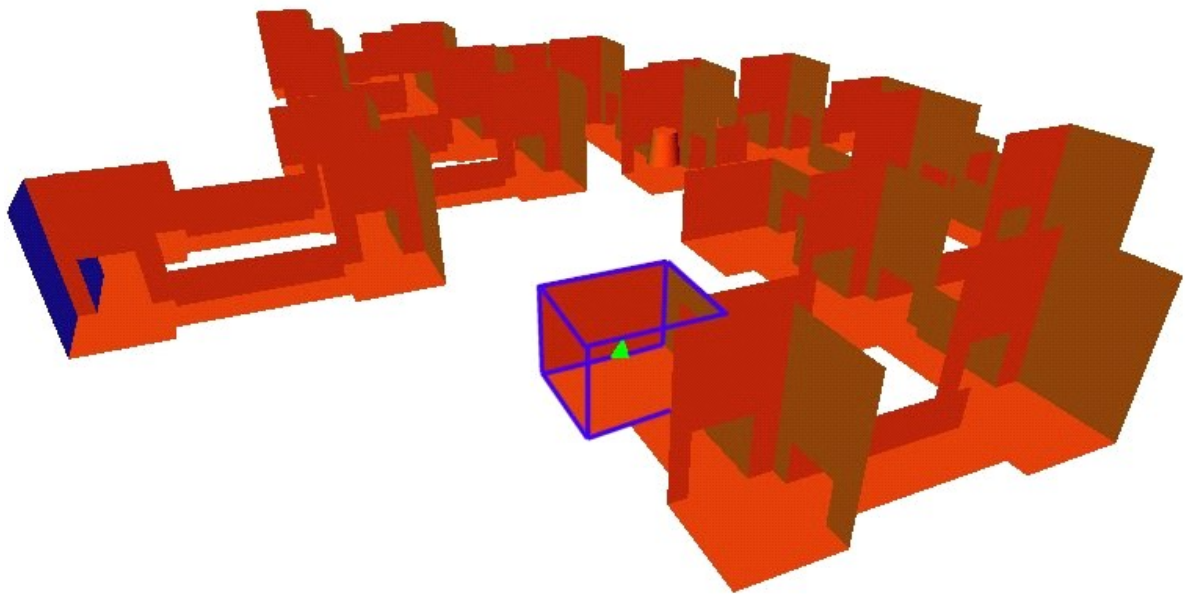


---

# View Space Linking, Solid Node Compression and Binary Space Partitioning for Visibility Determination in 3D walk-throughs

By Joel Anderson



## © Use of Thesis

This copy is the property of Edith Cowan University. However the literary rights of the author must also be respected. If any passage from this thesis is quoted or closely paraphrased in a paper or written work prepared by the user, the source of the passage must be acknowledged in the work. If the user desires to publish a paper or written work containing passages copied or closely paraphrased from this thesis, which passages would in total constitute an infringing copy for the purpose of the Copyright Act, he or she must first obtain the written permission of the author to do so.



---

# **View Space Linking, Solid Node Compression and Binary Space Partitioning for Visibility Determination in 3D walk-throughs**

By Joel Anderson

A thesis for a dissertation to be submitted in fulfilment of the requirements for the degree of:

Bachelor of Computer Science with Honours

Department of Computer Science,  
School of Computer and Information Science (SCIS),  
Edith Cowan University,  
Perth, Western Australia.

Supervisor:

Dr T. J. O'Neill.

20 January 2005

Last Updated 5 March 2006



## **Abstract**

Today's 3D games consumers are expecting more and more quality in their computer games. To enable high quality graphics at interactive rates, games programmers employ a technique known as hidden surface removal (HSR) or polygon culling. HSR is not just applicable to games; it may also be applied to any application that requires quality and interactive rates, including medical, military and building applications. One such commonly used technique for HSR is the binary space partition (BSP) tree, which is used for 3D "walk-throughs", otherwise known as 3D static environments or first person shooters. Recent developments in 3D accelerated hardware technology do not mean that HSR is becoming redundant; in fact, HSR is increasingly becoming more important to the graphics pipeline.

The well established potentially visible sets (PVS) BSP tree algorithm is used as a platform for exploring three enhancement algorithms: View Space Linking, Solid Node Compression and hardware accelerated occlusion. View Space Linking, Solid Node Compression and hardware accelerated occlusion are shown to reducing the amounts of nodes that are traversed in a BSP tree, improving tree traversal efficiency. These algorithms are proven (in cases) to improve overall efficiency.

## Acknowledgements

To begin with, thank you to my supervisor Dr Thomas O'Neill, who continued to have faith in me even though this thesis required more work than intended. Thank you for your patience, encouragement, friendship and humour. I always walked away from our meetings encouraged to put that little bit of extra effort in.

Thanks to my clever German email buddy, Christian Kothe, who explained to me some of the most detailed parts of Quake's hidden surface removal algorithms. I've lost contact with him so, if you're reading this, Christian; I would like to wish you well in your studies.

Thanks to my family who supported me through this process, even when I had to fly over to Melbourne to start work for a games development company two weeks before the due date. Without this support, I may never have completed this thesis.

Last but not certainly not least, I would like to thank you, my readers. I hope you find the algorithms described in this thesis useful and practical. You may also find the appendices useful as much of the details and explanation of techniques investigated were moved into that section to keep the size of the thesis down.

## Table of Contents

Use of thesis	
Title	
Abstract	
Copyright and Access Declaration	
Acknowledgements	
1 Introduction.....	1
1.1 Background to the Study.....	2
1.2 The Significance of the Study.....	5
1.3 Purpose of the Study.....	6
1.4 Research Questions.....	6
1.5 Definition of Terms.....	7
2 Literature Review.....	8
2.1 Graphics Performance Techniques.....	8
2.2 Simular Studies.....	13
3 Quake Engines.....	20
3.1 Quake 1.....	20
3.2 GLQuake and Quake 2.....	22
3.3 Quake 3.....	23
4 BSP tree Implementation.....	25
4.1 General BSP tree.....	25
4.2 BB BSP trees.....	36
4.3 PVS BSP trees .....	46
5 Materials and Methods.....	73
5.1 Target Market.....	73
5.2 Design and Procedure.....	73
5.3 Software Used.....	74
5.4 Data Analysis.....	75
5.5 Limitations.....	76
6 Implementation Specifics.....	78

6.1 Compilation.....	78
6.2 Runtime.....	83
7 View Space Linking.....	86
7.1 Link Generation.....	86
7.2 Rendering VSL.....	88
7.3 Other.....	90
8 Solid Node Compression.....	91
8.1 Tree Compression.....	91
8.2 Tree Traversal.....	96
9 Hardware Occlusion Culling .....	100
9.1 The Queues.....	100
9.2 Frame-to-Frame Coherency.....	102
10 Combining View Space Linking, Solid Node Compression and Hardware Occlusion Culling	103
10.1 Traversal.....	103
10.2 Collision Detection.....	103
10.3 Occlusion.....	103
11 Findings.....	105
11.1 Polygons Rendered.....	106
11.2 Performance.....	109
11.3 Other points of interest.....	113
12 Conclusion - The future.....	114
13 Reference List.....	118
14 Appendix 1 - List of Abbreviations.....	119
15 Appendix 2 - Quake Clone BSP Engines.....	120
16 Appendix 3 - Algorithms.....	122
16.1 Portals.....	122
16.2 Quadtrees/Octrees.....	123
16.3 KD-trees.....	129
16.4 Zero Run Length Encoding.....	132
17 Appendix 4 – Compiler Sample Code (C++).....	134



17.1 Class Properties.....	134
17.2 Polygon Classification.....	135
17.3 Polygon Intersection.....	138
17.4 Computing the Plane.....	138
17.5 Determining if a Polygon is Small.....	141
17.6 Plane Spanning Polygon.....	142
17.7 Split.....	143
17.8 Trim.....	146
17.9 Portal Generation.....	150
17.10 Zero Run Length Encoding.....	156
17.11 Link Generation.....	156
17.12 Mark Leaf.....	157
17.13 PVS Generation.....	158
17.14 Solid Node Compression.....	175
18 Appendix 5 – Engine Sample Code (D).....	179
18.1 Class Properties.....	179
18.2 View Frustum Culling.....	180
18.3 Collision Detection.....	182
18.4 Marked BB BSP tree Traversal.....	183
18.5 Render Linked BSP tree.....	185
18.6 Mark Small Containers.....	186
18.7 Solid Node Compression .....	186
18.8 Occlusion Culling.....	190
19 Appendix 6 - Examples.....	209
19.1 General BSP tree Compilation.....	209
19.2 General BSP tree at Runtime.....	212
19.3 BB BSP tree Compilation.....	218
19.4 Runtime BB BSP tree.....	220
19.5 Solid BSP tree Compilation.....	222
19.6 Portal Creation.....	225

19.7 Anti-penumbra Portal Culling.....	232
20 Appendix 7 – Using the demo.....	234
20.1 The Compiler.....	234
20.2 The Engine.....	246
21 Appendix 8 – Definition of terms.....	250
22 Glossary.....	275

# 1 Introduction

Allow a three-dimensional (3D) level designer a million polygons and they'll want two million. The more polygons that can be displayed on the computer screen, the more realistic the graphics will look. Today's users, particularly computer games players, expect to be able to travel around in large 3D worlds at interactive rates; therefore, quickly determining which polygons are not relevant to a scene, as they will not be rendered, is an important aspect of any 3D engine. Visibility determination or hidden surface removal (HSR) is the key to processing high-quality real-time computer graphics scenes, while still being able to maintain large 3D worlds. There will always be demand for better performance and better quality, no matter how powerful 3D accelerated hardware gets.

Graphics programmers use all kinds of efficient algorithms to discard polygons that cannot be seen. Over the years, many techniques have become less applicable in software applications, because hardware has outgrown them. In the foreseeable future there will always be demand for more detail in 3D graphics. What is needed is a technique that can reduce polygons efficiently while being scaleable to many forms of present, and future, computer hardware. A Binary Space Partition (BSP) tree is a data structure that has been used successfully to improve HSR in indoor environments. Historically, the BSP tree process has evolved to keep pace with modern technology. BSP tree HSR was used in ID™'s (Instinct-Driven software) popular games Doom in 1993 and Quake 1 in 1996. Although the BSP algorithm has evolved, the basic HSR concepts used in the Quake 1 engine can still be seen in many modern games including Quake 2, Quake 3, Return to Castle Wolfenstein, Jedi Knight II, Soldier of Fortune, Half-Life: Counter-Strike, Medal of Honor: Allied Assault, Star Trek: Elite Force and Anachronox.

## 1.1 Background to the Study

Problems of visibility determination, such as occlusion or field of view visibility, have been around ever since the first computer graphics were created. Some of the first forms of visibility determination appeared in word processors that needed some way to remove efficiently characters that were to appear on the screen. So algorithms, known as HSR, were invented to cull away large amounts of hidden objects. However, HSR is not just about culling polygons that are rendered to the screen; it is also about making the application more efficient. Therefore, the correct HSR solution for particular application needs to be chosen, as visibility determination is not an island problem. In some applications, the choice of spatial structure used for HSR can affect the performance of the algorithm. Furthermore, many of the spatial structures used in HSR for rendering are also required for collision detection (CD), constructive solid geometry (CSG) and polygon priority-order (PPO). As James (1999, pp. 24-27) explains, a BSP tree is a multiple-use data structure that enables good HSR for static indoor environments by moving much of the runtime bottleneck to pre-compilation time, although they can be used for dynamic outdoor scenes.

### 1.1.1 Origins of BSP trees

Binary Space Partitioning trees were originally conceived by Schumaker, Brand, Gilliland and Sharp (1969) for the quick ordering of polygons by their depth in the scene, known as priority lists. In discussing Shumaker et al. priority lists, James (1999, p. 8) stated: "Here, the list of objects can be ordered from highest to lowest priority, where the first object could have no other objects take priority over it, and the last could not take priority over the others." Sutherland, Sproull and Schumaker (1974) expanded the idea of priority lists to take into account clusters of priority in a binary tree form. "The tree has separating planes that are stored in the non-leaf nodes, and priority-orders of clusters at the leaves." (James, 1999, p. 10). Consequently, Fuchs, Kedem and Naylor (1979; 1980) developed the fundamental BSP tree principles (cited from James, 1999, p. 13).

### 1.1.2 Competitor Algorithms

Although the BSP tree was primarily being used to order polygons by depth (polygon priority-order or PPO), other applications were soon realized for improving performance such as:

- CD by reducing the amount of objects that need to be tested;
- HSR by quickly determining groups of polygons that are not visible, and preventing them from being sent further down the pipeline, where calculations become more expensive; and
- CSG by quickly determining object intersections.

There are, however, other potential solutions to these problems, namely:

- Z-buffers, beam trees and binary trees provide ways in which to priority-order polygons by distance (see Kmett, 1999a);
- Octrees and KD-trees provide an efficient method for determining CSG (see Butcher, 1999; Freidin, 1999-2000) is needed in solid BSP tree construction, which is the form of BSP used in Quake; and
- Octrees (Nuydens, 2000), KD-trees (Csabai, n.d.) and portal culling (Bikker, n.d.) are used to reduce the amounts of polygons need for processing (Cohen, Chrysanthou, Silva, and Durand, n.d.; Hofmann, 2000; Kmett, 1999a; Loisel, 1996; Sutherland et al., n.d.) and as spatial data structures for HSR and CD.

In most instances, BSP tree methods can be as efficient (if not more so) as these techniques, in their particular applications. Furthermore, hybrids of these techniques combined with the BSP trees have led to performance gains, for example in the Unreal (*Unreal Tournament's website*), Quake (*ID™ Software's website*) and Kage (Nettle, 2001) engines. Eventually, PPO was dropped altogether when the Z-buffer became more practical, as in ID™'s Quake 3 engine; however, the CD, HSR, CSG and PPO applications based on BSP trees are still widely used today.

### 1.1.3 ID™'s HSR Engines and Competitors

The company called "Instinct-Driven software" (ID™) has been a major driving force behind BSP tree technology. ID™ has lead the field in 3D walk-through engine technology ever since "Wolfenstein 3D" (1991). "Upon its release in May 1992, Wolfenstein 3D was an instant sensation and became something of a benchmark for PCs. When Intel wanted to demonstrate the performance of its new Pentium chip to reporters, it showed them a system running Wolfenstein." (Kushner, 2002). In the early 1990s, after several engine re-designs (Abrash, 1997, p. 1184), John Carmack (programmer and one of the three founders of ID™) and John Romero (ID™ programmer) decided to use BSP trees to accelerate "hidden surface removal" for the computer game Doom. Doom's BSP tree was a flat 2D sector-based tree that relied on ray casting for rendering and used hierarchal bounding box (BB) culling. This BSP tree algorithm for HSR became popular among developers due to Doom's popularity.

Following the success of Doom, other groundbreaking titles were produced, including Doom 2, Quake 1, Quake 2, Quake 3 and Return to Castle Wolfenstein. ID™'s game engines (see *ID Software's website*) have been licensed to other companies, producing games such as: Jedi Knight II, Soldier of Fortune, Half-Life, Half-Life: Counter-Strike, Medal of Honor: Allied Assault, Star Trek: Elite Force and Anachronox. All ID™'s 3D games (to date) that followed DOOM used BSP trees and each new game engine produced by ID™ was an improvement over its predecessors. The planned DOOM 3 will no doubt be as ground breaking as its predecessors.

Other companies/groups that have produced engines with HSR that are just as powerful as the Quake engines, are as follows:

- Unreal engines (Epic Games);
- KAGE engine (fluid studios);
- LithTech engine (Monolith Productions);
- Torque engine (garage games);
- Serious engine (CroTeam); and
- X-Ray engine (stalker).

However, these engines:

- do not use the BSP tree HSR algorithm;
- do not share enough information about the engines design;
- are outdoor focused as well as indoor;
- provide efficiency gains mainly at the expense of detail reduction; and
- are simply clones of the Quake's standard HSR architecture.

Therefore, these engines are not investigated in any detail.

### 1.1.4 Rationalee for improvement

It is evident that HSR will be a problem that is to be faced by programmers well into the future, and that it is unlikely to be solved completely by improvements in 3D accelerated hardware. "One of the great myths concerning computers is that one day we will have enough processing power." (Moller and Haines, 1999, p. 191). Well-designed HSR code can mean the difference between success or failure for a computer games company. Moreover, HSR algorithms are not as efficient as they could be. As stated by Michael (Abrash, 1996), "It would be nice to be able to chuck all the polygons at a rasterizer that was so fast that we didn't have to think any further. Problem: no such rasterizer. Bigger problem: level designers would just use more polygons." As Bikker (n.d.) suggests, many algorithms have been invented to deal with the HSR problem, however, none are perfect, and all appear to take up a significant portion of the CPU's time or do not remove as many objects as is most efficient.

While there has been much research conducted into enhancing the performance of the BSP algorithm, there is room for improvement. Furthermore, as technology develops some approaches become more viable, for instance:

- Recently, consumer 3D hardware has begun to support hardware occlusion; while not particularly fast at present, the technology is becoming cheaper and being provided as standard in the most popular 3D consumer cards; and
- memory (RAM) is and becoming cheaper, and therefore, more resources become available for pre-calculations.

## 1.2 The Significance of the Study

In some cases (Quake 3, Unreal), the BSP tree HSR algorithm can be a bottleneck taking up more than half of a frame's available time to determine the visible polygons. By freeing up time in the BSP tree HSR algorithm, there should be more time to improve the accuracy for tasks such as:

- artificial intelligence;
- sound;
- dynamics;
- image quality; and
- interactive display rates.

These improvements mean a higher quality, more realistic product for the end-user and perhaps a better selling product.

## 1.3 Purpose of the Study

The purpose of this study is to determine whether combining View Space Linking (VSL), Solid Node Compression (SNC) and hardware-accelerated occlusion culling (HOC) can improve HSR and the Potentially Visible Sets (PVS) BSP tree algorithm. Note the following criteria:

- VSL uses links in leaf nodes to short-circuit the BSP tree traversal;

- SNC can not only reduce the memory footprint of the tree, but also speed up traversals and collision detection; and
- HOC is a new technology that has recently been implemented in consumer 3D cards that are able to ask the 3D hardware whether an object is visible or not. HOC culling works best when culling large groups of data (bounding boxes). Although the 3D graphics card is fast at rejecting polygons, it suffers from lag because it has to travel through the entire rendering pipeline. In many cases, the results from one HOC test can span several frames. Therefore, the algorithm needs to perform other activities while the HOC is being processed and, also, needs to be able to make use of the HOC results from previous frames, for HOC to be of any benefit.

The VSL, SVC and HOC improvements are aimed mainly at the performance side of the BSP tree HSR algorithm, rather reducing the polygon count more than the present-day algorithm already does.

It is hypothesised that:

- VSL takes advantage of frame-to-frame coherency because a camera in one frame is likely to travel through portals that are in the immediate location;
- SNC reduces the size of the tree by removing nodes with one child and by moving solid nodes, therefore improving memory and performance efficiency;
- the BSP tree algorithm can be adapted to use HOC efficiently; and
- all three concepts can be combined to form a hybrid advantage.

## 1.4 Research Questions

The major research question is:

“What is the effect of using VSL, SNC and HOC with Binary Space Partitioning, that make use of potentially visible sets, on memory and CPU performance, in contrast to competitor algorithms?”

Sub questions are:

- “What is the affect of scene density (detail), map size, uniformity, scalability, dynamic objects, detailed brushes and priority-ordering on the performance of the algorithm when compared to competitor BSP PVS algorithms?”
- “How does the algorithm affect overall efficiency such as in frames per second, algorithmic efficiency, collision detection, compile time, load time and runtime when compared to competitor BSP PVS algorithms?”
- “What benefits are achieved by using the proposed algorithm as opposed to competitor HSR algorithms?”

## 1.5 Definition of Terms

It is assumed that the reader is familiar with general graphics principles. A brief discussion of terms, data types, algorithms and techniques used can be found in the Glossary and a detailed discussion about can be found in “Appendix 8 – Definition of terms”.



## 2 Literature Review

Without analysing the work of others, there would be little hope of developing something better. Therefore, the present state of techniques used to improve 3D rendering performance is investigated. “The more approaches you try, the larger your toolkit and the broader your understanding will be when you tackle your next project.” (Abrash, 1997, p. 1280). No one technique is best for all situations; therefore, the more techniques that are known, the better the chance of picking a good technique for the particular application.

### 2.1 Graphics Performance Techniques

Building a 3D engine is no easy feat says Abrash (2000), one of the developers of the Quake (1 and 2) engines. Therefore, it is important to investigate what 3D engines use today:

- backface culling;
- view frustum culling;
- spatial subdivision (HSR) techniques (other than BSP trees); and
- other related techniques.

#### 2.1.1 Backface culling

Backface culling is reasonably efficient; however the algorithm can be improved. The standard back-facing method involves testing every polygon. Most consumer 3D hardware provide backface culling because polygons are processed one at a time and do not keep any record of previously processed polygons, with the exception of vertex arrays and display lists. However, calculation can be made more efficient in software, using such things as hash structures or hierarchical trees known as “clustered culling” (Moller and Haines, 1999, pp. 193). Clustered culling uses approximations to group object normals together, therefore making it possible to remove several polygons with less processing.

#### Normal Masks

Hansong Zhang and Kenneth Hoff discuss an improved version of “Clustered Culling” method using a “Normal Mask”. This technique only requires one AND operation per polygon, which is a vast improvement over three multiplications and six additions per each backface polygon cull. However, there are still some pre-processing and memory overheads to consider. For further information on this technique the reader is directed to the paper “Fast Backface Culling Using Normal Masks” (Zhang and Hoff, 1997).

## The hierarchical (tree) structure

The hierarchical structure method involves dividing the polygons into groups and recursively subdividing those groups. For a binary tree the polygons are split into two even (as possible) groups by polygon normals, recursively until a predetermined cut-off point is reached. This hierarchical structure maintains some means to represent the bounds of the normals in each grouping. The parent nodes' normal boundaries encompass their child nodes normal boundaries.

During runtime, the structure is traversed by determining where the camera lies in the bounds of the (sub)node clusters contents. If the camera is between node boundaries then both sides are traversed, otherwise the node that is facing the viewer (if any) is traversed. If the current node direction bounds are all completely within the node, then there's no need to continue testing; and, all sub-nodes are guaranteed to be facing the camera. For further information of hierarchical back-face clusters, see the work of Kumar et al's (1996) on "Hierarchical Back-Face Computation".

## The Problems with Clustered Culling

It is essential to remember that clustered culling will on average remove the same (or less) polygons as per polygon back-face culling, which is on average only 50%. Polygons can still be outside the camera's field of view (FOV) or hidden by some other object that is in front and is considered visible by the back-facing algorithm.

Most "clustered culling" techniques require extra computations when a polygon's normal changes. If the majority of objects remain static (or move infrequently), this cost is reasonably low. However, if the majority of objects move every frame, then processing time will exceed that of the linear methods of backface culling.

It is important to differentiate BSP trees from the hierarchical culling, because BSP trees are not ordered by normal boundaries. However, for BSP trees, the backfacing face is determined at each node, therefore backface culling can be performed with no more cost. Furthermore, some clustered culling algorithms can potentially speed up the BSP algorithm.

### 2.1.2 View frustum culling

View frustum culling (VFC) is an important aspect of the HSR algorithm since it can be used many times per frame in a HSR algorithm, and can be a bottleneck. The standard view frustum culling technique (Morley, 2000) is to cull away the object by testing each vertex against the 6 planes in the view frustum. If an object is completely outside one plane then it is rejected, otherwise it may be considered to be within the view frustum. In situations when an object is outside but not rejected because it is crossing two planes, all the view frustum sides need to be classified by the planes that make up the object to determine if that object is actually outside. However, in many cases, for the sake of efficiency and at the cost of accuracy that last test can be dropped.

If the object is treated as a BB, then several enhancements can be made to increase the efficiency of view frustum culling (K. Hoff, 1996a, 1996b; Hoff, 1997; Moller and Haines, 1999, pp. 330-335) by testing fewer vertices and taking advantage of early rejection strategies. Assarsson & Moller (1999, p. 334; 2000) describe a technique where the frustum is broken up into eight “octants” and the object is tested against the 3 planes of the octant in which it is located. Furthermore, they take advantage of frame-to-frame coherency by realizing that an object in one frame will probably be within the same octant (or nearby) in the next frame.

If a volume hierarchy (such as, a tree of bounding boxes) is used, then the VFC algorithm can be made more efficient by rejecting many VFC tests in clusters. As the BB tree is traversed, if a BB is found to be completely inside a plane of the view frustum, then that plane no longer needs to be tested against that BB or its subsequent child BBs. Therefore, if a BB is completely outside the view frustum, then its children are not within the frustum also; and, if the object is completely within the frustum, then all its children are within (Abrash, 1996; 1997, p. 1280; Assarsson and Moller, 1999, 2000; *ID™ Software's website*).

### **2.1.3 Spatial subdivision (HSR) techniques**

Octrees/quadtrees, KD-trees and portal culling are just some of the common spatial subdivision structures used in today's HSR algorithms. As with most algorithms, there is no single technique perfect for every situation; BSP trees are no exception. Often the best approach is to use solutions from many algorithms creating a hybrid for the specific problem, such as Kmett's hybrid visibility algorithm (2001) which uses a BSP tree and an octree to achieve efficient collision detection with reasonable world sizes.

#### **Portal culling**

Portals can be used to determine a PVS (Teller, 1992a, 1992b; Teller and Séquin, 1991) or at run time (Bikker, n.d.) to perform HSR. When finding potentially visible polygons (sectors), the portal culling algorithm determines which portals (and therefore sectors) are visible through the sector where the camera is located. If a portal is not visible, then all its child portals are also not visible. Hence, portals can be used to clip out polygons in a sector that aren't visible. Bikker (n.d.) also discusses a portal technique called “active edges” where only edges that cause “concavity” are used as VF clippers, which can be used to improve efficiency.

## Octrees/Quadtrees and KD-trees

Occlusion culling can be performed on an octree or a KD-tree using a Z-pyramid (Moller and Haines, 1999, pp. 206-207), which is a list of Z-buffers, where the highest level starts at a low resolution (that is, 1x1) and the lower levels sequentially increase in resolution until they reach the screens resolution (which is essentially a standard Z-buffer). It is the responsibility of tree nodes (when they render) to record their depth value to the Z-pyramid. If the depth value is more distant (smaller) than the one on the Z-pyramid at that level, then the node is occluded. Otherwise, that node is recursively tested at each level of the Z-pyramid, until the lowest level is reached (the Z-buffer), or it is found occluded. If the Z-buffer is reached then a new value is written to the Z-pyramid. On writing a new value to the Z-pyramid, higher layers must also be updated with the most distant value (smallest) of the lower levels and writing stops when no values on that particular level change.

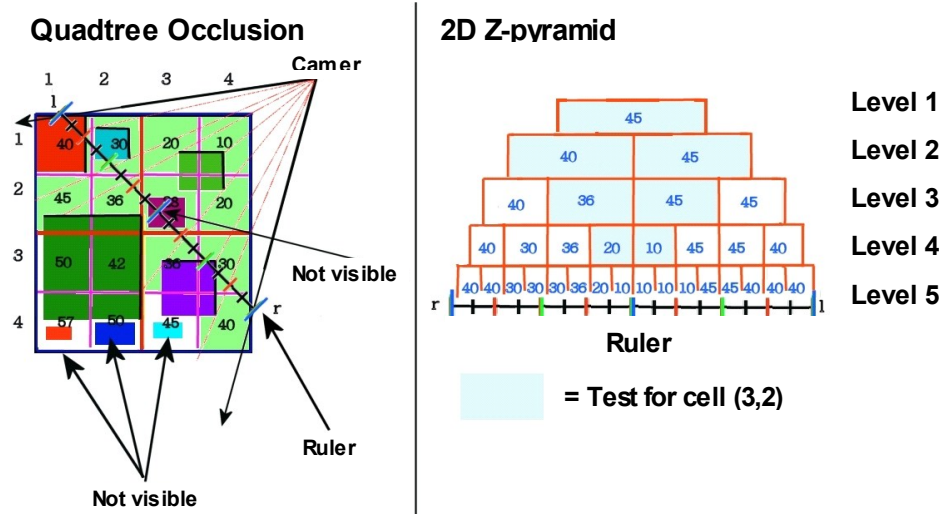


Figure 1. 2D Z-pyramid

Figure 1 demonstrates a 2D Z-Pyramid for a particular Quadtree scene. The ruler is used to determine incrementally what is in the Z-pyramid. The shaded area (back lines) shows the cells (edges) that are visible to the camera. The thin lines on the FOV are guidelines to indicate what the camera can see. Although cell (3,2) is visible, the object inside (3, 2) has potential to be removed in level 4 of the Z-Pyramid (as the light shaded areas in the Z-pyramid shows) because both 20 and 10 are closer than 28. Cell (1, 4) with 57 and (2, 4) with 50 will be discarded at level 1 because these cells are further than 45. Cell (3,4) will be discard at level 4 because 45 is further than 30.

### 2.1.4 Related but Beyond Scope Techniques

There are many other areas of a 3D rendering engine that can benefit from optimisation and special effects which deserve mention, including:

- state sorting (see Shreiner, 2001) used to minimise state changes, often implemented in a tree form;

- vertex lists (see Neider, Davis, and Woo, 1993) used to place dynamic objects in memory that is faster to access by the video card;
- display lists (see Neider et al., 1993) used to pre-compile 3D data into a format that is more efficient for the video card;
- simplification/quality reduction (see Bourke, 1997; Garland, 1999; Shaffer and Garland, 2001) used to reduce scene complexity when performance or distance reaches a certain threshold; and
- the latest hardware developments used to improve performance and enhance 3D graphics: Nvidia (Nvidia's Developer Relations website), ATI (ATI™'s Developer Page website), OpenGL (*OpenGL®*, 2003) and DirectX (*MSDN DirectX page*, 2003) provide extensive papers and examples on their new technologies including:
  - extensions (*OpenGL® Extension Registry*, 2003) provided by 3D hardware companies such as Nvidia and ATI to allow programmers to experiment with the latest 3D technology developments;
  - CG Language specifications (*Nvidia's Developer Relations website*), Render Monkey (ATI™'s Developer Page website), *OpenGL® 2.0* (*OpenGL® 2.0 website*, 2002) and DirectX (*MSDN DirectX page*, 2003) all provide high level vertex and pixel shader programming languages, allowing programmers to write efficient 3D programs that run inside the 3D accelerated hardware, primarily used for special effects;
  - performance documentation (*ATI™'s Developer Page website*, n.d.; *Nvidia's Developer Relations website*, n.d.; *OpenGL®*, 2004) which enables algorithms to make optimal use of the 3D hardware; and
  - demonstrations, examples and source of how to use the 3D accelerated hardware.

Although many of these techniques may be applied and optimised in the solution presented, they are not being covered in detail because they are beyond the scope of the spatial structures being investigated.

## 2.2 Simular Studies

This section contains an overview of recent studies (Hillesland, Salomon, Lastra, and Manocha, n.d.; James, 1996, 1999; James and Day, 1997, 1998; Teller, 1992a, 1992b; Teller and Séquin, 1991) that are similar to this thesis. There is no single HSR algorithm that can cater for all applications. Therefore, the scope of this discussion about each of the studies will be on works that:

- are based on HSR using BSP trees;
- deal with 3D indoor static environments, which support dynamic self-contained entities;
- result in a performance gain without lose of image quality; and
- use some form of hardware-accelerated occlusion culling.

## 2.2.1 Teller

The concept of PVS is introduced by Teller et al. (1991) and Teller (1992a; 1992b) to reduce the cost of hidden surface removal at runtime. Naturally, a PVS can not be stored for every viewpoint in the world; consequently, Teller uses an approximated PVS computed at each cell (room), taking advantage of spatial similarities that exist within indoor environments (James, 1999, pp. 105-106). The PVS does not take into account view frustum culling because any viewpoint in a particular cell is represented by the same PVS. So, to reduce the potentially visible polygon set for a particular viewpoint even more, the view frustum culling operation is performed at runtime.

Teller (1992b) explains how to compute the PVS using a BSP tree to determine the sectors for a particular world. The polygons in the tree are only subdivided to a particular level with large centrally located polygons taking the highest precedence as partitioners. The resulting leaves in the BSP tree become sectors. A portal between any two sectors is computed by clipping planes by the BSP tree. Once all portals have been determined indirect “portal sequences” can be efficiently found. At this stage the process is to find out what is visible from each sector (leaf) in the tree. Since BSP trees are convex, it is now possible to know which way (in succession) rooms connect to each other. Of course just because rooms are connected does not mean that one room is visible through another; therefore, a sightline needs to be computed.

An approach Teller (1992b) describes is to distribute viewing points throughout the sector and cull anything that cannot be seen from all the viewing points. The problem with this approach is that it is difficult to generate a small set of viewing points that will take into account every possible view position for the camera within that sector. For example, one possible solution to this problem is to use only vertices on each of the portal planes, but this still does not guarantee to find every polygon that is potentially visible. Therefore, Teller uses an anti-penumbra, which provides a way to make sure all polygons that can be seen from a certain sector will always be visible.

The anti-penumbra is used to determine if a sightline exists between sectors. However, the anti-penumbra only provides an approximation, so polygons that will never be seen may still be included in the PVS. This small amount of inaccuracy is normally considered insignificant in comparison to the large amount of definitely hidden polygons the anti-penumbra technique is able to remove. Teller (1992b) shows how his revolutionary approach (James, 1999, p. 105) is able to remove (on average) more than 99% (Teller, 1992b, p. 155) of objects for large world problems, in a reasonable time frame. Furthermore, Teller indicates that his method performs relatively better as the numbers of polygons increase.

## 2.2.2 James

James (1999) introduces several new techniques to improve run time performance for BSP trees. These techniques include conflict neutralization, web clustering and Hidden Face Determination (HFD) trees.

Conflict neutralization deals with reducing the amount of polygon splits that occur during the compilation stage of the BSP tree. The fundamental idea is that some polygons “protect” other polygons from being split, by causing the offending polygon(s) to be placed on the opposite side of the “victim” polygon’s sub-tree. However, if these protectors are placed into the other side of the sub-tree (from the “victim” polygon) by another polygon, these “protector” polygons can be prevented from protecting that particular “victim” polygon. These blockers of protector polygons are known as “damaging polygons” and also include polygons that cause splitting.

The James scheme builds a “neutralization tree” which is used to determine the value of a candidate-partitioning plane, by summing up the weighted values of protectors and damaging polygons. The process is applied recursively to each node in the tree, at varying depths. A candidate polygon is tested against every other polygon such that if it:

- causes a blocking of a protection polygon (on odd depths) or a polygon to split, a weight of negative one is given; and
- protects a particular polygon from being split, a weight of positive one-half is given which occurs on even depths.

Weighting sums at each depth are normalized, so that greater depths don’t constitute higher weightings due to the likelihood of increased conflicts at those depths. These weighting values are added to the root node (candidate node) once that depth (for that particular node) has been computed. The highest weight sum (for all candidate nodes) determines the node that is picked for that sub-tree. Neutralization can be performed at every node in the tree at every level in the tree; however, James found that a depth of three was the most practical, with smaller depths at larger sub-trees.

James’ web clustering is a form of spatial data structure that tries to avoid partitioning by forming a non-reflex angled “tri-junction” before the partition. In effect the partition branches into two partitions, which avoids causing a split. Tri-junctions can occur recursively as long as they do not split each other, which could cause an infinite cycle. Web clusters use auxiliary binary partitions, which means that the partitions can occur anywhere and do not need to be part of a polygon’s plane.

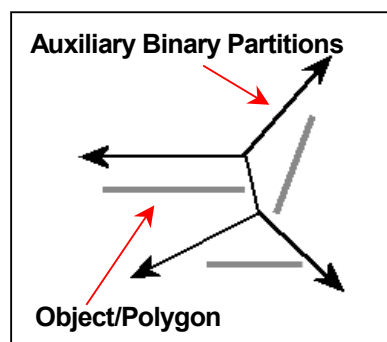


Figure 2. Web Clustering

shows a 2D example of a web-clustered world. The arrows represent the partition planes used to divide the grey polygons. Web clusters cannot always avoid splits but they can help in reducing them.

A HFD tree is a form of tree used to reject polygons that are definitely hidden from view. The HFD tree is an enhancement of the priority face determination (PFD) tree for hidden surface removal which was also developed in James and Day (1999; 1997).

The PFD tree is based on the BSP tree however, by sacrificing pre-processing time and memory. The PFD tree only requires a single-path traversal to determine a list of depth-ordered polygons. James (1999, p. 45) makes the observation that if the viewer is behind one or more polygon plane(s), within some convex hull, then other polygon orders and also the face direction (such as, back facing or front facing) should also be known. The known polygons are stored in the PFD tree at the first node where they are detected and stored in that order. For example in Figure 3 from either V1 or V2 if B is known, in a PFD tree C and E would also be known (to be behind B, for that view area); however in the example BSP tree, C would need to be re-tested.

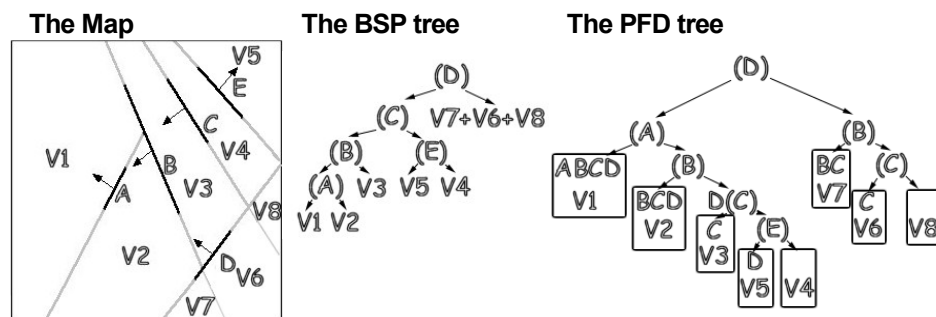


Figure 3. Polygon Dependencies

shows a 2D map that has been converted into a BSP tree and a PFD tree. The arrows on the map represent the normals of the polygons. The arrows on the trees represent pointers from the parent node to the children. In this particular case the BSP tree results in less view areas (prefixed by V) than the PFD tree. Every node (indicated by brackets) in the PFD tree can have a list of visible polygons (letters not within brackets) from the view area. Note that the squares in the PFD tree are simply used to show that each leaf of the tree is a viewing area (V), which may contain a list of ordered polygons.



As Figure 3 shows, the PFD tree's structure is similar to that of the BSP tree's structure but not the same. Each right (left) sub-tree contains all the polygons that are facing the viewer (not facing the viewer) from the convex hull that the ancestral parent nodes form. Polygons with orders can that be determined, from a particular convex hull (at some node) are stored as a list at that node, in the correct depth order. Polygons that are determined back facing at a particular convex hull are discarded at that node. Note that in a PFD tree, there may be many nodes representing the same polygon and many lists with the same polygons. Repetition means that the tree is always much larger than a general BSP tree with a size and processing time of  $O\left(N\left(1.5^{\log_{4/3}(n-1)}\right)\right)$  on average (James, 1999, pp. 59-60), where  $N$  represents the number of original polygons and  $n$  represents the number of polygons after splitting. Note that this equation does not allow for the size of polygon lists stored at each node, which takes up the majority of the tree's memory.

Traversing the tree is a simple matter of going down nodes that are facing the viewer, rendering the ordered list of polygons. Traversal stops at the first leaf node found in the tree. The average amount of nodes that need to be traversed is  $O(\log_{4/3}(n-1))$  which is much better than BSP trees  $O(n)$ . However, for large scenes, the PFD tree's memory requirements may exceed that of the system's requirements, causing many page faults in virtual memory and resulting in a slower algorithm.

The HFD tree adds the concept of area-to-area visibility to the PFD tree. James observes that convex hulls can act as occluders for polygons at each node in the tree. The HFD tree makes for a smaller PFD tree at the cost of increased pre-processing time. In fact if only tree nodes are counted, in some non-general cases the HFD tree can actually be smaller than the general BSP tree for the same map. However, the memory footprint of the PFD tree is always much larger than the BSP due to the lists of polygons that are stored at the HFD tree's nodes.

In the HFD compilation stage, when a polygon is to be added to a particular node, it is first checked to see if it is visible from the convex hull the ancestral parent polygons form; if not visible then it is not placed in that sub-tree. James also investigates several approaches to area-to-area visibility determination for the HFD tree in the compilation stage, observing that the visibility algorithm needs to be reasonably efficient due to the fact that it is the major bottleneck of the algorithm. Traversal of the tree doesn't change from the PFD algorithm; however, in most cases there will probably be fewer polygons to traverse and a smaller memory footprint. James indicates that the HFD algorithm would best suit cases where memory and pre-processing times are not a big issue, but runtime performance is.

### 2.2.3 Hillesland et al.

Hillesland, Salomon, Latra & Manocha (n.d.) describes a technique that uses the NV\_OCCLUSION\_QUERY OpenGL extension provided by Nvidia (*Nvidia's Developer Relations website*, n.d.) to gain a significant performance gain. The algorithm divides the model up into uniform and nested-grids (similar to an octree) made up of cubes. Since it can take a while for the results of the occlusion query to come back from the hardware accelerated 3D video card (or device), each slab in the grid is sent for testing at once using the NV\_OCCLUSION\_QUERY. Once a slab is known to be visible (at least in part), that slab is sub-divided again (recursively to some threshold) and re-tested. Polygons (triangles) that are within a cube are re-tested. Each cube has two lists of polygons containing:

- polygons that are completely inside that cube; and
- indexes to a global list of polygons that don't fit completely within that particular cube.

Furthermore, these polygons are marked (by a frame-counter) to make sure that polygons aren't rendered twice in the same frame.

Hillesland, Salomon, Lastra and Manocha (n.d.) indicates BSP trees, bounding box hierarchy and octrees (p. 3) as a possible alternative to the uniform and nested grid scheme. However, they dismiss these spatial structures as being too complex in terms of traversal order and intelligent construction (Hillesland et al., n.d., p. 3).

## 3 Quake Engines

ID™ is a leader in the field of HSR; therefore, it is important to examine how their engines work in more detail. It is important to have an understanding of these engines because this work is derived from, and compared to, these engines.

### 3.1 Quake 1

Quake 1 was the first in the series of games developed by Carmack that used the potentially visible sets (PVS) technique alongside a solid BSP tree (Hammersley, n.d.) for HSR. It was based on Teller et al. (1991) and Teller's (1992a; 1992b) work on visibility pre-processing. A solid BSP tree uses solid objects with each convex sector being completely sealed. In Quake, static polygons are maintained in nodes and dynamic objects are stored in the leaves. A Z-buffer is then used to catch any polygons that are missed, and to render dynamic objects that appear correctly depth ordered on screen. The entire algorithm can take up to  $O(n \log(n))$  time to determine potentially visible polygons; where  $n$  is the number of polygons included in the polygon splitting.

#### 3.1.1 Implementation

Each flag in the PVS represents a set of polygons that is currently visible for a particular convex hull. In Quake, each flag in the PVS also represents a cluster of leaves in the tree (McGuire, 2002). The PVS are stored at each empty leaf in the tree using zero-length-compression to conserve memory. The compression enables PVS is a efficient way of quickly reducing the amounts of potentially visible polygons (Abrash, 1997, pp. 1188-1189; Moller and Haines, 1999, pp. 200-201), while still maintaining reasonable memory requirements. Furthermore, a hierarchical bounding box is used in the tree to make frustum culling more efficient. The HSR steps for the Quake BSP algorithm are as follows:

The first step in Quake's rendering process is to find the PVS for the current camera position. If the camera has not changed position then there is no reason to do this search. The tree traverses down the branches containing nodes that are orientated towards the viewer. When a leaf is found the algorithm stops, for this leaf contains the convex hull the camera exists within. This step takes a maximum of  $O(\log(n))$  node visits and can be done non-recursively (without using a stack or list data structures); where  $n$  is the number of polygons. If the camera has changed but the PVS (for the leaf the camera is in) has not, then there is no need to do step 2, because the PVS tree data from the last frame can be used.

The next step is to decompress that leaf's PVS. Each potentially visible leaf (and the entire connecting parent branch) contained in the PVS is marked as visible in the tree. Nodes are marked using a frame counter (stored at each node) as opposed to a flag to avoid resetting all the nodes in the entire tree.

