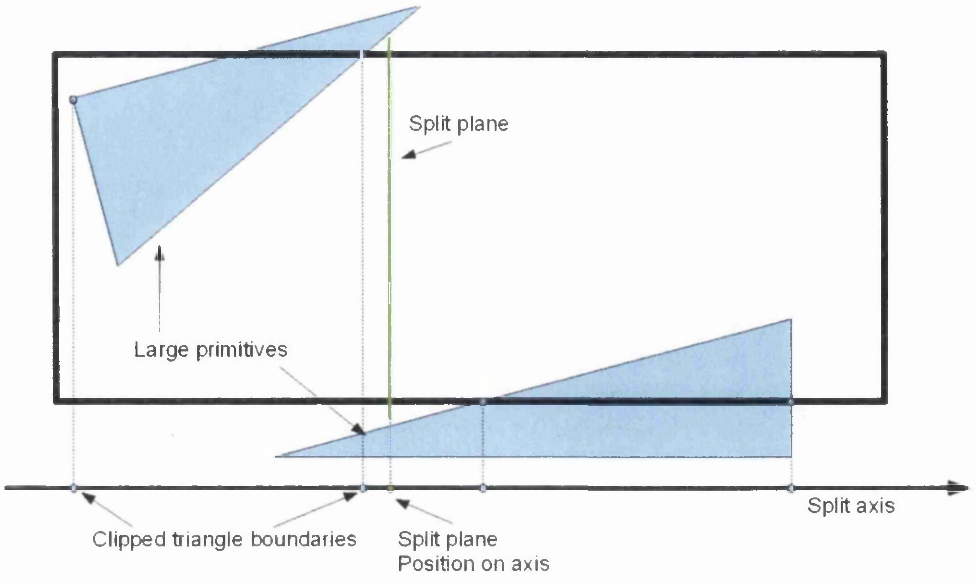


Figure 2.7: Potential for incorrect primitive counts. If the triangles are not clipped, then they can be incorrectly counted. The two triangles would be considered as being on both sides, if they are not clipped.

Improving Construction Performance – Even though the construction of SAH based kd-trees has been shown to be in the order of $O(Nlog(N))$, the constants associated with the construction are quite high, making the process quite slow. Accelerating the construction has thus been an active area of research. It has led to algorithms that base themselves on finding the approximate minimal cost point instead of the exact split point with the minimum SAH cost [PGSS06] [HMS06] [SSK07]. This is achieved by sampling the SAH cost at a few points and approximating the minimal cost by mathematically interpolating the cost between the sampled locations. Hunt et al. [HMS06] refine this by taking further samples in the interval in which the minimum point occurs. On the other hand, Popov et al. [PGSS06] use a higher number of samples to determine a better approximation. Using the sampling method along with parallelisation of the process was also investigated by Shevtsov et al. [SSK07] to further accelerate the construction process.

Once the tree is constructed, the scene can be ray traced by traversing every ray through the tree.

2.3.2.2 Traversal

Kd-trees are an efficient structure to traverse. The traversal of kd-trees has been an area of active research [SS92] [HKBv97] [Arv88]. A kd-tree is a binary tree enabling simple determination of the next node to traverse. The child node to traverse can be determined with just one plane–ray intersection. This is a major advantage for a kd-tree leading to a cheap per node traversal cost.

Another advantage of kd-trees, as of most space subdivision structures, is that they can be traversed in a true front-to-back order. When such a traversal is used, the objects in this node are guaranteed to be intersected before objects in nodes that are intersected at a later time (the exception being objects that span more than one node. As long as the t parameter of the object intersection is between the node's t_{entry} and t_{exit}, the object intersection is within the node and hence correct).

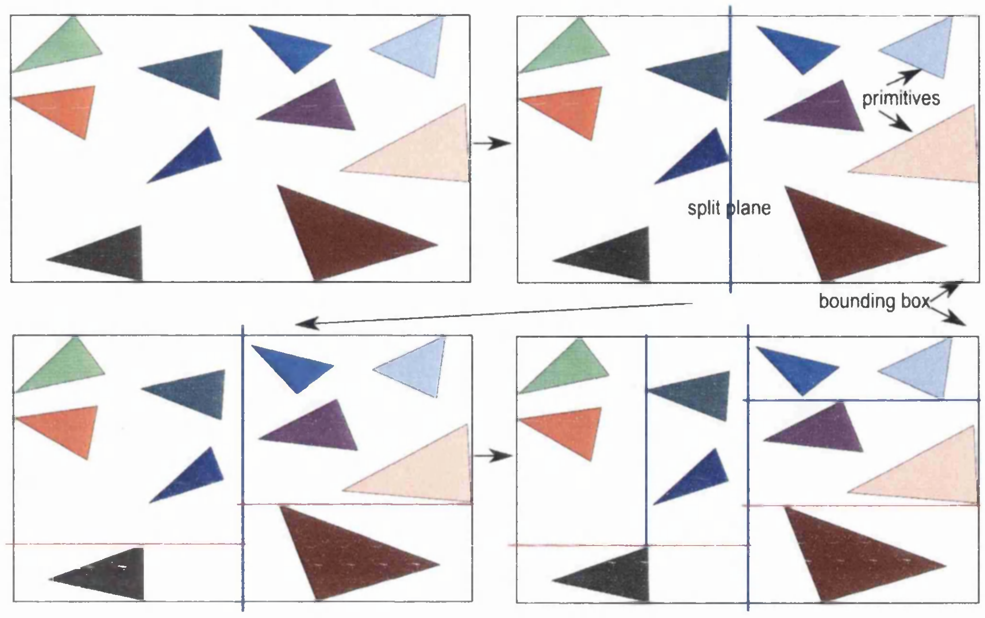


Figure 2.8: Kd-tree Construction with the Surface Area Heuristic and termination criteria – maximum triangles in leaf node = 2.

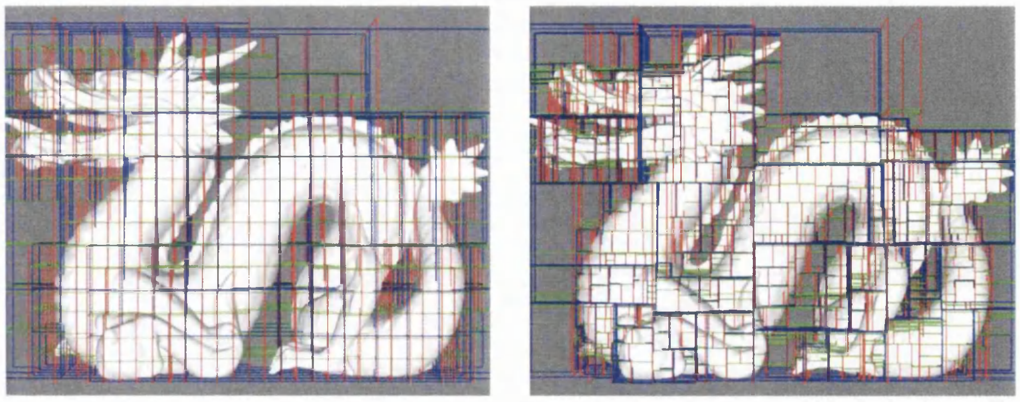
This property of space subdivision structures that allows rays to stop their traversal upon finding an intersection – called early ray termination – saves several traversal steps and object intersections due to occlusion in the scene.

Initially, it is to be determined whether the ray intersects the bounding box of the scene – or in other words, the root node of the kd-tree. This is determined by using the slabs method described in Section 2.2.4. The method also provides the values for t_{entry} and t_{exit} – the entry and exit parameters of the ray. If there is an intersection, then the ray intersects the root node and hence has to be traversed down the tree. Otherwise, the ray misses the root node and hence the entire scene.

If a ray intersects the root node, then it has to traverse the tree in a front-to-back order. Another property of space subdivision structures is that if a ray intersects the parent node, then it has to intersect at least one of the two child nodes. There may be cases when the ray traverses only one child node. The traversal order is determined by calculating the intersection parameter at the split plane, t_{split}, and comparing it to the values of t_{min} and t_{max}. t_{split} is computed as given in Equation 2.8, i.e., by using the term below.

$$t_{split} = (split_{axis} - o_{axis})d_{rec} \tag{2.13}$$

If $t_{split} > t_{min}$, then the ray intersects the left child node. Similarly if $t_{split} < t_{max}$, the right node of the tree is traversed. If both conditions are true, then the kd-tree first traverses the left child and then the right child node. As Figure 2.10 and Listing 2.4 shows, the three possible cases – ray traverses only the left node, ray traverses only the right node or ray traverses both nodes – are handled by just these two conditions.



(a) Space median kd-tree for Dragon (b) SAH kd-tree for Dragon

Figure 2.9: Space median and SAH kd-trees constructed on the Dragon model. The SAH kd-tree more closely wraps the model and reduces the void area.

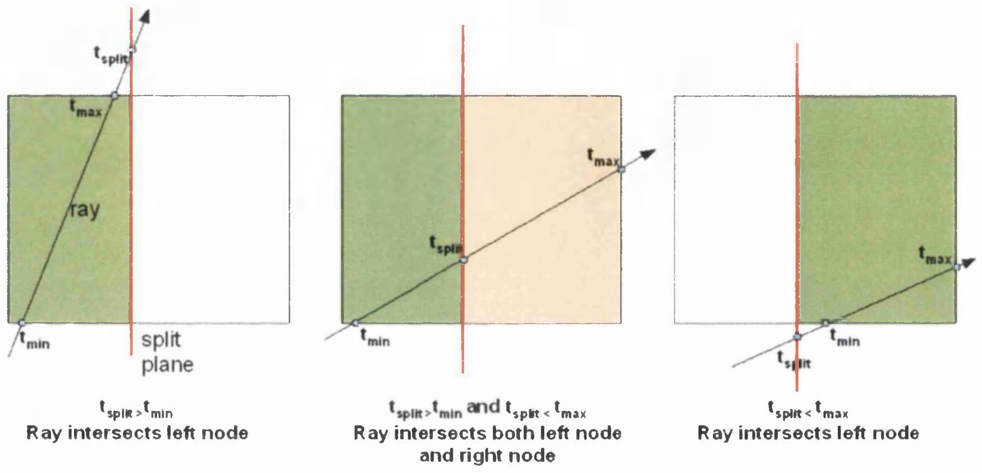


Figure 2.10: Kd-tree ray traversal.

If a ray traverses the tree and reaches a leaf node, then the ray has a chance of hitting one of the primitives contained by the leaf node. The ray then has to be intersected with each primitive and the closest object – as determined by the object with the smallest t intersection parameter less than t_{max} – is the first object that the ray intersects in the scene. The object can then be used to shade the corresponding pixel and spawn additional reflection, refraction and shadow rays as necessary. The pseudocode below shows the kd-tree traversal.

```
int RecursiveRayTraversal(node, tmin, tmax)
{
  if(node is a leaf node)
  {
    if(node is empty)
      return -1;
    return ProcessLeafNode(node);
  }
  currentAxis = GetAxis(node);
  splitPos = GetSplitPosition(node);
  t_sp = (splitPos - rayOrigin[axis])*rayDirectionReciprocal[axis];
  if((t_sp > tmin))
  {
    i = RecursiveRayTraversal(node->leftNode, tmin, min(tmax,t_sp));
    if(i != -1)
      return i;
  }
  if((t_sp < tmax))
  {
    i = RecursiveRayTraversal(node->rightNode, max(tmin, t_sp), tmax);
  }
  return i;
}
```

Listing 2.4: Recursive Ray traversal algorithm. Algorithm computes ray-split plane intersection parameters of the ray and traverses only the left node, only the right node or both child nodes, as shown in Figure 2.10.

The traversal of the kd-tree is simple and efficient – as shown by the pseudocode.

Research has concentrated on comparing several data structures with kd-trees and it was determined by Havran [Hav01] that statistically, at the time of writing his thesis, kd-trees were the best structures available for ray tracing static scenes. Recently, for some applications like dynamic rendering and incoherent rays, BVHs [WBS07] [DHK08] are said to be a better structure. However, it is a fact that kd-trees are among the best structures for ray tracing, especially for static scenes.

A disadvantage of space partitioning techniques is that the primitives may occur in more than one leaf node. This can lead to a ray being intersected against a primitive several times. One of the proposed solutions is mailboxing [APB87] [KA91] [AW87] – a technique that maintains a list of already intersected primitives by the ray and avoids duplicate intersections. It is debatable as to the advantages provided by mailboxing. The overheads associated with maintaining and checking the list leads to a situation where for scenes with largely simple primitives like triangles it may actually be preferable to intersect with the primitive again [Hav02]. Hunt [Hun08] suggests a simple modification to the SAH cost when mailboxing is included. A significant reduction in intersections is shown when using the method, but the performance gained is still in the range of 5% with a maximum improvement of around 10% for one particular scene, leaving the usefulness of mailboxing in doubt.

2.3.3 Octrees

Introduced by Glassner [Gla84], octrees were one of the first space subdivision structure used. An octree splits space into eight regular sub-spaces and the primitives in the scene are classified into these sub-spaces. Since space is proportionally divided, the primitives may span more than one sub-space. Figure 2.11 shows an octree structure in 2D. In 2D, since space is divided only in two dimensions, each node is divided twice and is divided into four sub spaces. The structure is thus called a quadtree.

The root node of an octree is a cube, and hence all child nodes of the octree are also cubes. For ray tracing, the fact that the splits are even and are not according to the primitives in the scene can

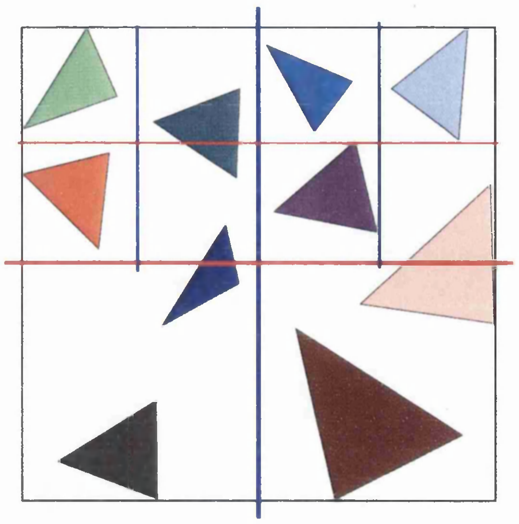


Figure 2.11: 2D version of an Octree (Quadtree). Space is split at even locations and along all the axes (in 2D).

be a serious disadvantage. As shown by kd-trees, effective separation of primitives can make a big difference in the number of nodes traversed as well as the number of objects intersected.

The traversal of octrees is slightly complicated by the fact that each non-leaf node of the octree has eight child nodes. Hence, to maintain the front-to-back order, the traversal order of these eight child nodes has to be considered. The uniformity of the subdivision also allows the use of *Three Dimensional Digital Differential Analyser* like algorithms [Sun91] more popular for grids. Thus, the traversal of octrees has been researched extensively and several methods of traversal have been published [Hav99].

The traversal of octrees can be in one of either forms. It can be a top down approach where the traversal starts from the root node and descends to the child nodes in a particular order until the intersection is found [AGL91] [GA93]. Or, it could be a bottom up approach where the traversal starts in the first node intersected by the ray and traverses the neighbouring voxels [Sam89] of this node until it finds an intersected object.

The algorithm for octree traversal by Revelles et al. [RUL00] is a top down traversal method that uses the parametric form of the ray and by comparisons of this parameter, decides the next voxel to traverse. Each voxel of the octree is indicated by an integer from 0 to 7. Each plane is indicated by a bit and by setting or clearing these bits. The bits are set or cleared based on the value of the t intersection parameter. The integer obtained by the bit operations determines the order in which the octree is traversed. The results provided show that for densely packed uniform scenes, the octree is a more efficient structure than a space median kd-tree as an octree is of lesser depth and hence there are fewer vertical tree traversals.

Applying the SAH to octrees has resulted in a structure called Octree-R [WSC+95]. In the Octree-R, a cost function based on MacDonald and Booth's SAH function is used to determine the split planes along the X, Y and Z axis. This results in a more complex structure – one with better separation of primitives. The authors claim a speed-up of 4% to 47% compared to normal octrees. As expected the construction of the Octree-R is more involved leading to slower construction times. However, they compare the Octree-R only to a normal octree and not to other structures like an SAH kd-tree or a BVH.

2.3.4 Grids

Grids partition the space into small subdivisions [FTI86] [CW88]. They were first applied to accelerate ray tracing by Fujimoto et al. [FTI86]. A Three Dimensional Digital Differential Analyser (3DDDA) traversal algorithm is developed for the traversal of rays through the grid. A faster traversal – also a variant of the DDDA algorithm – has also been used to traverse grids [AW87]. In this algorithm, the next voxel to travel is decided based upon the values of the t parameter of the ray. Initially, it computes the three entry t parameters and an increment value for each axis that corresponds to the length equal to the voxel's dimension along that particular axis. It results in a fairly simple algorithm that decides the next voxel to traverse with simple additions and comparisons.

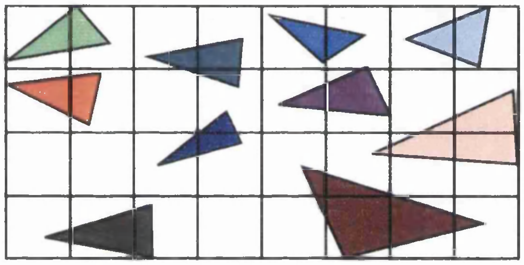


Figure 2.12: 2D version of a uniform grid. The entire bounding volume of the scene is divided into many smaller even spaces.

As with kd-trees, grids are a space subdivision structure in which objects can exist in more than one cell / voxel. To solve the problem of multiple intersections with the same object, Amanatides [AW87] uses the mailboxing technique described earlier. Each ray is given a *rayID* and for each object, the ray that most recently intersected it is maintained and checked. If the *rayID* is the same, then the object has already been intersected with the object and does not need to be intersected again. Otherwise, the intersection is carried out and the *rayID* is updated.

Ize et al. [ISP07] analyze grid creation strategies theoretically and empirically. Using simplified assumptions, a theory is developed. It is then tested empirically. The assumptions are that

- There are N primitives that are all points.
- All rays hit the bounding box of the scene – a cube.
- Each atomic function's cost can be treated as a constant.
- Mathematical operations rather than memory performance, etc., dominate performance.
- The grid has $m^3 = M$ cells.

Using these assumptions, for uniform grids, they deduce that the minimum time to trace a ray is given when the number of cells, M, in the grid is given as

$$M = 2N\frac{T_{intersection}}{T_{step}} \tag{2.14}$$

where
M – is the number of grid cells to create
N – is the number of primitives in the scene
$T_{intersection}$ – is the cost of intersection with a primitive
T_{step} – is the cost of traversing a grid cell

Ray tracing performance on a grid with M cells, calculated according to the equation above, is shown to be very close to the performance with the optimal grids selected empirically. The variation is in the range of $1\% - 5\%$ even for scenes like the conference room scene [GW09] for which the assumptions are far off. They also calculate the optimal number of cells when the scene is composed of triangles with poor aspect ratio and for multi-level grids that work reasonably well. To conclude, they also state that empirically, $M = O(N^{7/9})$ and $M = O(N^{4/3})$ produced the perfect results for manifold-like models with compact triangles and triangles with poor aspect ratio respectively.

Uniform grids work well for scenes that are uniform with primitives distributed evenly. However, for scenes that are non-uniform, they are not particularly well suited. This has led to the development of hierarchical grids and adaptive grids.

Jevans and Wyvill [JW89] initially create a uniform grid over the scene. These are then examined and in spaces where the object density is high, they subdivide those particular voxels into several (i.e., N^3) sub-voxels. The structure is very similar to an octree with the main difference being the number of subdivisions in the voxel. Cazals et al. [CDP95] create a hierarchy of uniform grids by finding clusters in the scene and dividing space occupied by each cluster into further uniform grids. Klimaszewski and Sederberg [KS97] create an adaptive grid structure by first constructing a BVH using Goldsmith and Salmon's algorithm. The boxes are then divided into voxels containing uniform grids to create the adaptive grid structure. Although a comparison with grids is provided, and the adaptive structure is shown to perform better, a comparison with BVHs is not provided.

Due to the simplicity, uniform grids are normally used instead of the adaptive and hierarchical versions. The simplicity reflects in the ease and speed of construction as well as traversal. Although the average traversal time per pixel can be slower than a kd-tree, a uniform grid can be constructed in a fraction of the cost leading to its use in rendering dynamic scenes as Section 2.6 will show.

The traversal of an individual grid cell by a ray is inexpensive and among the cheapest. However, the penalty for this is that there are a larger number of cell traversals if the cells are small enough. If the cells are large, then the average number of objects in each cell may be very high. The average complexity of ray tracing a scene with scattered compact primitives using grids is $O(\sqrt{N})$ [CW88] [ISP07], if N is the number of triangles in the scene. As the scene sizes increase, there is a greater increase in ray tracing times than in hierarchical structures like the kd-tree.

2.3.5 Object Subdivision

Object subdivision structures are a method of subdividing the scene based on the objects and primitives. These primitives are enclosed by simple bounding volumes so that they are only intersected when their bounding volume is hit by the ray.

This type of structure was first described by Clark [Cla76], though not specifically for ray tracing. A structure that grouped objects into hierarchies of bounding volumes was used by Whitted [Whi80]. In his first implementation of ray tracing, spheres were used as bounding volumes as they lead to very simple construction and traversal methods.

These structures can either be flat or more often hierarchies. In a flat object subdivision structure, bounding volumes are not organised as a hierarchy. However, it is advantageous to organise the bounding volumes into hierarchies. If a ray can skip bounding boxes higher up in the hierarchy, larger parts of the scene do not have to be tested. Hierarchies also allow smaller subsets of the primitives to be tested. Hence, most ray tracers use hierarchies of bounding volumes rather than flat bounding volumes. The hierarchical structure is more commonly referred to as a bounding volume hierarchy (BVH). Several different types of bounding volumes have been used.

2.3.5.1 Shapes of Bounding volumes

Spheres were one of the first bounding volumes used as they are very cheap to intersect with a ray [Whi80]. A method to create spheres that closely fit the volume was provided by Ritter [Rit90]. However, spheres do not bound most volumes very closely [WHG84]. Thus, they result in a hierarchy with a large number of bounding volumes resulting in higher traversals and primitive intersections.

Slabs of arbitrarily aligned planes were used as bounding volumes by Kay and Kajiya [KK86]. The arbitrarily aligned planes provided a very close fitting structure. However, the cost of intersection was quite high leading to higher ray tracing costs.

The most popular kind of bounding volumes are axis-aligned bounding boxes [WHG84] [Gut84] [BCG$^+$96]. Axis-aligned bounding boxes (AABB) are rectangular boxes that, as the name suggests, are aligned along a coordinate axis (one of X, Y or Z axes). Trees consisting of AABBs were called R-trees [Gut84] in the context of spatial searching. Due to their abundance, the name bounding volume hierarchies normally refers to a hierarchy of AABBs. As Weghorst [WHG84] demonstrates through the introduction of the concept of void area, AABBs have a relatively low intersection cost and at the same time fit the model reasonably closely. This compromise results in a very effective shape for bounding volumes in the context of ray tracing.

Oriented bounding boxes (OBBs) [GLM96] [Got00] are another kind of bounding volumes. Their alignment is decided by the alignment of their component objects so that they fit the scene more closely than their axis-aligned counterparts [RW80]. However, they are expensive to construct, traverse and store.

2.3.5.2 Construction

The first bounding volume hierarchies were constructed manually by user input [RW80]. This is of limited use as human construction is both tedious and prone to selecting non optimal options, especially when the scene consists of a large number of triangles. An automated approach was hence necessary. Kay and Kajiya [KK86] propose a few automated object grouping criteria. One of them is to group objects as they occur in the scene representation. This approach depends largely on the way the scene is represented. If the scene is represented so that closer objects are in succession, then the scheme results in good BVHs, otherwise the BVHs can be quite poor. Another proposed scheme is the median cut scheme whereby the objects are sorted according to proximity, either on one axis or according to all axis (through the use of an auxiliary structure like an octree or a kd-tree). The objects are then grouped into bounding slabs according to the median. This scheme results in reasonably good trees.

The SAH approach, proposed by Goldsmith and Salmon [GS87] aims to take the automated approach further by considering the probabilities of uniformly distributed rays hitting a bounding volume. The objects are inserted into the tree similar to insertion of data into a search tree and determined if the insertion was a good one or not by calculating the surface area of the new node. The order in which the objects are inserted is important as the node created depends on this order. They attempt inserting the objects in the order present in the model or in a randomised manner. This does not result in better trees, mainly due to the bottom up approach rather than the SAH itself [WMG+]. Although, the SAH for BVHs was not widely used, it led to the SAH for kd-trees that, as already discussed, was the major catalyst to accelerate ray tracing speeds to interactive / real time performance.

The main challenge with SAH for BVHs is to find the locally optimal bounding box, as there are $O(2^N)$ ways to split a set of primitives into two subsets [WBS07]. Müller [MF00] suggests a top down SAH construction method that is very similar to a kd-tree construction method. They use an SAH heuristic that has potential split positions at the two end points of the triangle along an axis and selects the box with the minimum cost. Similar to the kd-tree process, each split point along each axis is tested and the box with the minimal cost is selected. Masso and Lopez [ML03] use a different cost model and apply that using a Goldsmith and Salmon tree building approach with objects selected using another BVH built earlier using the object median or the approach suggested by Muller [MF00]. Ng and Trifonov [NT03] apply random perturbations to the split points and also investigate evolutionary approaches to improve the original tree generated. The method provides only marginal improvements on the original tree while adding computational overhead.

Wald et al.'s approach to building BVHs [WBS07] using the SAH is to find a good instead of optimal partition. They investigate evenly spaced candidate planes, the bounding sides of each triangle as candidate planes and planes placed at the centroid of each triangle. Unexpectedly, all approaches resulted in similar performance. The centroid based approach was ultimately preferred. The triangles were classified as being in either sub-node based on the position of the triangle's centroid.

As with kd-trees, the use of SAH makes the construction relatively slow. However, similar to the kd-tree heuristics, approximate SAH builds [Wal07] have been suggested to accelerate the build process. With better quality BVHs available interest has significantly increased, especially for deformable and dynamic scenes [WBS07] [LYMT06] [WMG+]. There has also been interest in BVHs to trace incoherent rays [DHK08] [WBB08] for which BVHs consisting of more than two child nodes at each level are shown to be an excellent structure, making efficient use of SIMD packets.

A problem with BVHs is that the bounding volume of large triangles can be very large, leading to a large empty space, especially for non-axis-aligned triangles with a poor aspect ratio. A solution to this problem is to split the triangles with one of the two approaches that have been proposed. The first solution, proposed by Ernst and Greiner [EG07], is to clip the larger triangles (based on the surface area) with axis-aligned planes to create smaller triangles prior to construction of

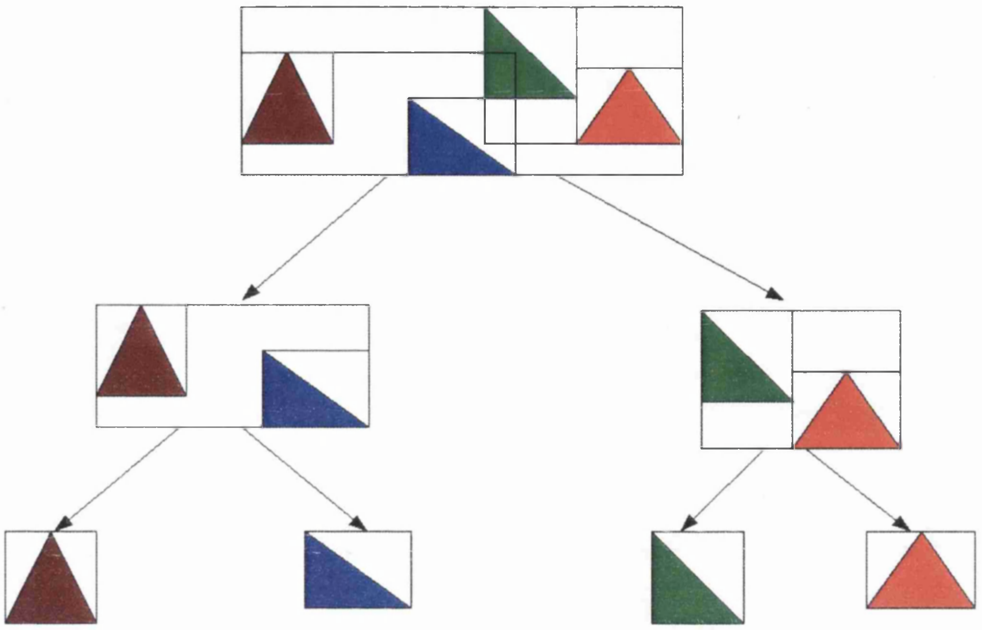


Figure 2.13: Bounding Volume Hierarchy. Each lower level consists of bounding volumes that enclose smaller parts of the model until the leaf node level where each bounding volume has a single primitive.

the BVH. This allows the bounding boxes to be closely aligned and reduces empty space in these boxes. The results show that it is effective for non-homogeneous scenes like the *Powerplant* scene. Another solution, proposed by Dammertz and Keller [DK08], is to split triangles along the longest side to create smaller triangles. A threshold that determines the number of triangles generated is determined by using a term called *edge volume* defined as the fraction of the volume of the bounding box. The resulting BVH shows significantly improved performance for problematic scenes without adding further triangles for scenes that do not benefit from subdivision.

2.3.5.3 Traversal

Depending on the shape of the bounding volume, the traversal of the BVH structure changes accordingly. For BVHs with spheres, the traversal consists mainly of intersecting the ray with the sphere for the particular node. Similary for AABBs, OBBs and slabs, intersection methods are used to determine whether a ray is to be traversed through a bounding volume or not. AABB based BVHs – the most popular BVH used – have a variety of intersection methods available as described in Section 2.2.4. One of the most frequently used intersection methods is the slabs method [KK86] which essentially intersects the ray with the component planes of the volume. It is a general solution that works irrespective of the orientation of the component planes. As applied to AABBs the six bounding planes of the box are intersected with the ray to determine if there is an intersection.

The slabs method has been optimised [WBMS05] [GM03] [BP04] [BP05] and is used by most recent ray tracers [DHK08] using BVHs. Although per node traversals are reasonably fast, it has the disadvantage of being slower than that of kd-trees. With the kd-tree, each traversal step involves only one ray–plane intersection. For BVHs, all the six planes of the box have to be intersected.

Another disadvantage of BVHs is that they cannot be traversed in a true front-to-back order. Consequently, early ray termination – used when a ray hits a primitive in a kd-tree leaf node – cannot be used. Since the child nodes are not ordered spatially, it is necessary to traverse all of them. Hence, for static scenes, kd-trees are better structures for ray tracing.

The traversal of the BVH can be described by the pseudocode below:

```
RecursiveRayTraversal(node)
{
  if(node is leaf node)
  {
    return closest object intersecting the ray
  }
  if(ray intersects first child node)
  {
    closestTrInFirstChild = RecursiveRayTraversal(first child node)
  }
  if(ray intersects second child node)
  {
    closestTrInSecondChild =RecursiveRayTraversal(second child node)
  }
  return ClosestObject(closestTrInFirstChild, closestTrInSecondChild)
}
```

Listing 2.5: BVH traversal. Ray has to descend down both branches (all branches if the tree has more than two branches) until the leaf nodes and select the closest triangle.

2.3.5.4 Advantages of BVHs

Although each step of a BVH's traversal is more expensive than that of a kd-tree, it has several advantages that make it competitive.

One advantage is that in a BVH, the objects are not duplicated. Hence, it is ensured that a ray intersects a particular object just once. In a kd-tree, an object may occur in several leaf nodes and consequently, if mailboxing is not used, a ray may have to compute the intersection with one particular object several times during the traversal process. Mailboxing, that eliminates these duplicate ray–object intersections does it at the cost of further overheads, making the advantages minimal. Hence, the property of a BVH that it inherently eliminates duplicate intersections is significant. This property also brings about a memory requirement advantage. Since objects are not duplicated in the leaf nodes, the memory footprint is much smaller than that of kd-trees.

Another advantage is that BVHs create a box at each sub-node, whereas kd-trees separate space only in one dimension. BVHs can, in theory, identify and separate empty space faster. This leads to trees that are shallower resulting in fewer traversal steps and lower memory requirements.

The major advantage of BVHs is realised when they are applied to dynamic rendering wherein geometry changes from one frame to the next. As described by Wald [WBS07], updating a BVH when a part of the geometry deforms is possible with relative ease. In extreme cases, a full rebuild maybe necessary. However, a full rebuild of a BVH would be faster than a kd-tree rebuild due to faster construction methods for BVHs.

Acceleration structures like grids, kd-trees and BVHs have been responsible for significant increase in ray tracing performance. Acceleration structures are one way of reducing the number of traversals and intersections per pixel. Although tracing a single ray through a well built BVH or a kd-tree performs reasonably well, in recent times, the observation that several rays follow a similar path through the tree has been used extensively to accelerate ray tracing. This method, called packet ray tracing, has enabled ray tracing to be competitive with the fastest rendering methods available for static scenes.

2.4 Packet Ray Tracing

Observation of the path a ray takes through an acceleration structure reveals that neighbouring rays take similar paths. In some cases, the neighbouring rays even intersect the same primitive. This property – called image coherence – has been used along with acceleration structures (that utilise object coherence) to accelerate ray tracing. This idea of image coherence is demonstrated well by Benthin [Ben06].

The use of image coherence to accelerate ray tracing has been attempted from very early on during the development of ray tracing algorithms and structures. The idea is to group several neighbouring rays into a packet and traverse them together through the structure.

Earlier attempts to utilise image coherence traced groups of rays with different boundary shapes. Heckbert and Hanrahan [HH84] traced beams – groups of rays with the actual primitives as boundary shapes – through the scene. When there was a high level of coherence in the scene, the beam tracer was shown to be much faster than standard ray tracers. They also showed that beam tracers could achieve anti-aliasing and further ray tracing effects like reflection and refraction – which can be a major problem with packet ray tracing. Shinya et al. [STN87] trace a group – or pencil – of rays using a paraxial ray and a system matrix to represent the group of rays. Shaft culling [HW91] that classifies objects as being inside or outside a shaft is another way coherence has been used to accelerate ray tracing. Pyramidal clipping by Zwaan et al. [ZRJ95] traces a pyramidal group of rays through a kd-tree and a grid by intersecting a convex polyhedron with a solid as given by Greene [Gre94].

It was determined that supercomputers with vector operations were well suited to ray trace packets of rays [PB85] as far back as 1985 when only advanced supercomputers of the time provided vector instructions. The idea has been very popular in recent times with the introduction of vector instructions for general purpose CPUs available in almost all current architectures. Wald [WBWS01] popularised this concept of tracing packets of rays using SIMD instructions. In addition to amortizing the calculations amongst the number of rays in the packet, Wald pointed out that cache and memory efficiency also improved with packet ray tracing as fewer nodes were accessed.

The SIMD instructions used were Intel's SSE instructions [SSE09b] [SSE09a] introduced in the Pentium III [Int08] processors. The introduction gave Pentium III and later processors eight SSE registers and several new instructions. The SSE registers were 128 bits wide allowing four floating point numbers to be in the register at a time. To complement this, the new instructions allowed these four floating point numbers in the registers to participate in arithmetic operations. As can be seen, this allows four floating point operations to be undertaken with a single instruction. Most of the instructions have performance comparable to the respective single floating point operation. Thus, the use of SSE instructions and registers are a good way to accelerate operations that can be parallelised – like ray tracing. Additionally, to ease development, the compilers provided intrinsics [VCI09] [Int09] that allow access to SSE instructions without the use of assembly code.

The use of SIMD instructions to trace packets is a brute force approach where all the rays are traversed through the tree. However, several other methods where only a few rays or representative boundary volumes of packets are traced have also been popular. The longest common traversal sequence [HB00] algorithm, stores the traversal history of a set of boundary rays and constructs the *longest common traversal sequence* (LCTS) from these. This LCTS is a set of nodes that each ray in the node traverses and thus the number of traversal steps can be reduced.

Beam tracing, introduced by Heckbert and Hanrahan [HH84], uses beams to intersect with primitives. However the fact that they do not use a hierarchical structure had results in relatively slow performance. Overbeck et al. [ORM07] recently published a similar algorithm that uses similar beams but traverses a kd-tree to show greatly accelerated performance. They start off with the entire viewing frustum as a beam and progressively sub-divide the beams according to the primitive boundaries. These beams are then traversed through the kd-tree using a frustum proxy method similar to the LCTS and the Multi Level Ray Tracing Algorithm [RSH05] (MLRT, MLRTA). The algorithm is extended for soft shadows and performance is shown to be much faster than previous methods.

The Multi Level Ray Tracing Algorithm [RSH05], traces a hierarchical beam of rays through a kd-tree constructed with parameters specialised to the algorithm to realise speeds that are almost real-time making it probably the fastest method of ray tracing. The entire scene is considered as a beam and by splitting these beams into tiles of various sizes depending on the nodes traversed, a hierarchical beam tree is built. Using this hierarchical beam tree, entry points deep inside the tree are found enabling a large part to be disregarded. An inverse frustum culling where the frustum is culled by the axis-aligned planes of the kd-tree is used to determine if a group of rays intersects an AABB or not.

Another important contribution of the paper is the application of interval arithmetic to perform packet ray tracing. Packet ray traversal with interval arithmetic works so that the traversal for the entire group can be determined using just one interval computation. The interval represents overestimates for the entire group and is a conservative computation. Partly due to this fact and partly due to reduced probability of the entire packet (all rays in a packet) hitting a node for larger packets, they cannot consist of a large number of rays. Reshetov et al. [RSH05] mention optimal packet sizes of 4×4. In our implementation, a packet size of 8×8 provided the best performance.

A very similar but slightly modified version of this traversal is used in our implementation. In order to trace a packet of rays, the rays with the earliest entry point and the latest exit point for each axis are selected. Using these six rays, the entire packet is traversed. If `rEntry[X]`, `rEntry[Y]`, `rEntry[Z]`, `rExit[X]`, `rExit[Y]`, `rExit[Z]` are the six rays, then the packet is traversed as shown by the following pseudocode:

```
char RecursiveRayTraversalInterval(node, tmin, tmax)
{
  if(node is a leaf node)
  {
    if(node is empty)
      return 0;
    ProcessLeafNode(nodeIndex);
    if(allRaysIntersected)
      return true;
  }
  axisCur= GetAxis(node);
  rayDirection = d[axisCur] > 0;

  temp = splitPoint[axis] - o[axisCur];
  tSpMin = temp*dRec[rEntry[axisCur]];
  tSpMax = temp*dRec[rExit[axisCur]];

  if(tSpMax > tmin)
  {
    allRaysIntersected = RecursiveRayTraversalInterval(node->leftNode, tmini,
         MIN(tmaxi,tSpMax));
    if(allRaysIntersected)
      return allRaysIntersected;
  }
  if(tSpMin < tmax)
  {
    allRaysIntersected = RecursiveRayTraversalInterval(node->rightNode, MAX(
        tmini, tSpMin), tmaxi);
  }
  return allRaysIntersected;
}
```

Listing 2.6: Recursive ray packet traversal algorithm using interval arithmetic. Algorithm computes ray-split plane intersection parameters of two boundary rays (as shown in Figure 2.14) and determines if the entire packet traverses the node or not.

As the pseudocode shows, the traversal method is similar to the single ray traversal. The only distinction is in the calculation of the t parameter and the traversal termination. Whereas only one ray is to be considered in the single ray version, the packet version uses the two rays that enter and exit the planes along the axis in consideration. It is to be noted that these rays are predetermined for the packet and hence no additional computation or determination is necessary during the traversal. The two rays, `rEntry[axis]` and `rExit[axis]`, are intersected with the split plane to get the two parameters, `tSpMin` and `tSpMax`. These parameters are substituted instead of the single t parameter used in the single ray version to determine the nodes to be traversed. Figure 2.14 shows the node traversal with further clarity.

Once the packet is traversed through the tree, at the leaf node the rays are decomposed into their individual components and tested against the primitives. However, to use SSE, the component rays in the packet are decomposed into groups of four and intersected against the primitives to generate the final image. Similar to single ray tracing, packet ray tracing can also use early ray termination with the difference being that all the component rays in the packet have to have found an intersection.

Packet ray tracing has brought about a major performance gain resulting in interactive to real

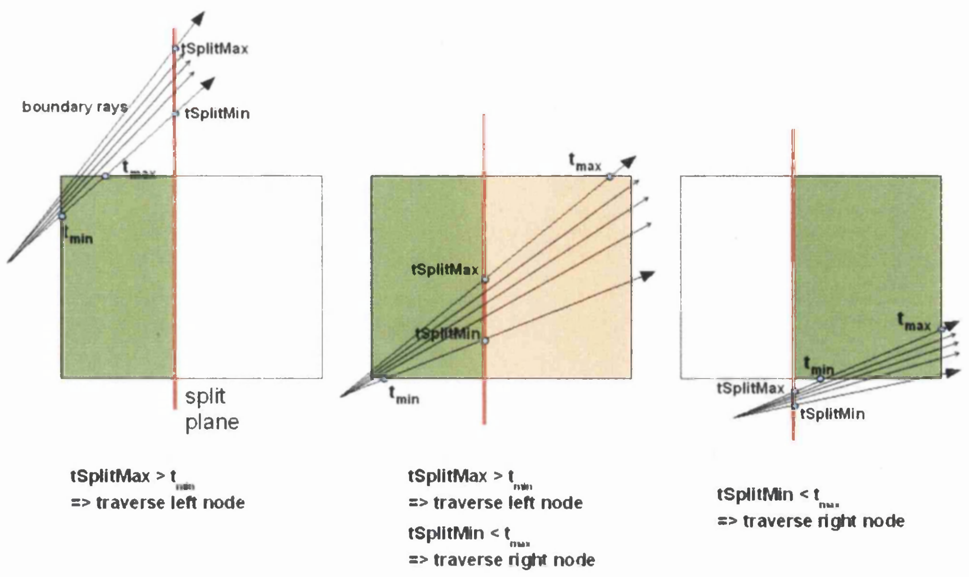


Figure 2.14: Kd-tree packet traversal.

time performance for ray tracing. However, a major disadvantage of packet ray tracers is that when the rays are not coherent, i.e., when the directions are not similar – frequently occurring for secondary rays like shadow, reflection and refraction rays, packet ray tracing cannot yield the same performance benefits as it can for primary rays. This problem is significant, as secondary effects are imperative for generation of high quality images.

2.5 Anti-Aliasing and Incoherent Rays

Since ray tracers trace only one ray per pixel, aliasing can occur at the edges of objects. There have been several solutions proposed to alleviate this problem. One of the simplest methods is to super-sample and trace more than one ray (4 to 16 rays) per pixel. This increases the resolution of the rendered image and significantly reduces aliasing artifacts. On the other hand, tracing several rays per pixel is computationally expensive. Hence, other methods have been attempted.

Adaptive sampling [Whi80] works by considering more samples at locations where aliasing can be most prominent. Rays are cast at the four corners of the pixel and if one or more intensities differ significantly from the others, then more rays are cast inside this area. Once the rays are traced, the weighted intensities are found and the pixel's colour is determined. This method adds further rays leading to additional cost.

Amanatides [Ama84] proposes representing a pixel as a rectangular area leading to the rays being pyramidal. However, for simpler calculations, approximating the pyramidal volume to a conical volume is proposed. Due to the fact that the conical ray covers a larger area of the pixel, the hard edges are reduced leading to a reduction in aliasing artifacts. However, tracing conical rays implies intersecting the acceleration structure and primitives with cones, which is expensive.

Another method proposed is stochastic sampling which adds random perturbations / jitter to the ray locations in a pixel [Coo86] to reduce the effects of aliasing. Several rays that pass through non-uniformly jittered positions inside the pixel are traced. The colour of the pixel is then determined by applying a resampling filter that calculates the value at the pixel. The idea borrows heavily from the workings of the human eye. Even though the eye uses a limited number of photoreceptors, it is not prone to aliasing. The method can be combined with super-sampling and adaptive sampling to reduce the effects of aliasing. Cook et al. [CPC84] use stochastic sampling for further physical effects such as motion blur and depth of field.

Although packet tracers with kd-trees are very fast, the coherence when tracing secondary rays like shadow, reflection and refraction rays is significantly reduced. The solution proposed is to return to tracing single rays, but optimizing the algorithm by making use of the SIMD instructions to traverse multi branching bounding volume hierarchies. QBVHs [DHK08] are BVHs that have four children at each node. These are constructed by collapsing a binary BVH to a quad BVH. This reduces the memory storage and memory bandwidth requirements by requiring fewer nodes. At the same time traversal is achieved using SIMD instructions that allow the same ray to be traced through the four child nodes simultaneously. In effect, this is opposite to SIMD packet tracing where four rays are traced through one node. The new *Larrabee* architecture from Intel [SCS$^+$08] proposes SIMD registers that are more than 4 wide, i.e., 16 wide, and hence multi-BVHs with more than 4 children have also been investigated [WBB08].

2.6 Dynamic Ray Tracing

Ray tracing dynamic scenes – i.e., scenes that change between frames is an area that is currently being heavily researched. With advanced methods available for ray tracing static scenes, the speed of ray tracing has reached interactive to real time performance. However, as discussed, most of this speed-up is due to the use of sophisticated data structures – mostly kd-trees built using the SAH. Unfortunately, construction takes a significant amount of time. Even though the creation of the SAH kd-tree has been achieved in $O(Nlog(N))$ time, it is still too slow if the tree needs to be created before rendering each frame. In addition, it is extremely difficult to update a kd-tree when triangles in a scene move.

Even then, there have been instances when kd-trees have been used for ray tracing dynamic scenes. Algorithmically, it was shown that kd-trees could be built in $O(Nlog(N))$ [WH06]. However, the constants associated with it are too high for dynamic rendering. Thus, approximation techniques where a few samples are used to find a reasonably good SAH split have been attempted [HMS06] [PGSS06]. In addition, due to increased popularity of multi-core processors, parallelizing the kd-tree building has also been attempted with two [Ben06] and four threads [SSK07] so that dynamic rendering could be achieved using kd-trees. However, in recent times, other structures have been used more frequently for dynamic ray tracing.

In a dynamic scene, either the entire structure has to be rebuilt from scratch at each frame or the data structure is built once at the beginning and updated as and when the scene changes. The second approach only works when the component triangles of the scenes do not change – i.e., triangles only move but triangles are not added or deleted from the scene.

If the data structure has to be rebuilt from scratch, the effective rendering time is the sum of the structure construction time and rendering time. It is generally believed that when more time is spent in creating a high quality structure, the rendering time decreases. Thus, an optimal structure would be one that is relatively easy to create and at the same time efficient to traverse.

One of the simplest data structures to create are grids and have been used for dynamic ray tracing effectively [WIK$^+$06]. Since grid construction is very quick, they can be very effective when used to ray trace dynamic scenes. An algorithm that is not based on the 3DDDA traversal algorithm is developed for grids, due to the inability of the 3DDDA algorithm to be effective when used with packets. The algorithm combines packet ray traversal, SIMD instruction usage and frustum culling. Instead of testing against a single cell of the grid, it intersects a slice of a grid with the packet to get all the intersected cells. In order to enable fast primitive intersection tests, the triangles are culled against the frustum formed by the packet. Mailboxing is used to ensure that intersection tests are not repeated. The combination of these methods enable grids to be used as an effective method for dynamic ray tracing.

Another class of structures that combines kd-trees and bounding volume hierarchies has also been investigated. These structures are called Bounding Interval Hierarchies (BIH) by Wachter and Keller [WK06] and Spatial kd-trees (S-kd-tree) by Havran et al. [HHS06]. Both these structures concentrate on creating kd-tree like structures but with faster construction. They use two split planes that effectively form a bounding volume to create a partition. The split planes fully enclose the primitives so that they are similar to bounding volume hierarchies. They are shown to be fast to create and traverse leading to an effective structure for ray tracing dynamic scenes. The differences between BIH and S-kd-tree are in that the BIH used a spatial median-like splitting method whereas the S-kd-tree used an SAH-like splitting method to identify good partitions.

In recent times, BVHs are also used to achieve dynamic ray tracing [WBS07] [LYMT06] [WMG$^+$]. Compared to kd-trees, BVHs are faster to create. In addition, it is relatively simple to update the BVH when a part of the scene moves. By applying packet ray tracing concepts developed for kd-trees, BVHs are shown to be an effective structure for dynamic ray tracing.

The vast research in ray tracing shows continued interest in new techniques to effectively undertake ray tracing. As a visibility determination method, ray tracing can be one of the methods used, and can be especially beneficial for very large scenes where it shows a logarithmic average complexity per pixel. However, rasterisation based methods are normally faster than ray tracing based methods, possibly due to simpler calculations and effective hardware implementations. A few of the more widely used and relevant methods of visibility determination in rasterisation will be described. However, since this thesis is mainly concerned with ray tracing like algorithms and structures, not all rasterisation based methods will be described.

2.7 Other Visibility Methods

Rasterisation, as defined by Hill [Hil00], is the process of taking high-level information like positions and colours of vertices to determine the colours of many pixels in the frame buffer. Visibility determination in rasterisation has been achieved using a variety of methods. In most scenes it is highly likely that there are several objects that overlap a pixel. The first object along the pixel occludes other objects. Thus, visibility / occlusion determination is very important to obtain accurate images. Since occluded objects do not contribute to the final image, they need not be rendered. Thus, by accurate and fast occlusion detection, performance can be improved by processing only the set of objects that are visible. A few methods to determine visibility [FvDFH90] [HB97] are by using area subdivision, scanline algorithms, Z-buffer, depth sorting, BSP trees, octrees, occlusion queries and *Hierarchical Occlusion Maps*. Some of these methods will be described in brief in the following sections.

2.7.1 Area Subdivision Methods

Area subdivision methods follow a divide-and-conquer strategy to determine visibility. Areas of the image are considered and if the polygons projected onto this area can be determined unambiguously, then they are drawn. Otherwise, the area is divided into smaller areas until the polygons can be unambiguously determined.

Warnock's [War69] area subdivision method divides each area into four smaller areas. At each stage, each area can be classified into one of the cases below.

- The area is surrounded by a single polygon – If there is a single polygon projection that completely surrounds the area, then the area can be filled with this polygon's colour. This is the polygon that is visible from all the pixels in the area being considered.
- Area contains or intersects a single polygon – In either case, the area is first filled with the background colour. If a single polygon intersects, then the intersecting part of the polygon is filled with the polygon's colour. If the polygon is contained by the area, the polygon is rendered.
- The area is disjoint from the polygons – The polygons have no effect on the area and hence the area is given the background colour.
- More than one polygon intersects, is contained by, or surrounds the area, but the closest polygon is a polygon containing the area – In this case, the area is given the colour of the closest surrounding polygon.
- More than one polygon intersects, is contained by, or surrounds the area, but a closest surrounding polygon cannot be identified – In this case, the area is further subdivided into four smaller areas and recursively tested for the above cases until either the determination can be made or until the pixel level is reached. If, even at the pixel level, the polygon cannot be identified, the Z values of the polygons at this point are calculated and the polygon with the closest Z value is selected.

While Warnock's method handled only polygons, Catmull [Cat74] introduces a subdivision method that handles curved surfaces. By subdividing the curved surfaces themselves into smaller patches, until a patch only covers a single pixel, the pixels occupied by the surface were identified. A noticeable difference between the two algorithms is that while Warnock's method subdivides screen space, Catmull's method subdivides the object.

While subdivision methods considered subdivided parts of the screen or objects, scanline algorithms processed the objects one scanline at a time. These methods have been quite popular due to their simplicity.

2.7.2 Scanline Algorithms

One of the earliest methods of visible surface determination was through the scanline method [WREE67] [Bou70] [Wat70]. Scan conversion is the process of converting a polygon from the world space to image space one scanline at a time. The methods described in the above papers are similar and the method introduced by Wylie et al., who generate images of objects created with triangles, will be described in brief. The view plane is considered as being made up of a series of scanlines. Initially, the triangles in the scene are projected to get their screen coordinates.

The vertices of each triangle are sorted according to the Y coordinate and Y-entry and exit tables are created. Using these, for every triangle, a Y occupied flag is maintained that indicates if a given triangle occupies a Y coordinate. Once this is done, the scanlines are considered one by one. For each scanline, the triangles intersected are obtained by checking the Y-occupied flag. The X-values of the intersection of the scanline with the projected triangle are calculated, sorted, and X-entry and X-exit tables are created. The individual pixels along the scanline are considered next. Similar to a Y-occupied table, an X-occupied table is created. If at a pixel, there is more than one triangle, the distances between the viewpoint and the triangle in world space are calculated and the closest triangle is selected to get the triangle for the pixel.

While the scanline method does not involve preprocessing, researchers have also investigated visibility determination methods whereby the scene is preprocessed. Two of these methods are the Depth Sorting method and the BSP tree method that essentially provide a priority to the objects in which they are to be rendered.

2.7.3 Visibility Determination by Depth Sorting

The depth sorting method was developed by Newell, Newell and Sancha [NNS72]. As described by Foley et al. [FvDFH90], it is a simple method wherein the polygons are first sorted according to farthest depth. When the depths overlap the polygons are split to resolve ambiguities. In the final step, the polygons are drawn back-to-front, i.e., according to decreasing depth. This determines accurate visibility by overwriting objects further away from the viewpoint with closer objects.

2.7.4 Visibility Determination using a BSP Tree

A BSP tree was used by Fuchs et al. [FKN80] to undertake visibility determination. They propose a new solution to the approach followed by Schumaker et al. [SBGS69]. The use of the BSP tree eliminates distance calculations to polygons. A scene containing polygons is taken as input and a binary tree – called a Binary Space Partitioning Tree (BSP tree) – is built. A simple building process is used whereby a splitting plane is selected and the polygons are classified as being on one side or the other of the splitting plane. If a polygon lies on both sides, it is split by the splitting plane. The first splitting plane is made as the root node. The process is recursively followed until each node has just one polygon. In order to determine visibility using this tree, a back-to-front traversal (achieved by an in-order traversal of the tree) determines the polygons that are written. The order ensures that polygons of lower priority are written before higher priority ones. In other words, polygons that are farther away are overwritten by polygons closer to the viewpoint to ensure that the right polygon is written to the right pixel. To handle one of the drawbacks – that there might be an increase in the number of polygons – a split plane that causes the least number of splits is selected.

The back-to-front traversal in the above method means that polygons cannot be skipped even if a closer polygon fully occludes farther polygons. To alleviate this, Gordon and Chen [GC91] traverse the BSP tree in a front-to-back manner. A scanline method is used to render the polygons and an auxiliary structure called the *dynamic screen* is used to identify areas of the screen that can be rendered over. The structure represents unlit (not yet rendered) pixels of each scanline. The pixels that the polygon can write to are identified by a merge process. This simple modification is shown to make rendering more efficient for scenes with a higher number of polygons.

2.7.5 Z-buffer

The Z-buffer method of visibility determination is a simple technique. An additional buffer with the closest Z value at each pixel is stored. As the rendering progresses, the Z-buffer is updated to maintain the Z value of the closest object at that point. When a new object is being rendered, if its Z value on a certain pixel is determined as being closer than the existing value, this object projects onto the pixel and the pixel's Z value is updated. It was first described by Catmull [Cat74]. Due to its simplicity, the Z-buffer has been one of the most popular methods to determine visibility. The simplicity also means that it is easily implemented in graphics hardware.

The main problem with the Z-buffer is that it can only determine visibility of a polygon at one pixel. This implies that each polygon that is being rasterised has to be processed to the single pixel level. It also means that a polygon cannot be determined as being occluded until it reaches the pixel level. This is a major disadvantage, as significantly larger number of polygons are processed in cases where occlusion is high. Compared to ray tracing – that processes very few occluded polygons – this approach to rasterisation appears very primitive. To alleviate some of the problems, hierarchical visibility methods have been proposed

2.7.6 Hierarchical Methods

Hierarchical Z-Buffer
Greene et al. [GKM93] introduce a technique called Hierarchical Z-buffer visibility. The technique described by them uses a ray tracing structure – an octree, and an adaptation of the traditional Z-buffer – a hierarchical Z pyramid. The Z pyramid is a hierarchical structure where the lowest level consists of the normal Z-buffer. Each higher level represents the farthest Z value of the four values it represents. Whenever a polygon is rasterised, the Z pyramid is updated to keep the values current. To further improve the speed, the octree nodes are rasterised and checked for visibility. If the octree node is not visible, then the polygons in it are also not visible and need not be processed. When the polygons have to be rasterised, the polygon is tested with the appropriate Z pyramid level by comparing the polygon's closest Z to the value in the pyramid. If it cannot be definitively answered, then the pyramid's next level is checked, again with the closest Z of the polygon. Although the closest Z of the polygon in the quadrant can be used, it is stated that this Z value is expensive to compute and hence the simpler approach is used.

Hierarchical Polygon Tiling with Coverage Masks
Another similar method is the use of coverage masks [Car84]. Coverage masks indicate areas of the screen that are covered by a polygon. Greene [Gre96] uses a modified version of coverage masks to accelerate visibility determination. The underlying idea of coverage masks is that for a given edge, all possible tiling patterns crossing a grid of samples can be pre-computed and later retrieved – indexed by the points of intersection. By *and*ing all the coverage masks of all edges of a polygon, the coverage mask for the polygon can be found. Similarly, by compositing all previously rendered polygons, the coverage mask for the image can be found. Greene modifies the coverage mask so that it indicates three states of the edge. A given sample is either inside the edge – indicated by a state of *Covered* in the mask, outside the edge – status of *Vacant* in the mask, or intersecting the edge – status of *Active* in the mask. These new masks are called *Triage* coverage masks.