The final step is to cull all the remaining polygons by the view frustum. This step needs to be performed even if the camera simply rotates around because any movement is likely to change what is visible in the view frustum. To find visible polygons, all the nodes marked visible in the tree are traversed from the root node. Hierarchical bounding volumes (cubes) are used to cull polygons efficiently by the view frustum. Each node in the tree has a BB that is just large enough to contain all of its child polygons and itself. If a node's BB is found to be completely on one side of a view frustum's plane, then all that node's children are also on that side. When a BB is completely found within the frustum, all children are also within the frustum so there is no need for more BB tests. If a BB is completely outside the view frustum, then all of the node's children can be rejected in a single test. Using this technique means that the majority of polygons are removed in large groups, but some still need to be removed individually.

The time to process a BB BSP tree is depended upon how many polygons are actually visible. It has a maximum time of  $O(n)$  node visits at runtime; where  $n$  is the number of polygons. Therefore, the entire algorithm takes  $O(n) + O(\log(n))$ : hence,  $O(n)$  node visits. Generally, only a low number of polygons are kept in the PVS and many more are rejected by the view frustum, meaning a result closer to that of  $O(\log(n))$  perframe.

### 3.1.2 Problems with the Quake BSP tree algorithm

Even with PVS and BB, polygons will still be sent for render that are not visible. These polygons will particularly remain around the larger nodes, which are more likely to be visible and have an all-encompassing BB. The root node, particularly, will always be sent to render because all paths lead to it and it always has part of its BB in the view frustum. Nevertheless, this method is particularly scaleable; because as level maps (the world) get bigger the polygonal savings increase. For example, a uniform map of 10,000-polygons, 99% (Moller and Haines, 1999, p. 201) of polygons could be culled. Therefore, only 100 polygons would progress through to the rendering pipeline. If the world was larger, more polygons could expect to be removed at an even better rate, which means that BB BSP trees with PVS are likely to be used well into the future.

### 3.1.3 Discussion

Quakes' PVS BSP tree algorithm, which has been used throughout all ID™'s Quake games (1, 2 and 3) is a scalable and efficient approach for removing hidden polygons. That said, the algorithm is by no means perfect, as there are still some polygons that will be sent to be rendered which are not actually visible. Many different implementations of Carmack's original Quake engine have spawned due to the engines popularity (see Appendix 2 - Quake Clone BSP Engines).

## 3.2 GLQuake and Quake 2

In ID™'s initial experiments with GLQuake, Quake 1 was modified to take advantage 3D hardware. Quake 2 followed GLQuake with more support for 3D hardware. An approach to hidden surface removal was needed to take advantage of the present day 3D hardware, which had progressed since GLQuake. 3D hardware was able to increase the number of polygons available per frame for rendering and provided automatic depth ordering by a hardware Z-buffer. Quake 2 still supported software rendering, so some compromises had to be made because ID™ didn't want to produce two separate BSP processing algorithms (one for software and one for hardware). The BSP part of Quake 2 ended up being similar to Quake 1's, and suffered from many of the same problems.

There are subtle differences between the Quake 1 engine and Quake 2 engine. Detailed brushes are stored in the tree leaves as whole objects, helping to prevent objects that are small and detailed from dividing the scene up too much. Considering that a small object is rarely partly visible, dividing those objects up into a BSP tree would make the tree less efficient. Detailed brushes that span two or more nodes in the BSP tree are simply sent down both branches. At runtime, objects are checked to make sure that they aren't drawn twice. DeWan (2000) discusses the subject of how Quake 2 and 3 uses detailed brushes in detail, in his web article "The BSP Process and Visibility".

One important change to the Quake BSP tree algorithm is that entity (dynamic objects) culling is no longer done in the BSP traversal. Once the visible areas have been defined, it is easy to determine visible entities just before rendering.

## 3.3 Quake 3

Not much information regarding Quake 3's engine has been released, however the Quake3 BSP compiler q3radiant has. Much of the information provided has been put together from comments made by John Carmack and the BSP file format for Quake 3.

### 3.3.1 The Implementation

Quake 3 was the first PC game that required 3D hardware support. That is to say it didn't support PCs without a 3D card. It is most likely a major reason why 3D cards became popular. By the time Quake 3 was released all the latest 3D hardware had fast Z-buffering included. Fast Z-buffering meant that there was little need to rely on the BSP to order polygons or even to determine backface culling (which is also supported by hardware). Therefore, except for the first stage where it determines the camera's location, the tree can be traversed in no particular priority-order and no back culling is needed. The Z-buffer takes care of getting the polygons in the right priority-order.

Polygons are not split in the tree; instead they are pushed down both sides of the tree as a whole. When the algorithm that is traversing all the visible nodes hits a leaf, it marks all of its faces as visible. The entire set of global faces, and any that are visible (marked), are sent to the renderer. Note that at this stage the Quake 1 hierarchical BB culling is being used to remove polygons that are not visible in the view frustum. In terms of  $m$  polygons that are global, this process takes  $O(m)$  visits to marked nodes. This technique is generally fast and avoids double rendering of duplicate polygons, because  $m$  is normally small. The algorithm also maintains polygon priority-order, which is useful in 3D hardware.

One deficiency of 3D hardware at that time (and it still is to a lesser degree now) was that 3D cards worked best when the textures were not thrashed around too much (because they had limited texture memory). The solution to this problem meant that for best performance, objects were sorted by texture priority-order, which is stored in a global array.

Like textures, each time a rendering state in the 3D hardware changes, the program takes a performance hit. Texture switching can be a slow operation, so it generally has first priority. However, other things such as lighting (which affects the state of the 3D card) can also cause bottlenecks. The compiler groups all objects with the same render state and renders them together, minimising state changes.

### 3.3.2 Discussion

Currently, there is not much publicly available information on the Quake 3 engine as ID™ is still selling that engine. What is known about the Quake 3D engine is that it groups objects by the render state and doesn't split polygons. In future, Quake3 engine source code will probably be released as its predecessors were. However, other than the Quake 3 engine, there are many other Quake 1/2/3 clone engines (Appendix 2 - Quake Clone BSP Engines) with their own ideas on 3D engine efficiency. The Quake3 engine is one of the most advanced engines available for modern hardware. Therefore, it is important to speculate at least on how things were implemented from information that is available.

## 4 BSP tree Implementation

The following sections aim to give an in depth look at three BSP algorithms: general BSP trees, BB BSP trees and PVS BB BSP trees. General BSP trees are used simply for priority-ordering of polygons. Once compiled, BSP trees takes  $O(n)$  to priority-order all polygons by the camera's view with a small memory overhead; where  $n$  is the number of polygons in the world.

General BSP trees were enhanced in DOOM using bounding volumes (bounding boxes). BB BSP trees can still produce priority-ordered polygons. However, more importantly, they can also dramatically reduce the amount of polygons that need to be sent to the rendering pipeline. BB BSP trees can have a maximum runtime cost of  $O(n)$  and a minimum of  $O(1)$ ; where  $n$  is the number of polygons in the world. The performance of BB BSP trees is affected by many variables, including tree structure and the scene. Therefore, it is difficult to get an accurate average, suffices to say that generally at least 95% of polygons can be removed for a large map.

PVS solid leaf BSP trees generally perform the best out of all three solutions when using 3D hardware as it can quickly discard many polygons that are definitely not visible. However, PVS trees trade pre-compilation time for better runtime performance. PVS trees generally employ the BB BSP tree algorithm as well to provide for view orientated culling.

### 4.1 General BSP tree

Shumacker et al. (1969) envisioned general BSP trees as a fast way of getting polygons in the right priority-order from the viewer's perspective. "The BSP works based on this premise –that is, at any point in the universe, it should be possible to compute the order of polygons to be rendered and create an efficient data structure to contain these orderings." (Lamothe, 1995, pp. 894) Furthermore, general BSP trees have many properties, which are exploited by algorithms such as BB BSP trees and PVS BSP trees.

#### 4.1.1 Compilation

Before a BSP tree can be used it must be built using a BSP compiler. Building a BSP tree with a compiler is normally a pre-processing procedure as it is slow on taking up to  $O(n^2)$ , not including balance and split reduction algorithms; where  $n$  is the number of polygons. The compiled tree is generally stored in a file (Moller and Haines, 1999, pp. 197) due to the process often being slow.

A general BSP tree can be thought of as a carving up of space into (non-closed) convex volumes. Each polygon is inserted into a binary tree in no particular priority-order. However, as will be shown later, the priority-order can have a significant impact on the efficiency of the tree. As each polygon is inserted into the binary tree they are classified as either left or right using the node's polygon plane (recursively) depending on whether it is behind or in front respectively.

Sometimes a polygon sits on both sides of the plane. In these cases the polygon is generally split down that plane into two new polygons which recursively continue down the tree. The original polygon is discarded (deleted). Splitting can make trees quite large; however it does prevent the crossover problem, which was described in "Problems solved by BSP trees". When a polygon reaches an empty (null) node, a new node with a reference to that polygon is created at that node. When all polygons have been inserted into the tree, the tree has been built.

The structure of the general BSP tree is made up of a normal, a list of vertices (that make up the polygon) and, a left (back) and right (front) node pointer to the child nodes for example:

```
Node
- Normal (X, Y, Z)
- Vertex_List (Xi, Yi, Zi)
- Front_Link (node)
- Back_Link (node)
```

There may be other user-defined components in this structure such as lighting or material and texture coordinates. In cases where more information is need for each polygon, a pointer to the information such as it is normal and material can be stored at each polygon. However, the vertex\_list (and texture coordinates) should stay separate because the polygon has a potential to be split. The data structure can be more optimal if it has pointers to lists of common data instead of duplicate lists when using the polygon splitting technique.



## BSP tree

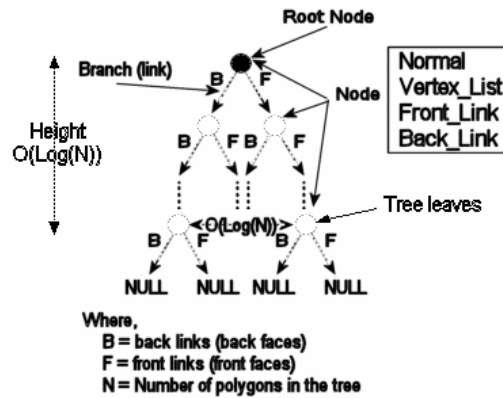


Figure 4. BSP tree structure

Figure 4 is an example a BSP tree. The dotted lines show that the connecting node is optional. Each node in the tree has a normal, vertex list, a front link and a back link. The links of the trees leaves point to NULL (nothing). What this diagram does not show is the non-balanced aspect of BSP trees where some nodes may have only one child. The base and height of a BSP tree, and most binary trees for that matter, has an average size of  $O(\log(n))$ ; where  $n$  is the number of polygons in the tree.

The following section will discuss the steps required, algorithmic details and necessary functions to implementing the general BSP algorithm. Note that the compilation algorithm described in the following text is only one of many.

The steps required for building a general BSP tree are:

1. Compute normal for all polygons.
2. Pick a start polygon.
3. Split the remaining polygons into two groups by the polygon that was picked.
4. Repeat the process for each child node.

See Appendix 6 (19.1) for an example of general BSP compilation.

## Classification

Classification is used to determine the side of a plane on which a polygon is. The basic plane equation shown in "The plane" is used for polygon classification. If the result is less than zero then the left side is traversed followed by the right side. Left and right sides are safely interchangeable; as it means that the front side and back side of the polygon will simply be reversed. If the result is zero, then an intersection has occurred. Further processing will be needed at this point to split the polygon up. See Appendix 4 (17.2) for example polygon classification code.

## Plane Classification

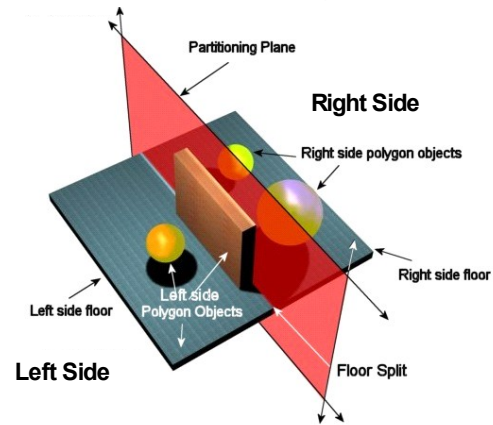


Figure 5. Plane Classification

Figure 5 divides a scene into two areas (left and right) using one of the polygon's planes as a partition in the scene. The arrows on the edge of the partition plane are used to represent an infinite plane. The floor exists on both sides of the splitting plane therefore it needs to be split into left and right polygons using a polygon intersection algorithm.

## Polygon Splitting

Polygon splitting is the process of dividing a polygon into two pieces along a plane. The following algorithm only supports convex polygons and, therefore, outputs convex polygons. However, all concave polygons can be converted into several convex polygons.

Polygon splitting is achieved by classifying every vertex in the polygon against the splitting plane. Vertices less than zero (more than zero) go on the polygon that is to be pushed down the left side (right side) of the tree. Vertices that intersect (generally rare) are converted into two vertices, one for each side. The vertices are processed in circular fashion so that edge intersections are detected when the one vertex is on one side, and the next is on the other. To determine the exact point of intersection along the plane the ray plane intersection is performed:

$$C_i = V_i - V_j$$

$$R_i = V_j + C_i D / (C_i \bullet N)$$

Where,

Let  $D$  be the dot Product of the (splitting) plane (calculated during the binary tree process)

Let  $V$  be the vertex in the polygon, which is a 3D vector (x, y, z)

Let  $N$  be the splitting plane normal, which is a 3D vector (x, y, z)

Let  $C$  be the vector resultant between two vertices, which is a 3D vector (x, y, z).

Let  $R$  be the position the vertex intersects the polygon

Let  $i$  represent the index of the vertex just before the intersection.

Let  $j$  represent the index of the vertex just after the intersection.

Let  $\bullet$  be dot product

Note that the dot product should be determined first (to avoid divide by zero errors). When the dot product operation result is zero, the vertex is on the plane.

Listing 1. Intersection

In Listing 1 the dot product of the plane has already been computed in the binary tree process so there is no reason to recompute that. New splitting vertices must be inserted into each polygon, after the first vertex and before the second vertex.

### Splitting a 2D polygon

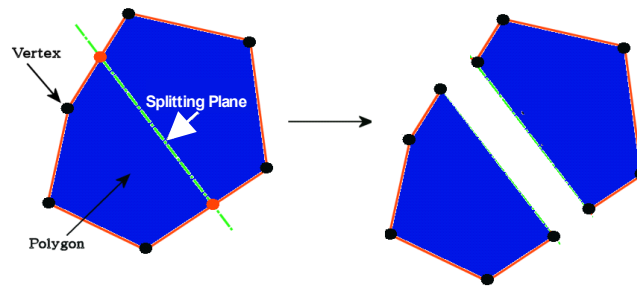


Figure 6. Polygon Splitting

Figure 6 shows the result of a 2D polygon split but it can easily be adapted to 3D. Note that the gap between the resulting polygons won't really exist; it is only for demonstration purposes.

This intersection technique is not accurate because it normally uses floating-point operations. Lowly accurate intersection algorithms (such as this one) tend to cause tiny fragments of polygon that not only clog up the tree and cause errors, but can cause artefacts during render time. A substantial difference can be noticed if double point precision is used. However, double precision operations still causes errors due to the overhead of storage and processing time. One partial solution to this accuracy problem is to group all polygons on the same plane and normal (with some leniency) and process them together.

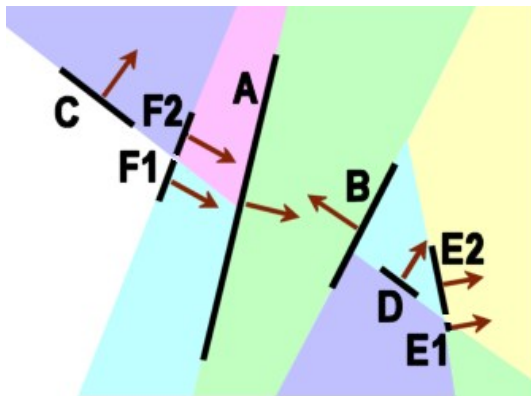
Note that if a vertex carries other details such as UV texture coordinates then those details must also undergo the same splitting processes. Children inherit most of their properties from their parents; therefore, all the polygon information (such as material) from the parent polygon (which these new polygons were made from) needs to be copied into both new polygons. Finally the old polygon should be removed. Intersection takes up to  $O(n)$ ; where  $n$  represents each vertex.

See 17.3 for example source code on polygon intersection and 17.7 for the polygon splitting algorithm.

## Convex Volumes

The BSP tree produced in 19.1 is made up of convex volumes as shown in Figure 7. Algorithms such as BB BSP trees and PVS BB BSP trees take advantage of BSP tree convex volumes by exploiting the fact that one convex volume will never overlap another convex volume. The colour shades in the diagram show each volume that is formed by each node in the tree.

**World Map (from above)**



**BSP tree (Make Groups)**

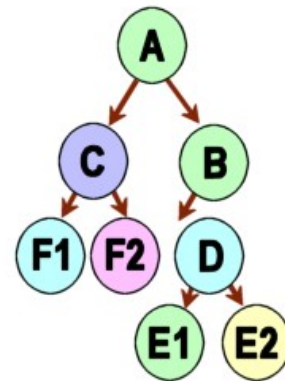


Figure 7. Convex Volumes

## Balancing Verse Splitting

Tree balancing is an important aspect to take into account because it has a large impact performance. The number of splits, and how balanced the tree is, depends on the order polygons are passed to the compilation algorithm (selection criteria). "Every polygon which is not located upon a unique side of the parent (partitioning plane) must be split into two parts thus increasing the number of fragments in the scene." (James, 1999, pp. 18)

If a tree has too many splits, then it becomes larger than need be as Figure 8 shows. However, if the tree is too linear (not balanced) in nature then it can have an effect on the height of the tree as Figure 9 shows. Although the height of the tree won't affect general BSP runtime, because the traversal needs to visit  $n$  nodes anyway, the height can effect the time of compilation (James, 1999, pp. 22). Furthermore, HSR BSP tree algorithms such as BB BSP trees and PVS BB BSP trees can benefit from a balanced BSP tree algorithm because the visible polygon determination from any viewpoint becomes more balanced. Sometimes it is possible to achieve perfect balance and minimise splits, however in most situations that is not the case. However, in investigations (although not enough tests where performed in this area to make a broad statement), balancing appeared to help reduce splits to a small extent in most cases. This is because, in many cases, balancing causes small combinations of possible splits at each level in the tree allowing the splitting algorithm to be more precise.

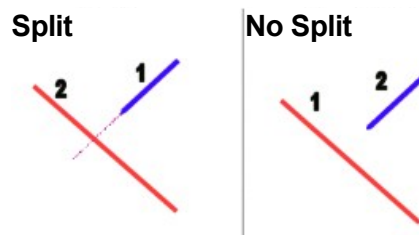


Figure 8. Splitting can be reduced by better selection

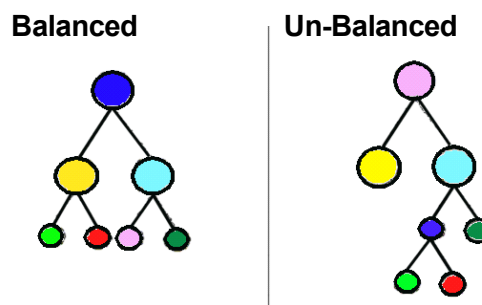


Figure 9. Balance Verse Un-Balanced BSP trees

The “Least-crossed criterion” (LCS) and the “Most-crossed criterion” (MCS) are some heuristics that can be used to reduce the amount of splits in a BSP tree (James, 1999, pp. 20 - 22). LCS means that a sample set of polygons are chosen and the plane that splits the smallest amount of polygons at that level are chosen as the partition plane at that level in the tree. MCS means that sample sets of polygons are chosen and the ones that get split the most are chosen as the partition plane, which means that this polygon will be removed from causing a high number of potential splits.

“Tie-breaking” (James, 1999, pp. 22) is the combination of the two techniques which uses the other algorithm (LCS or MCS) to break the tie when there are two polygons with the same best case result. James (1999, pp. 72) indicated that LCS ties broken by MCS were generally the better technique. In addition, James (1999, pp. 72 - 86) examines several more advanced split reduction techniques, “Enhanced Minimization” and “Conflict Neutralization”, in his “Conflict Minimization” section which also takes into the account the depth of the tree, where polygons can be used as split protectors for other polygons. “Conflict Neutralization” is discussed in 2.2.2.

A BSP tree can be balanced using a technique similar to that of AVL-binary trees (invented by GM Adel'son-Velskii and EM Landis) which is discussed in Fuller & Kent. (1999). The tree balancing process is to estimate the size of each side and pick a partitioner polygon at each node that gives a reasonably good balance. Determining a good balance can be complex due to the fact that splitting planes will offset the number of polygons on each side depending on the number of splitters chosen. Some balancing algorithms only look at the balance on one level (James, 1999, pp. 72). Due to the generally large size of BSP trees, a smaller sample of candidate partitioner polygons can be used to increase compilation speed.

To provide for both balancing and splitting, a hybrid costing system is used that weighs values of splits against balances using a costing mechanism. The lower the cost the more likely a particular combination is best. “A linear combination of the two efficiencies is used to rank the candidates, and the best one is chosen.” (A. Kumar and Kwatra, 1999, pp. 38). Lower splits are generally given the preferential weighting, although there is argument to which should be given the best weighting.

The bigger the tree becomes the more combinations of polygons there are, it is exponential with a potential rate of  $O(2^n)$ . Normally sampling a few polygons for example 4 or 5 (Foley, Dam, Feiner, and Hughes, 1990; James, 1999, 20) at each node, is significant to produce reasonable results. The best sampling amounts should be determined by the application being performed: that is, the more time the user has and the more processing power the user has the higher the sampling rates can be.

### 4.1.2 Runtime

General BSP trees are used to priority-order polygons by their distance to the camera. The process involves the camera being sent down the tree and classification by each polygon plane. Once the BSP tree is compiled traversal takes a general BSP tree  $O(n)$ , to priority-order a list of  $n$  polygons from any camera location in the world as the traversal visits each node once.

When the result of classification is less than (more than) zero, the back (front) tree is traversed, if that node exists. If zero, then it's up to the programmer to decide what action to take as the polygon is facing the viewer face on. If the programmer wants to swap the order the polygons render, it'd simply be a matter of switching the front and back node traversals around in the algorithm. After recursively traversing the front or back node (if that node exists) the other node is traversed. Whenever a node is reached it is sent down the rendering pipeline. As mentioned before, back-facing polygons can be culled at this stage, by simply not rendering them; nevertheless, those nodes must still be visited.

```

Function Bsp_TraverseGL ( node )
IF [ node exists]
    IF [  $c \bullet node.plane < 0$  ]
        //Camera is behind this wall process the front wall sub tree
        Bsp_TraverseGL node.back
        Draw node
        Bsp_TraverseGL node.front
    ELSE
        //Camera is in front of this will process the back wall
        Bsp_TraverseGL node.front
        Draw node //Note that this can be excluded
        Bsp_TraverseGL node.back
    END IF
END IF

```

Listing 2. Draw General BSP tree in order pseudo code

Listing 2 demonstrates the runtime code of the general BSP tree. The second line terminates traversal when the node is not found. The following line performs classification with the camera and the node to get the node's face position in respect to the camera. If the polygon is front facing, then (in this order) the:

- back node is traversed;
- current node is drawn; and
- front node is traversed.

Otherwise (back facing)

- front node is traversed;
- current node is drawn (optional as the face is backfacing anyway); and
- back node is traversed.

See 19.2 for an example of general BSP traversal.

### 4.1.3 Conclusion (General BSP Tree)

BSP tree compilation is a slow process which takes up to  $O(n^2)$  and is generally done once and saved to a file; where  $n$  is the number of polygons. If split minimisation and balancing is used, the process increases to  $O(n^3)$  or more, but generally because there are less splits and the tree is more balanced the process is closer to  $O(n^2 \log(n))$ . It is not uncommon for BSP map levels (including PVS and other pre-calculations such as lighting) to take hours to compile, depending on the speed of the CPU and size of the map (Abrash, 1997, pp. 1278). Once a tree is compiled, polygons can be priority-ordered at runtime with a maximum of  $O(n)$ , with cheap tests at each node (polygon).



While general BSP trees can greatly improve rendering time by providing view ordered polygons, they do not reduce the amount of polygons sent through the rendering pipeline (aside from back face culling). Furthermore, software based BSP trees for view ordering is generally not as fast as hardware Z-buffers.

Although using BSP trees for priority-ordering polygons is becoming redundant, they are still of use to many applications; such as polygon selection and translucent polygon priority-ordering, which Z-buffers are not good at. Furthermore, BSP trees have many properties that can be exploited, such as with the BB BSP tree algorithm and the PVS BSP trees.

## 4.2 BB BSP trees

The BB BSP tree algorithm uses hierarchical bounding boxes to perform view-orientated HSR. Bounding boxes are commonly used in game engines (for example in the doom, Quake, unreal, etc.) because they are easy to frustum cull and generally have a tighter fit than bounding spheres. Volumes such as k-DOP (discrete oriented polygon types) could also be used. In any case, the BV BSP tree technique is extremely scalable to both hardware and non-hardware accelerated situations. Figure 10 gives an example of an AABB BSP tree:

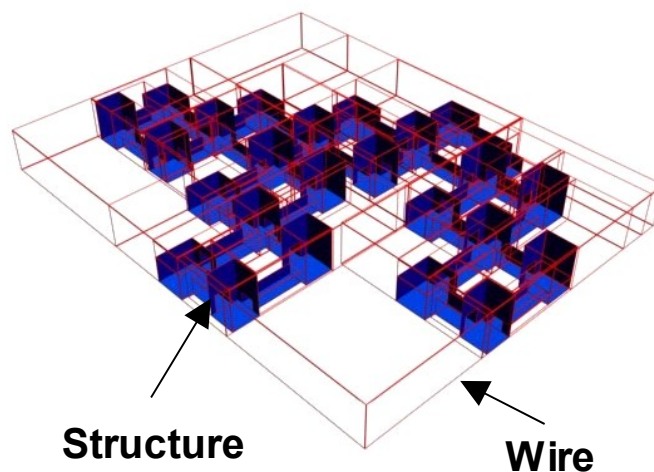
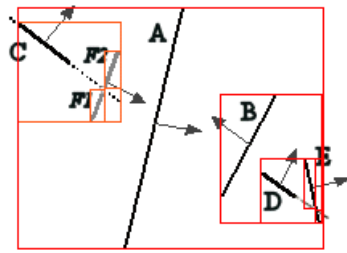


Figure 10. AABB BSP tree

In Figure 10 the bounding box is represented by the wire frame surrounding the structure (world). Each bounding box is subdivided into more bounding boxes until the tree leaf is reached.

The BB BSP tree approach is similar to that of octrees except that the algorithm generally generates a better fitting BV. Each node in the tree contains a minimum BB encompassing the area that the node and all its children fit within. At runtime any node that has a BB that is completely out of the VF is not rendered, including its children. Consequently, there is potential for half the remaining polygons in the tree to be discarded at every node in the tree.

**World Map (from above)**



**BSP tree (Make Groups)**

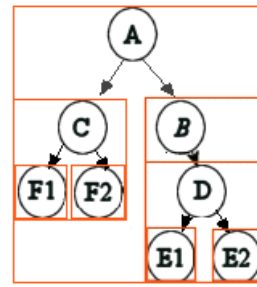


Figure 11. BB BSP tree

Figure 11 shows the results of adding BB to the BSP tree generated in 19.1. Notice that node B can be pruned in one test, removing nodes D, E1 and E2 with no further tests.

### 4.2.1 Compilation

The extra step need to change general trees into BB BSP trees is to compute a minimum BB at each node. The BB for a particular node must be computed after all its children (recursively) have been generated because the size of a particular node cannot be determined until all its child BBs have been determined.

One approach is to pass back the BB as each node completes traversal (in compilation). Therefore, when a leaf is reached, its polygon's BB is passed back to its parent polygon during the traversal. The parent node then computes the total minimum BB of itself and both its children. This BB is then passed back up the tree until it reaches the root, which should compute a BB of the entire scene.

It should be noted that some programmers choose not to compute the root's BB because the likelihood of the root's BB being visible (not being culled by the view frustum) is 100% in most situations. The leaf nodes' BB, or indeed higher nodes, may also be left out of this stage because the process becomes a one-to-one polygon test, which can be solved cheaply by hardware.

BB can be either axis aligned BB (AABB) or orientated for a best-fit BB known as Oriented BB (OBB). AABB BSP trees are similar to KD-trees in that each BB is aligned to a global axis. OBB can have a tighter fit than AABB but requires more calculations and storage than AABB. The difference between AABB and OBB calculations are negligible when compared to the increased accuracy in polygon culling, however AABB requires less storage space and is quicker to compile (Bergen, 1998, pp. 2). Moller and Haines argue that, "The convergence of the OBB to the underlying geometry of the models was generally better than that for either AABBs or spheres." (Moller and Haines, 1999, pp. 350). Having a tighter fit means that nodes have a better chance of not being found within the view frustum and, consequently, pruned further up the tree, which means fewer tests.

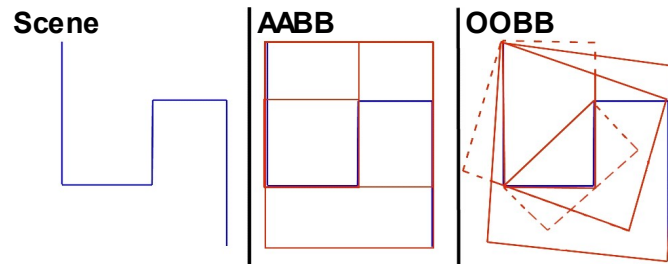


Figure 12. AABOBB verse OBB

Figure 12 demonstrates two different ways a 2D scene could be broken up into using AABOBB and OBB algorithms. Note that the dotted lines in OBB indicate parts that could be clipped out by BB frustum culling higher up the hierarchical structure.

Computing the BB is a process of finding the smallest (largest) X, Y and Z out of all the smallest (largest) vertices respectively which make up the BB. Therefore, for an AABOBB BSP tree the algorithm would look something like Listing 3.

```

//First Determine nodes polygon BB
Smallest.X = Largest number
Smallest.Y = Largest number
Smallest.Z = Largest number
Largest.X = Smallest number
Largest.Y = Smallest number
Largest.Z = Smallest number
FOR N = 0 TO [Number of nodes vertices]
    IF [Smallest.X > [nodes vetex (n)].X] Smallest.X = [nodes vetex (n)].X
    IF [Smallest.Y > [nodes vetex (n)].Y] Smallest.Y = [nodes vetex (n)].Y
    IF [Smallest.Z > [nodes vetex (n)].Z] Smallest.Z = [nodes vetex (n)].Z
    IF [Largest.X < [nodes vetex (n)].X] Largest.X = [nodes vetex (n)].X
    IF [Largest.Y < [nodes vetex (n)].Y] Largest.Y = [nodes vetex (n)].Y
    IF [Largest.Z < [nodes vetex (n)].Z] Largest.Z = [nodes vetex (n)].Z
END FOR
//Then compare that with its children's BB (if it is not a leaf)
IF [Left Node Exists]
    IF [Smallest.X > [Left Node BB].X] Smallest.X = [Left Node BB].X
    IF [Smallest.Y > [Left Node BB].Y] Smallest.Y = [Left Node BB].Y
    IF [Smallest.Z > [Left Node BB].Z] Smallest.Z = [Left Node BB].Z
    IF [Largest.X < [Left Node BB].X] Largest.X = [Left Node BB].X
    IF [Largest.Y < [Left Node BB].Y] Largest.Y = [Left Node BB].Y
    IF [Largest.Z < [Left Node BB].Z] Largest.Z = [Left Node BB].Z
END IF
IF [Right Node Exists]
    IF [Smallest.X > [Right Node BB].X] Smallest.X = [Right Node BB].X
    IF [Smallest.Y > [Right Node BB].Y] Smallest.Y = [Right Node BB].Y
    IF [Smallest.Z > [Right Node BB].Z] Smallest.Z = [Right Node BB].Z
    IF [Largest.X < [Right Node BB].X] Largest.X = [Right Node BB].X
    IF [Largest.Y < [Right Node BB].Y] Largest.Y = [Right Node BB].Y
    IF [Largest.Z < [Right Node BB].Z] Largest.Z = [Right Node BB].Z
END IF

```

Listing 3. Computing the BB

The algorithm in Listing 3 shows how to compute an axis-aligned BB. The result of which would not only be stored in the current nodes record, but also passed recursively up the nodes parents in the tree.

Best fitting OBB are more difficult and much slower to compute. There are various techniques proposed to calculate best fitting BB for a group of polygons (Gottschalk, 1999, pp. 38-62; Moller and Haines, 1999, pp. 351-352). However, there can be problems with these techniques when they do not take into account BB tree clipping. In many case orientating the boxes so that more is clipped can yield a tighter fitting than simply picking the best fitting bounding volume as Figure 13 shows.

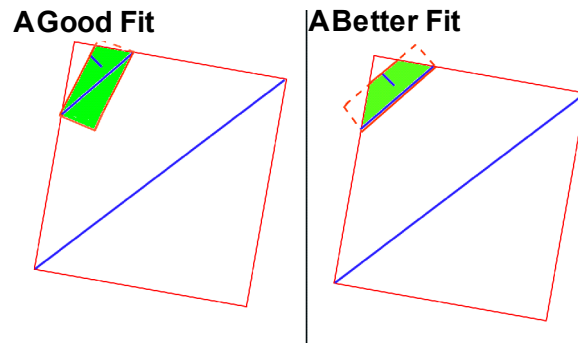


Figure 13. The clipping effect

Figure 13 shows how a best fitting BB for its contents may not always be the best choice. Although the BB is bigger, when clipping is taken into account, more of it is culled to generate a tighter fitting BB.

The extra time needed to create AABB BSP trees is  $O(n)$  therefore computation does not affect the big O at compile time. An OBB BSP tree requires more calculations to determine good fitting polygons.

The steps required to compile a BB BSP tree are similar to that of the general tree. The only difference is step 5 (in bold).

1. Compute normal for all polygons.
2. Pick a start polygon.
3. Split the remaining polygons into two groups by the polygon that was picked.
4. Repeat the process for each child node.
5. **Determine the BB at each node.**

See 19.3 for an example of BB compilation in a BSP.

## 4.2.2 Runtime

At runtime BB BSP trees are able to cull away many polygons that are definitely not visible within the view frustum with only a relatively small amount of tests. The BB BSP tree culling algorithm does this by removing polygons in clusters. As the algorithm progresses down the tree it prunes nodes (which include that node's children as well) that have BB that are not within the FV.

## BB Frustum Culling

There are many approaches to view frustum culling of BB. What follows is one such approach, which is described more completely by Mark Morley (Morley, 2000). To determine if an OBB is within or crossing a view frustum, each of the BB vertices' are tested against the view frustums planes using dot product classification. The algorithm can exit early if all vertices are found outside one of the frustum planes, meaning that the BB is outside. If vertices are found on both sides of the view frustum then the BB is considered crossing the frustum, plane as bounding box D is in Figure 14. If the BB is found to be within (or partially within) all frustum planes, then it is considered inside the frustum.

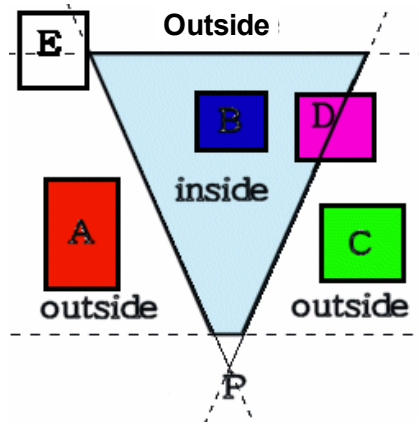


Figure 14. View Frustum and Objects

Figure 14 shows a 2D view frustum with bounding boxes A, B, C, D and E. A and C are outside the view frustum while B is within. D is partially within the frustum, so its contents are assumed inside. E (problem E) will be treated as inside even though it is outside the view frustum because it is in front and behind two planes.

The algorithm would look something like Listing 4:

```
FOR EACH [plane,  $P$  ]  
  IF [any vertex on inside of  $P$  ]  
    //It is potentially within the view frustum  
  ELSE  
    RETURN outside  
  END IF  
END FOR  
RETURN inside
```

Listing 4. BB View Frustum Culling

Problem E can be resolved by testing each of the BB planes by the view frustum vertices when an intersection is found. The second part of the algorithm (part 2) would be similar to the one presented in Listing 4 except the words BB and frustum would be switched around. Obviously, the down side of part 2 is a trade off, time for accuracy. Time can be saved at each node by not doing part 2, however increased accuracy (using part 2) can mean more polygons are discarded earlier in the pipeline.

AABB can be done the same way as OBB, except some its features can be exploited to make AABB process slightly faster then OBB. For example, the problem E could be solved with AABB by simply testing if any of the frustum points are within the AABB using greater-than and less-than signs.

Information on enhanced BB and frustum culling can be found in "Kenny Hoff's Graphics deport" (K. E. Hoff, 1996), particularly in the research paper "A "Fast" Method for Culling of Oriented-Bounding Boxes (OBBs) Against a Perspective Viewing Frustum" and "Real Time Rendering" (Moller and Haines, 1999) section "10.11.2 Frustum/Box Intersection".

## **Tree BB Frustum Culling**

There are several enhancements that can be made to BB trees because of the fact that a child BB inherits BB properties from its parent:

1. If a node's BB is outside of the view frustum, then ignore it and all of its children, effectively pruning them from the tree.
2. If a node is completely within a frustum, then there is no more need to keep testing its children against the BB. At this point the general BSP tree algorithm can be used.
3. If the BB is on the inside of one or more view frustum planes, then all it is children will also be, meaning that there is no need to test against that plane again. This method can also be used to accomplish 2.

Concept 1 can be used without 2 and 3 and concept 2 can be used without 3. However, the best combination is generally all three rules. A slight modification to Listing 4 to embrace these concepts is shown in Listing 5:

```

Function Bsp_TraverseGL ( node )
FOR EACH [testable frustum plane]
  IF [any node.volume.vertices on inside of the plane]
    IF [all node.volume.vertices on inside of the plane]
      REMOVE [plane from testable planes]
    END IF
  ELSE
    STOP PROCESSING [ node and it's children]
  END IF
END FOR
PROCESS node AS PER NORMAL //As shown in Listing 2

```

Listing 5. Tree BB View Frustum Culling

What Listing 5 does not show is the algorithm's recursive nature, where children may have fewer planes to test against as the algorithm traverses. Also note that the removed testing planes should only affect the children of the nodes being tested. Therefore, the recursive version of this could place an ignoring plane mask (6 bits) on the runtime stack.

If the algorithm uses a fast (hardware) Z-buffer then the algorithm could be simplified to Listing 6 because polygon ordering is not necessary.



```

Function Bsp_TraverseGL ( node )
IF [ node exists]
  FOR EACH [testable frustum plane]
    IF [any node.volume.vertices on inside of the plane]
      IF [all node.volume.vertices on inside of the plane]
        REMOVE [plane from testable planes]
      END IF
    ELSE
      STOP PROCESSING node AND ITS CHILDREN
    END IF
  END FOR
  Draw node
  Bsp_TraverseGL node.back
  Bsp_TraverseGL node.front
END IF

```

Listing 6. Tree BB View Frustum Culling without Order

BB BSP trees are an efficient and scaleable way of culling large amounts of polygons at runtime with minimal calculations. They produce better fitting boundaries than octrees but are more costly to calculate and are, therefore, not dynamic. At runtime BB BSP tree traversal has a minimum of  $O(\log(n))$  and a maximum of  $O(n)$ ; where  $n$  is the number of polygons in the world. The maximum case normally occurs when most of the polygons are visible to the view frustum which is rare in an enclosed indoor map.

*An algorithm used both to compute this mask and find out whether a BB is visible within a view frustum is shown in 18.2. An example of BB BSP tree frustum culling can be found in 19.4.*

## BB Occlusion

Although BB BSP trees algorithm presented in “Tree BB Frustum Culling” was able to remove polygons that were not visible to the viewer, it does not take care of occlusion. Possible solutions for adding occlusion culling to BB BSP trees are either to use a Z-pyramid as discussed in “Octrees/Quadrees and KD-trees” (Chapter 2) or a BSP tree with PVS as discussed in 4.3 or hardware accelerated occlusion testing, which is built into newer 3D cards.

The extensions GL\_HP\_occlusion\_test, GL\_NV\_occlusion\_query and GL\_ARB\_occlusion\_query are examples of OpenGL API extensions (OpenGL® Extension Registry, 2004) that can be used for hardware occlusion. The extension returns true if an object rendered to the scene changes the Z-buffer (the scene being rendered). A BB can be rendered to the scene invisibly in back to front order. When the result is false, all the nodes (including its children) can be rejected. The GL\_NV\_occlusion\_query and GL\_ARB\_occlusion\_query have extra features that allow the program to continue executing while it determines if a polygon is visible. BB BSP trees can work well with this extension, because while one node is being investigated another node waiting to be processed can be sent down the pipeline for testing.

### 4.2.3 Comparison

At first glance BB BSP trees appear to be a mixture of a BSP trees with an octree (KD-tree), however there are some subtle differences. BSP trees tend to push much of the processing into the compilation stage; whereas octrees (KD-tree) can be generated on the fly (Nuydens, n.d.). Furthermore, BSP trees have the potential to produce many more splits than octrees (Nuydens, n.d.) and work best for indoor worlds. However, BB BSP trees provide the ability to depth order the scene and produce tighter fitting BB (particularly with OBB) than octrees or KD-trees. Tighter fitting BB means that there is a greater chance for early rejection of more polygons.

Portals provide a means of polygon culling and depth ordering; however, polygon sectors can be extremely large or fragmented. Many small sectors require large amounts of clipping (Abrash, 1997), whereas large sectors made up of many polygons means that less polygons will be culled. With BB BSP trees a sector (volume) has a better chance of early removal because each volume has several chances of removal as the tree is traversed. Nevertheless, BB BSP trees by themselves do not address the problem of occlusion, like portals do. Consequently, the idea of PVS BB BSP trees, which could be considered a mixture of portal and BB BSP techniques, was invented.

### 4.2.4 Conclusion (BB BSP trees)

A BB BSP tree is simply a general BSP tree with BB added at each node, representing the volume each sub-tree (node) contains. BB BSP trees provide a tight fitting BB hierarchy, which provides for early rejection of large amounts of invisible polygons for a particular scene. The BB BSP tree algorithm is an excellent technique for removing polygons that are not within the view frustum. However, by itself, the algorithm does not provide for occlusion culling. Nevertheless it can be combined with other techniques such as the Z-pyramid, hardware occlusion or PVS to provide for this capability.

## 4.3 PVS BSP trees

The concept of PVS (potentially visible sets) was originally brought to light by Seth Teller (Teller, 1992a, 1992b, 1992c; Teller and Séquin, 1991). The PVS BSP tree algorithm provides an efficient and scaleable algorithm for indoor static 3D worlds. An indication of PVS BSP trees scalability is in the fact that it was used in Carmack's next three engines: Quake 2, Quake 3 and Return to Castle Wolfenstein. The tree works well in software engines (i.e., Quake) but is easily adapted to take advantage of the latest 3D hardware (Quake 2, Quake 3 and "Return to Castle Wolfenstein").

BSP trees with PVS sacrifice memory for speed. Leaves are grouped into clusters and information (the PVS) about what other clusters are visible are stored (compressed) at each related leaf node in the tree (McGuire, 2002). Therefore, by finding the leaf the player is within, all the potentially visible polygons can be determined in one branch traversal  $O(\log(n))$  without recursion; where  $n$  is the number of polygons in the world. The PVS stage of PVS BSP Tree traversal is generally faster than BSP tree traversal because only one path needs to be traversed.

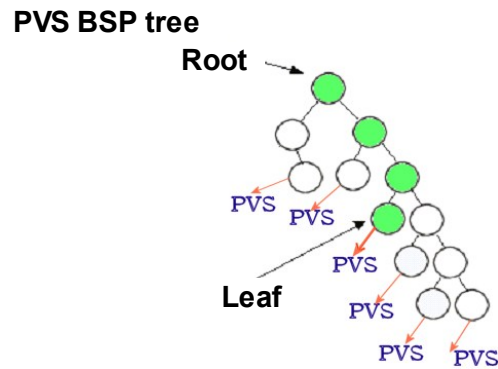


Figure 15. BSP Tree with a PVS at each leaf

Figure 15 is an example of a BSP tree with PVS. The shaded nodes indicate a potential path that could be taken to find the leaf volume where the camera resides.