Using these triage coverage masks, a coverage pyramid for the entire image is built. In the finest level, one bit coverage masks (indicating only *Covered* and *Vacant* states) are used. Thus, for a 512×512 image with 8×8 oversampling, the finest level of the pyramid consists of 512×512 one bit coverage masks. The higher levels of the pyramid consist of triage masks containing 64×64, 8×8 and 1×1 values. Each of the masks represent the corresponding area of the image. When a pixel is rendered, the lowest level of the mask is updated and the change is propagated upwards.

A BSP tree as described by Fuchs et al. [FKN80] is created and the polygons are processed in front-to-back order. Each of the polygons are rendered using a Warnock style subdivision method to make use of the logarithmic search properties of the subdivision method. At each step of the subdivision, the coverage mask is tested to see if the area being investigated is covered. The coverage mask of the polygon is found by using the pre-computed edge tables. By compositing the polygon's coverage mask and the corresponding mask in the coverage mask pyramid, it can be determined if the polygon is entirely hidden, entirely visible or its visibility is uncertain. When cells are entirely hidden, they can be ignored. If they are entirely visible, they can be displayed. When their visibility is uncertain, then they are recursively subdivided. Whenever a pixel is displayed, the coverage mask and the pyramid are updated. When all the polygons are recursively subdivided in a front-to-back manner, the image is generated.

To make the process faster, Greene also proposes to organise the scene into an octree of BSP trees. An octree is built on the scene in the first stage. In the second stage, at each leaf node of the octree, a BSP tree is built. First the octree is traversed in a front-to-back manner and the visible nodes are tested against the coverage masks to determine if they are visible. At a leaf node, the BSP tree is traversed in a front-to-back manner to render the polygons in the right order. By this process, only polygons with a high possibility of being visible are tiled, making optimum usage of coverage masks.

Hierarchical Occlusion Maps

Another hierarchical method to determine visibility is *Hierarchical Occlusion Maps* (HOMs) [ZMHH97] [Zha98]. HOMs are adapted for two of the algorithms described in the thesis – *Coherent Rendering* and *Row Tracing*.

An occlusion map is a gray scale image of the parts of the scene rendered. Using this, a pyramid or hierarchy of images are created. In this hierarchy, each pixel in each image represents a part of the image. The pixels indicate whether the corresponding area is occluded. As described, the benefit of HOMs over coverage masks is that HOMs are generated using graphics hardware.

Prior to rendering, the method selects a set of occluders that are an estimate of the objects that are likely to be visible. For static scenes, they are selected based on a few criteria like the size of the object, spatial locality (i.e., the bounding boxes are small compared to the size of the scene and the bounding boxes should have small aspect ratios – bounding boxes that are too big or ill-shaped would cause problems for the depth estimation), rendering complexity (simple objects with low polygon count) and redundancy (objects attached to other objects are not considered). For dynamic scenes, the occluders are selected at run time. One of the methods used is to select objects based on their distance from the viewpoint and until a particular object count is reached. Once occluders are selected, they are rendered and the occlusion maps are created.

To perform the occlusion test, the HOMs created earlier are used. There are two parts to the occlusion test. The first part tests if an object overlaps an area occupied by the occluders. If the first test is positive, the second part ensures that the depth of the object being rendered is greater than the occluders. i.e., the occluders are fully in front of the object.

The overlap test checks if the region occupied by the object overlaps an area occupied by the occluder objects. For this test, the area covered by the object is necessary. This can be achieved by projecting the object onto the screen. However, this is very prohibitive as the object may be quite complex. Hence, the bounding box's eight vertices are projected and the extent is used as the area to be tested. The traversal of the HOM is started at a level where the bounding box is enclosed by a pixel (corresponding length and breadth represented by the pixel is just greater than the bounding box's). If the pixel identified is opaque then the bounding box and consequently the object overlaps an area occupied by the occluders.

Once it has been identified that an object overlaps an area cumulatively overlapped by the occluders, it implies that the object may be occluded if it is completely behind the occluders. This is tested by a depth estimation buffer. The image area is divided into smaller areas and for each area, the farthest Z value is stored in the pixel. For each overlapped object, the buffer is tested. If the object's nearest depth is greater than the value in the depth buffer, it is occluded.

Coherent Hierarchical Culling
The Coherent Hierarchical Culling [BWPP04] method makes use of occlusion queries supported by recent graphics hardware. Hardware occlusion queries allow determination of whether a given object is occluded or not. The query takes the given object as input and the GPU returns the visible fragments of the object. The main problem, however, is that there is an associated latency. One kind of query is an NV_occlusion_query, an OpenGL extension introduced by NVIDIA on their Geforce 3 graphics cards. The NV_occlusion_query returns the number of visible pixels of the object. In addition, the NV Query also allows queuing queries before asking for the results. Bittner et al. aim to utilise this feature to improve visibility performance.

In the simplest method, a kd-tree is used and is traversed front-to-back. The bounding boxes of the kd-tree nodes are sent to the GPU to test for occlusion. The result is awaited and depending on the outcome, the tree is traversed. However, due to the latency of the query, this is not an efficient method. To overcome this, temporal coherence is used to reduce the number of occlusion queries. Also, the queries are issued and stored in a queue until done by the GPU. This allows interleaving occlusion determination instructions with instructions to show visible polygons. A breadth first traversal of the kd-tree based on a priority queue is also proposed to optimize utilization of occlusion queries. The nodes are prioritised according to the inverse of the minimal distance of the viewpoint to the node's bounding box. The use of temporal coherence is shown to be an effective method to accelerate visibility determination when a walk-through of a large model is attempted.

The above hierarchical methods are just a few of the methods that have been used to determine visibility efficiently in rasterisation. The popularity of hierarchical methods imply that they are useful for fast determination of visibility.

2.8 Summary

Due to the use of acceleration structures ray tracing has become a viable form of visibility determination / rendering. Acceleration structures enable ray tracing by intersecting with only a fraction of the primitives in a scene. Furthermore, intelligent structure creation strategies like the SAH have further improved the performance of ray tracing. The SAH, introduced initially for

BVHs, when adapted to kd-trees produces the best structure for ray tracing static scenes. Further performance benefits, especially for primary rays (relevant for visibility), have been brought about by making efficient use of image coherence (tracing a group of rays together). Through the use of SSE, four rays have been traced simultaneously. Groups of rays have also been traced using frustums or by using interval arithmetic. The introduction and development of packet tracing has culminated in ray tracing primary rays being considered almost a solved problem. Recent research, focusing more on incoherent rays and dynamic rendering has shown that intelligently built BVHs can be very competitive with kd-trees – even for static scenes. In addition, BVHs with their ability to be quickly built and partially rebuilt are believed to be a better structure for ray tracing dynamic scenes.

At the same time, several other visibility algorithms have also been developed. The Z-buffer algorithm is widely used due to its availability on most graphics hardware. Hierarchical methods combining ray tracing structures (kd-trees, octrees, etc) with hierarchical occlusion information (hierarchical Z-buffer, HOMs, coverage masks, etc) – are also widely researched.

In addition to algorithmic advances, recent hardware developments like SSE and multiple cores have enabled ray tracing – a highly parallelisable algorithm – to be a feasible alternative to rasterisation. The investigation of ray tracing structures and algorithms for visibility is thus believed to merit further consideration.

Chapter 3

RBSP Trees

Contents

This chapter introduces and studies a new structure based on kd-trees and BSP trees. The structure uses several arbitrarily aligned splitting directions to create a space partitioning structure that more closely wraps the scene. The structure is shown to reduce the number of node traversals as well as the number of triangle intersections and results in a more efficient structure for ray tracing. Part of the work in the chapter has been published as – *Kammaje, R.P.; Mora, B., "A Study of Restricted BSP Trees for Ray Tracing," IEEE Symposium on Interactive Ray Tracing, 2007. RT 2007., pp.55-62, 10-12 Sept. 2007.* [KM07]

Ray tracing is one of the most researched areas of Computer Graphics and the use of innovative data structures and algorithms has made it a feasible, even preferable, rendering method. However, due to the expensive nature of computations for each ray, single ray tracing is not very frequently used as a visibility method. The most frequent use of ray tracing is when realistic optical effects are desired. At the same time, it is also acknowledged that ray tracing is scalable, with an average complexity of $log(N)$ per pixel (where N is the number of primitives in the scene) [HB00] [Hur05] [WSS05] [HHS06] [YLM06] [WBWS01] as compared to a much higher $N \times s$ (where s is the average projection size (in pixels) per triangle) per pixel complexity of Z-buffer algorithms. Wald et al [WBWS01] show the linear and logarithmic complexities of Z-buffer based and ray tracing algorithms respectively. Due to this, and ray tracing's built in occlusion determination property – early ray termination, as the number of primitives in the scene increase, the complexity advantage of ray tracing manifests itself. For very large models a significant performance advantage over the brute force approach of hardware rasterisation based techniques may be obtained, even for visibility determination.

The complexity advantage of ray tracing is realised through data structures like kd-trees, octrees and BVHs. *Restricted Binary Space Partitioning trees*, henceforth referred to as *RBSP Trees*, are introduced as a new addition to this class of data structures. They are binary trees that subdivide space into two partitions at each step and classify the primitives into one of the two partitions. The structure attempts to achieve better ray tracing performance by combining advantageous concepts of general BSP trees and kd-trees.

3.1 Motivation

The primary motivation for the existence of *RBSP trees* is rooted in the concept of void area. As defined in [WHG84], void area is the difference in projected areas of the bounding volume and the actual model. For ray tracing, the void area indicates the space in which the rays miss the scene, but still have to be processed. Reducing the void area results in a decrease in the number of these rays, thus improving the efficiency of ray tracing.

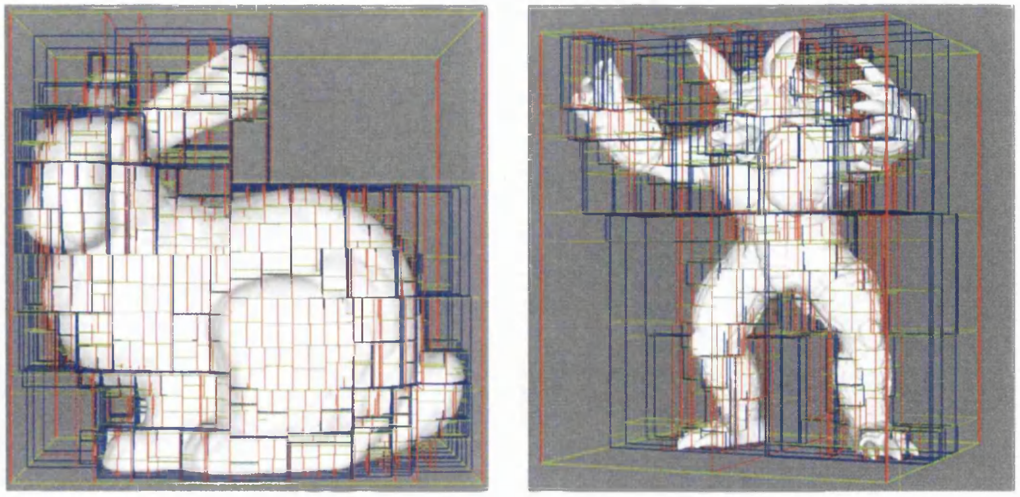
Although, the concept of void area is described for bounding volumes, it is equally applicable to space subdivision structures like the kd-tree. Each node of a space subdivision structure can be thought of as a bounding volume for the triangles enclosed by the node. Similar to bounding volumes, the number of rays that miss the triangles in the node, but still hit the node is given by the void area. Hence the concept, and the complimentary result pertaining to rays, is also believed to apply to space subdivision structures like kd-trees, octrees, and BSP trees.

It is also one of the reasons for the efficacy of the *Surface Area Heuristic (SAH)* for kd-trees. Although the SAH for kd-trees very effectively reduces the void area of kd-trees, it is limited by the kd-tree's property that splits can only be along the X, Y or Z axes. For scenes that predominantly consist of triangles that are axis-aligned (e.g. architectural scenes), the kd-tree's split axes are highly customised to the scene. Compared to general BSP trees, it also results in the structure being relatively easy and quick to build. This is due to the fact that the selection of a split plane requires significantly fewer SAH calculations.

Other kinds of scenes, in which the triangles are more arbitrarily aligned, expose the kd-tree's limitation. As Figure 3.1 shows, the difference between the projected areas of the kd-tree and the actual scene (i.e., the void area) is quite considerable even at lower tree depths. Having potential split axes that are arbitrary in alignment and number could potentially create a structure with a smaller void area. This is one of the main motivations for the introduction and study of the *RBSP tree*.

The *RBSP tree* can consist of split planes that are very close to the component triangles' alignment. As Figure 3.2 shows, compared to Figure 3.1, the *RBSP tree* has a considerably smaller void area than kd-trees. Thus, a study is warranted to explore the promise of better rendering performance provided by the use of *RBSP trees*.

Along with the concept of void area, Weghorst et al. [WHG84] also quantise the rendering time in Equation 2.11 which indicates that the rendering time varies according to four variables. A considerable increase in any of the four variables can significantly affect performance. On the other hand, efficient structures that reduce the number of bounding volume and primitive intersections can be very efficient. In addition to the number of intersections, the cost of this intersection is also a prominent variable with the possibility of increasing the ray tracing cost significantly. Hence, a



(a) Kd-tree for *Bunny* (b) Kd-tree for *Armadillo*

Figure 3.1: Kd-trees on scenes with predominantly non-axis-aligned triangles. Even though the SAH kd-trees converge to the model quickly, their void area is still significant. This is because kd-trees are restricted to using axis aligned splitting planes.

structure that reduces the number of intersections while being computationally cheap to traverse would be very efficient.

Determining a ray–kd-tree node intersection is computationally very cheap. However, prior to the introduction of the SAH, the kd-tree was not widely accepted as one of the fastest structures for ray tracing. The number of ray–node traversals and ray–primitive intersections with a non-SAH kd-tree was high. However, the SAH [MB90] – a heuristic that effectively reduces void area and separates empty spaces – changes this. The kd-tree thus built, results in a much lower number of intersections for ray tracing. The reduced number of intersections coupled with the cheap traversal cost realises the true potential of kd-trees – resulting in a significant performance increase.

BSP trees – in which the split axes can be aligned arbitrarily – are the general form of kd-trees. In its general form, the BSP tree has several disadvantages that make it unfeasible for ray tracing [FS88]. One among them is the cost of intersecting a BSP tree node. At each traversal step, the computationally expensive plane–ray intersection is necessary. However, if the other disadvantages were overcome, the advantage a BSP tree provides is that its splitting planes are selected from the scene itself. The BSP tree can, in theory, be a structure with the lowest void area possible. This would imply considerably reduced number of intersections that could, even with the increased traversal cost of BSP trees, be more efficient than kd-trees.

The high degree of difficulty in constructing a good BSP tree is one of the main drawbacks hindering it from being a feasible structure for ray tracing. The main advantage of the BSP tree – that its splitting planes are drawn from the scene – turns out be a major disadvantage in practice. The fact that there are as many potential splitting axes as there are triangles in the node being split means that the best splitting plane can be along any of these axes. The large number of potential split positions that need to be examined leads to very high construction costs. Another disadvantage of the BSP tree is that the splitting plane – indicated by four floats – has to be stored in its entirety at each node. The increased memory requirements brought about by this are significant – at least

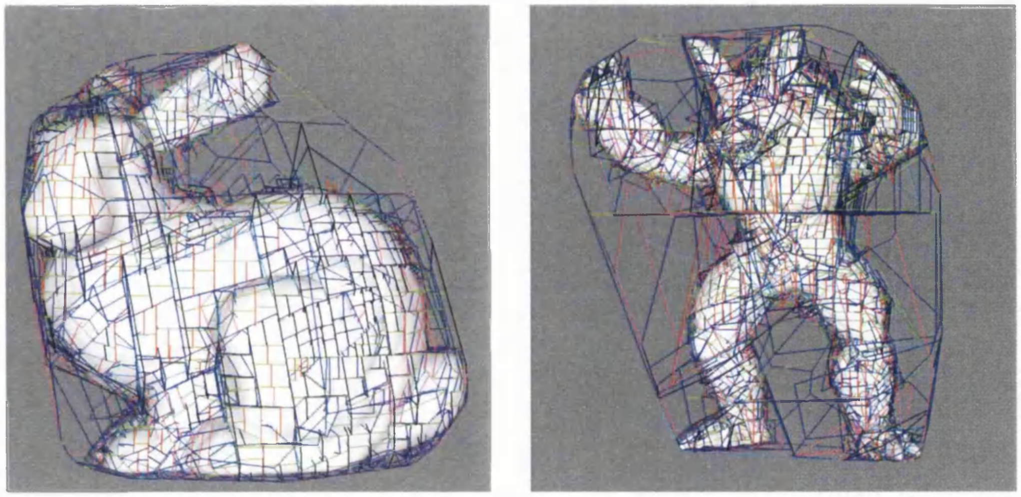

(a) *RBSP tree* for *Bunny* (b) *RBSP tree* for *Armadillo*

Figure 3.2: *RBSP trees* on scenes with predominantly non-axis-aligned triangles. The use of several additional splitting axes allows the *RBSP tree* to more closely wrap the model and minimise void area.

4× that required for a kd-tree node. It also leads to degraded performance due to increased cache misses. Because of these drawbacks, the BSP tree is generally not considered as a viable structure for ray tracing.

Irrespective of these disadvantages, the promise of reduced void area, resulting in a structure with significantly reduced intersection numbers, is hard to ignore. If a structure can combine the reduced intersection cost of a kd-tree with the reduced number of intersections of a general BSP tree while being simpler to construct, it could be the ideal structure for ray tracing. *RBSP trees* are an attempt at such a structure.

3.2 RBSP Trees Concept

RBSP trees are a specialised form of BSP trees in which the splitting planes are selected from a restricted set of predetermined planes that can be arbitrarily aligned. The reduced set of planes provides the opportunity to reduce the cost of intersection in addition to compact representation. In comparison to kd-trees, *RBSP trees* can have a wider selection of splitting planes available resulting in a reduced number of intersections.

The use of a structure with non-axis-aligned planes is not new to *RBSP trees*. Kay and Kajiya [KK86] investigate a structure that uses slabs of non-axis-aligned planes to build bounding volume hierarchies. However, bounding volume hierarchies are more expensive to traverse than space partitioning techniques. In addition, the limitation of hardware meant that they could only test it with a limited number of axes. It is also stated that additional axes would be very advantageous. The *RBSP tree* addresses these by being a space partitioning technique with a cheaper traversal cost and with the possibility of a larger set of splitting axes.

The splitting planes are predetermined as a set of axes, that are normals to the actual splitting planes. Subsequently, each splitting plane can be represented as a point on one of these axes. Further, if the minimum and maximum points along an axis for the scene are known, then any point along the axis between these two points can be represented by a discretisation enabling an easy and compact representation of the splitting plane. If the X, Y and Z axes are used as the set of axes for *RBSP tree* construction, a kd-tree is obtained. Thus, an *RBSP tree* is a more general form of a kd-tree.

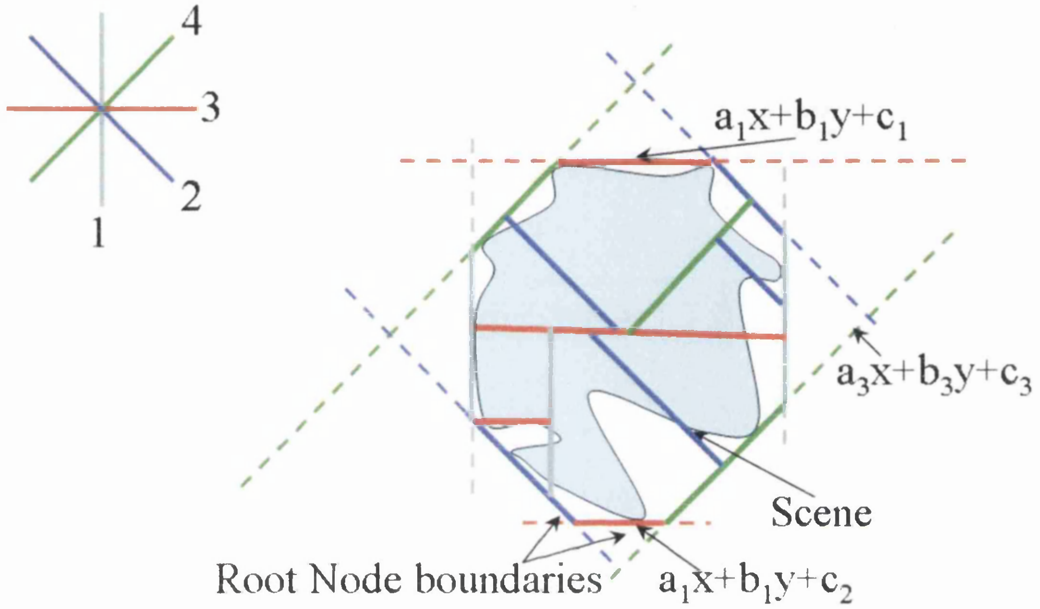


Figure 3.3: A 2D *RBSP tree*: In 2D, the splitting planes (lines) can directly be used instead of the normals (as done in 3D). Hence, the potential set of partitioning planes are shown (labelled 1,2,3,4). Using this set of axes, a tree that partitions space into two parts at each step is built.

Figure 3.3 shows a 2D *RBSP tree* and Figures 3.1 and 3.2 show kd-trees and *RBSP trees* built on the *Bunny* and *Armadillo* models. The figures reveal the potential of *RBSP trees*. For scenes that consist of predominantly non-axis-aligned triangles, the *RBSP tree* appears to reduce the empty space immensely. The splitting planes are more closely aligned with the planes in the scene.

The use of non-axis-aligned planes is not unique to *RBSP trees*. As described in Section 2.3.5.1, the structure developed by Kay and Kajiya [KK86] is one of the structures that uses non-axis-aligned planes. However, the structure proposed by them is a bounding volume hierarchy in contrast to the space subdivision nature of *RBSP trees*. This is an advantage for the *RBSP tree* as it needs just a linear interpolation or, at worst, a plane–ray intersection at each traversal step, whereas the BVH requires a ray–bounding-volume intersection at each traversal step. Klosowski et al. [KHM$^+$98] detail a similar structure with more planes but investigate its use for collision detection. Kay and Kajiya limit the number of plane normals used as they had limited hardware, however increasing the number of plane normals is mentioned to be beneficial.

The number of splitting planes that the *RBSP tree* can use is flexible, and theoretically not restricted to any number. However, as Section 3.4 will show, increasing the number of split axes significantly increases the construction time to obtain a good *RBSP tree* and thus becomes a limiting factor. In practice, it is necessary to restrict the number of split axes to around 24. Even so, this is significantly more than what Kay and Kajiya used and allows a range of alignments to be easily investigated.

In brief, the construction and use of the *RBSP tree* can be described as follows. Initially, a number for the split axes is fixed. Depending on the number, the actual split axes are determined, either manually or using some heuristic. Subsequently, the actual tree is constructed using planes along these axes. The scene is now ready for ray tracing using the constructed *RBSP tree*. To ray trace, the individual rays are traversed through the tree to find the closest intersection and the image is rendered.

An important feature of *RBSP trees* is the flexibility provided. *RBSP trees* can be used to simulate a kd-tree or a general BSP tree. They can be used both to accelerate ray tracing and also to study BSP trees. Due to the difficulty of constructing good BSP trees, *RBSP trees* can be used as a good substitute.

Geometrically, the *RBSP tree* has nodes formed by the intersection of a number of planes. Henceforth, since the number of splitting axes is variable, it will be represented by m. This implies that the number of planes is $2m$. Due to the planes being arbitrarily aligned, the nodes have a variable number of faces with a variable number of edges. Since m is the number of splitting axes used, the maximum number of faces that a node may possibly have is $2m$. However, a node need not have faces corresponding to all the planes. A given node may be missing either one or both the faces corresponding to a particular plane. This is an important distinction from a kd-tree where the nodes always consist of six rectangular faces and leads to several distinctions in the creation of the tree. The mid-point calculation undertaken during the space median heuristic is more involved to ensure that the mid-point of the two bounding points along the axis is inside the node. During the SAH process, the calculation of the surface area and the counting of triangles in a node is more complex. At the same time these geometric properties also ensure that the object is more closely wrapped by the node.

Although *RBSP trees* are conceptually simple, sufficient thought must be put into the implementation for it to be competitive with the extremely well researched kd-tree structure. The main parts are the data structure representation, construction heuristics and finally ray–structure traversal – each of which will be discussed in detail in the following sections.

3.3 Data Structure

Representing the *RBSP tree* in a compact and easy manner is important if *RBSP trees* are to be a viable alternative to kd-trees. It was noted that one of the main drawbacks of the general form of a BSP tree was its increased memory requirements due to the nodes' data. In order for *RBSP trees* to be an efficient structure for ray tracing, a compact representation is imperative. The *RBSP tree* has three pieces of information – information that is common to all the nodes, the nodes themselves, and a list of triangle pointers contained by all leaf nodes.

The header contains the common data for the tree and consists of the following information.

- List of Predetermined Split Axes – The axes determined prior to construction are stored in the header as an array of axes. The axes are normal vectors and are represented by three floating point values. Once the array is stored, the index of the axis in this array is used to indicate every node's split axis.

- Information needed only when loading or saving the tree.
  - Number of Split Axes.
  - Number of Nodes.
  - Number of Triangles – The number of triangle pointers contained by all the leaf nodes in the tree.

Compactness is an important attribute for representing nodes as it can affect the performance and memory requirement of the tree. In order to obtain the most compact representation, a node is represented by 8 bytes (64 bits). Table 3.1 indicates the exact structure of a node.

Represents	Bits used
A unique pointer to children	32
Index to split axis	16
Quantised value of split position	14
Leaf node flag	1
Unused bit	1

Table 3.1: *RBSP tree* node structure

The tree is stored as an array of nodes with each node pointing to its children through an index. Since the two child nodes are adjacent, only the index to the first node (ordered front-to-back along the axis used) is stored. The second child node is implicitly addressed by adding one to the stored index. Using the array form for representing the tree improves cache performance as the nodes and its children are closer in memory with this representation than if the pointer / tree representation were to be used. Also, it enables better memory efficiency through the use of implicit second child addressing.

With the child node pointer taking up 32 bits, it is important that the remaining data of a node is represented as efficiently as possible. Normally, the split position is stored using a single floating point value. However, this requires a further 32 bits which is prohibitive. The split position is the point along the axis selected where a plane that is normal to the axis splits the node into two parts. It is a point that lies between the extents of the parent node along this axis. Thus, if this distance between the end points of the node along the axis is discretised into a certain number of values, it is possible to use fewer bits to indicate the split point. For an *RBSP tree*, a discretisation of the distance into $2^{14} - 1(16383)$ points provides a good result – both in terms of memory usage and precision of the split point. Thus, the split point can be indicated in the node with just 14 bits. Although, there is an unused bit available to increase the accuracy of the discretisation, the current precision was found to produce good results and hence only 14 bits are used.

Since the header consists of all the potential split axes for the tree, the node's split axis can be indicated by just the index to the axis. Even though 16 bits are allocated for representing the index, the practical limit for the number of split axes is about 32 which can be represented by 5 bits. However, the only other piece of information needed in the node is a flag – requiring just one bit – to indicate if it is a leaf node. Hence, using 16 bits for the axes allows the data to be easily accessible.

When the construction heuristic determines that a node is a leaf node, the structure represents different information. The information stored in a leaf node is indicated by Table 3.2

Represents	Bits used
A pointer to the start of the triangle list	32
The number of triangles in the leaf node	16
Leaf node flag	1
Unused bits	15

Table 3.2: *RBSP tree* leaf node structure

The leaf node consists of a certain number of triangles. The triangles contained by all the leaf nodes are stored in a separate list. Each non empty leaf node has an index that indicates the first triangle contained by it. Together with the number of triangles, the index is used to obtain the triangles in this node. Figure 3.4 shows this diagramatically.

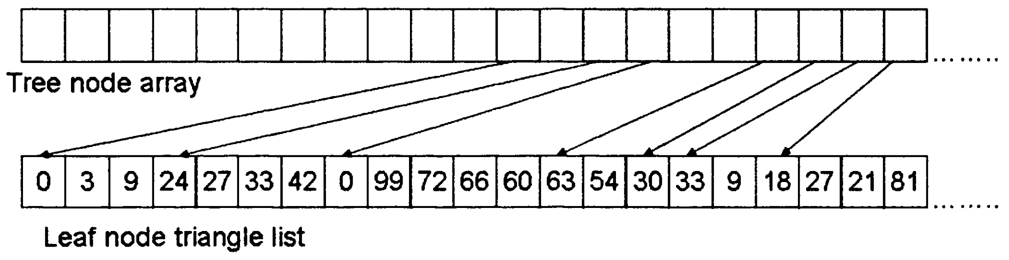


Figure 3.4: Leaf nodes and pointers to their component triangles. Nodes in green are leaf nodes that point to an index in a global list of leaf node triangles.

As shown, the data for a node (both internal as well as a leaf node) requires just 8 bytes. This is the same amount of memory required for a kd-tree node. Thus one of the claims of the *RBSP tree* – that it is compact to represent – is fulfilled. Having described the compact data structure to store the tree in memory, the construction process itself can be described.

3.4 Construction

The *RBSP tree* is a recursive structure in which every internal node can be considered as the root node of the corresponding subtree. A recursive tree construction process is thus the natural method to construct the structure. The following pseudocode describes the high level *RBSP tree* construction algorithm.

```
constructRBSPTreeMain()
{
  Vector[] splitAxes = findSplitAxesForTree();
  constructRBSPTree(root, allTriangles);
}
```

Listing 3.1: High level *RBSP tree* construction. First, the axes for the tree are selected and subsequently the tree is constructed in a recursive manner.

Before the actual construction of the tree, it is necessary to determine the directions of the potential split axes. One method to determine the axes automatically is to have them evenly distributed across space. This is achieved by a heuristic that uses evenly spaced points on a sphere [PSA07]. Once the number of axes that the *RBSP tree* will use is selected, the same number of evenly distributed points on a unit sphere with the origin as its center are found. The normalised vectors of the lines connecting the origin to these points provide a set of vectors that are aligned evenly across 3D space. These vectors are chosen as the potential split axes. However, due to the proliferation of scenes consisting of predominantly axis-aligned planes, it is desirable to include the X, Y and Z axes as potential axes. The axes that are closest in alignment to the three coordinate axes are replaced by the X, Y and Z axes.

The recursive *RBSP tree* construction algorithm, `constructRBSPTree`, is the same as the construction of a kd-tree, shown in Listing 2.3. The methods utilised for `findSplitAxis`, `findSplitPosition`, `findLeftTriangles`, `findRightTriangles` is where it differs from the kd-tree construction algorithm.

The methods `findSplitAxis` and `findSplitPosition` select an axis and a split position along this axis. These methods vary according to the heuristic used. On the other hand, counting and classifying the number of triangles on each side of the split plane is independent of the heuristic used. It is more involved than the process followed in the construction of the kd-tree, complicated by the fact that each node can have a variable number of polygons that themselves have a variable number of sides. A very straightforward method is used to achieve this. All the triangles in the node are clipped by all the component planes of the node to find the parts of the triangles that are contained by the node. Only triangles that have some part within the node are counted as being inside the node. Clipping the triangles ensures that each node is attributed only those triangles that have at least some part of themselves in the node.

The construction could be optimised by storing the clipped parts of the triangles for each node as the construction progresses down the tree. However, since the main aims are to investigate the advantages offered and to study the *RBSP tree*'s applicability for ray tracing, the construction process has not been optimised.

Initially, the extremal points of the root node are found for every split axis and stored as a list of points. The split point at which a node is split becomes the minimum point for one of the child nodes and the maximum point for the other. The list of node extremities are updated accordingly. This ensures that the bounding planes for the current node being processed is accurate. These bounding planes are the clipping planes for the triangles inside the node.

Two heuristics – Space Median and Surface Area Heuristic – to determine the split axis and split position are investigated. Both have been adapted from their well researched kd-tree versions and will be discussed in greater detail in Sections 3.4.1 and 3.4.2.

When the split plane for a node is found, the node is split and the triangles are clipped and classified. The process is continued down the tree until a termination criteria is met. When a node satisfies a termination criteria, it is considered as a leaf node. For the *RBSP tree*, only two termination criteria are used – the depth of the node in the tree and the number of triangles in the node. If the depth reaches a predetermined depth or if there are less than or equal to a certain number of triangles in the node, the node is made into a leaf node and the recursion stops. The number of triangles in a node at which it is made into a leaf node is fixed at 2. For the *RBSP tree*, the maximum depth is calculated as a function of the number of triangles in the scene. The term given by Havran and Bittner in [HB02] for kd-trees is used even for *RBSP trees*. It is:

$$d_{max} = k_1 log_2(N) + k_2 \tag{3.1}$$

where d_{max} – is the maximum depth of the tree
N – is the number of triangles in the scene
k_1 and k_2 – are constants chosen to achieve critical performance.

Havran and Bittner [HB02] use the values of $k_1 = 1.2$, $k_2 = 2$ that is also used for *RBSP trees*.

3.4.1 Space Median Construction

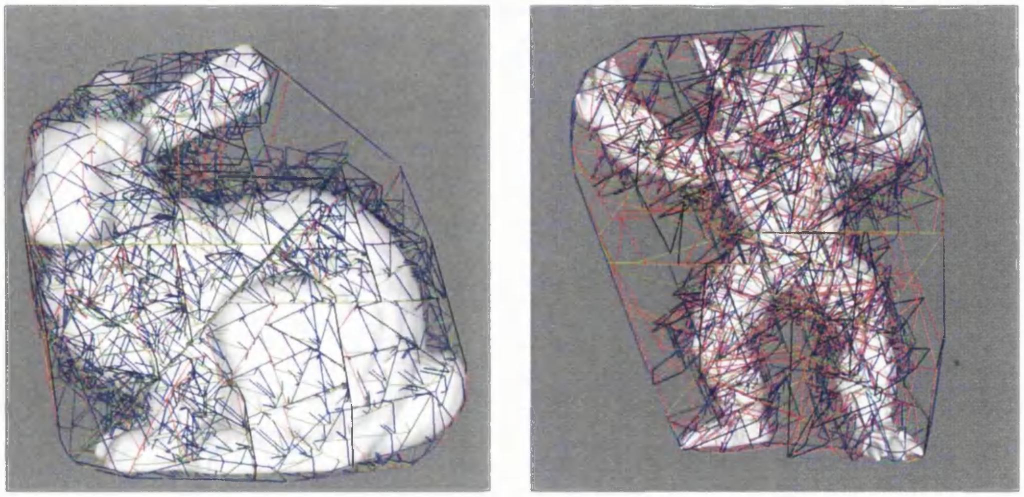
The simplest heuristic for building kd-trees is the space median, in which the axis is selected either on a round robin basis or the longest axis basis and the mid-point along this axis is selected as the split point. The same concept can be applied to *RBSP trees*. If there are m axes, then one of these axes is selected on a round robin basis. The mid-point along this axis is selected to obtain the split plane for the node.

Finding the mid-point along arbitrarily aligned axes is slightly more challenging. With kd-trees, the mid-points along the X, Y or Z are easily found as they are half of the sum of the minimum and the maximum value of the X, Y and Z coordinates of the node's bounding box. In comparison, to find the mid-point along the potential split axes, the bounding box's eight vertices are projected onto the axes to find the minimum and maximum projection point of the bounding box along each axis. The center point between these two extremities gives the necessary point along the axis to place the split plane.

Figure 3.5 shows that the space median construction results in poor trees, especially for the purpose of ray tracing. The reasons for this are quite obvious. The splits are aligned and placed arbitrarily. Due to the fact that a round robin scheme is being used for selecting the axes, there is no relation between the particular node and the splits. The problem is compounded by the split plane being placed at the mid-point of the axis with no consideration of the location of the triangles in the node. The poor quality of the trees created necessitate a better heuristic to realise the potential of multiple differently aligned planes.

3.4.2 Surface Area Heuristic

The *Surface Area Heuristic (SAH)* that results in kd-trees better suited for ray tracing is adapted to construct *RBSP trees*. It is based on the hypothesis that the probability of a ray hitting a node's triangles is directly proportional to the surface area of the node and the number of triangles in



(a) Space median *RBSP tree* for *Bunny* (b) Space median *RBSP tree* for *Armadillo*

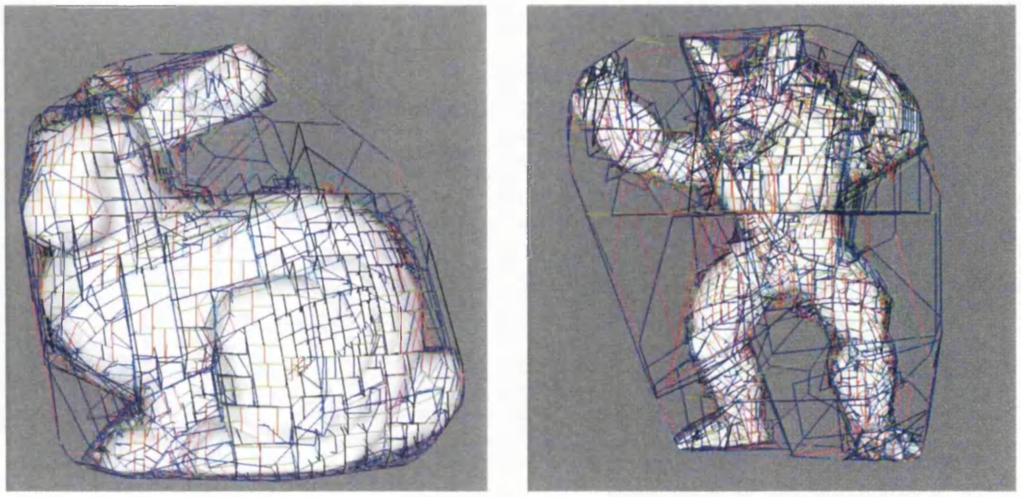
Figure 3.5: Space median *RBSP trees* on scenes with predominantly non-axis-aligned triangles. The split planes are not intelligently placed and hence the space median method to construct *RBSP trees* is not beneficial.

the node. This probability provides a cost function, given by Equation 2.12, that incorporates the probability of a ray hitting the node as well as the cost of intersecting the geometry in it. The axis and point at which the cost is a minimum provides the locally optimum split point for the node. Minimising this cost while selecting each split plane forms the essence of the SAH. The heuristic for kd-trees is very effective and increases the performance of ray tracing substantially.

For the *RBSP tree*, a heuristic like the SAH is even more important. It adds intelligence to the selection of a split axis and the placement of the split plane. As Figure 3.6 shows, *RBSP trees* built with the intelligence provided by the SAH are much better than those built with the space median heuristic. They have splits that are aligned and placed in relation to the contents and properties of the nodes.

Adapting the SAH to the *RBSP tree* construction is conceptually straightforward as Equation 2.12 can be used for the cost. However, calculating the cost is complicated by the necessity to examine several axes instead of just three.

The main purpose of the SAH, when applied to the *RBSP tree*, is to select the best axes among the numerous axes, and the best split point along that selected axis. All potential axes have to be considered as the optimum cost can be along any of the potential axes. There are an infinite number of potential split points along any axis. However as the SAH for the kd-tree does, the number of split points to be investigated can be limited to a finite number. Every triangle is first clipped by the bounding box of the node to obtain an accurate list of triangles contained in the node. Since the number of triangles can only differ at the two extremities of the clipped triangle along an axis, only these two points are added to the list of potential split points. The extremities along an axis are found by projecting the vertices of the clipped triangle onto the axis. By following this process for every triangle in the node, a list of possible split positions along the axis is found. Figure 3.7



(a) *RBSP tree* for *Bunny* (b) *RBSP tree* for *Armadillo*

Figure 3.6: SAH *RBSP trees* on scenes with predominantly non-axis-aligned triangles. Using the intelligence provided by the SAH, the *RBSP tree* constructed is of better quality.

shows the process with more clarity.

Once all the potential split points along an axis are determined, the SAH cost at each of these points is calculated. To calculate the SAH cost, the surface area of the left and right node created by a split plane placed at the point in consideration and the number of triangles in these nodes are to be determined. By using the list of projected end points, it is simple to determine the number of triangles on either side of a potential split point.

Computing the surface area of the two nodes created by the split is not as straightforward. This is compounded by the fact that the only values stored for a node are the extremities along each split axis. To calculate the surface area the node's actual polyhedron – found by clipping every bounding plane of the node (given by the split axis and an extremity along the axis) with every other bounding plane – is determined. The clipping process results in a list of polygons – the faces of the node's bounding volume. These faces are then clipped with the splitting plane in consideration to obtain the faces of the two split nodes that would be created with this split. Also, the splitting plane itself is clipped with each of the bounding planes to obtain the splitting plane's polygon. The polygons thus found can then be decomposed into triangles to calculate the surface areas of the two nodes potentially created by the split plane in consideration.

Plugging in the values for the number of triangles and the surface area in Equation 2.12, the SAH cost at each potential split point along each potential split axis is ascertained. The SAH also allows biasing the cost. Empty nodes are beneficial for ray tracing as they identify empty space allowing the rays hitting them to be skipped. This is considered in the heuristic for *RBSP tree* building by reducing the cost to 80% of the original cost if a potential split creates an empty node. The axis and the point with the minimum weighted cost is selected as the splitting plane for the node.

It is to be noted that while the SAH provides a method to create fairly good trees, the tree is not optimal. The cost calculated for each split is the local optimum that applies only to that node irrespective of earlier and future splits.

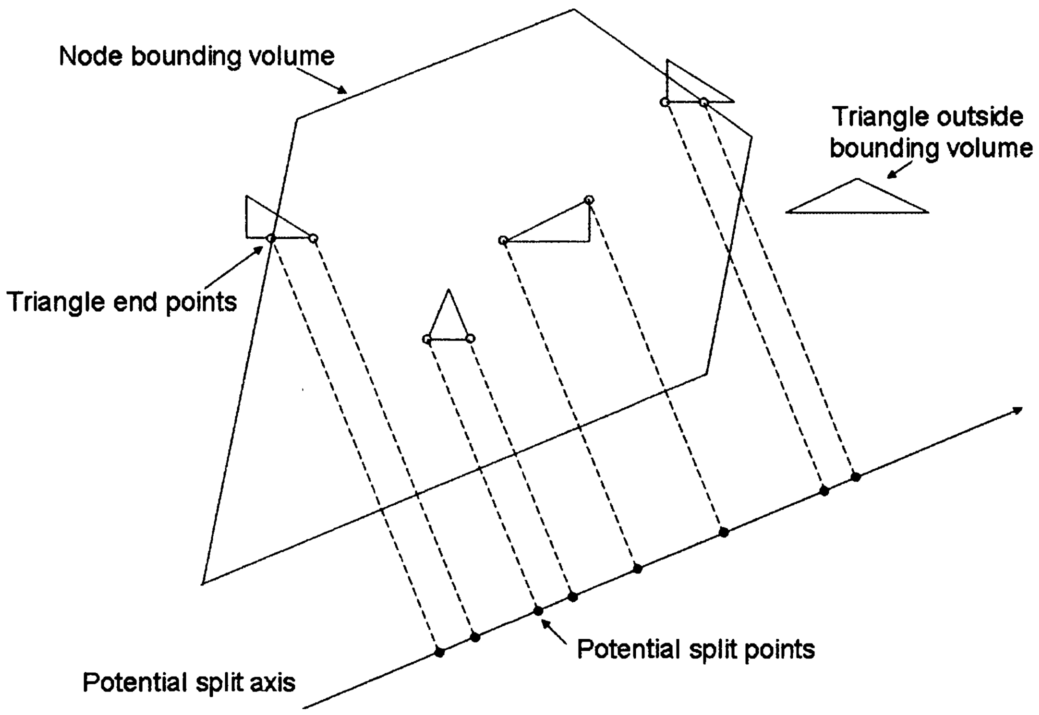


Figure 3.7: Potential split points for SAH along an axis. Projecting the end points of the clipped triangles onto the axis in consideration gives the potential split plane positions. The SAH cost at these potential points are calculated and the minimum point is selected as the locally optimal split position.

3.5 RBSP Tree Traversal

Constructing the tree prepares the scene for ray tracing. Each individual ray is traversed down the tree in a front-to-back order until it is determined if the ray misses the scene completely or intersects a primitive. When a ray hits a primitive, the pixel corresponding to the ray is shaded using the primitive's properties.

Using the parametric equation of the ray, given by Equation 2.1, the ray can be traversed through the tree using different methods – each with its own advantages and disadvantages. Each of these methods are described subsequently.

3.5.1 Algorithm 1.1 – Traversal by Linear Interpolation of Ray–Plane Intersection Parameter

This method for ray traversal is based on the kd-tree traversal [SS92] adapted to the *RBSP tree*. The high level algorithm can be given by the following two methods described in pseudocode.

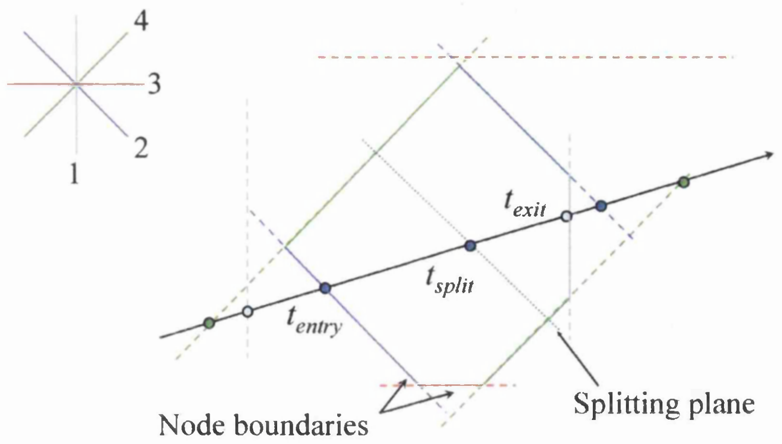


Figure 3.8: *RBSP tree* ray traversal (2D). Similar to the kd-tree traversal – if $t_{split} > t_{entry}$ the left node is traversed, and if $t_{split} < t_{exit}$ the right node is traversed. If both conditions hold, as in the figure, both nodes are traversed.

```
int rayTraverse()
{
  tMin = INFINITY;
  tMax = -INFINITY;
  //Compute the intersection parameters
  //to the entry and exit planes
  //along each axis
  for(each split axis index, i)
  {
    tEntry[i] = findTEntryBoundingPlane(i);
    tExit[i] = findTExitBoundingPlane(i);

    tMin = maxVal(tEntry, tMin[i]);
    tMax = minVal(tExit, tMax[i]);
  }
  if(tMin > tMax)
    return -1; // There is no intersection
               // with the bounding volume
               // of the node

  return recursiveRayTraversal(root, tEntry, tExit, tMin, tMax);
}
```

Listing 3.2: High Level *RBSP tree* traversal algorithm. The algorithm intersects the ray with each plane of the bounding volume of the tree. If there is no intersection, it terminates. Otherwise, the ray is recursively traversed down the tree.

```
int recursiveRayTraversal(RBSPNode node,
        float []tEntry, float []tExit,
        float tMin, float tMax)
{
  if (tMin > tMax       //ray does not intersect node
      or tMax < 0)      //intersection is behind origin
    return -1;
  if (node is a leaf node)
  {
    if (any triangle intersects the ray)
      return index of first triangle hit by the ray;
    return -1;
  }
  tSplit = findSplitParameter(node, tEntry, tExit);
  if(tSplit > tMin)
  {
    //Recursively traverse the left node
    triangleIndex=recursiveRayTraversal(node.left, tEntry, tExit,
                                  min(tSplit, tMin), tMax);
    if(triangleIndex != -1 )
      return triangleIndex;
  }
  if(tSplit < tMax)
  {
    //Recursively traverse the right node
    return recursiveRayTraversal(node.right, tEntry, tExit,
                               tMin, max(tSplit, tMax));
  }
}
```

Listing 3.3: Recursive *RBSP tree* traversal algorithm.

As the pseudocode shows, there are two main components that make up the tree traversal. The first step is to calculate the intersection parameters for the minimum and maximum bounding plane along each axis. Using these, the ray's entry and exit parameters, t_{entry} and t_{exit}, with respect to the bounding volume of the tree are found. If it is determined that the entry parameter is greater than the exit parameter, the ray does not intersect the tree and hence need not be processed further. Otherwise the ray is traversed down the tree.

The ray traversal begins at the root node and descends down the tree in a front-to-back order. At each internal (non-leaf) node, the ray's intersection parameter with the split plane of the node is calculated. It is important to consider the direction of the ray – indicated by the sign of the dot product between the ray and the split axis – during the calculation of the ray–split-plane intersection parameter, t_{split}. Since the entry and exit parameters along each axis has already been calculated, t_{split} can be quickly calculated by linear interpolation of the corresponding parameters. It is also to be noted that the node stores a discretisation of the distance between the entry and exit planes and hence has to be converted back to the actual value. The conversion back to actual values and the linear interpolation can be calculated as follows:

```
float findSplitParameter(node, float []tmin, float []tmax)
{
        tSplit = node.splitPosVal*(1/16383);
        //signRDirRec[axis] - Indicates ray direction along the split axis
        tEntry = signRDirRec[axis] ? tmin[i] : tmax[i];
        tExit = signRDirRec[axis] ? tmax[i] : tmin[i];
        return (tEntry + tSplit*(tExit-tEntry));
}
```

Listing 3.4: Listing shows the calculation of the ray–split-plane intersection parameter by linear interpolation.

It is to be ensured that the right values of t_{axis} and t_{axis} parameters are selected depending on the sign, or direction, of the ray. Another implementation detail is that the value of $(1/16383)$ ($2^{14} - 1$ as 14 bits are used to store the quantisation. See Table 3.1) is not calculated at each step, but is stored as a constant. Thus, the entire process of finding t_{split} is achieved with two multiplies, one addition and one subtraction.

As shown by Figure 3.8, the location of the split plane in relation to the node – mathematically represented by the value of t_{split} in relation to t_{entry} and t_{exit} – determines the nodes traversed. If the t_{split} parameter is greater than the t_{entry} parameter, it indicates that the split plane is located at a point along the axis that is after the point where the entry plane is located. In this case, the first node along the axis is traversed. If the split plane is located at a point before the exit plane, i.e., if $t_{split} < t_{exit}$, the ray hits the second node along the axis and the node is traversed. If both the cases are true, as in Figure 3.8, then the ray first traverses the first node. If it does not hit a triangle in the first node, the second node is traversed.

The tree traversal continues down the tree until a leaf node is reached. At a leaf node, the ray may hit one of the triangles contained in the node. As the triangles in the ray are not sorted according to the ray's path, it is necessary to test all the triangles for intersection. The intersection parameter is found for all triangles in the node using the method by Segura and Feito [SF01]. If the ray hits more than one triangle, the triangle with the lowest t parameter – the first triangle in the ray's path – is selected. On the other hand, the ray may miss all the triangles in the leaf node in which case, the tree traversal continues.

The method, as described, is a direct adaptation of one of the best kd-tree traversal methods. However, as the results show, the performance of this method depreciates when the number of splitting axes used is increased. This is despite the fact that, as expected, the number of node traversals and number of triangle intersections shows a consistent decrease. This depreciation of performance is attributed to the first method *RayTraverse*. Specifically, it can be traced to the calculation of entry and exit parameters for all the split axes. If m is the number of split axes used, then $2m$ ray–plane intersections are necessary. Calculating the entry and exit t parameters for each plane requires three dot products and one divide operation resulting in a large computational cost. Also, in cases where the depth of the tree is less than $2m$, the number of ray–plane intersections would be more than if the node were to be intersected individually. The *RayTraverse* method thus depends on m as much as it does on $log(N)$ (the order of the tree's depth).

To overcome this problem and realise the true potential of *RBSP trees*, alternate traversal methods are thought of and are described in the following sections.

3.5.2 Algorithm 1.2 – Traversal using SSE

SSE instructions allow four computations to be performed in parallel and are used to address the problem of decreased performance. The previously identified problematic part of the traversal – the initial entry and exit plane intersection calculation – is converted to SSE so that four entry and exit parameters are computed simultaneously in one iteration, thus reducing the cost of this process. The relevant part of the code in *RayTraverse* converted to SSE is

```
for(each split axis index, i)
{
  tMin[i] = findTEntryBoundingPlane(i);
  tMax[i] = findTExitBoundingPlane(i);
}
```

Listing 3.5: Part of code from RayTraverse that is computed using SSE.

It is changed so that every four split axes are handled in one iteration of the loop and can be written using SSE as shown below.

```
rayDirSSEX = load rayDirX into all 4 SSE components;
rayDirSSEY = load rayDirY into all 4 SSE components;
rayDirSSEZ = load rayDirZ into all 4 SSE components;

for(each 4 split axes starting at index, i)
{
  splitAxesX = X coordinate of the 4 split axes
  splitAxesY = Y coordinate of the 4 split axes
  splitAxesZ = Z coordinate of the 4 split axes

  tMinSSE[i/4] = findTEntryBoundingPlaneSSE();
  tMaxSSE[i/4] = findTExitBoundingPlaneSSE();
}
```

Listing 3.6: Using SSE to accelerate RBSP traversal. Instead of computing ray–split plane intersections individually, SSE is used to compute four entry plane and four exit plane intersections.

To obtain the best performance with SSE instructions, it is preferable to load the individual component coordinates of each ray and each split axis into separate SSE variables. With this, dot products require fewer instructions. Upon conversion to SSE, the calculations of the entry and exit parameters are achieved with fewer iterations and reduce the detrimental effect of increasing the number of planes. Results show that this method of optimising the traversal is very effective as long as the number of axes is relatively small (between 12 and 16). The advantage of SSE cannot mask the computational cost for larger number of axes. However, the results do show that that method realises the potential of *RBSP trees* better and achieves improved performance over an SAH kd-tree.

3.5.3 Alternate Traversal – Ray–Plane Intersection at Each Node

The performance of the two traversal methods described earlier is dependent on the number of split axes used. However, if possible, it is preferable for the traversal performance to be independent of the number of axes. To achieve this, the ray–split-plane intersection parameter calculation uses the ray–plane intersection, as shown in Listing 3.8, instead of linear interpolation used by the earlier method, as shown in Listing 3.4, to obtain the intersection parameter. Although the performance is worse for *RBSP trees* constructed with a small number of split axes, it makes the rendering time a function of the depth of the tree rather than the number of split axes. This method would thus be very useful to study and understand the effect of number of split axes on *RBSP trees*. The new high level algorithm can be given as:

```
int rayTraverse()
{
  //Determine two planes of bounding volume
  //at which the ray enters and exits
  findEntryPlane(entryPlanePoint, entryAxis);
  findExitPlane(exitPlanePoint, exitAxis);

  //Find the entry and exit
  //ray-plane intersection parameters
  tEntry = findRayPlaneIntersection(ray, entryPlanePoint, entryAxis);
  tExit = findRayPlaneIntersection(ray, exitPlanePoint, exitAxis);

  if(tEntry > tExit)
    return -1; //There is no intersection
               //between the ray and the
               //node's bounding volume

  return recursiveRayTraversal(root, tMin, tMax, tEntry, tExit);
}
```

Listing 3.7: High level algorithm for Alternate RBSP traversal. In this method, the entry and exit planes are determined using either the OpenGL method (described in Section 3.5.3.1) or the recursive divide method (described in Section 3.5.3.2). Code in blue shows the modifications compared to the earlier high level algorithm (given by Listing 3.2)

Compared to the high level algorithm described in Section 3.5.1, the main difference (shown in blue in Listing 3.7) is that alternate methods are necessary to determine the entry and exit planes, as we do not want to compute ray–plane intersections for all $2m$ planes. Consequently, the calculation of the ray–split plane intersection parameter, `tSplit` cannot be done using linear interpolation and is now achieved using a ray–plane intersection as shown by the pseudocode below:

```
float findSplitParameterPlaneRayIntersection(node)
{
  //Get the split position of the node along the axis
  tSplit = node.splitPosVal*(1/16383);
  //Find actual split point using linear interpolation
  splitPoint = bvPointFront[axis]
                   + tSplit * (bvPointBack[axis] - bvPointFront[axis]);
  //Intersect the ray with the plane
  //(plane is given by the normal and the splitpoint)
  tSplit = findRayPlaneIntersection(ray, splitPoint, planeNormals[axis]);
  return tSplit;
}
```

Listing 3.8: Calculating the ray–split plane intersection by using a full ray–plane intersection. This is necessary as linear interpolation of the ray cannot be performed as starting and ending points have not been calculated earlier.

Other than the computation of the `tSplit` parameter, the ray traversal is the same as shown by Listing 3.3.

3.5.3.1 Algorithm 2.1 – Entry and Exit Plane Determination with OpenGL

It is not straightforward to determine the entry and exit planes of the root nodes without intersecting the ray with each bounding plane of the root node. The first alternative is to employ OpenGL to find these planes. OpenGL or rasterisation methods are very quick if the number of rendered triangles is very small. The faces of the root node are found by clipping every bounding plane against every other bounding plane. Each face is assigned a unique colour for identification. The entry planes and exit planes are separately rendered with OpenGL. For the particular pixel being ray traced, the entry and exit planes can be determined by checking the pixel's colour in the rendered images. Figure 3.9 shows the entry planes being easily identifiable by their colours.

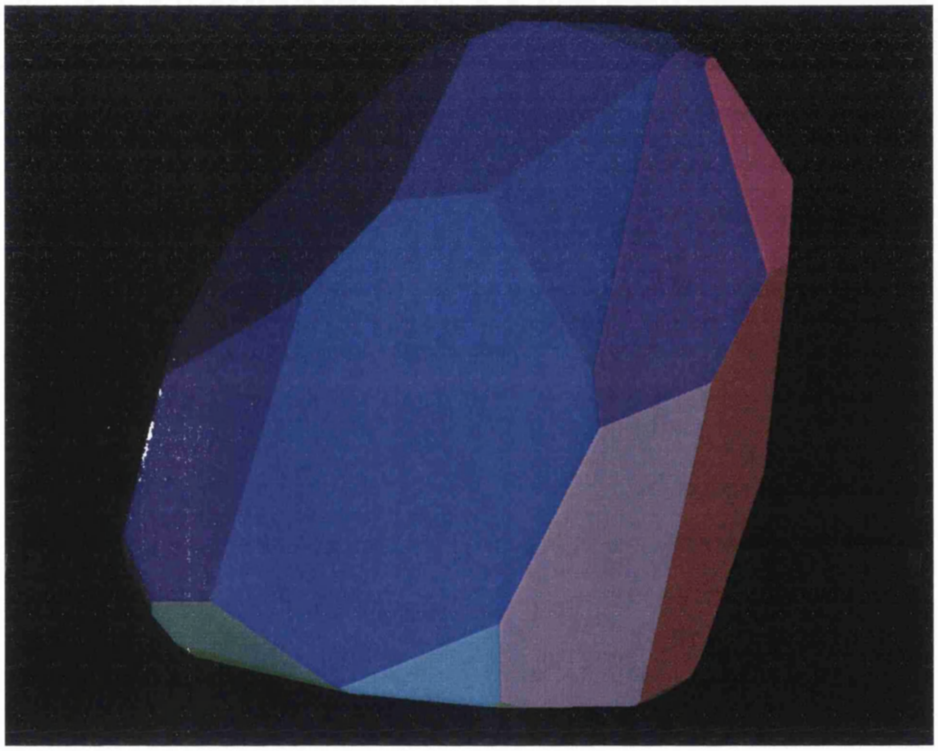


Figure 3.9: *RBSP tree* root node for the *Bunny* with each plane coloured differently.

Once the entry and exit planes are determined for a particular ray (pixel), the ray–plane intersection is used to calculate t_{entry} and t_{exit} for the ray. The method is very quick and the projection of the bounding planes is performed just once at the beginning of ray tracing. The method works very well for scenes where the viewpoint is always outside the bounding volume of the root node. When the viewpoint moves inside the scene, the entry planes are behind the viewpoint causing OpenGL projection to fail. Hence, an alternate method is necessary.

3.5.3.2 Algorithm 2.2 – Entry and Exit Plane Determination by Recursive Divide

A recursive process is formulated as a solution to the problem of finding the entry and exit planes. The method uses the property that in a convex polyhedron, if four boundary pixels of a rectangular region lie on the same face, then all the pixels that lie within these four pixels also lie on the same face. The process to find the entry and exit planes is very similar and hence only the process to determine the entry plane is described in greater detail.

```
determineEntryPlane(RectImageRegion r)
{
  if(r does not contain any pixels within)
    return;
  //Find the entry plane indices for
  //the four boundary pixels of r
  int entryPlane1, entryPlane2,
      entryPlane3,entryPlane4;

  entryPlane1 = findEntryPlane(r.corner1);
  entryPlane2 = findEntryPlane(r.corner2);
  entryPlane3 = findEntryPlane(r.corner3);
  entryPlane4 = findEntryPlane(r.corner4);

  if (entryPlane1 == entryPlane2 and
      entryPlane2 == entryPlane3 and
      entryPlane3 == entryPlane4)
  {
    //Set the entry plane indices of
    //all the pixels inside r
    setAllEntryPlanes(r);
    return;
  }
  else
  {
    rectImageRegion r1, r2, r3, r4;
    splitRegionIntoFour(r,r1,r2,r3,r4);
    determineEntryPlane(r1);
    determineEntryPlane(r2);
    determineEntryPlane(r3);
    determineEntryPlane(r4);
  }
}
```

Listing 3.9: Determining the entry plane for a group of rays using a recursive process.

The entry plane of the four corner pixels is calculated by finding the intersection of the ray with all the bounding planes of the node. If they are the same, then each ray that corresponds to a pixel within this rectangular region has the same entry plane. When the entry planes are different, the region is split into four smaller rectangular regions and a similar process is recursively followed until the entry planes of all the pixels have been found. It may appear that a significant number of entry and exit planes have to be found. However, the entry and exit planes of very few pixels need to be determined and this method is found to be very efficient. It is slightly slower than the OpenGL projection method, but is better than calculating all the entry planes for all the rays. It is also accurate for all viewpoints of all scenes and is therefore preferred. Figure 3.10 shows a few steps of the recursive method.

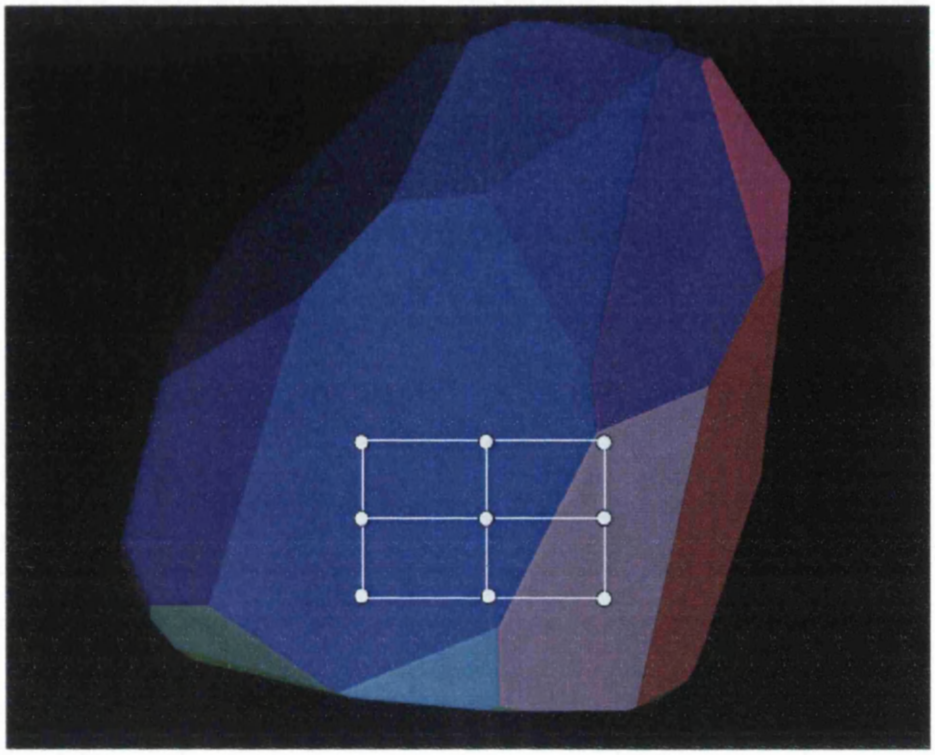


Figure 3.10: *RBSP tree* bounding volume / root node of the *Bunny* with each bounding plane coloured differently illustrating the recursive divide method. If the four corner pixels of a rectangular region have the same colour, then all the pixels inside this region have the same entry plane. Otherwise, the region is subdivided into four smaller regions and the four corner pixels are tested again.

3.6 Data Structure Visualised for Various Models

To visualise the data structure as built on various models, the non empty nodes at a particular depth are shown. Leaf nodes that are at this or higher depths are also shown. This visualisation enables identification of nodes traversed by the rays at the given depth.

Figure 3.11 shows this visualisation for *RBSP trees* built on the *Bunny*. The non empty nodes and leaf nodes are shown at various depths to show the traversed nodes. It is clear that trees built with more splitting axes more closely wrap the model, indicating that fewer rays traverse through each level as the number of splitting axes rise.

For a few other models, the same (non empty and leaf) nodes are shown in Figure 3.12, but only at a depth of 16. The figure shows a similar result to those seen in the *Bunny*. As the number of axes increase, the tree more closely wraps the model resulting in a reduction in the number of rays that traverse through the structure. This points to reduced node traversals and triangle intersections to ray trace the image. This fact is corroborated by the results in Section 3.7.

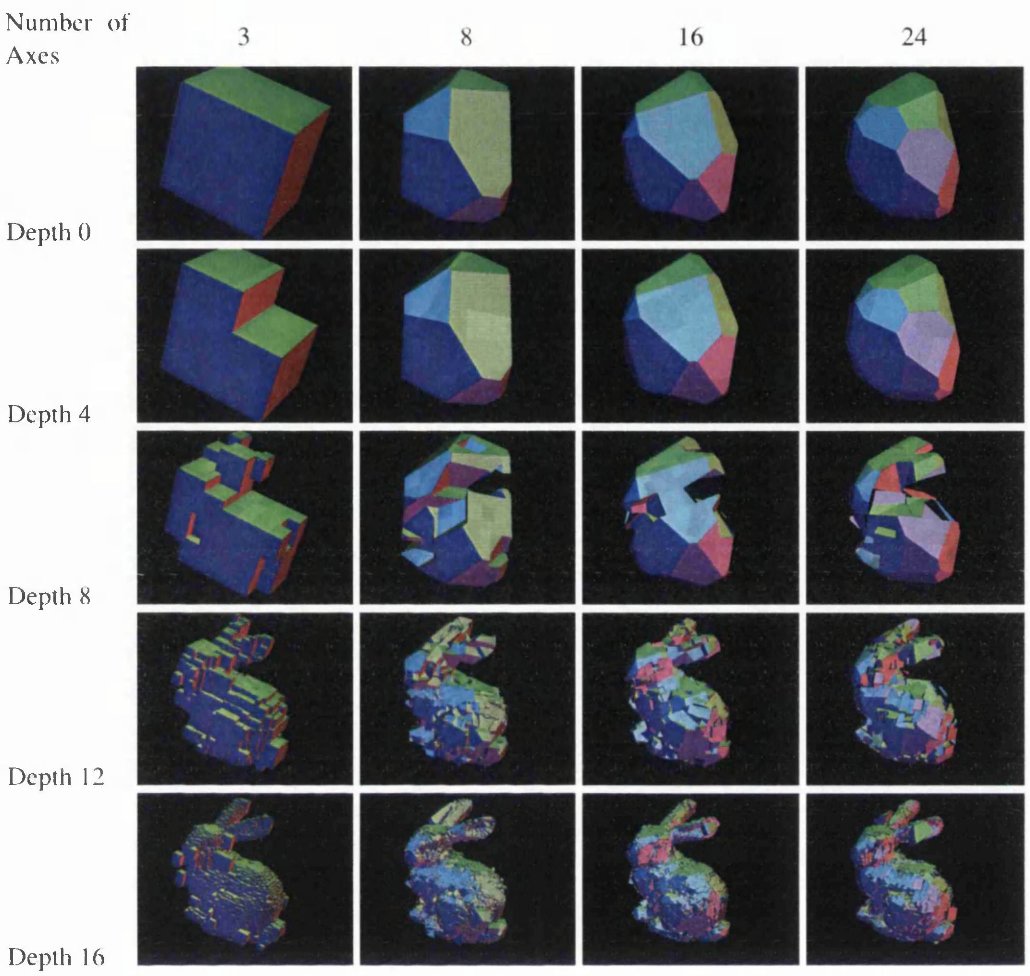


Figure 3.11: *RBSP tree* on the *Bunny* using 3,8,16 and 24 planes at various depths. Images show the non-empty nodes at the given depth and leaf nodes at higher depths. The visualisation shows that *RBSP trees* built with more axes converge to the model more quickly than those built with fewer axes.

3.7 Results

The aim of the new data structure is to enable efficient rendering performance by reducing the number of node traversals and the number of triangle intersections. Thus, these numbers in addition to the rendering times using the various traversal methods described will be presented in this section.

To accurately represent real world performance, several scenes are ray traced using the *RBSP tree*. For each scene used, several viewpoints are chosen and the images are generated from this viewpoint. Table 3.3 shows the models and the viewpoints used.

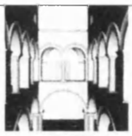
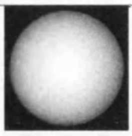
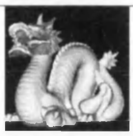
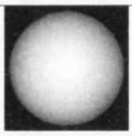
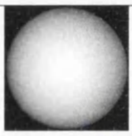
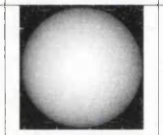
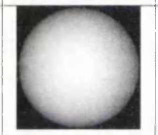
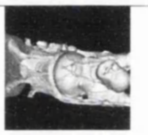
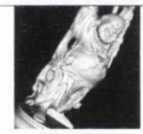
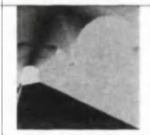
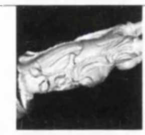
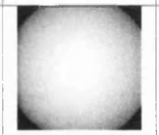
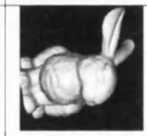
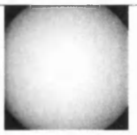
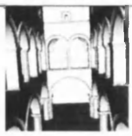
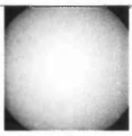
Scene	Sponza	Bunny	Armadillo	Dragon	Sphere	Happy Buddha
Views						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

Table 3.3: Scenes and viewpoints used to benchmark *RBSP trees*.

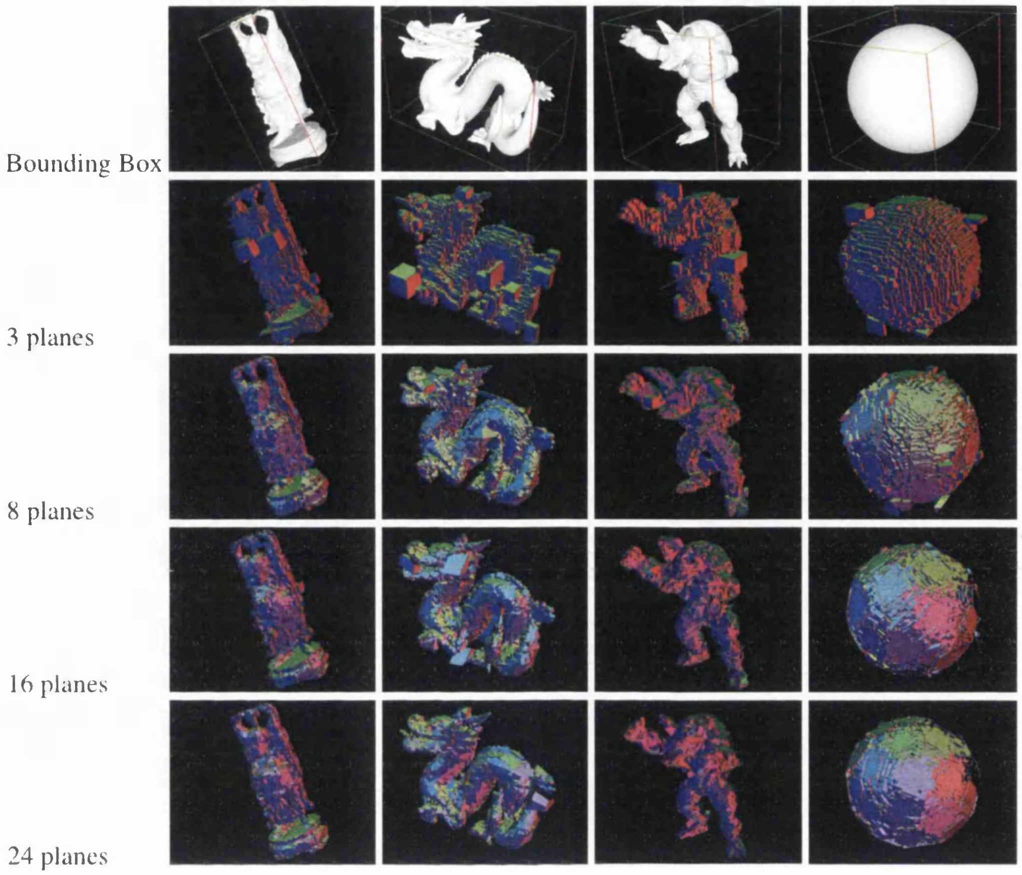


Figure 3.12: Images showing the bounding box and the *RBSP tree* using 3,8,16 and 24 planes for few other models. Images show the non empty nodes at depth 16 and leaf nodes at higher depths. As more axes are used to build the *RBSP trees*, their quality improves.

3.7.1 Node Traversals and Triangle Intersections

The graphs in Figures 3.13 and 3.14 show the number of node traversals and triangle intersections respectively for various models using *RBSP trees* built with varying numbers of split axes. *RBSP trees* with 3, 4, 8, 12, 16, 20 and 24 planes have been built to compare the effect of using variably aligned and numbered splitting planes. For non-axis-aligned scenes, the advantage provided by *RBSP trees* is very clear. As the number of splitting axes rise, there is a reduction in the number of node traversals and triangle intersections. Section 3.6 visually demonstrated the reduction in void area of *RBSP trees* as number of axes used to build them increase. The figures provide statistical support by showing reduced intersection and traversal numbers. If this reduction is translated to performance terms, *RBSP tree* would be a very good data structure for ray tracing.

They also reveal that it is not a very good structure for scenes consisting of predominantly axis-aligned triangles. The availability of differently aligned planes causes selection of splitting planes that are not closely aligned with the triangles. This aspect reveals a problem with the split plane selection heuristic – the SAH. It also suggests the suitability of kd-trees for such scenes as the

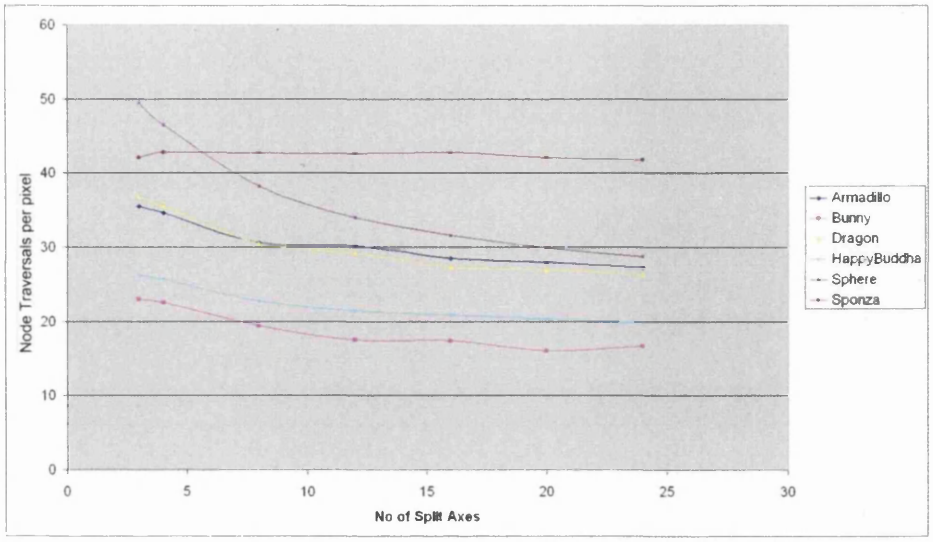


Figure 3.13: Variation of node traversals per pixel over number of split axes used to build the *RBSP tree* for ray tracing several models. The more the number of axes, the lower the number of node traversals (except for the *Sponza* scene).

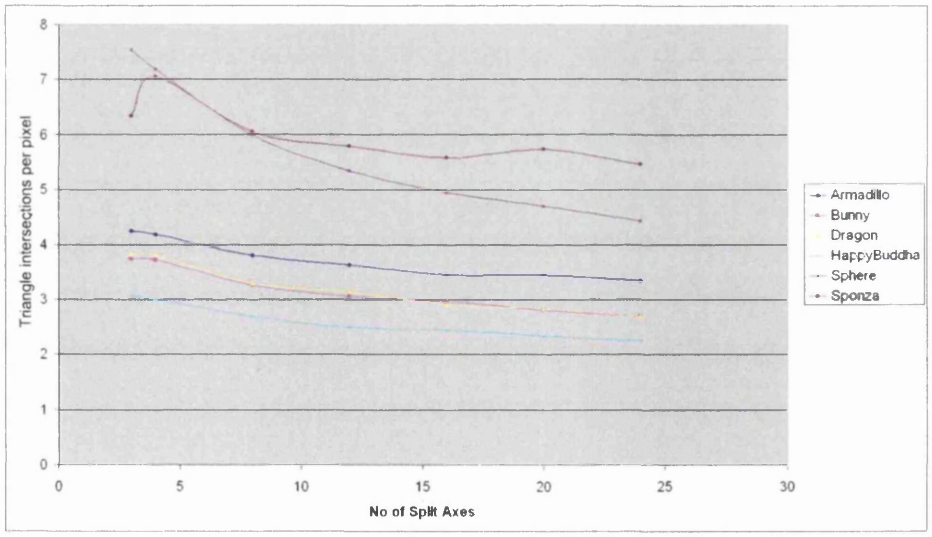
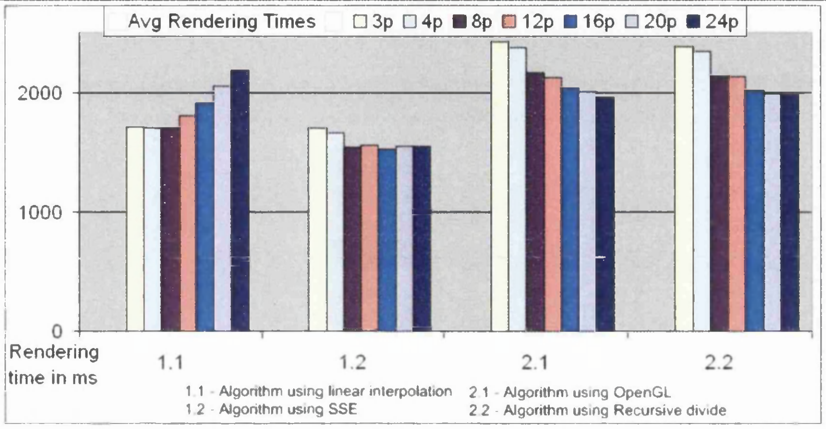
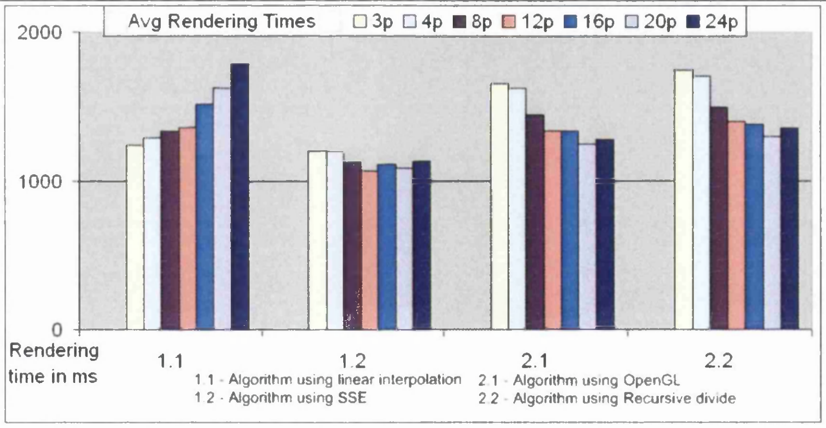


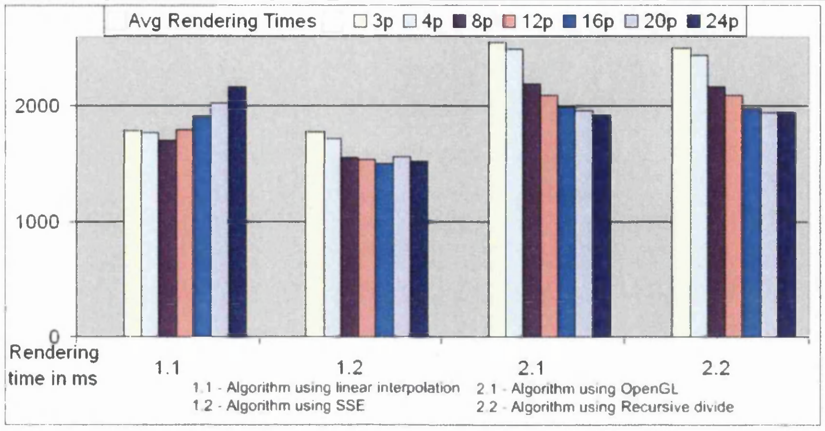
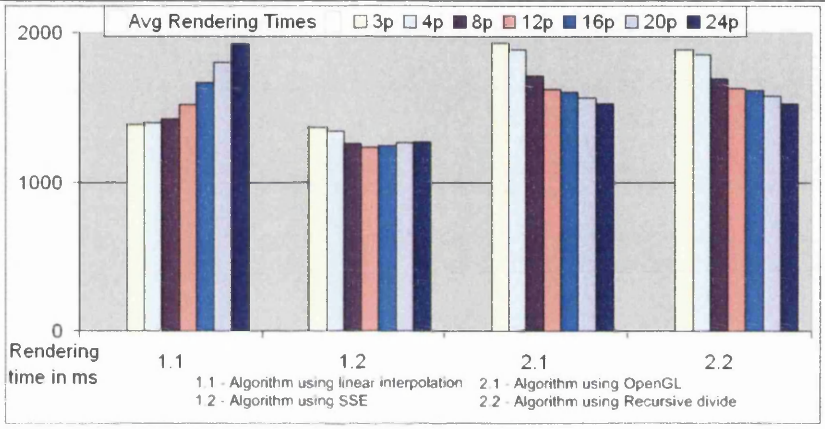
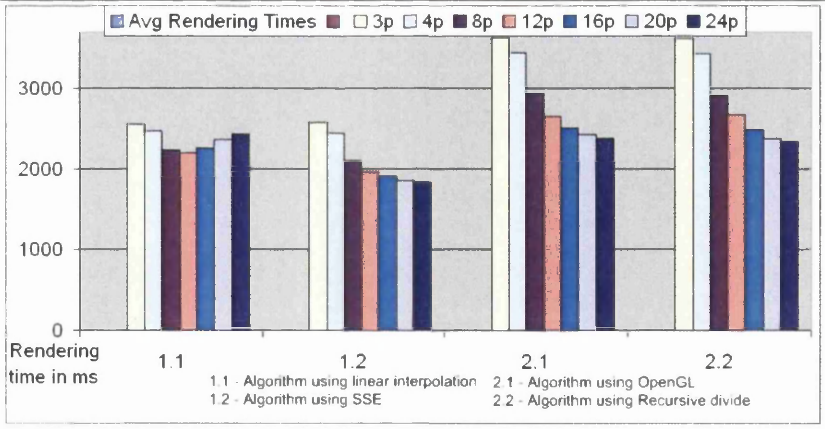
Figure 3.14: Variation of triangle intersections per pixel over number of split axes used to build the *RBSP tree* for ray tracing several models. As the number of axes used increase, the number of triangle intersections decrease.

availability of several splitting alignments is not advantageous and could in fact be detrimental due to more expensive traversals.

On the other hand, the advantages are clearly revealed by the results on the *Sphere* model. For this model, the split axes are perfectly aligned, as they are chosen by using evenly spaced points on a sphere. These axes are almost customised to a sphere and hence as the number of axes increase, the tree becomes more closely wrapped to the model. The *RBSP tree* for this model is very close to a BSP tree built on the model as the splitting planes are actually drawn from the model. This also suggests that if the splitting axes are chosen so that they are in close alignment to the scene triangles' alignment, the *RBSP tree* would be an even better structure for ray tracing.

3.7.2 Rendering Times

Scene Name	Rendering times with various methods and various number of axes
Armadillo	
Bunny	

Dragon	
Happy Buddha	
Sphere	

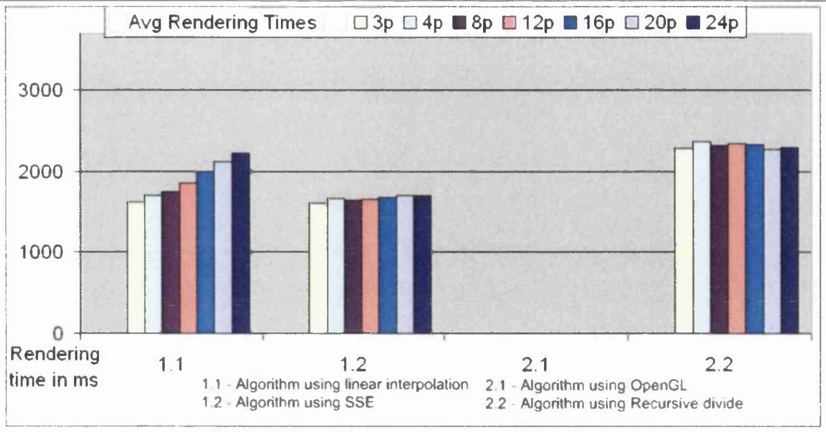
Sponza	

Table 3.4: Ray tracing times with different methods on *RBSP trees* with different number of splitting axes for various scenes. The various algorithms are:
Algorithm 1.1 – Traversal by Linear Interpolation of Ray–Plane Intersection Parameter.
Algorithm 1.2 – Traversal using SSE.
Algorithm 2.1 – Entry and Exit Plane Determination with OpenGL.
Algorithm 2.2 – Entry and Exit Plane Determination by Recursive Divide.

Section 3.5 described four different methods to traverse the *RBSP tree*. In order to evaluate the merits of each of these methods for ray tracing, the scenes have been rendered with the trees using all the four different traversal methods.

Algorithm 1.1 – Traversal similar to kd-tree traversal by using a linear interpolation at each step. The results for this method reveal the problem discussed in the earlier section. In this method, initially an intersection parameter calculation for each axis is necessary. This calculation is quite expensive, involving three dot products and a divide for each axis. As the number of axes used increase, these calculations dominate the rendering times. Thus, the rendering times increase as the number of axes used increase even though there is a decrease in the number of traversals and intersections.

Algorithm 1.2 – Traversal using SSE for initial calculations. This method realises the potential of *RBSP trees*. In this method, the initial calculations are computed in groups of four with SSE instructions. Hence, as the number of axes rise, initially there is a noticeable improvement in performance. However, the performance deteriorates when the number of axes rise above a certain number. This is similar to Method 3.5.1, but it just takes more axes for these to dominate. It is to be noted that this method results in a very fast ray tracing method – even better than on kd-trees. As long as the number of axes used is below a certain threshold – possibly 12 – this rendering method results in the best performance for ray tracing with *RBSP trees*.

Algorithms 2.1 and 2.2 – These are essentially the same method with slightly different approaches to finding the entry and exit planes. However, the methods of finding the entry and exit planes are quite efficient for both these methods, with the recursive divide method being slightly slower, as shown by the results. The graphs for these two methods are very similar showing that the performance increases as the number of axes increase. The performance increase directly reflects the improved numbers of intersections and traversals. The disadvantage of these methods is that they are generally slower than either method 3.5.1 or 3.5.2 due to increased number of ray–plane intersections.

As with the intersection number results, performance results show that *RBSP trees* provide exceptional performance for the *Sphere* model where the axes are perfectly aligned to the model. Results for method 3.5.2 show that significant performance increase is possible with *RBSP trees*. They reiterate the fact, that if properly adapted to the scene, *RBSP trees* produce trees that are better than kd-trees for ray tracing scenes with non-axis-aligned triangles. The results for the *Sponza* scene show that for scenes with predominantly axis-aligned triangles, *RBSP trees* constructed with the current SAH is worse than kd-trees.

3.7.3 Construction Times

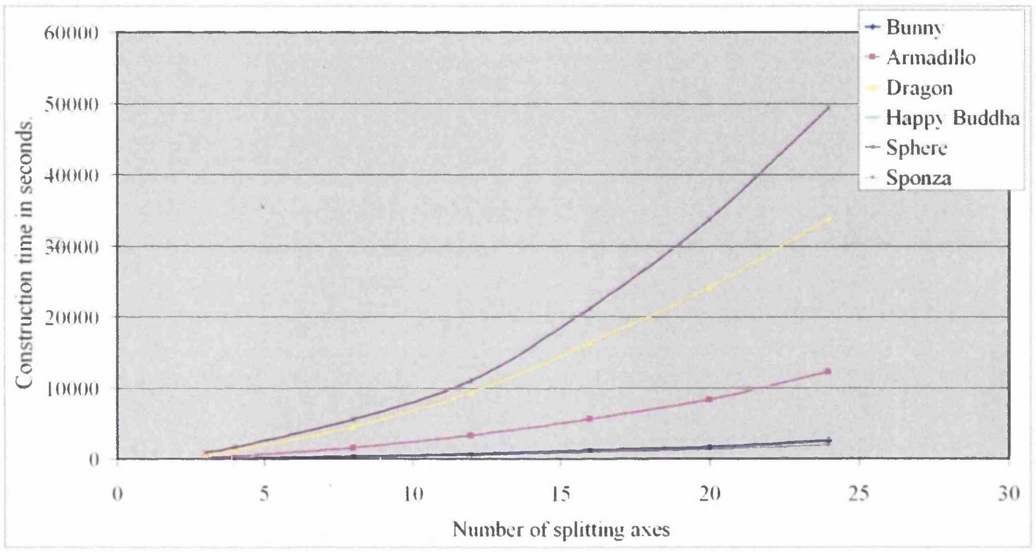


Figure 3.15: Variation of construction times of *RBSP trees* over number of split axes for various models. From the data graphed it was deduced that empirical complexity of *RBSP tree* construction is $O(m^{1.6} N log^2(N))$.

The construction times for *RBSP trees* is shown to be highly dependent on the number of splitting axes used to build the trees. The graphs for every model is identical and closer inspection reveals that the empirical complexity of tree construction appears to be approximately $O(m^{1.6} N log^2(N))$ [1] where m is the number of directions used and N is the number of triangles in the scene. This complexity makes it highly impractical to use more than 24 split axes as the time necessary for construction becomes prohibitive.

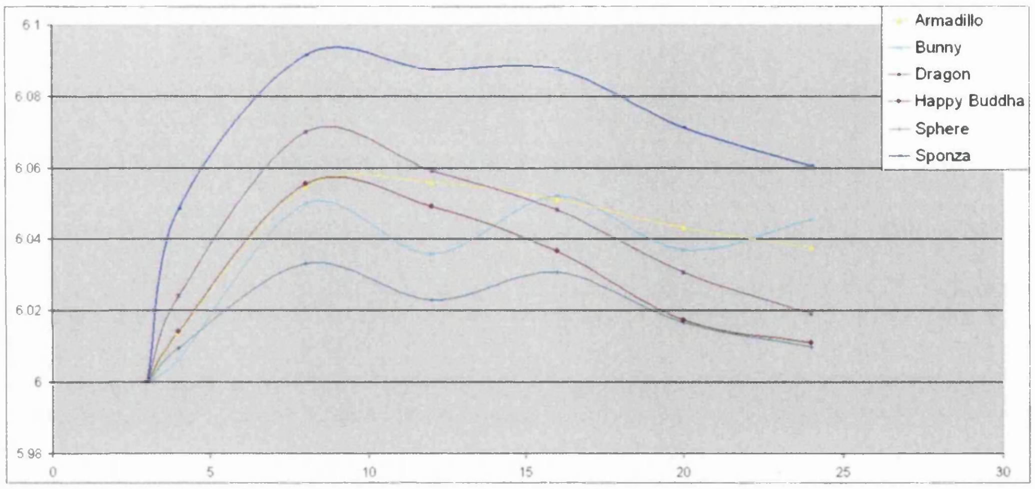


Figure 3.16: Faces per node of *RBSP trees* with various number of split axes for various models. The number of faces per node appears to be very close to six, irrespective of the number of splitting axes used.

3.7.4 Number of Faces in Node

Examining the average number of faces of all the nodes of *RBSP trees* with different number of splitting axes, reveals that it is very close to six irrespective of the number of axes. This is an interesting result. During construction of the tree, it implies that only six planes are relevant and this result may be used to reduce the complexity of construction.

3.8 Further Research on Structures with Non-Axis-Aligned Splitting Planes

The introduction of the *RBSP tree* in [KM07] has sparked interest in structures that use non-axis-aligned splitting planes [BCNJ08] [IWP08]. Budge et al. [BCNJ08] attempt to address some of the problems already mentioned in this chapter like slow construction times. Having noticed the potential of using numerous non-axis-aligned splitting planes in *RBSP trees*, Ize et al. [IWP08] attempt to use the even more general form of BSP trees that allows arbitrarily aligned planes. The main contributions of these two publications and comparisons to our work will be described in brief in the sections below.

3.8.1 Accelerated Building and Ray Tracing of Restricted BSP Trees – Budge et al.

Budge et al. [BCNJ08] present algorithms for building and ray tracing with *RBSP trees*. Our paper highlighted the slow construction of *RBSP trees*. In addition, in [KM07], we present only traversal method 3.5.3.2 – that, in this chapter, has been shown as the slowest traversal method. Budge et al. attempt to address the problems of *RBSP trees* as presented in [KM07].

The empirical complexity has been calculated using the observed construction times, graphed in Figure 3.16.

In [KM07] it was said that an empirical complexity of $O(m^{1.6}Nlog^2(N))$ was observed for constructing *RBSP trees*. Budge et al. reduce this to $O(m^3 + (mNlog(N)))$. The main reason for the high complexity was that the polyhedron of the node was found by clipping each plane with every other non parallel plane. This resulted in the observed high complexity. Budge et al. identify that the SAH is a problem that can be solved with dynamic programming to reduce this complexity.

The construction process in [BCNJ08] is an SAH heuristic with changed intersection and traversal costs of 500 and 1 respectively. An $O(Nlog^2(N))$ approach, in which the triangles are sorted at each SAH step, is used. However, instead of using a normal $O(Nlog(N))$ sorting method at each construction step, a radix sort is used to achieve an $O(Nlog(N))$ construction complexity (for kd-trees). Each k-DOP (the term used for a node's polyhedron along the lines of Klosowski et al. [KHM+98]) is represented by an edge-soup – made of line segments. Each line segment also has a pair of indices identifying the two faces to which the edge belongs. This representation is said to be compact, efficient to maintain during splits and is sufficient to compute the surface area of a k-DOP. No other information is stored for the k-DOPs.

Dynamic programming is a method used to solve problems with overlapping sub-problems. It works by first dividing the problem into several sub-problems that are again split until the sub-problems are simple to solve. The solutions from these are then combined to obtain the solution to the problem. In the surface area heuristic, along an axis, if the split points are sorted, then the surface areas of the first potential sub-node (of the two potential sub-nodes created due to the split) at these points are increasing. The surface area at the first point is found. The surface area at the second point is found by using the first solution and so on to reduce the calculation time. Using this approach, the expression for the areas is given as below.

$$\begin{aligned} Area_{leftOfSplit} &= C_{2i}(t_{split}^2 - t_i^2) + C_{1i}(t_{split} - t_i) + S_i \\ Area_{rightOfSplit} &= Area_{total} - Area_{leftOfSplit} \\ Area_{split} &= K_{2i}t_{split}^2 + K_{1i}t_{split} + K_{0i} \\ t_i &<= t_{split} < t_{i+1} \end{aligned}$$

where C_{2i}, C_{1i}, K_{2i}, K_{1i} and K_{0i} are coefficients initialised for each face. The details of calculating these are provided in [BCNJ08].

Budge et al. state that the traversal method in [KM07], Section 3.5.3.2 is not the best method. A method similar to the standard slabs method is proposed. Initially, the bounding box rather than the bounding volume is intersected by the ray. The ray origin and the ray direction reciprocal are pre-computed and the traversal is continued. SSE is used for accelerating the pre-computations.

If the traversal is performed without pre-computation, a general decrease in rendering times is noticed. However, they state that in this case, the trends are not clear. As stated earlier, we believe that the rendering times depend on the number of axes used. In case the number of axes are greater than the depth of the tree, not pre-computing can lead to a cheaper traversal, owing to fewer ray–plane intersections.

The traversal method is shown to be faster than the presented (unoptimised) times by a factor of 10. When the bounding box is used instead of the bounding volume, fewer triangle intersections but greater number of node traversals are seen. This is a disadvantage of using boxes in cases when the viewpoint is outside the model. A closer fitting bounding volume can mean that numerous rays can be terminated at the root node level itself.

They observed problems with the *RBSP tree* constructed for scenes like the *Sibenik* scene, and attributed the problems to the fact that due to increased empty space culling, the tree is deeper. However, we believe that the problem lies in the selection of planes to build the tree. The problem is similar to the problems with the *Sponza* scene which is dominated by axis-aligned triangles. A heuristic in which the directions are customised to the scene for which the tree is being built is suggested as one of the solutions. An approach used by Coming and Staadt [CS08] to achieve this customisation is suggested. This is an important benefit of *RBSP trees* – that the directions can be arbitrarily chosen. However, to take maximum advantage of *RBSP trees*, the directions have to be carefully selected. When that is done, *RBSP trees* may significantly outperform kd-trees.

Comparison of Traversal Methods The traversal methods described by Budge et al. are similar to methods 3.5.1 and 3.5.2 detailed in this chapter. The main difference is the use of the bounding box by Budge et al. rather than the bounding volume. In addition, since the ray's intersection parameters are pre-computed, a linear interpolation approach can be used to compute the t_{split} parameters. These are believed to increase efficiency, especially in cases where the viewpoint is outside the model. Also, while they state that their heuristic is not faster than the kd-tree, our approach – detailed in Section 3.5.2 – for tracing single rays is faster for *RBSP trees* than for kd-trees.

3.8.2 Ray Tracing with the BSP Tree – Ize et al.

Ize et al. [IWP08] take a more generalised approach. So far, it was believed that using arbitrary non-axis-aligned planes does not lead to a good structure for ray tracing. While a restricted set was used to make the problem more solvable, Ize et al. use a general BSP tree using popular ray tracing concepts – like the SAH – shown to be applicable for non-axis-aligned planes by the study on *RBSP trees*.

One of the main problems of using arbitrarily aligned planes is the construction of an effective structure for ray tracing. Ize et al. attempt this by reducing the number of split planes used at each split step. The SAH, with a polytope area calculation similar to ours – by clipping the node's polytope with the splitting plane and then computing the area by summing the areas of the faces, is used to select the best split position from among a set of planes. Ize et al. limit the number of planes at each triangle by using the triangle's properties. For each triangle the split planes used are

- The plane that defines the triangle face.
- The three planes that lie on the edges of the triangle and orthogonal to the triangle face.
- The same six axis-aligned planes used by the SAH for the kd-tree.

Restricting the planes as above limits the number of planes to be tested to $O(N)$. At each split plane, since triangles are not sorted along the axis, a helper structure – a bounding sphere hierarchy – is used to count the triangles at either side of the potential plane leading to a $log(N)$ cost for the search and an $O(Nlog^2(N))$ construction heuristic.

Another adjustment is made during the construction process to the SAH intersection cost. The cost for intersecting a ray with a plane aligned along one of the coordinate axes is less than intersecting with an arbitrarily aligned plane. Due to this, two costs, C_{bsp} and $C_{kd-tree}$ – indicating the cost of intersecting with an arbitrary plane and an axis-aligned plane respectively, are used during the

SAH process. For arbitrary planes, it is deduced that the cost linearly varies with the number of triangles in the node. Thus, the values of C_{bsp} is given as

$$C_{bsp} = \alpha C_i(N-1) + C_{kd-tree}$$

where α – is a user tunable parameter (a value of 0.1 was used in the paper for α).

If, after investigating all split planes, it is determined that the cost of splitting is not better than making this a leaf node, then a fixed cost is used for C_{bsp} and the SAH is process is run again.

The two modifications to the SAH cost computation – using two different costs and considering the cost of intersecting the non-axis-aligned planes as being linearly dependent on the number of triangles in the node – are interesting. The application of these ideas to the *RBSP tree* construction process is worth investigating. It may ensure that good trees are created for problematic scenes.

The traversal is a modified kd-tree algorithm. A bounding box is used to ensure that the ray has a good probability of hitting the scene. Each split plane is then intersected using the standard ray–plane intersection test – involving two dot products and a floating point division. However, due to limited floating point precision, an epsilon value is used and rays with intersection distances within an epsilon value of the split plane traverse both nodes. The BSP node traversal is roughly $1.75\times$ slower than the kd-tree node traversal. Hence, a standard kd-tree plane intersection test is used when the planes are axis-aligned. Another idea used is that in the nodes, the splitting plane may actually be the triangle itself. In these cases, the intersection need not be recalculated.

A disadvantage of using arbitrary planes instead of a restricted set is that each node requires 20 bytes, instead of 8 bytes for a kd-tree or an *RBSP tree* node. This leads to a significant increase in memory usage.

Ize et al. observe good results with the BSP tree. When single ray tracing with secondary rays are used, the structure outperforms the kd-tree. The single ray tracer is faster on the BSP tree than on the kd-tree. With SIMD, ray tracing on the BSP tree is as fast or faster than on the kd-tree. Due to increased traversal costs, a pure BSP tree is not as effective if axis-aligned planes with faster intersection methods are not used. In contrast to the *RBSP tree* where there is a decrease in both traversals and intersections, the BSP tree only brings about a decrease in the number of intersections.

When the viewpoint is outside, the *RBSP tree* would be able to terminate more inactive rays due to a better fitting bounding volume. This is supported by Ize et al. when an RBSP built for the top level is suggested. The number of planes are limited and investigating more split points – in the order of $O(N^3)$ – is believed to make the BSP tree more effective. However, using such a large number of split points is not viable. The memory requirement of general BSP trees is also quite high. When single rays are traced, they state that their structure is the fastest structure. This corroborates our belief that the use of more than three non-axis-aligned axes leads to a better structure for ray tracing than kd-trees, especially for single ray tracing. A point to be noted though is that Ize el al. have not tested the performance of their structure against *RBSP trees*, which could be faster due to faster traversal methods and cheaper memory requirements.

3.9 Summary

RBSP trees are an attempt to combine the best features of kd-trees and BSP trees as applicable to ray tracing. It can also be thought of as an experiment to determine the usefulness of arbitrarily aligned splitting planes. It has been assumed that the general form of BSP trees would not be a very effective structure for ray tracing. However, it is hard to ignore the promise of BSP trees being a structure that very closely wraps the scene being rendered to reduce the number of intersections and traversals. *RBSP trees* are a subset of BSP trees that can be built to be very similar to BSP trees. Thus, *RBSP trees*, in addition to being an excellent structure themselves, also enable the study of BSP trees for ray tracing.

RBSP trees are compact to represent, have relatively simple construction methods, and have efficient traversal methods. The efficiency advantage of having a greater choice in number and alignment of splitting axes is realised for scenes that pre-dominantly consist of non-axis-aligned triangles. The results show that for these scenes, as predicted, *RBSP trees* do reduce the number of intersections and traversals significantly, compared to kd-trees. This advantage is translated to a performance advantage when the right traversal method is used so that *RBSP trees* can be a faster method to ray trace a scene. The best traversal method is an SSE version similar to the kd-tree traversal method.

The introduction of *RBSP trees* has sparked interest in the use of space subdivision structures with non-axis-aligned splitting planes. Budge et al. [BCNJ08] attempt to address the problem of slow construction times through the use of dynamic programming to obtain a faster method to calculate the surface areas. Ize et al. [IWP08] apply some of the results from the *RBSP trees* to the more general BSP trees with arbitrary planes. Both of them show very good results confirming the belief that the use of non-axis-aligned planes is worthy of further study.

The study also reveals that for scenes that are dominated by axis-aligned triangles, the construction heuristic results in trees that are not as good as kd-trees. On the other hand, for scenes like the *Sphere* scene, where the splitting axes are essentially drawn from the scene, the heuristic constructs trees that are significantly better than kd-trees. Another disadvantage is that the construction times are significantly higher than that of a kd-tree and depend highly on the number of splitting planes used. Both these problems can be solved by future work enabling intelligent selection of available planes for the SAH upon investigation of the scene's component triangles' alignment. Using a customised set of directions would allow reducing the number of splitting axes to a manageable number. The optimum number of planes is found to be between 8-12.

As a method for visibility determination, the use of *RBSP trees* with single rays is probably not practical. However, when large number of incoherent rays are to be traced, as in global illumination methods, the *RBSP trees* are thought to be especially useful. This is also an area of research worth pursuing as part of the future work.

Chapter 4

Coherent Rendering

Contents

This chapter discusses a visibility / rendering algorithm that attempts to demonstrate that triangle mesh based scenes can be rendered at a lower complexity per pixel. The volume rendering algorithm that shows a constant complexity per pixel motivates the investigation. It is research undertaken as a part of the EPSRC grant that funded the investigation of lower complexity algorithms.

This chapter expands and describes in detail the methods and results reported in the technical report – *Benjamin Mora, Ravi Kammaje and Mark W. Jones, "On the Lower Complexity of Coherent Renderings," Swansea University, Technical Report, 2008.* [MKJ08].

The main algorithm, due to its similarity to his earlier work [MJC02] [ME05], was implemented by Dr. Benjamin Mora. However, a lot of the background work (e.g., the kd-tree construction for the algorithm, ray tracing implementations against which *Coherent Rendering* is compared with, etc.) as well as refinement to ensure that its application to triangle meshes is accurate has been undertaken by me. The technical report, due to the fact that it was a submission to a conference was limited in length. Hence, it is mainly concerned with the analysis of the complexity results and does not delve into the details of the workings of the algorithm. This chapter details the algorithm in full detail, using which the algorithm can be reproduced. Producing the benchmarks for the complexity results has been another of my contributions. Finally, I am responsible for the comparison to ray tracing and packet ray tracing in Section 4.6.2 and an analysis of the absolute performance using profilers, as seen in Section 4.7.2. The analysis has led to the *Row Tracing* algorithm, detailed in Chapter 5 which has resulted in a highly accelerated rendering method. Thus, though Dr. Mora was the first author of the report, I have been responsible for a significant part of the project, leading it to be an integral part of my PhD, and consequently the thesis.

4.1 Motivation

RBSP trees proved to be an effective method to reduce the number of node traversals and triangle intersections necessary for ray tracing by utilising the spatial coherence of objects in the scene. Another method of achieving fewer traversals and intersections per ray is to use image coherence – i.e., tracing the rays in groups and amortising the number of intersections and traversals over the group. The method, called packet ray tracing / packet tracing, has been very effective to accelerate ray tracing. As described in Chapter 2, packet tracing traverses groups of rays through the acceleration structure and intersects the component rays with the primitives. Several different forms of packets like frustum based, pyramidal and rectangular packets have been used to accelerate ray tracing.

The best performance for ray tracing has been observed by the MLRTA [RSH05] – a form of frustum based packet ray tracing that found entry points deep inside the tree to begin tracing the individual rays. They also detail a packet traversal method based on interval arithmetic that is relatively easy to implement. This method has been used to intersect packets through a variety of structures and primitives [Ben06] [BWS06] [WBS07].

As the packet tracer traverses through the tree, several rays in the packet may not intersect the node. Even in these cases, the nodes need to be traversed since a few rays intersect the node. This reduces the coherence and the amortisation[1] provided by traversing the tree with packets. Consequently, the packet cannot consist of a large number of rays. At the same time, larger sized packets lead to maximum amortisation of traversal costs, particularly when a large majority of the rays traverse the nodes. Thus, a packet size that is optimal reduces the number of inactive rays in the packet. In our implementations, a packet size of 8×8 was found to provide the best performance[2].

A packet size of 8×8 implies that a 1024×1024 image needs to traverse 16384 packets which is still quite a large number of packets. Obviously, the number of packets increases as the image size increases. Thus, an alternate method that uses larger packets could be even more effective.

The largest packet that can be used is one with rays through all the pixels of the image. The method introduced in this chapter, *Coherent Rendering*, is an algorithm that considers all the pixels of the image. The concept is adapted from object order ray casting [MJC02] – a high performance volume rendering method that produces excellent images. Adapting this to triangle meshes, providing a good implementation and subsequently studying the properties of such an algorithm is the motivation for the rendering method called *Coherent Rendering*. The use of the entire image as the packet is expected to produce an algorithm that potentially reduces the complexity of rendering. The reasons for this belief will be described in the following section.

4.1.1 Average Complexity

The average complexity of rendering is important as scene sizes increase. Z-buffer based rasterisation has an average complexity of $(N \times s)$ where N is the number of triangles in the scene and

[1] **amortisation** in the context of packet ray tracing refers to the cost of node traversals being shared by the number of rays in the packet. For most scenes, it can be observed that several neighbouring rays traverse the same path down a tree. Thus, traversing the same nodes for each of the rays is wasteful. When packets are traversed, the node is traversed just once and the cost is divided, or amortised, amongst the component rays, as long as they intersect the ray.

[2] In our implementations, packet sizes of 4×4, 8×8, 16×16 and 32×32 were used. Of these, the performance was the best when packet sizes were 8×8, followed closely by packet sizes of 16×16.

s is the average projection size of the triangles. A better logarithmic complexity per pixel using graphics hardware has however been shown by Wand et al. [WFP+01]. On average, ray tracing – through the use of hierarchical data structures like the kd-tree or the BVH – has a complexity of $O(log(N))$ per pixel [HB00] [Hur05] [WFMS05] [WSS05] [HHS06] [YLM06]. A simpler data structure like the grid is shown to have a relatively slower complexity of $O(N^{1/3})$ [CW88]. It has also been shown that worst case complexity of ray tracing is $O(log(N))$ per pixel at $O(N^4 + \epsilon)$ memory and preprocessing cost (where $\epsilon > 0$) [BHO+94]. However, this memory and preprocessing cost is prohibitive and average case complexity is lent more importance.

Finding the closest triangle at a pixel (i.e., the closest triangle intersecting the ray corresponding to a pixel) can be considered as a special case of a searching algorithm. This implies that results obtained for search algorithms should hold for ray tracing. Hence, it may be difficult to improve upon the logarithmic complexity per pixel when searching for a single element in the tree. Bentley [Ben79] shows that if instead of searching for one element in the tree, k neighbouring elements are to be found, it can be achieved in $O(log(N) + k)$ time, where N is the number of elements in the tree. Thus, for sufficiently large values of k, the search complexity would be proportional to k instead of $log(N)$.

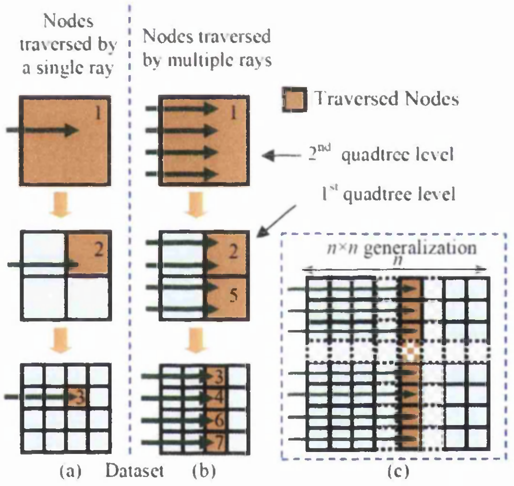
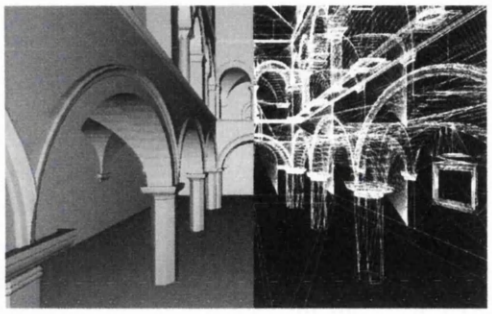


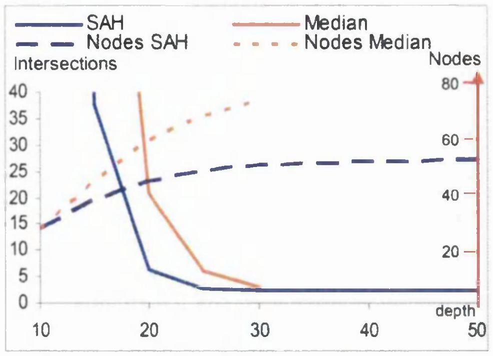
Figure 4.1: Naive recursive ray tracing example on a 4×4 voxelized grid. The number of distinctly traversed nodes is a geometric series $(1 + 2 + ... + N/2 + N = 2N - 1)$.

Packet ray tracing is one method that utilises this result to significantly accelerate ray tracing. Packet ray tracing attempts to find the closest intersection for a small set of pixels. The fact that the number of rays in the packet cannot be arbitrarily large (due to reasons of coherence mentioned before) implies that k is small enough to not make a significant difference to the complexity. Rendering times for packet ray tracing, hence, still appear to be logarithmic [WFMS05] [Hur05]. However, if k can be made significantly larger – by increasing packet sizes to very large sizes (entire image), then it would overshadow the term $log(N)$ and as k approaches very large numbers, the complexity would depend entirely on k providing a constant complexity per pixel. This result has been used by Jensen [Jen01] to search for photons in a photon map. Utilising this result to investigate if a better complexity for rendering could be achieved is the basis for the *Coherent Rendering*.