### 4.3.1 Compilation

There are two main stages to BSP tree with PVS compilation.

1. Solid Tree Generation.
2. Calculating the PVS.

## Solid Leaf tree Generation

A solid leaf BSP tree compilation has a few extra requirements over and above general BSP tree compilation:

- An object being compiled into the tree must be solid, therefore it is a good idea to produce a solid object using CSG. Any holes that are too costly to fill round such as windows looking outside can be filled with an invisible polygon.
- An object that lies exactly on the plane must be pushed down the side it faces or stored as a coplane. Note that in the general trees the side didn't matter.
- Every polygon must be used as a partitioning plane. That is, the last node in the tree has an empty front leaf and a solid back leaf.

Steps involved performing solid BSP creation:

1. Compute normals.

2. A partition plane is picked from any planes that have not already been partition planes.
3. All polygons are partitioned into two groups by the partitioning plane. If there is only one plane left then that plane is made a node with solid area on the back and an empty leaf on the front. If there are no planes left, then that leaf is made solid.
4. Repeat steps 2 - 5 recursively until there are no more partition planes to use.

See 19.5 for an example of solid BSP tree creation.

## Calculating the PVS

This stage involves taking the solid BSP tree and determining what leaves can be discarded at each leaf (polygons that are never visible from the current leaf) and compressing the information so that the PVS data does not consume too much memory.

There are 3 main stages to PVS generation:

1. Portal generation.
2. Anti-penumbra clipping.
3. Compress the Information.

Stage 1 pre-computes local portals for each sector (leaf) in the map. Once the portals are known indirect sector connections (by portal) can be determined. The anti-penumbra provides a way of looking though these indirect portals to determine what other sectors (and therefore polygons) are potentially visible though the indirect portals. The compression stage uses zero run length encoding (ZRLE) to compress the stored at the leaves into a reasonable size.

### Portal Generation

As explained in section 21.1.9 a portal is a gap that sits between two areas such as a doorway or window. There may be many ways to interpret portals for a particular scene as Figure 16 demonstrates.

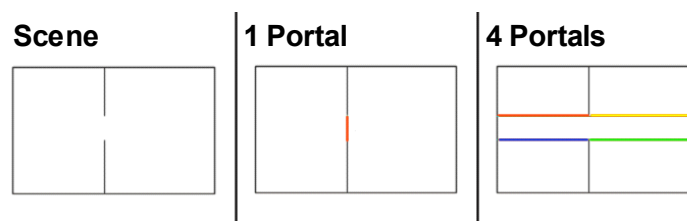


Figure 16. Portal Interpretation

Figure 16 shows a scene and two different ways a computer could interpret portals in that scene. The coloured lines represent portals. The higher (lower) the amount of portals to polygons the more accurate (less accurate) culling will be and the more (less) tests and clipping are needed. At some point having too many or little portals will become a bottleneck rather than an improvement, as Figure 17 shows.

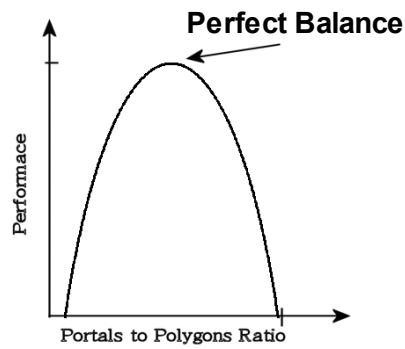


Figure 17. Portals/Polygon verse performance (illustration only)

Figure 17 demonstrates how the rate of portals to polygons can affect performance. This example is subjective because there are so many other factors affecting the perfect balance. Therefore, the example should only be used as a guide to understanding the relationship between amounts of portals and polygons. BSP trees generally create so many portals that the task becomes more about reducing the amount of portals rather than increasing them.

One way of creating portals is by creating large polygons that are aligned with planes in the tree and clipping them using the tree. Parts of portals that end up in solid space (because portals cannot exist in solid space) are removed, leaving leftover portal fragments. Each portal should end up linking one front and one back leaf. To perform portal creation the large polygon does not need to start at the top of the tree. Rather, the portal algorithm can start at the node it represents and be clipped by its parents and sectors that link to that node before being clipped by its children.

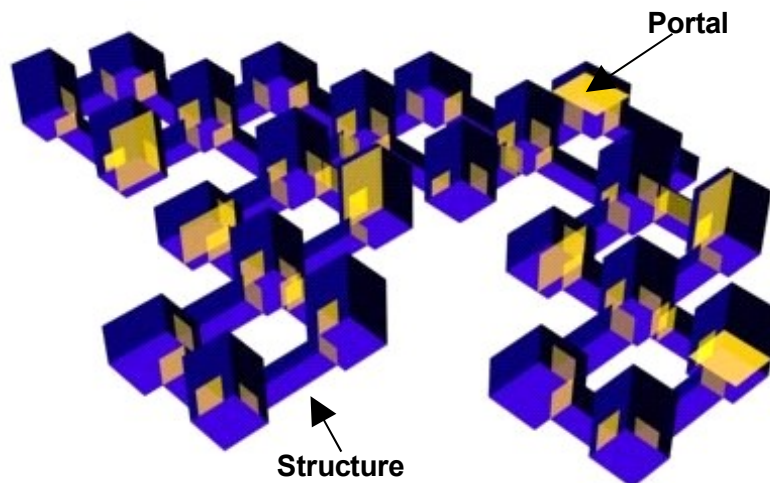


Figure 18. Portals

Figure 18 shows the portal that could exist in the given map. Portals are represented by the transparent bright polygons and structure (walls) by the darker polygons. Note that most portals are around doorways but some are created elsewhere.

In order to perform portal culling efficiently, two linked-lists (one for each side of the node) are kept at each portal. Each portal also maintains information about its two parents (back and front), its shape (polygon) and its plane. That is:

```
Portal
- Polygon
- Plane
- Back/Front parent node (array of 2 nodes for convenience)
- Link to Back/Front portal (array of 2 nodes for convenience)
```

Each tree node also has a field that points to the first portal in the linked list of portals at that node.

Steps involved in creating portals in a solid BSP tree:

1. Start at the root node N.
2. Create a portal (polygon) P larger than entire world along N's plane. For example, polygons that are larger than the entire world. The algorithm for creating this polygon is given in 17.6.
3. Clip P by the N's parent planes (recursively).
4. Push P down to the two child nodes (back and front) of node N, storing the back and front nodes in P as its front and back sectors.
5. Partition every portal pushed down to N by N's plane. If the portal ends up on the back (front) then push the portal down the back (front). If the portal is split then push the portal down both back and front, storing (overwriting) the back and front nodes in P as the portals front and back sectors. Note that at the root node there won't be any portals to process. Portals that land in solid space can be removed. It is also worthwhile to remove portals that become too small. An example algorithm for determining if a portal is too small can be found at 17.5.
6. N becomes the back child of N unless the front child of N is a leaf, repeat steps 2 – 8.
7. N becomes the front child of N unless the back child of N is a leaf, repeat steps 2 – 8.
8. Filter out any portals that:
  - don't have a back and front sector (leaf)
  - meet any of the world borders as there weren't clipped

*An example of the portal generation algorithm can be found at 19.6. Source code can be found at 17.9.*

### **Anti-penumbra Clipping**

Seth Teller details various algorithms to use Anti-Penumbra to compute potentially visible volumes in his paper "Computing the Antipenumbra of an Area Light Source" (Teller, 1992a).

### *Why Anti-penumbras?*

One approach to generating PVS is to use distributed viewing points throughout the sector and cull anything that cannot be seen from all those viewing points. However, how can a good and small set of viewing points that will take into account every possible view position the camera can be in within that sector to be generated? One possible approach to the point generation problem is to use only vertices on the each of portal planes, however this solution still does not guarantee to find every polygon that is potentially visible.

Since BSP trees are convex and because portals provide information about how rooms are connected to each other, anti-penumbras can be used to determine what is potentially visible from each sector (leaf) in the tree. Anti-penumbras provide a way to make sure all polygons that can be seen from a certain sector will always be visible. However, anti-penumbras only provide an approximation; therefore, polygons that will never be seen may still be included in the polygon set. This small amount of inaccuracy is normally considered insignificant in comparison to the large amount of definitely-not-visible polygons anti-penumbras can remove.

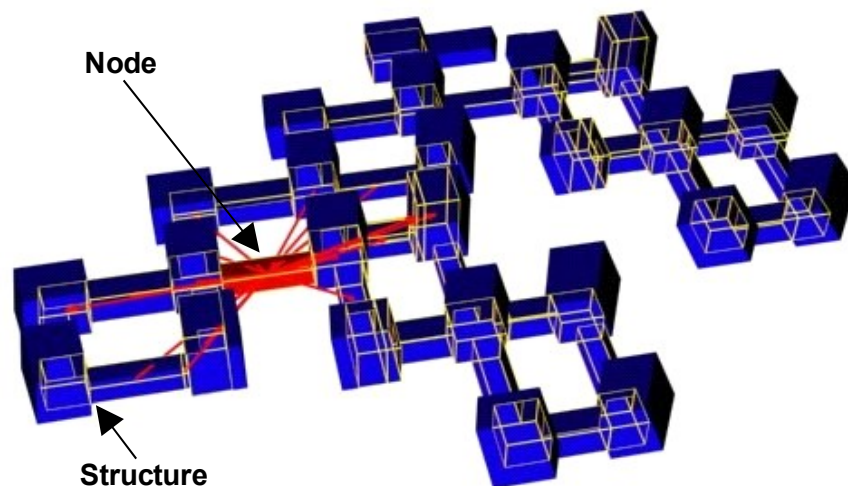


Figure 19. PVS of one node

Figure 19 shows the potential visibility of one node in the tree. The lines (in the star shape) point to the sectors the node (at the centre of the lines) can see. The wire boxes are explained in “Leaf Bounding Boxes”.

### *Creating an Anti-penumbra*

Creating an anti-penumbra requires two portals a linking (source) portal and a target (destination) portal. Creating a 2D anti-penumbra is a simple process of taking the two lines from the source portal and crossing them over so they pass through the two vertex points in the target portal.

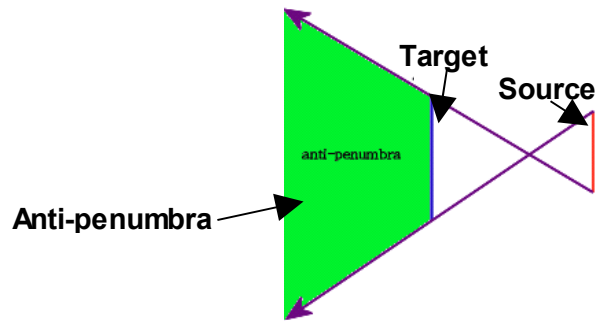


Figure 20. 2D anti-penumbra

Figure 20 shows a 2D anti-penumbra (shaded) which was created from two portals (target and source). Note that the shape the anti-penumbra makes is infinite (indicated by the arrows).

Producing a 3D anti-penumbra proves more difficult. The task is to produce the largest area possible within the anti-penumbra by creating clipping planes that pass through both the source and target portal vertices with the source portal on one side of the clipping plane and the target on the other. Triangles are used to help calculate these clipping planes by having one vertex on the source portal and two on the target portal. On the source polygon (for any given shape) vertices may not be reused. On the target shape the same edge (same two vertices) may not be reused for any given shape and must only go in one winding direction. Any number of sides can be used for each portal and they do not need to be the same.

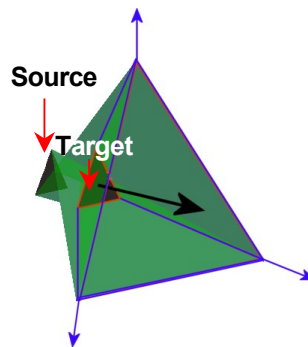


Figure 21. 3D anti-penumbra

Figure 21 illustrates a 3D anti-penumbra made out of triangular source and target portals.

To determine if a plane divides the source and target portal on opposite sides, the polygon classification algorithm can be used: that is to say, three points in the source and target portals are classified (using dot product) against the plane (triangle) being tested. If the results are opposites (one is negative and one positive), then this polygon has successfully found to a plane of the anti-penumbra.



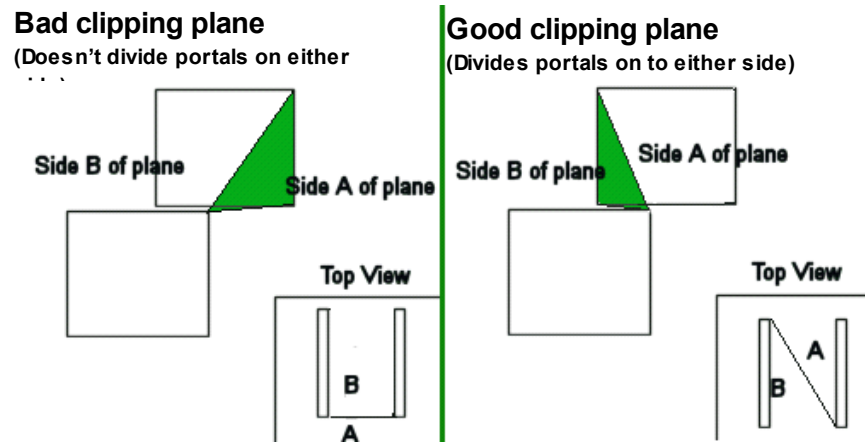


Figure 22. Anti-penumbra plane determination

Figure 22 shows the difference between a bad clipping plane and a good clipping plane (an anti-penumbra plane) using two 3D squares as portals. The top view (in the corner) shows what the result looks like from above.

The pseudo code for 3D anti-penumbra calculation would look something like this:

```

FOR EACH [vertex n in Source]
  FOR EACH [edge in Target]
    makePlaneFromTriangle(
      [Source.Vertex(n)], //From Source Portal
      [Target.Vertex(n)], //Edge vertex 1 from Target Portal
      [Target.Vertex([next edge after n])] //Edge vertex 2 from Target Portal
    )
    IF [Target and Source are on either sides of plane]
      ADD [to Anti-Penumbra Clip plane list]
    END IF
  END FOR
END FOR

```

Listing 7. Anti-penumbra creation

In Listing 7 the function `makePlaneFromTriangle` will produce a clipping plane (if possible) from the 3 vertices given. For increased anti-penumbra accuracy, the source portal and target portal can be switched in Listing 7 to produce a second set of anti-penumbra clipping planes. Note that these clipping planes will also need to be flipped so that they are facing the same direction as the first set. For performance reasons, it is a good idea to remove any duplicate planes that are created by adding the second anti-penumbra to the first anti-penumbra's clipping planes.

*Source code for anti-penumbra clip plane generation can be found at 17.13.1.*

### Clipping Portals

To find out if other portals are within the anti-penumbra, each potential portal needs to be classified by the each of the planes in the anti-penumbra. The classification algorithm can be exited early if a portal is found to be completely outside any one of the anti-penumbra planes. If a portal is found to be intersecting a plane, then it needs to be clipped by that plane using the polygon spitting algorithm (4.1.1), however, because only one side of the split is ever used it is faster to trim (17.8) the polygon. A portal may be trimmed several times.

### Anti-penumbra PVS Determination

The anti-penumbra PVS determination works by creating an anti-penumbra through the linking portal to target portal of two connecting portals. Any other portals visible within the anti-penumbra's area (which can only be in the directly neighbouring sector) become the new target portal(s). This process is performed recursively for any portals that are found within the frustum until there are no more portals found. If only a portion of the portal are visible within the anti-penumbra, then only that portion is used as the target portal, the rest are temporarily clipped away by the anti-penumbra.

```
FOR EACH [(source) leaf]
  FOR EACH [(source) portal in the source leaf]
    //Processes any leaf in the portal that is not the source leaf.
    [Use the source portal to determine what leaf it is connected with (the target leaf).]
    [Add the target leaf to the source leaf's PVS list]
  1. FOR EACH [(target) portal in the target leaf (or portal list)]
  2.   [Determine the target portals connecting leaf (destination leaf)]
  3.   [Add destination leaf to the source leaf's PVS list]
  4.   [Compute the anti-penumbra using the source and target leaf]
  5.   [Determine if any portals in the destination leaf are within the anti-penumbra]
  6.   [If portals where found within, do steps 1 to 6 treating any portals found as a portal list]
  END FOR
END FOR
END FOR
```

Listing 8. Portal culling using the anti-penumbra algorithm

Listing 8 demonstrates an algorithm that would determine a list of PVS for all leaves in the tree.

See 19.7 for an example using the PVS determination algorithm. Source code examples can be found at 17.13.

### Storing the PVS

PVS data is stored in byte arrays at each empty leaf node in the BSP tree. Therefore, one byte can indicate the visibility of eight leaves. That is: if the PVS set was 00010000, then only P4 would be drawn. One byte of PVS data is illustrated by Figure 23.

L1	L2	L3	L4	L5	L6	L7	L8
----	----	----	----	----	----	----	----

L = Leaf

IF (L1 = Set) [the leaf is potentially visible, otherwise it is definitely not visible.]

Figure 23. One byte of PVS

What happens if there are 80000 polygons? That's 10000 bytes per leaf (10000 bytes x 10000 leaves = 100,000,000 bytes = 100 Mb (average)). Many games levels today have well over 80000 polygons built and 100 Mb is a wastage, considering the type of information contained. Furthermore, to find out which leaves are visible, the entire 100,000,000 bytes would need to be processed every time the camera changes sectors just to determine if a few polygons are visible. Certainly a compression scheme is required.

Zero run length encoding (ZRLE) is used to reduce the amount of storage required to store the PVS at each node. ZRLE is able to take advantage of long runs of zeros, which are frequent in PVS data because most leaves are not visible (to that PVSs leaf).

See 16.4 for more information on ZRLE and 17.10 for ZRLE source code.

### *Enhanced Anti-penumbra PVS Determination*

In order to increase compile time performance some of the repetitive and expensive calculations can be pre-computed before PVS determination. A set (infront list) can be pre-computed for each portal to reject linking portals that are computationally cheap to determine. The following technique uses three layers of portal rejections to builds a portal infront list for quick rejections. Each layer uses the last layer (infront list level 1, 2 and 3) to improve infront list building performance.

### *Level 1 Infront List*

A portal can only be within the anti-penumbra if it is in front of the source portal (case 1). To simplify portal linkages, two-sided-portals are broken into two one-sided-portals which mean that portals that face each other, or face away from each other, are not visible to one another (case 2). Each leaf can have many portals but a one-way-portal only points to one leaf.

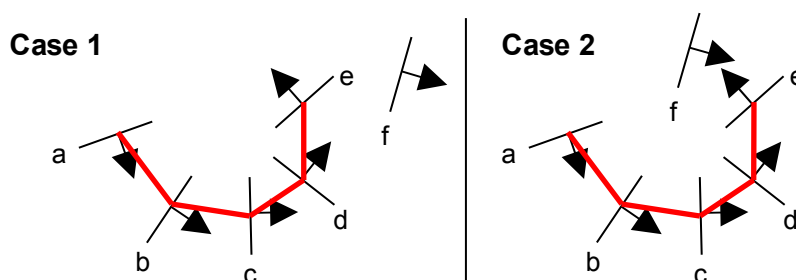


Figure 24. Portal in front of portals

Figure 24 shows two cases of one-sided-portals that are in front of the previous portal and face away from the previous portal. One-sided-portals are represented by lines with an arrow showing the direction the portal is facing (the normal). Portals a, b, c, d, e and f are linked to each other (by sector) in sequence. Specifically, a is linked to b which is linked to c which is linked to d which is linked to e which is linked to f. Thick lines show portals that are in front of each other. Thus, portal b is in front of portal a, portal c is in front of portal b, portal d is in front of portal c and portal e is in front of portal d. Although f links to portal e's sector, it isn't in front of e in case 1. In case 2, f is not in front of e because f faces e.

Information about which portals are visible from a particular portal can be stored at each portal in the form of a set (infront list). Classification can be used to determine if one portal is infront of the other.

```

FOR EACH [portal,  $p1$ ]
  FOR EACH [portal,  $p2$ ]
    IF [( $p2$  spanning or in front of  $p1$ 's plane) and ( $p1$  spanning or behind  $p2$ 's plane)]
      set [ $p1$ 's infront list,  $p1$ ]
    END IF
  END FOR
END FOR

```

Listing 9. Level 1 Infront list generation

Listing 9 shows an algorithm for pre-calculating which portals are infront of one another. Also, portals that are on the same plane as  $p1$  are not visible to  $p1$ , this includes  $p1$  itself. Source code for level 1 can be found at 17.13.3.

### Level 2 Infront List

Portals set in the infront list can be reduced into infront list 2 by looking at the graph sequence the linking portals make up, where graph sequence means a set of portals connected to one another in sequence. To be in a portal's infront list, each portal in the graph (of that portal) must:

- face away from previous portals that make up the graph sequence
- be in front of the previous portals in the graph sequence

For example:

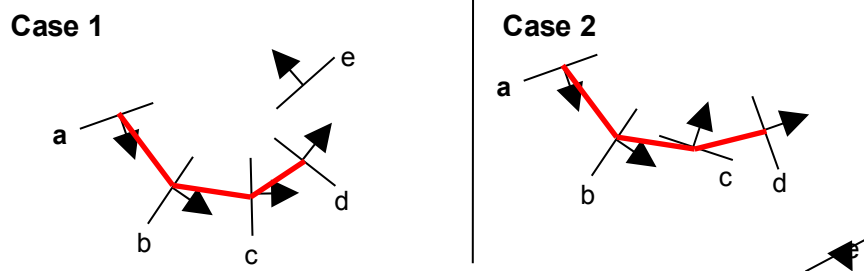


Figure 25. Portal in front of the previous portal

Figure 25 shows the portals potentially visible to portal, where a. a, b, c, d, e are connected to each other in sequence. In case 1, portal e is not visible to a because e faces a (and b); however, portal e would be in the infront list of c and d. In case 2, portal e is not visible to a because e is behind c and facing the wrong way to c; however, e would be visible to d. To work out the reduced infront list for a particular portal, the infront portals (of that portal) would be traversed and only portals that are in front of the portals sequence visited would be set in a new infront list (level 2 infront list).

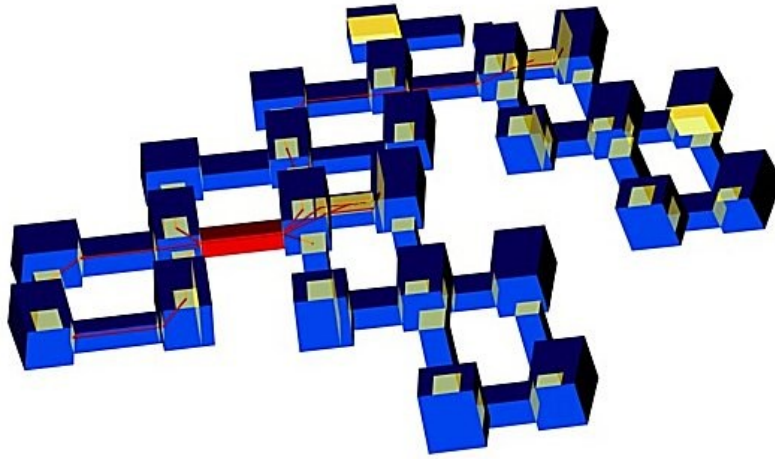


Figure 26. Level 2 infront list generation

Figure 26 shows an example of portals that are in front of the selected leaf's portals. The lines indicate the links between the portals.

Once the infront list 2 has been computed, it can substitute the infront test in the anti-penumbra PVS determination algorithm. Thus, each portal (except for the source portal) should be checked first to see whether it is within the infront list before generating the anti-penumbra.

*Source code for level 2 can be found at 17.13.4.*

### *Level 3 Infront List*

One way to reduce the level 2 infront list is to include only portals that are within the anti-penumbra of the source and target portal, where the target portal remains fixed for all child portals (of the target). Figure 27 gives an example of a level 3 infront list applied to the example given in Figure 26.

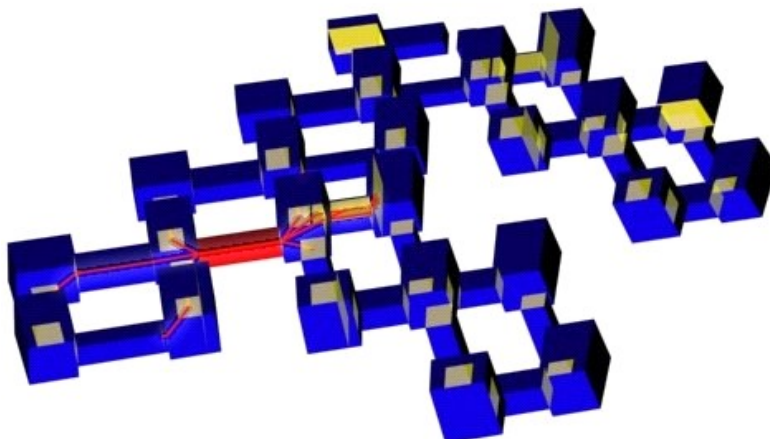


Figure 27. Level 3 infront list

Listing 10 gives an example of the algorithm for level 3 infront list generation.

```

FOR EACH [(source) portal]
  FOR EACH [(target) portal in the source portal's leaf]
    IF [target in front of source]
      [Compute the anti-penumbra using the source and target leaf]
      // The following for loop generally requires traversal
      FOR EACH [(destination) portal in front of source, target and previous destinations]
        IF [infront and portal in the destination leaf are within the anti-penumbra]
          set [p1's infront list, p1]
        END IF
      END FOR
    END IF
  END FOR
END IF
END FOR
END FOR

```

Listing 10. Level 3 infront list generation

The major difference between this algorithm and the anti-penumbra PVS determination algorithm is that the anti-penumbra is only generated for the first target portal in the sequence.

For example:

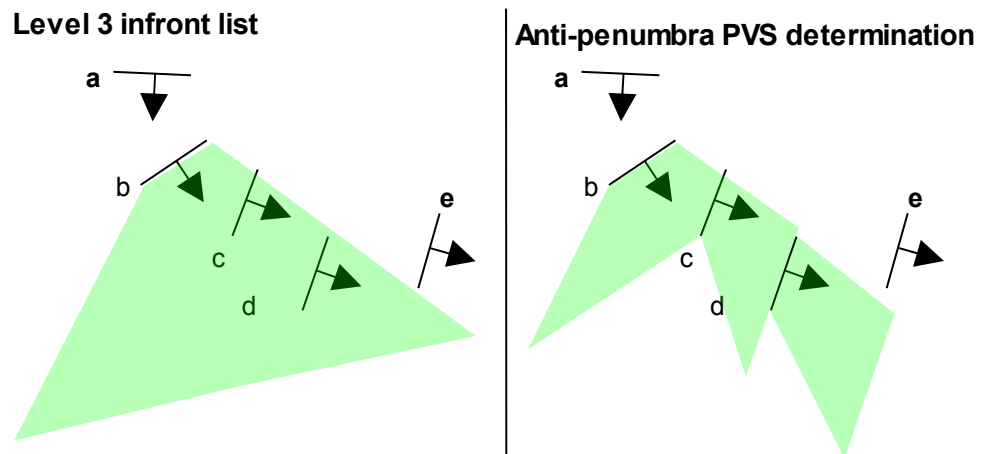


Figure 28. Listing 10. Infront list 3 verse Anti-penumbra PVS determination

Figure 28 shows the difference between level 3 infront list (L3) and the anti-penumbra PVS determination algorithm (APD). While L3 checks all portals potentially visible to portal a using the anti-penumbra generated from portals a to b, APD only checks the portals at the current node c and, then, re-creates a new anti-penumbra at d to check if e is visible. If there were more portals linking directly to a, then there would be more anti-penumbra created at a; however, the anti-penumbra would still only be generated one level deep, unlike APD. Using L3 to reduce the infront list more before applying the APD helps reduce calculations because no portal clipping is needed, and because many portals that normally would be tested against the anti-penumbra can get early rejections. As each portal infront list is updated, there may be less work for the next infront list generation because portals have been removed from the child portals infront list as Figure 29 shows.

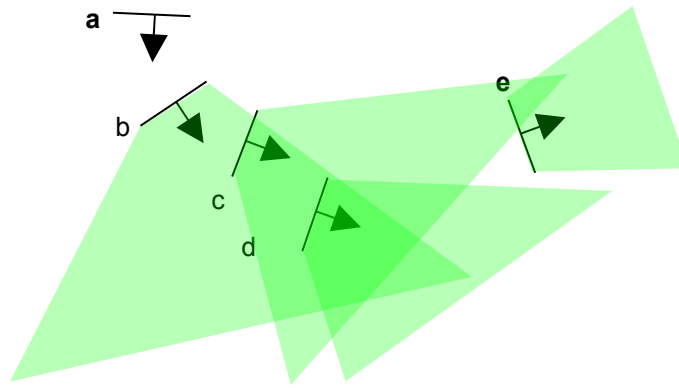


Figure 29. Level 3 early rejections

In Figure 29, if d was generated first, then e would not need to be tested against the c's and b's portals because it was rejected in d's infront list.

Source code for level 3 can be found at 17.13.5.

## Leaf Bounding Boxes

Each leaf in a PVS BSP tree can have a separate bounding box, so that if that leaf is not visible then all its related polygons can also be rejected. One way to compute the leaf bounding boxes is to clip each polygon by the BSP tree. The polygons that end up in a leaf can be used to compute the BB for that leaf. However (as Figure 30 shows), a better technique is to use the portals to compute the BB because if the camera can not see a particular portal then the camera cannot see the leaf that the portal belongs to.

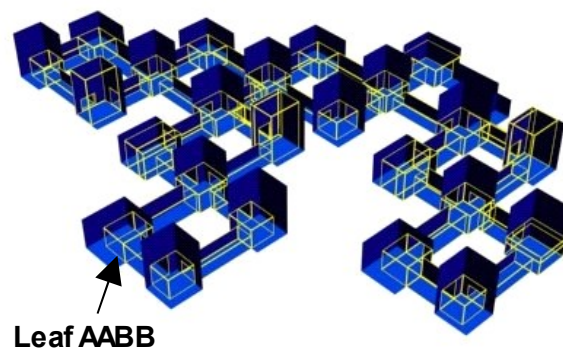


Figure 30. Leaf AABB (using portals BB)

Figure 30 shows a map with boxes used to indicate leaf bounding boxes (wire boxes).

## Detail

Details as shown by Figure 30 are small objects that are not large enough to be considered as part of the maps main structure. It is important to remove detail from the BSP creation stage because detail will:

- generally be mostly visible to the viewer at once; and
- increase the size of the BSP tree.

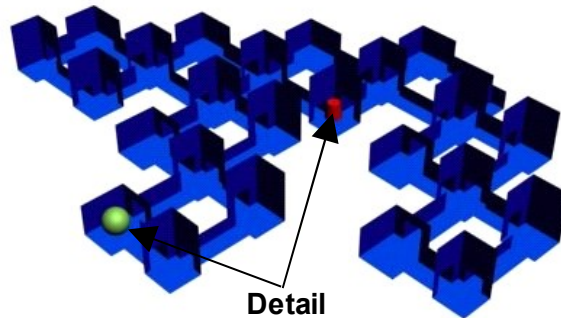


Figure 31. Detail

Detail can either be marked in the editor as detail or be detected and removed before traversal of the tree. To detect detail, adjacency information of vertices (or edges) needs to be flood-filled to find the size of a particular object. If the object is over some threshold value, then it is marked as detail; otherwise, it is added to the structure for BSP processing.

Once the tree is processed the detail can be clipped into the tree. Thus, the detail is sent down and classified at each node in the tree and generally stored in the leaves. Polygons that need to be split can be reformed in the leaf with an indication that they are shared by more than one leaf. Also, a bounding box can be computed for each detailed object so that object can be clipped if it is not within the view frustum.

### 4.3.2 Runtime

BSP with PVS provide an enhancement to BB BSP version of BSP trees by allowing for pre-determination of nodes that definitely cannot be visible from a particular sector (leaf). There are four major steps involved in finding the potentially visible polygons:

1. Perform collision detection.
2. Find the leaf the camera is in.
3. Uncompress the PVS and mark visible nodes (leaves).
4. Traverse the marked nodes, culling nodes that have BB not within the view frustum (similar 4.2).

### Collision Detection and Finding the Leaf the Camera is in

Traversal to find the leaf node is done as follows:



- If a solid node is reached, then the camera has moved into a solid area where it shouldn't be (inside a wall), which is a collision. Accordingly, a collision reaction algorithm would be run to put the camera back into an empty area. Consequently, collision detection can be determined in  $O(\log(n))$  time; where  $n$  is the number of polygons in the world (tree).



```

node = Root Node
WHILE [ node is not a Leaf ]
  IF [  $c \bullet n + d > 0$  ]
    //Camera is in front of this wall process the back wall
    node = node.back
  ELSE
    //Camera is behind this wall process the front wall sub tree
    node = node.front
  END IF
END WHILE

```

### Listing 11. Get leaf algorithm

55

Collision detection has a few more issues that need to be discussed in order to have a good collision scheme. Computer players (cameras) generally have a width and height associated with them. If the player has a width of zero, then the camera will be able to see through polygons the player collides with, because the camera view is wider than zero. In order to provide for width in a BSP tree, the various viewpoint around the player could be sent down the tree multiple times to make sure those points are also visible. For height the camera can simply be moved the height of the player below the viewpoint. However, while this strategy turns out to be good for height, it requires many traversals for width. A better solution (for width) is to project the planes in the BSP tree forward along their normals.

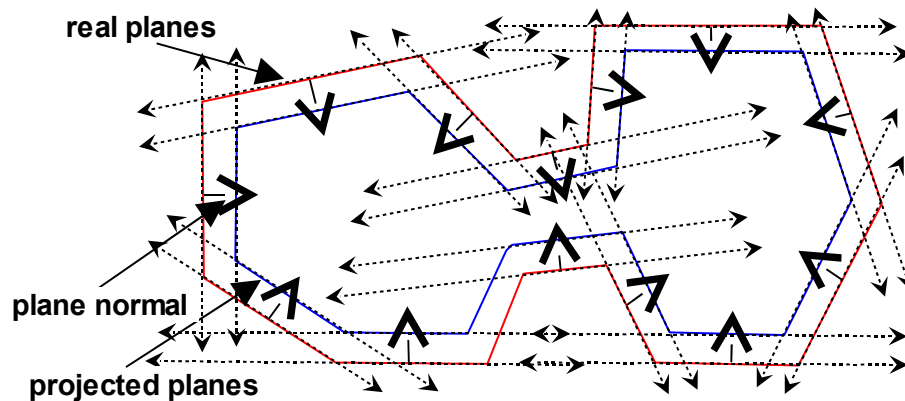


Figure 33. Map with planes projected forward along their normal

Figure 33 shows the planes in a map being pushed forward along their normal. The dotted lines are used to indicate that the planes effectively continue forever. Projecting a plane along its normal is achieved by changing the distance ( $D$ ). To move a plane forward increase  $D$ , to move back decrease  $D$ .

Another issue with CD is that it is a recursive process and the most relevant collision needs to be performed first. Thus, the planes involved in the collision need to be found and sorted by distance to the viewpoint, as the closest is most relevant. To find these planes, the tree needs to be traversed from a solid leaf and ends up at the root. Any plane found that is behind the current viewpoint, and in front of the old viewpoint, is a potential candidate. That candidate distance; from, the player (which is returned by classification) is compared against the closest distance and if the candidate is closer then it is made the new closest distance. Once the closest leaf has been found the collision reaction algorithm is performed using that leaf's plane. The process is repeated, until no collisions are found.

If the camera reaches an empty leaf node, the camera's leaf should be found using non-projected planes, as in rare occurrences using the projected plane can cause the viewpoint to cross portal boundaries in the tree (resulting in an incorrect leaf). Any detail at the camera's leaf needs to be collision tested. Detail collision testing can be performed using separate solid BSP trees for each object. A BB around the detail can be used to trivially reject collision tests. When no collisions are detected the algorithm the leaf PVS data is processed. Figure 34 gives an example of the camera's leaf (wire box) in a map for a particular viewpoint.

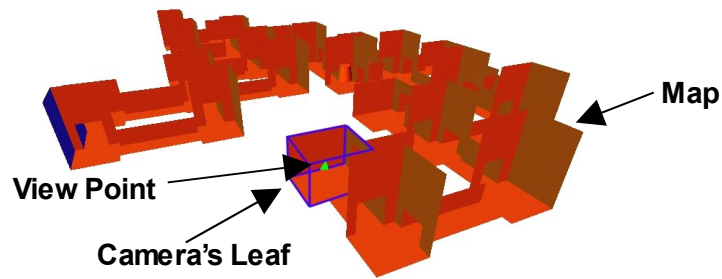


Figure 34. Camera's Leaf

Source code for collision detection and leaf finding can be found in 18.3.

## Uncompress and Mark

To uncompress the PVS data in the leaf, each byte in the set is looked at until all leaves have been visited. If a zero byte is found, then the following byte number (plus one and multiplied by eight) is added to the leaf offset. If the byte is non-zero, then each of the eight possible leaves for that byte is processed to see if it is potentially visible. As each byte is processed the leaf offset value is increased by eight. Listing 12 gives an example of the PVS uncompression algorithm.

```
offset = 0
WHILE [offset < Number of Leaves]
  IF [PVS byte equals zero]
    [Go to next PVS byte]
    offset = (PVS byte + 1) * 8 // + 1 is need because if zero there is at least one empty byte
    [Go to next PVS byte]
  IF [offset >= Number of Leaves] EXIT WHILE //Early exit
END IF

FOR [PVS bits 0 to 7]
  IF [PVS bit is set]
    [Record/Mark bit number + offset as potentially visible]
  END IF
END IF
offset = offset + 8
[Go to next PVS byte]
END WHILE
```

Listing 12. PVS uncompression

Figure 35 shows the potentially visible polygons contained in the current camera leaf.

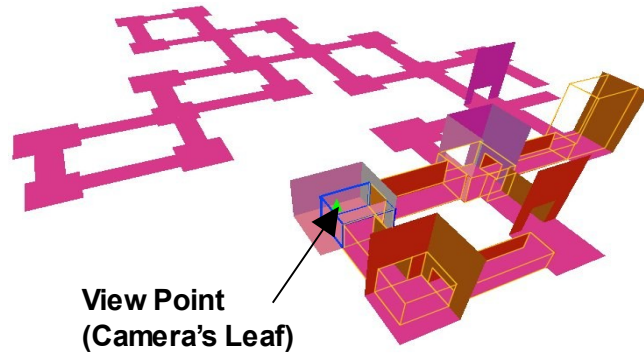


Figure 35. Potentially Visible Polygons

In Figure 35, the leaves BBs are shown as wire frame boxes. Note that the PVS data only includes leaf information, not information about the potential visibility of individual polygons. Therefore, polygons such as the ones that make up the floor are found to be potentially visible. In fact, the floor is the first polygon in the BSP tree for Figure 35; therefore, the floor is always found to be visible.

When the leaf data is uncompressed, leaves which have BB which are within the view frustum need to be marked with the current frame number. Marking a leaf includes marking all its parent nodes (recursively) and that leaf's detail. By marking the root node with the current frame number, leaf to root traversal only needs to check that the same frame number has not already been set. Note that Quake (1/2)'s engine incur an extra test by not marking the root node because also checking that the parent isn't the root node at every iteration. Assuming that the root node is set with the frame number, the marking algorithm for a particular leaf would look like Listing 13:

```

IF [Leaf's BB within view frustum]
  [mark leaf detail as potentially visible] //In some cases the detail could be drawn here
  current = leaf.Parent
  WHILE [ current.mark not current frame]
    current.mark = current frame
    current = current.parent //go to the parent of the current node
  END WHILE
END IF

```

Listing 13. Marking a leaf

One optimisation that can be made with the PVS set is not to record the camera leaf in the PVS set because this leaf will always be visible. Figure 36 gives an example of a tree marked with the camera leaf's PVS information.

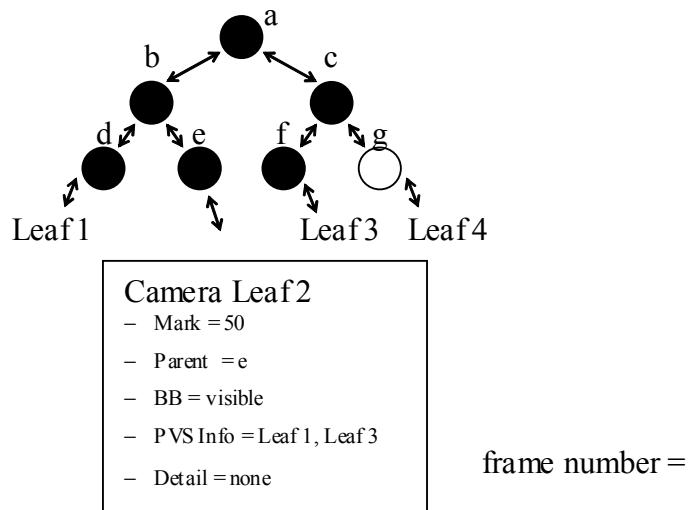


Figure 36. Tree marking

In Figure 36, nodes are represented by lowercase letters. Nodes that have been marked (with the current frame number 50) are shaded black. The records of the camera's leaf (2) are shown. Leaf 1 and 3 are visible to leaf 2, therefore these are marked in the tree. The marking algorithm for the particular situation in Figure 36 would first mark the root node a, with the frame number 50. Leaf 2 would then be marked which includes nodes e and b. After that, leaf 1 would be marked, which would only require d to be marked because b is already marked. Finally, leaf 3 would be marked which includes nodes f and c. Note that, for the worst case, no more than  $O(n)$  nodes need to be visited to mark the tree; where  $n$  is the number of nodes in the tree. However, for practical maps (for example Quake maps), the amount of nodes that need to be visited is much less, approaching  $O(\log(n))$  visits.

Leaves can be marked as each bit is uncompressed or the uncompressed data can be stored in an array and marked later. The advantage of the array method is that the PVS data only needs to be re-computed when the camera changes leaves (sectors).

## Traverse and Culling

Once the tree has been marked, tree traversal is basically the same as in 4.2, except only marked nodes are traversed. The traversal algorithm first checks if the node is marked with the current frame number. If the node is not marked then traversal stops as that node is definitely not visible, otherwise traversal continues. The rest of the algorithm is the same as Listing 6 (or Listing 5 if polygon ordering is important) as shown in Listing 14.

```
Function Bsp_TraverseGL ( node )
IF ( node exists AND node.mark equal to current frame)
  FOR EACH [testable frustum plane]
    IF [any node.volume.vertices on inside of the plane]
      IF [all node.volume.vertices on inside of the plane]
        REMOVE [plane from testable planes]
      END IF
    ELSE
      Stop processing node and its children
    END IF
  END FOR
  Draw node
  Bsp_TraverseGL node.back
  Bsp_TraverseGL node.front
END IF
```

Listing 14. Tree Marking and BB View Frustum Culling

If every node is marked and within the view frustum, then  $O(n)$  nodes need to be visited; however, traversal is more likely to approach  $O(\log(n))$  visits in practical maps because many nodes will be culled by PVS and frustum culling; where  $n$  is the number of nodes in the tree. Furthermore, the amount of visits required will always be less than or equal to the amount of visits required to mark the tree. Generally, since leaf finding (and collision detection) requires  $O(\log(n))$  and marking requires  $O(n) \geq O(\log(n))$  the overall cost of traversal has a maximum cost of  $O(n)$  that in practical cases approaches  $O(\log(n))$ .

An example of a scene that has been marked and culled by the view frustum is given in Figure 37.

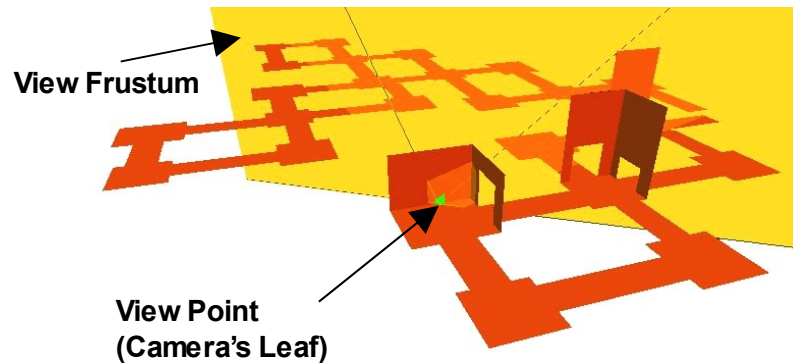


Figure 37. Marked and Culled

Figure 37 shows the view of the camera from a third person. Note that the view frustum is clipped to a certain distance so that the frustum fits into the picture. The entire floor is still shown as the floor is the first node in the tree. Polygons like the floor could be culled using an octree, or by projecting it on to a 2D BSP tree as these algorithms do not take 3D co-planes into account.