The application of Bentley's search result to 3D rendering is obviously not trivial at this point, even



(a) *Sponza* scene – Left side rendered with *Coherent Rendering* and right side showing the wireframe



(b) Average number of intersections and nodes traversed by a ray tracer for the *Sponza* scene with space median and SAH kd-trees. Average node traversals appears to be logarithmic whereas the number of triangle intersections appears to converge to a constant.

Figure 4.2: Ray tracing the *Sponza* scene

if it is highly probable that the intersection points of the different rays composing the image are coherently located in the search space. Figure 4.1 illustrates the application to the domain better by showcasing a 4×4 voxelised 2D world containing a 1D flat surface that will be ray traced. In the figure, rays and surfaces are axis-aligned and the generalisation of the same case to a $N \times N$ voxelisation will be discussed (N being a power of 2). To speed-up rendering, two quadtree levels ($log_2(N)$ levels in the general case) are constructed from the voxelised scene, indicating non-empty spaces. Figure 4.1(a) indicates the nodes (numbered by traversal order) traversed by a simple top-down recursive ray tracer for a single ray. In this example, three node traversals ($log_2(N) + 1$) are needed before hitting the surface, which happens immediately after reaching the first leaf node. If the same simple recursive ray tracer is called for four (i.e, r) different but similarly aligned rays as depicted in Figure 4.1(b), a recursive ray tracer would traverse 4×3 nodes (i.e, $r(log_2(N) + 1)$). However, it is noticeable that some of them are traversed several times (nodes 1, 2 and 5 in Figure 4.1(b)). Hence, if only distinct nodes traversed by the algorithm are counted, only seven nodes ($2N - 1 = 1 + 2 + 4 + ... + N/2 + N$) are traversed by four (i.e, r) rays.

Therefore, if an algorithm manages to traverse every node just once, the complexity of tree traversal would be reduced from logarithmic per ray to a constant per ray. This is the principle behind *Coherent Rendering*.

The assumption is that the logarithmic complexity of a regular recursive ray tracer is due to the tree traversal itself, and not due to the intersection tests. The average number of intersections per ray by a regular recursive ray tracer tends to a constant, provided the tree is well constructed. Ray tracing the *Sponza* scene 4.2(a) with an SAH kd-tree, shows that the average number of intersections per pixel tends to 2. At the same time, the number of traversal steps increases with maximum depth. This strongly supports the belief that the per-pixel logarithmic complexity of a regular ray tracer is mainly due to the tree traversal, and the intersection complexity is $O(1)$ and indicates that efforts should concentrate on lowering the number of tree traversals.

The volume rendering method – *Object Order Ray Casting (OORC)* by Mora et al. [MJC02] –

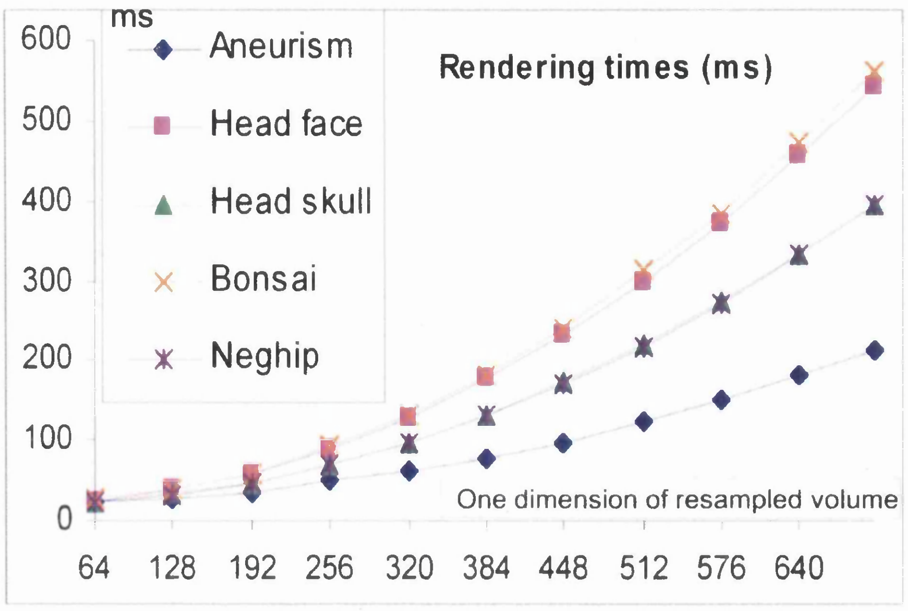
demonstrates an $O(1)$ complexity per pixel. This method and its main results will thus be detailed in brief before the *Coherent Rendering* algorithm is detailed.

4.1.2 Object Order Ray Casting

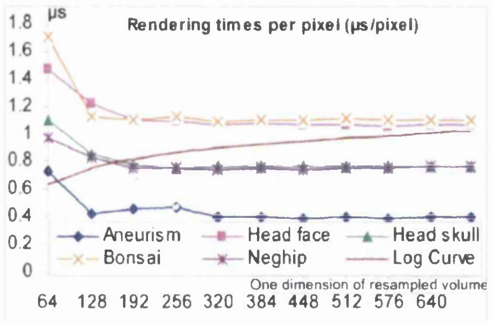
Object Order Ray Casting by Mora et al. [MJC02] is an object order method for volume renderings. However, instead of tracing a ray through the volume, cells of the volume / voxels were projected onto the screen. Since only orthogonal projections were considered, the voxels projected onto a similar area (hexagon) on the screen. By just displacing the mid-point of a pre-computed projection, a voxel's projection is found very efficiently. In addition, a min-max octree was used to identify and skip transparent regions of the volume. Using the octree also allows the front-to-back visibility order determination. Combined with *Hierarchical Occlusion Maps*, the visibility of a voxel / octree cell is determined. The combination of these structures for rendering is very efficient. A very similar *Maximum Intensity Projection* (MIP) algorithm [ME05] has been shown, both mathematically and empirically, to have an average complexity of $O(N^2)$ for an N^2 image of an N^3 volume leading to an $O(1)$ complexity per pixel. It is to be noted that since the algorithm renders only orthogonal projections, one dimension of the image and the volume are the same (N) leading to the aforementioned complexity result.

The important results from the *OORC* algorithm will be briefly described in order to demonstrate the $O(1)$ complexity that motivates us to adapt it to a mesh rendering context.

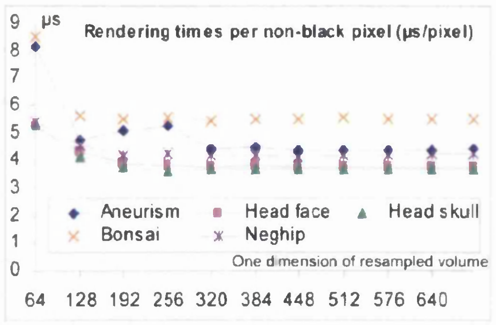
Four widely used datasets have been resampled from 64^3 to 700^3 voxels. The datasets are *Aneurism* (originally 256^3), *UNC head* ($256 \times 256 \times 225$), *Bonsai* ($256^3$) and *Neghip* ($64^3$). All renderings produce a 1024×1024 image using orthogonal projection (current code does not allow perspective projection). For all renderings other than the *Bonsai* dataset, the zoom value has been fixed to 1, which means that an axis-aligned projection of a voxel will have a 1×1 pixel footprint. The zoom has been reduced to 0.8 for the *Bonsai* dataset because the 700^3 dataset's isosurface would otherwise not fit the screen space.



(a) Isosurface rendering times according to one dimension of the volume size N



(b) Per pixel rendering times. Flat curves indicate that the complexity is proportional to N (and therefore $O(1)$ per pixel)



(c) Rendering time per non black pixel.

Figure 4.3: *Object Order Ray Casting* rendering times

Rendering times represent the average contribution of 30 different viewpoints. Finally, significant effort has been put into optimising the OORC algorithm, using a profiler and SIMD SSE instructions as well.

Figure 4.3(a) shows the rendering times according to the volume size. In order to demonstrate the complexity better, the preprocessing time is subtracted from these rendering times before dividing it by the number of pixels in the image(Figure 4.3(b)). The preprocessing time is the time required to initialise a few rendering parameters and the 1024×1024 image (which is oversized when rendering small volumes). Thus, only the tree traversal and the access to the relevant pixels of the image are taken into account, which demonstrates the convergence slightly sooner.

After analysing Figure 4.3(b), it appears that the curves are flat enough in the $[196^3 - 700^3]$ voxels

	Wald et al.	Mora et al.
Processor	Dual Opteron 1.8 GHz, 1MB	Athlon 64 X2 2GHz, 512KB
Cores used	2	1
Bonsai	5.2 fps	8.2 fps
Aneurism	6.2 fps	20 fps

Table 4.1: Comparison with ray tracing approach

range to validate the $O(N^2)$ complexity, especially when compared to a theoretical logarithmic curve in Figure 4.3(b). It is thought that the low efficiency of the algorithm for models smaller than 196^3 may be due to the overestimation used during the visibility process. A more accurate estimation of visibility could well lead to an earlier convergence.

Rendering times in Figure 4.3(b) are divided by the percentage of non-black pixels in each rendering to obtain rendering times per non black pixels (shown in Figure 4.3(c)), since some volumes like the *Aneurism* are rendered much faster as they have fewer affected pixels. One can see that rendering times per non-black pixel are now much closer. Only the *Bonsai* dataset has a slightly higher cost per non-black pixel, probably due to the different voxel/pixel aspect ratio used.

Finally, the rendering times for the *Aneurism* and *Bonsai* datasets are compared with the very interesting ray tracer by Wald et al. [WFMS05]. This algorithm – shown to be logarithmic – has been optimised using SIMD instructions as well, and unlike OORC, is able to perform perspective projection. The same isosurface, the same processor family, and the same image size (512^2) has been used. While it is necessary to be very careful while such comparisons are made since many parameters like shading and viewpoints may not be the same, this comparison will give us a good idea of the order of efficiency of the OORC algorithm. Results summarised in Table 4.1 show that the level of performance obtained with just a single thread by the OORC algorithm is much better even for small volumes. It is likely that the performance advantage is more pronounced as the volumes get larger due to the better complexity.

The method is a very efficient method to undertake volume rendering, showing significantly higher performance than previous methods. Analyzing the complexity of this method reveals that it has an average complexity of $O(1)$ per pixel. This led to the belief that if a similar algorithm could be implemented for triangular meshes, similar performance and computational complexity could be achieved. However, the algorithm for volumes was to be adapted for use with meshes. In addition, perspective projections – more popular in triangular mesh renderings – was necessary to be considered. *Coherent Rendering* is the adaptation of this object ray casting method to render triangular meshes.

4.2 Coherent Rendering – Concept and High Level Algorithm

As mentioned, *Coherent Rendering* is an adaptation of the *Object Order Ray Casting* method. It is an attempt to consider the entire viewing frustum as a packet of rays. However, in order to achieve this, concepts from both rasterisation and ray tracing are used. In addition, to perform the occlusion testing – achieved easily for single rays through early ray termination – the concept of *Hierarchical Occlusion Maps* [ZMHH97] [Zha98] is adapted.

The high level algorithm of *Coherent Rendering* first considers the root node of the kd-tree and determines if it is visible. If it is, then its children are considered in a front-to-back order with respect to the viewpoint. This is continued until either the node is determined as being not visible or a until the leaf node is reached. A node can be determined as being not visible if it is either outside the frustum or if it is occluded by already rendered parts of the scene. Once the traversal reaches the leaf node, it implies that some of the geometry in the leaf node may be visible. Thus, each triangle is tested for visibility and rasterised if it is visible. In this manner, the entire scene is considered as a packet and at the end of the process, each pixel is shaded accordingly.

The high level algorithm can be written as shown below.

```
CoherentRender()
{
        InitialiseConstants();
        TraverseTreeCoherent(rootNode);
        Shade Pixels
}
```

Listing 4.1: High Level Coherent Rendering algorithm.

One of the differences between rasterisation and *Coherent Rendering* is the use of a ray tracing structure like the kd-tree to achieve a better complexity for rendering. The front-to-back traversal of the structure ensures that geometry that is closer is processed prior to geometry that is further away from the viewpoint. Using this property, occlusion is detected for nodes (including the geometry in them) by using *Hierarchical Occlusion Maps* that are adapted to the algorithm.

The fact that in this method each node is traversed just once at most, is important to attain an improved rendering complexity. In addition, the triangles are not shared by too many nodes as long as a good tree can be built. Thus, except in cases where a good tree cannot be built, the number of triangles to rasterise is minimal. Minimising the number of these two operations – responsible for almost all of the computational time in a rendering system – is expected to lead to a very efficient method.

While the main traversal algorithm is well-known, the primary goal is mainly to ensure correct implementation of the entire pipeline and to observe the complexity improvement. Many papers / software are already using similar algorithms for rendering [Gre96] [ZMHH97]. For example, Bittner et al. [BWPP04] described a similar traversal using occlusion queries to test for occluded cells. However, the visibility function must actually perform a constant ($O(1)$) number of operations to ensure that the global average $O(1)$ complexity holds. One way to perform the test would be to query all rays intersecting the node and to check whether they are already opaque. However, even this would not be an $O(1)$ algorithm. The use of *HOMs*, that allow occlusion determination in constant time per pixel, is the main component in the attempt to obtain an $O(1)$ rendering method.

In the high level algorithm, there is an initialisation process whereby several important variables and constants necessary for the rendering process are initialised. Subsequently, the tree is traversed to determine the triangle that contributes to each pixel of the image. Finally, using the triangle and the position of the viewpoint and the lights, the pixel is shaded accordingly.

Of the three steps, the tree traversal including the leaf node processing constitutes the main computations in the algorithm. These methods including a few important auxiliary methods will be described in detail in the following sections.

4.3 Tree Traversal

Once the several variables and constants for this viewpoint have been computed, the tree is to be traversed to render the image. The traversal considers the nodes of the kd-tree structure in a front-to-back visibility order starting from the root node. It can be written using the pseudocode below.

```
TreeTraverseCoherent(node)
{
  Project node onto image
  if( node not within frustum or
        isOccluded(node) or
        node is fully between 4 pixels)
    return;
  if(isLeaf(node))
  {
    ProcessLeafNode(node);
    return;
  }
  side = viewpoint[axis] > splitpos;
  if(side > 0)
  {
    frontNode = node.leftNode;
    backNode = node.rightNode;
  }
  else
  {
    frontNode = node.rightNode;
    backNode = node.leftNode;
  }
  TreeTraverseCoherent(frontNode);
  TreeTraverseCoherent(backNode);
}
```

Listing 4.2: Tree traversal using the entire frustum. If the frustum intersects the root node, the tree is traversed in a front-to-back order until the leaf nodes where the triangles contributing to the image are determined.

Coherent Rendering, at the highest level, as shown by the pseudocode above, is a recursive algorithm that considers each node in a front-to-back order with respect to the viewpoint. At each node, the algorithm determines the node's visibility. If the node is visible, its child nodes are considered. The process continues until a leaf node is reached or until the node is determined as being occluded.

The main steps to determine if a node is visible or not are as follows:

- Project node onto the image.
- Determine frustum visibility.
- Test node for Occlusion.
- Test to see if node is too small.

Except the occlusion test – that will be described in much greater detail in Section 4.4 – the other operations will be detailed in the following subsections.

4.3.1 Node Projection

Projecting a node onto the image plane implies determining the bounding rectangle of the node. This is achieved by transforming the eight vertices of the node to image space. The vertices are multiplied by the global transformation matrix to first transform them (with pixels in the range of -1.0 to 1.0) and then to get the actual pixels occupied by them by multiplying and adding the half image width to the X components and half image height to the Y components. Once the pixels of the node's vertices are found, the maximum and minimum X and Y coordinates are selected to get the bounding rectangle of the node. Figure 4.4 shows the node projected onto the image space.

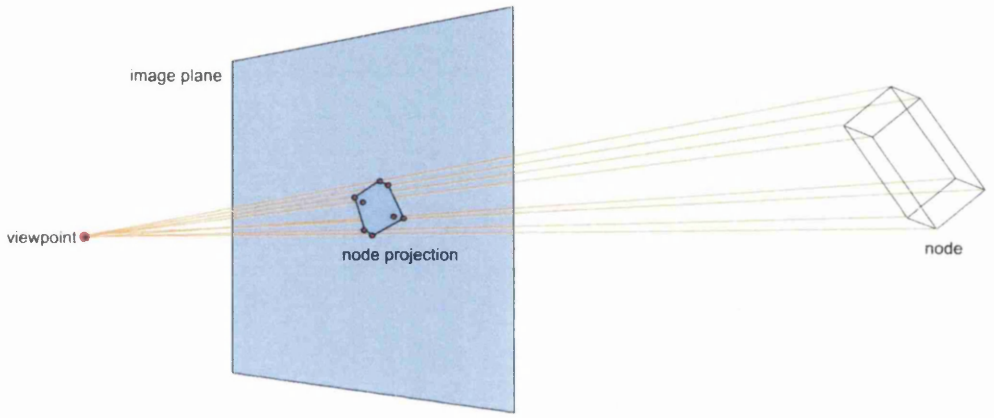


Figure 4.4: Node projection. *Coherent Rendering* projects all the eight vertices of the node onto the image plane to obtain the node projection for the root nodes.

The computations to convert a vertex from model space to image space can be given as:

$$\begin{aligned}
\mathbf{p} &= \mathbf{p_1} * M \\
p_x &= (p_x/p_w) * halfWidth + halfWidth \\
p_y &= (p_y/p_w) * halfHeight + halfHeight \\
p_z &= (p_z/p_w)
\end{aligned} \tag{4.1}$$

where $\mathbf{p_1}$ – the vertex to be projected
M – the global transformation matrix
$\mathbf{p}$ – the vertex after projection
p_x, p_y, p_z – X, Y, and Z coordinates of the point on the image
p_w – homogeneous coordinate
$halfWidth$, $halfHeight$ – Half of the image's width and height respectively

The above terms provide the calculations necessary to convert one point / vertex from model space to image space. Hence, to find the projection of the node on the image space, the eight vertices are transformed using the above calculations.

The above terms show that projecting a single point requires several expensive calculations – one matrix multiplication, one division and several additions and multiplications. Thus, the entire process of projecting a node onto the image is a very expensive operation.

In order to reduce a few calculations, the observation that when a kd-tree node is split, only four new vertices are created, is used. Thus, when a node is split into two, the eight vertices are divided into two sets of four vertices – one set for each child node. The other four vertices for the two child nodes are shared and are defined by the four vertices formed by intersecting the split plane with the node. Figure 4.5 shows this. Thus, instead of projecting all eight vertices of a node, four vertices of the split plane are projected and the list of projected vertices is maintained through the traversal process.

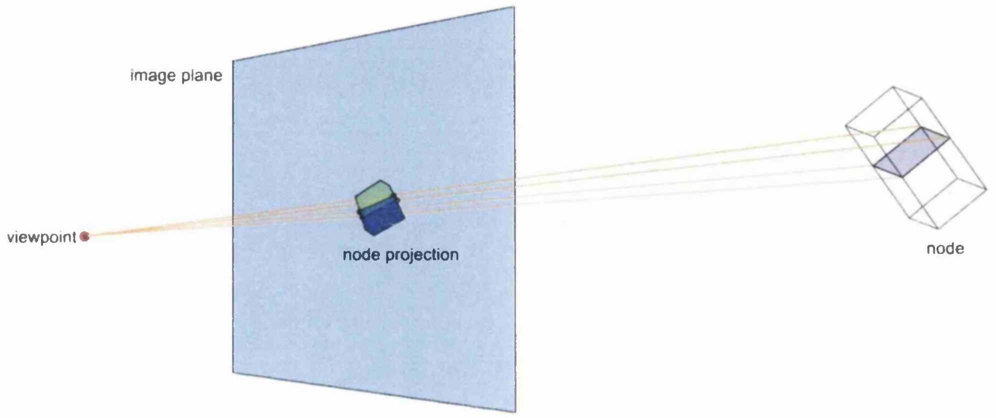


Figure 4.5: Node projection – projecting only the split plane. For non-root nodes, *Coherent Rendering* projects only the split plane to reduce computations.

Once the node has been projected onto the screen, it can be subjected to the visibility tests.

4.3.2 Frustum Visibility and Node Size Test

An unoccluded node is determined as not visible if it lies outside the frustum or if its projection falls between four neighbouring pixels (2 vertical and 2 horizontal pixels). These two tests are performed for every node at each of the traversal steps.

4.3.2.1 Frustum Visibility

With respect to the frustum, a node can either be partly inside the frustum, completely inside the frustum or completely outside the frustum. It is necessary to determine the case a node belongs to in order to determine the visibility of the node and its child nodes in consideration.

Node partially inside the frustum – To determine if a node is partly within the frustum, it is tested for intersection with the planes forming the frustum. Since the image is rectangular, the frustum is formed by four planes. The viewpoint forms the common point among all the planes. The other two planes are determined by the two end points of the edges of the image. Thus, a node is partly inside the frustum if the node intersects one or more planes of the frustum and a part of it is inside the frustum.

In the implementation, a clipping algorithm is used to determine if a node is partly within the frustum. The node is clipped with all the planes of the frustum. If the clipping algorithm determines that the polyhedron formed is not an empty polyhedron – detected when the number of vertices it comprises of is not zero – then a part of the node is within the frustum. When a node is determined as being partly inside the frustum, its child nodes have a possibility that they are fully outside / inside the node. Hence, child nodes of partly inside nodes need to be tested for frustum visibility at each recursive step until they are either fully outside the frustum or fully inside the frustum. This is shown in case 1 in Figure 4.6

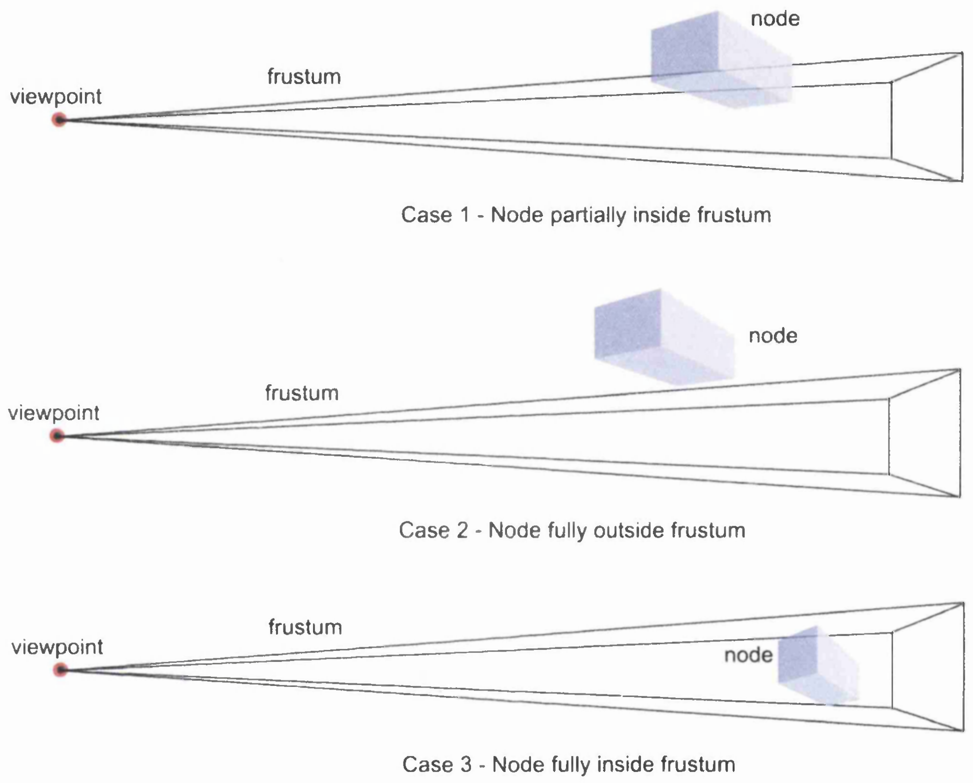

node
frustum
viewpoint
Case 1 - Node partially inside frustum
node
frustum
viewpoint
Case 2 - Node fully outside frustum
frustum
viewpoint
node
Case 3 - Node fully inside frustum

Figure 4.6: Frustum visibility of a node. Node can either be partially inside the frustum, fully outside the frustum or fully inside the frustum, as shown in the diagram.

Node fully inside or outside the frustum – When a node is determined as not being partly inside the frustum, it implies that the node is either fully inside or fully outside the frustum. Since the state of all the vertices of the node with respect to the frustum is the same (either inside or outside), it is sufficient to determine if one of the vertices is inside the frustum or not. If the signed distances of the point to all the four frustum planes are positive, it implies that the node is fully inside the frustum. Otherwise, it implies that the vertex and consequently the node is fully outside the frustum. Cases 2 and 3 in Figure 4.6 show these nodes that are fully inside and fully outside the frustum.

When a node is fully outside, the node can be skipped. When a node is fully inside, all child nodes are also fully inside. Hence, frustum visibility tests are not necessary for child nodes of fully inside nodes.

4.3.2.2 Node Size Test

In the *Coherent Rendering* algorithm, the pixels are considered as points rather than as squares. Thus, when a node becomes too small such that it lies between four pixels (two adjacent vertical pixels and two adjacent horizontal pixels), the objects in this node (or any child nodes / subtrees it may have) cannot contribute to the image. This occurs when the triangle density is high (a large number of small triangles). These nodes do not need to be processed.

This condition is easily tested. Once the node is projected onto the image plane, the vertices at the extremities along the X and Y axes of the image are determined. These values – the minimum and maximum values along a particular axis are used to determine whether the node is too small. When the integer part of the minimum and maximum values are the same along either axes, it is determined that the node cannot contribute to the image. Figure 4.7 shows a few examples of how this may happen. The pseudocode below describes the process of detecting if a node is too small.

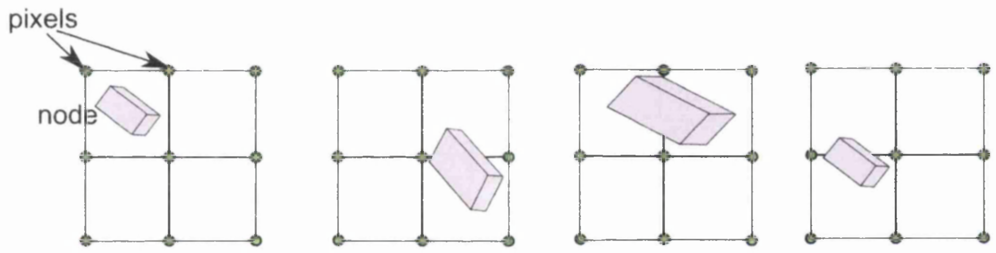


Figure 4.7: Examples of nodes occurring between four pixels. Since *Coherent Rendering* considers the pixels as points, geometry occuring between pixels are ignored.

```
pMinX = min(X values of eight vertices)
pMaxX = max(X values of eight vertices)

pMinY = min(Y values of eight vertices)
pMaxY = max(Y values of eight vertices)

nodeTooSmall = ((int)pMinX == (int)pMaxX or
                (int)pMinY == (int)pMaxY)
```

Listing 4.3: Pseudocode indicating how very small nodes, that are small enough to not contribute to the image, are determined.

If a node has been determined as being within the frustum and big enough, it can consist of triangles that are a part of the image. However, the nodes (and triangles) may be occluded by triangles that have already been rendered. Occlusion detection to ensure accurate visibility is achieved through the use of *Hierarchical Occlusion Maps*.

4.4 Occlusion Detection – Hierarchical Occlusion Maps

Occlusion detection is built into ray tracing when a space subdivision structure is traversed in a front-to-back manner. However, as packet sizes increase, the number of rays in the packet make it difficult to skip nodes due to the fact that the active rays may intersect some geometry in the node.

Similarly, with *Coherent Rendering* – which could be considered as a form of packet tracing with the entire image as the packet – it is difficult to use this property directly to determine occlusion. At the same time, it is important that occlusion is detected as early as possible so that nodes higher up in the tree can be skipped.

Since *Coherent Rendering* cannot use early ray termination directly, *Hierarchical Occlusion Maps* [ZMHH97] [Zha98] are adapted to serve a similar purpose. *Hierarchical Occlusion Maps* are conceptually simple and determine if regions of the image are occluded. They are adapted for use with *Coherent Rendering*. The usage mainly consists of two parts – updating them so that they indicate the current state of occlusion, and testing the corresponding HOM pixels to determine occlusion. The concept of HOMs as used in *Coherent Rendering*, as well as their usage is described in further sections.

With HOMs, occlusion is ascertained by testing just one pixel value. In addition, the update process is also proportional to the number of pixels in the image and is independent of the number of triangles in the scene. This is an important component in the efforts to investigate if an O(1) complexity rendering method is possible.

4.4.1 Concept of HOMs

As the name suggests, HOMs consists of hierarchically organised pixels. HOM pixels in the lower levels of the map indicate smaller areas with the lowest level indicating four pixels of the actual image. Each upper level HOM pixel combines four pixels from the corresponding lower level. At the highest level, a single HOM pixel represents the entire image. Figure 4.8 shows the HOM generated for the *Armadillo* model.

In order to minimise the number of HOM pixels accessed during traversal, the structure is modified slightly so that a given extent can be tested by examining just one HOM pixel. A region of influence of a pixel is determined as the four pixels closest to it (shown in Figure 4.9). i.e., when a pixel's triangle is determined, the closest four pixels are considered to have changed and hence the value in these four pixels are incremented. Due to this, a HOM pixel (covering four image pixels at the lowest level) can be incremented 16 times, with a value of 16 indicating that the corresponding area in the image is occluded.

The HOM pixels can have values between 0 and 16 – a value of 0 indicating that the pixels represented have not yet been rendered and a value of 16 indicating that the pixels are fully opaque and hence do not need to be processed again. A value between 0 and 16 indicates that a few pixels in the region of influence have been rendered, but not all of them are opaque yet. These values indicate that processing has to continue for areas that correspond to the pixel.

The HOM must be updated whenever a pixel is rendered so that it always maintains the current occlusion state for the rendering process. This is performed in the recursive rasterisation function described in Section 4.5.

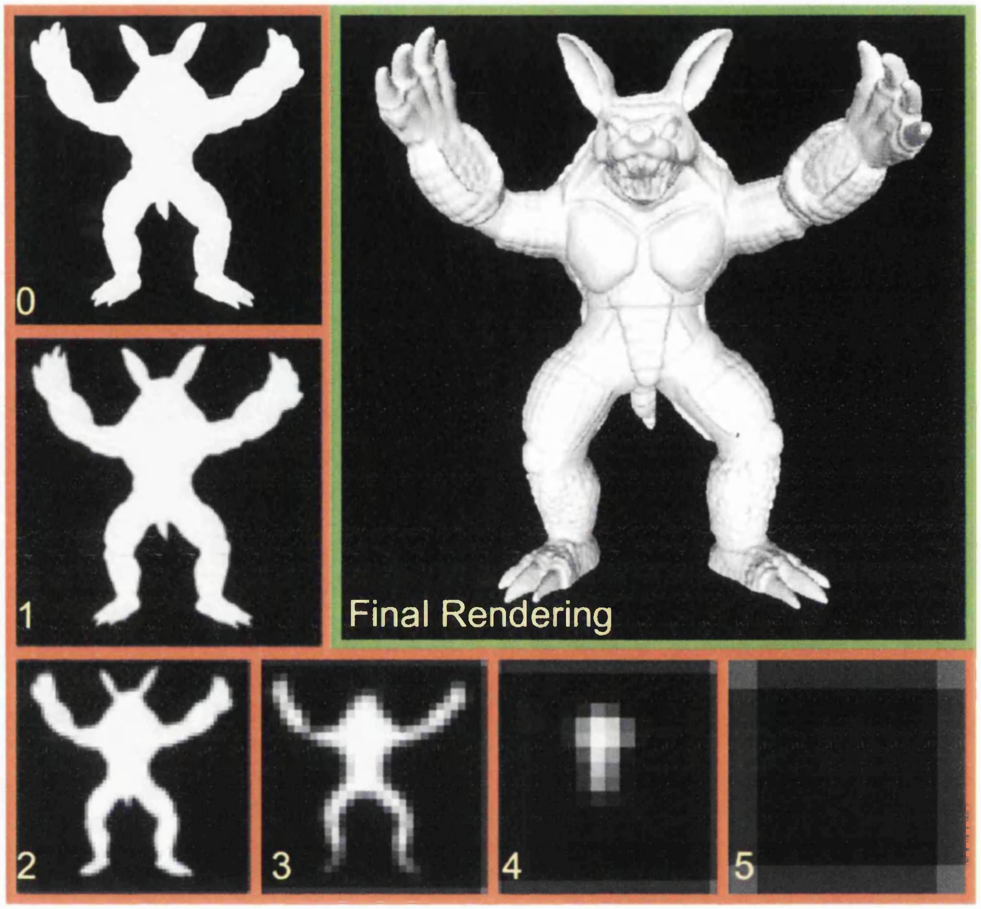


Figure 4.8: HOM for the *Armadillo* model. The image with the green border is the actual image of size 1024×1024. The image labeled 0 is the lowest level of the HOM and so on. Level 0 of the HOM consists of 256×256 pixels. The highest level (5) of the HOM has just one pixel.

The updating of HOMs – so that it indicates the current state of occlusion, and testing for occlusion using it will be described in the following sections.

4.4.2 HOM Update

When a pixel is rendered, it is set to the appropriate colour. At the same time, the pixel affected and three of its nearest pixels (as shown in Figure 4.9) are considered as being changed. Hence, the lowest level HOM pixel corresponding to these four pixels are incremented by 1. Each HOM pixel can be influenced by 16 lower level pixels and thus can be incremented 16 times. A value of 16 for the pixel value indicates that the region of the image represented by the pixel is opaque. This change is propagated upwards by incrementing the four closest pixels until either the topmost level is 16 or the value of a pixel is less than 16.

The concept of region of influence is an optimisation to the algorithm that allows only one HOM pixel to be tested. As will be discussed in Section 4.4.3, only the HOM pixel corresponding to the

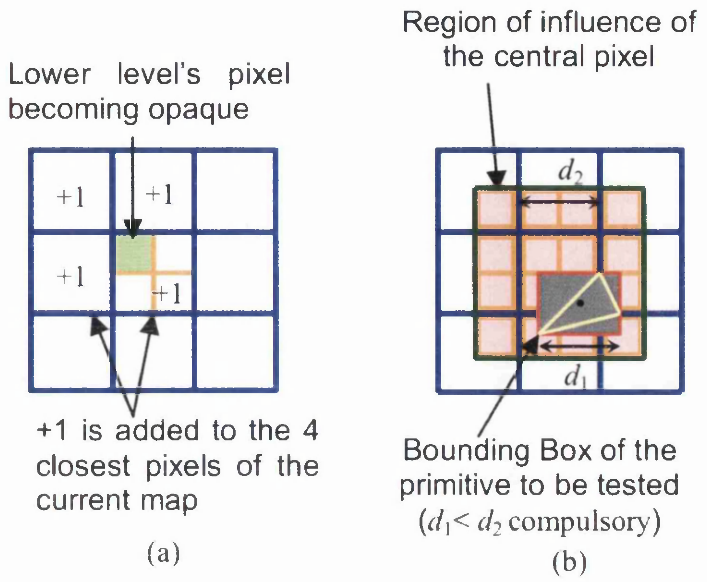

Lower level's pixel becoming opaque
+1
+1
+1
+1
+1 is added to the 4 closest pixels of the current map
(a)
Region of influence of the central pixel
d_2
d_1
Bounding Box of the primitive to be tested
($d_1 < d_2$ compulsory)
(b)

Figure 4.9: Region Of Influence in HOM. When a pixel is determined, the four closest pixels (including itself) are deemed to be in the region of influence. These four pixels can cause an increment in upto four upper level HOMs.

mid-point of projection's bounding box needs to be tested. If the concept of region of influence were not used, four pixels corresponding to the extremities of the bounding box would need to be tested.

Since the maximum number of updates to a HOM pixel is 16 and since the number of HOM pixels is a third of that of the original image ($1/4 + 1/16 + 1/64 + ... = 1/3$), the updating process requires at most $16p^2/3$ steps, where p^2 is the image size, leading to an $O(1)$ complexity per pixel for updating the HOM. It is to be noted that the complexity is $O(1)$ with respect to number of triangles and not number of pixels.

4.4.3 Occlusion Testing using HOMs

To perform the occlusion test, the exact HOM pixel corresponding to the pixel extent is tested. Once the pixel has been identified, the actual occlusion determination is just a single comparison against 16. When the HOM pixel's value is equal to 16, the space corresponding to the HOM pixel and resultantly the pixel extent is fully occluded, otherwise it is not. Occluded nodes and occluded parts of triangles do not need to be processed.

The most complex part of the test is determining the exact pixel to test. Two bits of information are needed – the level of the HOM and the bit of the HOM to check in the level. Both of these are calculated using the vertices of the node's bounding box.

The bounding box is determined by using the projection of the node / triangle being tested. The minimum and maximum values of the X and Y coordinates of the projection gives the necessary bounding box.

Finding the level of the HOM The level of the HOM is dependent on the maximum of the two lengths of the bounding box. The structure of the HOM is such that in the first level, each pixel indicates two X coordinates and two Y coordinates. At the second level, it indicates four coordinates each and so on. Hence, the level is given by the log_2 of the greater of the two extents. The formula used for determining the correct level is simply given by $l = Integer(log_2(d_l))$, where d_l is the longest edge (in pixels) of the box and l is the map level. l must also be clamped to $[0..log_2(N) - 2]$ since the bounding box may either smaller than a pixel or larger than the image size N. The log_2 of a floating point number can be determined efficiently by using the IEEE floating point representation's exponent part that gives the power of two just below the floating point number being represented. The expressions below provide the implementation to determine the HOM level to be tested.

```
dl = max(xExtentLength, yExtentLength)
HOMLevel = Exponent(dl)+1
```

Listing 4.4: Determining the HOM level. Exponent function uses the IEEE representation of a floating point number to easily determine the exponent.

Finding the Exact Pixel to Test at a Level Once the level has been determined, the pixel at that level is necessary. For this, the mid-point of the node projection's bounding square is used as the representative pixel. At each level, each coordinate is halved – for eg., if the pixel to be tested is $(100, 200)$, then at the first level the pixel to be tested would be $(50, 100)$. At the next levels, it would be $(25, 50)$, $(12, 25)$ and so on. It can be noticed that the pixel at the required level is the original pixel divided by $2^{HOMLevel}$. Since integer divisions by powers of 2 are sufficient, it can be achieved with just a *bitwise right shift* operation. The X and Y coordinates of the pixel to be tested at the required level are computed in the implementation as shown below.

```
HOMPixelX = x >> HOMLevel
HOMPixelY = y >> HOMLevel
```

Listing 4.5: Determining the HOM pixel to check using bit shift operators.

Once both the level and the pixel in the map are determined, it is simple to detect occlusion. If the pixel thus found has a value of 16, then the node or triangle part is occluded. Otherwise it is considered as not occluded and hence processing continues. It may be observed that the occlusion test involves very few operations.

HOMs are an integral part of the tree traversal process. Occluded nodes are easily detected using them and these nodes can be skipped. The traversal continues down the tree, skipping occluded and invisible nodes to finally reach the leaf nodes. At the leaf nodes, the triangles in them may be a part of the final image. Determining the parts of the triangle that are part of the final image is undertaken by the leaf node processing part of the algorithm.

4.5 Leaf Node Processing – Recursive Rasterisation

The tree nodes are tested for visibility, as described in the preceding sections, until either they are determined as being occluded or until a leaf node is reached. When a leaf node is reached, the node is fully or partly visible from the viewpoint. Consequently, some or all of the geometry it contains may also be visible from the viewpoint. The parts of the triangles in the leaf node are tested for occlusion until the pixel level and if they are not occluded, then the pixels they occupy are determined, coloured and shaded accordingly.

At the highest level the algorithm can be described with the pseudocode below.

```
ProcessLeafNode()
{
  tList = triangles in leaf node
  for(each triangle t in tList)
  {
    polygon = ClipWithBoundingBox(t, nodeBB);
    polyTrs = polygon.SplitIntoComponentTriangles();
    for(each component triangle cTr in polyTrs)
      RecursiveRasteriseTriangle(cTr.p1, cTr.p2, cTr.p3);
  }
}
```

Listing 4.6: Leaf node processing in *Coherent Rendering* algorithm. When a leaf node is reached, each triangle's part that is in the frustum is recursively rasterised.

The pseudocode shows the main components in determining the contents of the final image. It is to be recalled that only scenes consisting exclusively of triangles are considered. Every triangle identified and accessed (as will be shown in Appendix A.2.2) must be clipped with the bounding box of the node. This is to ensure that the parts of the triangle that are outside the node are not considered. The clipping is performed using a modified version of the Sutherland-Hodgman clipping algorithm [SH74].

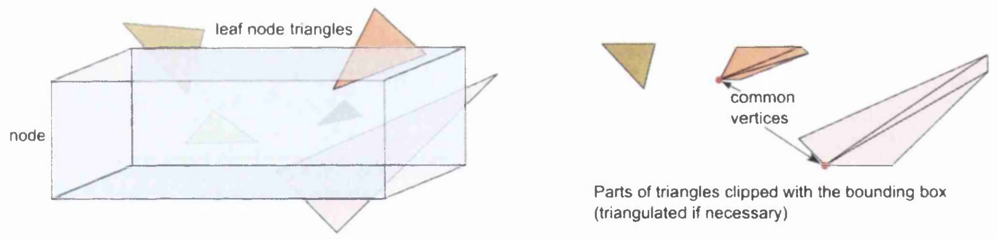

leaf node triangles
node
common vertices
Parts of triangles clipped with the bounding box (triangulated if necessary)

Figure 4.10: Leaf node triangles needing clipping due to partly enclosed triangles. To ensure accurate visibility, only parts of triangles that are fully contained by the node are to be considered. If other parts are considered, they result in rendering artifacts.

As Figure 4.10 shows, clipping these triangles leads to polygons that may not be triangles. These polygons are then broken up into their component triangles using a very simple method. One of the vertices of the polygon is considered as the common vertex and a fan of triangles is created with this vertex as the common vertex of all the triangles. The triangulation method is illustrated in Figure 4.10. This method, however, is not the best method for triangulation and leads to triangles of poor aspect ratio. Other methods wherein a point is placed in the centre of the polygon and connected to each vertex would result in better quality triangles. But, for the purposes of our algorithm, the triangulation method used by us provides satisfactory results.

Subsequent to clipping and triangulating, the component triangles are rasterised. The first step is to project the three vertices of the triangle to find the pixels that the triangle vertices occupy in image space. The projection method is computed as described in Section 4.3.1 – i.e., by multiplying the point by the global transformation matrix and converting the coordinates from OpenGL space to image space. The pixels affected by the triangle are then determined with a subdivision method as described by the pseudocode below.

```
RecursiveRasteriseTriangle(p1, p2, p3)
{
  minX = min(p1[x], p2[x], p3[x])
  maxX = max(p1[x], p2[x], p3[x])

  minY = min(p1[y], p2[y], p3[y])
  maxY = max(p1[y], p2[y], p3[y])

  minZ = min(p1[z], p2[z], p3[z])
  maxZ = max(p1[z], p2[z], p3[z])

  if(maxX < 0 || maxY < 0 ||
     minX > imageWidth  || minY > imageWidth
     || maxZ < -1 || minZ > 1)
  {
    //if triangle is outside image bounds
    return;
  }
  if(triangle is fully between 4 pixels)
    return;
  visible = CheckVisibilityHOM((minX + maxX)/2,  (minY+maxY)/2);
  if(!visible)
    return;
  if(maxX-minX < 1 && maxY-minY < 1)
  {
    setPixel(maxX, maxY);
    update HOM;
    return;
  }
  l = LongestEdge();
  p4 = PointOfTriangleNotIn(p1, p2, p3, l);
  middle = MidPoint(longestEdge);
  RecursiveRasteriseTriangle(longestEdge.p1,
                             longestEdge.p2, middle);
  RecursiveRasteriseTriangle(middle, p4, longestEdge.p1);
}
```

Listing 4.7: Recursive rasterisation of a triangle. This is a subdivision algorithm that divides the triangle recursively until it casts a projection on only one pixel, at which point the triangle can be attributed to the pixel.

The first step of the recursive process is to find the bounding box of the triangle being considered. This is determined as the minimum and maximum coordinates along each axis.

Once this is determined, if the bounding box is outside the frustum, the triangle cannot be in the final image. A test is undertaken to determine if the bounding box is within the bounds of the image. If the bounding box's maximum coordinate is less than zero, in which case the entire bounding box and consequently the triangle is outside the image. Similarly, if the minimum coordinate of the bounding box is greater than the image width / image height, then the triangle is outside the image. The bounds for the Z coordinate are -1 and 1 which are the depths of the near plane and the far plane. Parts of the scene not between these planes, indicated by Z values > 1 and < -1, are discarded.

If the triangle is determined as being within the image boundaries, the next test it has to undergo is the visibility / occlusion test. *Hierarchical Occlusion Maps* – described in Section 4.4 are again used to verify the triangle's visibility. The HOMs indicate if the pixels occupied by the triangle have already been rendered. Due to the front-to-back traversal order of nodes, if a pixel has already been rendered, it implies that for this pixel, the first triangle along the ray (conceptually) has been determined. Hence, no other triangle can be visible at this pixel. The HOMs allow easy determination of this condition.

When a triangle is not small enough to determine just one pixel, it is split into two smaller triangles. The triangle is divided into two smaller triangles at the mid-point of the longest edge of the triangle. The function is then recursively called for the two split triangles until the extent of the triangle is less than one pixel wide and one pixel high. Through each subdivision, the HOMs are tested to see if the split triangle is occluded. When the subdivided triangle's extents are less than than one, the pixel overlapped by the triangle extent is given the colour of the triangle.

Figure 4.11 shows the rasterisation process for a few different triangles.

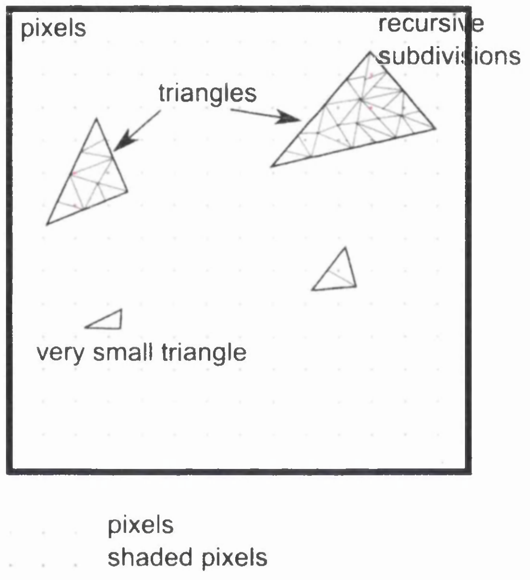
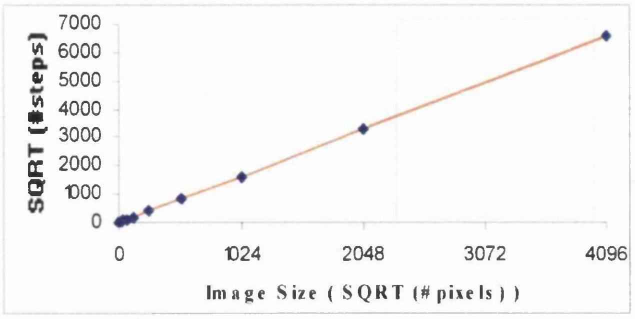


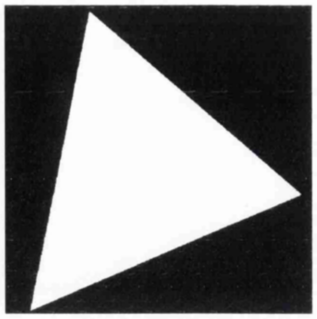
Figure 4.11: Rasterising triangles. A subdivision method is followed where the triangle is subdivided recursively until they span a single pixel. At this pixel, the triangle is determined as being visible.

Simplicity is favoured while implementing this algorithm. Similar ways to perform hierarchical rasterisation [War69] [Gre96] [GGW98] could have been considered, but the simpler method described was favoured. A crucial property of hierarchical rasterisation is that its complexity (i.e., the number of recursive calls) is proportional to the number of pixels of the projected triangle. Figure 4.12 shows the square root of the number of recursive calls for a single triangle projection according to one dimension (image width / image height) of different image sizes, demonstrating a perfect linearity of the algorithm. The square root is needed since a single dimension of the image is plotted on the X axis.

The recursive function is, on average, called approximately 8 times per pixel. The recursion



(a)



(b)

Figure 4.12: Linearity of the hierarchical rasterisation algorithm. The number of recurive rasterisation calls to rasterise the triangle (right) at different resolutions (from 8^2 to 4096^2) has been shown. It can be seen from the graph (left) that the number of calls increase linearly according to image size.

stops only when the size of the bounding box of the triangle is less than a pixel wide, but many subdivided triangles end up between pixels, making shading impossible and increasing the per pixel cost.

The process identifies the triangle projecting onto each pixel. When all the triangles for all the pixels have been identified, the visibility at every pixel is determined. The image can be rendered by shading the pixel depending on the position of the light source. A very simple scheme of shading is used where the dot product of the triangle normal and the ray is taken and the pixel is shaded accordingly. Thus, using the method described in the above sections, the visibility at each pixel is determined and using a basic shading process, an image is generated.

4.6 Results

Although the algorithm's main motivation is to investigate the possibility of a better complexity, it was expected that the algorithm would be competitive with packet ray tracers. Hence, in addition to complexity results, the absolute performance results in comparison to packet ray tracers are provided.

However, it is to be noted that, so far, no particular effort has been made to optimise the different parts of the algorithm. The main goal of *Coherent Rendering* is to investigate performance as scene sizes and images sizes increase. Also, kd-trees built using the space median heuristic have been chosen, since it appeared to be faster in most cases than the surface area heuristic.

4.6.1 Empirical Complexity

The algorithm has been empirically tested with different datasets on an AMD Athlon X2 3800+ (2 GHz, 2 cores, 512 KB cache per core) with 2 GB of memory available. All our algorithms are single-threaded and running on a 32-bit operating system.

Synthetic Benchmark For the first test, a synthetic plane (Figure 4.13(a)) was subdivided to see the evolution of the rendering times according to the number of triangles. A plane with 2 triangles is used as the base model and is subdivided to obtain scenes with 2^{s-1} triangles, where s is the number of subdivisions. In addition, 15 copies of the same subdivided plane were added behind the original one and the per pixel rendering times for both the single and multi plane scenes were studied. A unique viewpoint – shown in Figure 4.13(c) – was used.

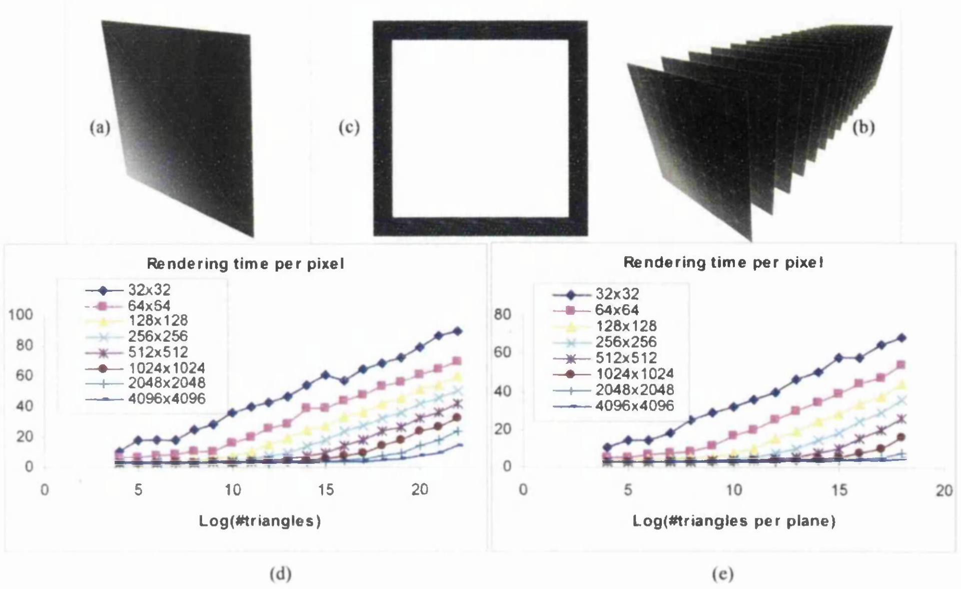


Figure 4.13: Synthetic Benchmarks. (a) original single plane mesh (no subdivision). (b) 16-plane scene subdivided 6 times. (c) is the final image obtained from all synthetic scenes rendered using a unique viewpoint. (d) and (e) are the variation of rendering times per pixel (μs) according to the scene complexity (i.e, number of subdivisions) and the image size for single plane and multi-plane respectively. Results show that rendering times are constant until the scene complexity becomes greater than the image complexity. At this point, rendering times become logarithmic. The similarity of (d) and (e) indicates that only the visible triangles are relevant.

Results show that for a fixed image size, the per-pixel rendering time is constant until a given point after which a logarithmic complexity takes over. Curves are also separated by a horizontal distance of two, indicating that every time the image size is doubled, this point moves 2 points further. Finally, the curves for both sets of scenes perfectly matches, which proves that the occluded triangles added to the scene do not affect rendering times and complexity at all, reflecting the efficiency of HOMs.

The per-pixel average complexity observed here can be summarised as: $O(max(1, log_2(visibletriangles) - c.log_2(imagesize)))$ where c – is a fixed constant depending on the implementation.

This result is important since it provides evidence that complexity of *Coherent Rendering* is now expressed as a function of two variables. In comparison, a regular ray tracer (not using coherence) is known to achieve $O(log_2(Total\ number\ of\ triangles))$ here.

For real-world datasets, results are unfortunately not so reliable, but it can be shown that increasing the image size still tends complexity to $O(1)$.

Benchmarking Common 3D Scenes The test scenes are summarised in Figure 4.14, with quantitative results compiled in Figure 4.15. Rendering times are expressed for the given viewpoints, which include (very) basic shading. To demonstrate the null contribution of hidden objects, the *Sponza* scene has been rendered from inside and outside, and three Stanford models have been added to this scene as well.

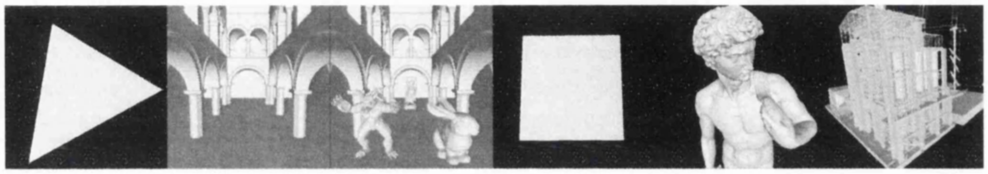


Figure 4.14: Mesh-based *Coherent Renderings*. From left to right: *Single triangle*, *Sponza* , *Sponza with Stanford Models*, *Sponza* from outside, *David* and *Powerplant*. All can be rendered at approximately the same speed, provided that the resulting image is large enough.

The most important result is visible in Figure 4.15(b), where rendering times have been divided by the number of non- black pixels to compensate for the footprint size of each model. All rendering times converge to a reference cost C, irrespective of the number of triangles in the scene. For instance, the *Powerplant* (12 million triangles) and the *Single Triangle* scene rendering times (after footprint normalisation) are less than 30% apart when rendering an 8192^2 image. If the algorithm was purely logarithmic, this ratio would have been more in the order of 23.

To better understand this property, numbers of calls to two critical functions – the tree traversal and the rasterisation functions – have been measured as well. Similar to rendering times, the number of rasterisation steps (Figure 4.15(d)) converge to an approximate constant of 8 steps per pixel, which demonstrates the average $O(1)$ rasterisation complexity per pixel. However, the equivalent operation in ray tracing (i.e., intersection test) is likely to be $O(1)$ if the right tree is used. Therefore, the number of traversal calls made by the algorithm must be analysed as well. With regular ray tracing, this number obviously increases linearly with the number of rays, as visible in [WFMS05]. In *Coherent Rendering*'s case (Figure 4.15(c)), the number of traversal steps is more or less independent of the image size, but obviously depends on the scene itself. This is where the complexity improvement comes from. It must be added that for all renderings except the *David* scene, the number of traversal steps are less than a million, which is a very low when compared to the numbers of cells traversed by other techniques such as those reported in [RSH05] [WIK$^+$06] for similarly sized scenes.

To get a better picture of the redundant nodes traversed by a regular ray tracer, a comparison between both methods is shown in Table 4.2. For every scene, two trees are constructed with different splitting heuristics. These trees are used by both rendering algorithms. A single ray tracer was used, but a 4×4 packet, even in the ideal case, would only divide these numbers by 16 and cannot reach the efficiency of *Coherent Rendering* in terms of nodes traversed. This clearly suggests that there is still some room for optimising node traversal in ray tracing.

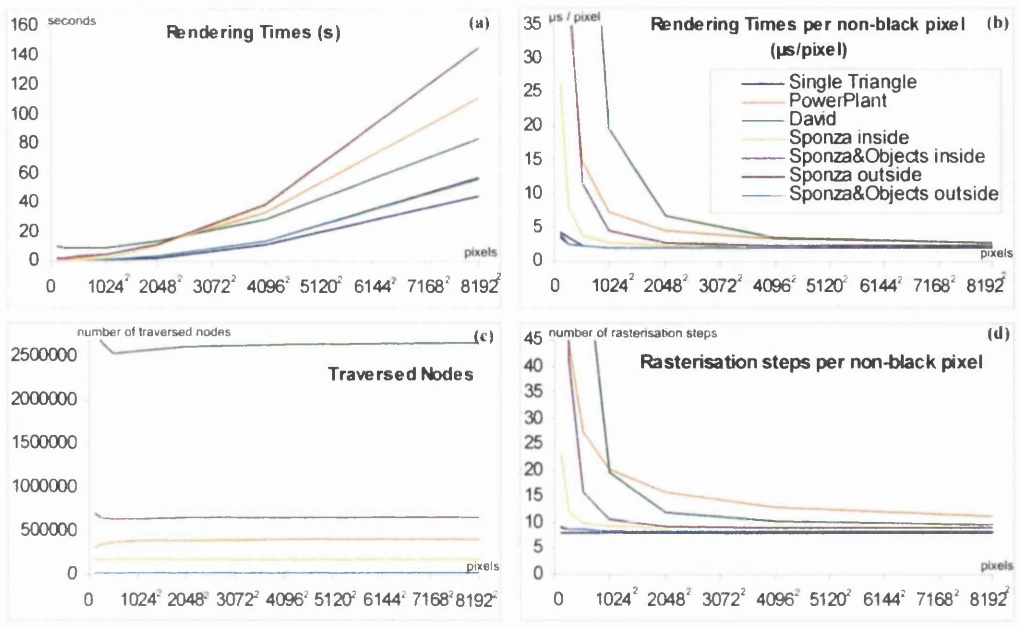


Figure 4.15: Analysis according to the image size. (a) Rendering times for the different scenes. (b) Rendering times per non-black pixel showing convergence to the same rendering time. (c) Number of nodes traversed – independent of the image size. (d) Number of rasterisation steps per non-black pixel – convergence to the same rasterisation cost per pixel. Note that the two *Sponza outside* curves overlap on all graphs.

	RT (Space Median Tree)	RT (SAH Tree)	*Coherent Rendering* (Space Median Tree)	*Coherent Rendering* (SAH Tree)
Single Triangle	1	1	#1	#1
PowerPlant	75	59	0.382	1.41
David	68.5	51	2.5	4.67
Sponza	87	61	0.176	0.169
Sponza and models	109	72	0.642	0.23
Sponza out	5.2	5.5	0.01	0.01
Sponza and models out	5.35	5.5	0.01	0.01

Table 4.2: Number of nodes (in millions) traversed by a recursive ray tracer (without packets) and the *Coherent Rendering* algorithm for a 1024×1024 image.

The *Sponza* case requires particular attention. It is clear that adding several objects to the scene decreases the convergence rate significantly, as long as those objects are visible in the final image. By using a viewpoint outside the *Sponza* atrium (though the *Stanford* models are still inside the frustum), the visible part of the mesh is considerably simplified. Convergence is then very early, and adding or not adding the *Stanford* models (1.5 million triangles) inside the atrium does not make much difference (curves from both datasets overlap on the graph). The number of nodes traversed for the *David* scene is large when compared to the *Powerplant* in which there is a high degree of occlusion. All in all the experiments show that the convergence speed is directly linked to the complexity of the visible part of the mesh. This does not change the complexity result itself, but may appear to be an issue in practice. Therefore, the use of on-the-fly simplification algorithms such as [WFP+01] [WWZ+06] could possibly be an interesting extension to the algorithm when the visible mesh density is greater than the pixel density.

Finally, the current rendering times are about 10-15 times slower than the state-of-the-art MLRT algorithm, however there is plenty of room for optimisation. The main point is that for reasonable rendering sizes (1024×1024 to 2048×2048), all the per-pixel rendering times but *Davids* are only between one and three times slower than the rendering time of a single triangle. This demonstrates that the complexity assumption is valid and scalable.

4.6.2 Absolute Performance

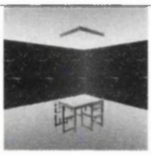
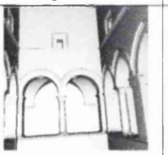
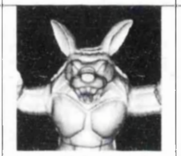
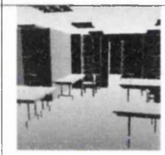
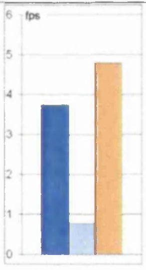
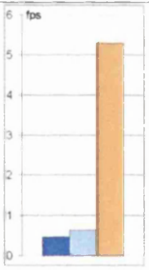
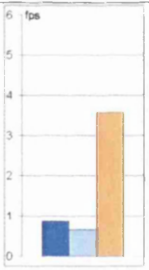
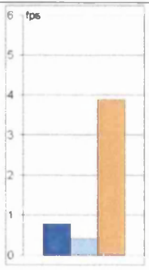
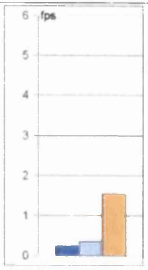
Scene	ERW6	Sponza	Armadillo	Sodahall	Powerplant
scene					
kd-tree maximum depth	12	24	26	27	25
kd-tree leaf node triangles	1	2	2	2	5
Coherent Rendering vs *RT* and *packet RT(8×8)*					
Legend	🟦 Coherent rendering		⬜ Ray tracing single		🟧 Ray tracing packet 8x8

Table 4.3: Performances of *Coherent Rendering* vs single ray tracing and packet ray tracing (8×8 rays) in **frames per second**. It is to be noted that the packet ray tracing implementation has been optimised through the use of SSE instructions. Absolute performance of *Coherent Rendering* is observed to be slower than that of packet ray tracing.

Although the main experiments were performed to verify complexity assumptions, the absolute performance in comparison to packet ray tracers is also considered. To measure the effectiveness of the algorithm, several models have been considered. These models are rendered using the *Coherent Rendering* algorithm described. To determine the fastest performance, the implementation was run on a PC with a Core 2 Quad processor, but single threaded. The algorithm was developed and compiled as a 64 bit application with the Visual Studio 2005 C++ compiler to run on Windows XP 64.

To measure the absolute performance, the scenes shown in Table 4.3 are rendered with ray tracing, packet ray tracing and *Coherent Rendering*. Kd-trees with parameters that differ according to scene sizes are used as the underlying data structure. Details of the kd-tree thus built are also provided. An image size of 1024×1024 is used to benchmark the algorithm.

Table 4.3 provides the results of rendering using this algorithm. The results show the performance of *Coherent Rendering* in comparison to single ray tracing and packet ray tracing with 8×8 rays. They indicate that *Coherent Rendering* is faster than single ray tracing but cannot be competitive with packet ray tracing.

While the absolute performance of the algorithm is not competitive with a packet ray tracers performance, the main point of the algorithm is to investigate its complexity per pixel. The reasons for the slow performance are discussed in Section 4.7.2.

4.7 Discussion

4.7.1 Complexity

As the results demonstrate, *Coherent Rendering* allows convergence to constant complexity per pixel. While *Coherent Rendering* is already very efficient for isosurfaces, more work remains to be done for the triangle mesh version since the algorithm currently suffers from a high constant and many questions are still open. For instance, the motivation for such a technique when fast packet ray tracers [RSH05] exist could be questioned. First, this research is a proof of concept, which verifies that lower average complexities exist for rendering, which has never been demonstrated before.

Using a similar methodology , researchers can investigate empirically whether their technique allows lower complexities or not. A very simple way to do so is to compare rendering times between various scenes and a single triangle scene. The per-pixel times would indicate whether the new algorithm is of lower complexity. This would allow differentiation between pure optimisations and complexity advances and testing whether techniques are optimal and scalable or not. This methodology could, for instance, be applied to the open problem of secondary rays.

There are many areas where the application of the rendering pipeline is less trivial, like direct illumination of the scene. An approximation of direct illumination is possible at the same complexity if a shadow mapping method – that requires rendering an image from the light source(s) – is used. Similarly, this algorithm can be used for global illumination as instant radiosity [Kel97] demonstrates it. However, shadow mapping and instant radiosity just estimate visibility. A fully correct solution would need to consider an irregular grid of pixels for the secondary rays. While it may be possible, it will be scientifically very challenging to develop such an algorithm and to ensure optimal complexity at the same time. Another question is whether this algorithm is portable to graphics hardware or not. For example, some parts of the algorithm like triangle / box clipping are already possible on the latest graphics card generation. In truth, the whole algorithm is a lot closer to hardware-based algorithms and pipelines than ray tracing algorithms. Current graphics hardware manufacturers already implement limited hierarchical rasterisation with hierarchical Z-buffers. For instance, 3-level hierarchical Z- buffer is currently implemented on ATI graphics cards. A full pyramid could actually be more suitable. Fast access to a pixel's Z hierarchy by the CPUs would be necessary to determine visibility. This would require a fast communication channel with low latencies between the GPU and the CPU and a few companies are already working in that direction. It is believed that this would be much more efficient than occlusion queries, and would thus be a great feature in GPUs.

Finally, a worst-case rendering scenario has not been discussed since real-world datasets that would not work have not been found. It is clear that if the HOM pixels cannot become opaque (e.g., the final image is made of many widely spread black pixels), the visibility test will always pass and more nodes will be traversed. Actually, this is the same worst case as with regular ray tracers because the *Coherent Rendering* algorithm theoretically traverses the same nodes as a regular ray tracer.

4.7.2 Absolute Performance

Though, the important motivation of the algorithm is to investigate and demonstrate the better complexity, a usable algorithm needs to be competitive with packet ray tracers. In this area, the *Coherent Rendering* algorithm is slower than packet ray tracers warranting investigations to identify the causes for the relatively poor performance. This would also assist in optimising the algorithm.

Figure 4.16 shows the algorithm profiled (by rendering the *Sponza* scene numerous times). The profiling results show that the recursive process of splitting triangles and identifying the pixels is the main bottleneck. The next major bottleneck is identifying the bits in the HOM and performing the visibility test, followed by the vector subtract, dot product and add operations.

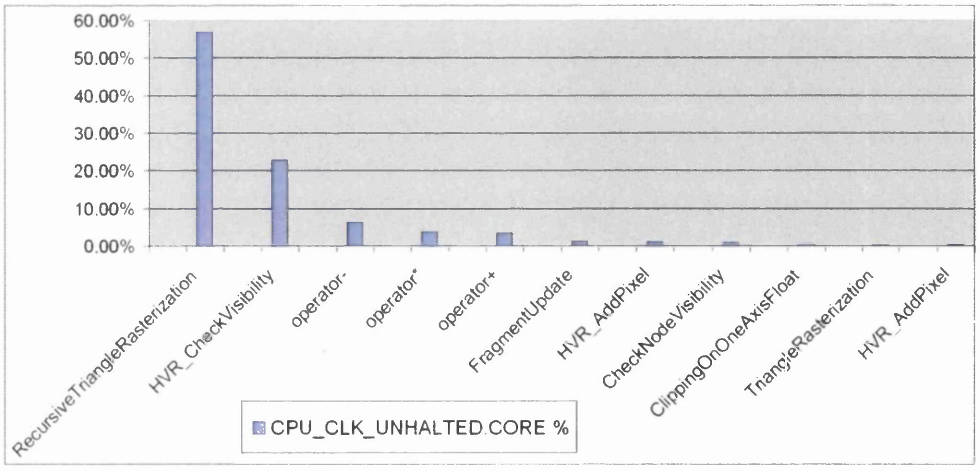


Figure 4.16: Profiling the *Coherent Rendering* algorithm.

The number of recursive rasterisation steps – see Section 4.5 – is quite high as each triangle has to be split several times until the pixel level is reached. Since the rasterisation is done in 2D, the steps are expensive. Since the triangles are split until they span less than a pixel, many triangles – especially large triangles occupying several pixels are split numerous times before the sub-pixel level is reached, at which point, it may be determined that the pixel is not shaded by the triangle in consideration. This causes the recursive rasterisation process to be performed a large number of times, sometimes without effect. The fact that the rasterisation algorithm does not identify multiple pixels in one iteration makes it expensive.

The large number of recursive rasterisation calls also means that each time a triangle is split, the mid-point of the vector has to be found leading to several calls to vector adds. In addition, at every recursive step, the triangle is tested for visibility leading to numerous occlusion determination calls. Due to the fact that the operations are in 2D, they are expensive leading to an expensive rendering algorithm.

One method to alleviate this is to use a simple ray tracing approach at the leaf node level. i.e., when the *Coherent Rendering* algorithm's tree traversal reaches the leaf node, the ray corresponding to the leaf node projection can be intersected with the leaf node's triangles to obtain the right triangle for the pixel. While the algorithm could be faster, it would lose the occlusion test done during the rasterisation process which may be detrimental. This method, though, has not been investigated and could merit further study.

However, the advantage of *Coherent Rendering* is that the number of tree traversals are minimised as evidenced by the profiler results where the tree traversal method does not appear even in the top 11 most time consuming methods. In a ray tracing algorithm, the complexity of triangle intersections in a ray tracer can be assumed to be a constant. However, each ray has to traverse the tree to get to the leaf node, performing an average of $log(N)$ traversals leading to an $O(log(N))$ complexity per pixel. Since in *Coherent Rendering*, the number of nodes traversed is minimal and almost the same for all image sizes, its complexity approaches $O(1)$ when image sizes increase.

4.8 Summary

The *Coherent Rendering* algorithm discussed in the chapter combines concepts from rasterisation and ray tracing to introduce a new rendering algorithm. In a ray tracer, the complexity per pixel is determined as being due to the tree traversals and not the triangle intersections themselves. Packet ray tracers amortise this by traversing the nodes with groups of rays together. However, due to coherence issues, these packets are limited to a certain size – optimal size of 8×8 in our implementation. *Coherent Rendering* takes this approach further by proposing an algorithm where the entire image's pixels are considered as a packet. The nodes visible from the viewpoint are determined. Further, using the concept of *Hierarchical Occlusion Maps*, the early ray termination property of ray tracers is applied to *Coherent Rendering* to perform occlusion detection.

The main reason for investigating the *Coherent Rendering* algorithm is to show that it may result in an $O(1)$ renderer. The results show that as image sizes increase, the rendering times appear to converge to times needed to render a single triangle. On an 8192×8192 pixel image, the rendering time for the *Powerplant* scene (12 million triangles) is around 30% more than the time to render the *Single Triangle* scene. This shows that the method is definitely not logarithmic. It is expected that if the image size is increased further, the difference in this rendering time would be even smaller.

At the same time, the algorithm was expected to be very competitive with ray tracers. However, due to expensive projection and rasterisation operations performed in 2D, the algorithm's performance suffers. Due to this, though the algorithm manages to be faster than single ray tracers for most cases, it fails to compete with packet ray tracers. Through optimisations like the use of SSE, the algorithm can be implemented in a more efficient manner. However, the fact that its performance is not competitive with modern packet ray tracing methods is currently a drawback. Since, the 2D nature of the algorithm has been identified as being the cause of the poor performance, it is believed that an algorithm that performs these operations in 1D would be much faster. The result is the algorithm called *Row Tracing* – discussed in chapter 5.

Chapter 5

Row Tracing

Contents

Row Tracing is introduced as a new visibility / rendering method that is based on *Coherent Rendering*. At the same time, it aims to improve the absolute performance so that it is competitive with packet ray tracers. This chapter explains in detail the method described by us in [KM08] – *Kammaje, Ravi P.; Mora, Benjamin, "Row tracing using Hierarchical Occlusion Maps", IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008., pp.27-34, 9-10 Aug. 2008.*

5.1 Motivation

Recent research has popularised the use of packet ray tracing as a rendering method through the use of groups of rays that have a variety of shapes and sizes [PKGH97] [WBWS01] [RSH05] [Wal04] [WBS07] [ORM07]. The acceleration is brought about by traversing the entire packet of rays, thus amortising the computational costs resulting in a much lower cost per ray. Recent research [RSH05] [WBS07] [BWS06] has enabled the use of larger packets through interval arithmetic.

New methods of data acquisition and sophisticated scanning technologies have resulted in increased scene sizes. For larger scene sizes, the computational complexity of packet ray tracing –

shown to be logarithmic [HB00] [Hur05] [WSS05] [HHS06] [YLM06] – makes it the preferred method of rendering. Packet ray tracing is also trivially parallelisable, allowing it to take maximal advantage of the recent trend towards multi-core processors. This scalability of packet ray tracers over scene sizes as well as over multiple cores has resulted in it being touted as a potential alternative to rasterisation.

Packet ray tracing algorithms work exceptionally when the size of the packets are relatively small. On the other hand, using a larger group of rays that traverse a similar path would maximise cache coherence and reduce memory bandwidth requirements. However, as the number of rays are increased, the coherence reduces, leading to a performance penalty, as the component rays traverse different paths. Thus, to achieve the best performance, an optimal (but relatively small) number of rays per packet is necessary.

Another disadvantage of larger packets is that the early ray termination property of ray tracers that allows in-built occlusion testing can be used only if all the component rays / pixels have already found intersections. For smaller packets, the first hit object for most rays in the packet are determined at almost the same traversal step and traversal can be stopped at this point. However, for larger packets, due to the fact that the rays are far apart the object intersections may occur at widely separated nodes. Thus, the early termination property cannot be effectively used.

During intersection tests, at most four rays can be tested against a single triangle through the use of SIMD / SSE instructions (on current architectures). Packets with a greater number of rays have to be split into smaller groups of four each and intersected with the triangle.

These inefficiencies due to the use of larger rectangular packets of rays point to the possibility that rectangular packets may not be the most efficient grouping of rays to maximally utilise the coherence provided by the data structure.

In Chapter 4, a new algorithm that aims to achieve a lower empirical computational complexity was introduced. However, due to the use of 2D structures, it suffers from expensive intersection, projection and occlusion detection costs making it an unfeasible algorithm to use in its current form.

In both *Coherent Rendering* and packet ray tracing, the main inefficiencies exist due to the fact that rays are grouped into 2D packets instead of a simpler structure. Hence, the idea of using an algorithm similar to *Coherent Rendering*, but with packets spanning a row of the image is considered. The algorithm, called *Row Tracing*, is an attempt at an algorithm that preserves the advantages of both packet ray tracing and *Coherent Rendering* while minimising the disadvantages.

5.2 Concept

Conceptually, *Row Tracing* is a very simple algorithm and is very similar to ray tracing. Instead of tracing rays through a structure like a kd-tree or octree constructed on the scene, entire lines / rows of the image are traced through the structure. At the leaf nodes of the structure, the row is intersected with the triangles of the leaf node to obtain the intersected objects for the affected pixels of the row.

The use of an entire row of the image allows amortisation of traversal and intersection costs. It also allows the algorithm to use the dual property of a row of pixels – that it can be considered either as a 2D plane of rays (as shown in Figure 5.1), or as a 1D line of pixels – at different points of

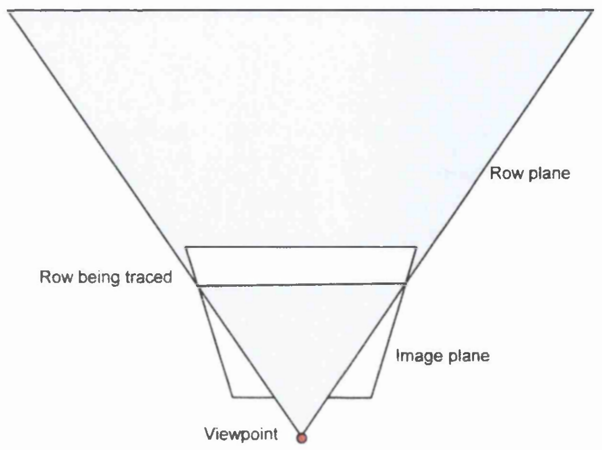


Figure 5.1: A row of the image as a 2D plane. Tracing this 2D plane through the tree is the basic idea of the *Row Tracing* algorithm.

the algorithm. This enables the algorithm to select from several efficient methods for the integral parts of the algorithm.

The row is considered as a 2D plane of rays when traversing through the tree or when intersecting with triangles. Due to plane–box intersections and plane–triangle intersections being cheap, the coherence used does not incur additional costs. Intersecting a single ray with a box or a triangle is of almost similar computational cost to that of intersecting a plane with a box or a triangle. The traversal and intersection cost – already quite low – is amortised across the number of active rays. Further, plane–node intersections and plane–triangle intersections are achieved through the use of simple dot products – vectorisable with SIMD instructions.

At other points of the algorithm, considering the row as a 1D line allows simplification and optimisation through the use of 1D versions of several operations. Primitive clipping, occlusion testing and frustum bounds testing are a few key operations benefiting due to the 1D nature of the row.

A disadvantage of tracing rows as compared to single rays is that it loses the early ray termination property used very effectively for occlusion testing. When a ray is traversed through a space subdivision structure in front-to-back order, the ray traversal is stopped if it has intersected an object, thus eliminating a large number of unnecessary operations. *Row Tracing* loses this ability due to the large span of the packet. To overcome this disadvantage, a 1D version of *Hierarchical Occlusion Maps* (HOMs) introduced by Zhang et al. [ZMHH97] is used. The HOM for a row allows skipping of nodes that overlap already rendered parts of the row – in effect, an early ray termination like test for nodes.

Row Tracing combines features and abilities of ray tracing and rasterisation. The ability of using spatial subdivision data structures is borrowed from ray tracers. Generating the image by scan-lines, projecting points, etc are concepts adapted from rasterisation. This combination, in addition to the use of *Hierarchical Occlusion Maps*, is expected to enable *Row Tracing* to demonstrate performance advantage over current packet ray tracers. As scene sizes increase, *Row Tracing* – due to its (speculated) logarithmic complexity – should show better performance than Z-buffer based visibility (OpenGL) when the scene consists of a large number of triangles. Table 5.1 provides a comparison between rasterisation, ray tracing and *Row Tracing* and it is clear that *Row Tracing* is a hybrid algorithm inheriting properties of both algorithms.

	Rasterisation	Ray Tracing	Row Tracing
Logarithmic Complexity	×	✓	✓
Cheaper per-pixel scanline algorithm	✓	×	✓
Multi-core / Multi-CPU Parallelisation	×	✓	✓
Shadows	✓	✓	✓

Table 5.1: Comparison between Rasterisation, Ray Tracing and *Row Tracing*

For the purposes of the algorithm, it has been implemented with kd-trees and octrees as the underlying data structures.

5.3 High Level Algorithm

The high level *Row Tracing* algorithm is very similar to ray tracing. The image's rows are individually traced through a hierarchical structure. When the traversal ends – the triangle occupying each pixel of the row is determined. When all the rows of the image are similarly traversed, the triangles for each pixel are identified to determine visibility at each pixel. These pixels can then be shaded according to the triangle and the lights.

The high level algorithm is very similar to the high level *Coherent Rendering* algorithm (Listing 4.1). Listing 5.1 shows the algorithm with the major differences between the *Coherent Rendering* and the *Row Tracing* algorithm highlighted in blue.

```
RowTrace()
{
  for(each image row)
  {
    InitialiseRowConstants();
    TraverseTree(row, rootNode);
    Shade row;
  }
}
```

Listing 5.1: High Level *Row Tracing* algorithm. Each row of the image is initialised, traversed down the tree and finally the pixels determined are coloured appropriately. At the high level, the algorithm is very similar to the *Coherent Rendering* algorithm (Listing 4.1). Differences between the two are highlighted (in blue).

The row of the image to be traced is iteratively selected and by an initialisation process the row plane equation and other attributes associated with a row are initialised. Subsequent to this process, the row is ready to be traversed through the data structure. The row traverses the tree in a front-to-back order until it is determined that the node cannot contribute to the row's pixels or until a leaf node is reached. At the leaf node, the row is tested against the leaf node's triangles to ascertain the pixels occupied. Once the triangles have been determined for each pixel of the row, the visibility at each pixel of the row is determined. It is to be noted that in the pseudocode the leaf node processing method is not explicitly called as it is called by the `TraverseTree` method upon reaching a leaf node.

The considerations for the underlying data structure along with the three parts of *Row Tracing* – Initialisation, Tree traversal and Leaf node processing – will be detailed in the following sections.

5.4 Datastructures for Row Tracing

As with ray tracing, rendering times for *Row Tracing* depend on the number of nodes traversed, the number of intersections and cost per traversal and intersection. The total rendering time is given by Weghorst et al. [WHG84] in Equation 2.11. Since the only primitives being considered are triangles, cost of primitive intersection is a constant. Hence, a structure that is well suited for *Row Tracing* minimises the number of nodes traversed with a low per node intersection that at the same time effectively separates the triangles so that the number of triangles to be intersected are low.

Row–triangle intersections are very cheap and are amortised across several pixels and hence, the second term of the cost, $N_{PI} \times C_{PI}$, will be low when compared to the first term. Due to this, *Row Tracing* prefers data structures that minimise the first term, $N_T \times C_T$, even at a slight penalty to the second term. Data structures with axis-aligned boxes can be intersected against a row plane at a low cost [Hof96] and hence are particularly suited for *Row Tracing*. A kd-tree built with the surface area heuristic, with its effective separation of triangles is a good structure for *Row Tracing*. Another structure considered is the octree – due to properties that enable a cheaper cost for intersection, C_T. These two data structures are thus considered.

5.4.1 Kd-tree

A kd-tree is one of the most researched structure for ray tracing. Due to this, there are several well known algorithms and heuristics for the construction and ray traversal of a kd-tree. The most popular heuristic is the *Surface Area Heuristic*(SAH) generally accepted to produce the best kd-trees for ray tracing. However, *Row Tracing* is a different algorithm – the best kd-tree for ray tracing is not necessarily the best for *Row Tracing*. The advantage with an SAH kd-tree is that it quickly separates empty space and selects a locally optimal split position at each step. This partitioning scheme results in a very good tree that should be suitable for use with the *Row Tracing* algorithm.

5.4.2 Octree

The octree is normally not discussed with respect to ray tracing, but it is a very simple structure that is easy to create. In addition, it is a very efficient structure for *Row Tracing* due to its property that each node at a particular depth is a cube of the same size. This property enables the use of a simple optimisation that reduces the cost of row–node intersection, $tNode$, making it a good structure to investigate for *Row Tracing*.

Thus, both the kd-tree and the octree have advantages that make it suitable for *Row Tracing*. In addition, by implementing and demonstrating the effectivity of *Row Tracing* on more than one data structure with axis-aligned bounding boxes, the easy adaptability of *Row Tracing* over such data structures is shown.

The details of the algorithm starting with the tree traversal by the row will be discussed in further sections.

5.5 Tree Traversal

For *Row Tracing* on both the kd-tree and the octree, the row is traversed down the tree in a front-to-back order. Several attributes required for the traversal computations are constants – either for all the rows or for a single row. An initialisation process handles the computation of these attributes.

5.5.1 Plane–Box intersection

Both the kd-tree and the octree have nodes are axis aligned boxes. Hence, each traversal consists of a row plane–box intersection. The method described by Hoff [Hof96], that allows several optimisations will be described in brief before the tree traversal algorithm is described.

A box intersects a plane if its eight vertices lie on both sides of a plane. Naively, this can be found by calculating the signed distances of the eight vertices to the plane. However, as Figure 5.2 shows, it is sufficient to test two vertices. These two vertices are the extremities of the box in relation to the plane. Figure 5.2 shows that these two vertices are the vertices that have the minimum and maximum projection onto the normal of the plane. Thus, it is sufficient to test if these two points are on the same side or different sides of the plane.

Hoff also explains the determination of these two points. If **N** is the normal of plane, the vertex at the extremity in the direction of the normal is given by the pseudocode below:

```
DetermineMaxVertex()
{
  if(N.x>0) //RIGHT
    if(N.y>0) //RIGHT,TOP
      if(N.z>0) //RIGHT,TOP,FRONT
        v1 = RIGHT, TOP, FRONT
      else
        v1 = RIGHT,TOP,BACK
    else //RIGHT,BOTTOM
      if(N.z>0)
        v1 = RIGHT,BOTTOM,FRONT
      else
        v1 = RIGHT,BOTTOM,BACK
  else //LEFT
    if(N.y>0) //LEFT,TOP
      if(N.z>0)
        v1 = LEFT,TOP,FRONT
      else
        v1 = LEFT,TOP,BACK
    else //LEFT,BOTTOM
      if(N.z>0)
        v1 = LEFT,BOTTOM,FRONT
      else
        v1 = LEFT,BOTTOM,BACK
}
```

Listing 5.2: Determining the vertex at the extremity in the direction of the normal. **N** – is the normal to the row-plane and **v1** – is the vertex at the extremity.

The vertex, $\mathbf{v_2}$, at the extremity in the negative direction of the normal is just the negative of $\mathbf{v_1}$. i.e., if $\mathbf{v_1}$ is determined as the LEFT, BOTTOM, BACK vertex, then $\mathbf{v_2}$ is the RIGHT, TOP, FRONT vertex.

If $\mathbf{v_1}$ and $\mathbf{v_2}$ are the two vertices with the minimum and maximum projection onto the plane's normal and if the plane's equation is $Ax + By + Cz + D = 0$, then the intersection is determined by using the set of equations below.

$$\begin{aligned} d_v &= Ax_v + By_v + Cz_v + D \\ d_v &= Ax_v + By_v + Cz_v + D \\ intersection &= (d_v == 0) \; or \; (d_v == 0) \; or \; ((d_v > 0)! = (d_v > 0)) \end{aligned} \tag{5.1}$$

Using this method, it is sufficient to calculate the plane-vertex distance for just two vertices. In addition, in any structure with a hierarchy of bounding boxes, like kd-trees, octrees or BVHs, the nodes are all oriented similarly. Hence, these two vertices can be determined during an initialisation process and the appropriate vertex can be accessed when necessary, allowing further optimisations.

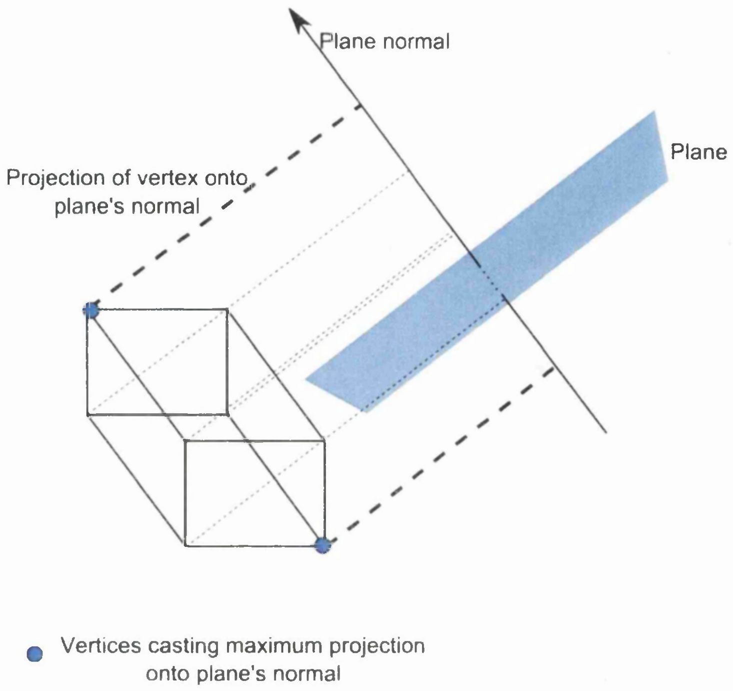

Plane normal
Plane
Projection of vertex onto plane's normal
Vertices casting maximum projection onto plane's normal

Figure 5.2: Intersection of a box and a plane. As shown, it is sufficient to determine if the two extreme points of the box in relation to the normal of the plane are on the same side or either side of the plane.

5.5.2 Initialisation

There are two parts to the initialisation process – one that occurs prior to the tree traversal and one that occurs prior to every row's traversal. The first process computes the global values that remain the same for all the rows. The second process – performed for every row – computes the values relevant to the row being traced.

- Global Values – These are the attributes that do not change across the rows and can hence be considered as global across all the rows. Initialising these just once before the tree traversal starts is advantageous. The attributes considered as global are:
  - Global transformation matrix – This is the transformation matrix that is used to convert a point from model space to image space.
  - Near plane – This is the plane upon which the image is rendered. It is variant upon the viewpoint used and is constant for all the rows. The four vertices of this plane are determined by applying the global transformation to the four vertices of the OpenGL near plane coordinates. Subsequently, the equation of the near plane is found by finding the normal to the near plane. If the near plane is given by $A_{np}x + B_{np}y + C_{np}z + D_{np} = 0$, the normal vector provides the values of A_{np}, B_{np} and C_{np}. Plugging these and the values of x, y, and z from one of the near plane's vertices into the equation, the value of D_{np} is computed and is stored.
  - Indices of the node's vertices casting maximum projection on near plane normal – In

order to find if a node is in front of the near plane, it is more efficient to find the two vertices of the node that casts the maximum projection onto the normal of the near plane, as described in Section 5.5.1. The same two vertices of every node casts this maximum projection. The indices of these vertices, $index_{np}$ and $index_{np}$, are found and saved.

- Indices of node's vertices casting maximum projection on rows 0 and 3 of matrix – As will be shown in the following sections, finding the node projection's overestimate is an essential part of *Row Tracing*. To find this with minimal cost, row 0 and 3 of the global transform matrix are considered as vectors and the node's vertices at the end point of the diagonal casting the maximum projection onto these rows are determined as given by Section 5.5.1. Indices of these vertices are stored.

- Per row constants – The attributes that vary for every node need to be initialised at the beginning of that row's traversal. The attributes initialised at this time are:
  - Row Plane Equation – This is a plane of rays with the viewpoint and the two end points of the row defining the plane. The plane equation is determined by calculating the normal that provides the corresponding A_r , B_r , C_r values and then calculating the value of D_r for the row.
  - Indices of node's vertices casting maximum projection on row plane normal – As an intersection is to be calculated for each node and the row during traversal, the indices of the diagonal casting the maximum projection onto the row plane's normal is determined and stored.

5.5.3 Tree Traversal Algorithm

The various variables corresponding to a row are pre-computed by the initialisation process so that the tree can be efficiently traversed in a front-to-back order. Traversal is similar to ray traversal – it starts from the root node of the tree and continues in a front-to-back order. It can be written with the following pseudocode.

```
TraverseTree(row, node)
{
  if(node is empty or
      node does not intersect row)
    return;
  proj = FindNodeProjection();
  if(proj is outside frustum or
      proj is occluded)
    return;
  if(node is a leaf node)
  {
    ProcessLeafNode(node);
    return;
  }
  for(each childNode sorted in
        front-to-back order)
    TraverseTree(row, childNode);
}
```

Listing 5.3: Single Row traversal algorithm.

From the pseudocode, it can be observed that traversal stops if the node does not intersect the row, or if the node projection falls outside the frustum or if the node projection is determined as being occluded. Otherwise, traversal continues down the tree until it reaches a node that meets the exit criteria, or until a leaf node is reached. At the leaf node, the triangles contained by the leaf node are rasterised onto the pixels of the row. At the end of the traversal process, the closest triangle corresponding to each pixel of the row is determined.

As mentioned, the traversal process tests the node for several criteria. In order to perform these tests as efficiently as possible, fast methods for row-plane–node intersection and node projection are determined. These methods are described in the following sections. Before that however, to better explain the node projection operation, the well known process of projecting a point onto the screen / row in the context of *Row Tracing* is provided.

5.5.4 Projection of a Point onto the Row

Projecting a point onto the row – i.e, converting the point from the object space to the image space to find the actual pixel occupied by the point on the row – is achieved by multiplying the point by the transformation matrix and converting these co–ordinates from OpenGL space to screen space. However, in *Row Tracing*, since the Y coordinate is fixed (by the row being traversed), only the X and Z coordinates are necessary.

If p is the point to be projected, the value of X and Z coordinates can thus be found using

$$\begin{aligned} x &= \mathbf{p}.\mathbf{m_0} \\ z &= \mathbf{p}.\mathbf{m_2} \\ w &= \mathbf{p}.\mathbf{m_3} \\ x &= (x/w) * halfWidth + halfWidth \\ z &= (z/w) * halfWidth \end{aligned} \tag{5.2}$$

where $\mathbf{m_0}$, $\mathbf{m_2}$ and $\mathbf{m_3}$ – are the first, third and fourth row of the global transformation matrix respectively $halfWidth$ – is half of the image's width x, z, w – are the values of the X coordinate, the Z coordinate and the homogenous coordinate respectively The values of x and z thus found are the coordinates of $\mathbf{p}$ on the screen.

5.5.5 Kd-tree Node Projection

The naive and most obvious method of finding a node's projection is to project the eight vertices onto the row. Projecting a point was described by Equations 5.2. For conducting the node occlusion test, only the X coordinate is necessary. Hence, the naive projection calculation can be shown as:

$$\begin{aligned} x_1 &= \mathbf{v_1}.\mathbf{m_0} \\ x_2 &= \mathbf{v_2}.\mathbf{m_0} \\ &\cdots \end{aligned}$$

$$
\begin{aligned}
x_8 &= \mathbf{v_8.m_0} \\
\\
w_1 &= \mathbf{v_1.m_3} \\
w_2 &= \mathbf{v_2.m_3} \\
&\;\; . \;\; . \;\; . \\
w_8 &= \mathbf{v_8.m_3} \\
\\
x_1 &= (x_1/w_1) * halfWidth + halfWidth \\
x_2 &= (x_2/w_2) * halfWidth + halfWidth \\
&\;\; . \;\; . \;\; . \\
x_8 &= (x_8/w_8) * halfWidth + halfWidth \\
\\
x_{min} &= min(x_1, x_2, .., x_8) \\
x_{max} &= max(x_1, x_2, .., x_8)
\end{aligned}
\tag{5.3}
$$

$\mathbf{v_1..v_8}$ are the eight vertices of the node.
$x_1..x_8$ are the X coordinates of the eight vertices of the node.
$w_1..w_8$ are the homogenous coordinates of the eight vertices of the node.
x_{min} and x_{max} are the extremeties of the actual projection of the node.
$\mathbf{m_0}$ and $\mathbf{m_3}$ are the first row and the fourth row of the global transformation matrix.

However, as Equations 5.3 show, this method is computationally exorbitant, requiring 16 dot products, eight divides and several further operations. It can be observed that to perform an occlusion test or a frustum bounds test, the exact node projection is not necessary. As shown in Figure 5.3, a slight overestimate of the node's projection works almost as well. Using the overestimate results in a few more traversals than otherwise, but if the overestimate is small enough, the trade-off results in hugely reduced cost for traversal. The method to compute an overestimate is discussed.

We first define x_1, x_2, w_1 and w_2 as follows:

$$
\begin{aligned}
x_1 &= min(\mathbf{v_1.m_0}, \mathbf{v_2.m_0}, ..., \mathbf{v_8.m_0}) \\
x_2 &= max(\mathbf{v_1.m_0}, \mathbf{v_2.m_0}, ..., \mathbf{v_8.m_0}) \\
w_1 &= min(\mathbf{v_1.m_3}, \mathbf{v_2.m_3}, ..., \mathbf{v_8.m_3}) \\
w_2 &= max(\mathbf{v_1.m_3}, \mathbf{v_2.m_3}, ..., \mathbf{v_8.m_3})
\end{aligned}
\tag{5.4}
$$

$\mathbf{v_x}$ and $\mathbf{v_x}$ can be defined as the vertices casting the maximum projections on $\mathbf{m_0}$ and can be determined using the method given in Section 5.5.1. Hence, $\mathbf{v_x}$ $.\mathbf{m_0}$ and $\mathbf{v_x}$ $.\mathbf{m_0}$ provide values of x_1 and x_2. Similarly, $\mathbf{v_w}$ and $\mathbf{v_w}$ can be determined as the vertices of the node casting the maximum projection onto $\mathbf{m_3}$. Using these, the minimum and maximum values of x can be determined as follows:

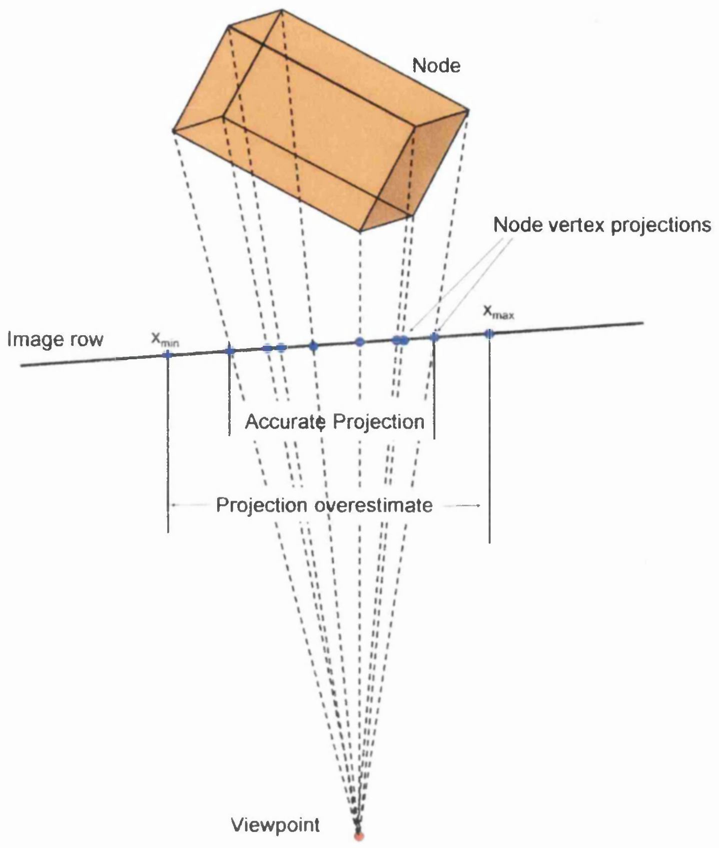


Figure 5.3: Node Projection onto a Row. x_{min} and x_{max} indicate the node's overestimate. If the overestimate is occluded, then the actual projection is also occluded. Hence, this overestimate can be used for occlusion detection.

$$\begin{aligned} x_1 &= \mathbf{v_x} . \mathbf{m_0} \\ x_2 &= \mathbf{v_x} . \mathbf{m_0} \\ w_1 &= 1/(\mathbf{v_w} . \mathbf{m_3}) \\ w_2 &= 1/(\mathbf{v_w} . \mathbf{m_3}) \end{aligned} \tag{5.5}$$

Since, we are unclear if $x_1 > x_2$ and if $w_1 > w_2$, and also since x_1, x_2, w_1 and w_2 maybe negative or less than 1, we calculate the minimum and maximum values of x as follows:

$$\begin{aligned} x_{min} &= min(x_1 w_1,\ x_1 w_2,\ x_2 w_1,\ x_2 w_2) \\ x_{max} &= max(x_1 w_1,\ x_1 w_2,\ x_2 w_1,\ x_2 w_2) \\ x_{min} &= x_{min} * halfWidth + halfWidth \\ x_{max} &= x_{max} * halfWidth + halfWidth \end{aligned} \tag{5.6}$$

where x_{min} and x_{max} represent the extremities of the node projection's overestimate.

x_{min} and x_{max} are used to find the level and the exact bits needed to check the HOM for occlusion. The details of using the HOM to detect occlusion is provided in Section 5.8. Also, x_{min} and x_{max} are used to test the node for frustum inclusion. When x_{min} and x_{max} do not span any pixel between zero and the width of the image, the node is outside the frustum.

5.5.6 Octree Node Projection

The method used to find the projection of an octree node onto the row is essentially the same as that described for the kd-tree. However, the property of an octree that every node at a given depth is a cube of the same size can be used to optimise the vertex determination calculation. As Figure 5.4 shows, if the mid-point of the octree node is known, then each of its vertices can be found by adding (or subtracting) a vector to the midpoint of the octree. To use this property, the unit vectors to be added to the midpoint to obtain the necessary vertices are pre-computed before the tree traversal. The half lengths of a node's diagonal at every depth of the octree are also pre-computed. At traversal time, using the unit vector and the half length, the corresponding vertices are easily found by vector additions. When SIMD / SSE code is used, a vector addition is achieved in one instruction and is cheaper compared to the expensive indexing operations performed by the kd-tree version.

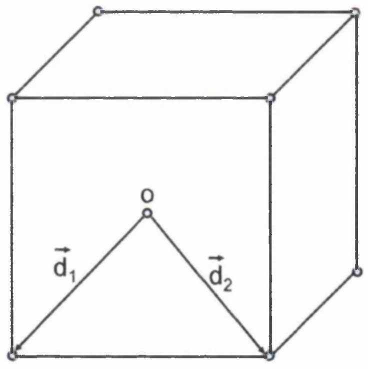


Figure 5.4: Calculating the octree node's vertices. **o** is the mid-point of the octree, $\mathbf{d_1}$ and $\mathbf{d_2}$ are vectors to be added to **o** to obtain the vertices.

5.5.7 Row–Kd-tree Node Intersection

A row–kd-tree node intersection is computed at every traversal step and is one of the most frequently performed operations of the algorithm. Hence, it is important that it is computationally as cheap as possible.

If all eight vertices of a kd-tree node lie on the same side of the row plane, the node does not intersect the row plane. Using the method in [Hof96], intersection can be determined by computing the signed distances of just two vertices (end points of a diagonal that casts the maximum projection onto the row plane's normal). The same diagonal of every node casts this maximum projection onto the row. Hence, the two appropriate vertices can be pre-determined at the beginning of traversal of each row – in the per row initialisation process. This reduces the number of operations significantly. Figure 5.5 shows a diagram that assists in understanding this better.

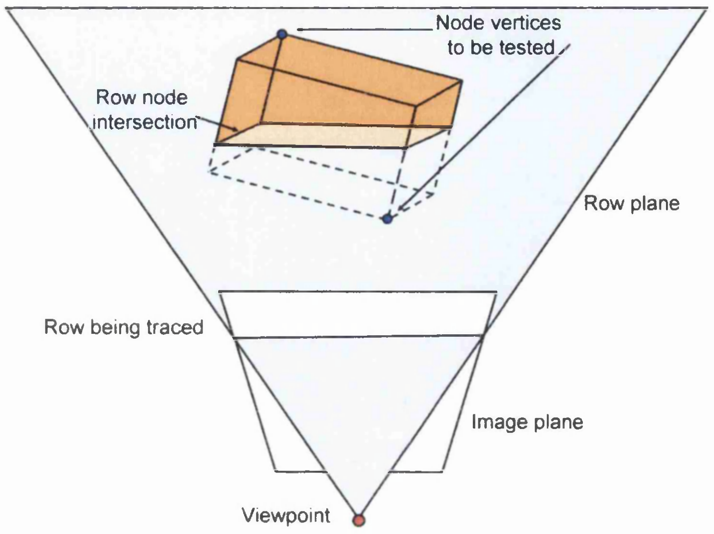


Figure 5.5: Row–Node intersection. If the two extreme vertices of the node with respect to the row plane's normal (shown in blue) are on opposite sides of the row plane, then it intersects the node.

Using the equation of the row-plane, $A_r\ X + B_r\ Y + C_r\ Z + D_r = 0$, the signed distance is calculated by substituting the X, Y and Z values of the vertex into the equation. If d_1 and d_2 are the signed distances of the two vertices, then there is an intersection only if $sign(d_1)! = sign(d_2)$. The operation is very cheap and can be achieved using just two dot products – efficiently calculated with SSE instructions.

5.5.8 Row–Octree Node Intersection

A method that is similar to the Row–kd-tree node intersection (Section 5.5.7) is used to find the intersection between the octree node and the row plane. However, as an optimisation, the property of octrees described in Section 5.5.6 and as shown by Figure 5.4 is again used. Using this, the two vertices are calculated using just a vector addition each instead of an expensive indexing operation.

If a node is determined as intersecting the row, traversal continues down the tree until a leaf node is reached, unless another exit criteria is encountered.

5.6 Leaf Node Processing

At the leaf node, the triangles in it can be a part of the final image. Determining the appropriate parts of these triangles is implemented by the function `ProcessLeafNode` shown below.

```
ProcessLeafNode(node)
{
  for(each Triangle t in node)
  {
    l = Intersect(t, rowPlane);
    l = ClipWithBoundingBox(l);
    p = ProjectOntoImageLine(l);
    pList.add(p);
  }
  extent = MinMax(pList);
  RecursiveRender(extent, pList);
}
```

Listing 5.4: Process the leaf node. When the leaf node is reached during the traversal process, the triangles in it have to be tested against the row to determine which of them contribute to the pixels in the row.

As the pseudocode describes, processing the leaf node involves a few steps that warrant further discussion.

5.6.1 Row–Triangles Intersection

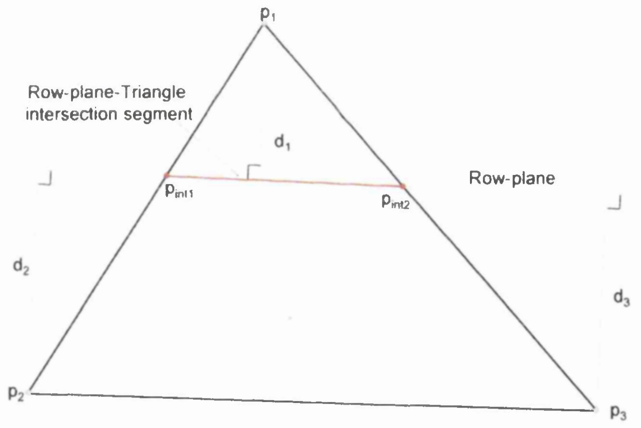


Figure 5.6: Row-plane–triangle intersection. If the vertices of the triangle lie on opposite sides of the row plane, there is an intersection.

The intersection between a row and a triangle is essentially a plane–triangle intersection. It is easily computed using the equation of the row plane. The calculations are better described by the equations below and Figure 5.6. In the equations below, the variables (like p_1, p_2, etc) correspond to the geometric entities shown in Figure 5.6.

$$\begin{aligned} rowEquation => Ax + By + Cz + D &= 0 \\ d_1 &= Ax_1 + Bx_1 + Cz_1 + D \\ d_2 &= Ax_2 + Bx_2 + Cz_2 + D \\ d_3 &= Ax_3 + Bx_3 + Cz_3 + D \end{aligned} \tag{5.7}$$

If d_1, d_2 and d_3 have the same sign, it implies that the entire triangle is on one side of the row plane and hence does not intersect the plane. However, if any one distance is differently signed than the other two, then there are parts of the triangle on both sides of the row plane and hence there is an intersection. If there is an intersection, the end points, p_{int} and p_{int} , of the intersection segment are given by:

$$\begin{aligned} d_{int} &= d_1/(d_1 + d_2) \\ d_{int} &= d_1/(d_1 + d_3) \\ \mathbf{p_{int}} &= \mathbf{p_1} + d_{int}\,(\mathbf{p_2} - \mathbf{p_1}) \\ \mathbf{p_{int}} &= \mathbf{p_1} + d_{int}\,(\mathbf{p_3} - \mathbf{p_1}) \end{aligned} \tag{5.8}$$

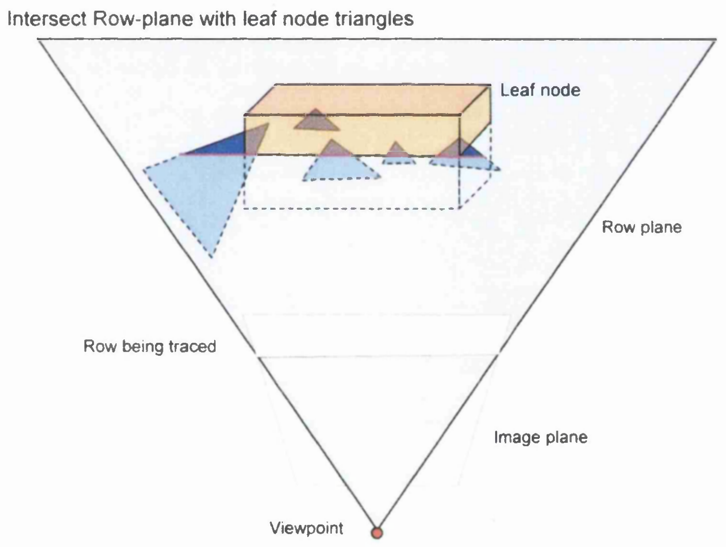


Figure 5.7: Row–leaf-node-triangles intersection. Intersecting the row-plane with the triangles in a node gives several line segments.

The intersection is found for every triangle in the node to obtain a list of line segments. The list is maintained as a list of start and end points of the segments. If the list is not empty, the algorithm proceeds to the next step. Figure 5.7 shows the intersection segments formed by intersecting a row with all the triangles in a leaf node.

5.6.2 Segments Partially in Front

The algorithm should render only parts of the geometry that are in front of the viewpoint / near plane. Relevant parts of the triangles can be on both sides of the near plane only if the node consisting them lies on both sides of the near plane. This is easily determined by finding the signed distances of the two corresponding pre-determined vertices of the node to the near plane. If the signed distances of these points have different signs, the node lies on either side of the near plane and can contain triangles partially on both sides. Only triangle segments in these nodes are to be tested for being partly in front of the near plane.

Segments that are fully in front of the near plane are rendered directly without any additional processing. Similarly, segments that are fully behind the near plane are discarded as they are not part of the final image. However, for the few triangles that have parts in front of and behind the near plane, additional processing is necessary to ensure that only parts that are in front of the near plane are rendered.

Testing whether a triangle lies on both sides of the near plane essentially reduces to a problem of intersection between the near plane and intersection segment. If it is determined that a segment intersects the near plane, the segment is clipped by the near plane and the part that is in front is selected for rendering. If the segment is fully in front or fully behind, then they are rendered as it is or discarded respectively.

At the end of this step, a modified list of segments, containing only the clipped parts that are fully in front of the near plane, is generated.

5.6.3 Clipping Intersection Segments

In a leaf node, parts of the triangle can be outside the node. This can result in intersection segments that may be partly or completely outside the leaf node as seen in the example in Figure 5.7. If these segments are rendered as they are, the resulting image would be incorrect. To ensure that the image consists of the right parts of the right triangles, these segments are clipped against the leaf node's bounding box so that only parts that are inside the node are rendered.

The operation is achieved by using the line equation of the segment and by intersecting the line with the three entry faces and three exit faces of the node. The set of equations below detail the process.

$$\begin{aligned}
t_x &= (bb[x_{entry}] - p_1)/(p_2 - p_1) \\
t_x &= (bb[x_{exit}] - p_1)/(p_2 - p_1) \\
t_y &= (bb[y_{entry}] - p_1)/(p_2 - p_1) \\
t_y &= (bb[y_{exit}] - p_1)/(p_2 - p_1) \\
t_z &= (bb[z_{entry}] - p_1)/(p_2 - p_1) \\
t_z &= (bb[z_{exit}] - p_1)/(p_2 - p_1)
\end{aligned} \tag{5.9}$$

$$\begin{aligned}
t_{entry} &= max(t_x \quad , t_y \quad , t_z \quad) \\
t_{exit} &= min(t_x \quad , t_y \quad , t_z \quad) \\
\mathbf{p_{int}} &= \mathbf{p_1} + t_{entry}(\mathbf{p_2} - \mathbf{p_1}) \\
\mathbf{p_{int}} &= \mathbf{p_1} + t_{exit}(\mathbf{p_2} - \mathbf{p_1})
\end{aligned} \tag{5.10}$$

where t_x , t_x , t_y , t_y , t_z , t_z – are the clipped entry and exit parameters of the intersection segment.
bb – is the array representing the bounding box of the node. It consists of 6 values – the minimum and maximum point along each axis.
$\mathbf{p_{int}}$, $\mathbf{p_{int}}$ – are the extremities of the clipped segment.

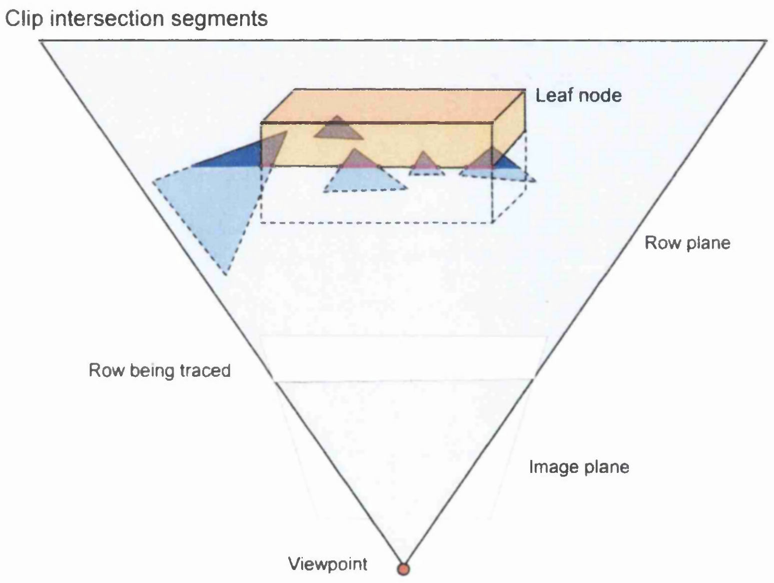


Figure 5.8: Intersection segment clipping. Intersection line segments need to be clipped to ensure that only parts of triangle inside the node are considered.

All segments are clipped against the bounding planes of the node and only parts of these segments that are fully contained by the node are retained. The other parts are discarded. The list of segments is modified accordingly. Figure 5.8 shows the intersection segments clipped by the bounding box of the node.

5.6.4 Projection of Clipped Segments

The list of clipped segments contains the parts of the leaf node triangles that can be a part of the final image. By projecting each clipped segment onto the row and determining the correct visibility, the triangles occupying each affected pixel is found. As shown by Figure 5.9, each segment's two end points are projected onto the row using the method described in Section 5.5.4 to obtain the required projection.