*Example code for traversal and culling of a marked BSP tree is given in 18.4.*

### 4.3.3 Conclusion (PVS BSP trees)

PVS BSP trees can cull large amounts of polygons by sacrificing memory for efficiency. Each node in the tree contains a compressed PVS list of potentially visible leaves. If a leaf is visible, then it's possible that its parent nodes are also visible. View frustum culling can be applied by marking all the potentially visible nodes and applying the BB BSP tree traversal algorithm on marked nodes (4.2.2). Although PVS BSP tree runtime algorithm can take up to  $O(n)$  culling invisible polygons, in general the algorithm requires closer to  $O(\log(n))$ ; where  $n$  is the number of nodes in the tree. Furthermore, the more polygons there are to cull the better the algorithm performs.

## 5 Materials and Methods

The basis of the thesis is on the creation of a “better” 3D HSR algorithm for static indoor environments.

### 5.1 Target Market

The HSR BSP tree algorithm to be investigated is primarily targeted at 3D interactive virtual environments, in particular, computer games engines. In most games engines, collision detection and ease of level creation are important components of the 3D engine, both being dependent on the HSR algorithm. The algorithm does have the potential for use in the military, medical/surgical and building construction industries in the form of efficient real-time 3D simulation of static indoor-environments. The target market is narrowed further to developers who require indoor static environments, otherwise known as “walk throughs”, such as first person shooters. Nevertheless, dynamic environments are not impossible with the HSR BSP tree; however, this algorithm should support the static part of the environments, more efficiently.

### 5.2 Design and Procedure

To determine if the new techniques VSL and SNC are feasible, the algorithm is implemented in the form of a program called the “study engine”. However, to come to any conclusion about the advantages/disadvantages of the new techniques, a comparison algorithm or “comparison engine” has also been developed. Before any engine can be built test data, in the form of world maps, is compiled into the appropriate forms using two BSP tree compilation algorithms, one for the “study engine” and one for the “comparison engine”.

The BSP tree compiler has options to switch between the creation of the comparison BSP tree or the study BSP tree. The compiler has various parameter settings such as the size of detailed brushes, so that more than one form of BSP tree can be compiled. The VSL SNC and HOC algorithms are able to be tested individually or in combinations. The data is saved into a binary file format that is read by the BSP tree engines. The generalized BSP tree compilation stages (for both engines) are listed in Table 1 (new stages to the traditional BSP tree algorithm are highlighted in bold):



## Process

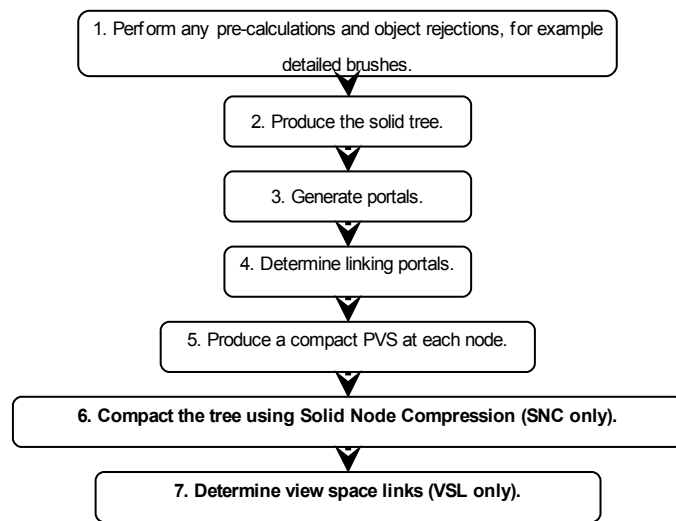


Table 1. BSP Tree Compilation Design

Creation of the comparison engine and the study engine goes hand-in-hand, as many of the components are the same. The comparison engine is not an exact replica of the Quake engines. Apart from copyright restrictions, there are several modern hardware enhancements that at least Quake 1 does not use; however, perhaps Quake 3 does. New concepts are taken into account, such as the improvement in 3D accelerated hardware and subsequent research papers. The study engine is exactly the same as the comparison engine except in the areas of investigation (VSL, SNL and HOC). Therefore, a comparison of the engines makes it possible to determine whether the algorithms are an improvement. The fundamental main-loop for the comparison and study engines' design are outlined in Figure 38 (the primary differences are highlighted in bold):

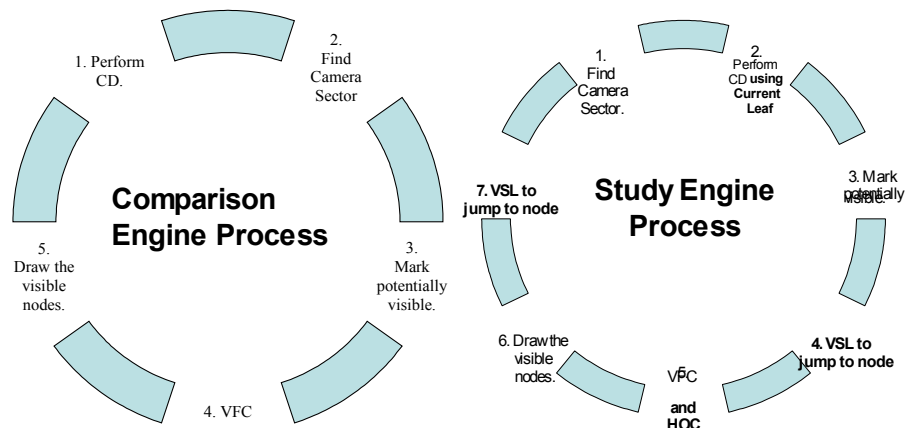


Figure 38. Comparison Engine Process / Study Engine Process

## 5.3 Software Used

Software used:

Software	Description
3D Studio Max (3DSMax) + plug-in software development kit (SDK)	For generation of maps and conversion to the BSP tree format, compilation.
Visual C++	For writing of the 3DsMax BSP tree conversion plug-in.
Digital Mar's D language	For the development of the 3D engines.
Excel	For generation of the statistics.
Word	For writing of the report.
OpenGL	For efficient access to 3D accelerated hardware without having to delve into hardware specifics.
Dig	To create the openGL context in D.

Table 2. Software to be used

Table 2 shows a list of the software used in creating the compilers and engines.

## 5.4 Data Analysis

The examination of every case for the algorithm is a NP-complete problem (as it is with most BSP algorithms).

It is therefore a good idea to choose a set of maps that can approximate the affects of these features: density, map size, uniformity, scalability, dynamic objects, detailed brushes and priority-ordering on the engine. In order for the engine to work the map needs to be:

- Indoor, meaning that most of the worlds are constructed from rooms, which prevent the entire world being visible at once.
- Solid, meaning that every polygon edge is connected with another polygon's edge, except in the case of polygons that could be seen from outside the rooms, where the camera would never be anyway.
- Static, meaning that the walls cannot move around, although the camera can move around the walls.

However finding/producing such maps proved to be a difficult task due two two reasons:

- All free maps found had geometric error, causing the map to leak
- Too little time was allocated for creating/fixing maps complex enough to show the real power for the engine.

Results are logged into a text file every time the BSP tree is compiled or run. The text file is in a form that is easily transferable to a spreadsheet application, so that graphs and charts can be produced. For each stage of the BSP cycle, times are logged into individual txt files for the particular map being processed. Times can be stacked so that each stage can be broken down into sub-stages.

The results of the two engines are compared in a variety of ways by:

- using different types of maps, to determine the effect on performance of using:
- scene densities (details);
- map sizes;

- uniformities;
- amounts of polygons;
- dynamic objects; and
- brushes;

and by getting performance information from timing of processing, such as:

- frames per second;
- algorithmic efficiency;
- collision detection;
- compilation;
- map loading;
- runtime.

The experiments have been run multiple times in order to reduce the noise factor that is caused largely by multi-threading and caching of the CPU.

## 5.5 Limitations

Hidden surface removal is a broad topic; therefore the focus is constrained to BSP tree HSR for solid static indoor 3D worlds. Other limitations and restrictions include the resulting paper, engine and compiler.

### 5.5.1 This Paper

To limit the potential size of the thesis, the discussion on the algorithms implementation is limited to the new approaches used in the BSP tree HSR algorithm. A brief discussion is given on the overall implementation approach of the engine and overflow topics are either explained in the appendices or left as a research exercise for the reader.

### 5.5.2 The Engine/Compiler

The resulting engine is not fully completed; it is constructed to tackle the specific problems that are being investigated.

In an attempt to reduce the scope of the engine/compiler development these restrictions are followed:

- *Optimisation of the two algorithms is high level, focused on the HSR aspects of the engine.* Therefore, performances rates are not expected to be anywhere near those of the Quake games. However, the benefits (if any) of the applied optimisation technique should be obvious.

- *“Eye-candy” does not contribute to the resulting data analysis, however the algorithm should allow for such things.* Issues such as optimisation aspects (that is, scene graphs, efficient file loading) and special effect (like, bump mapping, light maps, etc...) are only be included when they can be done with little effort. Furthermore, many scenes may have more then one rendering pass (of the polygons) to achieve some special effect (like, stencil shadows). It is important to optimise the algorithm for “eye-candy”, for example, the HSR shouldn’t be necessary for each pass. However, the base performance rates are determined by looking at a single pass without textured or lit polygons and without special effects. Furthermore, the base performance rates allows the reader to make comparisons with other works that use the same strategy, such as James (1999) and Teller (1992b).
  
- The code is kept high level (for example in a language such as C++ or D), and does not delve into low-level languages such as ASM. Keeping the engine code high level helps simplify development and make the procedure easier to explain.
  
- *Be supported from a subset of known works.* It may be possible to hunt down, read and understand every paper on HSR with BSP trees; however, realistically it is impractical. Furthermore, for business reasons many 3D engine developers have kept their HSR solutions proprietary. Therefore, the comparative algorithm and related discussions is limited to a subset of works that is comparable to current 3D HSR algorithms.
  
- *Time constrains have an effect on the size of maps that can be processed though the engine’s compiler.* Even with the powerful CPUs of today, BSP map compiling is a long process, which can take days to compute.

## 6 Implementation Specifics

The test code was built in two stages in two different languages. The compilation stage is an export C++ plug-in for Discreet 3Dsmx that interprets the geometry, compiles the BSP tree and outputs the result to an mbs file. An mbs file is a file format invented for the specific purposes for storing the comparison and study engine BSP data for this thesis.

It is important to reiterate that the comparison engine is not meant to out perform any of the Quake engines. It is merely used to compare the study algorithms on a level playing field. Furthermore, much of the comparison engine code and ideas are reused in the study engine code. There are a lot of innovations in the source, some of which are discussed:

### 6.1 Compilation

The compilation process can take anywhere from a few seconds to a few minutes depending on the map complexity.

#### 6.1.1 Detail Gathering

Detailed polygon groups are clustered together by scanning neighbouring polygons. A list of vertices that share edges is used to trace the surface of each object. Any object that is within the bounds of the threshold (generally 512x512x512 units) detail. Anything left over is treated as structural planes to be used in the BSP tree generation. Detail also contains a BSP tree of planes, used for collision detection and BB used for early rejection during culling.

Detail is stored in the leaves of the tree. Detail that spans more than one leaf is split and stored in two groups, called single group and shared group. The shared group contains polygons that span more than one leaf, while the single group contains only polygons that are in the current leaf. At run-time, the shared group can be checked to make sure that polygons are not drawn more than once for a particular frame.

#### 6.1.2 On Plane Grouping

Planes are grouped together before BSP generation and while generating the tree. A map data structure is used to find polygons that share the same planes. The advantage of computing most of the planes before BSP generation is that it reduces the amount of repetition. Thus, the same plane is not tested more than once in polygon picking.

A note of interest is that while Quake 3 (radiant) marks polygons that are on the same plane with the same index before BSP processing, these polygons are still added into the queue as single polygons. However, if polygons are grouped by planes beforehand  $n$  is reduced when it comes to polygon picking which requires  $O(n^2)$ ; where  $n$  is the number of polygons to select from in the pick.

Planes that are almost the same are treated as the same plane by reducing the accuracy of comparisons. Furthermore, floating point errors can cause problems when computing two planes that are almost the same. For example, one such problem occurs when a polygon is found to be on two different planes. Essentially the user will not notice the difference between two planes that almost lie on the same plane.

### 6.1.3 Split minimisation and balancing

Picking uses cost analysis to compare different polygon choices against one another. These operations are factored into the cost for each polygon:

- How many potential splits a polygon could cause (least crossed).
- How much a polygon could be potentially split (most crossed).
- The amount of polygon fragments.
- Whether the polygon is axis aligned or not.

Although least-crossed criterion, most-crossed criterion and most-crossed criterion tie break least-crossed criterion were experimented with and provided as options in the study compiler. Generally, least-crossed criterion tie break most-crossed criterion was found to provide the smaller trees and so was used for all the tests. The initial least-crossed criterion collects a list of potential ties. If by the end of the process there is more than one tie, the polygons in question (the list of ties) are tested against most-crossed criterion.

Tree balancing was found to affect performance and tree size. Too much weight on balance and the tree would grow large; however, a small weight towards balancing did appear to reduce the tree size while improving balance. This improvement is believed to be so; because each time a group of polygons are split more-evenly there are less possible splits than if the polygons were more-one sided. A balancing factor of 8% was found (on average) to generate the best results in the maps tested. Rather than using a separate stage, the amount of balance can be performed with a simple modification to least-crossed criterion to improve performance.

Putting the polygons with more fragments at the top of the tree reduces the tree size for the simple reason that less polygons are taken out of the list of potential splitters. However, favouring many polygons does mean that there are fewer polygons to be BB culled during tree traversal. Nevertheless, it is possible to use a second HSR algorithm such as a 2D BSP tree (after projecting the plane on to the screen) or octree or to perform more detailed culling on these polygons.

Picking axis aligned polygons is part of Quakes picking strategy, however Quake favours axis aligned walls over axis-aligned floor/ceiling rather than favour all axis-aligned polygons. Axis aligned polygons are favoured because:

- 1) The calculations are less susceptible to floating point errors.
- 2) The three possible planes are more likely to divide the scene up more evenly because there is less likelihood of polygon collision.
- 3) Axis aligned polygons are generally walls which are good splitters.

Polygons that are split are stored as the original polygon in the tree which means that there are less polygon fragments to store, thereby improving efficiency.

### **6.1.4 PVS generation**

PVS generation is by far the longest part of the generation process, which can take more than 99% of the processing time. Therefore, it is a prime area for optimisation. However, many optimisations that can be applied to PVS generation reduce the accuracy and, thus, increase the amount of polygons in the PVS set. As explained in 4.3.1, PVS is broken up into 3 stages.

#### **Level 1**

Apart from doing trivial visibility rejection of portals, level 1 also computes a priority-order for portals that will be processed by level 2. Priority ordering portals helps improve efficiency when processing the portals. In the general case, each time a portal is processed the result can be used to reduce the amount of work required to process other portals. Therefore, by processing portal visibility by order of complexity, with the simplest being first, efficiency can be improved.

The more portals that the target portal can see add to the complexity of that portal because there are more possible pathways to check. Furthermore, the larger the number of close portals (i.e., portals that are visible through a low number of gateway portals) visible to the target portal the lower the chance of culling because that portal probably has a wide visibility. While computing the distance and size of portals that are close to the target portal is expensive it is relatively cheap to use local portals (portals that are directly visible to the target portal). A value of +1 was given for each potentially visible portal and +1 for each local portal at each portal; so effectively local portals are worth twice as much as other visible portals and so have higher priority when processing nodes. Source code for level 1 can be found at 17.13.3.

## Level 2

Level 2 performs basic anti-penumbra culling as described in 4.3.1. At the cost of accuracy, when a known portal is found or some threshold depth is reached the algorithm switches over to using the already computed list of visible portals. This technique was used because the program was found to be too slow (in the order of days) to recompute/compute visibility when the depth got large, as the combinations of nodes can in some instances grow exponentially. Refactoring helps remove some of the portals that this optimisation misses.

Refactoring is essentially a way of merging visibility information found in neighbouring portals into the target portal. Thus any portal that a local portal can't see, the target portal can't see either. Refactoring is performed before and after anti-penumbra clipping is performed.

In refactoring, the visibility portal sets from the locally visible portals are combined using the logical 'or' operation (not forget the visible portals itself) into a new visibility portal set and then that list is processed using the logical 'and' operation with target portal. The refactoring algorithm for portals looks like this:



```

Make new visible list vis

FOR EACH [portal, t]
    vis = copy of first visible (in target) portal's infront list

    FOR EACH [visible portal in t except the first one, p]
        vis = vis OR p.infront
        vis [p] = visible //Or portal itself into the list
    END FOR

    t.infront = vis AND t.infront

END FOR

```

Listing 15. Refactoring a portal

The refactoring algorithm can be run many times, each time improving accuracy a bit more as each portal's visibility set is reduced. The cut off point for the number of times refactoring is done can be some fixed threshold, or when the portals' visibility lists stop improving. The former was chosen in the given implementation because it is cheaper computationally to implement. A default of 5% of the number portals seemed to work well on the maps tested.

Level 2 computes priority-order for level 3 in much the same way as level 2, except local nodes are not included as they appear to be of less value at this stage. A lookup table for each byte in the sets used to speed up the calculation of the amount of visible nodes in a set. Thus, rather than workout how many bits are set in a particular byte using shifting, the number is looked up in a 256 sized array.

*Source code for level 2 can be found at 17.13.4.*

## Level 3

Level 3 performs the more complex anti-penumra clipping by taking depth into account. Level 3 also performs compression of the PVS information. Using the priority information generated per portal in level 2, priority per leaf is computed. Thus each leaf's priority is the sum of all its local nodes' priority. Although sorting leaves helps reduce the time required by level 3, it is by far the most expensive operation of any in BSP tree generation, in some instances taking more then 90% of the total time to complete.

Level 3 also performs refactoring per-leaf after the sets have been worked out, which is essentially the same as refactoring per portal except that it is applied to the leaf-visibility information. A default of 5% is also used for leaf-refactoring.

Other optimizations include not clipping the portal when it contains no other visible portals and reusing information computed from other leaves.

*Source code for level 3 can be found at 17.13.5.*

### **6.1.5 Static State Graph**

Every state change in 3D acceleration has the potential to reduce performance so the objective of a static state graph is to reduce the amount of state changes. The static state graph (scene graph) is compiled in such a way as to minimise state changes.

Each type of state (for example texture, ambient colour, diffuse colour) is called an element. Elements with fewer state changes are placed at the top of the tree. Elements also have priorities so that slower state-changes such as texture-mapping can be given higher priority in the tree. However, elements with only one state always go at the top of the tree as these states are static.

Note that elements with only one state could be essentially removed from the tree and simply initiated globally; however, that was not done in this implementation to keep things simple. Multi-texturing and multi-passing can be handled by scene graphs although that is not shown in the implementation. Polygons are stored in the leaf (a dummy node) of the tree depending on what elements they are composed of.

## **6.2 Runtime**

The runtime stage is the engine, written in D, which loads the mbs file, allowing the user to move around and perform various operations to test the engine. A description of how to use the engine is provided in section 20.

### **6.2.1 Occlusion**

Polygons are drawn in order (closest to furthest) to aid with hardware occlusion of individual polygons. If hardware occlusion was not being used the back/front test could be removed in the traversal code. In order to perform occlusion while processing the tree without reshuffling the states, a cheap (for example no lighting, blending, tessellation or texture mapping) version of the object is rendered for occlusion, with the depth test set to less than. If that object is found visible by the occlusion test then it is rendered again using the static state graph. The depth test is set to equals to avoid overdraw on the second pass. For a multi-passing system, the first pass for occlusion could be rendered visible for a pass that does not require expensive state changes, essentially getting occlusion testing for free.

## 6.2.2 Pruning

Rather than perform BB tests on small groups of nodes of which there is almost no benefit in doing, nodes with fewer than some threshold (for example 15) children are processed using a simpler function. The rationale being: hardware (or drivers) should be able to clip individual or small groups of polygons reasonably efficiently. Nodes with less than that threshold of children are flagged as being too small for BB tests and tested at each node. The algorithm for generating the flags for small containers is shown in Listing 16:

```
Function Mark_Small_Containers(node)  
  count = 0  
  IF [node]  
    count = Mark_Small_Containers(node.back) +  
            Mark_Small_Containers(node.back)  
    IF [count < threshold]  
      Set small container flag in node  
    END IF  
  END IF  
  RETURN count
```

Listing 16. Mark Small Containers

While this pruning does incur a small extra test at every node, the extra BB tests required without the pruning can be more expensive. Once an object is found to be completely inside or is flagged a different function is called which processes the rest of the nodes without testing.

*Source code for marking small containers can be found at 18.6.*

## 6.2.3 Static State Graph

At runtime the polygons that are visible are not rendered when they are instead added in the dummy node (leaf of the tree) of which they belong. The leaf's parent elements (nodes) are traversed (recursively) and marked up to the root, if that leaf has not been already by another polygon in its list.

The tree is then traversed from the root, going down the marked elements (states) that are enabled (active) first and then the other states. Each time a state is visited that state is made the active state which means that it is enabled in OpenGL™, thus even across frames state changes can be minimised. When a leaf is reached, all the polygons in that dummy node are rendered.

Although the dummy node could be cleared at this point it is not because there may be multiple passes of the tree to render each element in a different way. The dummy node is cleared on in the next round of marking. In this way state changes are minimised.

## 7 View Space Linking

A link is simply a pointer to the top most visible node from the last leaf the camera was in. When a general BSP is traversed, the traversal starts at the top of the tree. Traversal from the top of the tree means that there are nodes that always must be traversed which amounts to extra calculations. Furthermore, since the camera generally stays around the same nodes it is bound to repeat the same calculations between frames. View Space Linking helps reduce the amount of leaves needing to be traversed from the top of the tree by providing shortcuts (links) in the tree leaves.

### 7.1 Link Generation

Link generation occurs after the tree has been build and the PVS have been determined. The process involves working out the first point in the tree where traversal would be required to travel down two paths. Thus, the steps required to determine links for each leaf in the tree are:

1. Pick a leaf
2. Mark all the PVS leaves parent nodes (recursively) that are visible from the current leaf.
3. Traverse the tree to find the first node which has two children that are marked. This is the link node.
4. Store the link node for that leaf at that leaf.
5. Repeat 1 - 4 for every other leaf not processed yet.

The algorithm for link generation is shown in Listing 17.

```

FOR EACH [leaf, l]
  //Mark visible leaves
  Increment [ frame ]
  Mark root node
  FOR EACH [visible leaf in l]
    Mark [visible leaf's parent nodes with frame]
  END FOR

  //Traverse the tree
  node = Root Node
  WHILE [node node is not a Leaf]
    IF [node.back.mark equal to frame]
      IF [node.front.mark equal to frame]
        EXIT LOOP
      END IF
      node = node.back
    ELSE
      node = node.front
    END IF
  END WHILE
  l.link = node
END FOR

```

Listing 17. Link Generation

Source code for Link generation can be found in 17.11.

Figure 39 shows an example of a link node that has been generated for Leaf 2.

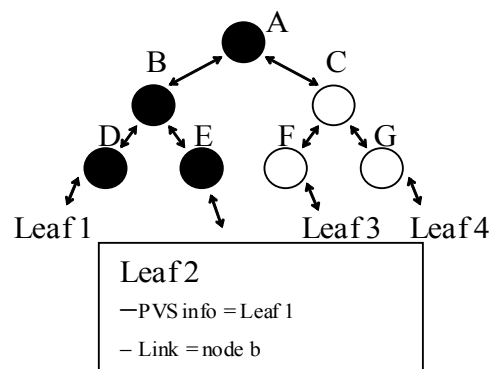


Figure 39. Link Generation

In Figure 39, as Leaf 2 is the leaf being processed some of its fields are shown. To generate a link for Leaf 2 the steps would be as follows:

1. Leaf 2 parents (up to leaf A) are marked (indicated by the dark shade) which means that nodes E, B and A are marked.
2. Then leaf 1 is marked as it is A leaf that is visible to leaf 2 which means node D is marked.

3. The tree marked leaves are traversed until the first node with two marked children (B) is found. That is B becomes Leaf 2's link node.

## 7.2 Rendering VSL

At run-time the node link is contained in the camera's leaf. The parents of the root node are rendered without the general BB testing or marking as these tests are almost certain to be true if the tests were performed. The tree is marked from the link node (as opposed to the root node) and the tree is traversed from the link node.

Figure 40 shows how traversal would occur if the camera was in Leaf 2

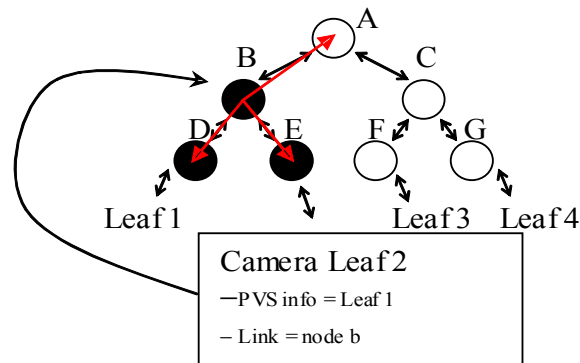


Figure 40. Link Traversal

In Figure 39 the only marked nodes (indicated by the dark shade) are shown by the shaded circles. In this instance the BSP algorithm would first traverse up the tree to render leaf A; then the algorithm would mark all the potentially visible leaves up to leaf B. Finally the tree would be traversed from leaf B, performing the BB culling. In this instance node linking has saved (among other overheads) a BB test at B (which would be true anyway) and a visit to C which would be rejected because it was not marked.

The algorithm for link traversal is simply a modification to the rendering stage of BSP rendering.

```
//Draw parent nodes
node = [Camera Leaf].link.parent
WHILE [ node exists]
    Draw node
    node = node.parent
END WHILE

//Uncompress and mark potentially visible polygons
Uncompress PVS
Mark root node //This is where marking will terminate
FOR EACH [visible leaf in I ]
    Mark [visible leaf's parent nodes with current frame]
END FOR
```

Traverse marked nodes in tree starting from [Camera Leaf].link and draw visible nodes

The source code for link traversal can be found at 18.5.

### 7.3 Other

Collision detection is still required to start at the top of the tree as there are solid nodes that may be found to cause collisions above the link. However, if VSP is combined with SNC, the link can be used to improve collision detection efficiency.

## 8 Solid Node Compression

Solid node compression works by pushing solid nodes to the bottom of the tree. Without solid nodes the tree becomes more balanced and can, therefore, be sorted sequentially in arrays allowing for reasonable memory savings. Furthermore, SNC reduces the amount of nodes that require BB testing as only nodes that are worth testing (nodes with two children) will be tested. Thus nodes with one child are removed from the tree and stored in the tree leaf and only used for collision detection.

### 8.1 Tree Compression

Tree compression is a three stage process.

#### 8.1.1 Stage 1 - Get Compact Root

The first stage involves finding the first node with more than two children, which becomes the root node. Once the root node has been found the nodes that previously were its parents are merged into that node. Thus, all the polygons in the parent nodes are added to the list of polygons at that node. The BB from the top-most previous root node becomes the BB of the new root node. Collision information is pushed down the tree nodes using a linked list. Thus, every plane from the solid node is appended on to the linked list which is passed down the tree.

The steps required to get the compact root are:

1. Find the first node from the root node that does not have a solid leaf. That node becomes the new root node. The following code shows the algorithm to get this node:

```
newroot = [root node]
WHILE [newroot node back is a solid and front is not a leaf]
    newroot = current.front
END IF
```

2. Use the old root node's BB for the new root node as the following code shows:

```
newroot.BB = [root node].BB
```

3. Change the parents (recursively) of the new root node into a collision info linked-list and copy parent node polygons into the newroot node

```
parent = newroot.parent
WHILE [parent]
    [append parent's plane to the collision info linked list]
    [append parent's polygons on to newroot]
    parent = parent.parent //Go to the parent of the current parent
END WHILE
```

4. Remove all nodes before newroot, making it the root node.



Source code for getting the compact root can be found at 17.14.1.

## 8.1.2 Stage 2 – Remove Solids

Stage 2 involves merging nodes which have a solid child (back node) into their front node or parent node (if front is a leaf). Collision information is prepended on to the front of the linked list and pushed down to the child. Note that because a particular parent node may have more than one child, a node in the linked-list may gain more than one parent, that way redundancy is reduced. When a leaf is reached the front of the link list that has been passed down is used as the link as Figure 41 shows:

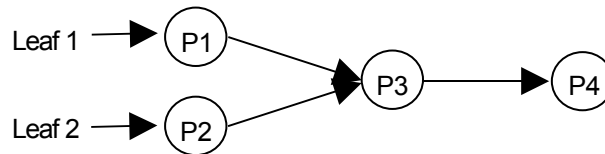


Figure 41. Lead Collision information

Figure 41 shows the collision information for leaf 1 and 2. Each collision node contains a link and the solid collision plane.

The remove solids process is recursive as follows:

If the current back node is solid then:

- the node's plane is prepended on the linked list that is made the front of the linked list;
- if the front is a leaf, then the objects in the node are added to the parent node's objects; otherwise, they are added to the front nodes objects; and
- the solid node is unlinked from the parent of node by replacing it with its front node

If the back node is not solid then the back node is traversed.

If the front node is not a leaf then the front node is also traversed.

Listing 18 shows the pseudo code for the solid removal algorithm.

```
Function Remove_Solids_FromBSP(node, link)

back = node.back
front = node.front

IF [back is a solid node]
    parent = node.parent

    [prepend node's plane on to link]

    //Find out which side
```

```

IF [front is a leaf]
    [set front nodes collision info to link]
    [append objects in node to the parent of node]
ELSE
    [append objects in node to the front of node]
END IF

//Unlink solid node from parent
IF [parent.back equals node]
    parent.back = current.front
ELSE
    parent.front = current.front
END IF
ELSE
    Remove_Solids_FromBSP(back, link) //Traverse
END IF

IF [front not leaf]
    Remove_Solids_FromBSP(front, link) //Traverse
END IF

```

Listing 18. Remove Solids From BSP

The initial parameters passed into Listing 18 would be the node found with the get compact root function and the linked list generated in the compact root function.

*The algorithm for removing solids is shown in 17.14.2.*

### 8.1.3 Stage 3 – Compress

Stage 3 involves packing nodes into groups of nodes. The algorithm works out the maximum amount of nodes that make up a balanced tree for each group. The compilation algorithm navigates the tree row by row. While no leaf has been found that node is added to the current group. Once a leaf has been found for a particular row each non-leaf-node on the rest of the row is placed into a new group and the compression starts again for those sub-trees. Each node-group or leaf is linked into the tree using a list of “head nodes”, where each head node can point to a leaf or node as Figure 42 shows. The sign bit in the headnode may be used to indicate whether a head node is a leaf or a group.

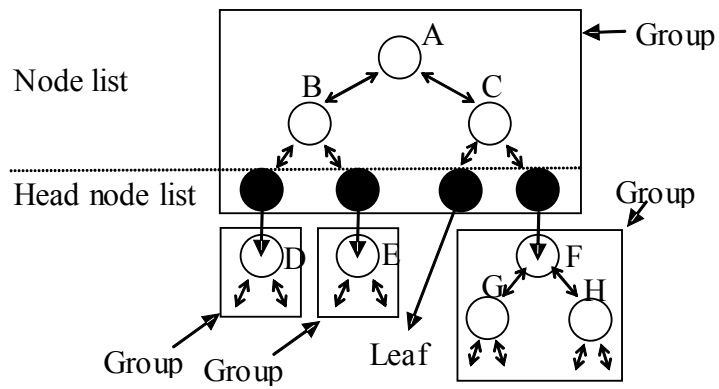


Figure 42. Compressed tree

In Figure 42 head nodes are represented in black and nodes in white. The links without nodes at the bottom of the tree are used to indicate that the entire tree has not been shown. A group is made up of two arrays, a node list and a head node list. The arrays are sorted in row order, so the node item in group 1's node list is A, follow by B and C. Group 1's head node list contains 4 groups which point to group 2, group 3, leaf 1 and group 4 respectfully. Listing 19 gives an example of an algorithm that can be used to compress the tree.

```

root.node = [root node found by get compact root]
[push root on to the headQueue]
WHILE [headQueue not empty]
  cgroup = [pop headQueue] //get the head node by popping the queue
  [push cgroup.node on to nodeQueue]

  WHILE [nodeQueue not empty]
    current = [pop nodeQueue]
    IF [current is not a Leaf]
      [add current to cgroup's nodelist]
      [mark current] //so that it is not reused
      [push back of current on to nodeQueue]
      [push front of current on to nodeQueue]
    ELSE
      //The leaf has been found therefore there the rest of the queue must be processed as new groups.
      [clear nodeQueue]
      FOR [each nodes in nodeQueue, n]
        IF [n.back is marked]
          IF [n.back is Leaf node]
            [add n.back as leaf to cgroup's headlist]
          ELSE
            [add n.back as new group to cgroup's headlist]
            [push new group on to headQueue]
          END IF
        END IF
        IF [n.front is marked]
          IF [n.front is Leaf node]
            [add n.front as leaf to cgroup's headlist]
          ELSE
            [add n.front as new group to cgroup's headlist]
            [push new group on to headQueue]
          END IF
        END IF
      END FOR
    END IF
  END WHILE
END WHILE

```

Listing 19. Algorithm for compressing the tree

Listing 19 shows how two queues can be used to create the compressed BSP tree. Queues are useful because they allow the tree to be traversed in row order. The headQueue is used to store the groups of nodes waiting to be processed while nodeQueue holds the nodes being looked at for the next group being created. The tree's root node is added to the queue to get the process started. While non-leaf nodes are found in the queue, those nodes are added to the nodeQueue. When a leaf node is found head nodes (leaf or group) are created for each of the node's children that are currently in the group. Any new groups that are added to the headQueue to be processed in a subsequent pass.

*The algorithm for stage 3 is shown in 17.14.3.*

## 8.2 Tree Traversal

Traversal of the SNC tree is much the same as general BSP tree traversal except that:

- parent and child locations need to be computed as these are no longer contained in the tree;
- collision detection is done in the camera's leaf;
- the current node and group need to be retained as the tree is traversed termed a head node pair (HNP);
- each group needs a head node which points to the group's parent group; and
- leaf nodes need to store their parent nodes and head node called head node parent iterator (HNPI).

See 18.7.1 for an idea of what the data structures used for SNC traversal look like.

### 8.2.1 Computing Parent locations

Finding the parent of a node is necessary for marking the tree. If the current HNP node property is the first node in the current HNP group property then, the parent node and group is found by the following formula (Listing 20):

```
parent group = [from the parent group property stored at the current head node]
node offset in group = ([of location of current's head node in the parent group] + [the amount of nodes in the
parent group] - 1)/2
```

Listing 20.

Otherwise, the parent node can be found by subtracting one from the node offset and dividing by two. The algorithm to find the parent node from a HNP looks like Listing 21 shows:

```
Function Node_GetParent ( current , parent )

//get the Head Node Parent from the given head node

offset = current.node
IF [ offset = 0 ]
    //The group needs to change
    offset = [ current.phnode offset location in current.phnode.parent ] -
current.phnode.parent.nodelist.length

    parent.phnode = current.phnode.parent
ELSE
    //The group stays the same
    parent.phnode = current.phnode
END IF
parent.node = (offset - 1) / 2
```

Listing 21. Get Parent Node

Where:

- the current node is the node to get the parent off,
- parent is the resulting node,
- nodelist is the list contained in the headnode,
- phnode is the headnode which gives the location of the group,
- node is an offset for nodelist and used to represent the current tree node.

Note that the algorithm above is a simplified version of the actual code which uses pointers instead of offsets. Furthermore, because parent node traversal is linear in nature the same variable for current can also be used for parent. More detailed source using that approach can be found in 18.7.2.

### 8.2.2 Computing Child locations

Getting the back and front nodes for a current node is necessary for traversal. The back node in a particular group for a particular node can be computed using:

$$\text{index} = \text{backnode} = [\text{node offset}] * 2 + 1$$

The front node can be computed using:

$$\text{index} = \text{frontnode} = [\text{node offset}] * 2 + 2$$

If the range exceeds the size of the node list in that group then the result is a headnode can be found by using the value as an index for that groups head list but subtracting the size of the node list in that group; that is,  
head node = index – nodelist.length.

The head node can either represent a leaf (for example, the sign bit may be used to indicate a leaf) or otherwise a group. If the head node is a group, then the node is the first node in that group. Therefore, the algorithm for determining the front and back nodes is shown in Listing 22.

```

nodeindex = [node offset] * 2 + 1 //back

IF [nodelist.length > nodeindex]
    back.node = group.nodelist[nodeindex]
    back.hnode = group

    nodeindex = nodeindex + 1 //front
    IF [nodelist.length > nodeindex]
        front.node = group.nodelist[nodeindex]
        front.hnode = group
    ELSE
        headnode = group.headlist[nodeindex - nodelist.length]
        IF [headnode is a leaf]
            front.leaf = [leaf from headnode]
        ELSE
            front.hnode = [group from headnode]
            front.node = front.hnode.nodelist[0]
        END IF
    END IF
ELSE //Both front and back are head nodes
    headnodeindex = nodeindex - nodelist.length
    backheadnode = group.headlist[headnodeindex]
    frontheadnode = group.headlist[headnodeindex + 1]

    IF [backheadnode is a leaf]
        back.leaf = [leaf from backheadnode]
    ELSE
        back.hnode = [group from backheadnode]
        back.node = back.hnode.nodelist[0]
    END IF

    IF [frontheadnode is a leaf]
        front.leaf = [leaf from frontheadnode]
    ELSE
        front.hnode = [group from frontheadnode]
        front.node = front.hnode.nodelist[0]
    END IF
END IF

```

Listing 22. Getting back and front

The algorithm requires a maximum of three tests to determine a back and front node; however, most of the time only two tests are required because groups generally have a few nodes in them. The leaf test is also required by a general BSP tree; therefore, the generally there is only one extra test required when compared to a general BSP tree.

Note that in some cases the compression stage 3 can be ignored and standard back/front pointers could be used. In these cases there would still be an efficiency advantage in tree traversal without the overhead of extra calculations. However, the increased memory and use of back/front pointers can increase the probability of cache misses for large trees.

Source code for computing the child locations can be found in 18.7.3.

### 8.2.3 Collision Detection

Collision detection is performed once the camera node has been found. Collision detection is performed by traversing all the planes contained in the leaf's linked list of collision information. If the player is behind a plane then a collision has occurred. Only the closest plane in a collision is remembered. If a collision is found in the collision information, the leaf's parent node (recursively to the root of the tree) must be traversed to find out if any of the nodes in the leaf are in collision and are closer. Once a collision has been fixed, the collision algorithm must be tested again until there are no more collisions.

With SNC trees collision detection is no worse than a general tree because the same amounts of nodes are traversed. In many instances, SNC tree collision detection can be more efficient because solid nodes are a linked-list of planes which has less overhead than a BSP tree.



## 9 Hardware Occlusion Culling

BSP trees can be used to accelerate occlusion by sorting polygons in back to front order. When combined with multi-passing the first pass can perform the occlusion culling at the same time, which saves time on subsequent passes. HOC can be improved by removing clusters of polygons at once using bounding volumes.

### 9.1 The Queues

Essentially, the nodes' BB can be checked in BSP tree traversal in the same way BB are checked against the view frustum. However, the results of hardware occlusion are not available instantly. Queues can be used to build up a log of waiting queries to process while waiting for results to return. Multi-threading is not used as for single processors as multi-threading was found to cause more thrashing overhead than it was worth, although more investigation would be required in that area to come to a definitive conclusion.

In the implementation six queues are used:

- Ready Queue (RQ)
- Ready Queue for no view frustum (RQNF)
- Occlusion Queue (OQ)
- Occlusion Queue for no view frustum (OQNF)
- Occlusion Queue For Single Detail
- Occlusion Queue For Shared Detail

Circular queues are used because the queue maximum size is known and they are memory efficient when the size is being continually popped and pulled. The RQ and RQNF are used to build up a list of nodes before processing, so that there is always work to do while there are nodes potentially visible. Ready queue nodes are always ready for processing; therefore ready queues do not perform any occlusion culling on the nodes they contain. When a node is processed its children are added to a queue for later processing. RQs contain a fixed threshold of elements. When the RQ threshold overflows that element is added to an OQ instead.

The RQNF and OQNF hold lists of ready items that no-longer require VF tests, but still may require other tests. Thus, nodes that are completely within the view frustum are not tested against the view frustum. Whenever an operation occurs on RQ or OQ also occurs on RQNF except without the VF tests. Therefore, from this point on, only RQ and OQ will be discussed as the process is much the same with RQNF and OQNF.

When a node is added to an occlusion queue that item's BB area is rendered with hardware occlusion enabled without performing any expensive rendering operations (such as lighting, texturing or blending) and without writing to the z-buffer or visibility buffer. The results of the invisible render are called queries.

Queries in the OQ are polled and processed as they are completed. If a node is found to be visible then all its children are also processed and added to the relevant queues. If there are no items ready in the OQ, then the RQ queries are processed. If there are no results ready in the RQ and OQ, then the OQ is forced to process its contents even if the results are not ready.

The OQ can have a fixed size. When the OQ overflows the remaining nodes are added to the RQ which causes the RQ to exceed its threshold value. Therefore, the RQ must be large enough to contain all polygons in the scene minus the size of the OQ. Using this strategy means that the algorithm will always have something to do and isn't stuck waiting for the graphics card to get back with the results. However, it is still a good idea to put non-graphics card related code (for example AI) before forcing OQ lists to process in order to give occlusion queries a chance to get back with a result.

Detail occlusion is performed after the rest of the tree has been rendered, using the BB of the detail, so that the detailed objects can make the maximum use of any occluders. Note that any detail in the current leaf will not be occluded by walls, so it is possible to render that detail first, which will act as an occluder for walls.

*Source code for occlusion culling using queues and BSP trees can be found at 18.8*

## 9.2 Frame-to-Frame Coherency

Frame-to-frame coherency can be taken advantage of because the ARB Occlusion query returns a value representing how visible the resulting render was. Thus, marking the nodes with more than a threshold amount showing with the current frame number and then not testing that node for some number of frames. Performance is improved because what is visible in one frame is likely to be visible in subsequent frames. Anything that is determined probably to be visible is added to the RQ as opposed to the OQ.

## 10 Combining View Space Linking, Solid Node Compression and Hardware Occlusion Culling

Combining VSL, SNC and HOC has several advantages in the areas of polygon rejection, traversal and collision detection.

### 10.1 Traversal

VSL and SNC help reduce traversal times for finding the camera, marking and rendering because there are simply less nodes to traverse to find the relevant leaves.

### 10.2 Collision Detection

Using the combined approach makes it possible to short-cut collision detection using the links because there are no longer any solid nodes to collide with as the tree is traversed. Furthermore, for improved accuracy an accessible link for collision detection can be computed in the same way as the view space linking nodes are computed (see “View Space Linking”) by replacing step 1 with:

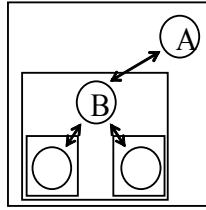
1. Mark all the leaves (and their parents) that are accessible from the current leaf.

Then, when collision detection occurs, the algorithm can start from the accessible node rather than the top of the tree. However, since all visible nodes are also accessible but not all accessible nodes are visible, the link used for visible nodes (vis link) is almost as efficient for collision detection and requires less overhead because vis link are being reused.

### 10.3 Occlusion

Occlusion benefits from both VSL and SNC because there are fewer nodes to perform occlusion on, without reducing accuracy, as important occluders are not removed. There are fewer nodes with one child in the tree, therefore, AABB frustum and occlusion rejection becomes more valuable. Thus, nodes with one child (which may have more children, recursively) are never worth performing AABB culling on because, although all the children may be rejected, the child node will perform the same but more precise tests anyway. In effect, with more single nodes (for example Figure 43) the AABB tests are performing a single test for a single polygon and rejection is performed more efficiently by the hardware. Therefore, VSL and SNC trees require less AABB frustum and occlusion tests without reducing the amount of rejections.

Tree with single child



No single child tree

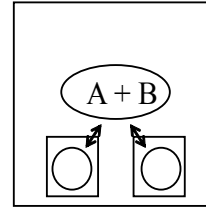


Figure 43. Single linked node versus no-single link node

In Figure 43 if the tree was occlusion tested using the tree with a single node link then these are the possible outcomes:

- A is rejected therefore B is rejected (1 test)
- A is accepted but B is rejected (2 tests)
- A is accepted and B is accepted (2 tests)

If the tree without a single link is used, then these are the possible outcomes:

- A + B is rejected (1 test)
- A + B is accepted (1 test)

Therefore, considering that B will efficiently be removed in hardware there is no advantage in performing extra tests.

Another advantage of using occlusion with SNC is that the RQ will fill up earlier with waiting requests allowing more nodes to be placed on the OQ earlier in the pipeline. Thus, two child nodes generally cause the RQ to be increased more often because there are more nodes in RQ at each stage. As a result the RQ can be made larger, allowing for more time for the card to get back with an occlusion result.

Note that there will be some single nodes found in the tree because of marking however there will be less single nodes to find then without SNC.

## 11 Findings

Nine rendering algorithms were benchmarked to determine how they compared, including a brute force algorithm that did not apply HSR. The algorithms were benchmarked on an AMD-XP 1800+ machine which contained a Geforce 3 and 512 GB of ram. All benchmarks were run with vertical synchronisation off. Only one map was used which consisted of 194452 and 685920 vertices. Figure 44 shows four images of the map.

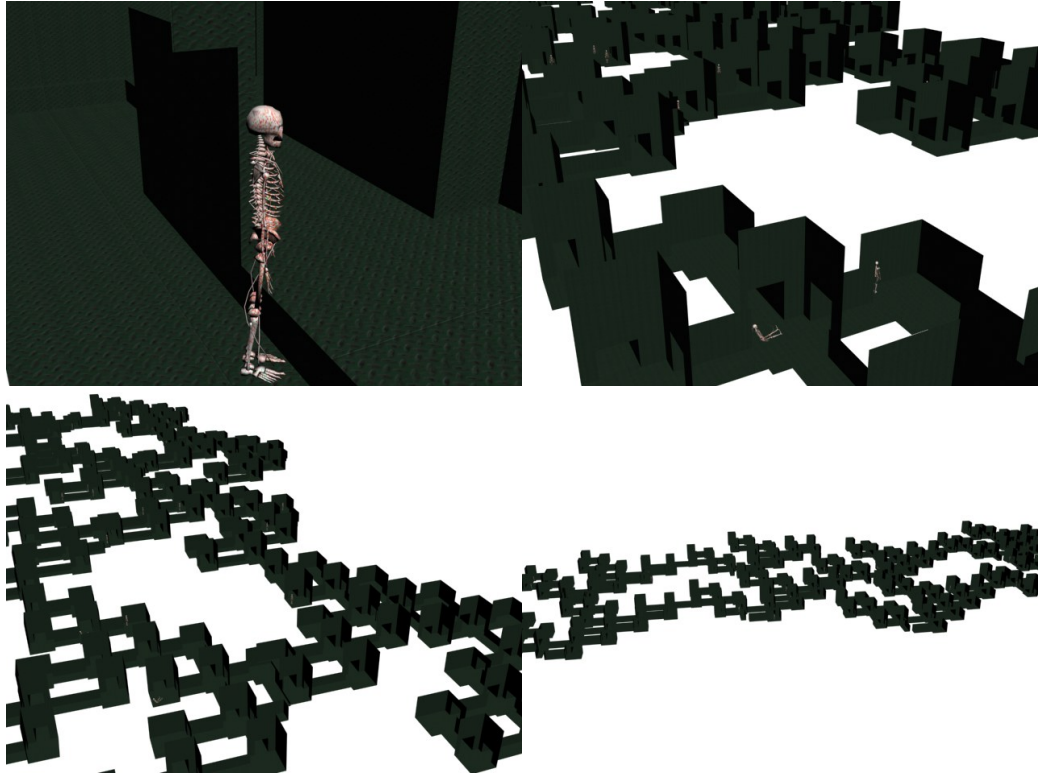


Figure 44. Map used for benchmark tests

Each algorithm was benchmarked using three camera paths shown in Figure 45. Each path was travelled five times to help reduce anomalies. The camera paths are shown by the bright line. In total 135 benchmark runs were performed.

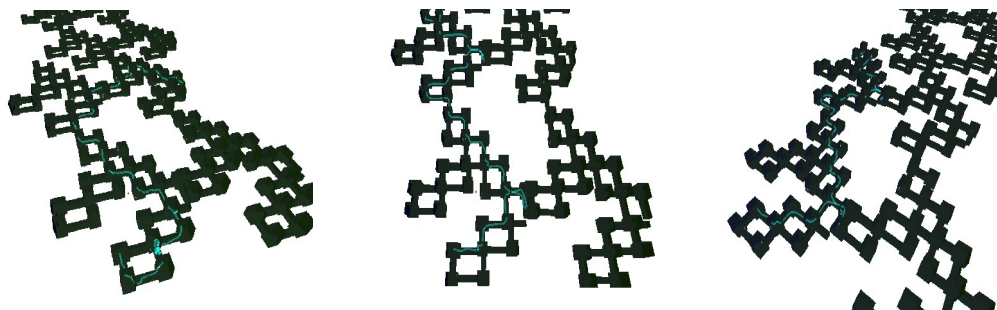
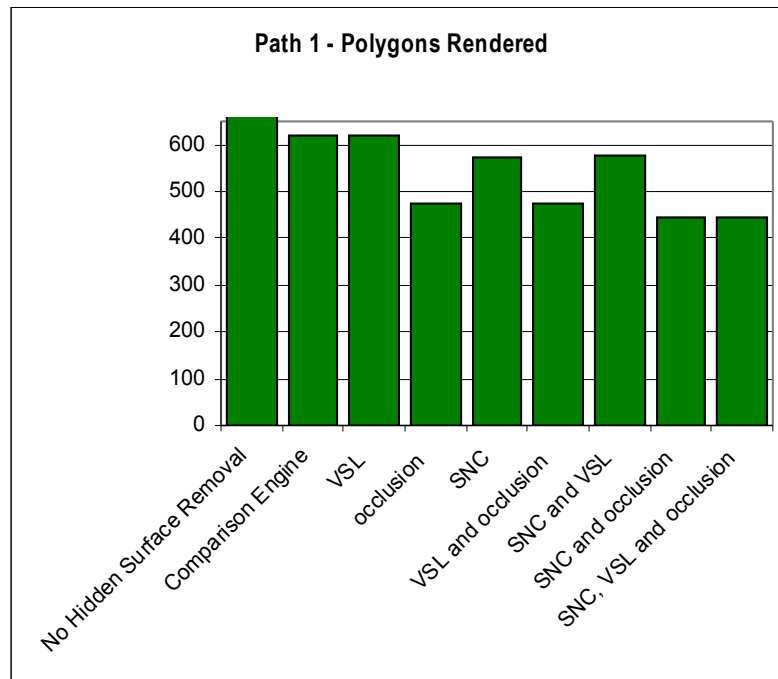


Figure 45. Camera Paths

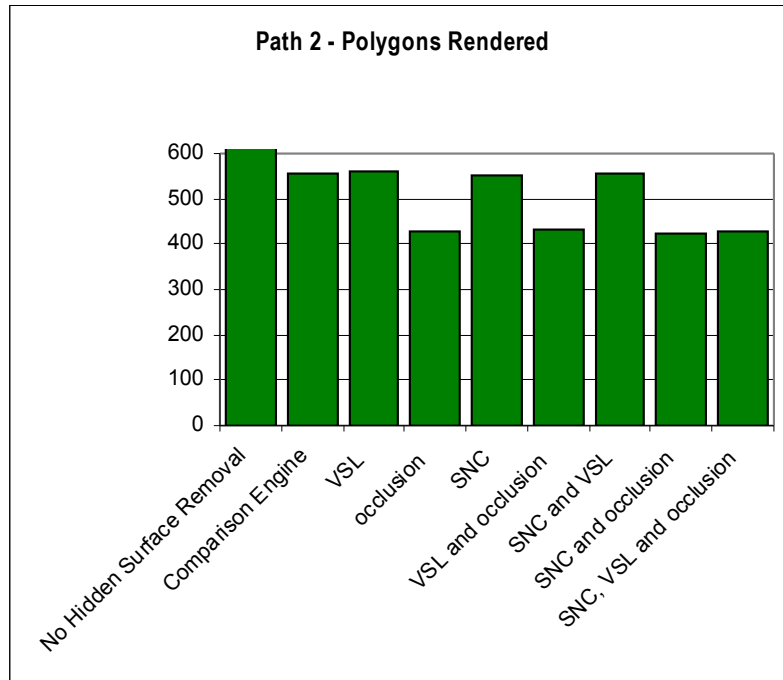
Two major aspects, polygons rendered and performance, were benchmarked to help determine the value of the algorithm.

## **11.1 Polygons Rendered**

The Listing 23 shows how the algorithms performed with polygon removal for each path. The number of polygons rendered was logged to a file for per frame. The charts below show the average results for each path.

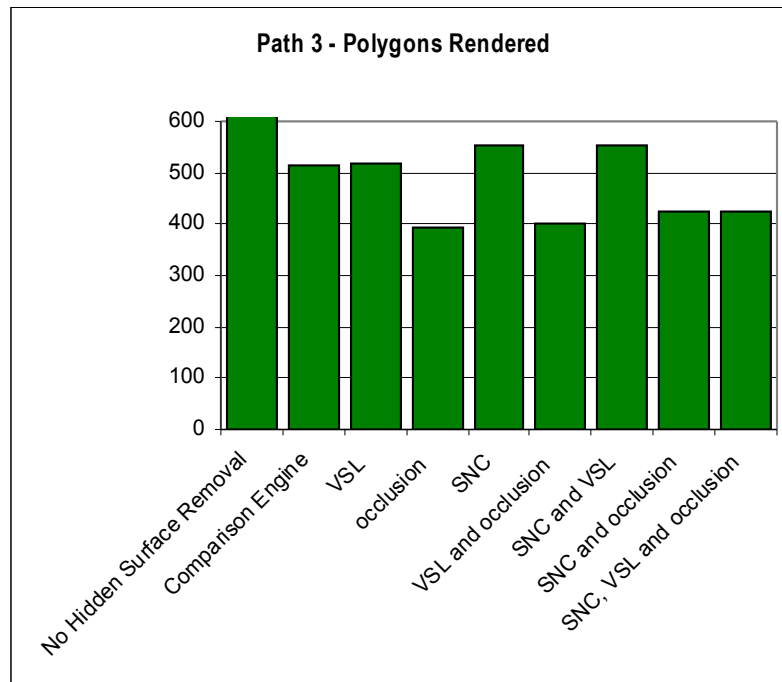


Name	Polygons rendered
No Hidden Surface Removal	194452
Comparison Engine	618.7113466
VSL	618.5596639
occlusion	475.3769063
SNC	575.0611889
VSL and occlusion	476.1462185
SNC and VSL	578.5482104
SNC and occlusion	443.6728914
SNC, VSL and occlusion	444.6632431



Name	Polygons rendered
No Hidden Surface Removal	194452
Comparison Engine	557.1912503
VSL	559.1172397
occlusion	428.9293696
SNC	552.5526743
VSL and occlusion	430.4339064
SNC and VSL	556.9705826
SNC and occlusion	423.9988061
SNC, VSL and occlusion	427.5454155





Name	Polygons rendered
No Hidden Surface Removal	194452
Comparison Engine	512.6551767
VSL	518.9232486
occlusion	394.3580492
SNC	553.382517
VSL and occlusion	399.5198595
SNC and VSL	553.7963629
SNC and occlusion	425.3871461
SNC, VSL and occlusion	426.2240959

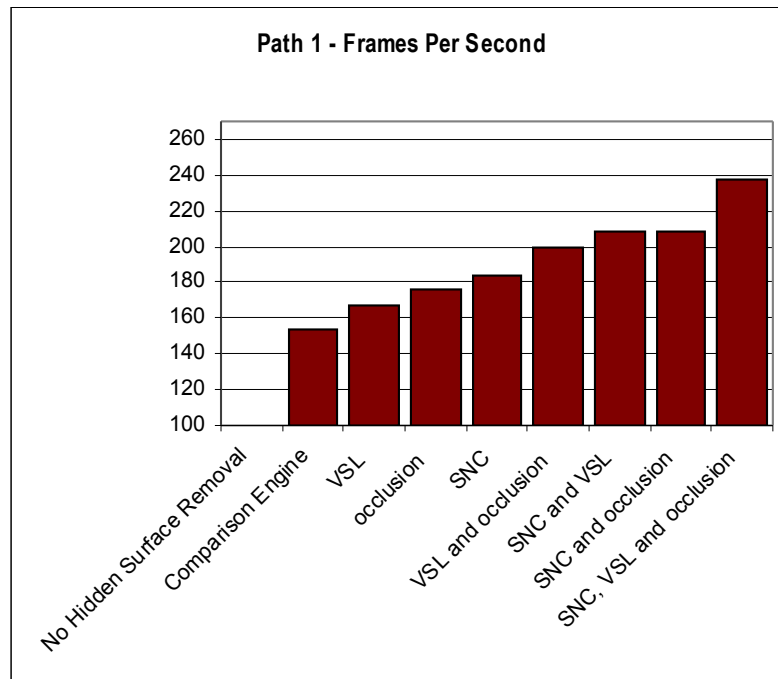
Listing 23. Polygons Rendered

Note that with 194452 polygons, “no hidden surface removal” is just too big to display relatively to the other results.

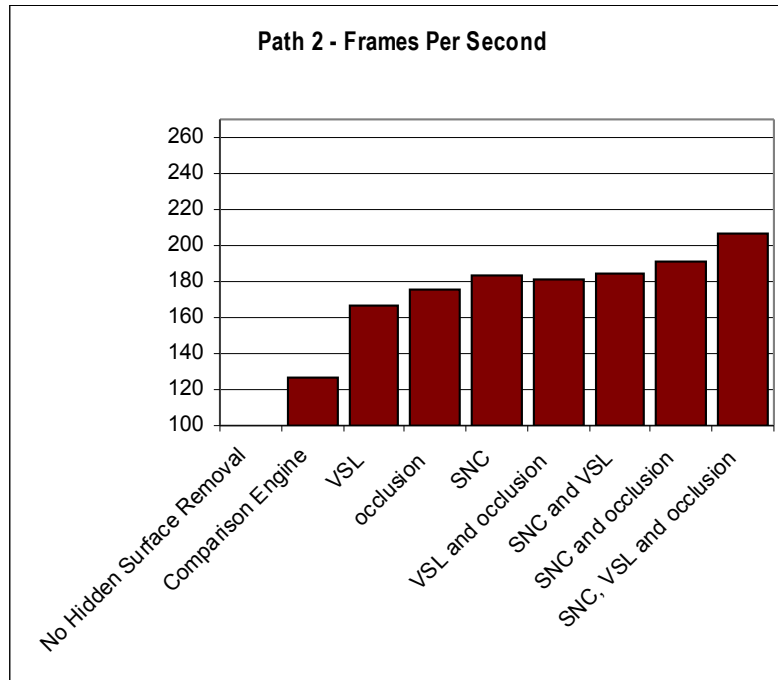
As the results in Listing 23 show, with an average improvement of about 22%, occlusion removal had the greatest impact on hidden polygon removal. When running the algorithm the Geforce 3 card often did not return occlusion results in time to remove all the occluded polygons. Therefore, as 3D cards improve and as more passes are applied to the occlusion removal algorithms, the better the occlusion algorithms should perform. The SNC and VSL algorithms seem to perform better in path 1, however not so well in path 2 and 3. Therefore they cannot be considered any better at removing polygons than the comparison engine.

## 11.2Performance

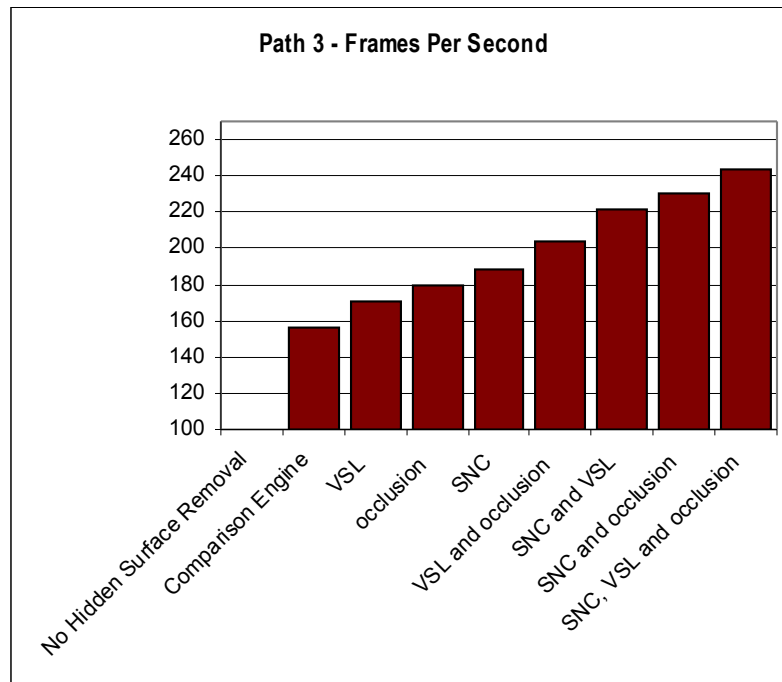
Performance was tested by measuring average FPS. The average frames per second were log to a file ten times and then the total average FPS were used to compile the results in Listing 24 for each path.



Name	Total FPS
No Hidden Surface Removal	1.301731422
Comparison Engine	153.5245253
VSL	166.8429395
occlusion	175.5987214
SNC	183.4877706
VSL and occlusion	199.9726485
SNC and VSL	208.2868476
SNC and occlusion	208.5363082
SNC, VSL and occlusion	237.9824591



Name	Total FPS
No Hidden Surface Removal	1.301826405
Comparison Engine	126.9897083
VSL	166.8429395
occlusion	175.5987214
SNC	183.4877706
VSL and occlusion	180.9205453
SNC and VSL	184.6235702
SNC and occlusion	190.6664392
SNC, VSL and occlusion	206.844946



Name	Total FPS
No Hidden Surface Removal	1.29842033
Comparison Engine	156.4269051
VSL	170.3174232
occlusion	179.3640736
SNC	187.9081181
VSL and occlusion	204.1132435
SNC and VSL	221.8626585
SNC and occlusion	230.5767007
SNC, VSL and occlusion	243.9475892

Listing 24. Frames per Second

As Listing 24 shows, “No Hidden Surface Removal” highlights the importance of using HSR even if that is the “comparison engine”. Running at the non-interactive rate of 1.3 fps, the “No hidden surface removal” algorithm took 15 hours to complete all path runs when compared to the other algorithms which took around 5 to 10 minutes.

The combination of SNC VSL and occlusion outperforms all other algorithms benchmarked, an improvement over the comparison engine of about 35%. Separately VSL, SNC and occlusion provide up to a 16% improvement. In some cases a 16% improvement in rendering performance is not worth the effort; however it is expected that these results would improve with more complex scenes.

## **11.3 Other points of interest**

- When the camera was looking at a detailed object, performance slowed down and the number of polygons increased. The slow down in performance was the same no-matter what algorithm was used with the exception of the "No Hidden Surface Removal" algorithm.
- Although SNC performed slightly faster and VSL slightly slower in some benchmarks of loading times, results were only different by about 0.1% and therefore insignificant.
- Collision detection in SNC improved about 31%. However, because rendering took up the majority of the time per frame, the result is not noticeable. The improvement in collision detection would be more noticeable in scenes where there are more objects performing collision detection at the same time.
- To be conclusive about these finds many more detailed maps would need to be designed and benchmarked.
- All the data from the benchmarks are provided in the xls files on the CD that comes with this thesis.

## 12 Conclusion - The future

HSR is an important part of the rendering pipeline because HSR allows for more polygons to be rendered in the same amount of time, increasing realism and interactivity. There are many data structures that can be used to help aid HSR such as octrees, KD-trees, portals and Z-pyramids. A BSP tree is one such data structure that can be used to help improve HSR. BSP trees have been used in many of the world's popular 3D engines, including ID™'s Quake engines.

BSP trees work by tightly sub-dividing the world into convex areas using planes as splitters. As the world is subdivided into recursive convex volumes, the scene can efficiently be polygon ordered by traversing the closest convex volumes to the camera first. Balancing and minimisation algorithms, such as LCS or MCS, increase the efficiency of traversal. Using BSP trees for merely sorting polygons is less useful nowadays with hardware Z-buffers. However, there are many improvements that can be applied to the general BSP tree to improve HSR, such as storing PVS in the leaves of the tree and culling hidden nodes at run time using bounding volumes.

PVS generation is relatively expensive and is generally pre-computed at compile time rather than at run-time. PVS can be generated for a leaf by traversing the portals that are visible to that leaf, using an anti-penumbra to reject hidden leaves. At run-time, nodes that are in the PVS for the leaf that the camera is within are marked. The marked nodes in the tree are traversed to determine which nodes are visible.

Although, bounding volumes are normally cheaper to compute than PVS, they are also generally pre-computed. Each node's bounding volume is the minimum bounding volume that will hold all of that node's child nodes. Then only nodes with visible bounding volumes need be traversed. Using PVS and bounding volumes together can (in many cases) cost-effectively removed more than 99% of polygons.

The VSL algorithm is able to help the BSP tree algorithm to take advantage of frame-to-frame coherencies. The VSL algorithm works by using the last convex leaf the camera was in to reduce the amount of nodes that are traversed in the next frame. That is, a link stored in each leaf is used to skip relatively expensive operations in the next frame that would normally be performed without VSL. This link is found by marking all the potentially visible nodes for the current leaf and, then, traversing the tree to find out where the tree first splits in two.

The SNC algorithm is used to reduce the size of the tree that is to be traversed, thus resulting in performance improvement. The SNC algorithm reduces the size of the tree by reallocating singularly linked nodes into the leaf nodes of the tree, thus increasing the balance of the tree. Collision detection is enhanced in SNC because only a singular linked list ever needs to be traversed, as opposed to a single path down a binary tree where the direction being followed needs to be taken into account. Furthermore, because the tree is mostly balanced, the size of the tree can be compressed by dividing the tree into several arrays; removing the need to store back, front and parent links at each node.

The HOC can take advantage of BSP trees traversal algorithms ability to order the scene in addition to enabling performance gains through early rejections. The presented implementation of HOC performs BB HOC test at potentially critical nodes in the tree. If a node is not visible then so are all that node's children (recursively). Nodes that are above a certain size (volume or children) or below a certain size are not tested for occlusion because in these cases the average test will likely cost more than the savings it provides. Furthermore, because HOC tests can take a while to return, specialised queues can be used to make use of the CPU if the 3d card is not fast enough.

As has been shown SNC, VSL and HOC occlusion culling can be used together to get combined performance improvements of up to 35% for static 3D environments. VSL and SNC benefit each other because the link can be used to skip several collision tests that would normally be required in VSL or SNC by themselves. HOC benefits from SNC and VSL as the nodes in the tree are more likely to be occluded. As hardware improves and more is demanded of each polygon, the benefit of using algorithms such as SNC, VSL and HOC should become more evident.

There is still room for improvement in the given algorithms. The given BSP tree PVS generation takes much time and therefore could be frustrating to developers. When generating the PVS data many paths may be revisited many times only to perform the same operations. These paths are an obvious case for caching. The BB BSP tree could be used to cull nodes quickly that are not visible to the particular PVS. Furthermore, if the mesh gets too complex the mesh could be divided up into several smaller meshes.

The amount of structural planes in the BSP tree can be reduced by removing detail, however in many causes the tree can still be too large to process in reasonable time. More polygons could be removed from the tree by pre-sorting polygons into convex groups. Rather than divide the scene up by planes, the scene would be divided up by the convex objects. By reducing the amount of structural planes in the tree there is less work for BSP tree, at compilation and run-time.

The engine provided on the CD is obviously not complete; however it is a good starting point. A professional HSR engine would probably apply techniques such as simplification, support for dynamic objects and serialisation from a slow medium such as a hard-drive. At compile-time, simplified geometry could be used as the structural planes, having the more complex geometry as detail. Then, any detail could be simplified at run-time using one of the common techniques.

Dynamic objects could easily be supported in the same way detail was supported in the algorithms. Whenever an object moves its location, the tree would be re-calculated by traversing the tree to see which leaves the dynamic object is contained within. VSL and SNC would help reduce the amount of nodes required to be traversed, taking advantage of frame-to-frame coherency of the dynamic object.

SNC and VSL make the tree easy to stream because each of the SNC groups could be efficiently be allocated/un-allocated and VSL could be used to predict which groups will be needed in the future. Furthermore, smaller sized indexes (such as 8bits per index as opposed to 32bit) could be used for the group back/front links because the number of groups mean there is smaller addressable space than in general trees.

As technology evolves who knows what direction it will go? We can only learn from our past. HSR will stay a necessary part of the software graphics pipeline as long as there is a bridging bottleneck between the 3D hardware and the CPU. BSP trees have been used a long time and have evolved much. Using algorithms like the ones presented here, will hopefully extend BSP tree shelf life.

The techniques described in this thesis are just a foundation to build upon. I encourage you to look outside the box and create something impressive. The world of computer graphics is only limited by imagination.

*If your have not already and are interested in learning more about computer graphics techniques please check out the appendixes, as they contain many details about computer graphics that could not be covered in the body of the thesis.*



## 13 Reference List

- Abrash, M. (1996). *CGDC Quake Talk*, 2002, from <http://www.gamers.org/dEngine/Quake/papers/mikeab-cgdc.html>
- Abrash, M. (1997). *Michael Abrashes Graphics Programming Black Book, Special Edition*, 2002, from [www.ddj.com](http://www.ddj.com)
- Abrash, M. (2000). *Quake's 3-D Engine: The Big Picture*, 2000, from <http://www.gamedev.net/reference/articles/article987.asp>
- Assarsson, U., & Moller, T. (1999). *Optimized View Frustum Culling Algorithms*: Chalmers University of Technology
- Department of Computer Engineering.
- Assarsson, U., & Moller, T. (2000). Optimized View Frustum Culling Algorithms for Bounding Boxes. *journals of graphics tools*.
- ATI™'s Developer Page website. (n.d.). 2003, from <http://mirror.ati.com/developer/index.html>
- Bergen, G. V. D. (1998). *Efficient Collision Detection of Complex Deformable Models using AABB Trees*. Eindhoven University of Technology, The Netherlands.
- Bikker, J. (n.d.). *Building a 3D Portal Engine*, 2002, from <http://www.flipcode.com/portal/>
- Bourke, P. (1997). *Surface (polygonal) Simplification*, 2002, from <http://astronomy.swin.edu.au/~pbourke/modelling/surfsimp/>
- Butcher, C. (1999). *Interactive CSG*, 2002, from <http://www.finality.net/otago/sketch.html>
- Cohen, D., Chrysanthou, Y., Silva, C. T., & Durand, F. (n.d.). *A Survey of Visibility for Walkthrough Applications*, 2003, from <http://graphics.lcs.mit.edu/~freda/PUBLI/surveyTVCG.pdf>
- Csabai, I. (n.d.). *Space Partitioning Experiments*, 2002, from <http://tarkus.pha.jhu.edu/database/kdtree.html>
- DeWan, G. (2000). *The BSP Process and Visibility*. Retrieved 2003, from [http://www.gamedesign.net/archive/old.gamedesign.net/Quake2/tutorials/dewan\\_hint/](http://www.gamedesign.net/archive/old.gamedesign.net/Quake2/tutorials/dewan_hint/)
- Eberly, D. H. (2000). *3D Game Engine Design*. San Francisco: MK.
- Foley, J., Dam, A. v., Feiner, S., & Hughes, J. (1990). *Computer Graphics: Principles and Practice, second edition*. MA: Addison-Wesley.

- Foley, J., Dam, A. v., Feiner, S., Hughes, J., & Phillips, R. (1994). *Introduction to Computer Graphics*, Addison-Wesley: Addison-Wesley.
- Freidin, B. (1999-2000). *Hierarchical Solid Boolean Modeling*, 2003, from <http://bork.hampshire.edu/~bernie/demo1/flythrough%20paper.pdf>
- Fuchs, H., Abram, G. D., & Grant, E. D. (1983). On visible surface generation by a priori tree structures. *ACM SIGGRAPH Computer Graphics*, 17(3), 63-72.
- Fuchs, H., Kedem, Z. M., & Naylor, B. F. (1979). Predetermining Visible Priority in 3-D Scenes. *ACM SIGGRAPH Computer Graphics*, 13(2), 175-181.
- Fuchs, H., Kedem, Z. M., & Naylor, B. F. (1980). On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH '80*, 14(3), 124-133.
- Garland, M. (1999). *Quadric-Based Polygonal Surface Simplification*. Unpublished Doctor of Philosophy, Carnegie Mellon University.
- Gottschalk, S. (1999). *Collision Queries using Oriented Bounding Boxes*. Unpublished Ph.D., niversity of North Carolina, Chapel Hill.
- Hammersley, T. (n.d.). *BSP Trees*, from <http://tfpsly.planet-d.net/Docs/TomHammersley/bsp.htm>
- Hearn, D., & Baker, M. P. (1986). *Computer Graphics C Version*. Upper Saddle River, New Jersey: Prentice Hall.
- Hillesland, K., Salomon, B., Lastra, A., & Manocha, D. (n.d.). *Fast and Simple Occlusion Culling Based on Hardware Depth Queries*, 2003, from <ftp://ftp.cs.unc.edu/pub/publications/techreports/02-039.pdf>
- Hoff, K. (1996a). A "Fast" Method for Culling of Oriented-Bounding Boxes (OBBs) Against a Perspective Viewing Frustum in Large "Walkthrough" Models, from <http://www.cs.unc.edu/~hoff/research/vfculler/viewcull.html>
- Hoff, K. (1996b). *Testing the Perspective Viewing Frustum Module*, 2003
- Hoff, K. (1997). *FAST AABB/View-Frustum Overlap Test*, from <http://www.cs.unc.edu/~hoff/research/vfculler/boxvfc/boxvfc.html>
- Hoff, K. E. (1996). *Kenny Hoff's Graphics Depot*, from <http://www.cs.unc.edu/~hoff/>
- Hofmann, J. S. (2000). *A Survey of Real-Time Rendering Algorithms*, from <http://www.seas.gwu.edu/~graphics/cs367/real-rendering.pdf>
- ID Software's website. 2003, from <http://www.idsoftware.com>

- ID™ Software's website*. (n.d.). 2003, from <http://www.idsoftware.com>
- James, A. (1996). *Interactive building walk-through*: University of East Anglia.
- James, A. (1999). *Binary Space Partitioning for Accelerated Hidden Surface Removal and Rendering of Static Environments*. Unpublished Doctor of Philosophy, University of East Anglia, East Anglia.
- James, A., & Day, A. M. (1997, March). *The priority face determination tree*. Paper presented at the Eurographics UK '97, UK.
- James, A., & Day, A. M. (1998). The priority face determination tree for hidden surface removal. *Computer graphics Forum*, 17(1), 55-71.
- Kmett, E. (1999a). *Fine Occlusion Culling Algorithms*, 2003, from <http://www.flipcode.com/harmless/issue01.htm>
- Kmett, E. (1999b). *Scene Traversal Algorithms*, 2003, from <http://www.flipcode.com/harmless/issue02.htm>
- Kmett, E. (2001). *A Hybrid Approach to Visibility*, 2003, from <http://www.flipcode.com/harmless/issue04.htm>
- Kumar, A., & Kwatra, V. (1999). *Algorithms for Efficient Dynamic Hidden Surface Removal for Curved Surfaces*. Unpublished BTech Project Final Report, Indian Institute of Technology, Delhi.
- Kushner, D. (2002). *The Wizardry of ID*, 2002, from <http://www.spectrum.ieee.org/WEBONLY/publicfeature/aug02/id.html>
- Lamothe, A. (1995). *Black Art of 3D Game Programming Writing Your Own High Speed Polygon Video Games in C*. Corte Madera, CA: WAITE GROUP PRESS TM.
- Loisel, S. (1996). *Zed3D - A compact reference for 3D computer graphics programming* (V .95b).
- McGuire, M. (2002). *Quake 2 BSP File Format*, 2003, from <http://dev.niaka.com/Quake/article/flipCode%20-%20Tutorial%20-%20Quake%20%20BSP%20File%20Format.htm>
- Moller, T., & Haines, E. (1999). *Real-Time Rendering* (1 ed.). Natick, Massachusetts: A K Peters.
- Morley, M. (2000). *Frustum Culling in OpenGL*, 2003, from <http://www.markmorley.com/opengl/frustumculling.html>
- MSDN DirectX page*. (2003). 2003, from <http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000410>

- Neider, J., Davis, T., & Woo, M. (1993). *The Red Book*. Massachusetts Menlo Park: Addison-Wesley Publishing Company.
- Nettle, P. (2001). *Fluid Studios Publications*, 2002, from <http://www.fluidstudios.com/publications.html>
- Nuydens, T. (2000). *Octrees*, 2003, from <http://www.delphi3D.net/articles/viewarticle.php?article=octrees.htm>
- Nvidia's Developer Relations website*. (n.d.). 2003, from <http://developer.nvidia.com/>
- OpenGL®. (2003). 2003, from [www.opengl.org](http://www.opengl.org)
- OpenGL®. (2004). 2004, from [www.opengl.org](http://www.opengl.org)
- OpenGL® 2.0 website*. (2002). 2003, from <http://www.3Dlabs.com/support/developer/ogl2/index.htm>
- OpenGL® Extension Registry*. (2003). from <http://oss.sgi.com/projects/ogl-sample/registry/>
- OpenGL® Extension Registry*. (2004). from <http://oss.sgi.com/projects/ogl-sample/registry/>
- Picciotto, N. (1995). *NP-Completeness, Cryptology, and Knapsacks*, 2003, from <http://www.derf.net/knapsack/>
- Schumaker, R. A., Brand, R., Gilliland, M., & Sharp, W. (1969). *Study for applying computer-generated images to visual simulation*. (No. AFHRL-TR-69-14): US Airforce Human Resources Laboratory.
- Shaffer, E., & Garland, M. (2001). *Efficient Adaptive Simplification of Massive Meshes*. Paper presented at the IEEE Visualization 2001.
- Shreiner, D. (2001). *Performance OpenGL: Platform Independent Techniques*, 2003, from <http://www.plunk.org/~shreiner/SIGGRAPH/s2001/Performance.OpenGL/perfogl.pdf>
- Sung, R. C. W., Corney, J. R., & Clark, D. E. R. (2000). *Octree Based Recognition of Assembly Features*. Paper presented at the ASME 2000 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Baltimore, Maryland.
- Sutherland, Sproull, & Schumaker. (1974). A Characterization of Ten Hidden Surface Algorithms. *ACM Computing Surveys*, 6(1), 1-55.
- Teller, S. J. (1992a). *Computing the Antipenumbra of an Area Light Source*. Paper presented at the Siggraph '92.
- Teller, S. J. (1992b). *Visibility Computations in Densely Occluded Polyhedral Environments*. Unpublished Doctor of Philosophy, University of California at Berkeley, California.

Teller, S. J. (1992c). *Visibility Computations in Densely Occluded Polyhedral Enviroments*.

Unpublished Dortor of Philosophy, University of California at Berkeley, California.

Teller, S. J., & Séquin, C. (1991). *Visibility Preprocessing for Interactive Walkthroughs*. Paper presented at the Siggraph '91.

*Unreal Tournament's website*. (n.d.). 2003, from <http://www.unrealtournament.com/>

## 14 Appendix 1 - List of Abbreviations

---

2D	Two Dimensional
3D	Three Dimensional
3DSMa	3D Studio Max
x	Anti-penumbra      PVS      determination
APD	algorithm
BB	Bounding Box
BSP	Binary Space Partition
CD	Collision Detection
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
FOV	Field Of View
FPS	Frames Per Second
HFD	Hidden Face Determination
HNP	Head Node Pair
HNPI	head node parent Iterator
HOC	Hardware Accelerated Occlusion Culling
HSR	Hidden Surface Removal
ID™	Instinct-Driven software
KD	K-Dimensional
L3	Level 3 Infront List
LCS	Least-Crossed criterion
MCS	Most-Crossed criterion
MSDN	Microsoft Developer Network
PFD	Priority Face Determination
PPO	Polygon Priority Order
PVS	Potentially Visible Sets
RQ	Ready Queue
RQNF	Ready Queue for No view Frustum
SNC	Solid Node Compression
VF	View Frustum
VFC	View Frustum Culling
VSL	View Space Linking
ZRLE	Zero Run Length Encoding

---

## 15 Appendix 2 - Quake Clone BSP Engines

Due to Quakes popularity many clone Quake engines were created which are available for free on the World Wide Web. These engines are particularly useful in providing a different perspective on the BSP/PVS algorithm used in the Quakes. Therefore a listing of some of these engines has been provided below. Note that some of these links may no longer exist or may have moved.

Aguado, I. C., Rueda, J. L. F., Alberich, G. R., & Rodríguez, J. C. E. (27/04/2000). *Titan Engine*. Retrieved, from the World Wide Web: <http://talika.eii.us.es/~titan/titan/index.html>

Bily, T. (04/10/1996). *Xvizn*. Retrieved, from the World Wide Web: <http://atrey.karlin.mff.cuni.cz/~tomby/3De.html>

Carmack, J. (23/12/1999). *Quake 1/2/3*. Retrieved, from the World Wide Web: <http://www.idsoftware.com>

Coumans, E. (20/01/1997). *Cookie's 3D Engine*. Retrieved, from the World Wide Web: <http://members.xoom.com/gamedev/download/sources/demosrc.zip>

Dukes, K. (26/05/1999). *DJDOOM*. Retrieved, from the World Wide Web: <http://www.geocities.com/ResearchTriangle/System/4503/>

Fortuna, A. (17/09/1999). *Arnold 2*. Retrieved, from the World Wide Web: <http://www.geocities.com/SiliconValley/Lab/3716/>

Frisbie, P. (15/08/1997). *EGLE*. Goloshubin, A. (27/09/1996). *Poly Engine*. Lanza, S. (28/09/1999). *Twister*. Retrieved, from the World Wide Web: <http://gpi.eden.it/twister>

Nikos, K. (03/11/1997). *CLIFF*. Retrieved, from the World Wide Web: <http://www.csd.uch.gr/~komod/engine/main.htm>

Scarsi, R., & Rossi, M. (25/04/1997). *SQuake*. Retrieved, 9/2/2003, from the World Wide Web: <http://staff.polito.it/~scarsi/squake.html>

Sun, P. F. (30/01/1998). *p3D*. Retrieved, from the World Wide Web: <http://www.undergrad.math.uwaterloo.ca/~pfsun/engine.html>

Turnbull, R. (05/10/1998). *Blitz Engine*. Retrieved, 11.12.1999, from the World Wide Web: <http://thesauce.3Dfiles.com/Blitz/>

Tyberghein, J. (25/08/1997). *Crystal Space*. Retrieved 11.12.1999, from the World Wide Web: <http://crystal.sourceforge.net/>

Wilkinson, J., Redeker, R., Ward, T., Teunissen, J., IV, R. M.-M., Hale, F., McGrath, T. C., Roberts, J., Olsen, A., Galbraith, S., Koropoff, B., Currie, B., Gavrillov, A. E., & Ison, C. *QuakeForge*. Retrieved 9/2/2003, from the World Wide Web: <http://www.quakeforge.net/>

Willemsen, F., & Langosch, C. (30/12/1996). *W O F - 3D engine*. Retrieved, from the World Wide Web: <http://www.geocities.com/CapeCanaveral/8518/>

## 16 Appendix 3 - Algorithms

### *Algorithmic descriptions and details*

This section describes and discusses many of the algorithms mentioned in the thesis. Although these algorithm discussions are far too large to put into the thesis body, they are important to understand none-the-less, because:

- the presented solution was derived from these algorithms
- a good understanding of the following information is required to understand this work
- many smaller but no-less important details about the algorithms are discussed here
- to show other potential solutions to the problems, investigated

## 16.1 Portals

Among other strengths, portal engines are useful for reducing the amount of polygons processed in HSR and CD.

### **16.1.1 Portals for HSR**

Once portals have been created the runtime process is relatively simple. Initially, the sector containing the camera is found. The initial sector can be pre-calculated and recorded at compilation time. Collision detection determines if the camera crosses one of the local portals into a new sector. A local portal is defined as a portal that belongs to and links the current sector (where the camera is) to other sectors. To display a scene all local and indirect linking sectors that are visible in the camera's FOV are drawn. Bounding Boxes (BB) can be attached to each portal (or sector) to more efficiently determine if the portal (sector) is within the current view frustum of the camera.

If portals are convex, depth priority-ordering using the painters algorithm can be accomplished by drawing the furthest sectors first. Overdraw can be completely removed by drawing the closest sectors first and each time clipping by the linking portal. Portals can be clipped using other portals, which mean that walls act as occluders. A disadvantage of portals is that if there are many portals, clipping becomes a major bottleneck. However, when sectors contain many polygons or clipping is cheap, using portals as clippers can dramatically increase performance.

Portals are not useful in landscape type situations because every viewpoint can virtually see every other viewpoint. However, portals are useful in a hybrid situation for caves and building on the landscape.

### **16.1.2 Portals and CD**

A moving object needs only to perform collision tests against the polygons and portals in the sector, in which it is contained. On the rare occlusion that an object is found to have passed through a portal, then collision detection needs to be run again (recursively) for that portal's sector.



Consider a large level with a thousand rooms (sectors). Most of the time collision detection will only need to be used on one or two rooms, which means that on average 99.8% of polygons would be culled. However, note that these results are subject to the uniformity of the level. If the camera happens to be able to see 90% of the scene from its current view frustum, then results such as 99.8% are impossible with this strategy (and most other strategies). Using portal algorithms can be an excellent and simple way of reducing polygon count for collision detection.

## 16.2 Quadtrees/Octrees

Quadtrees use 2D rectangles while octrees are the 3D form of quadtrees using bounding cubes (boxes).

### 16.2.1 Building a Quadtree/Octree

To build a quadtree or an octree the root nodes bounding box is created, large enough to contain the entire scene. The root node is equally divided into four sub-node boxes. Each sub nodes bounding box (BB) contains either some or no polygons. If a BB contains more than some cut-off threshold, it is subdivided recursively until that threshold is met; where cut-off threshold either relates to the depth of the tree or the number of polygons in that node.

It is possible for a polygon to span many nodes. Writing the polygon many times in each node it span is one solution to this spanning problem. This technique requires extra tests for duplications. Another slightly different approach is to write the polygon once at the node it spans, however node data can't be stored in the trees leaves. Alternatively, the polygon can be subdivided along the boundaries it spans into individual polygons. If the polygon threshold is more than one, then depth priority-ordering may not hold true and extra special case handling will be needed. Tree depths are typically around  $O(\log(n))$  in height, and take  $O(n \log(n))$  time to create, where  $n$  is the number of polygons.

### 16.2.2 Adaptive Octrees

Adaptive octrees (quadtrees) allow the position at the centre of the octree (quadtree) to be displaced. Adaptive octrees (quadtrees) can be beneficial because they produce tighter fitting trees. However, algorithms such as neighbour traversal are more difficult to solve, as polygons do not align properly. These trees also require additional calculations per node, but are optimistically offset by having balanced trees to process. Furthermore adaptive octrees require more pre-processing to determine best fits for a particular data set. With the exception of nearest neighbour, KD-trees are generally considered a better alternative for balanced trees. "In practice, I tend to prefer a KD-tree to an adaptive octree, but in some cases, the adaptive octree may consume less memory. In addition, the neighbour front-to-back traversal of an adaptive octree tends to be faster than for a KD-tree, because within a given octree node the partitioning planes line up." (Kmett, 1999b).