Once the X and Z coordinates of the two end points of the segment are found, any point on the line segment can be found using linear interpolation. The X coordinate is calculated by linear interpolation. For perspective projections, the Z coordinate is computed by linearly interpolating $1/Z$. To simplify the computation, the relation of $1/Z$ with respect to the X coordinate is found as follows.

$$\begin{aligned} b &= 1/Z_1 \\ a &= (Z_1 - Z_2)/(Z_1 * Z_2 * (X_2 - X_1)) \\ 1/Z_x &= aX + b \end{aligned} \tag{5.11}$$

where a and b – are the necessary coefficients.
X_1, Z_1, X_2, Z_2 – are X and Z coordinates of the two end points of the clipped intersection segment.
$1/Z_x$ – is the corresponding $1/Z$ value at any X.

The values of a, b and $1/Z$ are calculated for each segment. These values are stored in a list for use by later parts of the algorithm.

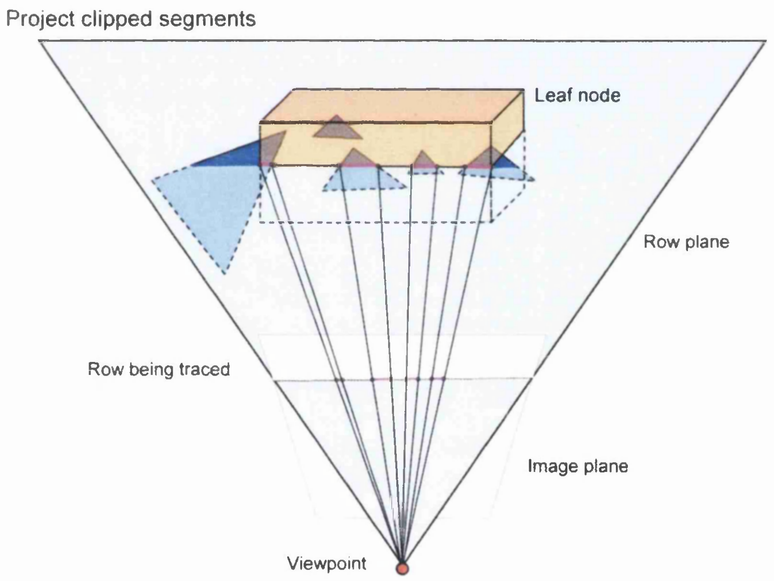


Figure 5.9: Clipped segments projection. The clipped intersection segments are projected onto the row to obtain the pixels affected by the triangles in the node.

5.6.5 Rasterising the Segments

The X, a and b values are used to rasterise the segments – i.e., to determine and shade the pixels affected by the segment. An important consideration is to attribute the right triangle to the pixel by ensuring accurate visibility if several triangles project onto it.

HOMs are used to determine whether pixels under consideration have already been rasterised and are hence occluded by triangles in other leaf nodes. By recursively subdividing the extent of the node's triangles and testing the subdivided extent against the HOMs, the visibility of each subdivided extent is determined. The subdivision and occlusion tests are continued until the subdivided extent is contained fully by an eight pixel group determined by the HOM. Subdivision is stopped at this point as it is more efficient to use SIMD instructions for further processing. Visibility determination using HOMs will be discussed in detail in Section 5.8.4.

At the end of subdivision, the intra-node visibility in addition to the inter-node visibility is undertaken. Inter-node visibility order – the visibility order of triangles in different nodes – is determined using the lower levels of the HOM. However, intra-node visibility – accurate visibility of triangles within the node – is determined by using a Z-buffer like algorithm. For the eight pixels, the maximum values of $1/Z$ coordinates is maintained. When a clipped line segment projects onto a pixel, the value of $1/Z$ of this line segment is compared against the existing value. If it is greater, then the pixel's existing triangle is replaced by this triangle and the Z-buffer's value is updated to reflect this.

An observation is that if a leaf node has triangles that do not overlap, intra-node visibility is not necessary and hence the Z-buffer algorithm is also not necessary. However, to simplify the implementation, this is implemented only if a leaf node has a single triangle or two non-overlapping triangles.

5.7 Final Image Generation

The process so far details the method to find the corresponding triangle for each pixel on a particular row. Once the triangle has been determined for the pixel, shading can be computed using one of the popular methods. Since visibility is the main concern, the shading used by *Row Tracing* is a simplified method that aims to emphasise speed.

5.7.1 Simplified Shading for Row Tracing

The simplified shading used for *Row Tracing* is an adaptation of the method used in [MJC00] [MJC02] that uses normal coding [Gla90]. Directions are quantised and the normals of the triangles are then classified into one of the quantised directions for which shading is calculated.

The advantage of this process is that for a viewpoint, there are a fixed number of quantised directions for which shading can be pre-computed and stored in an array prior to tree traversal. For *Row Tracing*, a quantised set of 65535 directions are used. Once the tree traversal is completed, a pixel's shading is found by just an array indexing operation.

For every pixel in each row, if a triangle projects onto it, shading is computed and the pixel's colour is set accordingly. When all the rows in the image are similarly shaded after completing the tree traversal and leaf node processing, all the pixels in the image are appropriately shaded to obtain the final image.

The algorithm can be used without using HOMs, in which case visibility is always determined by the Z-buffer algorithm. However, the use of HOMs is an important optimisation that accelerates the rendering process significantly by incorporating an occlusion detection method – one that mimics the early ray termination used by ray tracers – into *Row Tracing*.

5.8 1D Hierarchical Occlusion Maps (HOMs)

Hierarchical Occlusion Maps (HOMs), as introduced by Zhang et al. [ZMHH97], is a very efficient method to determine occluded areas of an image. When combined with a front-to-back traversal of a structure like a kd-tree, the areas that have already been rendered occlude other triangles projecting onto these pixels. HOMs were also used in *Coherent Rendering* discussed in chapter 4. They are adapted for use with *Row Tracing* – the 1D nature making HOMs simpler and more efficient.

Through the use of 1D HOMs, it is possible to mimic the early ray termination feature of ray tracing whereby once a ray is determined to have hit an object, it can stop the traversal. However, due to the fact that a row encapsulates a large number of rays, the early ray termination property cannot be used directly. HOMs are one method to adapt this property of ray tracing to *Row Tracing*.

The HOM is an array of bits with each bit indicating if the part of the row corresponding to that bit is occluded. The HOM is a hierarchical structure. Each bit at the lowest level of the HOM corresponds to a pixel in the row being traced. Figure 5.10 shows a HOM in its initial state when the entire row is unoccluded. In the figure, the row is indicated by the coloured line with each red or green part representing a pixel. A bit at each higher level combines the occlusion status from the two corresponding bits at the level immediately below it – i.e., if the two corresponding bits are set, then the upper level bit is also set, otherwise the bit is not set. Essentially, the upper level bit is a *bitwise and* of the two corresponding lower bits. The bits below are combined to obtain the higher level bits and when the highest level bit is set, it indicates that the entire row is occluded.

To represent a HOM with minimal wastage, an array of `chars` is used in the implementation as the structure for 1D HOMs. As a `char` is just eight bits, this reduces any wastage that may occur by using a bigger data type. For an image that is 1024 pixels wide, the lowest level of the HOM uses just 128 `chars`. The upper levels use 64, 32...1 `chars` allowing the HOM for the row to be represented by just 256 `chars` – a negligible amount of memory. The reduced memory requirement for the HOM also improves the cache coherence.

Prior to traversal of a row, the entire row is empty, i.e., none of its pixels are rendered. At this point, the HOMs – that at all points indicate the occlusion state of the row – is initialised so that all bits are zero, as Figure 5.10 shows.

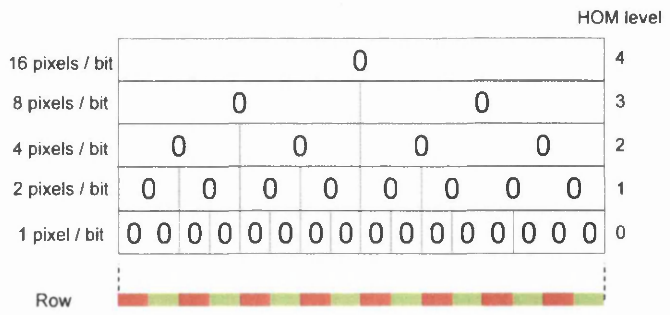
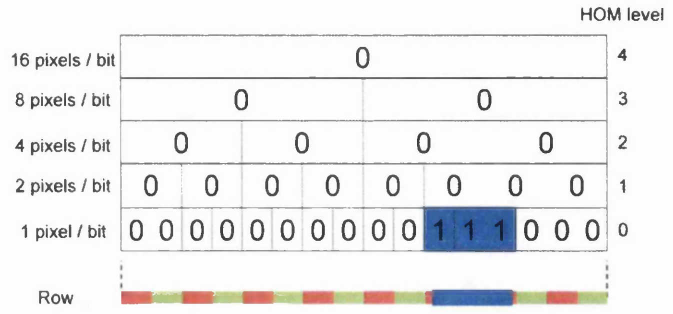
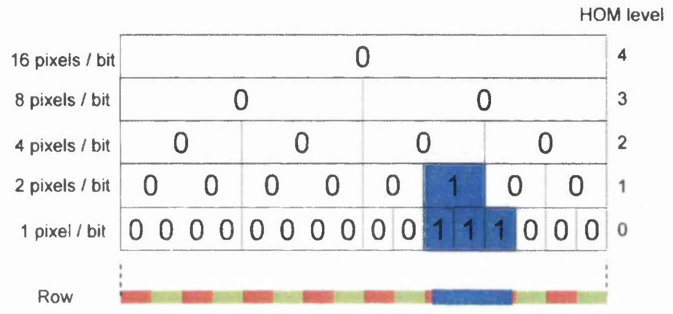


Figure 5.10: Initial state of the HOM.

The use of HOMs involve – updating it when a triangle is rasterised so that it represents the current occlusion state of the row at all times, testing it during tree traversal time. The HOMs are also



(a) Pixels and HOM bits updated (in blue). If the blue part on the row is rasterised, then the first level HOM is first updated.



(b) Upper level bit updated (in blue)

Figure 5.11: Updating the HOM.

checked when triangle segments are rasterised.

5.8.1 HOM Update

The HOM needs to be updated when triangles in the leaf node are rasterised so that the HOMs indicate the current occlusion state of the parts of the row. At the leaf node, when a triangle is determined overlap a part of the row, the bits corresponding to the affected pixels of the row are set to one. These bits indicate that any further attempts to render another triangle onto these pixels should fail. Each bit at the lowest level and its binary adjacent bit are *and*ed to obtain the upper level bit. A similar operation is undertaken at every consecutive level until the upper level bit is zero or until the highest level bit is reached. Figure 5.11 shows the process.

Each HOM consists of a series of `char`s making it preferable to set and update the HOM upwards in groups of 8 bits or 1 `char` at a time. Accordingly, during the rendering part (Section 5.6.5), the update is performed when the pixel width is eight pixels corresponding to the 8 bits in a `char` of the HOM at the lowest level. Hence, the state of these 8 pixels is combined into a `char` in which a bit with a value of 1 indicates that the corresponding pixel has been rendered and a bit with a value of 0 indicates that it has not yet been rendered. The `char` thus created is *bitwise and*ed with the corresponding HOM `char`'s current value to get the new occlusion status for these pixels.

Finding the upper level bits is slightly complicated by the lack of simple horizontal *bitwise and* methods. The most obvious method is to make a copy of the `char`, use bitwise shifts on the copy, and finally *bitwise and* the original `char` and the shifted copy as shown in Figure 5.12. This process is expensive – especially as this occurs during the rendering process. To optimise this, for each of the 256 values that the `char` can take, the horizontally *bitwise and*ed values are pre-computed and stored. This simplifies the expensive combination operation to one of indexing an array – a relatively less expensive operation.

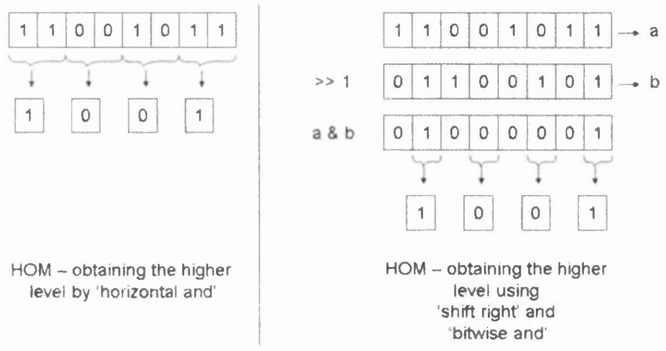

1 1 0 0 1 0 1 1
1 0 0 1
HOM – obtaining the higher level by 'horizontal and'
1 1 0 0 1 0 1 1 → a
>> 1
0 1 1 0 0 1 0 1 → b
a & b
0 1 0 0 0 0 0 1
1 0 0 1
HOM – obtaining the higher level using 'shift right' and 'bitwise and'

Figure 5.12: HOM update - combining bits horizontally.

The HOM thus updated will always reflect the current occlusion state of the row being considered. It is used to determine if subsequent rendering attempts correspond to already rendered parts of the row.

5.8.2 Occlusion Testing using HOMs

The real value of HOMs is realised at traversal time, when they enable detection of occluded parts of the row to allow large parts of the tree to be skipped. Given a segment of pixels – a starting pixel and an ending pixel – the occlusion testing process uses the HOMs to determine whether this segment is occluded or not. To conduct the test, the exact level and bits of the HOM that hold this information is needed. If the bits thus determined are all set, then the span of pixels are occluded. Otherwise, the span of pixels are not occluded and processing would have to continue further. The process is shown in Figure 5.13

The most challenging part of the occlusion test is to determine the necessary level and bits of the HOM.

Determining the level – The level of the HOM to be checked depends on the length of the pixel span to be tested. The level is obtained by taking the log_2 of this length and rounding up to the next integer. The log_2 of a floating point number is easily found – it is the exponent part in the IEEE floating point representation. Since the exponent provides the log_2 of a power of 2 lower than the `float` in consideration, it is incremented to get a power of 2 greater than the `float`. For pixel lengths that are exact powers of two the exponent is accurate. A HOM bit at this level indicates the occlusion status of 2^{level} pixels. It is possible that the pixel extent may span either one or two bits at this level.

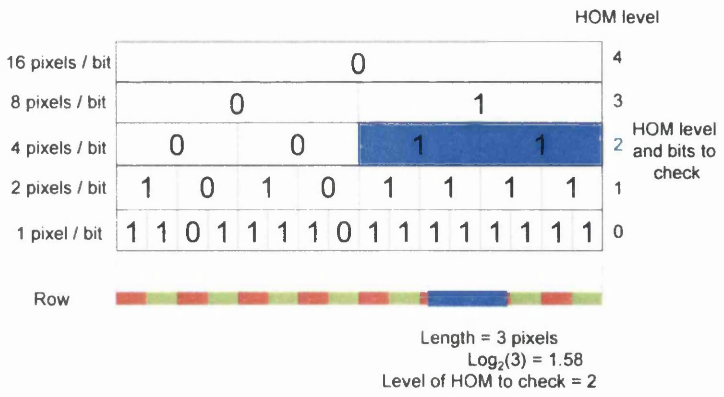


Figure 5.13: Occlusion testing. If the blue part on the row is to be tested for occlusion, 2 bits (shown in blue) at level 2 of the HOM is tested.

Determining the bits of the HOM at the level – At the level of the HOM, determining the corresponding bits of the HOM is done using the actual pixel values of the *start* and *end* pixels of the pixel extent. The respective pixel values are divided by 2^{level} – achieved easily by shifting the pixel value by *level* bits to the right – to get the bit(s) to be tested. This may be one or two bits depending on the start and end pixels.

Using the level and bits thus computed, occlusion is easily determined by testing if the bits are set. If they are set, the pixel extent is occluded and hence processing can stop.

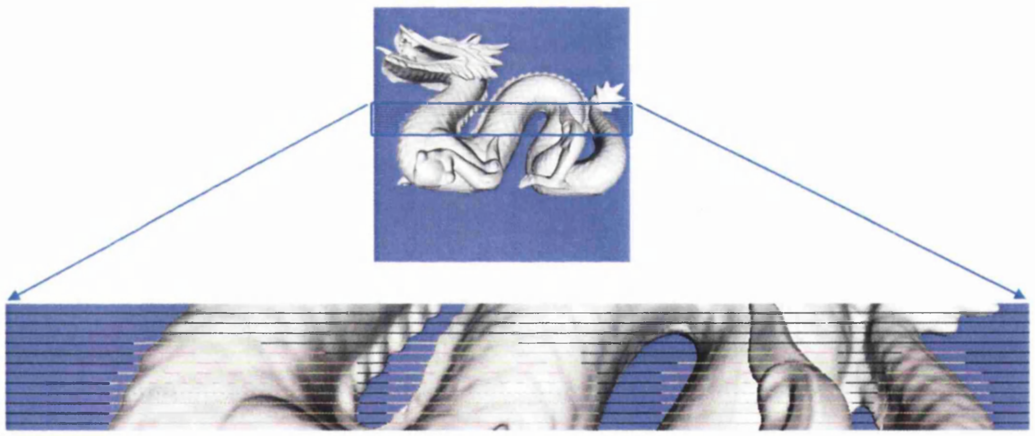


Figure 5.14: Final HOM on the *Dragon* model. Orange line represents the actual pixel. Each horizontal line above the orange line is a level of the HOM. In the HOM, each single coloured block indicates a bit of the HOM at that level.

The occlusion testing is used during the tree traversal to test and skip occluded nodes. It is also used in the rasterisation steps to determine accurate visibility of triangles and to skip pixel extents that maybe occluded.

5.8.3 Node Occlusion Testing

Checking the HOM is the simpler part of the node occlusion test. The more complex part is to determine an efficient method to find the pixels that the node occupy when projected onto the row. As detailed in Sections 5.5.5 and 5.5.6 and Figure 5.3, finding an overestimate is preferred to finding the exact projection due to the computational cost. Although, it would result in a few false negatives – i.e., a few nodes may be determined as unoccluded when in fact they are occluded, using the overestimate is preferred to the more expensive method of finding the exact node projection.

5.8.4 Recursive Rasterisation of Leaf Node Triangles

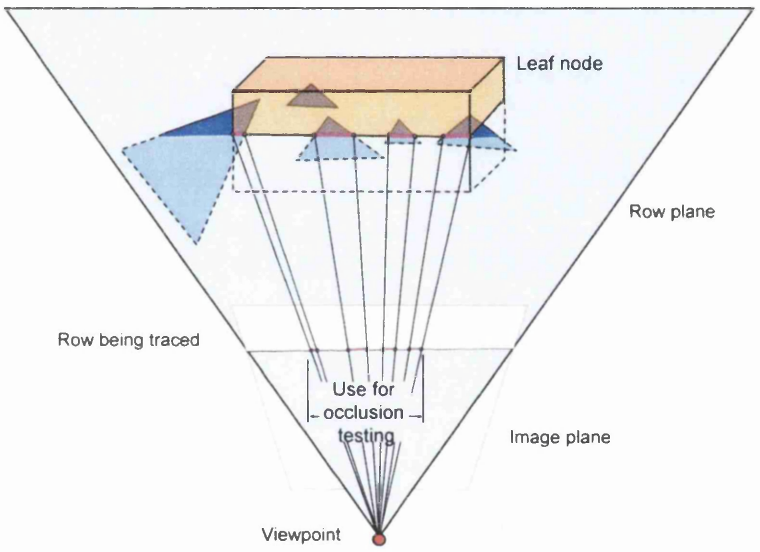


Figure 5.15: Leaf node intersection segments – Clipped and projected onto the row. These projections on the image row are tested for occlusion against the HOMs to determine inter-node visibility.

In Section 5.6.5, it was stated that a recursive process is followed to rasterise the triangle segments. HOMs are used even in this recursive process to efficiently determine inter-node visibility. The pixel extent to be tested for occlusion is given by the minimum and maximum projected points along the row as shown in Figure 5.15. If this extent is occluded, as determined by checking the HOM, then rasterisation can stop. However, if the extent is not occluded, it is subdivided into two sub-extents – each spanning half of the original extent – that are tested for occlusion. The process is continued until either the sub-extent is occluded or until the sub-extent is indicated by one bit in the HOM at a level where each bit indicates 8 pixels – i.e., a bit in level 3 of the HOM shown in Figure 5.13. The process is explained by the pseudocode below.

```
RecursiveRender(extent, pList)
{
  if(extent is occluded)
    return;
  if(extent is contained
      inside a block of 8 pixels)
  {
    Rasterise relevant parts of
      each projection in pList
      using a Z-buffer algorithm;
    Update Occlusion map
      with rasterised pixels;
  }
  else
  {
    RecursiveRender(first half of extent);
    RecursiveRender(second half of extent);
  }
}
```

Listing 5.5: Recursively render parts of the triangle that project onto this row. The method divides the part into two until the parts span eight pixels. At this point a Z-buffer algorithm is used to determine accurate visibility for these eight pixels.

If the extent – fully indicated by a bit in the level 3 of the HOM (Figure 5.15) – is not occluded, then the Z-buffer like structure is used to determine occlusion. However, in order to test whether a pixel has already been rasterised by triangles in other nodes, the corresponding lowest level bits (indicating occlusion status for a single pixel) of the HOM is tested. The pixel is rasterised only if the HOMs indicate that it is unoccluded.

In this manner HOMs are used both to optimise *Row Tracing* and also to ensure accurate visibility. It is an important part of *Row Tracing* that enhances the performance – especially for highly occluded scenes – to be very competitive with packet ray tracers and rasterisation methods like OpenGL.

5.9 Packet Row Tracing

Row Tracing makes effective use of the coherence provided by the underlying datastructure to accelerate the speed of rendering. However, upon closer observation, it is noticed that the algorithm can further utilise the coherence provided by the data structure. Several adjacent rows traverse a similar path down the tree and possibly hit the same object. This in turn points to the possibility of increased rendering performance by tracing groups or packets of rows through the tree. A new algorithm – *Packet Row Tracing* – was implemented to fulfil this promise.

5.9.1 High Level Algorithm

The high level algorithm is very similar to *Row Tracing*, as the pseudocode below shows. However, due to the use of groups / packets of rows instead of a single row, the algorithm needs minor modifications. The first change is that each component row has its own attributes that have to be computed prior to traversal of the packet. However, the major change is the tree traversal of the packet. Although it is very similar to the single row variant, it has to be adapted to handle packets of rows instead of a single row. It will be discussed in greater detail in Section 5.9.2. Finally, once the tree traversal completes, the triangles occupying all the pixels of all the component rows have been determined. This high level algorithm is given by the pseudocode below.

```
RowPacketTrace()
{
  for(each row packet in image)
  {
    for(each row r in row packet)
      InitialiseRowConstants();
    TraverseTreePacket(rootNode);
    for(each row r in row packet)
      Shade r;
  }
}
```

Listing 5.6: High Level *Packet Row Tracing* algorithm. Instead of initialising a single row and tracing it (as shown in Listing 5.1), rows are processed in groups. i.e., they are initialised, traversed and finally their pixels are shaded in groups. The algorithm is very similar to the single row version and major deviations from Listing 5.1 are highlighed in blue.

The initialisation and shading are just iterating over all the rows using the same methods described in the single row versions. The tree traversal, though, attempts traversing several rows down the tree minimising the number of node traversals necessary – reducing the calculations and improving the cache coherence and memory bandwidth usage of the algorithm.

5.9.2 Tree Traversal

The tree traversal is adapted so that the tree is traversed by a group of adjacent rows instead of a single row. The pseudocode below shows the process as implemented.

```
TraverseTreePacket(node)
{
  if(node is empty or
      entire RowPacket misses node)
      return
  proj = FindNodeProjection();
  if(proj is occluded in all rows or
      proj is outside frustum)
    return;
  if(node is leaf node)
  {
    for(each row in packet)
      ProcessLeafNode(row, node);
    return;
  }
  if(entire RowPacket intersects node)
  {
    for(each childNode sorted in
        front-to-back order)
      TraverseTreePacket(childNode);
  }
  else if(RowPacket partially
          intersects node)
  {
    for(each row in packet)
      TraverseTree(row, node);
  }
}
```

Listing 5.7: Row-packet–tree traversal algorithm. This is very similar to the single row traversal shown in Listing 5.3 and the major differences are highlighted in blue. The differences enable the algorithm to process a packet of rows instead of a single row.

The early termination criteria for the tree traversal are very similar to the ones for the single row variant. The first condition is if the node is empty, then traversal can stop. This condition does not vary for the packet version.

The next condition is packet–node intersection. As shown in Figure 5.16, the intersection is determined by using just the two outer boundary rows – the top and bottom rows of the packet. The signed distances between the node and these two are computed, similar to the single row version – described in Sections 5.5.7 and 5.5.8.

It may be recalled from Section 5.5.1 that a plane–node intersection can be achieved efficiently by predetermining the two vertices of the node casting the maximum projection onto the row-plane's normal. If $\mathbf{v_1}$ and $\mathbf{v_2}$ are the two vertices thus determined then we can define:
d_{top} and d_{top} – as the signed distances of the top row to $\mathbf{v_1}$ and $\mathbf{v_2}$ respectively and
d_{bottom} and d_{bottom} – as the signed distances of the bottom row to $\mathbf{v_1}$ and $\mathbf{v_2}$ respectively.
Using these signed distances, the cases are determined and handled as follows:

Case 1 – Entire packet intersects Node – The case occurs when:

$$\begin{aligned} sign(d_{top}) \ &!= \ sign(d_{top}) \ and \\ sign(d_{bottom}) \ &!= \ sign(d_{bottom}) \end{aligned}$$

This indicates that both the boundary rows intersect the node, as shown in Figure 5.16(a). Consequently, all the component rows of the packet also intersect the node, thus necessitating traversal down the tree.

Case 2 – Entire packet misses Node – Both the rows miss the node and are on the same side of the node, as shown in Figure 5.16(b). The case is determined when:

$$\begin{aligned} sign(d_{top}) \ &== \ sign(d_{top}) \ and \\ sign(d_{bottom}) \ &== \ sign(d_{bottom}) \ and \\ sign(d_{top}) \ &== \ sign(d_{bottom}) \end{aligned}$$

During the calculation of the distances, it is to be ensured that the normals to the rows are not directed towards each other. In this case, none of the rows in the packet intersect the node and hence the traversal can stop at this node.

Case 3 – Part of packet intersects Node – This can occur when:

- One of the boundary rows intersect the node and the other does not. This is detected if:

$$\begin{aligned} sign(d_{top}\ ) &\ != & sign(d_{top}\ )\ and \\ sign(d_{bottom}\ ) &\ == & sign(d_{bottom}\ ) \end{aligned}$$

or

$$\begin{aligned} sign(d_{top}\ ) &\ == & sign(d_{top}\ )\ and \\ sign(d_{bottom}\ ) &\ != & sign(d_{bottom}\ ) \end{aligned}$$

This case is illustrated by Figure 5.16(c).

- The node span is smaller than the span of the packet, as shown in Figure 5.16(d). This is detected when:

$$\begin{aligned} sign(d_{top}\ ) &\ = & sign(d_{top}\ )\ and \\ sign(d_{bottom}\ ) &\ = & sign(d_{bottom}\ )\ and \\ sign(d_{top}\ ) &\ != & sign(d_{bottom}\ ) \end{aligned}$$

When only a part of the packet intersects the node (Figure 5.16(c) or Figure 5.16(d)), the algorithm continues traversal using the single row variant with each of the component rows using this node as the starting node for single row traversal.

Node Occlusion testing – When a group of rows are being traversed, if the coherence is good enough, the occlusion of one of the rows may indicate that the entire group is occluded. However, to test this accurately, the method used is to simply test the occlusion maps of all the component rows individually.

Since the transformation matrix is used to project the node onto the screen, the node's maximum extent – given by the minimum and maximum X coordinate occupied by it on the screen – applies to all the rows. It is sufficient to find the node projection's overestimate as detailed in Sections 5.5.5 and 5.5.6 just once for the entire packet of rows to get the pixel extent for the occlusion test.

It may be recalled that during the initialisation process, constants for each row are being created and maintained. During this process, HOMs for each of the component rows are also created and initialised. These HOMs are updated when triangles are rasterised and kept up to date with the occlusion status of the row. During tree traversal, upon finding the node projection overestimate, it is tested against every single row's HOM. If any one of the HOMs indicates that the node is not occluded, the occlusion test can stop as the traversal needs to continue in this case. However, if all the HOMs indicate that the node is occluded, tree traversal can stop for the entire packet.

As with the single row version, if all the tests indicate that traversal should continue, traversal continues until either the node is occluded or a leaf node is reached.

Leaf node processing – If traversal continues down the tree in packet mode until a leaf node is reached, then all the component rows intersect the leaf node and can have some of their pixels

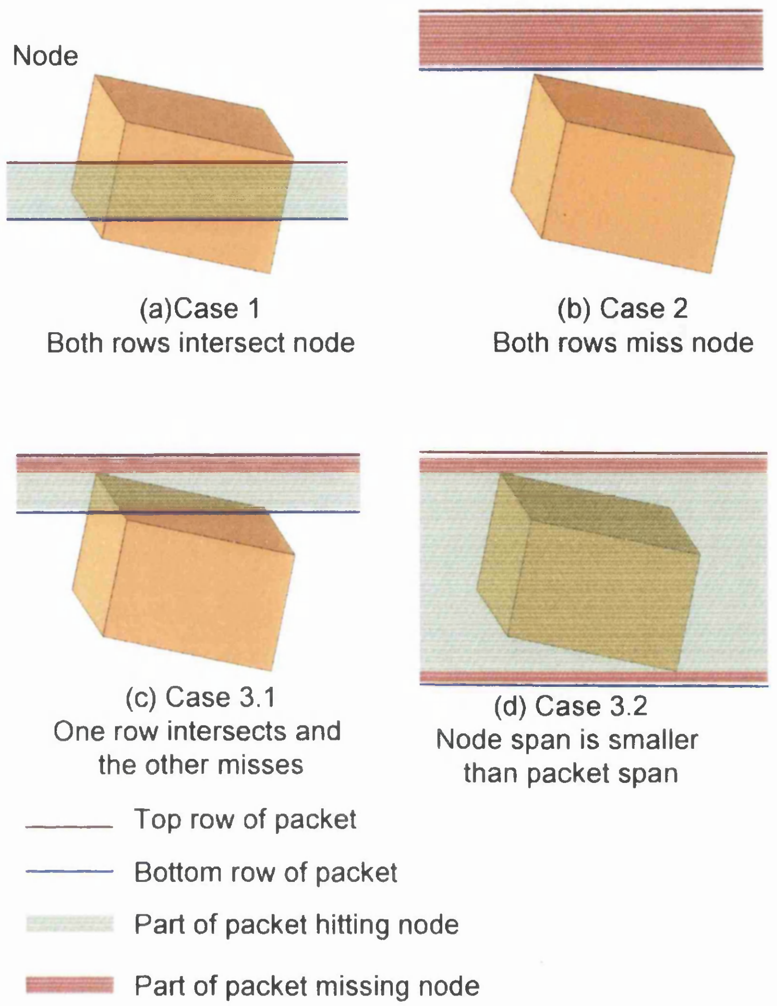


Figure 5.16: Row-Packet–Node intersection.

determined at this leaf node. In this implementation of *Packet Row Tracing*, all the component rows at the leaf nodes are treated individually and are separately processed using the same method used by the single row variant (section 5.6).

Final image generation – At the end of the tree traversal by a packet of rows, the triangles at every pixel of each of the component rows is determined. These triangles are used to shade the pixels corresponding to each pixel of each row. By tracing all the rows of the image, by grouping them into packets, the final image is generated and displayed.

An important consideration for *Packet Row Tracing* is the number of rows in each packet. The performance of the algorithm is dependent on this number. A small number does not maximise the coherence enough, whereas with a very large number the available coherence is not sufficient. In our implementations, packet sizes of 8, 16 or 32 were found to be very good with a packet size of 16 providing the best results on most scenes.

5.10 Low Level Optimisations

The algorithms – *Row Tracing* and *Packet Row Tracing* – as implemented without any low level optimisations are reasonably fast. However, in order to maximise the performance, a few implementation optimisations are used. The two main optimisations are the use of SIMD instructions (see Appendix B) and multi-threading.

5.10.1 Multi-Threading

Row Tracing is highly parallelisable as each row is independent of other rows. Thus, processing of each row can occur simultaneously without interfering with the processing of other rows. This property of *Row Tracing* is inherited from ray tracing that is similarly trivially parallelisable.

Multi-threading over the number of cores available accelerates *Row Tracing* and the packet variant immensely. As the results section – Section 5.11 – will show, the speed-ups achieved is almost perfect with a speed-up of 3.8× on average (for the fastest variant of *Row Tracing*) on a quad core processor.

The key to achieve the best performance when an algorithm is multi-threaded over multiple cores is to ensure that all the threads have as similar a workload as possible. This ensures that all the threads finish the work allocated almost simultaneously. To achieve the best distribution of work across threads, a round robin allocation of the rows / packets of rows to process is implemented. As Figure 5.17 shows, this means that the first row / first packet of rows is allocated to the first thread, the second row / packet to the second thread and so on. As adjacent rows follow a very similar path down the tree, this method of load balancing ensures that the threads have workloads as close to each other as possible.

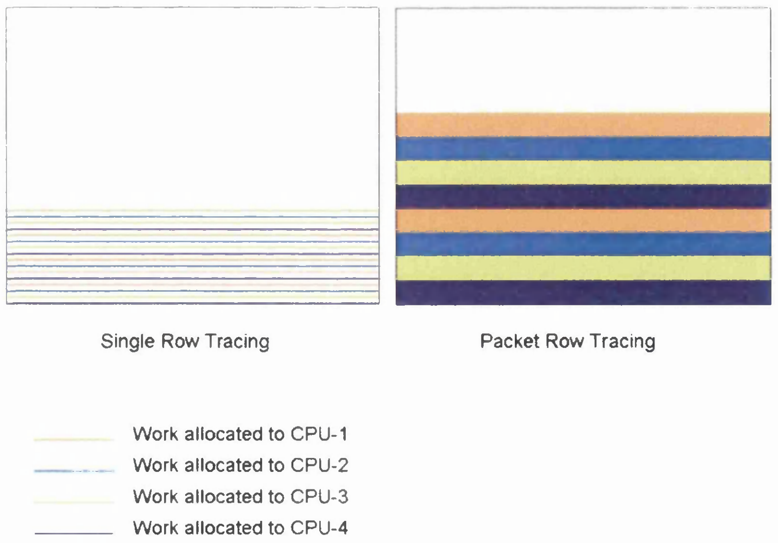


Figure 5.17: Load balancing of *Row Tracing* and *Packet Row Tracing*. Each coloured block is the work allocated to a thread.

5.11 Results

The algorithm has been implemented in C++ using the Visual C++ IDE. The results are tabulated by running the algorithm on a computer with the Intel Core 2 Quad 2.4 GHz processor, 4 GB of RAM and a Geforce 8800 GTX with 768 MB of video memory, running Windows XP64. A range of models are used to study the algorithm.

Figure 5.18 shows the performance of *Row Tracing* and its packet variant in comparison to the performance of packet ray tracing and OpenGL on the same scenes. It can be noticed that the performances of *Row Tracing* and *Packet Row Tracing* are excellent. It is also noticeable that *Row Tracing* and *Packet Row Tracing* are extremely amenable to multi-threading. The other point of interest from the graph is the adaptability of *Row Tracing* over different data structures like kd-trees and octrees.

In order to observe these results better, subsets of Figure 5.18, that isolate particular results are shown in the sections that follow.

5.11.1 Row Tracing vs Packet Ray Tracing

In order to better understand the performance difference between *Row Tracing* and packet ray tracing, the **best results** for these two rendering methods are isolated in Figure 5.19 and Table 5.2. The figure and the table show that the fastest version of *Row Tracing* is significantly faster than the fastest version of ray tracing. The advantage is particularly noticeable when the scene consists of large triangles like in *ERW6*, *Sponza* and *Sodahall* scene with advantages of 7.41×, 1.93× and 3.31×. This reveals the advantage of cheap triangle intersections amortised over a large number of pixels. For other scenes – the *Powerplant* and *Armadillo* scenes – that have a large number of (possibly small) triangles visible, *Row Tracing* shows advantages of 1.93× and 1.09×. It can thus be inferred that *Packet Row Tracing* is a very viable alternative to packet ray tracing as a visibility method.

Scene	ERW6	Sponza	Armadillo	Sodahall	Powerplant
Row Tracing (kd-tree)	128	40	20.6	49.2	6.67
Ray Tracing 8×8	17.3	20.6	14.2	14.8	6.09
OpenGL	1000	500	333	8.40	1.49
Row Tracing speed-up vs Packet Ray Tracing	7.41×	1.93×	1.45×	3.31×	1.09×
Row Tracing speed-up vs OpenGL	0.13×	0.08×	0.06×	5.86×	4.48×

Table 5.2: Best performances of packet ray tracing and *Row Tracing* in **frames per second**. 8×8 ray packets and *Packet Row Tracing*'s kd-tree versions have been used respectively.

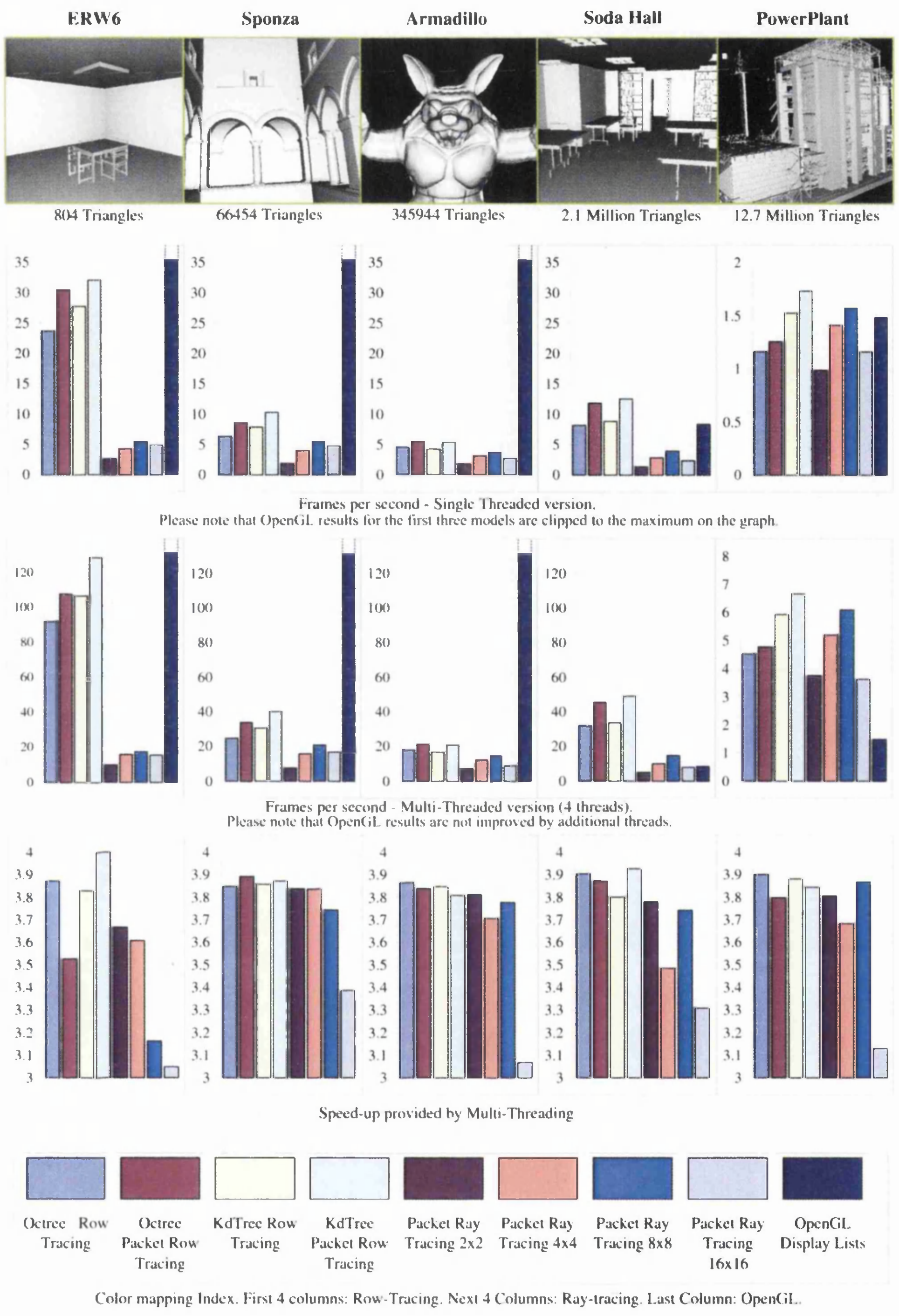


Figure 5.18: Performance of *Row Tracing* vs Packet Ray Tracing and OpenGL.

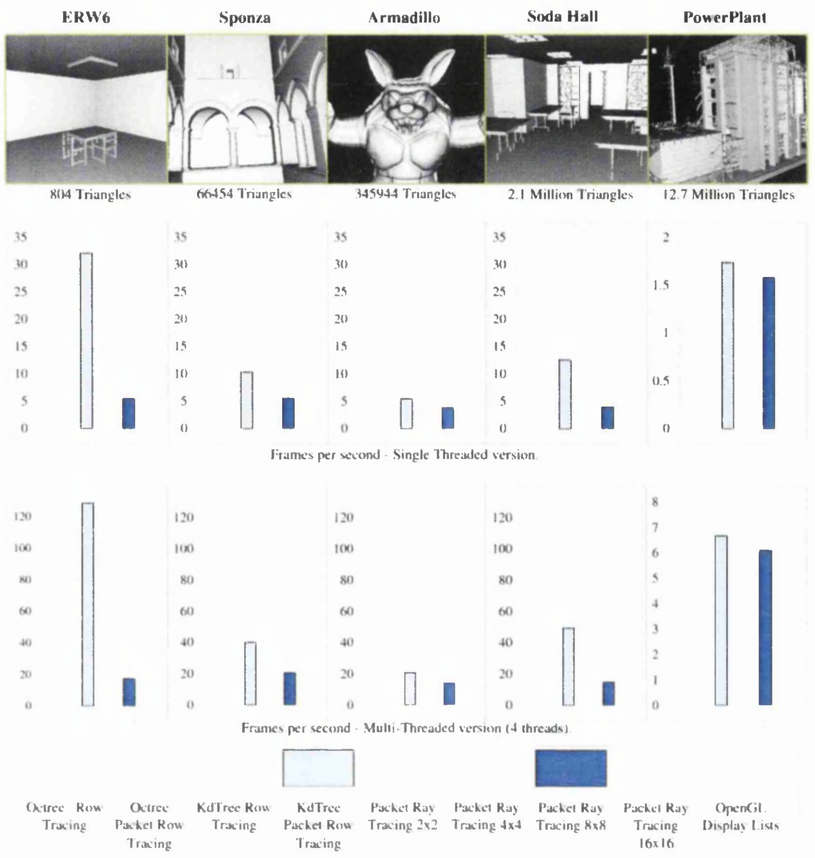


Figure 5.19: Performance of *Packet Row Tracing* vs Packet Ray Tracing – Single threaded and Multi-threaded versions respectively. For smaller sized models *Packet Row Tracing* is much faster than packet ray tracing. For larger models, though the performance difference is not that significant, *Packet Row Tracing* is still faster.

5.11.2 Row Tracing vs OpenGL

Since *Row Tracing* can be thought of as an algorithm that combines ray tracing and OpenGL, a comparison against OpenGL is necessary. Table 5.2 and Figure 5.20 compares the fastest *Row Tracing* variant vs OpenGL implemented with display lists. It is clear that for smaller models – with less than a million triangles – OpenGL is significantly faster than *Row Tracing*. However, with an increase in the scene sizes, OpenGL performance decreases dramatically as evidenced by the performance of OpenGL on the larger models – *Sodahall* (2.1 million triangles) and *Powerplant* (12.7 million triangles). For these scenes, *Row Tracing* is 5.86× and 4.48× faster respectively.

This shows that *Row Tracing* inherits the complexity advantage of ray tracing making it very scalable over scene sizes. It also indicates the dependence of OpenGL on available video memory. The display lists for the larger scenes probably do not fit within the 768 MB of video memory available. This also contributes to the performance reduction as parts of the model would have to be paged. *Row Tracing*, with its low memory usage can handle such scenes efficiently.

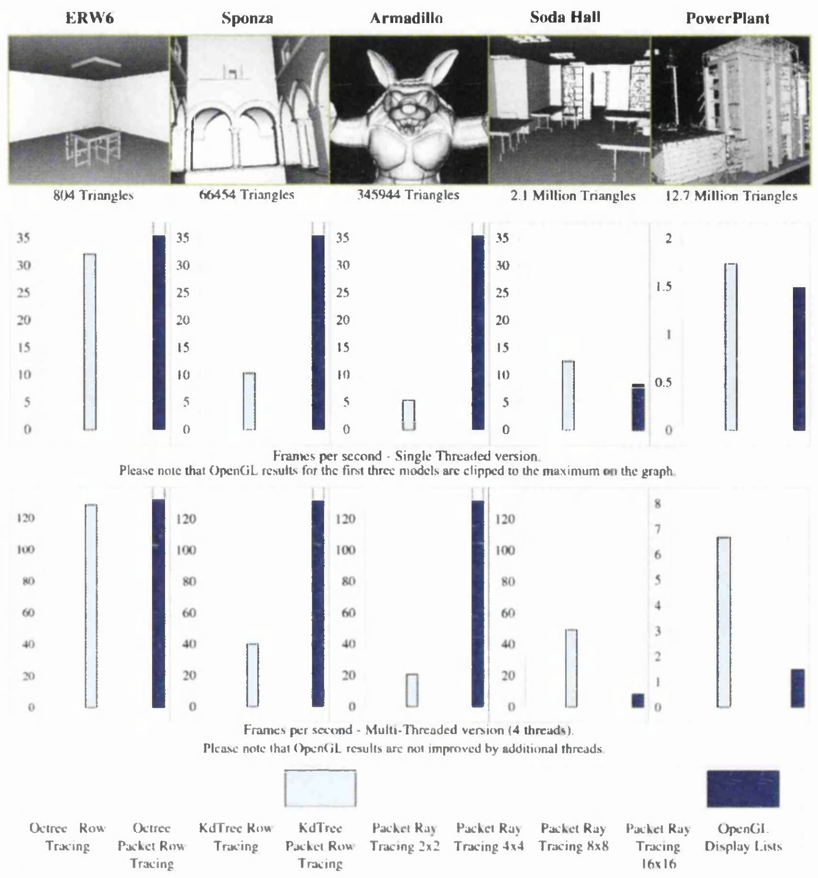


Figure 5.20: Performance of *Row Tracing* (Single threaded and Multi-threaded versions) vs OpenGL. OpenGL performance for first three models clipped to 35 fps and 120 fps in both charts. For smaller models, OpenGL is much faster. However, for larger models, *Packet Row Tracing*, especially when multi-threaded, is faster than OpenGL.

5.11.3 Row Tracing Performance on Kd-trees vs Octrees

The performance of *Row Tracing* on octrees is very good – close to its performance on an SAH kd-tree. Figure 5.21 shows that the difference in performance is small for most scenes. Especially for densely packed scenes like the *Armadillo* scene, when the SAH cannot effectively segregate the triangles, the octree actually shows a slight advantage over the kd-tree. This is due to the computational simplicity of intersecting an octree node as against the slightly more expensive kd-tree node intersection.

This result is significant as octrees are a very simple structure that are very easy and fast to create. By using fast construction methods together with fast rendering methods, *Row Tracing* with the octree can be used as an effective algorithm to perform dynamic rendering. The octree results also show the adaptability of *Row Tracing* on structures that are made up of cuboids / Axis-Aligned Bounding Boxes (AABBs). When combined with recent data structures like BIH (Bounding Interval Hierarchies) [WK06], BVH (Bounding Volume Hierarchies) [WBS07], etc that have fast construction methods available, it points to the use of *Row Tracing* for dynamic rendering.

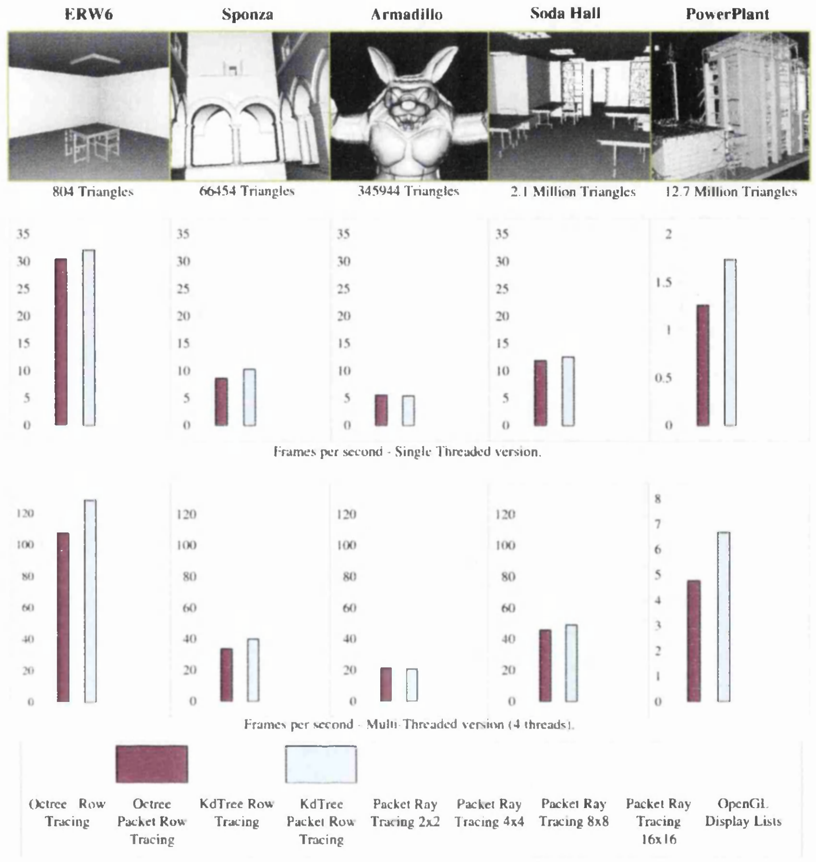


Figure 5.21: Performance of *Row Tracing* on kd-trees and octrees – Single threaded and Multi-threaded versions respectively. Shows that *Row tracing* works well on both kd-trees and octrees.

5.11.4 Performance of Row Tracing vs Packet Row Tracing

Figure 5.22 and Table 5.3 show that *Packet Row Tracing* is considerably faster than *Row Tracing*. The use of multiple row results in a 1.05× to 1.46× acceleration in rendering performance. While this is considerable, it is noticable that the acceleration due to the use of multiple rows is not in the same league as acceleration obtained by packet ray tracing vs single ray tracing. *Row Tracing* already makes uses of a lot of the coherence provided by the scene. The use of packets can thus only slightly improve coherence utilisation. In addition, the use of packets implies that some of the coherence is lost as the component rows may not traverse the same nodes, at which time *Packet Row Tracing* reverts to the single row version. Having said that, the use of packets does result in a considerable improvement in rendering time and makes *Packet Row Tracing* very useful.

Scene	ERW6	Sponza	Armadillo	Sodahall	Powerplant
Row Tracing (kd-tree)	106.38	30.48	16.42	33.78	5.93
Packet Row Tracing (kd-tree)	128.21	40.00	20.66	49.26	6.67
Packet Row Tracing speed-up vs *Row Tracing*	1.21×	1.31×	1.26×	1.46×	1.12×
Row Tracing (octtree)	91.74	24.63	17.79	32.05	4.54
Packet Row Tracing (oct-tree)	107.53	33.67	21.37	45.87	4.78
Packet Row Tracing speed-up vs *Row Tracing*	1.17×	1.20×	1.45×	1.43×	1.05×

Table 5.3: Performance comparison between *Row Tracing* and *Packet Row Tracing*.

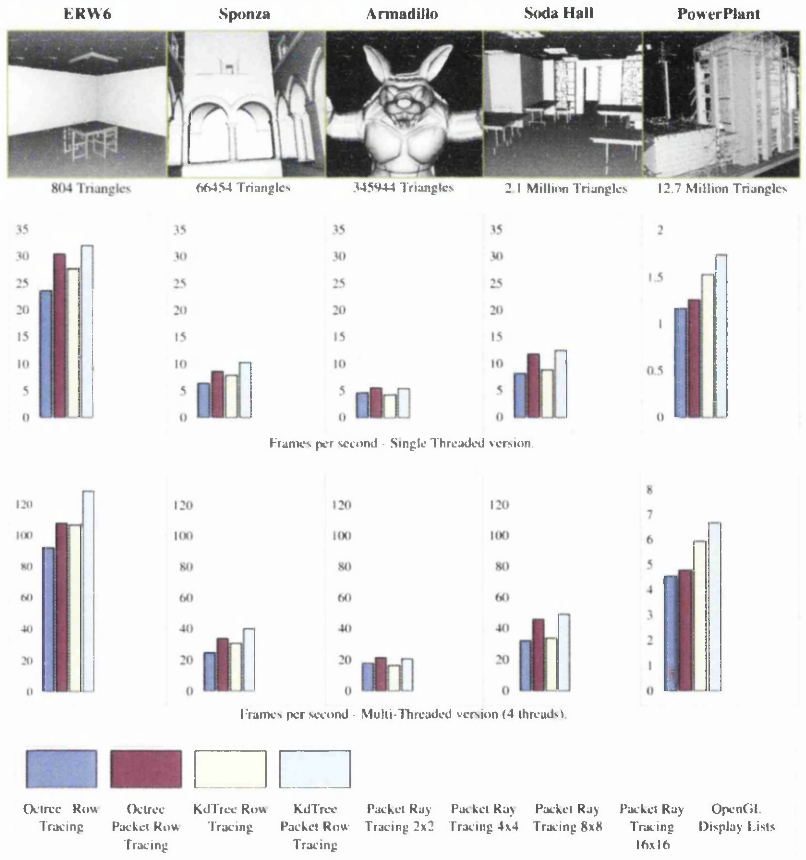


Figure 5.22: Performance comparison between *Row Tracing* and *Packet Row Tracing* on both octrees and kd-trees. It is clear that *Packet Row Tracing* is considerably faster than single *Row Tracing*.

5.11.5 Multi-core Performance

The performance of *Row Tracing*, in theory, should scale when the number of cores are increased. Figure 5.23 shows the speed-ups achieved by *Row Tracing* when multi-threaded and run on a quad core processor. It can be inferred that the algorithm is exceptionally suited to be multi-threaded. The speed-ups for the best variant of *Row Tracing* – *Packet Row Tracing* with kd-trees – is about 3.8×, thus, confirming the claims. In addition, it also shows the low memory bandwidth requirements of *Row Tracing*, since for a multi-threaded application, the major bottlenecks are load balancing and memory bandwidth issues.

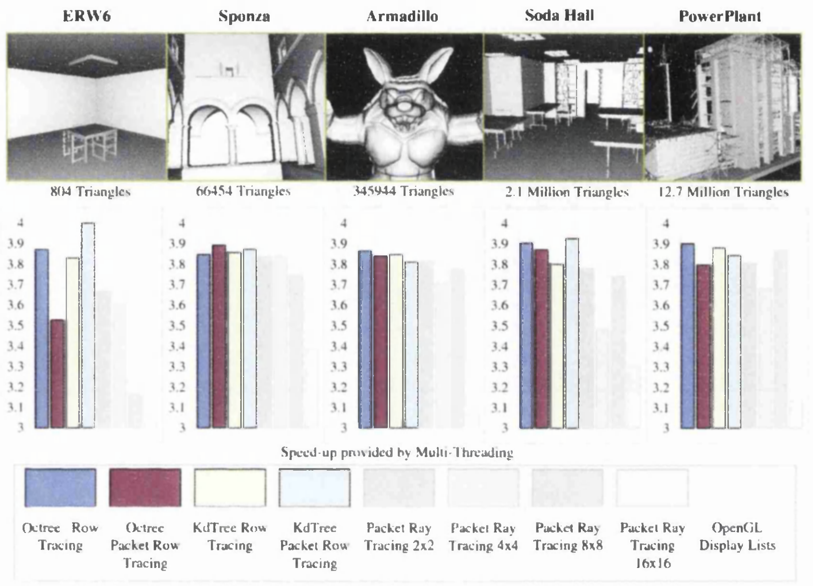
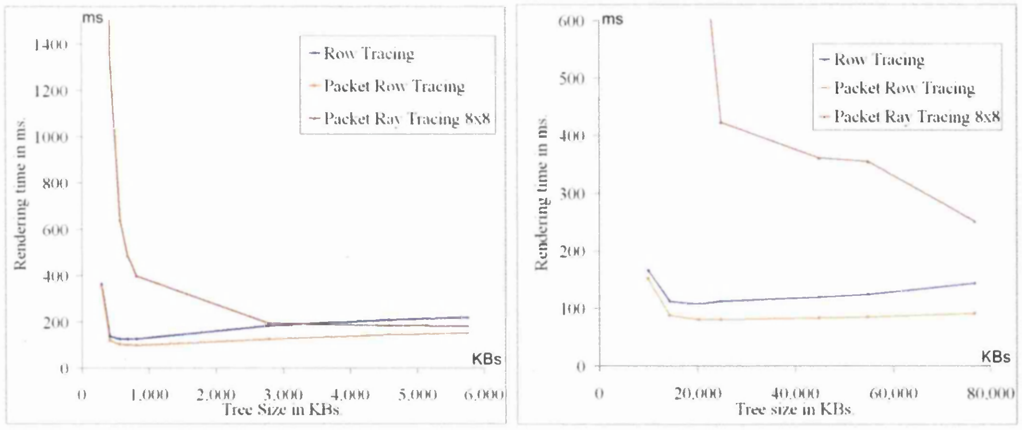


Figure 5.23: Speed-ups achieved by *Row Tracing* by using four threads a quad core CPU. Accelerations of close to 4× suggest that *Row Tracing* is highly parallelisable.