### 16.2.3 Rendering (HSR) with Quadtrees/Octrees

Octrees are useful for HSR in a number of ways:

#### Nearest Neighbour

The nearest neighbour technique can be used to determine nodes that are beside each other, hence the priority-order of the objects. The “ASME 2000 Design Engineering Technical Conferences and Computers and Information in Engineering Conference” (Sung, Corney, and Clark, 2000) proceedings gives a good overview of the nearest neighbour algorithm. The proceedings explained that nearest neighbour with octrees can either be done with pointers to parent nodes or by using a pointerless octree with a reflection table and adjacent table. Note that some nearest neighbours will be empty and some will be a different size to the node being neighbour searched.

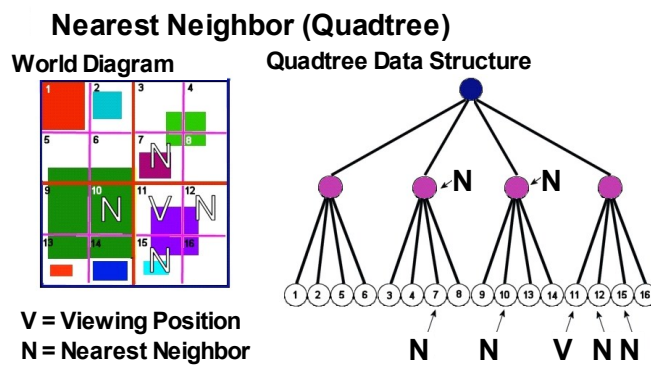


Figure 46. Nearest Neighbour (quadtree)

Figure 46 shows potential nearest neighbours in a quadtree. There are two diagrams in the example. The “World Diagram” shows how the world would look subdivided by a quadtree. The second diagram “Quadtree Data Structure” shows how the quadtree appears as a data structure. The numbered boxes on the “World Diagram” correlate to the numbers in the “Quadtree Data Structure”. When finding V’s nearest neighbours there are several possibilities. If the nodes 7 or 10 did not exist (they were empty) then their parent nodes would become the nearest node.

#### Frustum Culling

The HSR algorithm starts at the root node culling away any sub-nodes that are not within the field of view (FOV). If a node is occluded (or not within the FOV), then no children need to be processed. This process is done recursively until a leaf node is reached. Nodes are rendered to the screen buffer when they contain polygons and they make it past both the FOV and occlusion tests. If occlusion isn’t used, then some form of depth sorting needs to be applied such as Z-buffer to make sure polygons are priority-ordered correctly.

When an octree receives poor data, it may require more than twice as much processing when compared to a non-culling linear method. These trees generally take  $O(\log(n))$  to generate a list of potentially visible polygons, but can be as slow as  $O(n)$ ; where  $n$  is the number of polygons. For example, when running well, an octree can cull more than 99% of the scene with little extra processing.

## Potentially Visible Sets (PVS)

PVS can be applied to octrees by determining at the leaf nodes what other neighbouring cubes are potentially visible. The tree only needs to be followed down one path to find the camera's position in the tree. Occlusion would not be used in this search because it would have already been mostly determined in the PVS. The PVS structure could be optimised using bounding boxes that are completely or almost completely full to represent a group of objects (which could be polygons). These objects would still need to be FOV tested so the larger groups would gain early culling advantages. The drawback of using this PVS technique with an octree is that the technique requires extra memory, does not deal with all occlusions and does not produce polygons in depth (Z) priority-order. Benefits of using PVS with octrees are that it reduces traversals and can eliminate many occluded polygons.

A derivative of the mentioned PVS technique is to place the neighbouring and potentially visible node links (not necessarily leaves) in leaf nodes (similar to "nearest neighbour" algorithm), which we'll call the nearest-neighbour PVS octree algorithm (NPOA). A new data structure may or may not be required depending on whether diagonals are included as linking neighbours. If diagonals are included then eight (twenty-eight) pointers are required for quadtrees (octrees). Without diagonals, more nodes need to be tagged potentially visible, however only four (eight) pointers are needed for a (an) quadtree (octree).

Once the leaf node has been found, visible nodes are determined by successively following the links in the leaf nodes that are within the FOV using ray casting. If the nodes within the ray have children, then those nodes are traversed to find leaves that are within the FOV and the process is repeated (recursively) of each child. Since moving objects are likely to be placed into neighbouring nodes NPOA can be a more efficient way of moving objects in the tree. However the NPOA does require an extra level of storage and more calculations than the other PVS methods. An advantage of NPOA is that it can handle overdraw with polygon priority-ordering.

## Frame-to-Frame coherency

Frame-to-Frame coherency can be taken advantage of by octrees (quadtrees) by realizing that nodes that were not visible in the previous frame will not likely be visible in the next frame. To take advantage of frame-to-frame coherency a pointer (V) to the last node (in the last frame) that fit entirely around the FOV is kept. V can efficiently be worked out as the tree is processed. The last node that doesn't result in any sides of the quads being crossed over becomes V. On processing the next frame for viewing, traversal begins at V.

While  $V$  doesn't fit into the new FOV,  $V$  is set to its parent and the process is repeated.  $V$  may have to be traversed all the way up to the root node to find a node that completely fits the FOV. If data is stored only in leaves of the tree then the tree simply needs to be traversed as-per-normal (node to leaf) from node  $V$ . However, if nodes other than leaves are used to hold objects then the parent (recursively) need to be sent to render. Note that rendering the parent nodes is an  $O(n)$  process since each node has one parent; where  $n$  is the number of polygons. Parents themselves aren't traversed, since only one node will be visible (the previous parent node) and that will already be sent to render. After rendering all the parent nodes, traversal can be done as per-normal (node to leaf) from the node  $V$ .

## 16.2.4 Octrees, collision detection and moving objects

Several algorithms for determining collision detection with octrees are:

### Ray casting

Octrees provide an efficient means for reducing the amount of objects needed for collision detection (CD). The basic approach is to traverse down each node in the tree that the movement vector (ray) intersects. If all leaves are found to be empty then there has been no collision. When a leaf that contains polygons is found, more traditional collision detection is applied to those individual polygons to determine if there has been a collision. This octree CD algorithm is a form of ray casting and can be used to determine selection (for example, objects under the mouse). On average this collision detection algorithm performs at  $O(\log(n))$  because the algorithm only needs to transcend one path of the tree; where  $n$  is the number of objects (polygons) stored in the tree. However  $O(n)$  is still possible (although highly unlikely), if the movement vector transcends all nodes.

### Moving objects

An alternative to the octree ray casting CD technique is to treat moving objects as dynamic objects. When an object moves out of its node's bounding volume, the object needs to be removed and reinserted into the tree at a cost of  $O(\log(n))$ . It is important to note that when an item is removed, some clean up mechanism is needed to remove excess nodes so that the tree does not keep on growing, it is not enough to simply tag a node empty. A simple removal technique is to delete any empty leaves of the tree. One problem with this CD technique is that if the object moves to far then the algorithm may skip the object was meant to be in the collision. In cases where the object moves to much the ray casting technique could be applied or the object inserted several times along its movement path.

The fact that objects that move tend to hang around the same BV can be taken advantage of by traversing the tree in two directions (up and down). Once the node is no longer within its local volume, the object is pushed up the tree

using its parent nodes. If the object fits in one of the parent node then it is recursively pushed down the tree again. Otherwise, the object is recursively pushed up the tree until the object fits into the BV at that node. There is a small possibility that the node may have to travel all the way up to the top of the tree before travelling down again, at a cost double the amount of time required when compared to a re-insert from the trees top. If the object is being transported (for example the star trek beam) then it is best to insert the object from the top of the tree. Furthermore, this traversal technique requires an additional cost of a parent link at each node. One advantage of octrees over BSP trees are that dynamic objects can be more efficiently inserted into the tree.

### 16.2.5 Octrees and CSG

Octrees can be used to perform CSG, as an alternative to BSP trees. Both objects are put into an octree and their leaves are mark as empty or solid depending on whether they are inside or outside the object. Note that leaves that are half inside the object (because they are intersected by one of the objects surfaces) are marked as solid. As with the BSP CSG algorithm (see “Problems solved by BSP trees”, “Constructive solid geometry (CSG)”) the objects in trees are inserted into each other (to determine inside/outside) and the resulting nodes are then determined by the CSG operation (as described in “Problems solved by BSP trees”, “Constructive solid geometry (CSG)”).

A node that is found completely inside of the other nodes is marked as inside. A node that is found outside the tree is marked as outside. When an intersection occurs between a solid node (that contains an object) and the object must be split into two (inside and outside). An object may be split into several objects and sent down different paths in the tree. The octree generally does not assist in determining which side of the object is inside and which is outside. The normal (direction it is facing) of both planes can be used to determine which side is which. Anything behind (in front) of the normal in the node's object is inside (outside).

This algorithm generally takes  $O(n \log(n))$  to process, however octree CSG is generally slower then solid BSP trees because it relies on a slow intersection testing; where  $n$  is the number of polygons in both objects. Furthermore, it is likely that a CSG operation will fragment an object into more pieces then a BSP tree. Therefore it is a good idea to perform an optimisation algorithm on the resulting mesh to reduce unnecessary fragments. Chris Butcher offers an alternative variant of this octree CSG technique in his document “Interactive CSG” (Butcher, 1999). It allows octrees to track changes in dynamic CSG situations using a “changed flag”.

## 16.3 KD-trees

KD-trees are a useful algorithm for HSR, CSG and CD. KD-trees contain many of the same features as octrees/quadrees such as being able to handle indoor and outdoor environments efficiently. However these trees are generally better balanced and therefore a better fit then octrees/quadrees. However nearest neighbour for KD-trees can be less efficient then it is for octrees/quadrees.

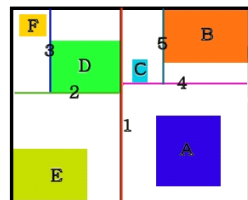
### 16.3.1 Building a KD-tree

Each node in a KD-tree represents a partitioning plane along one of the 2 (2D) or 3 (3D) coordinate axis. These rules are normally used in deciding on what partitioning planes to use for partitioning:

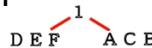
1. It must divide the objects into two even (as possible) groups, normally using the median.
2. The splitting direction (horizontal, vertical or forward) chosen as the splitting plane depends on the depth of the tree. For example in a 2D KD-tree, if the root node is horizontal, then the child node of that will be vertical. The child node of that child will then be horizontal again (recursively). Therefore, there is no need to store the direction of the splitting plane. Having said that, it is possible to use one or two bits per node to indicate non-sequential splitting directions.

Objects are pushed down the side of the tree depending on which side of the plane they are on. For example, objects on the left (right) of the partitioning plane will be pushed down the left (right) side of the tree. The creation process is recursive and only stops when some threshold condition is met. For instance, the algorithm may stop when there is one object remaining. This remaining object would then become a leaf. Once all the objects have been placed in a tree leaf, the KD-tree has been built. Sometimes objects result in being between a partitioning plane, in which case they are either placed in the tree at that node, or more generally split into two new objects along that plane. KD-trees generally cost  $O(n(\log(n)))$  to build and consume  $O(n)$  storage; where  $n$  is the number of objects (polygons).

**Building a KD-Tree**



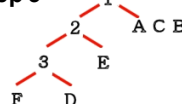
**Step 1**



**Step 2**



**Step 3**



**Step 4**



**Step 5**

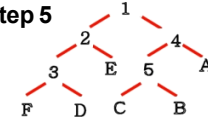


Figure 47. Building a KD-tree

Figure 47 demonstrates the process used to build a KD-tree.

Steps:

1. Divides the scene in half with D, E and F on one side and A, C and B on the other.
2. Divides the left side of the tree into to more (F D and E).
3. Divides the left side yet again into F and D.
4. The left side of the root node has no more items to divide so the right side is divided into CB and A.
5. The final leaf nodes C and B are created though division and now the tree is complete.

Note that this example does not demonstrate cases where a splitting plane goes through an object. In those cases the object is split into two new objects using the partitioning plane.

### 16.3.2 Rendering (HSR) with KD-trees

Various ways to perform HSR with KD-trees are:

#### Frustum Culling

When performing frustum culling, the algorithm starts at the root node. Like octrees the algorithm determines which side(s) of the tree the camera FOV exists in and traverses that (those) node(s). When a leaf node is found, that leaf's object is rendered. KD-trees generally take  $O(\log(n))$  to deduce a set of objects (polygons) that are potentially visible, but can be as slow as  $O(n)$ ; where  $n$  is the number of objects (polygons) in the entire tree. Groups of polygons have a better chance of being discarded earlier in a KD-tree than in an octree/quadtree. If the entire KD-tree needs to be processed then it can take longer than octrees (quadtrees) due to the greater amount of nodes to traverse. Generally, KD-trees are more efficient than octrees at render time because they are better balanced.

#### Nearest Neighbour

Occlusion can be performed on KD-trees in a similar manner to that of octrees (see "nearest neighbour", 16.2.3). The algorithm is slightly more difficult because KD-trees are not as evenly aligned to neighbours as octrees are. Therefore the traversal must walk up the tree to a higher node before traversing down, meaning more nodes need to be visited than with KD-trees.

### 16.3.3 Collision Detection and KD-trees

Collision detection with KD-trees is similar to the techniques used with octrees. For each node, it determines which side of the partitioning plane the movement vector sits on (left, right or both) and traverses that (those) node(s) in the tree. Traversal continues until a leaf node is found. Unlike octrees (quadtrees), all leaves in a KD-tree contain at least one object. For each leaf found that contains one object (or more), further tests are carried out to determine if that (those) objects have collided with the movement vector.

Like octrees, KD-trees the CD algorithm has a minimum runtime cost of  $O(\log(n))$  and a maximum of  $O(n)$ . KD-trees require twice (for 2D) or three (for 3D) times as many nodes as quadtrees or octrees respectively. Furthermore the dynamic object method of collision detection (discussed in 16.2.4) can be also applied to KD-trees.

### 16.3.4 KD-trees and CSG

The KD-tree CSG algorithm is similar to the octree/quadtree CSG algorithm (see 16.2.5). As with octrees, the objects are inserted into the two KD-trees that are divided into solid and empty areas. The trees are then inserted into each other to determine where they intersect and the Boolean operation is applied. KD-trees generally have a tighter fit than octrees; therefore less intersection testing is required. The KD-tree CSG algorithm generally takes  $O(n(\log(n)))$  to process, where  $n$  is the total number of polygons in both objects involved.

## 16.4 Zero Run Length Encoding

ZRLE compilation replaces runs of zeros with one zero and a number following indicating how many zeros. If the value is any other than zero, that number is simply placed in the array as per normal. Bytes seem to work particularly well for as the unit size for PVS ZRLE compression.

The example,

5	0	0	0	0	0	0	0	33	0	0	0	0	0	0	8
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Where, each cell is a byte.

Could be compressed down to,

5	0	7	33	0	6	8	0	33	1	0	14
---	---	---	----	---	---	---	---	----	---	---	----

Which means the data is only 18.75% of the original size.

The ZRLE compression algorithm for ZRLE looks something like this:

```
//Assumes that list of portals is already in a bit list
//NewList is the newly generated list (like a pointer list)
//OldList is the old bit list of some type we are compressing (like a pointer list)
Create NewList
WHILE (not at end of Old List)
  IF [OldList = 0]
    NewList = 0 //Set indication
    NewList goes to next byte
    NewList = 0 //Set counter to Zero
  DO
    OldList goes to next byte
    //Increase count
    NewList = NewList + 1
  WHILE (OldList = 0)
END IF
WHILE [OldList != 0]
  NewList = OldList
  [OldList goes to next byte]
  [NewList goes to next byte]
```



```
END WHILE
```

Listing 25. ZRLE algorithm

See 17.10 for ZRLE source code.

## 17 Appendix 4 – Compiler Sample Code (C++)

This section contains important C++ code segments used in compilation of the BSP tree. The symbols “...” are used to remove irrelevant portions of code. However the complete code can be found in “”. Note that the code presented in this section may have been simplified at the cost of efficiency to improve readability.

### 17.1 Class Properties

In order to understand the following algorithms a brief description of the class properties are given below.

PolygonV is represented by a list of vertices. It is generally used as a temporary polygon where other details such as texture coordinates are not important.

```
class PolygonV
{
public:
    std::vector<Vector> Vertex; //List of vertices using an STL vector
    ...
}
```

Listing 26.

ScenePolySLite is a polygon that may be used during rendering; therefore it contains more information than PolygonV. Vertex information, texture information and normal information are all stored in the global arrays VertexList, TextList and NormalList, respectively. VertexIndexList, TextIndexList and NormalIndexList are all arrays of indices to the relevant global array.

```
class ScenePolySLite : public SceneObject
{
public:
    unsigned int Vertices; //Is the amount of faces
    unsigned int *VertexIndexList; //A list of indices to vertices
    unsigned int *TextIndexList; //A list of indices to UV texture coordinates
    unsigned int *NormalIndexList; //A list of indices to normals
    ...
}
```

Listing 27.

```
class Plane
{
private:
    Vector Normal; //Normal from a list (3)
    float Dist; //The planes distance
    unsigned int Align; //What axis are used
    ...
}
```

```
}
```

Listing 28.

```
class Vector
{
public:
    float    X, Y, Z; //3D position in space or direction
    ...
}
```

Listing 29.

```
class PortalOneWay
{
public:
    PolygonV Polygon;
    Plane ThePlane;    //The plane
    Node* Leaf;        //Nodes owner leaf (node in front)
    int Priority;       //Amount of might-sees
    BYTE* InFront;     //Array of might-sees (level 1)
    int Mark;          //Portals to reduce repeat calculations
    ...
}
```

Listing 30.

## 17.2 Polygon Classification

Potential outcomes of polygon classification are:

```
enum Side
{
    FRONT = 1,
    BACK = 2,
    SPAN = 4,
    ONPLANE = 8,
    BACK_N_FRONT = SPAN | BACK,
    FRONT_N_BACK = SPAN | FRONT,
    ON_BACK_N_FRONT = BACK_N_FRONT | ONPLANE,
    ON_FRONT_N_BACK = FRONT_N_BACK | ONPLANE,
};
```

Listing 31.

There are 4 main outcomes, FRONT, BACK, SPAN and ONPLANE. However, to help with splitting information about the split's side is retained in BACK\_N\_FRONT, FRONT\_N\_BACK, ON\_BACK\_N\_FRONT and ON\_FRONT\_N\_BACK. If the spanning plane is BACK\_N\_FRONT or ON\_BACK\_N\_FRONT it means that vertices before the crossover point (point of split) are on the backside of the plane, otherwise vertices before are on the front side. The ON in the name means that there is an intersection point on the plane.

This gets the side of the plane for PolygonV. Note that iterator can be used later in the splitting algorithm. Epsilon, generally 0.1, is used to provide a bit of tolerance for floating point errors.

```

/*!
 * Gets the side which this polygon is on of the given plane
 */
Side PolygonV::getSideOfPlane(const Plane& testPlane, Vector* & iterator, float epsilon)
{
    //If completely in front or behind, all vertices (v) must be tested,
    //2v-1 and 2v respectfully = O(v).
    //If on plane completely (worst case, rare, but at least one per partition
    //(the partition itself),
    //3v = O(v)
    //If spanning a maximum of 2v = O(v), and a minimum of constant 3 = O(1)

    //TODO, if on one side, and counts > 2 onplanes, then the polygon must be
    //on that side.
    unsigned int n = 2;
    float result;
    int Vertices = Vertex.size();

    //Initiate the iterator
    iterator = &Vertex[0];

    Vector *endIterator = &Vertex[Vertices-1];

    //Loop through each vertex
    //Note that this loop is split-friendly, that is, if there is a split
    //it'll exit early, at the cost of a small additional overhead.
    do
    {
        result = testPlane.computeSide(*(iterator++));

        //Work out which side it is on (if any)
        if (result > epsilon)
        {
            //Determine if that changes
            do
            {
                result = testPlane.computeSide(*iterator);
                if (result <= epsilon) //if onplane or FRONT_N_BACK
                {
                    if (result < -epsilon) //FRONT_N_BACK
                        return FRONT_N_BACK;
                    else //ONPLANE
                    {
                        //Test the next iterator (if possible)
                        if (iterator==endIterator)
                            return FRONT;

                        result = testPlane.computeSide(++iterator);
                        return (result < -epsilon)?ON_FRONT_N_BACK:FRONT;
                    }
                }
            } while (iterator++!=endIterator);

            return FRONT;
        }
        else if (result < -epsilon)

```

```

{
    //Determine if that changes
    do
    {
        result = testPlane.computeSide(*iterator);
        if (result >= -epsilon) //if onplane or BACK_N_FRONT
        {
            if (result > epsilon) //BACK_N_FRONT
                return BACK_N_FRONT;
            else //ONPLANE
            {
                //Test the next iterator (if possible)
                if (iterator!=endliterator)
                    return BACK;

                result = testPlane.computeSide(*(++iterator));
                return (result > epsilon)?ON_BACK_N_FRONT:BACK;
            }
        }

    } while (iterator++!=endliterator);

    return BACK;
}
} while (--n); //When onplane reaches 2 it must be on one side or the other

//At this point it can only be FRONT, BACK or ONPLANE, not SPAN

//Determine if it's back or front
do
{
    result = testPlane.computeSide(*iterator);

    //Work out which side it is on (if any)
    if (result > epsilon)
        return FRONT;
    else if (result < -epsilon)
        return BACK;
} while (iterator++!=endliterator);

//ONPLANE is returned when nothing else if found
return ONPLANE;
}

```

Listing 32.

Used for classifying a vertex against the plane. If the result of computeSide equals zero then the vertex is on plane, otherwise it is on one side or the other.

```

float Plane::computeSide(const Vector &vertex) const
{
    return vertex.dot(Normal) + Dist;
}

```

Listing 33.

## 17.3 Polygon Intersection

This intersection for the Plane object works out the intersection between two points for the given plane.

```

Vector Plane::intersect(const Vector &vertex1, const Vector &vertex2) const
{
    //Computes the intersection point of the line
    //from point A to point B with the partition
    //plane. This is a simple ray-plane intersection.
    //Insert/Create intersection into both sides.

    Vector temp = vertex2 - vertex1;

    //Work out gradient
    float m = (-Dist - vertex1.dot(Normal))/temp.dot(Normal);

    //Work out resulting position
    return vertex1 + temp * m;
}

```

Listing 34.

## 17.4 Computing the Plane

First make sure that the three points aren't aligned.

```

/*!
 * Computes the plane for this polygon, note this should only ever be called once!
 */
Plane ScenePolySLite::computePlane()
{
    Vector tri[3];

    if (Vertices > 3) //Make sure it's not a straight line (this type of polygon should probably be neutralized)
    {
        const float INTERCEPT_ROUND = 1024.0f;

        unsigned int index = 2;
        Vector pos1 = VertexList[VertexIndexList[0]].Vertex - VertexList[VertexIndexList[1]].Vertex;
        float length = pos1.mag();
        while (length == 0)
        {
            pos1 = VertexList[VertexIndexList[index-1]].Vertex - VertexList[VertexIndexList[index]].Vertex;
            length = pos1.mag();
            if (++index == Vertices) return Plane(Vector(0,0,0), 0); //Should only reach here if it's really a dot.
        }

        tri[0] = VertexList[VertexIndexList[index-2]].Vertex;
        tri[1] = VertexList[VertexIndexList[index-1]].Vertex;

        pos1 /= length; //Normalize

        //Generalize
        pos1.X = int(pos1.X * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
        pos1.Y = int(pos1.Y * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
        pos1.Z = int(pos1.Z * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
    }
}

```

```

Vector pos2;
do
{
    pos2 = VertexList[VertexIndexList[index-1]].Vertex - VertexList[VertexIndexList[index]].Vertex;
    length = pos2.mag();

    while (length == 0)
    {
        if (++index == Vertices) return Plane(Vector(0,0,0), 0); //Should only reach here if it's really a dot.
        pos2 = VertexList[VertexIndexList[index-1]].Vertex - VertexList[VertexIndexList[index]].Vertex;
        length = pos2.mag();
    }

    pos2 /= length; //Normalize

    //Generalize
    pos2.X = int(pos2.X * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
    pos2.Y = int(pos2.Y * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
    pos2.Z = int(pos2.Z * INTERCEPT_ROUND + .5f) / INTERCEPT_ROUND;
    index++;
} while (pos2 == pos1);

tri[2] = VertexList[VertexIndexList[index-1]].Vertex;
}
else
{
    tri[0] = VertexList[VertexIndexList[0]].Vertex;
    tri[1] = VertexList[VertexIndexList[1]].Vertex;
    tri[2] = VertexList[VertexIndexList[2]].Vertex;
}

pair<Vector, float> tempPlane = getPlane(tri);
return Plane(tempPlane.first, tempPlane.second);
}

```

Listing 35.

Then the plane is computed. In this example floating point is rounded to help:

1. find axis aligned planes, which are generally better for BSP trees.
2. grid align planes that are probably the same but appear not to be due to some floating point error.

```

const float NORMAL_EPSILON = .00001f;
const double FLOAT_NORMAL_ROUNDING = 65536.0;
const double FLOAT_ROUNDING = 1024.0;
/*
 * Computes the plane for a polygon.
 * \param triangleList is a array of vectors(3), one of each
 * vertex on the triangle. The order of points will affect the winding.
 * \return Returns the face normal + distance for the polygon given.
 */
pair<Vector, float> getPlane(Vector* TriangleList)
{
    //Note that hi-precision is used, to help reduce floating point errors
    //This function should only be called once per polygon, so the extra processing should be no big problem.

    //Vector returnNorm; //Returned

```

```

double returnNorm[3];
double normalizeGL[3];
double U[3];
double V[3];

U[0] = TriangleList[1].X - TriangleList[0].X;
U[1] = TriangleList[1].Y - TriangleList[0].Y;
U[2] = TriangleList[1].Z - TriangleList[0].Z;

V[0] = TriangleList[2].X - TriangleList[0].X;
V[1] = TriangleList[2].Y - TriangleList[0].Y;
V[2] = TriangleList[2].Z - TriangleList[0].Z;

//Cross product
normalizeGL[0] = U[1] * V[2] - U[2] * V[1];
normalizeGL[1] = U[2] * V[0] - U[0] * V[2];
normalizeGL[2] = U[0] * V[1] - U[1] * V[0];

// get the length of the vector sqrt(Dot product)
double length = sqrt(normalizeGL[0] * normalizeGL[0] + normalizeGL[1] * normalizeGL[1] + normalizeGL[2] * normalizeGL[2]);

float D;

if (length != 0)
{
    //Normalize
    returnNorm[0] = normalizeGL[0] / length;
    returnNorm[1] = normalizeGL[1] / length;
    returnNorm[2] = normalizeGL[2] / length;

    //Work out the distance
    D = -(float)(returnNorm[0] * TriangleList[0].X + returnNorm[1] * TriangleList[0].Y + returnNorm[2] * TriangleList[0].Z);

    //Fix up some floating point errors (axis align are important planes)
    if (fabsf((float)returnNorm[0]) > 1 - NORMAL_EPSILON ||
        ((float)returnNorm[1] < NORMAL_EPSILON && (float)returnNorm[2] < NORMAL_EPSILON &&
         (float)returnNorm[1] > -NORMAL_EPSILON && (float)returnNorm[2] > -NORMAL_EPSILON))
    {returnNorm[0] = (returnNorm[0]>0)?1:-1; returnNorm[1] = 0; returnNorm[2] = 0;}
    else if (fabsf((float)returnNorm[1]) > 1 - NORMAL_EPSILON ||
        ((float)returnNorm[0] < NORMAL_EPSILON && (float)returnNorm[2] < NORMAL_EPSILON &&
         (float)returnNorm[0] > -NORMAL_EPSILON && (float)returnNorm[2] > -NORMAL_EPSILON))
    {returnNorm[0] = 0; returnNorm[1] = (returnNorm[1]>0)?1:-1; returnNorm[2] = 0;}
    else if (fabsf((float)returnNorm[2]) > 1 - NORMAL_EPSILON ||
        ((float)returnNorm[0] < NORMAL_EPSILON && (float)returnNorm[1] < NORMAL_EPSILON &&
         (float)returnNorm[0] > -NORMAL_EPSILON && (float)returnNorm[1] > -NORMAL_EPSILON))
    {returnNorm[0] = 0; returnNorm[1] = 0; returnNorm[2] = (returnNorm[2]>0)?1:-1;}
    else
    {
        //Generalize (to create approximate normals that are near each other).
        returnNorm[0] = int(returnNorm[0] * FLOAT_NORMAL_ROUNDING + .5f)/FLOAT_NORMAL_ROUNDING;
        returnNorm[1] = int(returnNorm[1] * FLOAT_NORMAL_ROUNDING + .5f)/FLOAT_NORMAL_ROUNDING;
        returnNorm[2] = int(returnNorm[2] * FLOAT_NORMAL_ROUNDING + .5f)/FLOAT_NORMAL_ROUNDING;

        //Fix up some floating point errors (axis align are important planes)
        if (fabsf((float)returnNorm[0]) > 1 - NORMAL_EPSILON || ((float)returnNorm[1] < NORMAL_EPSILON && (float)returnNorm[2] < NORMAL_EPSILON &&
            (float)returnNorm[1] > -NORMAL_EPSILON && (float)returnNorm[2] > -NORMAL_EPSILON))
        {returnNorm[0] = (returnNorm[0]>0)?1:-1; returnNorm[1] = 0; returnNorm[2] = 0;}
        else if (fabsf((float)returnNorm[1]) > 1 - NORMAL_EPSILON || ((float)returnNorm[0] < NORMAL_EPSILON && (float)returnNorm[2] < NORMAL_EPSILON &&
            (float)returnNorm[0] > -NORMAL_EPSILON && (float)returnNorm[2] > -NORMAL_EPSILON))
        {returnNorm[0] = 0; returnNorm[1] = (returnNorm[1]>0)?1:-1; returnNorm[2] = 0;}
    }
}

```



```

        else if (fabsf((float)returnNorm[2]) > 1 - NORMAL_EPSILON || ((float)returnNorm[0] < NORMAL_EPSILON && (float)returnNorm[1] < NORMAL_EPSILON &&
            (float)returnNorm[0] > -NORMAL_EPSILON && (float)returnNorm[1] > -NORMAL_EPSILON))
        {returnNorm[0] = 0; returnNorm[1] = 0; returnNorm[2] = (returnNorm[2]>0)?1:-1;}
    }

    else
    {
        returnNorm[0] = returnNorm[1] = returnNorm[2] = 0;
        D = 0;
    }

    //Convert down to float vector
    Vector newNormal;
    newNormal.X = (float)returnNorm[0];
    newNormal.Y = (float)returnNorm[1];
    newNormal.Z = (float)returnNorm[2];

    return make_pair(newNormal, D);
}

```

Listing 36.

## 17.5 Determining if a Polygon is Small

Listing 37 returns true if a polygon is considered to small.

```

/*!
 * Used to determine if this polygon is too small.
 * \param epsilon the factor determining how small the object is.
 * \Return Returns true if this polygon is too small to be considered a polygon.
 */
bool PolygonV::isSmall(float epsilon)
{
    //For an object to not be small, it must have at least 3 good edges

    int Vertices = Vertex.size();

    Vector *CV1 = &Vertex[0],
            *CV2 = &Vertex[Vertices-1],
            temp;

    unsigned int n, goodEdge = 0;

    temp = *CV1 - *CV2;
    temp *= temp;
    if (temp.sum() > epsilon)
        goodEdge++;

    CV2 = &Vertex[1];
    Vector *CPEnd = &Vertex[Vertices];
    //Check the other edges
    for (n = 0; CV2 != CPEnd; CV1++, CV2++)
    {
        temp = *CV1 - *CV2;
        temp *= temp;
        if (temp.sum() > epsilon && ++goodEdge == 3)
            return false; //It's not too small
    }
}

```

```

    return true; //It's to small
}

```

Listing 37. isSmall

## 17.6 Plane Spanning Polygon

Listing 38 coverts polygonV into a polygon that spans the given plane.

```

/*
 * Creates a polygon that spans a plane. The size of the polygon is determined by
 * maxsize.
 */
void PolygonV::spanPoly(Plane &usePlane, float maxsize)
{
    pair<Vector, float> major = usePlane.getNormal().getNonMajorAxis();

    //Calculate u v and mid
    Vector u = major.first + usePlane.getNormal() * -major.second;
    u.normalize();
    u *= maxsize;
    Vector v = u.cross(usePlane.getNormal());

    Vector mid = -usePlane.getNormal() * usePlane.getDist();

    Vector mAddu = mid + u;
    Vector mSubu = mid - u;

    Vertex.resize(4);
    Vertex[0] = mAddu - v;
    Vertex[1] = mAddu + v;
    Vertex[2] = mSubu + v;
    Vertex[3] = mSubu - v;
}

```

Listing 38.

## 17.7 Split

Returns which side of the plane the given polygon is on. If the polygon is on two sides of the plane then the polygon is split into two new polygons.

```

/*
 * Splits the polygon in two (if necessary), returning the two new polygons.
 *
 * \param partition is the plane to use as the partition.
 * \param polyToSplit is the index to the triangle being split
 * \param newPolys is the index of the first new triangle, if a split
 * occurred the other new triangle can be found by adding 1.
 * \return Returns the side this polygon is on of the plane.
 */
Side PolygonV::checkNsplit(Plane& partition, PolygonV *newPolys, float epsilon)
{
    //This is an example of a checkNsplit function that uses the results from the check
    //to aid in splitting. There are many shorter versions of polygon
    //splitters on the web,
    //however this one takes a different approach (to what I've have seen

```

```

//at least). Polygons
//are treated as spans of front(F) and back(B) faces. The cross over
//point is found,
//and then that is used to determine where, and how, the polygon is split.
//Joel Anderson

//Extract the polygon we are splitting
Vector* crossover1;

Side theResult = getSideOfPlane(partition, crossover1, epsilon);
if (theResult & SPAN) //If spanning
{
    //Otherwise split it into two
    split(partition, (unsigned int)(crossover1 - &Vertex[0]), theResult, newPolys, epsilon);
    return SPAN;
}
else
{
    return theResult; //Otherwise return the result
}
}

```

Listing 39.

```

void PolygonV::split(Plane& partition, unsigned int crossover1, Side theResult, PolygonV *newPolys, float epsilon)
{
    unsigned int endIndex = Vertex.size()-1,
        startIndex = 0,
        p1End = crossover1, //polygon 1's end vertex
        p1VarSize = 0,
        p2Offset = 0,
        p2Size, //The amount of fixed vertices in polygon 2
        p1Size, //The amount of fixed vertices in polygon 1
        p1Start; //polygon 1's start vertex

    Vector stitchV1,
        stitchV12;

    stitchV1.X = FLT_MAX;
    PolygonV *pFront = &newPolys[0], //Front polygon
        *pBack = &newPolys[1]; //Back polygon, gets next index along

    Plane partitionCpy;
    if (theResult & BACK)
    {
        //Flip back and front
        pFront++; pBack--;

        //Inverse the plane if facing backwards
        partitionCpy = partition.inverse();
    }
    else
    {
        partitionCpy = partition;
    }

    //Determine which side the last vertex is on
    float result1 = partitionCpy.computeSide(Vertex[endIndex]);
    {
        if (result1 <= epsilon)
        {

```

```

//FFFB BB or FFFBB, etc... situation (F = front, B = back)
//FFF|BBB| or FFF|BB| //Where splits are |, between (crossover1-1)
//and crossover1, end array and start array

//The significant thing is
//1) String of F's and then B's

//>Do polygon 1 and then object2 (because of memory alignment)
p1Start = Vertex.size(); //polygon 1's start vertex (starts at the end of the list, because it wraps around)
p1Size = p1End + 1;
p2Size = Vertex.size() - p1End + 1; //The amount of fixed vertices in polygon 2

//>Determine if stitching is need, or a existing point can be used
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result1 >= -epsilon)
{
    //The last point is on the plane
    p1VarSize = 1; //Increase size
}
//If the first point is not on the plane
else if (partitionCpy.computeSide(Vertex[startIndex]) > epsilon)
{
    stitchVI = partitionCpy.intersect(Vertex[endIndex], Vertex[startIndex]);
    //Insert an intersection
    p1VarSize = 1; //Increase size
    p2Size++;
    p2Offset = 1;
}
else
{
    p2Size++;
    if (theResult & ONPLANE)
        p1Size--;
    p2Offset = 1;
}

//Allocate room for vertex array 1 and 2
pFront->Vertex.resize(p1Size+p1VarSize);
pBack->Vertex.resize(p2Size);

if (result1 >= -epsilon)
{
    pFront->Vertex[0] = Vertex[endIndex]; //Polygon 1
}
else if (stitchVI.X != FLT_MAX)
{
    pBack->Vertex[pBack->Vertex.size()-1] = pFront->Vertex[0] = stitchVI; //Polygon 1 and 2
}
else
{
    pBack->Vertex[pBack->Vertex.size()-1] = Vertex[startIndex]; //Polygon 2
}
}
else
{
    //FFBBBBF, FFFBBFF, etc... situation (F = front, B = back)
    //FF|BBB|F //Where splits are |, between (crossover1-1) and crossover1, between (crossover2-1) and crossover2

    //The significant thing is
    //1) that it wraps around at the end
    //2) String of F's and then B's and then F again

```

```

//3) Both intersections are at places where the first vertex in the crossover will be the only possible on-plane point

//>Determine the position of the second crossover, which hasn't been found yet.

//Find the end of the sequence
float result2;
unsigned int crossover2; //The second crossover starts it's search at crossover1
for (crossover2 = crossover1; (result2 = partitionCpy.computeSide(Vertex[++crossover2])) < -epsilon; ) //

p1Start = crossover2; //polygon 1's start vertex
p2Size = p1Start - p1End + 2; //The amount of fixed vertices in polygon 2
p1Size = Vertex.size() - p2Size + 3; //The amount of fixed vertices in polygon 1

//Determine if stitching is need
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result2 > epsilon) //The points are on two sides of the plane, therefore the intersection needs to be found
{
    //Create an intersection
    stitchV1 = partitionCpy.intersect(Vertex[crossover2-1], Vertex[crossover2]); //Polygon 1
    p1VarSize = 1;
    p2Offset = 1;
}

//Allocate room for vertex array 1 and 2
pFront->Vertex.resize(p1Size+p1VarSize);
pBack->Vertex.resize(p2Size);

//Copy each vertex from the start of the sequence to the end of the array
dupEachVertex(p1Start, Vertex.size() - p1Start, &pFront->Vertex[p1VarSize]); //Polygon 1

//Determine if stitching is need
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result2 > epsilon) //The points are on two sides of the plane, therefore the intersection needs to be found
{
    //Insert an intersection
    pBack->Vertex[pBack->Vertex.size()-1] = pFront->Vertex[0] = stitchV1; //Polygon 1 and 2
}
}

//Copy each vertex up to the crossover point, for polygon 1
dupEachVertex(0, p1End, &pFront->Vertex[p1VarSize + Vertex.size() - p1Start]); //Polygon 1

//Copy the other points on to polygon 2
dupEachVertex(p1End, p2Size-1-p2Offset, &pBack->Vertex[1]); //Polygon 2

//Determine if stitching is need, or a existing point can be used
//First check that it's not a onplane point (in which case simply duplicate that point)
//The next point is on the plane, which makes calculations easy, just duplicate that vertex
stitchV12 = (theResult & ONPLANE)?Vertex[crossover1-1]:partition.intersect(Vertex[crossover1-1], Vertex[crossover1]);
pFront->Vertex[pFront->Vertex.size()-1] = pBack->Vertex[0] = stitchV12; //Polygon 1 and 2

//3< vertices is not a polygon
ASSERT(pFront->Vertex.size() > 2);
ASSERT(pBack->Vertex.size() > 2);

```

```
}
```

Listing 40.

## 17.8 Trim

Checks the side of a polygon and if it's spanning it is trimmed.

```
Side PolygonV::checkNTrim(Plane& partition, float epsilon)
{
    //Extract the polygon we are splitting
    Vector* crossover1;

    Side theResult = getSideOfPlane(partition, crossover1, epsilon);
    if (theResult & SPAN) //If spanning
    {
        trim(theResult, partition, (unsigned int)(crossover1 - &Vertex[0]), epsilon);
        return SPAN;
    }
    else
    {
        return theResult; //Otherwise return the result
    }
}
```

Listing 41.

Trims one side of given polygon.

```
void PolygonV::trim(Side theResult, Plane partition, unsigned int crossover1, float epsilon)
{
    assert(theResult & Side.SPAN);
    //Do polygon 1
    unsigned int endIndex = Vertex.size()-1,
        startIndex = 0,
        p1End = crossover1,           //polygon 1's end vertex
        p1VarSize = 0,
        p1Size,                       //The amount of fixed vertices in polygon 1
        p1Start,                      //polygon 1's start vertex
        p1Offset=0;

    Vector stitchV1;                  //unsigned int.max should never be reached, it's used to indicate that there was never an intersection
    stitchV1.X = FLT_MAX;             //FLT_MAX should never be reached, it's used to indicate that there was never an intersection

    Vector* newV1LO = NULL;

    //Inverse the plane if facing backwards
    //Plane partitionCpy = (theResult & BACK)?partition.inverse():partition;
    // ?partition.inverse():partition;

    //Determine which side the last vertex is on
    float result1 = partition.computeSide(Vertex[endIndex]);
    if (theResult & BACK)
    {
        if (result1 >= -epsilon)
        {
            //FFFFBBB or FFFBBB, etc... situation (F = front, B = back)
            //FFF[B]BB or FFF[B]BB //Where splits are |, between (crossover1-1) and crossover1, end array and start array
        }
    }
}
```

```

//The significant thing is
//1) String of F's and then B's

//>Do polygon 1 and then object2 (because of memory alignment)
p1Size = Vertex.size() - p1End + 1; //The amount of fixed vertices in polygon 2

//>Determine if stitching is need, or a existing point can be used
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result1 > epsilon)
{
    //The last point is on the plane
    if (partition.computeSide(Vertex[startIndex]) < -epsilon)
        stitchVI = partition.intersect(Vertex[endIndex], Vertex[startIndex]);
    p1Size++;
    p1Offset = 1;
}

//Allocate room for vertex array
newVILO = new Vector[p1Size];

if (result1 > epsilon)
    newVILO[p1Size-1] = (stitchVI.X != FLT_MAX)?stitchVI:Vertex[startIndex];
}
else
{
    //FFBBBBF, FFFBBFF, etc... situation (F = front, B = back)
    //FF[BBB]F //Where splits are |, between (crossover1-1) and crossover1, between (crossover2-1) and crossover2

    //The significant thing is
    //1) that it wraps around at the end
    //2) String of F's and then B's and then F again
    //3) Both intersections are at places where the first vertex in the crossover will be the only possible on-plane point

    //>Determine the position of the second crossover, which hasn't been found yet.
    //Find the end of the sequence
    float result2;
    unsigned int crossover2 = crossover1; //The second crossover starts it's search at crossover1
    for (; (result2 = partition.computeSide(Vertex[++crossover2])) > epsilon;) { }; //

    p1Size = crossover2 - p1End + 2; //The amount of fixed vertices in polygon 2

    //Allocate room for vertex array
    newVILO = new Vector[p1Size];

    //Determine if stitching is need
    //First check that it's not a onplane point (in which case simply duplicate that point)
    if (result2 < -epsilon) //The points are on two sides of the plane, therefore the intersection needs to be found
    {
        //Create an intersection
        stitchVI = partition.intersect(Vertex[(crossover2-1)], Vertex[crossover2]); //Polygon 1
        p1Offset = 1;

        newVILO[p1Size-1] = stitchVI;
    }
}

//Copy the other points on to the polygon
dupEachVertex(p1End, p1Size-1-p1Offset, &newVILO[1]);

//Determine if stitching is need, or a existing point can be used

```

```

//First check that it's not a onplane point (in which case simply duplicate that point)
//The next point is on the plane, which makes calculations easy, just duplicate that vertex
newVILO[0] = (theResult & ONPLANE)?Vertex[crossover1-1]:partition.intersect(Vertex[crossover1-1], Vertex[crossover1]); //Polygon 1 and 2
p1VarSize = 0;
}
else
{
    if (result1 <= epsilon)
    {
        //FFFFBBB or FFFBB, etc... situation (F = front, B = back)
        //FFF|BBB| or FFF|BB| //Where splits are |, between (crossover1-1) and crossover1, end array and start array

        //The significant thing is
        //1) String of F's and then B's

        //>Do polygon 1 and then object2 (because of memory alignment)
        p1Start = Vertex.size(); //polygon 1's start vertex (starts at the end of the list, because it wraps around)
        p1Size = p1End + 1;

        //>Determine if stitching is need, or a existing point can be used
        //First check that it's not a onplane point (in which case simply duplicate that point)
        if (result1 >= -epsilon)
        {
            //The last point is on the plane
            p1VarSize = 1; //Increase size
        }
        //If the first point is not on the plane
        else if (partition.computeSide(Vertex[startIndex]) > epsilon)
        {
            stitchVI = partition.intersect(Vertex[endIndex], Vertex[startIndex]);
            //Insert an intersection
            p1VarSize = 1; //Increase size
        }
        else
        {
            if (theResult & ONPLANE)
                p1Size--;
        }

        newVILO = new Vector[p1Size+p1VarSize];

        if (result1 >= -epsilon)
        {
            newVILO[0] = Vertex[endIndex]; //Polygon 1
        }
        else if (stitchVI.X != FLT_MAX)
        {
            newVILO[0] = stitchVI; //Polygon 1 and 2
        }
    }
    else
    {
        //FFBBBBF, FFFBBFF, etc... situation (F = front, B = back)
        //FF|BBB|F //Where splits are |, between (crossover1-1) and crossover1, between (crossover2-1) and crossover2

        //The significant thing is
        //1) that it wraps around at the end
        //2) String of F's and then B's and then F again
        //3) Both intersections are at places where the first vertex in the crossover will be the only possible on-plane point

        //>Determine the position of the second crossover, which hasn't been found yet.
    }
}

```



```

//Find the end of the sequence
float result2;
unsigned int crossover2; //The second crossover starts its search at crossover1
for (crossover2 = crossover1; (result2 = partition.computeSide(Vertex[++crossover2])) < -epsilon;) { }; //

p1Start = crossover2; //polygon 1's start vertex
p1Size = Vertex.size() - p1Start + p1End + 1; //The amount of fixed vertices in polygon 1

//Determine if stitching is need
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result2 > epsilon) //The points are on two sides of the plane, therefore the intersection needs to be found
{
    //Create an intersection
    stitchV1 = partition.intersect(Vertex[crossover2-1], Vertex[crossover2]); //Polygon 1
    p1VarSize = 1;
}

//Allocate room for vertex array
newVILO = new Vector[p1Size+p1VarSize];

//Copy each vertex from the start of the sequence to the end of the array
dupEachVertex(p1Start, Vertex.size() - p1Start, &newVILO[p1VarSize]);

//Determine if stitching is need
//First check that it's not a onplane point (in which case simply duplicate that point)
if (result2 > epsilon) //The points are on two sides of the plane, therefore the intersection needs to be found
{
    //Insert an intersection
    newVILO[0] = stitchV1; //Polygon 1 and 2
}
}

//Copy each vertex up to the crossover point, for polygon 1
dupEachVertex(0, p1End, &newVILO[p1VarSize + Vertex.size() - p1Start]);

//Determine if stitching is need, or a existing point can be used
//First check that it's not a onplane point (in which case simply duplicate that point)
//The next point is on the plane, which makes calculations easy, just duplicate that vertex
newVILO[p1Size+p1VarSize-1] = (theResult & ONPLANE)?Vertex[crossover1-1]:partition.intersect(Vertex[(crossover1-1)], Vertex[crossover1]);
}

//Set the polygon to it's new vertex list
Vertex.resize(p1Size+p1VarSize);
copy( &newVILO[0], &newVILO[p1Size+p1VarSize], Vertex.begin() );

delete [] newVILO;

//3< vertices is not a polygon
assert(Vertex.size() > 2);
}

```

Listing 42.