5.11.6 Performance vs Tree Size

Figure 5.24 shows the performance of *Row Tracing* and packet ray tracing across kd-trees of different sizes. The performance of *Row Tracing* is very good for small tree sizes. It improves significantly when the tree sizes are moderately increased. The optimal tree size for *Row Tracing* is very small and performance degrades slightly when tree sizes are increased beyond this. In comparison, the performance of packet ray tracing is poor when tree sizes are very small. It improves only when the tree size is increased significantly. The optimal tree size for packet ray tracing is significantly greater than that of *Row Tracing*.

This is another important result when considering *Row Tracing* for dynamic rendering. In a dynamic context, the structure has to be constructed prior to rendering each frame. A smaller tree size implies lower construction times leading to better dynamic rendering performance.



(a) Sponza scene (b) Sodahall scene

Figure 5.24: Rendering times using *Row Tracing*, *Packet Row Tracing* and Packet Ray Tracing over tree sizes show that while packet ray tracing needs larger trees, *Row Tracing* and *Packet Row Tracing* work very well with smaller trees pointing to the possibility of using *Row Tracing* for dynamic scenes.

5.11.7 Row Tracing with and without HOMs

As mentioned, *Row Tracing* can be used without HOMs. In this case, the visibility is accurately determined using a partial Z-buffer. To measure the effectiveness of HOMs as implemented for *Row Tracing*, the performance is determined with and without the use of HOMs. The results are shown in Figure 5.25.

The times indicate that HOMs are very effective. For the fastest version of *Row Tracing* – multi-threaded packet *Row Tracing* on SAH kd-trees – the HOMs are responsible speed-ups of 1.2×, 1.5×, 2.3×, 21.7× and 36× for the five scenes – *ERW6*, *Sponza*, *Armadillo*, *Sodahall*, and *Powerplant* scenes – respectively. The benefit of HOMs is seen more for larger scenes like the *Sodahall* and *Powerplant* scenes. Only for the very small *ERW6* scene, the HOM's effect is negligible. This shows that the HOMs, and in a similar vein – the early ray termination used by ray tracers is highly effective in improving the rendering performance. It is also an indication that HOMs are a highly effective method to transfer the early ray termination property to *Row Tracing*.

5.12 Summary

The results show the effectiveness of the algorithm as a visibility method. For scenes with predominantly large triangles visible, *Row Tracing* shows exceptional performance in comparison to packet ray tracing. This is attributed to the amortisation of intersection calculations over a large number of pixels. The traversal calculations are also averaged over a number of pixels, but the smaller tree sizes – implying fewer traversals – work in favour of *Row Tracing* making it much more efficient. Finally, the adaptation of the early ray termination property – used extensively in

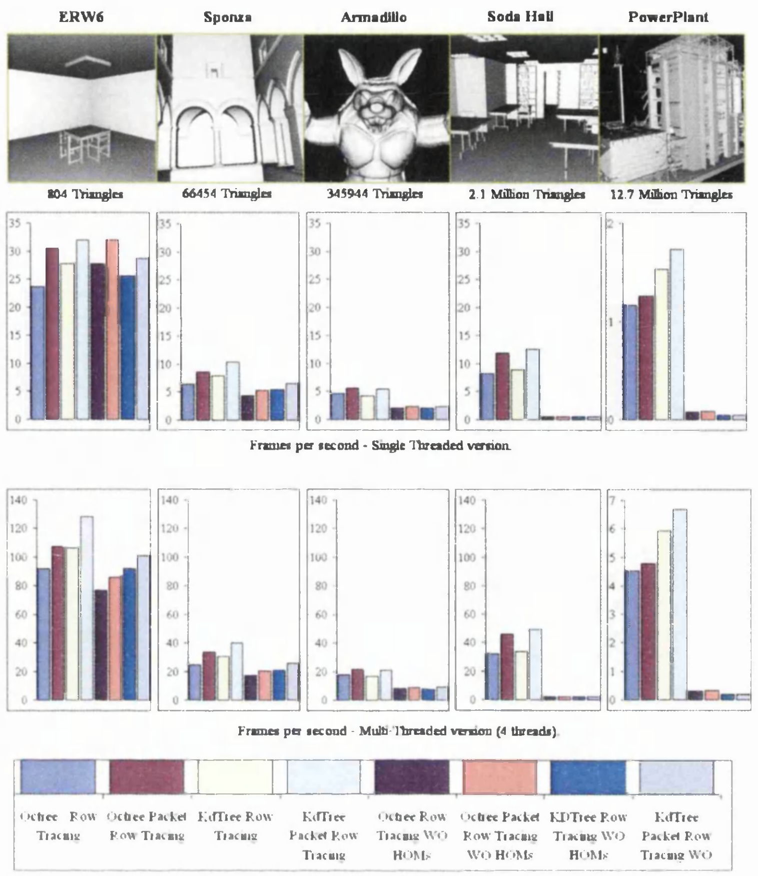


Figure 5.25: Performance of *Row Tracing* on kd-trees and octrees with and without HOMs. HOMs are shown to be highly effective as an occlusion detection method.

ray tracers – to *Row Tracing* through a 1D version of HOMs proves extremely beneficial, leading to a very efficient algorithm.

In comparison to OpenGL, *Row Tracing* shows an advantage when the sizes of the scenes are large. As the scene sizes increase, OpenGL – using a brute force approach – shows deteriorated performance, with ray tracing and *Row Tracing* outperforming it for these scenes. At the same time, the performance of *Row Tracing* for smaller scenes is very good. It can thus be inferred that *Row Tracing* has a better complexity over scene sizes than basic OpenGL with display lists. The results thus show that *Row Tracing* scales very effectively both ways according to the scene size.

Another inference from the results is the potential suitability of *Row Tracing* for dynamic rendering. *Row Tracing* works very well with a very simple data structure like the octree. Octrees – being simpler to create – are better suited for a dynamic rendering context. In addition, *Row Tracing* works well when tree sizes are small. Again, smaller trees are faster to create – pointing to very good potential for dynamic rendering. The utilisation of *Row Tracing* in a dynamic rendering context is one of the obvious areas for further investigation.

Row Tracing is also very easily parallelised – a property that is inherited from ray tracers. This is a very valuable advantage when there is a trend towards increased number of CPU cores. The results show an almost perfect speed-up with the number of cores. Thus, when the number of cores increase, *Row Tracing* performance would increase without any modifications to the implementation.

Other future work include using techniques used in rasterisation - shadow, reflection and refraction mapping, shadow volumes, etc – to add secondary ray effects. *Row Tracing* inherits the inability of rasterisation based methods to physically model secondary rays and these have to be implemented with methods similar to rasterisation. There is also scope for further optimisations like testing entire packets for occlusion, faster leaf node processing for packets by intersecting groups of rows with the leaf node triangles, and possibly splitting packets into smaller packets at divergence nodes during traversal to further accelerate *Packet Row Tracing*.

It has been seen that the optimal trees for *Row Tracing* are much different than trees for ray tracing. For a cube, the probability of a plane intersecting it is proportional to the sum of the edge lengths [Hai07]. Instead of using the SAH, a modified version that uses the sum of edges measure could be used instead of the surface area to create a tree with better properties for *Row Tracing*.

Row Tracing is introduced as a novel algorithm that aims to improve visibility determination / rendering performance. It can be considered as a hybrid algorithm combining rasterisation and ray tracing concepts – fast scanline rasterisation at the leaf node level and fast front-to-back tree traversal of an octree or a kd-tree. The use of HOMs to determine accurate visibility transforms it into a very effective algorithm, especially in cases where there is significant occlusion – as in most computer graphics scenes. The fact that rows are traced independently of each other implies that it is highly parallelisable. In addition to these factors, *Row Tracing* and *Packet Row Tracing* are simple algorithms to implement. These factors enable *Row Tracing* to be a very viable alternative to either packet ray tracing or rasterisation based visibility determination / rendering methods.

Chapter 6

Conclusions and Future Work

Conclusions

This research has made several contributions while investigating the use of ray tracing data structures and algorithms in the context of visibility. The aim of the research was to investigate the effectiveness of ray tracing structures like kd-trees, octrees, etc and associated algorithms from a purely visibility context – i.e., when only the first intersected object is to be found without considering additional optical effects like reflections, refractions and indirect lighting. Three new methods to achieve this have been developed and studied.

- *RBSP trees* – A new structure based on kd-trees, but more general, *RBSP trees* provide ability to have more than three splitting axes. The axes may be in any arbitrary direction as long as they are predetermined prior to tree construction. They allow the tree to fit closely to the scene being rendered, thus reducing the number of node traversals and primitive intersections.

  Construction and use of *RBSP trees* to generate images has been discussed extensively in Chapter 3. Due to their similarity to kd-trees, several well known algorithms and heuristics for both construction and traversal can be adapted. Through the use of the Surface Area Heuristic to construct them, it has been possible to construct good trees for most scenes.

  Results show that, for scenes with predominantly non-axis-aligned triangles, *RBSP trees* are responsible for a reduction in the number of node traversals and triangle intersections. Unoptimised rendering times also show that *RBSP trees* are more efficient than kd-trees for these scenes. At the same time, *RBSP trees* are slower than kd-trees when the scene consists of predominantly axis-aligned triangles like the *Sponza* scene. In addition, another problem is the fact that construction times, as described in this thesis, are dependent on the number of axes used and can be quite long.

  During the axes selection phase of the construction, the axes can be drawn from the scene itself so that the *RBSP tree* provides an even closer fitting structure. By this, fewer axes would be sufficient. The results showed that when ray tracing on *RBSP trees*, for most scenes, the best results were obtained with 8-12 axes and the rendering times went up with trees constructed using more than 8-12 axes. In addition, for the *Sphere* scene, for which the axes were, in effect, customised, *RBSP trees* produced the best acceleration compared to kd-trees.

It is notable that the introduction of *RBSP trees* has drawn more attention to the use of structures with non-axis-aligned splitting axes. Budge et al. address some of the problems of *RBSP trees* as described in our publication. Budge et al.'s improvements make *RBSP trees* faster to construct and traverse. Ize et al. use the more general form of BSP trees and show that they are quite effective for ray tracing.

- *Coherent Rendering* – This is an algorithm aiming to demonstrate a reduced complexity for primary visibility using essentially ray tracing structures and concepts like the front-to-back traversal of a kd-tree. The method is an adaptation of the volume rendering algorithm by Mora et al. that was very efficient and showed that per pixel rendering complexities are constant.

  The algorithm uses several concepts from both rasterisation and ray tracing based visibility methods like *Hierarchical Occlusion Maps*, polygon subdivision and front-to-back traversal of kd-trees. By combining these, an algorithm that achieves better coherence is developed. It is shown that as image sizes increase, the time per pixel converges to a constant. This time per pixel is only about 30% greater for the 12+ million *Powerplant* scene than for the *Single Triangle* scene when large image sizes are used. This shows that complexity is definitely less than logarithmic as image sizes increase.

- *Row Tracing* and *Packet Row Tracing* – *Row Tracing* is an algorithm similar to scanline rendering in that it processes one row of the image at time. However, in contrast to scanline renderers, the intersections between the row and the objects are done in object space. In addition, ray tracing structures like kd-trees and octrees are used. An adaptation of *Hierarchical Occlusion Maps* are used to determine already occluded areas of the image. The combination of concepts from both rasterisation and ray tracing produces a very effective and parallelisable method of determining visibility.

  In addition, upon closer inspection, it is observed that neighbouring rows traverse a similar path down the tree. To maximise this coherence, groups of rows are traversed down the tree with a small change to the traversal algorithm. Tracing groups of 16 neighbouring rows through the tree is observed to produce the best results.

  Results show that *Row Tracing* and *Packet Row Tracing* are very effective algorithms. When scene sizes are small, *Row Tracing* and *Packet Row Tracing* are much faster than packet ray tracing. Comparing the algorithms with OpenGL (that uses a normal Z-buffer algorithm) shows that *Row Tracing* and *Packet Row Tracing* are much faster when scene sizes are large. For these scenes, packet ray tracing is also faster than OpenGL owing to its better complexity. However, *Packet Row Tracing* is faster than packet ray tracing for these scenes too.

The methods are new techniques, two of which have been published. However, as with any research, each method has given rise to new ideas that could further improve the techniques. These will be briefly described.

Future Work

For each technique, some areas of improvement and extensions are easily identifiable.

- *RBSP trees* – The flexibility of *RBSP trees* to have several axes that may be in arbitrary directions means that *RBSP trees* can adapt very well to the scene. It is believed that by using 8-12 axes customised to the scene, *RBSP trees* could be the best structure for ray tracing single rays.

Another method to customise the tree to the scene is to have a large number of axes – for eg. 100 directions – and at each step a subset of this used so that the construction process is simplified. The subset would be determined based on geometric properties of the geometry in the scene. This would alleviate the problem of slow construction times seen when the number of axes increases. The constructed tree may sufficiently reduce the number of traversals and intersections to overshadow the increased traversal cost.

Although in this thesis only primary rays are considered to determine primary visibility, it is believed that the real value of *RBSP trees* would be when several incoherent rays are to be traced through the scene like in global illumination algorithms. In these methods, the ability of *RBSP trees* to reduce the number of traversals and intersections would be truly valuable.

Although some effort has been put into optimising the traversal methods, the effort has not been significant. Budge et al. [BCNJ08] state that their optimised methods are $10\times$ faster than ours. This shows that there is considerable margin for optimisation.

- *Coherent Rendering* – The *Coherent Rendering* algorithm described is unoptimised and the main aim was to demonstrate that a lower complexity was possible. However, the absolute performance is not very competitive. To address this, the algorithm can be optimised through the use of SSE instructions and by a cycle of optimising and profiling. The performance after optimisation may reach competitive levels.

  In rasterisation, shadows are generated through the use of shadow maps. Similar techniques are used for refraction and reflection. The generation of additional images / maps could also be achieved with a lower complexity leading to a full fledged low complexity renderer.

- *Row Tracing* and *Packet Row Tracing* – As described in the future work for *Coherent Rendering*, shading, reflection and refraction could be simulated through shadow maps, etc.

  A major advantage of *Row Tracing* is that it works very well with simple structures. This was shown by the efficiency of *Row Tracing* on octrees. The difference in rendering times on an SAH kd-tree and an octree is not very significant. Simple structures like the octree are simpler and much faster to build leading to applications for rendering dynamic scenes. In addition, the fact that *Row Tracing* works well on kd-trees and octrees imply that it would work well with any structure consisting of axis-aligned bounding boxes. Hence, they can be used with structures that are fast to build like grids or BVHs for use in a dynamic context.

  The best results for *Row Tracing* have been obtained on kd-trees built using the SAH. However, when a plane is traversed, the probability of a plane intersecting a box should be used instead of the surface area. This probability – of a random plane intersecting for a box – is mentioned by Haines [Hai07] as the sum of the edges of the box. Replacing the surface area with the sum of edges would thus be more accurate and could create kd-trees that are better for *Row Tracing*.

Bibliography

[AC97] John Amanatides and Kin Choi. Ray Tracing Triangular Meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium*, pages 43–52, 1997.

[AGL91] Mark Agate, Richard L. Grimsdale, and Paul F. Lister. The HERO Algorithm for Ray-Tracing Octrees. In *Advances in Computer Graphics Hardware IV (Eurographics'89 Workshop)*, pages 61–73, London, UK, 1991. Springer-Verlag.

[AM01] Tomas Akenine-Mller. Fast 3D Triangle-Box Overlap Testing. *Journal of Graphics Tools*, 6(1):29–33, 2001.

[Ama84] John Amanatides. Ray Tracing with Cones. *SIGGRAPH Computer Graphics*, 18(3):129–135, 1984.

[AMH02] Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[APB87] Bruno Arnaldi, Thierry Priol, and Kadi Bouatouch. A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes. *The Visual Computer*, 3(2):98–108, August 1987.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[Are88] Jeff Arenberg. Ray/Triangle Intersection with Barycentric Coordinates. *Ray Tracing News*, 1(11), 1988.

[Arv88] J Arvo. Linear-time Voxel Walking for Octrees. *Ray Tracing News*, 1(5), 1988.

[AW87] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics '87*, pages 3–10, Amsterdam, August 1987. North-Holland.

[Bad90] Didier Badouel. An Efficient Ray-Polygon Intersection. *Graphics Gems*, pages 390–393, 1990.

[BCG$^+$96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOXTREE: A Hierarchical Representation for Surfaces in 3D. *Computer Graphics Forum*, 15(3):387–396, 1996.

[BCNJ08] Brian C. Budge, Daniel Coming, Derek Norpchen, and Kenneth I. Joy. Accelerated Building and Ray Tracing of Restricted BSP Trees. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 167–174, Aug. 2008.

[Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben79] J.L. Bentley. Multidimensional Binary Search Trees in Database Applications. *IEEE Transactions on Software Engineering*, SE-5(4):333–340, July 1979.

[Ben06] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures.* PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[BHO$^+$94] M. De Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. Van Kreveldt. Efficient Ray Shooting and Hidden Surface Removal. In *Algorithmica*, pages 21–30, 1994.

[Bou70] W. Jack Bouknight. A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Presentations. *Communications of the ACM*, 13(9):527–536, 1970.

[BP04] Thierry Berger-Perrin. SSE Ray/Box Intersection Test. available at `http://www.flipcode.com/archives/SSE_RayBox_Intersection_Test.shtml`, 2004.

[BP05] Thierry Berger-Perrin. Branchless Ray/Box intersections. available at `http://ompf.org/ray/ray_box.html`, 2005.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum, Proceedings of Eurographics, 2004*, 23(3):615–624, September 2004.

[BWS06] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical report, School of Computing, University of Utah., 2006.

[Car84] Loren Carpenter. The A-buffer, An Antialiased Hidden Surface Method. *SIGGRAPH Computer Graphics*, 18(3):103–108, 1984.

[Cat74] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, The University of Utah, 1974.

[CDP95] F. Cazals, G. Drettakis, and C. Puech. Filtering, Clustering and Hierarchy Construction: A New Solution for Ray-Tracing Complex Scenes. *Computer Graphics Forum*, 14(3):371–382, September 1995.

[Cha01] Allen Y. Chang. A Survey of Geometric Data Structures for Ray Tracing. Technical report, Polytechnic University, Brooklyn, Long Island, Westchester., 2001.

[Cla76] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[Coo86] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. *SIGGRAPH Computer Graphics*, 18(3):137–145, 1984.

[CS08] Daniel S. Coming and Oliver G. Staadt. Velocity-Aligned Discrete Oriented Polytopes for Dynamic Collision Detection. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):1–12, 2008.

[CW88] J G Cleary and G Wyvill. Analysis of an Algorithm for Fast Ray Tracing using Uniform Space Subdivision. *The Visual Computer*, (4):65–83, 1988.

[DHK08] H. Dammertz, J. Hanika, and A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum*, 27(4):1225–1233, June 2008.

[DK08] Holger Dammertz and Alexander Keller. The Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 155–158, Aug. 2008.

[EG07] M. Ernst and G. Greiner. Early Split Clipping for Bounding Volume Hierarchies. *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.*, pages 73–78, Sept. 2007.

[EGMM07] Martin Eisemann, Thorsten Grosch, Marcus Magnor, and Stefan Mueller. Fast Ray/Axis-Aligned Bounding Box Overlap Tests using Ray Slopes. *Journal of Graphic Tools*, 12(4), 12 2007.

[Eri97] Jeff Erickson. Plücker Coordinates. *Ray Tracing News*, 10(3), 1997.

[Eri07] Christer Ericson. Plucker coordinates considered harmful! available at `http://realtimecollisiondetection.net/blog/?p=13`, 2007.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by A Priori Tree Structures. *Computer Graphics, (Proceedings of SIGGRAPH 1980)*, pages 124–133, 1980.

[FLT08] Fast light Toolkit. available at `http://www.fltk.org`, 2008.

[FS88] Donald S. Fussell and K. R. Subramanian. Fast Ray Tracing using K-d Trees. Technical Report CS-TR-88-07, University of Texas at Austin, 1, 1988.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray-Tracing System. *Computer Graphics and Applications, IEEE*, 6(4):16–26, April 1986.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[GA93] I. Gargantini and H. H. Atkinson. Ray Tracing an Octree: Numerical Evaluation of the First Intersection. *Computer Graphics Forum*, 12(4):199–210, October 1993.

[GC91] Dan Gordon and Shuhong Chen. Front-to-Back Display of BSP Trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, 1991.

[GGW98] Jon Genetti, Dan Gordon, and Glen Williams. Adaptive Supersampling in Object Space using Pyramidal Rays. *Computer Graphics Forum*, 17(1):29–54, 1998.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer Visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM.

[Gla84] Andrew S. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[Gla90] Andrew S. Glassner. Normal coding. *Graphics Gems*, pages 257–264, 1990.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, New York, NY, USA, 1996. ACM.

[GM03] Markus Geimer and Stefan Müller. A Cross-Platform Framework for Interactive Ray Tracing. In *Proc. of GI Graphiktag*, pages 25–34, 2003.

[Got00] Stefan Aric Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, The University of North Carolina at Chapel Hill, 2000.

[Gre94] Ned Greene. Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. *Graphics gems IV*, pages 74–82, 1994.

[Gre96] Ned Greene. Hierarchical Polygon Tiling with Coverage Masks. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1996. ACM.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the Interaction of Light between Diffuse Surfaces. *SIGGRAPH Computer Graphics*, 18(3):213–222, 1984.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM.

[GW09] Anat Grynberg and Greg Ward. Conference Room. available at `http://radsite.lbl.gov/mgf/scenes.html`, 2009.

[Hai89] Eric Haines. Essential Ray Tracing Algorithms. *An Introduction to Ray Tracing*, pages 33–77, 1989.

[Hai94] Eric Haines. *Point in Polygon Strategies*, pages 24–46. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[Hai07] Eric Haines. Puzzle: Plane Intersection with Spheres vs. Boxes. *Ray Tracing News*, 20(1), 2007.

[Hav99] Vlastimil Havran. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News*, 12(2), 1999.

[Hav01] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, Czech Republic., 2001.

[Hav02] Vlastimil Havran. Mailboxing, yea or nay? *Ray Tracing News*, 15(1), 2002.

[HB97] Donald Hearn and M. Pauline Baker. *Computer Graphics (2nd ed.): C version*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[HB00] Vlastimil Havran and Jiří Bittner. LCTS: Ray Shooting using Longest Common Traversal Sequences. In *Proceedings of Eurographics (EG'00)*, pages 59–70, Interlaken, Switzerland, 2000.

[HB02] V. Havran and J. Bittner. On Improving Kd-trees for Ray Shooting. In *Proceedings of WSCG, pages 209–216, 2002.*, 2002.

[HH84] Paul S. Heckbert and Pat Hanrahan. Beam Tracing Polygonal Objects. In *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 119–127, New York, NY, USA, 1984. ACM.

[HHS06] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 71–80, September 2006.

[Hil00] Francis J. Hill. *Computer Graphics Using OpenGL*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[HKBv97] Vlastimil Havran, Tomás Kopal, Jiří Bittner, and Jiří Žára. Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools*, 2(4):15–23, 1997.

[HKRS02] Jim Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov. Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon*, 2002.

[HMS06] W. Hunt, W.R. Mark, and G. Stoll. Fast Kd-tree Construction with an Adaptive Error-Bounded Heuristic. *IEEE Symposium on Interactive Ray Tracing, 2006.*, pages 81–88, Sept. 2006.

[Hof96] Kenny Hoff. A Faster Overlap Test for a Plane and a Bounding Box. available at `http://www.cs.unc.edu/~hoff/research/vfculler/boxplane.html`, 1996.

[Hun08] Warren A. Hunt. Corrections to the Surface Area Metric with Respect to Mail-Boxing. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 77–80, Aug. 2008.

[Hur05] J. Hurley. Ray Tracing goes Mainstream. *Intel Technology Journal 9*, (2):99–108, 2005.

[HW91] Eric Haines and John Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proceedings of the Eurographics Workshop on Rendering*, pages 122–138. Springer Verlag, 1991.

[Int08] Intel. Pentium III Processors Technical Documents. available at `http://www.intel.com/design/intarch/pentiumiii/docs_pentiumiii_pga370.htm`, 2008.

[Int09] Intel C++ Compiler User and Reference Guides. 2009.

[ISP07] T. Ize, P. Shirley, and S. Parker. Grid Creation Strategies for Efficient Ray Tracing. *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.*, pages 27–32, Sept. 2007.

[IWP08] Thiago Ize, Ingo Wald, and Steven G. Parker. Ray Tracing with the BSP Tree. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 159–166, Aug. 2008.

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[Jon00] Ray Jones. Intersecting a Ray and a Triangle with Plucker Coordinates. *Ray Tracing News*, 13(1), 2000.

[JW89] D Jevans and B Wyvill. Adaptive Voxel Subdivision for Ray Tracing. In *Proceedings of Graphics Interface 89*, pages 164–172, 1989.

[KA91] D. Kirk and J. Arvo. Improved Ray Tagging for Voxel-Based Ray Tracing. In *Graphics Gems II*, pages 264–266. Academic Press, 1991.

[Kap85] M Kaplan. Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, pages 149–158. Addison Wesley, 1985.

[Kel97] Alexander Keller. Instant Radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[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, Mar. 1998.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM.

[KM07] Ravi P. Kammaje and B. Mora. A Study of Restricted BSP Trees for Ray Tracing. *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.*, pages 55–62, Sept. 2007.

[KM08] Ravi P. Kammaje and Benjamin Mora. Row Tracing using Hierarchical Occlusion Maps. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 27–34, Aug. 2008.

[KS97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster Ray Tracing Using Adaptive Grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, 1997.

[KS06] A. Kensler and P. Shirley. Optimizing Ray-Triangle Intersection via Automated Search. *IEEE Symposium on Interactive Ray Tracing, 2006.*, pages 33–38, Sept. 2006.

[LYMT06] C. Lauterbach, Sung-Eui Yoon, Dinesh Manocha, and D. Tuft. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing, 2006.*, pages 39–46, Sept. 2006.

[Mah05] Jeffrey A. Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies.* PhD thesis, University of Calgary, Calgary, Canada, 2005.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[ME05] Benjamin Mora and David S. Ebert. Low-Complexity Maximum Intensity Projection. *ACM Transactions on Graphics*, 24(4):1392–1416, 2005.

[MF00] Gordon Müller and Dieter W. Fellner. Hybrid Scene Structuring with Application to Ray Tracing. In *Proceedings of International Conference on Visual Computing (1999)*, 2000.

[MJC00] Benjamin Mora, Jean-Pierre Jessel, and René Caubet. Accelerating Volume Rendering with Quantized Voxels. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 63–70, New York, NY, USA, 2000. ACM.

[MJC02] Benjamin Mora, Jean Pierre Jessel, and René Caubet. A New Object-Order Ray-Casting Algorithm. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 203–210, Washington, DC, USA, 2002. IEEE Computer Society.

[MKJ08] Benjamin Mora, Ravi P. Kammaje, and Mark W. Jones. On the Lower Complexity of Coherent Renderings. Technical report, Department of Computer Science, Swansea University, 2008.

[ML03] Jos Pascual Molina Massó and Pascual González López. Automatic Hybrid Hierarchy Creation: a Cost-Model Based Approach. *Computer Graphics Forum*, 22(1):5–14, 2003.

[MT97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[MW04] Jeffrey Mahovsky and Brian Wyvill. Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plucker Coordinates. *Journal of Graphics Tools*, 9(1):35–46, 2004.

[NNS72] M E Newell, R G Newell, and T L Sancha. A New Approach to the Shaded Picture Problem. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.

[NT03] K. Ng and B. Trifonov. Automatic Bounding Volume Hierarchy Generation using Stochastic Search Methods. In *CPSC532D Mini-Workshop "Stochastic Search Algorithms"*, April 2003.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 1998.

[ORM07] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Proceedings of Eurographics Symposium on Rendering*, pages 85–98, Jun 2007.

[PB85] D.J. Plunkett and M.J. Bailey. The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, Aug. 1985.

[PGSS06] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with Streaming Construction of SAH KD-Trees. *IEEE Symposium on Interactive Ray Tracing, 2006.*, pages 89–94, Sept. 2006.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[PKGH97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[PSA07] Matt Pharr, Ken Shoemake, and Other Anonymous Authors. Evenly Distributed Points on Sphere. available at `http://www.cgafaq.info/wiki/Evenly_distributed_points_on_sphere`, 2007.

[ray] Ray-Polygon Intersection. available at `http://www.siggraph.org/education/materials/HyperGraph/raytrace/raypolygon_intersection.htm`.

[Rit90] Jack Ritter. An Efficient Bounding Sphere. *Graphics gems*, pages 301–303, 1990.

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, 24(3):1176–1185, 2005.

[RUL00] J. Revelles, C. Urea, and M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. In *Proceedings of WSCG, 2000*, pages 212–219, 2000.

[RW80] Steven M. Rubin and Turner Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics, (Proceedings of SIGGRAPH 1980)*, pages 110–116, 1980.

[Sam89] Hanan Samet. Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers And Graphics*, 13:445–460, 1989.

[SB87] John M. Snyder and Alan H. Barr. Ray Tracing Complex Models Containing Surface Tessellations. *SIGGRAPH Computer Graphics*, 21(4):119–128, 1987.

[SBGS69] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp. Study for Applying Computer Generated Images to Visual Simulation. Technical report, General Electric, 1969.

[SCS$^+$08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core X86 Architecture for Visual Computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[SF90] K. R. Subramanian and Donald S. Fussell. Factors Affecting Performance of Ray Tracing Hierarchies. Technical report, Austin, TX, USA, 1990.

[SF91] K. R. Subramanian and Donald S. Fussell. Automatic Termination Criteria for Ray Tracing Hierarchies. In *Proceedings of Graphics Interface 91*, pages 93–100, 1991.

[SF01] R. J. Segura and F. R. Feito. Algorithms To Test Ray-Triangle Intersection. In *Proceedings of WSCG.*, pages 200–205, 2001.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant Polygon Clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[Sho98] Ken Shoemake. Plücker Coordinate Tutorial. *Ray Tracing News*, 11(1), 1998.

[SL96] Peter Soderquist and Miriam Leeser. Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations. *ACM Computing Surveys*, 28(3):518–564, 1996.

[SS92] Kelvin Sung and Peter Shirley. Ray tracing with the BSP tree. *Graphics Gems III*, pages 271–274, 1992.

[SSE09a] SSE2. *Wikipedia*, 2009.

[SSE09b] Streaming SIMD Extensions. *Wikipedia*, 2009.

[SSK07] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum (Proceedings of Eurographics).*, 26(3), 2007.

[STN87] Mikio Shinya, T. Takahashi, and Seiichiro Naito. Principles and Applications of Pencil Tracing. *SIGGRAPH Computer Graphics*, 21(4):45–54, 1987.

[Sub91] Kalpathi Raman Subramanian. *Adapting Search Structures to Scene Characteristics for Ray Tracing.* PhD thesis, University of Texas at Austin, Austin, TX, USA, 1991.

[Sun91] Kelvin Sung. A DDA octree traversal algorithm for ray tracing. In *Proceedings of Eurographics '91*, pages 73–85, 1991.

[Sze03] Laszlo Szecsi. *Graphics Programming Methods*, chapter An Effective Implementation of the K-D tree, pages 315–326. Charles River Media, Inc., Rockland, MA, USA, 2003.

[TH99] Seth Teller and Michael Hohmeyer. Determining the Lines Through Four Lines. *Journal of Graphics Tools*, 4(3):11–22, 1999.

[Thi87] William Charles Thibault. *Application of Binary Space Partitioning Trees to Geometric Modeling and Ray-Tracing.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1987. Director-Bruce F. Naylor.

[VCI09] Visual C++ Language Reference, Compiler Intrinsics. available at `http://msdn.microsoft.com/en-us/library/26td21ds.aspx`, 2009.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wal05] Ingo Wald. The RTRT core. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 5, New York, NY, USA, 2005. ACM.

[Wal07] I. Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.*, pages 33–40, Sept. 2007.

[War69] John Edward Warnock. *A Hidden Surface Algorithm for Computer Generated Halftone Pictures.* PhD thesis, The University of Utah, 1969.

[Wat70] Gary Scott Watkins. *A Real Time Visible Surface Algorithm.* PhD thesis, The University of Utah, 1970.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, pages 49–57, Aug. 2008.

[WBMS05] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools*, 10(1):49–54, 2005.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):485–493, 2007.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum (Proceedings of*

EUROGRAPHICS 2001, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001.

[WFMS05] Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005.

[WFP$^+$01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The Randomized Z-buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 361–370. ACM Press / ACM SIGGRAPH, 2001.

[WH06] I. Wald and V. Havran. On Building Fast Kd-Trees for Ray Tracing, and on Doing That in O(N log N). *IEEE Symposium on Interactive Ray Tracing, 2006.*, pages 61–69, Sept. 2006.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, 3(1):52–69, 1984.

[Whi80] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[WIK$^+$06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, pages 485–493, 2006.

[WK06] C Wachter and A Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[WMG$^+$] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, pages 89–116.

[Woo90] Andrew Woo. Fast Ray-Box Intersection. *Graphics gems*, pages 395–396, 1990.

[WREE67] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl. Half-Tone Perspective Drawings by Computer. In *AFIPS '67 (Fall): Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 49–58, New York, NY, USA, 1967. ACM.

[WSC$^+$95] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-R: An Adaptive Octree for Efficient Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, 1995.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005.

[WWZ$^+$06] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. Guided Visibility Sampling. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 494–502, New York, NY, USA, 2006. ACM.

[YLM06] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Fast LOD-based Ray Tracing of Massive Models. *The Visual Computer*, 22(9):772–784, 2006.

[Zha98] Hansong Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1998.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff. Visibility Culling using Hierarchical Occlusion Maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[ZRJ95] Maurice Van Der Zwaan, Erik Reinhard, and Frederik W. Jansen. Pyramid Clipping for Efficient Ray Traversal. In *Proceedings of Eurographics Workshop on Rendering 95*, pages 1–10. Springer, 1995.

Appendices

Appendix A

Software Design

A software system that uses several different underlying data structures and algorithms such as ours must have an extensible design in which it is easy to add new rendering methods. The main aim of the software is to assist and ease research. Thus, the emphasis is on the software being a platform through which several algorithms can be easily implemented and compared. An object oriented approach is used to develop the system, with the majority of rendering methods having their own classes. The implementation follows a modular design that should be easy to understand for a new developer in addition to being new components.

The system was implemented in C++, using Visual Studio 2005 / 2008 as the IDE. For a majority of the application and benchmarks, the inbuilt compiler has been used. However, for a few algorithms, like the *Row Tracing* algorithm, where maximal performance was desired, Intel's C++ compiler was used to compile the application. In addition, profiling was undertaken with the Intel's VTune profiling application to identify the bottlenecks in the application.

A renderer, especially one that should be able to use several interchangeable data structures and algorithms, is an intricate system with several classes interacting with each other. The aim of the software is to provide the user with an easy to use system whereby s/he can easily select the viewpoint. The application allows the selection of the camera parameters like perspective (angle of view), zoom, camera location, etc through mouse based interaction with the scene. The system should also allow the user to import models / scenes into the native format of the system. So far, the application is able to import the Stanford ply format, obj format, 3DS format and a raw triangle format. A separate importer for VRML models, currently not integrated into the system, has also been implemented using the OpenVRML library. Importers for several formats were necessary to have a set of commonly used models so that comparisons could be made against published algorithms using the same scenes.

This chapter will detail the design issues, including the software and user interface design.

A.1 Software Architecture

The software was developed using an object oriented methodology. The main algorithms and data structures were encapsulated into their own separate classes. The class diagram in Figure A.1 shows the class diagram for the system.

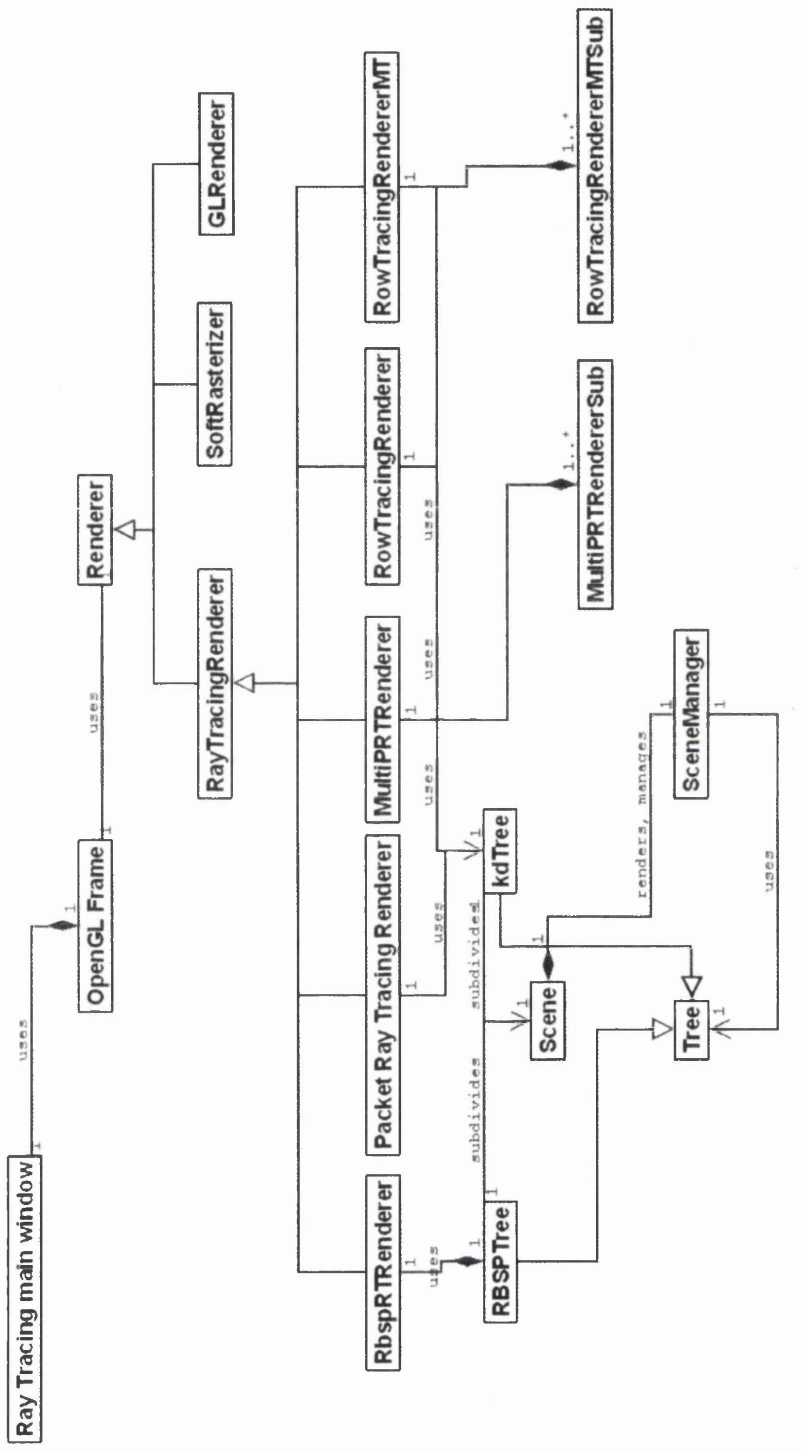


Figure A.1: Software Class diagram.

The diagram shows that the architecture is a very simple one that follows object oriented concepts. The data – the scene to be rendered – is abstracted from the auxiliary structures (the trees), the user interface and the rendering algorithms.

The user interface classes – not shown in their entirety, but represented by classes 'Ray Tracing main window' and 'OpenGL Frame' in the diagram – are responsible for displaying the user interface and receiving input from the user. The user inputs are received by the interface class and passed on to the back end of the application. The class *SceneManager* is responsible for the action of passing these commands from the front end to the back end. The scene to be rendered is represented by the 'Scene' class that enables it to be in memory and its components to be accessible to the renderer. The *SceneManager* creates the scene to be rendered by either loading one that is stored in the hard disk or by importing one that exists in a different format. Once the scene is loaded, most algorithms work with a tree as the underlying structure. Hence, a tree with a user selected set of parameters is either loaded (if already created and stored onto the hard disk) or created. Finally, when a rendering type is selected, the *SceneManager* creates the appropriate renderer to render the scene. The *SceneManager* is thus a very important class.

A.1.1 Renderers and Data Structures

The application was designed to allow integrating several data structures and algorithms for rendering. The first step was to declare a set of abstract super classes to define the main functionalities.

As the application's main task is to render scenes, an abstract class – *Renderer* – was created that defined the main functions to be implemented by any rendering method.

Renderer – This class is the super class for all software rendering methods. It is an abstract class defining several abstract methods that its subclasses have to implement. The main abstract method it defines is:

- *render* – This method has to be implemented by each subclass, i.e., each new rendering method. The subclasses use the specific algorithm to generate the image and the bitmap to be rendered is stored in an object of the *FrameBuffer* class. This object is then passed onto the *OpenGL Frame* object to display the contents of the current *FrameBuffer*.

As an abstract super-class, it is a placeholder for all rendering methods. Each rendering method is defined as a subclass that implements this *render* method. A class *RayTracingRender* implements a baseline ray tracing method using kd-trees. This method is used as the method against new algorithms and data structures are compared. The *Coherent Rendering* method described in chapter 4 is implemented using the subclass *SoftRasterizer*.

However, when the method used is very similar to ray tracing, it is possible that several functionalities of ray tracing are duplicated. Hence, these rendering methods – Ray tracing using *RBSP trees* (implemented by class *RbspRTRenderer*), Packet ray tracing (class *PacketRayTracingRenderer* – also a method used as a baseline for comparison), Multi threaded packet ray tracer (class *MultiPRTRenderer* – another baseline rendering method), *Row Tracing* (class *RowTracingRenderer* – implementing single threaded *Row Tracing* and *Packet Row Tracing*) and multi-threaded Row tracer (class *RowTracingRendererMT* – implementing multi threaded *Row Tracing* and *Packet Row Tracing*) – derive from *RayTracingRenderer*.

The manager class – *SceneManager* – manages the delegation of work to the rendering methods based on the user's selection. Depending on the rendering method chosen, this class creates the appropriate renderer object with chosen rendering parameters. The object then renders the scene and displays the image.

As demonstrated, the addition of new rendering algorithms is very simple. The new algorithm is implemented in its own class. The only modifications necessary to the implementation are to the *SceneManager* class that has to create an object of the new renderer's class.

Further, the several data structures are abstracted out of the rendering method so that a renderer chooses whatever data structure it deems suitable. For eg., *Row Tracing* has been implemented on kd-trees as well as octrees. This is easily achieved as the data structure classes are separated from the rendering classes.

The data structures are also implemented in a similar manner. Since the structures investigated are all tree structures, an abstract super class called *Tree* defines the main tree structure. All trees are implemented as subclasses to this class.

The first data structure implemented was the *kd-tree*, defined in its own class *KDTree*.

kdTree The class represents a kd-tree. It provides the data variables and methods necessary to create and use kd-trees.

A few of the important data members of the kd-tree class are as follows.

- *noNodes* – The number of nodes in the kd-tree. Since the tree is stored as an array in memory for efficiency, the number of nodes is used to allocate the nodes and to ensure that out of bounds array items are not accessed.
- *nodeArray* – The nodes of the kd-tree are stored as an array in this structure. The array consists of *KDTreeNode* items that span eight bytes each. Each *KDTreeNode* represents either an internal node of the tree or a leaf node of the tree. The structures of these two kinds of nodes are shown in Tables A.1 A.2 respectively.

Represents	Bits used
Unique pointer to children	32
Quantized value of split position	14
Leaf node flag	1
Unused bit	1
Split axis	16

Table A.1: Kd-tree node structure.

Represents	Bits used
Pointer to start of triangle list	32
Number of triangles in node	16
Leaf node flag	1
Unused bits	15

Table A.2: Kd-tree leaf node structure.

- *noTotalTriangles* – The number of triangles contained by all the leaf nodes of the kdTree.
- *triangleArray* – In the leaf node of the kd-tree, an index an item of this array containing a list of triangles is stored. The index points to the first triangle contained in the leaf node. The leaf node also specifies the number of triangles in it. Together, the two members specify the triangles contained in the leaf node.
- *kdTreeSplitMethod* – Indicates the splitting method of the construction process. The split method could be one of space median, improved space median, SAH or octree. The octree can be considered as a version of a kd-tree where space is split across all three axes and has eight child nodes instead of three. However, the octree due to its uniformity does not need its split position to be represented explicitly and can thus be represented with just 32 bits that indicate the location of the first child node.

In addition to the data members, the *kdTree* class also provides methods to create the kd-tree with several different heuristics. The main methods provided are:

- *ConstructTree* – The method that constructs the tree. It is a recursive method that works in two passes. Due to the fact that nodes are stored in an array, the first pass counts the number of nodes and triangles necessary and the second pass includes the data in these nodes and triangles. The main factor in the quality of the kd-tree is the split method and this is implemented in a separate method that returns the split position to be used for the particular construction step. Upon determination of the split position, the construction step then classifies the triangles as being on one of the sides or both sides of the split and recurses if necessary.
- *CountNodesAndTriangles* – As mentioned earlier, the construction step is a two pass process. This method is performed in the first pass and only counts the number of nodes and triangles that the final tree must contain.
- *findPosSAH* – Uses the *Surface Area Heuristic* to determine the locally optimal *SAH* position. The method uses several auxiliary methods to achieve this. It first clips the triangles in the node by the bounding planes to obtain the potential split positions. Then, it sorts these split points along all three axes. At each of these points, the *SAH* cost is determined along three axes and the point with the minimum cost is determined as the locally optimum *SAH* position.
- *SortTriangles* – This is the method that classifies the triangle as being to the left or to the right of the split in consideration. An in place method is used that minimises slow memory allocation and de-allocation operations.
- *saveTreeBin* – Once the tree with the given parameters is created, it is saved to the hard disk for further use. The tree is stored in a binary format that is just a binary dump of the bytes in the *nodeArray* followed by the bytes in the *triangleArray*.
- *loadTreeBin* – If a scene already has a tree constructed and saved, this method loads the tree from the hard disk rather than constructing the tree from scratch.
- *TraceRay* – The method traces a ray through the kdTree. It initialises several variables necessary for the traversal and calls the *RecursiveRayTraversal* method.
- *RecursiveRayTraversal* – The tree is recursively traversed by a ray in front-to-back order until either a leaf node is reached or until an exit criteria is reached.

- *ProcessLeafNode* – If the recursive traversal reaches the leaf node, then this method processes the leaf node and the ray. The ray is intersected with all the triangles in the node and if there are one or more intersected triangles, the one with the closest intersection – as given by the t intersection parameter – is returned.

As the *KDTree* class shows, the class for each data structure defines all the functionalities necessary. They define construction of the structure with several heuristics if necessary. Since the *KDTree* was initially developed in our system for ray tracing, the ray traversal code is integrated in this class. However, this is planned to be moved to the *RayTracingRenderer* class to make the design more coherent. In addition, the structure class is responsible for serialising – loading and saving the structure from and to hard disks – the structure.

Since the application considers the octree as a special form of kd-trees, a separate class has not been created for octrees. Instead, the *kdTreeSplitMethod* variable indicates that the tree is an octree and the *ConstructTree* method ensures that an octree is created.

The other structure developed is the *RBSP tree* – described in Chapter 3. Although it is similar to kd-trees, it has very specific construction and ray traversal methods and hence has been included in its own class. The class defines the construction, loading and saving of *RBSP trees*. Rendering using *RBSP trees* is handled by the *RBSPTreeRenderer* class.

A.1.2 Scene Structure

Another consideration is the lower level consideration of how the actual scene is represented in memory. This can have a major impact on the efficiency and extensibility of the system. It is necessary to represent the scene so that it incorporates the main features of most rendering systems like textures, lights, vertex normals, etc. in addition to the actual geometry. It is necessary to point out at this time that the geometry of the scene consists solely of triangles. Also, the scene has to be represented so that vertices shared between triangles are not duplicated. In addition, it also necessary to abstract the scene from the rest of the application so that the actions pertaining to it are achieved independently. The class *Scene* is thus implemented to fulfil these objectives.

Scene – The *Scene* class represents scenes to be rendered. The structure of an object of this class is as follows:

- *vertices* – This is a list of vertices of the scene's triangles. The list contains four floating point values for each vertex. Although, a vertex can be represented by just three floats, four values are used with a dummy value at the end so that the data is better aligned.
- *nbOfVertices* – Indicates how many vertices exist in the scene. This is necessary as the vertices are stored in an array that does not contain size information.
- *triangles* – This is a list of indices that point to a vertex in the vertices array. Each triangle is indicated by the indices of the first coordinate of each of the three vertices forming the triangle. The coordinates of the vertices are obtained by taking the three consecutive floating point number for each vertex.
- *nbOfTriangles* – Indicates the number of triangles in the scene.
- *vertexNormals* – A method similar to the vertices is used to represent the normals at every vertex. The number of vertex normals is the same as the number of vertices. Hence, the

same number – *nbOfVertices* – can be used. Four floating point values in which one is a dummy is used to indicate the normal vector at the vertex.

- *textureDimensions* – The dimensions of the textures used are stored in this value. It stores two integers per texture to indicate the length and the breadth of the texture.
- *textures* – This is a two dimensional byte array consisting of RGB values of a texture. The array contains all the textures used in the scene and is referenced to obtain the colour due to the texture.
- *vertexColorsOrTextureCoordinates* – This is the colour of a vertex or its texture coordinates. The texture coordinates allow determining the colour at a pixel after the texture has been applied. In case it is a colour, an RGB value is stored.
- *triangleTextureNb* – This is a pointer to a texture for every triangle in the scene. The texture's colour is mapped onto the triangle to obtain the textured triangle.
- *lightSourcePatches* – Indicates the lights in a scene. Every light source is indicated with a list of triangles that form the light source. The representation is similar to how triangles are stored.
- *lightSourceRadiance* – Provides the radiance values for every light source, and is indicated by one floating point value per light source.
- *materials* – This is a list of materials contained in the scene. The materials are defined in a separate class that define values for diffuse, ambient and specular components in addition to the opacity, refraction index and shininess values for the material. This list contains a list of materials used by the scene.
- *nbOfMaterials* – The number of materials contained in the scene.
- *vertexMaterialIndex* – A pointer to the material at this particular vertex.

The *Scene* class also contains a few methods to manipulate the scene. The main methods of the *Scene* class are:

- *LoadScene* – This method loads a scene stored in this format to memory from the hard disk.
- *SaveScene* – Saves the currently loaded scene to the hard disk.
- *AutoGenerateNormals* – Generates the normals at the vertices using the triangle's normal.
- *InvertNormals* – On a few occasions, the normals are directed opposite to how the scene is structured. In these cases, this method can invert the normals so that the triangles of the scene are oriented in the right direction.
- *ComputeBoundingBox* – Computes the bounding box of the scene. The bounding box of the scene is represented by six coordinates – the minimum and maximum coordinates along each axis. This is computed by determining the minimum and maximum coordinates of all the triangles in the scene.
- *operator+=* – Combines two scenes to obtain a combined scene incorporating the two scenes.

The class thus designed represents the scene and the necessary components with minimal duplication of data. The application currently loads and operates on only one scene at a time, though loading and operating on multiple scenes is relatively easy. Again, the *SceneManager* class manages the application by being the central object with which other components communicate. The *SceneManager* class – being the main class sending and receiving messages – is thus an important class of the application.

SceneManager The *SceneManager* is a class that manages the operations of the application. It is the direct interface between the user interface, the back end rendering classes and other classes that can be independent. It is thus responsible for calling the appropriate methods when the user requests an action. Its main data members are:

- *scene* – The *scene* object that is to be rendered.
- *renderer* – The *renderer* object that is responsible to render the scene. According to the method chosen, the appropriate *renderer* object is constructed and the *render* method is called.
- *tree* – The *tree* using which the scene is rendered.

Using the above data members, the *SceneManager* is responsible for loading the scene and the corresponding tree. It allows the user to specify the parameters with which s/he wants to render the scene. It also provides functionality to change the type of tree being used and the rendering method used, to import scenes of different formats into the application and finally to save the parameters for a rendering or even to save the scene being rendered in native format. It is to be noted that the *SceneManager* does not actually implement any of the operations and just calls the corresponding method from the corresponding object. The methods that allow the *SceneManager* to achieve its responsibilities are:

- *ChangeRenderingStyle* – The method allows changing the rendering type from OpenGL based to ray tracing, ray tracing with *RBSP trees*, *Row Tracing* and packet ray tracing. The method creates a new renderer of the appropriate type and calls its *render* method.
- *ChangeKDTreeMethod* – Changes the kd-tree's splitting method between space median, SAH and octree types.
- *ChangeKDTreeParameters* – Changes other parameters of the kd-tree like the maximum depth of the tree and number of primitives contained in the leaf node.
- *ChangeRBSPTreeParameters* – Similar to the kd-tree, the parameters of the *RBSP tree* can be changed. In addition to the maximum depth and the number of triangles in the leaf node, the *RBSP tree* has an additional parameter – the number of axes used to construct the tree.
- *LoadOrConstructTree* – Checks if the tree has already been constructed. If it has been, the tree is loaded into the main memory. Otherwise, the tree construction method is called.
- *SavePreferences* – Preferences are parameters of a particular rendering scenario. It defines the scene to render, camera parameters, the tree to use along with its main properties. These parameters are saved for further retrieval by the *LoadPreferences* method. The preferences make it easy to compare several algorithms and data structures with a given scenario. It is also helpful when debugging problematic scenarios.
- *LoadPreferences* – This loads the preferences including the scene and the tree given by the parameters. Subsequently, it applies the camera parameters stored in the parameters file to the camera so that the viewpoint is exactly as it was when the preferences were saved.

- *Load3DS, LoadPly, LoadIW, LoadPlyDir* – Allows a user to import scenes of various formats into the format used by the application. The method calls an importer that imports the scene.
- *LoadScene* – Loads the scene from memory by calling the appropriate method in the *Scene* class.
- *SaveScene* – Saves the scene to the hard disk by calling the appropriate method in the *Scene* class.
- *LoadSceneFromXML* – Using XML files, several scenes can be combined into one scene. This is very useful when comparing several different methods of rendering. The *SceneManager* calls the XML importer to achieve this functionality.

The description of the *SceneManager* clarifies the role of the *SceneManager* as being integral to the extensibility and simplicity of the design. By being an intermediary class between the various components of the application, it allows the application to be modular, simple and extensible.

Thus, with this combination of individual rendering classes for each rendering method, separate classes for every major data structure, a detailed *Scene* class and a manager class in the *SceneManager* class, the design is flexible enough that developing a combination of structures and rendering methods is possible very easily.

A.2 Scene and Tree Data Structure Representation

The structures for the scene and the tree aim to maximise memory efficiency. This also achieves good cache performance. While the members and methods of these structures has been described earlier, the section below aims to explain them with more clarity.

A.2.1 Scene Structure

In SectionA.1.2, the class structure for representing a scene was described. However, from the class structure, it maybe difficult to identify the relations between the various data members of the scene. Figure A.2 aims to clarify the structure of the scene and its representation in memory once it has been loaded.

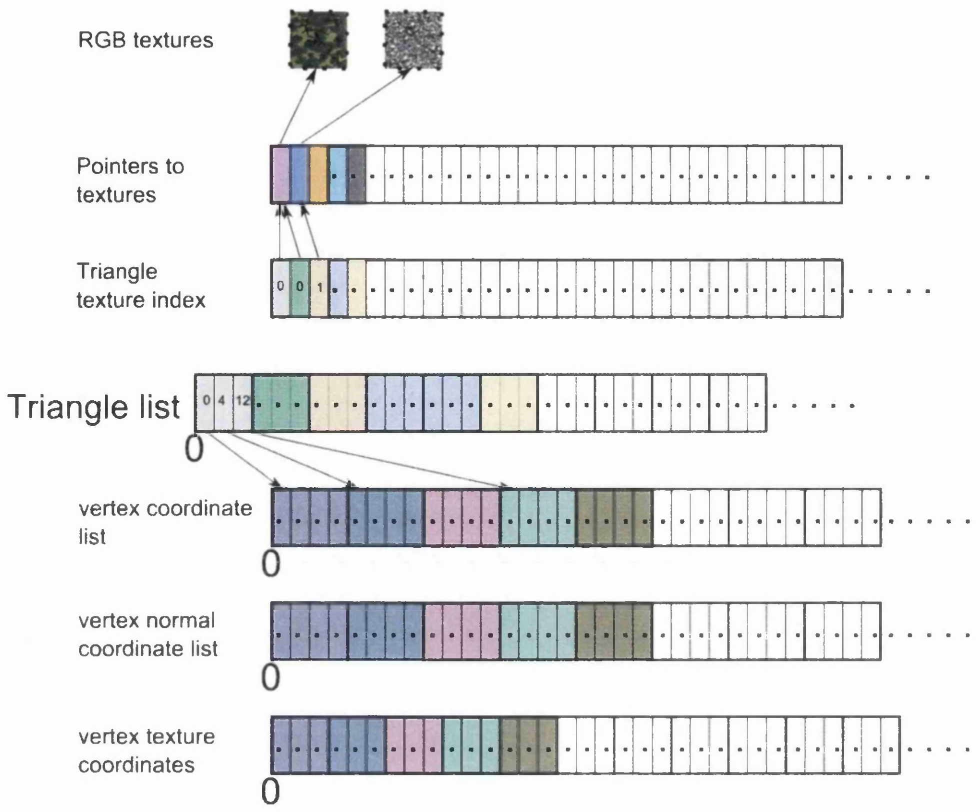


Figure A.2: Structure of a *Scene* in memory.

Figure A.2 showing only the main objects of the scene, indicates that the scene object consists of a number of arrays for the main elements of the scene. Each triangle in the scene consists of three pointers to the vertex array that consists of a list of all the vertices. The vertex list is composed of four components per vertex – the X, Y and Z coordinates and a fourth component that is always the value 1. The additional component is added for memory alignment issues. The vertex normals are similarly stored. The vertex texture coordinates are stored in a similar array, but with three components per vertex as it is accessed only during the shading part of the rendering. Textures are loaded into memory as 2D RGB values and an array of pointers to the 2D textures is created. If a triangle is textured, a pointer in the texture index array indicates the particular texture for the triangle in consideration. In this way, the major components of the scene are represented.

A.2.2 Kd-tree Data Structure

The kd-tree has a structure that minimises the memory usage, thus enabling optimal cache usage. It consists of a kd-tree class that represents the tree. The nodes are defined in this class. In order to save memory and maximise coherence, the nodes of the tree are stored as an array. Due to this representation, the two child nodes can be easily indicated by just one index as long as the two child nodes are stored in adjacent items of the array. In addition to saving memory, the array representation ensures that sub-trees are stored closer, meaning that if a node is loaded into the cache, several adjacent nodes that are also loaded have a high probability of being accessed.

In this representation, the first element of the array is the root node of the tree. The two child nodes of the root node are stored as the next two elements in the array. Similarly, each of the nodes has two child nodes unless they are leaf nodes.

Obviously, nodes can be of two types – internal nodes and leaf nodes. Both kinds of nodes can be indicated with just 8 bytes (64 bits). Internal nodes need to store the split position, the axis that splits it, an indication of if it is a leaf node and finally a pointer to the first child node.

Represents	Bits used
Unique pointer to children	32
Quantised value of split position	14
Leaf node flag	1
Unused bit	1
Split axis	16

Table A.3: kd-tree node structure

Each internal node is stored with the above structure. However, when a leaf node is to be represented, it has to store different information – the number of triangles in the node, the triangles in the node and an indicator specifying if it is a leaf node.

Represents	Bits used
Pointer to start of triangle list	32
Number of triangles in node	16
Leaf node flag	1
Unused bits	15

Table A.4: kd-tree leaf node structure

Storing the actual triangles in the node is impractical. Hence, an index is stored. The index points to a triangle in the list of triangles that contains all the triangles of all the leaf nodes. The triangle is the first triangle in the node. In addition, the number of triangles in the node is also stored. The first triangle and the number of triangles thus enable identification of the triangles in the node. These triangles are stored as indices pointing to the list of triangles in the *Scene* object. The *Scene* object then allows access to the vertices of the triangle. The representation of triangles is made clear by Figure A.3

The data thus represented provides a modular and efficient way to access the data. It minimises redundant data being stored keeping the structures small. This makes the application memory efficient and also improves performance by better cache coherence and reduced paging.

A.3 User Interface Design

As an application designed mainly for research purposes, the user interface for the application had several aims. It would have to be an application that:

- Is Cross platform.

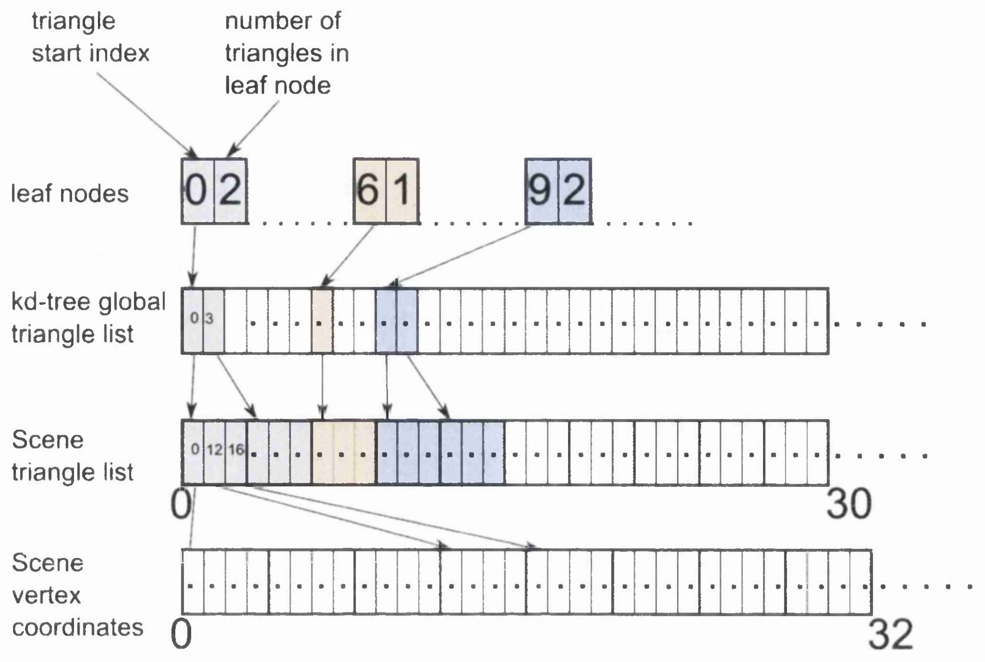


Figure A.3: Obtaining the leaf node triangles.

- Is simple to use / has a minimal learning curve.
- Allows easy selection of a scene.
- Allows easy manipulation of the camera.
- Assists debugging.
- Enables rendering with different methods.
- Enables selection of different data structures.
- Enables comparison between different methods.

The interface has been designed with these aims in mind. The user interface design of the application with the above aims in mind, is detailed below.

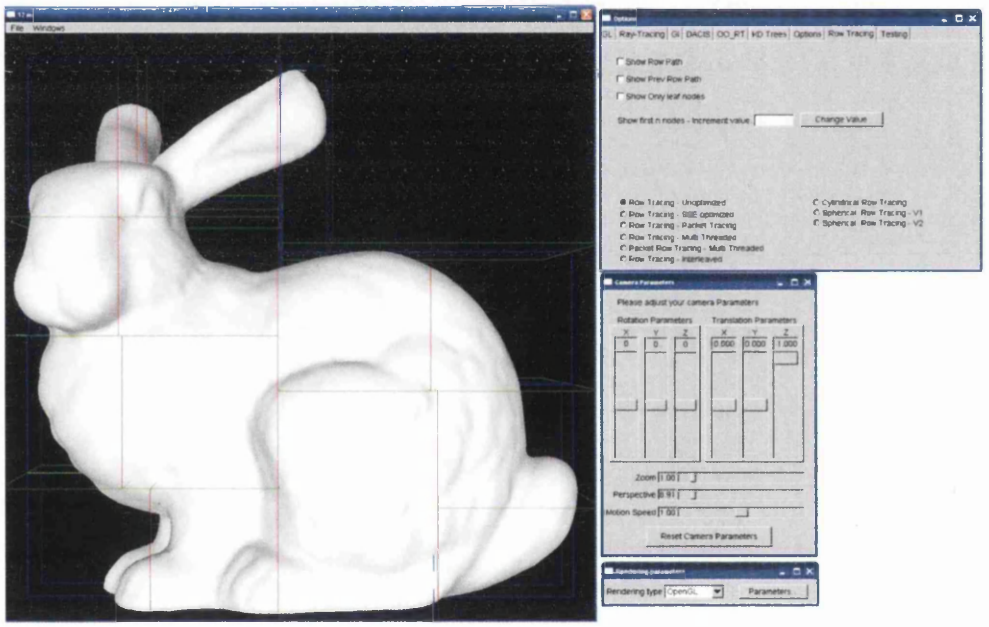
Cross platform – Although, currently the application only works on Windows, it was decided early on during the development that a cross platform application is desired. Compatibility with the Linux / Unix platform was desirable due to the robustness of the system as also due to the possibility of faster performance. For this purpose, FLTK – the Fast Light Toolkit – was selected as the GUI toolkit. From FLTK's website [FLT08] –

> 'FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX/Linux (X11), Microsoft Windows, and MacOS X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL and its built-in GLUT emulation.
>
> FLTK is designed to be small and modular enough to be statically linked, but works fine as a shared library. FLTK also includes an excellent UI builder called FLUID

that can be used to create applications in minutes.'

As the quote from FLTK suggests, it is a cross platform GUI development toolkit that is lightweight. Additionally it supports OpenGL – a key feature for graphics applications. Development of the user interface is easily achieved through the provided FLUID tool that is fairly simple to use. These factors made the decision to use FLTK a straightforward one.

Simple to use / has a minimal learning curve – The application developed by us has to be simple to use. It was necessary that it popularly used functions could be accessed with minimal effort. Thus, it was imperative that the application followed a standard approach – that it had a GUI where all the functions were accessible using a mouse. The main windows of the application can be shown by Figure A.4.


File Windows
GL | Ray-Tracing | GI | DACIS | OO_RT | KD Trees | Options | Row Tracing | Testing
Show Row Path
Show Prev Row Path
Show Only leaf nodes
Show first n nodes - Increment value
Change Value
Row Tracing - Unoptimized
Row Tracing - SSE optimized
Row Tracing - Packet Tracing
Row Tracing - Multi Threaded
Packet Row Tracing - Multi Threaded
Row Tracing - Interleaved
Cylindrical Row Tracing
Spherical Row Tracing - V1
Spherical Row Tracing - V2
Camera Parameters
Please adjust your camera Parameters
Rotation Parameters
Translation Parameters
X Y Z
X Y Z
Reset Camera Parameters
Parameters

Rotation X 0, Y 0, Z 0
Translation X 0.000, Y 0.000, Z 1.000
Zoom 1.00
Perspective 0.91
Motion Speed 1.00
Rendering type OpenGL

Figure A.4: Application Windows

The figure shows the application's simplicity. It has four windows with the largest one being the one in which the image is rendered.

Easy selection of scene When a scene is loaded, it is in application's memory and is displayed in the largest window. The scene is chosen by selecting the appropriate item in the *File* menu shown in Figure A.5. As the figure shows, the menu enables scenes of several different formats to be loaded using the GUI. In this manner, the scene is selected in an easy manner.

Easy manipulation of camera – To compare algorithms, it is necessary to set a viewpoint that is suitable. To achieve this, easy navigation through the scene and setting the necessary viewpoint and camera parameters is necessary. The application enables this easily by allowing

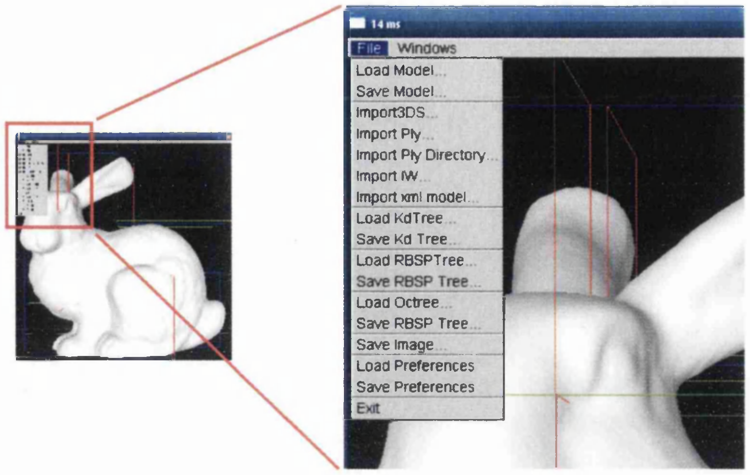
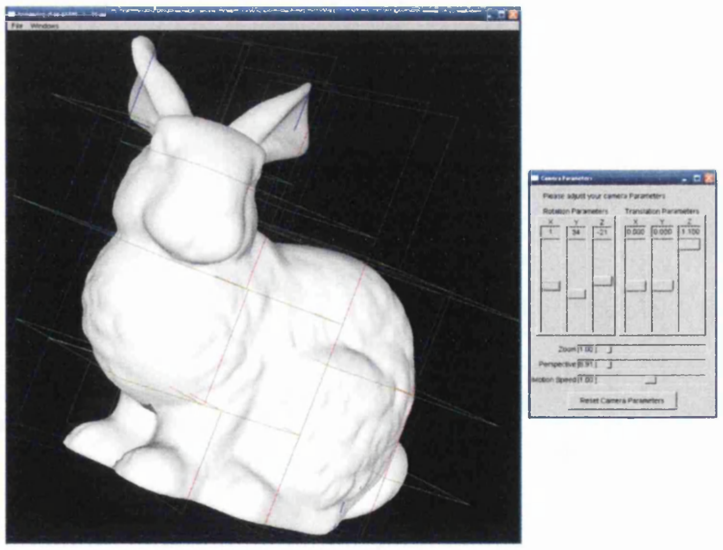


Figure A.5: File Menu

the user to rotate, translate and scale the model using the mouse. The application also allows setting the perspective angle and the zoom factor of the camera. The user can move / rotate the camera by dragging the right mouse button and the left mouse button respectively. In addition, the application provides these functionalities by way of sliders that allow fine tuning the camera settings if necessary. Figure A.6 shows the main window in which the scene can be navigated with the mouse. Figure A.6 also shows the window with sliders with which rotations and translations can be applied to the model.

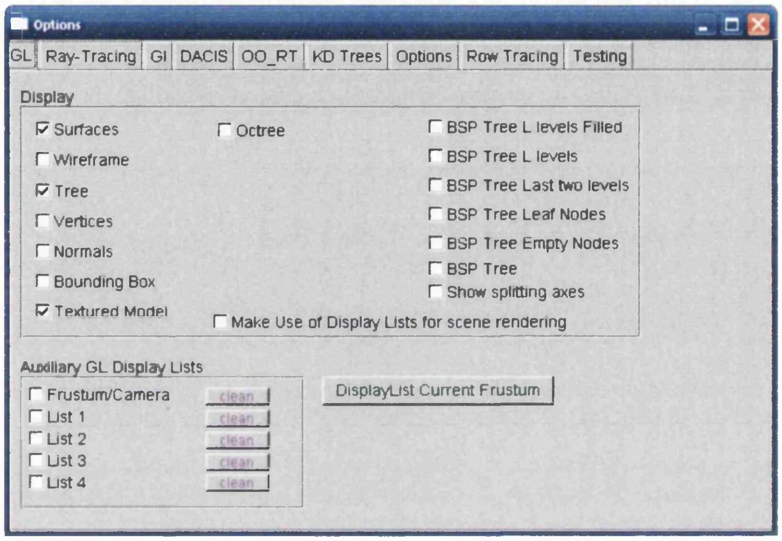


Rotation Parameters: X 1, Y 34, Z -21
Translation Parameters: X 0.000, Y 0.000, Z 1.100
Zoom 1.00
Perspective 0.91
Motion Speed 1.00

Figure A.6: Setting the camera parameters

Assist debugging – During the development of new algorithms and data structures, there are bound to be several instances where things do not work as expected. During these instances, it is useful to visualise the method and the underlying data structures.

When a new structure is being developed, it needs to be visualised so that the problems can be identified and fixed with minimal effort. The user interface allows the user to easily select one of the several data structures implemented and to visualise it. Figure A.7 shows how several aspects of a data structure can be visualised depending on what is necessary.


Options

GL | Ray-Tracing | GI | DACIS | OO_RT | KD Trees | Options | Row Tracing | Testing

Display

- [x] Surfaces
- [ ] Wireframe
- [x] Tree
- [ ] Vertices
- [ ] Normals
- [ ] Bounding Box
- [x] Textured Model
- [ ] Octree
- [ ] BSP Tree L levels Filled
- [ ] BSP Tree L levels
- [ ] BSP Tree Last two levels
- [ ] BSP Tree Leaf Nodes
- [ ] BSP Tree Empty Nodes
- [ ] BSP Tree
- [ ] Show splitting axes
- [ ] Make Use of Display Lists for scene rendering

Auxiliary GL Display Lists

- [ ] Frustum/Camera clean
- [ ] List 1 clean
- [ ] List 2 clean
- [ ] List 3 clean
- [ ] List 4 clean

DisplayList Current Frustum

Figure A.7: Options to visualise the scene and the data structure

The Figure A.8 shows the *RBSP tree* visualised using the wireframe method where each node's edges are shown. During the development of the structure, this visualisation enabled us to judge the quality of the tree and to identify the problems. As Figure A.7 shows, several other types of visualisations have been added to assist the development process. These visualisations also show the characteristics and quality of the new structure developed.

Enable rendering with different methods – The user interface makes it very simple to choose the rendering method. As Figure A.9 shows, the rendering method can be chosen and changed easily, making it easy to compare different rendering methods. Combined with the ability to select and change data structures for the scene, this is a very effective method to test several algorithms over different structures.

Enable selection of different data structures – As mentioned earlier, selecting the underlying data structure is very important for the research. The ability to select different underlying data structures and change it at run time is one of the key features of the application. When a structure is selected, the application attempts to find the file in which the tree is stored. If the file is found, the application loads the tree. Otherwise, the application calls the tree construction process with the selected parameters. The window also shows the different parameters that can be selected.

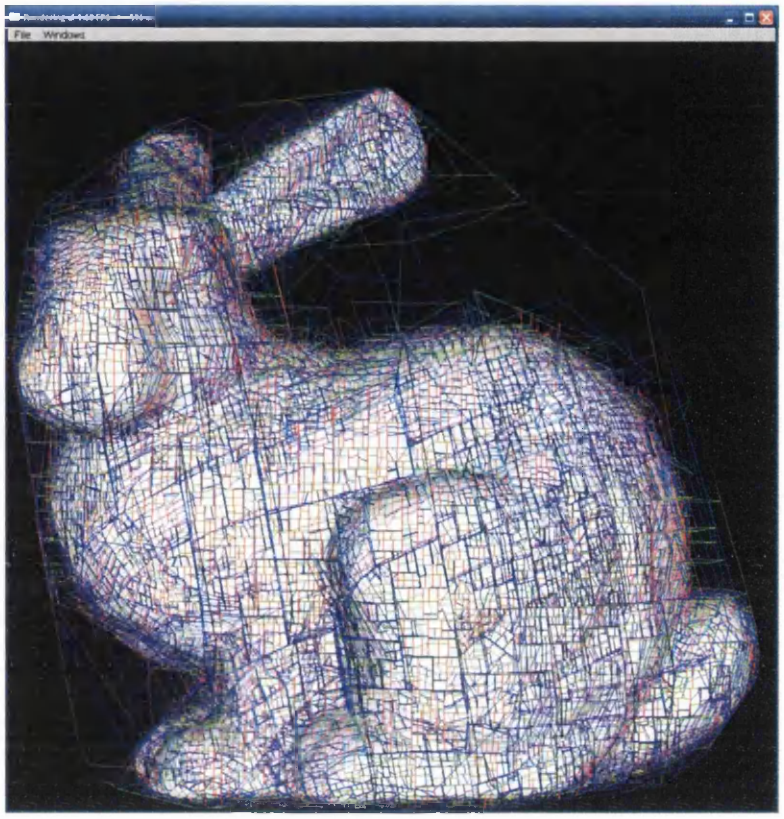


Figure A.8: *RBSP tree* visualised on the *Bunny*.

This also makes it possible to compare different variations of the same structure that are built with different parameters.

Enable comparison between different methods – The user interface with its ability to select several different rendering methods and several different data structures is a very effective tool to compare the different methods implemented with different underlying structures.

As shown, the user interface has been customised so that several important features focusing on research are possible easily and with minimal effort. The application fulfils its goal of being easy and simple to use while at the same time offering several powerful features for researchers. It is fully developed using the FLUID tool that makes it easy to add new elements and to make modifications to existing features. The combination makes the user interface a good fit for the application.

A.4 Summary

As the section shows, the software was implemented to assist the research. The system is developed in a modular manner. Several data structures and algorithms can be used to render a particular scene. This allows easy comparison between the rendering methods. The system also

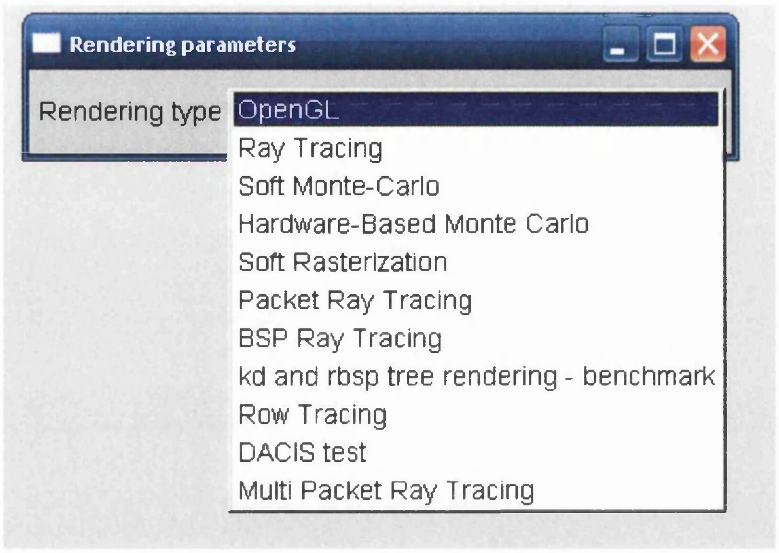


Figure A.9: Selecting a rendering method

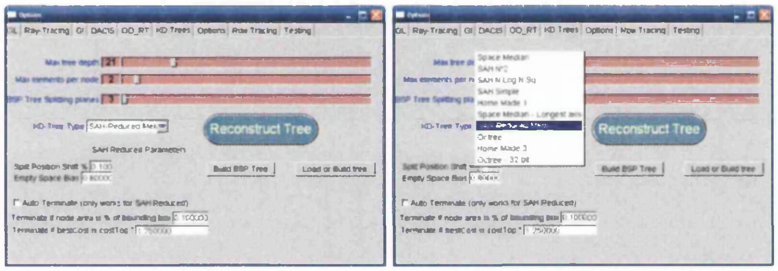

Max tree depth 21
Max elements per node 2
BSP Tree Splitting planes 3
KD-Tree Type SAH Reduced Me
SAH Reduced Parameters
Split Position Shift % 0.100
Empty Space Bias 0.80000
Auto Terminate (only works for SAH Reduced)
Terminate if node area is % of bounding box 0.100000
Terminate if bestCost is costTop * 1.750000

Figure A.10: Selecting a data structure

has a user interface that is simple and provides ways to access the key features with minimal effort. Finally, it is extensible enabling development and use of new rendering methods. All in all, the system architecture has been a major factor in enabling our research.

Appendix B

Optimising Row Tracing with SSE instructions

B.0.1 SIMD Instructions

SIMD – Single Instruction, Multiple Data – instructions, though available earlier on other architectures, are a recent development on X86 processors that enable data level parallelism [SSE09b] [SSE09a]. They enable operating on a larger set of data using a single instruction. Intel's SSE instructions allow performing four floating point operations in parallel using a single instruction. Judicious use of SSE instructions can significantly accelerate the performance. *Row Tracing* provides several instances where the use of such instructions are very beneficial. Intrinsics enable implementing these instructions relatively easily and are supported by both Visual C++ [VCI09] and Intel C++ compilers [Int09]. A few of these instances and the application of SSE to these are detailed below.

Triangle intersection and clipping Intersecting triangles in a leaf node and clipping the intersection segment – as necessary for the leaf node processing – is one of the most frequently performed operations of the algorithm. It is paramount that this part of the algorithm is as efficient as possible. The process, described in detail in Sections 5.6.1 and 5.6.3, is implemented using SIMD instructions to optimise them.

The SSE code below determines whether there is an intersection between the row plane and a triangle and computes the intersection line segment if there is an intersection. The SSE instructions in the code below leads to a performance boost by vectorising the operations a few operations. The operations three dot products, vector additions and vector subtractions are achieved using SSE.

```
//load the three vertices of the triangle into SSE variables
sseP1 = _mm_loadu_ps(verts+scTrs[x]);
sseP2 = _mm_loadu_ps(verts+scTrs[x+1]);
sseP3 = _mm_loadu_ps(verts+scTrs[x+2]);

//find the signed distances between the three points and the Row
    Plane
r1 = _mm_mul_ps(sseP1, sseImagePlane);
r2 = _mm_mul_ps(sseP2, sseImagePlane);
r3 = _mm_mul_ps(sseP3, sseImagePlane);
r4 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(1,0,1,0));
```

```
        r5 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(3,2,3,2));
        r6 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(1,0,1,0));
        r7 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(3,2,3,2));
        r4 = _mm_add_ps(r4, r5);
        r6 = _mm_add_ps(r6, r7);
        r3 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,2,2,0));
        r4 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,3,3,1));
        r7 = _mm_add_ps(r4, r3);

        //check if there is an intersection
        dirs = 7&(_mm_movemask_ps(r7));
        if((dirs==0) || (dirs == 7))
                return false;
        //Find d0/d0-d1 , d1/ d1-d2, d2/ d2-d0
        r1 = _mm_shuffle_ps(r7, r7, _MM_SHUFFLE(0,0,2,1));
        r1 = _mm_sub_ps(r7, r1);
        r1 = _mm_div_ps(r7, r1);
        //Linearly interpolate to get p1, p2, plane intersection point
        r2 = _mm_sub_ps(sseP2, sseP1);
        r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(0,0,0,0));
        r2 = _mm_mul_ps(r2, r3);
        r5 = _mm_add_ps(r2, sseP1);
        //Linearly interpolate to get p2, p3, plane intersection point
        r2 = _mm_sub_ps(sseP3, sseP2);
        r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(1,1,1,1));
        r2 = _mm_mul_ps(r2, r3);
        r4 = _mm_add_ps(r2, sseP2);

        //Select the right two points out of three
        r2 = _mm_sub_ps(sseP1, sseP3);
        r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(2,2,2,2));
        r2 = _mm_mul_ps(r2, r3);
        r3 = _mm_add_ps(r2, sseP3);

        r1 = _mm_cmpgt_ps(r7, ZERO_SSE);
        r2 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(0,0,2,1));
        r2 = _mm_xor_ps(r2, r1);
        r2 = _mm_andnot_ps(r2, MASK_TRUE);

        r1 = _mm_shuffle_ps(r2, r2, _MM_SHUFFLE(0,0,0,0));
        r6 = _mm_shuffle_ps(r2, r2, _MM_SHUFFLE(1,1,1,1));

        r2 = _mm_and_ps(r3, r1);
        r5 = _mm_andnot_ps(r1, r5);
        r5 = _mm_or_ps(r5, r2);

        r2 = _mm_and_ps(r3, r6);
        r4 = _mm_andnot_ps(r6, r4);
        r4 = _mm_or_ps(r4, r2);
        //Select the right two points out of three
```

Listing B.1: Triangle–Row intersection using SSE.

Once the intersection segment is found, it is to be ensured that the segment is fully in front of the near plane. The code below is called only if the node's bounding box lies on both sides of the near plane. SSE instructions optimise the operation by vectorising the calculation of two dot products, clipping with the near plane and finding the point intersection (if there is one) – both of which use the parametric equation of the line.

```
      //handle cases when the intersection segment is partly in front
      //and partly behind the near plane
    if(bbPartlyInFront)
    {
      r9 = _mm_mul_ps(nearPlaneSSE, r4);
      r8 = _mm_mul_ps(nearPlaneSSE, r5);
      r6 = _mm_unpacklo_ps(r9, r8);
      r3 = _mm_unpackhi_ps(r9, r8);
      r9 = _mm_add_ps(r6,r3);
      r3 =  _mm_movehl_ps( r9, r9);
      r9 = _mm_add_ps(r9,r3);
      float d1 = M128_F32(r9)[0];
      float d2 = M128_F32(r9)[1];
      char p1Behind = (((d1 > 0)) != gv_nearPlaneFarPlaneSign);
      char p2Behind = (((d2 > 0)) != gv_nearPlaneFarPlaneSign);
      if(p1Behind && p2Behind)
        return false;
      if(p1Behind)
      {
        d1 = d1/(d1-d2);
        r9 = _mm_sub_ps(r5, r4);
        r1 = _mm_set_ps1(d1);
        r9 = _mm_mul_ps(r9, r1);
        r4 = _mm_add_ps(r4, r9);
      }
      else if(p2Behind)
      {
        d1 = d2/(d2-d1);
        r9 = _mm_sub_ps(r4, r5);
        r1 = _mm_set_ps1(d1);
        r9 = _mm_mul_ps(r9, r1);
        r5 = _mm_add_ps(r5, r9);
      }
    }
```

Listing B.2: SSE version of algorithm that handles cases when the triangle part being rendered is partly in front and partly behind the viewpoint.

Finally, the code below ensures that only parts of segments that are within the node's bounding box are considered. Similar to the above operation, SSE code vectorises finding the intersection to the three entry and exit planes by using the parametric equation of the line.

```
  //clamp the intersection line to the bounding box
  r9 = _mm_sub_ps(r5,r4);
  r8 = _mm_rcp_ps(r9);
  r8 = _mm_min_ps(INFINITY_SSE_V2, r8);
  t1 = _mm_sub_ps(minVertexSSE, r4);
  t1 = _mm_mul_ps(t1, r8);
  t2 = _mm_sub_ps(maxVertexSSE, r4);
  t2 = _mm_mul_ps(t2, r8);
  t3 = _mm_min_ps(t1,t2);
  t2 = _mm_max_ps(t1,t2);
  t1 = _mm_shuffle_ps(t3, t3, _MM_SHUFFLE(0,0,2,1));
  t1 = _mm_max_ps(t1, t3);
  t3 = _mm_shuffle_ps(t1, t1, _MM_SHUFFLE(1,1,1,1));
  t1 = _mm_max_ss(t1,t3);
  t1 = _mm_max_ss(t1,ZERO_SSE);
  t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(0,0,2,1));
  t3 = _mm_min_ps(t3, t2);
  t2 = _mm_shuffle_ps(t3, t3, _MM_SHUFFLE(1,1,1,1));
  t3 = _mm_min_ss(t3,t2);
  t3 = _mm_min_ss(t3,ONE_SSE);
  if(_mm_ucomigt_ss(t1, t3))
    return false;
  r8 = _mm_shuffle_ps(t1,t1,_MM_SHUFFLE(0,0,0,0));
  r7 = _mm_shuffle_ps(t3,t3,_MM_SHUFFLE(0,0,0,0));
  t1 = _mm_mul_ps(r9, r8);
  t1 = _mm_add_ps(r4, t1);
  t2 = _mm_mul_ps(r9, r7);
  t2 = _mm_add_ps(r4, t2);
  //clamp the intersection line to the bounding box

  return true;
```

Listing B.3: SSE version of Clamping the intersection line to the bounding box.

Node Projection Overestimate onto the Row Finding the node projection overestimate is an operation that is undertaken at every traversal step. Hence, it is imperative that this is as optimised as possible. SSE allows the calculation of this overestimate using the code below. The optimisation occurs due to the vectorisation of calculation of four dot products, scalar multiplication to a vector and addition of a constant to all the components of the vector.

```
void CalculateNodeOverestimate()
{
  minVertexSSE = *wMinDiag;
  maxVertexSSE = *wMaxDiag;
  r5 = *xMaxDiag;
  r1 = *xMinDiag;
  ////////////////////////////////////////////////////
  //FourDotProds(        t1, t2, r8, r9,
  //                              m3, m3, m0, m0);
  r7 = _mm_mul_ps(maxVertexSSE, m3);
  r4 = _mm_mul_ps(minVertexSSE, m3);
  r5 = _mm_mul_ps(r5, m0);
  r1 = _mm_mul_ps(r1, m0);

  r8 = _mm_movelh_ps( r7, r4);
  r9 = _mm_movehl_ps(r4, r7);
  r7 = _mm_movelh_ps( r5, r1);
  r4 = _mm_movehl_ps(r1, r5);

  r8 = _mm_add_ps(r8, r9);
  r4 = _mm_add_ps(r4, r7);

  r7 = _mm_shuffle_ps(r8, r4, _MM_SHUFFLE(2,0,2,0));
  r8 = _mm_shuffle_ps(r8, r4, _MM_SHUFFLE(3,1,3,1));

  r8 = _mm_add_ps(r8,r7);
  //FourDotProds - Results are in r8
  //r8 = (m3.w1, m3.w2, m0.x1, m0.x2)
  ///////////////////////////////////////////////////

  r1 = _mm_shuffle_ps(r8, r8, _MM_SHUFFLE(1,1,0,0));
  r7 = _mm_movehl_ps(r8, r8);
  r1 = _mm_rcp_ps(r1);
```

```
    r1 = _mm_mul_ps(r7, r1);
    r1 = _mm_mul_ps(r1, HALF_WIDTH_SSE);
    r8 = _mm_add_ps(r1, HALF_WIDTH_SSE);

    r5 = _mm_movehl_ps(r8, r8);
    r1 = _mm_min_ps(r8, r5);
    r7 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(1,1,1,1));
    r7 = _mm_min_ss(r7, r1);

    r3 = _mm_max_ps(r8, r5);
    r4 = _mm_shuffle_ps(r3, r3, _MM_SHUFFLE(1,1,1,1));
    r5 = _mm_max_ss(r3, r4);
}
```

Listing B.4: Finding the row overestimate using SSE

Rasterising the Last 8 Pixels – With eight pixels, the processing can be easily done using two SSE units to optimise the process. The code below shows the implementation in which each `float` component of an SSE variable corresponds to a pixel. By using two SSE units and two iterations, the triangles for the eight pixels are determined.

```
Rasterise8PixelsSSE()//int minX)
{
  int minX = minXInt-(minXInt&7), maxX = minX+7, i, startX, endX;
  trIntLine = intPoints;
  lineTrs = (__m128 *)(gv_lineTriangles+minX);
  r5 = MINUS_ONE;
  r6 = MINUS_ONE;
  M128_I32(r1)[0] = gv_lineOcclusionMap[(occlMaxDepthStartIndex+minX)>>3];
  r1 = _mm_shuffle_ps(r1,r1,_MM_SHUFFLE(0,0,0,0));
  r8 = _mm_and_ps(r1, MASK_OCCL_LOW);
  r8 = _mm_cmpeq_ps(r8, ZERO_SSE);
  r9 = _mm_and_ps(r1, MASK_OCCL_HIGH);
  r9 = _mm_cmpeq_ps(r9, ZERO_SSE);

  first4Pixels = _mm_set_ps1(minX);
  minXPlus4 = _mm_add_ps(first4Pixels, FOUR);

  first4Pixels = _mm_add_ps(first4Pixels, zeroToThree);
  second4Pixels = _mm_add_ps(minXPlus4, zeroToThree);
  maxXm128 = _mm_set_ps1(maxX);

  for(i=0; i < intPointSize; i++)
  {
    startX = trIntLine->x1;
    startX = MAX_2(startX,minX);
    endX = trIntLine->x2;
    endX = MIN_2(endX, maxX);

    if(startX <= endX)
    {
      startXM128 = _mm_set_ps1(startX);
      endXM128 = _mm_set_ps1(endX);
      bM128 = _mm_set_ps1(trIntLine->b);
      aM128 = _mm_set_ps1(trIntLine->a);
      M128_I32(trM128)[0] = trIntLine->tr;
      trM128 = _mm_shuffle_ps(trM128,trM128, _MM_SHUFFLE(0,0,0,0));
```

```
        ////Rasterise the first 4 pixels
        r7 = _mm_cmple_ps(first4Pixels, endXM128);
        r3 = _mm_cmpge_ps(first4Pixels, startXM128);
        r7 = _mm_and_ps(r7, r3);
        r3 = _mm_cmple_ps(first4Pixels, minXPlus4);
        r3 = _mm_and_ps(r3, r7);
        r3 = _mm_andnot_ps(r3, MASK_TRUE);
        r7 = _mm_mul_ps(aM128, first4Pixels);
        r7 = _mm_add_ps(r7, bM128);
        r7 = _mm_or_ps(r7, r3);
        r1 = _mm_or_ps(r3, trM128);
        r4 = _mm_cmpgt_ps(r7, r5);
        r4 = _mm_and_ps(r4, r8);
        r5 = _mm_max_ps(r7, r5);
        r1 = _mm_and_ps(r4, r1);
        r2 = _mm_andnot_ps(r4, lineTrs[0]);
        lineTrs[0] = _mm_or_ps(r1, r2);

        //Rasterise the second 4 pixels
        r7 = _mm_cmple_ps(second4Pixels, endXM128);
        r7 = _mm_and_ps(r7, _mm_cmpge_ps(second4Pixels, startXM128));
        r3 = _mm_and_ps(r7, _mm_cmple_ps(second4Pixels, maxXm128));
        r3 = _mm_andnot_ps(r3, MASK_TRUE);
        r7 = _mm_mul_ps(aM128, second4Pixels);
        r7 = _mm_add_ps(r7, bM128);
        r7 = _mm_or_ps(r7, r3);
        r3 = _mm_or_ps(r3, trM128);
        r4 = _mm_cmpgt_ps(r7, r6);
        r4 = _mm_and_ps(r4, r9);
        r6 = _mm_max_ps(r7, r6);
        r1 = _mm_and_ps(r4, r3);
        r2 = _mm_andnot_ps(r4, lineTrs[1]);
        lineTrs[1] = _mm_or_ps(r1, r2);
      }
      trIntLine++;

    }
      r5 = _mm_cmpneq_ps(r5, MINUS_ONE);
      r6 = _mm_cmpneq_ps(r6, MINUS_ONE);

      unsigned char shadedFlag= _mm_movemask_ps(r5) | (_mm_movemask_ps(r6) <<
          4);
      UpdateOcclusionMapBy8(minX, shadedFlag);
}
```

Listing B.5: Rasterising the last eight pixels using SSE.

Appendix C

Low level Optimizations

Low level optimizations enable a well designed algorithm to run even faster. Some of the low level optimizations used in the implementation were multi threading the application and the use of data level parallelism through the use of SSE instructions. Multi threading provided a speed-up of around $3.5\times$ - $3.9\times$ for many renderings. SSE provided a speed-up of $2\times$ - $3\times$ the non-SSE method. The use of these two forms of optimization has enabled us to speed up our algorithm to competitive levels.

SSE

The code for a few methods optimized with SSE are given below. In the below code, it is to be noted that $r1, r2, r3, r4, r5, r6, r7, r8, r9$ are SSE variables that are defined as class variables.

Four dot Products with SSE

The method is a part of the method that finds the projection of the X-coordinate of two points onto the image line in the row tracing algorithm. It computes four dot products – $\mathbf{l1.m1}$, $\mathbf{l2.m2}$, $\mathbf{l3.m3}$, $\mathbf{l4.m4}$ – and stores the value in $r8$. Since we do not use a structure of arrays as recommended as the best method to use SSE, we have to use shuffles to horizontally add the value. However, the number of shuffles and horizontal moves is kept to a minimum.

```
FourDotProds(l1, l2, l3, l4, m1, m2, m3, m4)
{
        r7 = _mm_mul_ps(l1, m1);
        r4 = _mm_mul_ps(l2, m2);
        r5 = _mm_mul_ps(l3, m3);
        r1 = _mm_mul_ps(l4, m4);

        r8 = _mm_movelh_ps( r7, r4);
        r9 = _mm_movehl_ps(r4, r7);
        r3 = _mm_movelh_ps( r5, r1);
        r4 = _mm_movehl_ps(r1, r5);

        r8 = _mm_add_ps(r8, r9);
        r4 = _mm_add_ps(r4, r3);
        r3 = _mm_shuffle_ps(r8, r4, _MM_SHUFFLE(2,0,2,0));
        r8 = _mm_shuffle_ps(r8, r4, _MM_SHUFFLE(3,1,3,1));

        r8 = _mm_add_ps(r8,r3);
}
```

If a structure of arrays as suggested for use with SSE were to be used, the data would first have to be reorganized into this layout. Consequently, the dot products could be calculated with a reduced number of instructions.

The layout can be changed to a SSE friendly nature by using the macro already defined as a part of Visual C++. The macro can be given by

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3) {                        \
            __m128 tmp3, tmp2, tmp1, tmp0;                                    \
                                                                              \
            tmp0    = _mm_shuffle_ps((row0), (row1), 0x44);                   \
            tmp2    = _mm_shuffle_ps((row0), (row1), 0xEE);                   \
            tmp1    = _mm_shuffle_ps((row2), (row3), 0x44);                   \
            tmp3    = _mm_shuffle_ps((row2), (row3), 0xEE);                   \
                                                                              \
            (row0) = _mm_shuffle_ps(tmp0, tmp1, 0x88);                        \
            (row1) = _mm_shuffle_ps(tmp0, tmp1, 0xDD);                        \
            (row2) = _mm_shuffle_ps(tmp2, tmp3, 0x88);                        \
            (row3) = _mm_shuffle_ps(tmp2, tmp3, 0xDD);                        \
        }
```

The macro takes in the rows in the normal format and transposes it so that all the x, y and z components are now in a single row each. In this format, four dot products can be achieved with fewer instructions. If **lx**, **ly**, **lz** and **mx**, **my**, **mz** indicate the components of the four vectors arranged in a structure of arrays format, then the four dot products can be achieved very easily as shown below.

```
FourDotProds(lx, ly, lz, mx, my, mz)
{
        r7 = _mm_mul_ps(lx, mx);
        r4 = _mm_mul_ps(ly, my);
        r5 = _mm_mul_ps(lz, mz);

        r8 = _mm_add_ps(r7, r4);
        r4 = _mm_add_ps(r8, r5);
}
```

Three dot Products with SSE

In row tracing, there are also cases when three dot products are necessary. This is when a point's X and Z co-ordinates are to be projected onto the image row. Though, the same method used for four dot products can be used, it is possible to eliminate one multiply instruction when three dot products are calculated. Thus, three dot products are achieved by the below code that computes **l1**.**d**, **l2**.**d**, **l3**.**d**.

```
ThreeDotProds(l1, l2, l3, d)
{
        r1 = _mm_mul_ps(l1, d);
        r2 = _mm_mul_ps(l2, d);
        r3 = _mm_mul_ps(l3, d);

        r4 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(1,0,1,0));
        r5 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(3,2,3,2));
        r6 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(1,0,1,0));
        r7 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(3,2,3,2));

        r4 = _mm_add_ps(r4, r5);
        r6 = _mm_add_ps(r6, r7);

        r3 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,2,2,0));
        r4 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,3,3,1));

        r7 = _mm_add_ps(r4, r3);
}
```

Determining entry and exit planes for RBSP trees with SSE

For RBSP trees, due to the existence of several axes, determining the entry and exit planes can be done in groups of four axes using SSE. SSE thus allows the computation of four entry and exit planes in the same time as one plane when SSE is not used – improving performance significantly. The below code achieves this by considering groups of four axes each.

```
for(i=0, k=0; i < noSplitPlanes; i+=4, k++)
{
 dirRec = _mm_add_ps(_mm_mul_ps(vDirSSEX, planeNormalsSSEAll[k*3]),
 _mm_add_ps(_mm_mul_ps(vDirSSEY, planeNormalsSSEAll[k*3+1]),
 _mm_mul_ps(vDirSSEZ, planeNormalsSSEAll[k*3+2])));

 flag = _mm_cmpeq_ps(dirRec, ZERO);
 flag = _mm_and_ps(flag, FEPSILON_M128);
 dirRec = _mm_add_ps(dirRec, flag);

 templ =_mm_rcp_ps(dirRec);
 dirRec = _mm_sub_ps(_mm_add_ps(templ,templ),_mm_mul_ps(_mm_mul_ps(templ,
     templ),dirRec));

 templ = _mm_mul_ps(tSSEMin[k], dirRec);
 temp2 = _mm_mul_ps(tSSEMax[k], dirRec);

 flag = _mm_cmpgt_ps(dirRec, ZERO);
 tSSE0 = _mm_or_ps(_mm_and_ps(templ, flag), _mm_andnot_ps(flag, temp2));
 tSSE1 = _mm_or_ps(_mm_and_ps(temp2, flag), _mm_andnot_ps(flag, templ));

 tminSSE = _mm_max_ps(tminSSE, tSSE0);
 tmaxSSE = _mm_min_ps(tmaxSSE, tSSE1);

 _mm_store_ps((t+2*i), _mm_unpacklo_ps(tSSE0, tSSE1));
 _mm_store_ps((t+2*i+4), _mm_unpackhi_ps(tSSE0, tSSE1));

 signs = _mm_movemask_ps(flag);
 rayDirs[i] = (signs & 1);
 rayDirs[i+1] = (signs & 2)>>1;
 rayDirs[i+2] = (signs & 4)>>2;
 rayDirs[i+3] = (signs & 8)>>3;
}
```

```
tmin = MAX_2(tminSSE.m128_f32[0], tminSSE.m128_f32[1]);
tmin = MAX_2(tmin, tminSSE.m128_f32[2]);
tmin = MAX_2(tmin, tminSSE.m128_f32[3]);

tmax = MIN_2(tmaxSSE.m128_f32[0], tmaxSSE.m128_f32[1]);
tmax = MIN_2(tmax, tmaxSSE.m128_f32[2]);
tmax = MIN_2(tmax, tmaxSSE.m128_f32[3]);


if(tmax < 0 || tmin > tmax)
 return -1;
```

Row / Plane intersection

For the row tracing algorithm, it is necessary to perform a row / plane intersection at each traversal step. The test involves two dot products followed by a test of the signs. It can be implemented in SSE to achieve speed-up as shown below.

```
ImageLineIntersectsBBSSE()
{
                r1 = _mm_mul_ps(sseImagePlane, minVertexSSE);
                r2 = _mm_mul_ps(sseImagePlane, maxVertexSSE);

                r3 = _mm_unpacklo_ps(r1, r2);
                r4 = _mm_unpackhi_ps(r1, r2);

                r3 = _mm_add_ps(r3,r4);
                r4 = _mm_shuffle_ps(r3,r3, _MM_SHUFFLE(3,2,3,2));
                r3 = _mm_add_ps(r3,r4);

                r3 = _mm_cmpgt_ps(r3, zero);
                r3 = _mm_and_ps(r3, one);
                r4 = _mm_shuffle_ps(r3,r3, _MM_SHUFFLE(0,0,0,1));
                return _mm_comineq_ss(r3,r4);
}
```

Row / Triangle intersection clamping in SSE

Intersecting triangles in a leaf node and clipping the intersection segment – as necessary for the leaf node processing – is one of the most frequently performed operations of the algorithm. It is paramount that this part of the algorithm is as efficient as possible. The process, described in detail in section 5.6.1 and 5.6.3, is implemented using SIMD instructions in order to optimize them.

The below code lists the SSE code to determine whether there is an intersection between the row plane and a triangle and computes the intersection line segment if there is an intersection. The SSE instructions in the below code vectorizes the operation by performing the three dot products and vector additions and subtractions leading to a performance boost.

```
        //load the three vertices of the triangle into SSE variables
        sseP1 = _mm_loadu_ps(verts+scTrs[x]);
        sseP2 = _mm_loadu_ps(verts+scTrs[x+1]);
```

```
    sseP3 = _mm_loadu_ps(verts+scTrs[x+2]);

    //find the signed distances between the three points and the Row
        Plane
    r1 = _mm_mul_ps(sseP1, sseImagePlane);
    r2 = _mm_mul_ps(sseP2, sseImagePlane);
    r3 = _mm_mul_ps(sseP3, sseImagePlane);
    r4 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(1,0,1,0));
    r5 = _mm_shuffle_ps(r1, r2, _MM_SHUFFLE(3,2,3,2));
    r6 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(1,0,1,0));
    r7 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(3,2,3,2));
    r4 = _mm_add_ps(r4, r5);
    r6 = _mm_add_ps(r6, r7);
    r3 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,2,2,0));
    r4 = _mm_shuffle_ps(r4, r6, _MM_SHUFFLE(3,3,3,1));
    r7 = _mm_add_ps(r4, r3);

    //check if there is an intersection
    dirs = 7&(_mm_movemask_ps(r7));
    if((dirs==0) || (dirs == 7))
            return false;
    //Find d0/d0-d1 , d1/ d1-d2, d2/ d2-d0
    r1 = _mm_shuffle_ps(r7, r7, _MM_SHUFFLE(0,0,2,1));
    r1 = _mm_sub_ps(r7, r1);
    r1 = _mm_div_ps(r7, r1);
    //Linearly interpolate to get p1, p2, plane intersection point
    r2 = _mm_sub_ps(sseP2, sseP1);
    r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(0,0,0,0));
    r2 = _mm_mul_ps(r2, r3);
    r5 = _mm_add_ps(r2, sseP1);
    //Linearly interpolate to get p2, p3, plane intersection point
    r2 = _mm_sub_ps(sseP3, sseP2);
    r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(1,1,1,1));
    r2 = _mm_mul_ps(r2, r3);
    r4 = _mm_add_ps(r2, sseP2);

    //Select the right two points out of three
    r2 = _mm_sub_ps(sseP1, sseP3);
    r3 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(2,2,2,2));
    r2 = _mm_mul_ps(r2, r3);
    r3 = _mm_add_ps(r2, sseP3);

    r1 = _mm_cmpgt_ps(r7, ZERO_SSE);
    r2 = _mm_shuffle_ps(r1, r1, _MM_SHUFFLE(0,0,2,1));
    r2 = _mm_xor_ps(r2, r1);
    r2 = _mm_andnot_ps(r2, MASK_TRUE);

    r1 = _mm_shuffle_ps(r2, r2, _MM_SHUFFLE(0,0,0,0));
    r6 = _mm_shuffle_ps(r2, r2, _MM_SHUFFLE(1,1,1,1));

    r2 = _mm_and_ps(r3, r1);
    r5 = _mm_andnot_ps(r1, r5);
    r5 = _mm_or_ps(r5, r2);

    r2 = _mm_and_ps(r3, r6);
    r4 = _mm_andnot_ps(r6, r4);
    r4 = _mm_or_ps(r4, r2);
    //Select the right two points out of three
```

Once the intersection segment is found, it is to be ensured that the segment if fully in front of the near plane. The below code is called only if the node's bounding box lies on both sides of the near plane. SSE instructions optimize the operation by vectorizing the calculation of two dot products, clipping with the near plane and finding the intersection point (if there is one) – both of which use the parametric equation of the line.

```
        //handle cases when the intersection segment is partly in front
        //and partly behind the near plane
        if(bPartlyInFront)
        {
                r9 = _mm_mul_ps(nearPlaneSSE, r4);
                r8 = _mm_mul_ps(nearPlaneSSE, r5);
                r6 = _mm_unpacklo_ps(r9, r8);
                r3 = _mm_unpackhi_ps(r9, r8);
                r9 = _mm_add_ps(r6,r3);
                r3 =  _mm_movehl_ps( r9, r9);
                r9 = _mm_add_ps(r9,r3);
                float d1 = M128_F32(r9)[0];
                float d2 = M128_F32(r9)[1];
                char p1Behind = (((d1 > 0)) != gv_nearPlaneFarPlaneSign);
                char p2Behind = (((d2 > 0)) != gv_nearPlaneFarPlaneSign);
                if(p1Behind && p2Behind)
                        return false;
                if(p1Behind)
                {
                        d1 = d1/(d1-d2);
                        r9 = _mm_sub_ps(r5, r4);
                        r1 = _mm_set_ps1(d1);
                        r9 = _mm_mul_ps(r9, r1);
                        r4 = _mm_add_ps(r4, r9);
                }
                else if(p2Behind)
                {
                        d1 = d2/(d2-d1);
                        r9 = _mm_sub_ps(r4, r5);
                        r1 = _mm_set_ps1(d1);
                        r9 = _mm_mul_ps(r9, r1);
                        r5 = _mm_add_ps(r5, r9);
                }
        }
```

Finally, the below code ensures that only parts of segments that are within the node's bounding box are considered. Similar to the above operation, SSE code vectorizes finding the intersection to the three entry and exit planes by using the parametric equation of the line.

```
        //clamp the intersection line to the bounding box
        r9 = _mm_sub_ps(r5,r4);
        r8 = _mm_rcp_ps(r9);
        r8 = _mm_min_ps(INFINITY_SSE_V2, r8);
        t1 = _mm_sub_ps(minVertexSSE, r4);
        t1 = _mm_mul_ps(t1, r8);
        t2 = _mm_sub_ps(maxVertexSSE, r4);
        t2 = _mm_mul_ps(t2, r8);
        t3 = _mm_min_ps(t1,t2);
        t2 = _mm_max_ps(t1,t2);
        t1 = _mm_shuffle_ps(t3, t3, _MM_SHUFFLE(0,0,2,1));
        t1 = _mm_max_ps(t1, t3);
        t3 = _mm_shuffle_ps(t1, t1, _MM_SHUFFLE(1,1,1,1));
        t1 = _mm_max_ss(t1,t3);
        t1 = _mm_max_ss(t1,ZERO_SSE);
        t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(0,0,2,1));
        t3 = _mm_min_ps(t3, t2);
        t2 = _mm_shuffle_ps(t3, t3, _MM_SHUFFLE(1,1,1,1));
        t3 = _mm_min_ss(t3,t2);
        t3 = _mm_min_ss(t3,ONE_SSE);
        if(_mm_ucomigt_ss(t1, t3))
                return false;
        r8 = _mm_shuffle_ps(t1,t1,_MM_SHUFFLE(0,0,0,0));
        r7 = _mm_shuffle_ps(t3,t3,_MM_SHUFFLE(0,0,0,0));
        t1 = _mm_mul_ps(r9, r8);
        t1 = _mm_add_ps(r4, t1);
        t2 = _mm_mul_ps(r9, r7);
        t2 = _mm_add_ps(r4, t2);
        //clamp the intersection line to the bounding box

        return true;
```

Packet Ray tracing implementation

Packet Ray tracing was implemented with a version of interval arithmetic, as described in chapter 2. The version is implemented without the use of SSE. It uses two boundary rays for each axis to traverse the entire packet. The implementation of the recursive packet traversal method is given below.

```
char RecursiveRayTraversalIntervalSSE(int nodeIndex, float tmini, float tmaxi
    )//, Interval *ti)
{
        packetNodeTraversals++;
        KDTreeNode1 *node = nodeArray+nodeIndex;
        if(IS_LEAF_P(node))
        {
                if(node->params==3)
                        return 0;
                return ProcessLeafNode(nodeIndex);
        }
        unsigned char   axisCur= GET_AXIS_P(node),
                                        sign = signs[axisCur],
                                        axisCurMinIndex = axisCur<<1;

        float bbSp = GET_POS_P(node) * NODE_DIVISION_PRECISION;

        bbSp = bb[axisCurMinIndex] + (bb[axisCurMinIndex+1] -bb[
            axisCurMinIndex])*bbSp;
        float temp = (bbSp - vpF[axisCur]) ;
        float tSpMax = temp*rayDirRecMax[axisCur];
        temp = temp*rayDirRecMin[axisCur];

        float tSpMin;
        if(temp> tSpMax)
        {
                tSpMin = tSpMax;
                tSpMax = temp;
        }
        else
```

```
                tSpMin = temp;


        if(tSpMax > tmini && tSpMax > 0)
        {
                temp = bb[axisCurMinIndex+sign];
                bb[axisCurMinIndex+sign] = bbSp;
                axisCur = RecursiveRayTraversalIntervalSSE(node->leftNode+1-
                    sign, tmini, MIN_2(tmaxi,tSpMax));//, ti);
                bb[axisCurMinIndex+sign] = temp;
                if(axisCur)
                        return axisCur;
        }
        if(tSpMin < tmaxi)
        {
                temp = bb[axisCurMinIndex+1-sign];
                bb[axisCurMinIndex+1-sign] = bbSp;
                axisCur = RecursiveRayTraversalIntervalSSE(node->leftNode+
                    sign, MAX_2(tmini, tSpMin), tmaxi);//, ti);
                bb[axisCurMinIndex+1-sign] = temp;
        }
        return axisCur;
}
```

Multi threading

Another valuable tool valuable in the optimization process is multi threading the application. This was implemented with the pthreads library that allowed implementing the threads in a simple manner. The below code shows how the ray tracing code was multi threaded.

```
CastRays()
{
        int i;
        for(i=0; i < noThreads; i++)
        {
                pthread_create(&t[i], NULL, RowTracingRenderer2MTSub::
                    do_thread, &rt[i]);
        }
        for(i=0; i < noThreads; i++)
        {
                pthread_join(t[i], NULL);
        }
}

static void *RowTracingRenderer2MTSub::do_thread(void* param)
{
        static_cast<RowTracingRenderer2MTSub*>(param)->CastRays();
        return NULL;
}
```

The method creates as many threads as defined and then calls the ray tracing method defined in the *sub* class – RowTracingRenderer2MTSub in this case.

To be able to be called by a pthread, the method has to be a static function that did not depend on the object. The 'do_thread' method is thus defined as a static method that takes in the

'RowTracingRenderer2MTSub' object as a parameter and calls its 'CastRays' method. Objects of RowTracingRenderer2MTSub represent objects allocated to each thread performing the work allocated to that thread.