## 17.9 Portal Generation

This code creates portals from the BSP tree. There may be some bad portals created after calling createPortals to generate the portals therefore filterPortals should be called to find and remove any bad portals that can be found.

```

int BSPStandard::pushPortal(int& list, int toAdd, int side)
{
    PortalList[toAdd].Link[side] = list;
    return list = toAdd;
}

```

```

void BSPStandard::addPortal(int portal, Node* node, int side)
{
    Portal* currentPortal = &PortalList[portal];
    currentPortal->Parent[side] = node;
    currentPortal->Link[side] = node->Portals;
    node->Portals = portal;
}

```

```

void BSPStandard::popPortal(Portal &parentPortal, int side)
{
    parentPortal.Parent[side]->Portals = parentPortal.Link[side];
    parentPortal.Parent[side] = NULL; //Needed?
}

```

```

const float PORTAL_EPSILON = 0.001f;
void BSPStandard::createSpanPortal(Node* node)
{
    //Note that the algorithm below is similar to the Quake3 Radiant algorithm,
    //although the code is different
    int newPortalIndex = newPortal(1);
    Portal* newPortal = &PortalList[newPortalIndex];
    Node* childNode = node;

    newPortal->Polygon.spanPoly(PlaneListQuick[node->ThePlane]);

    Plane* currentPlane;
    Node* currentNode;

    Plane p;

    //Trim polygon/portal by it's parents
    int partitionNode;
    for (partitionNode = childNode->Parent; partitionNode != ROOT_PARENT; partitionNode = currentNode->Parent)
    {
        currentNode = &NodeListFixed[partitionNode];
        currentPlane = &PlaneListQuick[currentNode->ThePlane];

        //Determine which side
        p = (&NodeListFixed[currentNode->Back] == childNode)?currentPlane->inverse():*currentPlane;
        newPortal->Polygon.checkNTTrim(p);

        childNode = currentNode;
    }

    if (newPortal->Polygon.isSmall() || NodeListFixed[node->Back].ThePlane == SOLID_NODE)
    {
        //Free memory (should be cheap as it's not the top)
        PortalList.resize(PortalList.size() - 1);
        debugSmallPortalCount++;
    }
    else

```

```

{
    newPortal->ThePlane = node->ThePlane;
    addPortal(newPortalIndex, &NodeListFixed[node->Back], 1);
    addPortal(newPortalIndex, &NodeListFixed[node->Front], 0);
}
}

```

```

void BSPStandard::partitionPortal(Node* parentNode) //int theNode)
{
    //Create the initial portal
    createSpanPortal(parentNode);

    Plane *parentPlane = &PlaneListQuick[parentNode->ThePlane],
        *currentPlane;

    Portal* currentPortal;
    int portalIndex, link, side;

    PolygonV newPolygon[2];

    //Divide each portal at this node, by this node.
    //If it ends up on one side, then send it down that side, otherwise send it down both.
    for (portalIndex = parentNode->Portals; portalIndex != PORTAL_END; portalIndex = link)
    {
        currentPortal = &PortalList[portalIndex];
        currentPlane = &PlaneListQuick[currentPortal->ThePlane];

        side = (currentPortal->Parent[0] == parentNode)?0:1;
        link = currentPortal->Link[side];
        popPortal(*currentPortal, side); //remove it from the stack, since it's the one currently being traversed

        switch (currentPortal->Polygon.checkNSplit(*parentPlane, newPolygon, PEPSILON * 2.0f))
        {
            case SPAN:
            {
                if (newPolygon[0].isSmall())
                {
                    if (newPolygon[1].isSmall())
                    {
                        //Free the memory used for those objects (two small objects)
                        popPortal(*currentPortal, !side);
                        debugSmallPortalCount+=2;
                    }
                    else
                    {
                        currentPortal->Polygon = newPolygon[1]; //Update polygon
                        addPortal(portalIndex, &NodeListFixed[parentNode->Back], side);
                        debugSmallPortalCount++;
                    }
                }
            }
            else if (newPolygon[1].isSmall() || NodeListFixed[parentNode->Back].ThePlane == SOLID_NODE)
            {
                currentPortal->Polygon = newPolygon[0]; //Update polygon
                addPortal(portalIndex, &NodeListFixed[parentNode->Front], side);
                debugSmallPortalCount++;
            }
            else //Both are ok sizes
            {
                //Two new polygons have been created (the old one should be deleted now, but is currently not).
            }
        }
    }
}

```

```

    int newPortalIndex = newPortal(1);
    Portal* newPortal = &PortalList[newPortalIndex];

    currentPortal = &PortalList[portalIndex]; //Grab a fresh copy of the current portal in case it has changed on a realloc

    //Copy over common information
    currentPortal->Polygon = newPolygon[0];
    *newPortal = *currentPortal; //Copy over information
    newPortal->Polygon = newPolygon[1]; //currentPortal->Polygon;

    addPortal(portalIndex, &NodeListFixed[parentNode->Front], side);
    addPortal(newPortalIndex, &NodeListFixed[parentNode->Back], side);
    addPortal(newPortalIndex, currentPortal->Parent[!side], !side);
}
}
break;
case BACK:
    addPortal(portalIndex, &NodeListFixed[parentNode->Back], side);
break;
case FRONT:
    addPortal(portalIndex, &NodeListFixed[parentNode->Front], side);
break;
case ONPLANE:
    {
        //Add it to the back and front list
        int newPortalIndex = newPortal(1);
        Portal* newPortal = &PortalList[newPortalIndex];
        //Copy over common information
        *newPortal = *currentPortal; //Copy over information

        addPortal(portalIndex, &NodeListFixed[parentNode->Front], side);
        addPortal(newPortalIndex, &NodeListFixed[parentNode->Back], side);
        addPortal(newPortalIndex, currentPortal->Parent[!side], !side);
    }
break;
}
}

//Empty portals at this node (they have been moved down the next list)
parentNode->Portals = PORTAL_END;
}

```

```

void BSPStandard::createPortals()
{
    PortalList.clear();
    PortalList.reserve(NodeList.size()); //Resize the portals in one block, to speed up allocation

    //Traverses the entire tree using a loop, back nodes are visited first.
    //This only works because nodes are ordered back-to-front. This is optimized
    //further by having the leaves at the end of the tree.

    int n, size = NonLeafNodeCount;
    for (n=0; n<size; n++)
    {
        //If (NodeList[n].ThePlane >= 0) //If not a (SOLID or EMPTY) LEAF NODE
        partitionPortal(&NodeListFixed[n]);
    }
}

```

```

bool BSPStandard::isGoodPortal(const Portal& toCheck) const
{
    if (toCheck.Parent[0] && toCheck.Parent[0]->ThePlane == LEAF_NODE && toCheck.Parent[1]->ThePlane == LEAF_NODE)
    {
        //Remove infinite portals (note that some infinite portals could be useful for looking out windows)

        const float BB_APPROX = 1.0f;

        AABB Lite bound = NodeListFixed[0].BB; //The bounding box at node 0 should hold the entire world
        bound.Min.X -= BB_APPROX;
        bound.Min.Y -= BB_APPROX;
        bound.Min.Z -= BB_APPROX;
        bound.Max.X += BB_APPROX;
        bound.Max.Y += BB_APPROX;
        bound.Max.Z += BB_APPROX;

        int n;
        for (n = toCheck.Polygon.Vertex.size(); --n>=0; )
        {
            if (!bound.fitWithin(toCheck.Polygon.Vertex[n])) //Part of this array could be pre-extracted.
                break;
        }

        return (n == -1);
    }
    else
    {
        return false;
    }
}

```

```

void BSPStandard::filterPortals()
{
    //1) Create one directional portals at each node
    int portalIndex, link, side, idcount=0;
    int n, offset = NonLeafNodeCount,
        size = offset + FrontLeafCount,
        goodCount = 0;

    vector<bool> goodPortals; //LUT for good portals
    goodPortals.reserve(FrontLeafCount); //This may get really big

    //First work out the amount of good portals so we can block allocate
    //Note that goodPortals will require block allocations but
    //1) a bit takes up much less memory then PortalOneWay
    //2) PortalOneWay needs to be in a sequential array for ID.
    for (n = offset; n < size; n++)
    {
        Node* currentLeaf = &NodeListFixed[n];

        //Traverse each portal at that node to get a count
        for (portalIndex = currentLeaf->Portals; portalIndex != PORTAL_END; portalIndex = link)
        {
            Portal* thePortal = &PortalList[portalIndex];
            side = (thePortal->Parent[0] == currentLeaf)?0:1;
            link = thePortal->Link[side];
        }
    }
}

```

```

    if (isGoodPortal(*thePortal))
    {
        goodCount++;
        goodPortals.push_back(true);
    }
    else
    {
        debugBadPortalCount++;
        goodPortals.push_back(false);
    }
}
}

PortalCount = goodCount;

OWPList = new PortalOneWay[goodCount];
vector<bool>::iterator good = goodPortals.begin();
PortalOneWay* iterP = OWPList;

MaxFacePCount = 0;

PortalPVSSize = (goodCount+7)>>3;
GlobalInFrontList = new BYTE[PortalPVSSize*PortalCount];
ZeroMemory(GlobalInFrontList, PortalPVSSize*PortalCount); //Mark all bits zero (unknown yet)
BYTE* glIter = GlobalInFrontList;

for (n = offset; n < size; n++)
{
    Node* currentLeaf = &NodeListFixed[n];

    currentLeaf->FacePortals = iterP;
    currentLeaf->FacePCount = 0;

    for (portalIndex = currentLeaf->Portals; portalIndex != PORTAL_END; portalIndex = link)
    {
        Portal* thePortal = &PortalList[portalIndex];
        side = (thePortal->Parent[0] == currentLeaf)?0:1;
        link = thePortal->Link[side];

        if (*good)
        {
            iterP->Polygon = thePortal->Polygon;
            //flip the plane if the portal is facing the other direction
            iterP->ThePlane = (side==0)?PlaneListQuick[thePortal->ThePlane].inverse():PlaneListQuick[thePortal->ThePlane];
            iterP->Leaf = thePortal->Parent[side];

            iterP->Priority = 0; //Unknown yet

            //Allocate space for the PVS
            iterP->InFront = glIter;

            glIter += PortalPVSSize;

            iterP++; //Go to next portal
            currentLeaf->FacePCount++;
        }
        good++;
    }
}

if (currentLeaf->FacePCount > MaxFacePCount)

```

```

        MaxFacePCount = currentLeaf->FacePCount;
    }

    Node *currentLeaf, *endLeaf = &NodeListFixed[NonLeafNodeCount + FrontLeafCount];
    PortalOneWay *sourcePortal,
        *endSourcePortal;

    int maxPrint = 6;

    Vector minmax;
    //Fill in lead info
    for (currentLeaf=&NodeListFixed[NonLeafNodeCount]; currentLeaf!=endLeaf; ++currentLeaf)
    {
        if (currentLeaf->FacePCount > 0)
        {
            minmax = currentLeaf->FacePortals[0].Polygon.Vertex[0];
            currentLeaf->BB.set(minmax, minmax);
            endSourcePortal = &currentLeaf->FacePortals[currentLeaf->FacePCount];
            for (sourcePortal=&currentLeaf->FacePortals[0]; sourcePortal!=endSourcePortal; ++sourcePortal)
            {
                currentLeaf->BB.add(sourcePortal->Polygon);
            }
        }
        else //Most leaves should have at lease one portal!!!!
        {
            Vector minmax(0,0,0);
            currentLeaf->BB.set(minmax, minmax);
            if (--maxPrint > 0) //Don't print to many otherwise it may take to long
            {
                unsigned int leaf = (unsigned int) (currentLeaf - &NodeListFixed[NonLeafNodeCount]);
                cout << "Warning isolated room at leaf " << leaf << endl;
            }
        }
    }

    if (CleanDebug)
        PortalList.clear(); //No longer needed (used for debugging)
}

```

## 17.10 Zero Run Length Encoding

This ZRLE algorithm compresses the given array into the global array2.

```

int ZRLEcompress(BYTE* array, vector<BYTE>& array2, int size)
{
    int i, offset = array2.size(),
        bitsize = ((size-1)>>3)+1;

    for (i=0; i<bitsize; i++)
    {
        array2.push_back(array[i]);
        if (array[i] == 0) //RLE condition
        {
            int starti = i;
            while (i < size && i - starti <= 255 && array[++i] == 0 );

            array2.push_back(i - starti - 1); //set run length
            i--; //Go back one
        }
    }
}

```

```

    }
}
return offset;
}

```

Listing 43.

## 17.11 Link Generation

After the BSP tree has been built this algorithm generates links for each leaf. Links are found and stored in the NodeLinkMap so that they can easily be stored to file. When the links are loaded they will be stored at each leaf.

```

/*virtual*/ void BSPLink::generateLinks()
{
    //Find the linkages
    int leaf = 0;

    //Extract root node
    int maxNodeIndex = FrontLeafCount + NonLeafNodeCount;

    NodeLinkMap.resize(NonLeafNodeCount);

    //For each leaf
    Node *currentLeaf, *endLeaf = &NodeList[NonLeafNodeCount + FrontLeafCount];
    for (currentLeaf=&NodeList[NonLeafNodeCount]; currentLeaf!=endLeaf; ++currentLeaf)
    {
        //Mark root
        currentLeaf->markVisLeaf(++leaf, NodeListFixed, NonLeafNodeCount, maxNodeIndex, PVSLIST);

        //Assign this to something in the node (the commented out code is what
        //will essentially happen when the bsp file is loaded.)
        //currentLeaf->Link = (int)(NodeListFixed[0].getFirstDecision(frame, NodeListFixed) - NodeListFixed); //Re-use the SortValue field to hold linkage

        int nodeIndex = (int)(NodeListFixed[0].getFirstDecision(leaf, NodeListFixed) - NodeListFixed);
        NodeLinkMap[nodeIndex].push_back(leaf-1);
    }
}

```

Listing 44.

Finds and returns the first node in the tree where a decision needs to be made as to which node to traverse.

```

/*!
 * Returns the first marked node that has two marked child nodes
 */
BSPStandard::Node* BSPStandard::Node::getFirstDecision(int frame, Node* nodeListFixed)
{
    //Extract node list
    Node *current = this, *back, *front;
    do
    {
        back = &nodeListFixed[current->Back];
        front = &nodeListFixed[current->Front];
        if (back->Mark == frame)
        {
            if (front->Mark == frame) break; //Decision point found
            current = back;
        }
    }
}

```



```

        else
            current = front;
    }
    while ( current->ThePlane != LEAF_NODE );

    if (current->ThePlane == LEAF_NODE) //leaf (should occur only in non-linking rooms)
        return &nodeListFixed[current->Parent];

    return current;

```

Listing 45.

Note that the code for leaf marking can be found in 17.12.

## 17.12 Mark Leaf

This algorithm marks all the parent nodes for the given leaf.

```

/*!
 * Marks the leaf and all its parents
 * Assumes that root is marked
 */
void BSPStandard::Node::markLeaf(int frame, Node* nodeListFixed)
{
    Node* current = &nodeListFixed[this->Parent];
    while ( current->Mark != frame )
    {
        current->Mark = frame;
        current = &nodeListFixed[current->Parent];
    }
}

```

Listing 46.

This algorithm marks all the visible leaves and parents for the given leaf.

```

/*!
 * Marks all the visible leaves and their parents
 */
void BSPStandard::Node::markVisLeaf(int frame, Node* nodeList, int nonLeafNodeCount, int maxNodeIndex, std::vector<BYTE>& PVSList)
{
    int nodeIndex;
    BYTE* PVS;
    BYTE cPVS, cbit;

    nodeList[0].Mark = frame;

    this->markLeaf(frame, &nodeList[0]);

    //For each visible leaf in the current leaf's set
    PVS = &PVSList[this->PVSListIndex];

    nodeIndex = nonLeafNodeCount;
    while (nodeIndex < maxNodeIndex)
    {
        //Extract next set
        if (*PVS == 0)
        {
            PVS++;
        }
    }
}

```

```

        nodeIndex += ((*PVS) + 1) < 3;
        PVS++;
        if (nodeIndex >= maxNodeIndex) break;
    }

    cPVS = *PVS;
    for (cbit=0; cbit<8; cbit++)
    {
        //Add that link
        if (cPVS & (1<<cbit) )
            nodeList[nodeIndex + cbit].markLeaf(frame, nodeList);
    }

    nodeIndex += 8; //Move on to next 8
    PVS++;
}
}

```

Listing 47.

## 17.13 PVS Generation

PVS generation is used to work out what leaves are visible from each particular leaf. Note that not all the relevant functions are shown here as that would make the document too long however the code is available in the source listing.

### 17.13.1 Anti-Penumbra Clip Plane Generation

The Listing 48 generates anti-penumbra around the given portals.

```

int getAntiPenumbraClippers(PolygonV& sourceP, PolygonV& otherP, bool flipPlane, Plane* outClipPlanes)
{
    //Find each clipping plane that makes up the anti-penumbra for the two given portals

    //Note that static is used purely for perform reasons (keeps them off the stack)
    static int n, m, sizeOther, sizeSource;
    static Vector tri[3];
    static Plane *testPlane;
    static Side sourceS, otherS;

    testPlane = outClipPlanes;
    sizeOther = otherP.Vertex.size();
    sizeSource = sourceP.Vertex.size();

    //Loop through every combination of possible planes, until one is found that splits
    //one portal on the front and the other on the back
    for (n=sizeOther; --n>=0;)
    {
        //The first two vertices of the plane come from the other portal
        tri[0] = otherP.Vertex[n];
        tri[1] = otherP.Vertex[(n+1) % sizeOther];

        for (m=sizeSource; --m>=0;)

```

```

{
    //The other comes from the source portal
    tri[2] = sourceP.Vertex[m];

    //Make sure the plane is valid
    if (getPlaneQuick(tri, testPlane->Normal, testPlane->Dist))
    {
        //Make sure that the other and the source portal are on two different sides
        sourceS = sourceP.getSideOfPlane(*testPlane);
        otherS = otherP.getSideOfPlane(*testPlane);
        if ((sourceS | otherS) == (FRONT | BACK))
        {
            if (flipPlane != (otherS == BACK))
                testPlane->inverseMe();

            testPlane++; //Go to next plane
            break; //The correct plane for this set was found
        }
    }
}
}

return (int) (testPlane - outClipPlanes);
}

```

Listing 48.

Listing 49 produces a more precise anti-penumbra by using the inverse anti-penumbra as well. Duplicate anti-penumbra planes generated by the second anti-penumbra are removed.

```

int getAntiPenumbraClippers(PolygonV& sourceP, PolygonV& otherP, Plane* outClipPlanes)
{
    int nClippers = getAntiPenumbraClippers(sourceP, otherP, false, outClipPlanes);
    int nClippers2 = getAntiPenumbraClippers(otherP, sourceP, true, &outClipPlanes[nClippers]);

    //Check for onplane
    int n, m;
    for (n=nClippers; --n>=0; )
    {
        for (m=nClippers2+nClippers; --m>=nClippers; )
        {
            if (outClipPlanes[n] == outClipPlanes[m])
                outClipPlanes[m].setAlign(1000); //Mark it
            else
                outClipPlanes[m].setAlign(0);
        }
    }

    int offset = 0, total = nClippers+nClippers2;
    for (n=nClippers; n < total; ++n)
    {
        outClipPlanes[n-offset] = outClipPlanes[n];
        if (outClipPlanes[n].getAlign() == 1000)
        {
            ++offset;
            --nClippers2;
        }
    }

    return nClippers+nClippers2;
}

```

```
}
```

[Listing 49.](#)

## 17.13.2 Common

A list of common functions that are used in many of the PVS methods.

```
void BSPStandard::setLeafBit(BYTE* PVSArray, const Node* leaf)
{
    setBit(PVSArray, (int)(leaf - NodeListFixed) - NonLeafNodeCount);
}
```

[Listing 50.](#)

```
void BSPStandard::unsetLeafBit(BYTE* PVSArray, const Node* leaf)
{
    unsetBit(PVSArray, (int)(leaf - NodeListFixed) - NonLeafNodeCount);
}
```

[Listing 51.](#)

```
void BSPStandard::setOWPBit(BYTE* array, const PortalOneWay* portal)
{
    setBit(array, (int)(portal - OWPList));
}
```

[Listing 52.](#)

```
void BSPStandard::unsetOWPBit(BYTE* array, const PortalOneWay* portal)
{
    unsetBit(array, (int)(portal - OWPList));
}
```

[Listing 53.](#)

```
inline bool isBitSet(BYTE* array, int pos)
{
    return (array[pos>>3] & (1<<(pos&7)))?true:false;
}
```

[Listing 54.](#)

```
bool BSPStandard::isLeafSet(BYTE* array, const Node* leaf)
{
    return isBitSet(array, (int)(leaf - NodeListFixed) - NonLeafNodeCount);
}
```

[Listing 55.](#)

```
bool BSPStandard::isOWPSet(BYTE* array, const PortalOneWay* portal)
{
    return isBitSet(array, (int)(portal - OWPList));
}
```

```
}
```

Listing 56.

## 17.13.3 Level 1

This is the first stage of PVS generation. The algorithm rejects portals that are easily removable from the list of potentially visible portals (infront list). The algorithm also works out priority for portals to be used in the next stage.

```
void BSPStandard::createBasicVis()
{
    int n, p;
    MaxPortalVertices = 0; //Calculates the maximum vertices in a portal

    OWPListOrdered = new PortalOneWay*[PortalCount];

    /* All visible one-way-portals must be behind the next one
    /* /-> /-> (correct = Front, Back)
    /* <- /-> (wrong = Back, Back)
    /* /-> <- / (wrong = Front, Front)
    // Calculate the nodes that are in front of one another for each node (deep 1)
    //Find all the infront bits for each portal
    //O(N^2)
    PortalOneWay *p1, *p2;
    for (n = 0; n < PortalCount; ++n)
    {
        p1 = &OWPList[n];
        OWPListOrdered[n] = p1;

        if (MaxPortalVertices < p1->Polygon.Vertex.size()) MaxPortalVertices = p1->Polygon.Vertex.size();
        for (p = 0; p < PortalCount; p++)
        { //Put everything that is infront or spanning in p1
            p2 = &OWPList[p];

            if ((p2->getSideOfPlane(p1->ThePlane) & (SPAN | FRONT)) &&
                (p1->getSideOfPlane(p2->ThePlane) & (SPAN | BACK)))
            {
                setBit(p1->InFront, p);
                //Put portals that are behind more portals at the end of the list
                p1->Priority++;
            }
        }
    }

    for (n = 0; n < PortalCount; ++n)
    {
        p1 = &OWPList[n];
        if (p1->Priority > 0)
        {
            for (p = 0; p < PortalCount; p++)
            {
                p2 = &OWPList[p];

                //If two portals can see one another
```

```

        if (isBitSet(p1->InFront, p) && isBitSet(p2->InFront, n))
        {
            unSetBit(p1->InFront, p);
            unSetBit(p2->InFront, n);
            p1->Priority--;
            p2->Priority--;
        }
    }
}

int count;
PortalOneWay *iterEnd;

//Find any portals that can't see any other portal
for (n = 0; n < PortalCount; ++n)
{
    p1 = &OWPList[n];
    if (p1->Priority > 0)
    {
        count = 0; //Counts how many visible locals there are

        //Check that all neighbor portals are visible
        iterEnd = &p1->Leaf->FacePortals[p1->Leaf->FacePCount];
        for (p2 = &p1->Leaf->FacePortals[0]; p2 != iterEnd; ++p2)
        {
            if (isOWPSet(p1->InFront, p2))
            {
                p1->Priority++; //Portals with more local portals take longer to do
                count++;
            }
        }

        if (count == 0)
        {
            p1->Priority = 0;
            ZeroMemory(p1->InFront, PortalPVSSize); //If no local portals are visible then none are visible
        }
    }
    else
    {
        ZeroMemory(p1->InFront, PortalPVSSize);
    }
}
}

```

Listing 57. Level 1 PVS generation

## 17.13.4 Level 2

Level 2 clips local portals' visible portals by the anti-penumbra. Visible portal lists (infront) are also refactored at this stage.

```

const static BYTE BitCount[] =
{0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2,
3, 3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3,
3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3,
4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4,

```

```

3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5,
6, 6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4,
4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5,
6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5,
3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3,
4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6,
6, 7, 6, 7, 7, 8};

```

Listing 58.

*//BitCount could be pre-calculated using the function below.*

```

void preCalcBitCount(BYTE *bitCount)
{
    BYTE num, bit;
    for (num=1; num<255; ++num)
    {
        bitCount[num] = 0;
        for (bit=0; bit<8; ++bit)
            if (num & (1<<bit))
                bitCount[num]++;
    }
    bitCount[0] = 0;
    bitCount[255] = 8;
}

```

Listing 59.

```

void reCalcSize(BSPStandard::PortalOneWay &portal, int portalPVSSize)
{
    BYTE *iterB, *iterEndB = &portal.InFront[portalPVSSize];
    portal.Priority = 0;

    for (iterB=portal.InFront; iterB!=iterEndB; ++iterB)
        portal.Priority += BitCount[(*iterB)];
}

```

Listing 60.

```

void BSPStandard::recalcSizeAll()
{
    PortalOneWay *p1;
    int n;
    for (n = 0; n < PortalCount; ++n)
    {
        p1 = OWPListOrdered[n];
        if (p1->Priority > 0)
            reCalcSize(*p1, PortalPVSSize);
    }
}

```

Listing 61.

```

void BSPStandard::markKnownPortals(Plane *testPlanes, int nClipCount, PortalOneWay &p1, int testlength)
{
    int n;
    PolygonV* currentPoly;
    PortalOneWay *p4;

    //Boolean add all the bits together form the stack
    TestList[testlength++] = P1AndP2;
}

```

```

BYTE *testPortals = (BYTE*) stackAnd((int**)TestList, TempCopyListSize, testlength, TempCopyList, PortalPVSSize);

//For each portal potentially visible to p1
int portal=0, bit, index;
int p1Byte, cbyte;
BYTE *iter, *p1Iter, *iterEnd = &testPortals[PortalPVSSize];
for (iter=testPortals, p1Iter=p1.InFront; iter!=iterEnd; ++iter, ++p1Iter)
{
    p1Byte = *p1Iter;
    cbyte = *iter;
    if ((cbyte | p1Byte) != p1Byte && cbyte) //If there's something different
    {
        for (bit=0; bit<8; ++bit)
        {
            if (cbyte & (1<<bit) && !(p1Byte & (1<<bit)))
            {
                index = portal+bit;
                p4 = &OWPList[index];
                if (!p4->Leaf->Mark && p4->Mark != -FrameNumb)
                {
                    if (p4->Mark != FrameNumb) //If not already done
                    {
                        //Check that it is not the back side of any clipper
                        currentPoly = &p4->Polygon;
                        for (n=nClipCount; --n>=0 && currentPoly->getSideOfPlane(testPlanes[n]) != BACK; );

                        if (n == -1) //If all tested true
                        {
                            setBit(p1.InFront, index); //Mark as potentially Visible
                            p4->Mark = FrameNumb;
                        }
                        else
                        {
                            p4->Mark = -FrameNumb;
                        }
                    }
                }
            }
        }
        portal+=8;
    }
}
}

```

Listing 62.

```

void BSPStandard::markKnownPortals(Plane *testPlanes, int nClipCount, PortalOneWay &p1)
{
    int n;
    PolygonV* currentPoly;
    PortalOneWay *p4;

    //For each portal potentially visible to p1
    int portal=0, bit, index;
    int p1Byte, cbyte;
    BYTE *iter, *p1Iter, *iterEnd = &P1AndP2[PortalPVSSize];
    for (iter=P1AndP2, p1Iter=p1.InFront; iter!=iterEnd; ++iter, ++p1Iter)
    {
        p1Byte = *p1Iter;
        cbyte = *iter;
        if ((cbyte | p1Byte) != p1Byte && cbyte) //If there's something different

```



```

    {
        for (bit=0; bit<8; ++bit)
        {
            if (cbyte & (1<<bit) && !(p1Byte & (1<<bit)))
            {
                index = portal+bit;
                p4 = &OWPList[index];

                //Check that it is not the back side of any clipper
                currentPoly = &p4->Polygon;
                for (n=nClipCount; --n>=0 && currentPoly->getSideOfPlane(testPlanes[n]) != BACK; );

                if (n == -1) //If all tested true
                    setBit(p1.InFront, index); //Mark as potentially Visible
            }
        }
        portal+=8;
    }
}

```

Listing 63.

```

const int Max_PreProcess_Depth = 6;
void BSPStandard::markDecendentPortals_r(Plane *testPlanes, int nClipCount, PortalOneWay &p1, PortalOneWay &p2, PortalOneWay &p3, int testlength)
{
    if (p3.Priority > 0)
    {
        if (p3.Mark == 1 || testlength > Max_PreProcess_Depth) //If completed take a short cut
        {
            TestList[testlength++] = p3.InFront;
            return markKnownPortals(testPlanes, nClipCount, p1, testlength);
        }
        else
        {
            int n;
            PolygonV* currentPoly;
            PortalOneWay *p4;

            p3.Leaf->Mark = true;

            TestList[testlength++] = p3.InFront;

            PortalOneWay *endPortal = &p3.Leaf->FacePortals[p3.Leaf->FacePCount];
            for (p4 = &p3.Leaf->FacePortals[0]; p4!=endPortal; ++p4)
            {
                //Check that it hasn't been rejected yet and that it's in front of all previous portals
                if (!p4->Leaf->Mark && p4->Mark != -FrameNumb)
                {
                    if (p4->Mark == FrameNumb)
                    {
                        if (isInFrontQuick(testlength-1, TestList, (int)(p4 - OWPList)) && p4->Priority > 0)
                        {
                            TestList[testlength] = p4->InFront;
                            markKnownPortals(testPlanes, nClipCount, p1, testlength+1);
                            //markDecendentPortals_r(testPlanes, nClipCount, p1, p2, *p4, testlength); //Recurse
                        }
                    }
                    else if (isOWPSet(P1AndP2, p4) && isInFrontQuick(testlength, TestList, (int)(p4 - OWPList)))
                    {
                        //Check that it is not the back side of any clipper

```

```

        currentPoly = &p4->Polygon;
        for (n=nClipCount; --n>=0 && currentPoly->getSideOfPlane(testPlanes[n]) != BACK; );

        if (n == -1) //If all tested true
        {
            setOWPBit(p1.InFront, p4); //Mark as potentially Visible
            p4->Mark = FrameNumb;

            markDecendentPortals_r(testPlanes, nClipCount, p1, p2, *p4, testlength); //Recurse
        }
        else
        {
            p4->Mark = -FrameNumb;
        }
    }
}

p3.Leaf->Mark = false;
}
}
}
}

```

Listing 64

```

void BSPStandard::markDecendentPortals(Plane *testPlanes, int nClipCount, PortalOneWay &p1, PortalOneWay &p2)
{
    if (p2.Mark == 1) //If completed take a short cut
    {
        markKnownPortals(testPlanes, nClipCount, p1);
    }
    else
    {
        //Extract data
        int n;
        PolygonV* currentPoly;

        //For each linking portal
        PortalOneWay *p3 = &p2.Leaf->FacePortals[0], *iterEnd = &p2.Leaf->FacePortals[p2.Leaf->FacePCount];
        for (;p3!=iterEnd; ++p3)
        {
            //Check that p3 is in p1's local set (is potentially visible)
            if (isOWPSet(P1AndP2, p3))
            {
                currentPoly = &p3->Polygon;
                for (n=nClipCount; --n>=0 && currentPoly->getSideOfPlane(testPlanes[n]) != BACK; );

                if (n == -1)
                {
                    setOWPBit(p1.InFront, p3);
                    p3->Mark = FrameNumb;

                    markDecendentPortals_r(testPlanes, nClipCount, p1, p2, *p3, 1);
                }
                else
                {
                    p3->Mark = -FrameNumb;
                }
            }
        }
    }
}

```

```
}
```

Listing 65.

```
inline void andArray(BYTE* dest, BYTE* source1, BYTE* source2, int byteSize, int int64Size)
{
    __int64 *dest64 = (__int64*)dest,
              *source64 = (__int64*)source2;
    int i;
    memcpy((void*)dest, (void*)source1, byteSize);
    for (i=0; i<int64Size; ++i)
        dest64[i] &= source64[i];
}
```

Listing 66.

```
void BSPStandard::visBasicAntiPen()
{
    //Sort by value
    qsort( (void*) OWPListOrdered, PortalCount, sizeof(BSPStandard::PortalOneWay*), cmpPortal);

    reFactorPortalLinks();
    recalcSizeAll();

    PolygonV *sourceP, *otherP;

    Plane* clipperCashe = new Plane[MaxPortalVertices*2];
    PortalOneWay *p1 = NULL, *p2 = NULL;

    int n, nClippers;

    //Create the second set
    GlobalInAntiPenList = new BYTE[PortalPVSSize*PortalCount];
    ZeroMemory(GlobalInAntiPenList, PortalPVSSize*PortalCount); //Mark all bits zero (unknown yet)

    TempCopyListSize = (PortalPVSSize + 3)/4;
    TempCopyList = new int[TempCopyListSize];

    TestList = new BYTE[PortalCount];
    TestList[0] = new BYTE[PortalPVSSize]; //Used for quick termination
    ZeroMemory(TestList[0], PortalPVSSize);

    BYTE * gAlter = GlobalInAntiPenList;
    FrameNumb = 2;
    BYTE* oldList;

    MaxPortalDups = 0;

    //2) Check remaining portals in the sets
    for (n = 0; n < PortalCount; n++)
    {
        p1 = OWPListOrdered[n];

        oldList = p1->InFront;
        p1->InFront = gAlter;
        gAlter += PortalPVSSize;

        if (p1->Priority > 0)
        {
            sourceP = &p1->Polygon;
        }
    }
}
```

```

//For each sub-portal
PortalOneWay *p2 = &p1->Leaf->FacePortals[0], *iterEnd = &p1->Leaf->FacePortals[p1->Leaf->FacePCount];
for (;p2!=iterEnd; ++p2)
{
    if (isOWPSet(oldList, p2)) //Check that p2 is within p1's local set
    {
        setOWPBit(p1->InFront, p2);
        if (p2->Priority > 0)
        {
            //And P1 and P2 together
            andArray(P1AndP2, oldList, p2->InFront, PortalPVSSize, P1AndP2ListSize);

            otherP = &p2->Polygon;
            nClippers = getAntiPenumbraClippers(*sourceP, *otherP, clipperCashe);
            markDecendentPortals(clipperCashe, nClippers, *p1, *p2);
            FrameNumb++;
        }
    }
}

//Re-workout Priority
reCalcSize(*p1, PortalPVSSize);

//Work out maximum so the amount of temporary portals for getPVS can be determined
if (p1->Priority > MaxPortalDups) MaxPortalDups = p1->Priority;
}
//Mark as Completed
p1->Mark = 1;
}

delete [] GlobalInFrontList; //Nolonger needed
delete [] clipperCashe;
delete [] TempCopyList;
delete TestList[0];
delete [] TestList;

for (n=(int)((float)PortalCount) * RefactorPortals); --n>0; )
    reFactorPortalLinks(); //Refactor 10 times for more accuracy

recalcSizeAll();

delete [] OWPListOrdered;
}

```

Listing 67.

## 17.13.5Level 3

Level 3 does deep anti-penumbra rejection and compresses the information.

```

void BSPStandard::buildLeafPVS(int testlength, PortalOneWay &sourcePortal, PortalOneWay &otherPortal, PolygonV& clippedPoly, BYTE* PVSAarray, BYTE* leafCheck, Plane* clipperCashe)
{
    //Set the leaf
    setLeafBit(PVSAarray, otherPortal.Leaf);

    Node* otherLeaf = otherPortal.Leaf;

    int result;
}

```

```

int nClippers = getAntiPenumbraClippers(sourcePortal.Polygon, clippedPoly, clipperCashe);

if (!leafCheck) //If a close leaf has not been found yet
    leafCheck = otherPortal.Leaf->PVSLISTTemp;

PolygonV *clipPoly = &ClipPolyCashe[testlength];

//Mark to prevent re-visit
otherPortal.Leaf->Mark = true;

PortalOneWay *destPortal = &otherLeaf->FacePortals[0], *endPortal = &otherLeaf->FacePortals[otherLeaf->FacePCount];
if (!leafCheck)
{
    for (; destPortal!=endPortal; ++destPortal)
    {
        if (!destPortal->Leaf->Mark &&
            destPortal->Mark != (int) sourcePortal.InFront &&
            (!destPortal->Leaf->PVSLISTTemp || isLeafSet(destPortal->Leaf->PVSLISTTemp, sourcePortal.Leaf)) &&
            isOWPSet(P1AndP2, destPortal) && isOWPSet(otherPortal.InFront, destPortal)) //Check that it is in front of the other portals
        {
            if (destPortal->Priority > 0)
            {
                *clipPoly = destPortal->Polygon; //Make copy
                result = checkPlanesSpan(clipperCashe, nClippers, *clipPoly);
                if (result)
                {
                    if (result == 1)
                        destPortal->Mark = (int) sourcePortal.InFront; //Infront is used as an ID

                    buildLeafPVS(testlength+1, sourcePortal, *destPortal, *clipPoly, PVSAArray, leafCheck, &clipperCashe[nClippers]);
                }
            }
            else if (!isLeafSet(PVSAArray, destPortal->Leaf) && checkPlanesNoClip(clipperCashe, nClippers, destPortal->Polygon))
            {
                if (checkPlanesNoClip(clipperCashe, nClippers, destPortal->Polygon))
                    setLeafBit(PVSAArray, destPortal->Leaf);
            }
        }
    }
}
else
{
    for (; destPortal!=endPortal; ++destPortal)
    {
        if (!destPortal->Leaf->Mark &&
            destPortal->Mark != (int) sourcePortal.InFront &&
            (!destPortal->Leaf->PVSLISTTemp || isLeafSet(destPortal->Leaf->PVSLISTTemp, sourcePortal.Leaf)) &&
            isLeafSet(leafCheck, destPortal->Leaf) && //Check that it's leaf is visible from the furthest processed leaf
            isOWPSet(P1AndP2, destPortal) && isOWPSet(otherPortal.InFront, destPortal) ) //Check that it is in front of the other portals
        {
            if (destPortal->Priority > 0)
            {
                *clipPoly = destPortal->Polygon; //Make copy
                result = checkPlanesSpan(clipperCashe, nClippers, *clipPoly);
                if (result)
                {
                    if (result == 1)
                        destPortal->Mark = (int) sourcePortal.InFront; //Infront is used as an ID

                    buildLeafPVS(testlength+1, sourcePortal, *destPortal, *clipPoly, PVSAArray, leafCheck, &clipperCashe[nClippers]);
                }
            }
        }
    }
}
}

```

```

        }
        else if (!isLeafSet(PVSAArray, destPortal->Leaf) && checkPlanesNoClip(clipperCashe, nClippers, destPortal->Polygon))
        {
            setLeafBit(PVSAArray, destPortal->Leaf);
        }
    }
}

otherPortal.Leaf->Mark = false;
}

```

Listing 68.

```

int ZRLEcompress(BYTE* array, vector<BYTE>& array2, int size)
{
    int i, offset = array2.size(),
        bitsize = ((size-1)>3)+1;

    for (i=0; i<bitsize; i++)
    {
        array2.push_back(array[i]);
        if (array[i] == 0) //RLE condition
        {
            int starti = i;
            while (i < size && i - starti < 255 && array[++i] == 0 );

            array2.push_back(i - starti - 1); //set run length
            i--; //Go back one
        }
    }
    return offset;
}

```

Listing 69.

```

/*
 * Counts the amount of visible leaves
 */
int ZRLEcount(BYTE* PVSAArray, int size)
{
    int count=0;
    BYTE* PVS = PVSAArray;
    int nodeIndex=0;
    BYTE cPVS, cbit;
    while (nodeIndex < size)
    {
        //Extract next set
        if (*PVS == 0)
        {
            PVS++;
            nodeIndex += ((*PVS) + 1)<<3;
        }
        else
        {
            cPVS = *PVS;
            for (cbit=0; cbit<8; ++cbit)
            {
                //Add that link
                if (cPVS & (1<<cbit) )

```

```

        count++;
    }
    nodeIndex += 8; //Move on to next 8
}
PVS++;
}
return count;
}

```

Listing 70.

```

inline bool checkPlanes(Plane *testPlanes, int count, PolygonV& polygon)
{
    static int n;
    static Side result;
    for (n=count; --n>=0;)
    {
        result = polygon.checkNTrim(testPlanes[n]);
        if (result == BACK || (result == SPAN && polygon.isSmall()))
            break;
    }
    return (n==1)?true:false; //If it reached the end
}

```

Listing 71.

```

/*
 * Returns -1 if spanning
 */
inline int checkPlanesSpan(Plane *testPlanes, int count, PolygonV& polygon)
{
    static int n;
    static Side result;
    static int spanning;
    spanning = 1;
    for (n=count; --n>=0;)
    {
        result = polygon.checkNTrim(testPlanes[n]);
        if (result == BACK || (result == SPAN && polygon.isSmall()))
            break;
        if (result == SPAN)
            spanning = -1;
    }
    return (n==1)?spanning:false; //If it reached the end
}

```

Listing 72.

```

inline bool checkPlanesNoClip(Plane *testPlanes, int count, PolygonV& polygon)
{
    static int n;
    static Side result;
    for (n=count; --n>=0;)
    {
        result = polygon.getSideOfPlane(testPlanes[n]);
        if (result == BACK || (result == SPAN && polygon.isSmall()))
            break;
    }
    return (n==1)?true:false; //If it reached the end
}

```

```
}
```

Listing 73.

```
int cmpNode( const void *arg1, const void *arg2 )
{
    return *((BSPStandard::Node**)arg1)->SortValue - *((BSPStandard::Node**)arg2)->SortValue;
}
```

Listing 74.

```
void BSPStandard::reFactorLeaves(Node** leafOrdered)
{
    int PVSSize = (FrontLeafCount+7)>>3;
    int tempCopyListSize = (PVSSize + 7)>>3;
    __int64* tempCopyList = new __int64[tempCopyListSize]; //Take out of loop

    __int64* visList;
    Node *n1;
    PortalOneWay *p1, *iterEnd;

    int n, i;

    for (n = 0; n < FrontLeafCount; ++n) //Note that this loop is a candidate for parallel processing
    {
        n1 = leafOrdered[n];
        if (n1->FacePCount > 0)
        {
            iterEnd = &n1->FacePortals[n1->FacePCount];
            p1 = &n1->FacePortals[0];

            //Find the first connected leaf
            if (p1->Priority > 0)
                memcpy((void*)tempCopyList, (void*)p1->Leaf->PVSLISTTemp, PVSSize);
            else
                ZeroMemory((void*)tempCopyList, PVSSize);

            setLeafBit((BYTE*)tempCopyList, p1->Leaf);

            //Or connected leaves together
            for (++p1; p1!=iterEnd; ++p1)
            {
                if (p1->Priority > 0)
                {
                    visList = (__int64*)p1->Leaf->PVSLISTTemp;
                    for (i=0; i<tempCopyListSize; ++i)
                        tempCopyList[i] |= visList[i];
                }

                //Or leaf itself
                setLeafBit((BYTE*)tempCopyList, p1->Leaf);
            }

            //And with main portal
            for (i=0; i<PVSSize; ++i) //Note that it's necessary that PortalPVSSize is used here to prevent overflow
                n1->PVSLISTTemp[i] &= ((BYTE*)tempCopyList)[i];
        }
    }

    delete [] tempCopyList;
}
```



```

}

void BSPStandard::getPVS()
{
    int n;

    PortalOneWay *sourcePortal,
        *endSourcePortal,
        *otherPortal,
        *endOtherPortal;

    //List of leafs sorted in a more efficient order for processing
    Node** leafOrdered = new Node*[FrontLeafCount];

    Node *currentLeaf, *endLeaf = &NodeListFixed[NonLeafNodeCount + FrontLeafCount];
    for (currentLeaf=&NodeListFixed[NonLeafNodeCount], n=0; currentLeaf!=endLeaf; ++currentLeaf, ++n)
    {
        leafOrdered[n] = currentLeaf;
        currentLeaf->SortValue = 0;

        endSourcePortal = &currentLeaf->FacePortals[currentLeaf->FacePCount];
        for (sourcePortal=&currentLeaf->FacePortals[0]; sourcePortal!=endSourcePortal; ++sourcePortal)
            currentLeaf->SortValue += sourcePortal->Priority;
    }

    //Sort by nodes with the most to see (i.e. nodes with lowest visibility first)
    qsort((void*) leafOrdered, FrontLeafCount, sizeof(BSPStandard::Node*), cmpNode);

    int size = FrontLeafCount;

    //Create some space for the uncompressed leaf PVS, note that this space is recycled for each leaf.
    int PVSSize = (FrontLeafCount+7)>>3;

    //Allocate enough memory for each leaf
    BYTE *PVSGlobal = new BYTE[PVSSize*FrontLeafCount],
        *PVSGlter = PVSGlobal;
    ZeroMemory(PVSGlobal, PVSSize*FrontLeafCount); //Mark all bits zero (empty)

    Plane* clipperCache = new Plane[MaxPortalDups*MaxPortalVertices*2]; //Note that this could memory overflow, but probably won't
    ClipPolyCache = new PolygonV[MaxPortalDups];

    for (n = 0; n < FrontLeafCount; ++n, PVSGlter+=PVSSize)
    {
        currentLeaf = leafOrdered[n]; //Get the next most-likely-to-be-easy leaf
        currentLeaf->PVSTemp = PVSGlter;
        endSourcePortal = &currentLeaf->FacePortals[currentLeaf->FacePCount];
        for (sourcePortal=&currentLeaf->FacePortals[0]; sourcePortal!=endSourcePortal; ++sourcePortal)
        {
            Node* otherLeaf = sourcePortal->Leaf;
            setLeafBit(currentLeaf->PVSTemp, otherLeaf);

            if (sourcePortal->Priority > 0)
            {
                endOtherPortal = &otherLeaf->FacePortals[otherLeaf->FacePCount];
                for (otherPortal=&otherLeaf->FacePortals[0]; otherPortal!=endOtherPortal; ++otherPortal)
                {
                    if (isOWPSet(sourcePortal->InFront, otherPortal)) //If both are facing the right way
                    {
                        if (otherPortal->Priority > 0)

```

```

        {
            andArray(P1AndP2, otherPortal->InFront, sourcePortal->InFront, PortalPVSSize, P1AndP2ListSize);
            buildLeafPVS(0, *sourcePortal, *otherPortal, otherPortal->Polygon, currentLeaf->PVSLISTTemp, otherLeaf->PVSLISTTemp, clipperCache);
        }
        else
        {
            setLeafBit(currentLeaf->PVSLISTTemp, otherPortal->Leaf);
        }
    }
}
}
}

delete [] clipperCache;
delete [] ClipPolyCache;

//Refactor leaves
for (n=(int)((float)FrontLeafCount) * RefactorPortals; --n>0; )
    reFactorLeaves(leafOrdered);

delete [] leafOrdered;

PVSLIST.reserve(PVSSize);

MaxVisibleCount = 0;
for (currentLeaf=&NodeListFixed[NonLeafNodeCount]; currentLeaf!=endLeaf; ++currentLeaf)
{
    currentLeaf->PVSLISTIndex = ZRLEcompress(currentLeaf->PVSLISTTemp, PVSLIST, FrontLeafCount); //TODO - Better compression with offsets and overlap

    //Find the biggest amount of PVS visible
    int visCount = ZRLEcount(&PVSLIST[currentLeaf->PVSLISTIndex], FrontLeafCount);
    if (visCount > MaxVisibleCount) MaxVisibleCount = visCount;
}

//Free the global arrays
delete [] PVSGlobal;

PVSLIST.reserve(PVSLIST.size()); //Free up memory
}

```

Listing 76.

## 17.14 Solid Node Compression

Solid node compression compilation process steps are initiated though the compact function shown below:

```

void BSPCompact::compact()
{
    CollisionListSize = 0;
    CollisionList = new CollisionInfo[NonLeafNodeCount];

    Node* root = getCompactRoot();

    //Remove solids
    removeSolids_r(root, CollisionListSize-1);

    //Compress BSP tree
    compress(root);
}

```

```
}
```

Listing 77.

Various data structures used in compiling a solid node tree are:

```
class HeadNode
{
public:
    int Leaf; // -1 if not leaf
    std::vector<Node*> List;
    std::vector<HeadNode> HeadList;
};
```

Listing 78.

```
class CollisionInfo : public LinkedList<CollisionInfo>
{
public:
    int ThePlane;
};
```

Listing 79.

## 17.14.1 Finding the Compact Root

```
BSPCompact::Node* BSPCompact::getCompactRoot()
{
    Node *current = NodeListFixed;

    while ( NodeListFixed[current->Back].ThePlane == SOLID_NODE && NodeListFixed[current->Front].ThePlane != LEAF_NODE)
        current = &NodeListFixed[current->Front];

    current->BB = NodeListFixed[0].BB;

    int parentIndex = current->Parent;
    Node *parent;
    ObjectLL *EndObjLL = current->Object->getEnd(0);

    while (parentIndex >= 0)
    {
        parent = &NodeListFixed[parentIndex];
        //Move object info to the node found
        EndObjLL->Link[0] = parent->Object;
        EndObjLL = parent->Object->getEnd(0);
        parentIndex = parent->Parent;

        CollisionList[CollisionListSize].Link = &CollisionList[CollisionListSize-1];
        CollisionList[CollisionListSize].ThePlane = parent->ThePlane;
        CollisionListSize++;
    }

    CollisionList[0].Link = NULL;
    current->FacePCount = CollisionListSize-1;

    return current;
}
```

```
}
```

Listing 80. Getting the Compact Root

## 17.14.2 Removing Solids

```
void BSPCompact::removeSolids_r(Node* current, int link)
{
    Node *back = &NodeListFixed[current->Back],
        *front = &NodeListFixed[current->Front];

    if (back->ThePlane == SOLID_NODE)
    {
        //Add polygons to parent node
        Node *parent = &NodeListFixed[current->Parent];

        if (front->ThePlane == LEAF_NODE)
        {
            parent->Object->getEnd(0)->Link[0] = current->Object;
        }
        else
        {
            front->Object->getEnd(0)->Link[0] = current->Object;
            front->BB = current->BB;
        }

        //Make back of tree (the node may not be at the back, it is just used for linking)
        current->Back = link;
        CollisionList[CollisionListSize].Link = &CollisionList[link];
        CollisionList[CollisionListSize].ThePlane = current->ThePlane;
        link = CollisionListSize;
        CollisionListSize++;

        //Unlink solid node from parent
        if (parent->Back == (int)(current - NodeListFixed))
            parent->Back = current->Front;
        else
            parent->Front = current->Front;

        front->Parent = current->Parent;
    }
    else
    {
        removeSolids_r(back, link); //Traverse
    }

    if (front->ThePlane == LEAF_NODE)
        //Copy tree into leaf node
        front->FacePCount = link; //reuse FacePCount for node collision information
    else
        removeSolids_r(front, link);
}
```

## 17.14.3 Compress BSP tree

```

void BSPCompact::compress(Node* root)
{
    deque<Node*> nodeQueue;
    deque<HeadNode*> headQueue;

    Root.Leaf = (int)root;
    headQueue.push_back(&Root);

    Node* current;

    int n;
    for (n=0; n<NonLeafNodeCount; ++n)
        NodeListFixed[n].Mark = 0; //Clear all marks (may not be necessary)

    while (headQueue.size() > 0)
    {
        HeadNode* head = headQueue.front();
        headQueue.pop_front();
        nodeQueue.push_back((Node*)head->Leaf);
        head->Leaf = HEAD_NODE; //Indicate that this is not a leaf
        while (nodeQueue.size() > 0)
        {
            current = nodeQueue.front();
            nodeQueue.pop_front();
            int ThePlane = current->ThePlane;
            ASSERT(ThePlane != SOLID_NODE); //Solid linking nodes are stored at the bottom of the tree

            if (ThePlane == LEAF_NODE)
            {
                nodeQueue.clear();

                int size = head->List.size();
                head->HeadList.reserve(size*2); //Make sure memory does not change
                Node **currentNode, **endNode = &head->List[size];
                for (currentNode=&head->List[0]; currentNode!=endNode; ++currentNode)
                {
                    Node *back = &NodeListFixed[(currentNode->Back),
                        *front = &NodeListFixed[(currentNode->Front);

                    if (back->Mark != -1)
                    {
                        int size = head->HeadList.size();
                        head->HeadList.resize(size+1);
                        HeadNode *currentHead = &head->HeadList[size];
                        int ThePlane = back->ThePlane;
                        if (ThePlane == LEAF_NODE)
                        {
                            currentHead->Leaf = (*currentNode)->Back - NonLeafNodeCount;
                        }
                        else
                        {
                            currentHead->Leaf = (int)(back);
                            //Push on to the head nodeQueue for later processing (could be done later)
                            headQueue.push_back(currentHead);
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    }
    if (front->Mark != -1)
    {
        int size = head->HeadList.size();
        head->HeadList.resize(size+1);
        HeadNode *currentHead = &head->HeadList[size];
        int ThePlane = front->ThePlane;
        if (ThePlane == LEAF_NODE)
        {
            currentHead->Leaf = (*currentNode)->Front - NonLeafNodeCount;
        }
        else
        {
            currentHead->Leaf = (int)(front);
            headQueue.push_back(currentHead);
        }
    }
}
}
else
{
    //Append to head list
    head->List.push_back(current);
    current->Mark = -1; //Mark so it is not reused
    nodeQueue.push_back(&NodeListFixed[current->Back]);
    nodeQueue.push_back(&NodeListFixed[current->Front]);
    ASSERT(NodeListFixed[current->Back].ThePlane != SOLID_NODE);
}
}
}
}
}

```

Listing 82. Compress

## 18 Appendix 5 – Engine Sample Code (D)

This section contains important D code segments used in the engine of the BSP tree. The symbols “...” are used to remove irrelevant portions of code. However the complete code can be found in “”. Note that the code presented in this section may have been simplified at the cost of efficiency to improve readability.

### 18.1 Class Properties

In order to understand the following algorithms a brief description of the class properties are given below.

```
struct vec3
{
public:
    float x, y, z;
    ...
}
```

Listing 83.

```
struct AABBLite
{
    vec3 Min, Max;
}
```

Listing 84.

```
struct Plane
{
public:
    vec3 normal;
    float dist;
    ...
}
```

Listing 85.

```
struct Frustum
{
    Plane Frust[6];
    ...
}
```

Listing 86.

```
struct Node
{
    int PlaneIndex; //Use indexes?
```

```

int Mark;
Node* Parent;
AABBLite BB;
PolygonKeeper* [] PolyList;
Node* Front;
Node* Back;
...
}

```

Listing 87.

## 18.2 View Frustum Culling

In view frustum culling each BB must be tested against each plane to see whether it is inside or outside. Below is the test used against a single plane (method of Plane).

```

/*
 * Used for testing a plane against the AABB, returns the number of points over
 * the frustum.
 */
int isIn(AABBLite BB, byte bitpos, inout byte alreadyInside) //Method of Plane
{
    vec3 fmin, fmax;
    float fnorm;

    fmin = normal * BB.Min;
    fmax = normal * BB.Max;
    fnorm = dist;

    float fminXY = fmin.x + fmin.y,
          fmaxXY = fmax.x + fmax.y,
          fmaxminXY = fmax.x + fmin.y,
          fminmaxXY = fmin.x + fmax.y,
          fminZN = fmin.z + fnorm,
          fmaxZN = fmax.z + fnorm;

    //Note, assuming the C rule that true = 1
    int c = ( fmaxminXY + fminZN > 0 ) + ( fminXY + fminZN > 0 ) +
            ( fminmaxXY + fminZN > 0 ) + ( fmaxXY + fminZN > 0 ) +
            ( fmaxminXY + fmaxZN > 0 ) + ( fminXY + fmaxZN > 0 ) +
            ( fminmaxXY + fmaxZN > 0 ) + ( fmaxXY + fmaxZN > 0 );

    if( c == 8 )
        alreadyInside |= bitpos; //Set the bit

    return c;
}

```

Listing 88.

Bit pos indicate which plane is being tested. AlreadyInside contains a bit list of planes that already have the current BB as inside. The alreadyInside mask is used to avoid retests because once a BB is inside a plane; it is always inside that plane.

The following algorithm tests all 6 planes and returns true if the BB is visible within the view frustum (method of Frustum).

```

/*

```



```

/* Determines if a AABB is within the view frustum and if it's intersecting,
 * and what plane.
 */
bool isln(AABBLite BB, inout byte alreadyInside) //Method of Frustum
{
    //Go to each of the 6 planes and see if the AABB is behind.
    if (Frust[0].isln(BB, 1, alreadyInside) == 0) return false;
    if (Frust[1].isln(BB, 2, alreadyInside) == 0) return false;
    if (Frust[2].isln(BB, 4, alreadyInside) == 0) return false;
    if (Frust[3].isln(BB, 8, alreadyInside) == 0) return false;
    if (Frust[4].isln(BB, 16, alreadyInside) == 0) return false;
    if (Frust[5].isln(BB, 32, alreadyInside) == 0) return false;
    return true;
}

```

Listing 89.

## 18.3 Collision Detection

This collisionCheck is a method of a BSP tree node that returns true if the current node is a closer collision then other nodes tested.

```

/*
 * Determines if a plane has switched sides.
 */
bool collisionCheck(vec3 oldCamaraPos, vec3 newCamaraPos, inout float closestDist, float boundary)
{
    float dist = PlaneList[PlaneIndex].computeSide(newCamaraPos) - boundary;

    if ( dist < 0 && dist > closestDist && PlaneList[PlaneIndex].computeSide(oldCamaraPos) - boundary > 0)
    {
        closestDist = dist;
        return true;
    }
    return false;
}

```

Listing 90.

getleaf is a standalone function that returns the leaf that the camera is in (solid or empty).

```

/*
 * Finds the leaf the camera is in. boundary is used for collision detection to move
 * planes closer to the camera.
 */
Leaf* getLeaf(Node* root, vec3 cameraPos, float boundary)
{
    assert(root); //Must be at least one node"
    OldCameraPos = cameraPos;

    Node* current = root;

    while ( 1 )
    {
        if ( PlaneList[current.PlaneIndex].computeSide(cameraPos) - boundary > 0 ) //Front
        {
            current = current.Front;

            if ( current.PlaneIndex < 0 )

```

```

        break;
    }
    else if ( current.Back == null ) //Back
        break;
    else
        current = current.Back;
}
return cast(Leaf*)(current);
}

```

Listing 91.

getCollisionNode is a standalone function that returns the node that is closest to the camera (if any).

```

/*
 * Finds the closest node plane to the camera
 */
Node* getCollisionNode(Node* root, Node* solid, vec3 oldCameraPos, vec3 newCameraPos, float boundary)
{
    Node* current = solid, result;
    float closestDist = -float.max;

    while ( current != null )
    {
        if ( current.collisionCheck(oldCameraPos, newCameraPos, closestDist, boundary) )
            result = current;

        current = current.Parent;
    }

    return result;
}

```

Listing 92.

collisionCheck is a standalone function that returns the plane (if any) that is closest to the player. It returns -1 if the plane has moved to far. Note, in D storing a key value within a pointer is generally considered a bad idea because of the GC however in this case it is probably not a problem.

```

/*
 * Performs the entire collision check
 */
Plane* collisionCheck(Node* root, vec3 oldCameraPos, vec3 cameraPos, float boundary)
{
    Leaf* endLeaf = getLeaf(root, cameraPos, boundary);

    if ( endLeaf.PVSIndex < 0 )
        //Structural was successful so check for detail collisions
        return endLeaf.collisionCheck(oldCameraPos, cameraPos);

    Node* collisionNode = getCollisionNode(root, (Node*)endLeaf, oldCameraPos, cameraPos, boundary);

    if ( collisionNode )
        return &PlaneList[collisionNode.PlaneIndex];
    else
        return (Plane*) -1; //Tricksy - Most probably moved to far
}

```

## 18.4 Marked BB BSP tree Traversal

BSP tree traversal is used to frustum cull potentially visible nodes in clusters.

drawPVS(Node\*) sets up the recursive BSP tree traversal. The first node is almost certain to be within the view frustum (as it contains the entire tree) so it is drawn without any frustum tests. Note that back leaves (solid nodes) are recognised by null node links and leaf nodes (empty nodes) are recognised by the negative part of the node's plane index.

```
void drawPVS(Node* current)
{
    current.draw();

    if ( current.Back ) //If not solid
        drawPVS(current.Back, 0);

    if ( current.Front.PlaneIndex >= 0 ) //If not leaf
        drawPVS(current.Front, 0);
}
```

Listing 94.

drawPVS(Node\*, byte) recursively draws the BSP tree nodes. Any nodes that are not marked or outside the frustum are not traversed. If the node is completely within the frustum then no more tests are required so drawPVSNofV is called.

```
void drawPVS(Node* current, byte alreadyInside)
{
    if ( current.Mark == MarkedFrame &&
        BSPPVSTree.view.isIn(current.BB, alreadyInside) ) //If within the frustum
    {
        current.draw();

        if ( alreadyInside == 63 ) //No more frustum testing needed
        {
            if ( current.Back ) //If not solid
                drawPVSNofV(current.Back);

            if ( current.Front.PlaneIndex >= 0 ) //If not leaf
                drawPVSNofV(current.Front);
        } else
        {
            if ( current.Back ) //If not solid
                drawPVS(current.Back, alreadyInside);

            if ( current.Front.PlaneIndex >= 0 ) //If not leaf
                drawPVS(current.Front, alreadyInside);
        }
    }
}
```

Listing 95.

drawPVSNoVF traverses the nodes in the BSP tree but only culls nodes that are not marked.

```
void drawPVSNoVF(Node* current)
{
    if ( current.Mark == MarkedFrame )
    {
        current.draw();

        if ( current.Back ) //If not solid
            drawPVSNoVF(current.Back);

        if ( current.Front.PlaneIndex >= 0 ) //If not leaf
            drawPVSNoVF(current.Front);
    }
}
```

Listing 96.

## 18.5 Render Linked BSP tree

This version of DrawPVS draws the BSP tree using view space linking to short-cut the amount of nodes that require processing (method of BSPtreelink which extends bsptree)

```
bool drawPVS()
{
    if ( CurrentLeaf ) //If not empty
    {
        //Extract the link node
        topLink = (cast(Node*)CurrentLeaf.Info.Link);

        //Draw the parents of the link node
        Node* linkParent = topLink.Parent;
        if (linkParent)
            linkParent.drawParents();

        //Mark each bit in the PVS up the parent
        uncompressPVS();
        mark(*topLink); //NodeList[0];

        //Traverse the trees PVS nodes
        super.drawPVS(topLink);
    }

    return true;
}
```

Listing 97.

Draw parents is used to draw the parents of the given node (method of BSPtree node)

```
void drawParents()
{
    Node* me = this;
    do
```

```

{
    me.draw();
    me = me.Parent;
} while ( me );
}

```

Listing 98.

## 18.6 Mark Small Containers

Mark\_small\_containers puts a flag in BSP nodes that contain less then MIN\_OCC\_CONTAINER. Mark\_small\_containers is a member of node.

```

int markSmallContainers()
{
    int count = 0;
    if ( Back )
        count += Back.markSmallContainers() + 1;

    if ( Front.PlaneIndex >= 0 ) //If not leaf
        count += Front.markSmallContainers() + 1;

    if (count < MIN_OCC_CONTAINER)
        Flag |= FLAG_STOP_CULLING;

    return count;
}

```

Listing 99.

## 18.7 Solid Node Compression

### 18.7.1 Data Structures

Some of the common data structures used in solid node compression:

```

struct Node //Note that variable order counts
{
    int PlaneIndex; //Use indexes?
    AABB Lite BB;
    NodeInfo Info; //More data about the node
    uint Flag; //Used for things like transparency/water ect...
    int Mark;
}

```

Listing 100.

```

struct Leaf //Note that variable order counts
{
    public:
    int PVSIndex;
    HNodeParentIter HParent;
    AABB Lite BB;
    LeafInfo Info; //More data about the leaf
    CollisionInfo* CollisionList;
}

```

```
}
```

Listing 101.

```
struct HNodePair
{
    Node* node; //null if leaf
    HeadNodeInfo* hnode;
}
```

Listing 102.

```
struct HNodeParentIter
{
    Node* node; //null if leaf
    HeadNode* phnode;
}
```

Listing 103.

```
struct HeadNodeInfo
{
    NodeArray NodeList;
    HeadNode [] HeadList;
    HeadNode* HParent;
}
```

Listing 104.

```
struct HeadNode
{
    int LeafOrHead;
}
```

Listing 105.

## 18.7.2 Getting the parent

The following method is a member of HNodeParentIter and will change the HNodeParentIter into its parent. It is useful to change the node this way because parent traversal is a linear process so the same variable can be reused.

```
void makeParent() //Should only be called if parent exists
{
    HeadNodeInfo* hnode = &GHeadList[phnode.LeafOrHead];
    int offset = node - hnode.NodeList[0];

    if (offset == 0) //If root of this headNode
    {
        HeadNodeInfo* hnodeParent = &GHeadList[hnode.HParent.LeafOrHead];
        offset = phnode - &hnodeParent.HeadList[0] + hnodeParent.NodeList.length;
        phnode = hnode.HParent;
        hnode = hnodeParent;
    }
    node = hnode.NodeList[(offset-1)/2];
}
```

### 18.7.3 Getting Children

The following code is an object orientated way of getting child information from nodes. The frontnback methods have been simplified into smaller methods (rather than provide a single function) to increase reusability of code.

```

struct HNodePair
{
    Node* node; //null if leaf
    HeadNodeInfo* hnode;

    //Returns the leaf index if that was found
    Leaf* decodeType(int refcode)
    {
        if (refcode > 0)
        {
            hnode = &GHeadList[refcode];
            node = hnode.NodeList[0];
            return null;
        }
        else
        {
            node = null;
            return cast(Leaf*) &LeafList[-refcode];
        }
    }

    void decodeTypeQ(int refcode)
    {
        if (refcode > 0)
        {
            hnode = &GHeadList[refcode];
            node = hnode.NodeList[0];
        }
        else
        {
            node = null;
        }
    }
}

```

Listing 107.

```

struct HeadNodeInfo
{
    NodeArray NodeList;
    HeadNode [] HeadList;
    HeadNode* HParent;

    //It is slightly faster to get both front and back at the same time.
    void backnfront(Node* node, out HNodePair back, out HNodePair front)
    {
        //Compute back node offset
        uint nodeIndex = cast(uint)(node - NodeList[0]) * 2 + 1;
    }
}

```

```

        if (NodeList.length > nodeIndex)
        {
            back.node = NodeList[nodeIndex];
            back.hnode = this;

            getNode(nodeIndex+1, front);
        }
        else
        {
            //Get the head node offset
            nodeIndex -= NodeList.length;

            //Work out if this is a leaf or not
            HeadList[nodeIndex].decodeTypeQ(back);

            //Work out if this is a leaf or not
            HeadList[nodeIndex+1].decodeTypeQ(front);
        }
    }
}

void back(Node* node, inout HNodePair back, out Leaf* backL)
{
    //Compute back node offset
    //printf("back = %d\n", (uint)(node - &NodeList[0]) * 2 + 1);
    getNode(cast(uint)(node - NodeList[0]) * 2 + 1, back, backL);
}

void front(Node* node, inout HNodePair front, out Leaf* frontL)
{
    //Compute front node offset
    //printf("back = %d\n", (uint)(node - &NodeList[0]) * 2 + 2);
    getNode(cast(uint)(node - NodeList[0]) * 2 + 2, front, frontL);
}

void getNode(uint index, inout HNodePair pnode, out Leaf* nodeL)
{
    if (NodeList.length > index)
        pnode.node = NodeList[index];
    else //Work out if this is a leaf or not
        nodeL = HeadList[index - NodeList.length].decodeTypeQ(pnode);
}

void getNode(uint index, inout HNodePair pnode)
{
    if (NodeList.length > index)
    {
        pnode.node = NodeList[index];
        pnode.hnode = this;
    }
    else //Work out if this is a leaf or not
        HeadList[index - NodeList.length].decodeTypeQ(pnode);
}
}

```

Listing 108.

struct HeadNode



```

{
    int LeafOrHead;

    Leaf* decodeType(inout HNodePair type)
    {
        return type.decodeType(LeafOrHead);
    }

    //Doesn't return leaf
    void decodeTypeQ(inout HNodePair type)
    {
        type.decodeTypeQ(LeafOrHead);
    }
}

```

Listing 109.

## 18.8 Occlusion Culling

### 18.8.1 The Occlusion Queue

The ready queue is used to process lists of occlusion tests. Note that this version needs support from Glee (see 20) to access the OpenGL™ extensions however the difference (when compared to manually ported extensions) is trivial.

```

/*!
 * The current position of the last shared part of the array
 */
uint QBIter;

public void resetQuerySize()
{
    QBIter = 0;
}

struct ArrayOffset { int length; int start; }

private
{
    /*!
     * Used to enable/disable Occlusion globally. It will not enable if
     */
    bool OcclusionEnabled = false; //GL_ARB_occlusion_query;

    /*!
     * If this is on then occlusion mode is on.
     */
    bool OcclusionMode = false;

    /*!
     * If this is on then BB occlusion mode is on.
     */
    bool OcclusionModeBB = false;

    /*!

```

```

    * Disables things that will slow down occlusion.
    */
    void disableAll()
    {
        glDisable(GL_BLEND);
        glDisable(GL_ALPHA_TEST);
        glDisable(GL_AUTO_NORMAL);
        glDisable(GL_COLOR_MATERIAL);
        glDisable(GL_FOG);
        glDisable(GL_LIGHTING);
        glDisable(GL_NORMALIZE);
        glDisable(GL_TEXTURE_2D);
        glDisable(GL_TEXTURE_1D);
        glEnable(GL_CULL_FACE);
        glEnable(GL_DEPTH_TEST);
    }

    void enableBBocc()
    {
        if (!OcclusionModeBB)
        {
            glDepthMask(GL_FALSE);
            OcclusionModeBB = true;
        }
    }

    void disableBBocc()
    {
        if (OcclusionModeBB)
        {
            OcclusionModeBB = false;
            glDepthMask(GL_TRUE);
        }
    }
}

void enableOcclusion(bool state)
{
    if (GLEE_NV_occlusion_query && strstr((char*) glGetString(GL_EXTENSIONS), (char*) "GL_ARB_occlusion_query")) //If possible
    {
        /* This extention is not implemented properly in Glee, yet
        GLuint bitsSupported;
        glGetQueryiv(GL_QUERY_COUNTER_BITS_ARB, &bitsSupported);
        if (bitsSupported == 0)
        {
            printf("Warning - GL_ARB_occlusion_query does not support enough bits, performance my be reduced!");
            OcclusionEnabled = false;
        }
        else*/
        {
            OcclusionEnabled = state;
        }
    }
    else
    {
        if (state == true)
            printf("Warning - GL_ARB_occlusion_query is not available on this system, performance my be reduced!");
        OcclusionEnabled = false;
    }
}

```

```

bool isOcclusionEnabled()
{
    return OcclusionEnabled;
}

/*!
 * Turns occlusion on and disables anything that takes time to render, ie blending, texturing, lighting
 */
void occlusionOn()
{
    if (OcclusionEnabled && !OcclusionMode)
    {
        //todo - shorten this
        glPushAttrib(GL_COLOR_BUFFER_BIT);
        glPushAttrib(GL_DEPTH_BUFFER_BIT);
        glPushAttrib(GL_ENABLE_BIT);
        glPushAttrib(GL_FOG_BIT);
        glPushAttrib(GL_HINT_BIT);
        glPushAttrib(GL_LIGHTING_BIT);
        glPushAttrib(GL_POLYGON_BIT);
        glPushAttrib(GL_TEXTURE_BIT);

        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
        glDepthMask(GL_TRUE);
        glDepthFunc(GL_LESS);

        disableAll();

        OcclusionMode = true;
    }
}

/*!
 * Turns occlusion off and enables anything that was taken off by occlusionOn
 */
void occlusionOff()
{
    if (OcclusionEnabled && OcclusionMode)
    {
        OcclusionMode = false;
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();
        glPopAttrib();

        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

        glFlush();
    }
}

void enableDrawOcclusion()
{
    if (OcclusionEnabled)
    {
        glDepthMask(GL_FALSE);
    }
}

```

```

        glDepthFunc(GL_EQUAL);
    }
    else
    {
        disableDrawOcclusion();
    }
}

void disableDrawOcclusion()
{
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LESS);
}

/*!
 * Turns BBocclusion on. Note that occlusionOn must also be on
 */
void BBocclusionOn()
{
    if (OcclusionEnabled)
    {
        enableBBoccc();
    }
}

void BBocclusionOff()
{
    if (OcclusionEnabled)
    {
        disableBBoccc();
    }
}

/*!
 * Type is used to hold information for each query
 * Note that each time a Query class is created, the Buffer is re-created
 */
struct OcclusionQuery (Type)
{
    /*!
     * The 1/SeekAhead = the percentage (0.0-1.0) to make sure is within the buffer
     */
    const int SeekAhead = 3;

    /*!
     * Queries hold the current polygons being processed
     */
    private ArrayOffset Queries;

    /*!
     * The position in the circular queue for adding the query
     */
    private uint PosAdd;

    /*!
     * The position in the circular queue for getting the result
     */

```

```

private uint PosGet = 0;

/*!
 * Info holds detail's about the current polygon being processed
 */
private Type[] Info;

/*!
 * The current number being checked
 */
private int Amount;

/*!
 * Initiates stuff for occlusion culling
 * Buffer is the maximum number of queries to store
 */
void capacity(int shareSize)
{
    if (shareSize > 0)
    {
        Info.length = shareSize;
        Queries.start = QBlter+1;
        Queries.length = shareSize;
        QBlter += shareSize;
    }
}

/*!
 *
 */
int capacity()
{
    return Info.length;
}

/*!
 * Adds a the type on to the queue.
 */
bool add(AABBLite BB, Type info)
{
    if (Amount == Queries.length)
    {
        //assert(0);
        return false;
    }
    else
    {
        if (OcclusionEnabled)
        {
            enableBBocc();

            //Render BB
            glBeginQueryARB(GL_SAMPLES_PASSED_ARB, PosAdd+Queries.start);
            BB.draw();
            glEndQueryARB(GL_SAMPLES_PASSED_ARB);
        }

        Info[PosAdd] = info;
    }
}

```

```

        if ( ++PosAdd >= Queries.length )
            PosAdd = 0;

        Amount++;

        return true;
    }
}

/*!
 * Adds a the type on to the queue.
 */
bool add(GLuint BBdisplayList, Type info)
{
    if (Amount == Queries.length)
    {
        //assert(0);
        return false;
    }
    else
    {
        if (OcclusionEnabled)
        {
            enableBBocc();

            //Render BB
            glBeginQueryARB(GL_SAMPLES_PASSED_ARB, PosAdd+Queries.start);
            glCallList(BBdisplayList);
            glEndQueryARB(GL_SAMPLES_PASSED_ARB);
        }

        Info[PosAdd] = info;

        if ( ++PosAdd >= Queries.length )
            PosAdd = 0;

        Amount++;

        return true;
    }
}

/*!
 * Begins a query, must be closed with end.
 */
void begin(Type info)
{
    assert (Amount != Queries.length);
    {
        if (OcclusionEnabled)
        {
            glBeginQueryARB(GL_SAMPLES_PASSED_ARB, PosAdd+Queries.start);
        }

        Info[PosAdd] = info;

        if ( ++PosAdd >= Queries.length )
            PosAdd = 0;

        Amount++;
    }
}

```

```

}

/*!
 * Ends the query. Should only run if begin succeeds
 */
void end()
{
    if (OcclusionEnabled)
    {
        glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    }
}

private void popNext()
{
    if ( ++PosGet >= Queries.length ) //Wrap around
        PosGet = 0;

    Amount--;
}

/*!
 * Returns the next ready query or null
 * //todo - change to for-each
 */
bool next(inout GLuint count, out Type info)
{
    if ( Amount == 0 )
        return false;

    if (OcclusionEnabled)
    {
        GLint available;
        glGetQueryObjectivARB(PosGet+Queries.start, GL_QUERY_RESULT_AVAILABLE_ARB, &available);

        if ( available )
        {
            glGetQueryObjectuiARB(PosGet+Queries.start, GL_QUERY_RESULT_ARB, &count);
            //printf((count > 0)? "hit %d \n": "miss %d\n", count);

            info = Info[PosGet];
            popNext();

            return true;
        }
    }
    else
    {
        info = Info[PosGet];
        popNext();
        count = int.max;
        return true;
    }

    //printf("%d\n", PosGet);
    return false;
}

/*!
 * Outputs the next query ready or not. If it is not ready then it will be marked visible.

```

```

    * If failed then count is -1.
    * Returns false when there are no more queries
    */
    bool forceNext(out GLuint count, out Type info)
    {
        if ( Amount == 0 )
            return false;

        info = Info[PosGet];

        if (OcclusionEnabled)
        {
            GLint available;
            glGetQueryObjectivARB(PosGet+Queries.start, GL_QUERY_RESULT_AVAILABLE_ARB, &available);
            if ( available )
            {
                glGetQueryObjectiARB(PosGet+Queries.start, GL_QUERY_RESULT_ARB, &count);
                //printf((count > 0)?"hit\n":"miss\n");
            }
            else
            {
                count = -1;
            }
        }
        else
        {
            count = int.max;
        }

        popNext();

        return true;
    }

    /*!
    * Returns true if there are more queries left waiting in the que
    */
    bool isEmpty()
    {
        return ( Amount == 0 );
    }

    /*!
    * Checks ahead to make sure that there are queries ready to grab
    */
    bool checkAhead()
    {
        if ( Amount == 0 )
            return false;

        if (OcclusionEnabled)
        {
            uint GetAhead = PosGet + Amount/3;
            if ( GetAhead >= Queries.length ) //Wrap around
                GetAhead -= Queries.length;

            GLint available;
            glGetQueryObjectivARB(GetAhead+Queries.start, GL_QUERY_RESULT_AVAILABLE_ARB, &available);

            return (available)?true:false;
        }
    }

```



```

    }
    else
    {
        return true;
    }
}
}

```

## 18.8.2 The Ready Queue

The ready queue is essentially a normal queue. The code below shows an example with all the methods necessary for source code's occlusion culling.

```

struct circularQueue(Type) //Circular queue
{
    void capacity(uint len)
    {
        Q.length = len;
        //TODO - fix wrap-around problems
    }
    uint capacity() { return Q.length; }

    bool add(Type element) //Push on to back of queue, returns true if successful
    {
        if (Amount == Q.length)
        {
            return false;
        }
        else
        {
            Q[Qend] = element;

            if ( ++Qend >= Q.length )
                Qend = 0;

            Amount++;

            return true;
        }
    }

    bool next(out Type element) //Pull of front off queue (returns true if successful)
    {
        if (isEmpty()) return false;

        element = Q[Qstart];

        if ( ++Qstart >= Q.length ) //Go to next
            Qstart = 0;

        Amount--;

        return true;
    }

    bool isEmpty() { return (Amount == 0); }

    bool isFull() { return (Amount == Q.length); }
}

```

```

uint contains() { return Amount; }

private:

    Type [] Q;

    uint Qstart = 0;
    uint Qend = 0;
    uint Amount = 0;
}

```

## 18.8.3 Occlusion BSP Tree

The entire occlusion tree is given here however the tree that it is derived from will have to be looked up in the source as it is impractical to print that much code.

```

OcclusionQuery!(BrushSingle*) OccBSingle;
OcclusionQuery!(BrushShared*) OccBShared;

BDetail* [] DBrushList;
BDetail* [] DBrushListNoVF;
uint DBrushListSize;
uint DBrushListNoVFSize;

struct BSPTOcclusion
{
    const uint MostlyVisible = 3000; //How many pixels to be considered mostly visible
    const uint FrameCoherancy = 5; //How many frames to use in frame-to-frame coherancy

    struct NodeInfo
    {
        PolygonKeeper* [] PolyList;
        int MarkOcc; //Used to indicate if a Occtest needs to be repeated

        void load(inout ubyte *iter, PolygonKeeper [] polyList, void* parent)
        {
            PolyList.length = readInt(iter);

            foreach (int n, PolygonKeeper* val; PolyList)
                PolyList[n] = &polyList[readInt(iter)];
        }
    }

    struct LeafInfo
    {
        BDetail Detail;

        void load(inout ubyte *iter, PolygonKeeper [] polyList, void* parent)
        {
            Detail.load(iter, polyList);
        }

        void draw(Frustum frust)
        {
            DBrushList[DBrushListSize] = &Detail;
            DBrushListSize++;
        }
    }
}

```

```

    }
    void draw()
    {
        DBrushListNoVF[DBrushListNoVFSize] = &Detail;
        DBrushListNoVFSize++;
    }
}

static void drawTest(BDetail *Detail) //No frustum test
{
    foreach (inout BrushSingle val; Detail.Single)
    {
        if (val.Mark > MarkedFrame - FrameCoherancy)
        {
            OccBSingle.add(val.BB, &val);
        }
        else
        {
            BBocclusionOff();
            val.draw();
        }
    }

    foreach (inout BrushShared val; Detail.Shared)
    {
        if (val.Mark > MarkedFrame - FrameCoherancy)
        {
            OccBShared.add(val.BB, &val);
        }
        else
        {
            BBocclusionOff();
            val.draw();
        }
    }
}

static void drawTest(BDetail *Detail, Frustum frust)
{
    foreach (inout BrushSingle val; Detail.Single)
    {
        if ( frust.isIn(val.BB) )
        {
            if (val.Mark > MarkedFrame - FrameCoherancy)
            {
                OccBSingle.add(val.BB, &val);
            }
            else
            {
                BBocclusionOff();
                val.draw();
            }
        }
    }

    foreach (inout BrushShared val; Detail.Shared)
    {
        if ( frust.isIn(val.BB) )
        {

```

```

        if (val.Mark > MarkedFrame - FrameCoherancy)
        {
            OccBShared.add(val.BB, &val);
        }
        else
        {
            BBocclusionOff();
            val.draw();
        }
    }
}

static void processDetail(Frustum frust)
{
    for (int n=0; n<DBrushListNoVFSize; ++n)
        drawTest(DBrushListNoVF[n]);

    DBrushListNoVFSize=0;

    for (int n=0; n<DBrushListSize; ++n)
        drawTest(DBrushList[n], frust);

    DBrushListSize=0;

    BBocclusionOff();

    BrushSingle* bSingle;
    uint count;
    while (OccBSingle.forceNext(count, bSingle))
    {
        if (count != 0) //TODO - mark visibility
        {
            if (count > MostlyVisible) bSingle.Mark = MarkedFrame;
            bSingle.draw();
        }
    }

    BrushShared* bShared;
    while (OccBShared.forceNext(count, bShared))
    {
        if (count != 0) //TODO - mark visibility
        {
            if (count > MostlyVisible) bShared.Mark = MarkedFrame;
            bShared.draw();
        }
    }
}

alias Standard.NodeLeaf NodeLeaf;

class Tree(Node, Leaf) : public Standard.BSP!(Node, Leaf).Tree
{
public:
    this()
    {
        enableOcclusion(true);
    }
}

```

```

        if (isOcclusionEnabled())
            printf("Occlusion is enabled\n");
    }

    ~this() { }

void load(inout ubyte *iter, inout PolygonKeeper [] polyList)
{
    super.load(iter, polyList);

    int nDetailSingle = 0, nDetailShared = 0;
    foreach (Leaf val; LeafList)
    {
        nDetailSingle += val.Info.Detail.Single.length;
        nDetailShared += val.Info.Detail.Shared.length;
    }

    NodeDisplayLists = glGenLists(NodeList.length);

    GLuint list = NodeDisplayLists;

    //Create BB display lists
    foreach (inout Node val; NodeList)
    {
        glNewList(list++, GL_COMPILE);
        val.BB.draw();
        glEndList();
    }

    Occlusion.capacity(BufferOcclusion);
    OcclusionNoVF.capacity(BufferOcclusion);

    OccBSingle.capacity(nDetailSingle);
    OccBShared.capacity(nDetailShared);

    DBrushList.length = LeafList.length;
    DBrushListNoVF.length = LeafList.length;

    ReadyQ.capacity = NodeList.length;
    ReadyQNoVF.capacity = NodeList.length;
}

private GLuint nodeDisplayList(Node* node)
{
    return NodeDisplayLists + cast(uint)(node - &NodeList[0]);
}

private void draw(Node* node)
{
    BBocclusionOff(); //Make sure occlusion is off
    node.draw();
}

private void addOccNode(Node* current, byte alreadyInside)
{
    OccInfo infoAdd;
    infoAdd.Current = current;
    infoAdd.AlreadyInside = alreadyInside;
    //If it hasn't been visible recently and if its not behind the near or far plane, then
    //use occlusion (if there is enough room)

```

```

        if (current.Info.MarkOcc > MarkedFrame - FrameCoherancy || !(alreadyInside & Frustum.NEAR_BIT) || !Occlusion.add(nodeDisplayList(current), infoAdd))
        {
            ReadyQ.add(infoAdd);
        }
    }

    private void addOccNodeNoVF(Node* current)
    {
        if (current.Info.MarkOcc > MarkedFrame - FrameCoherancy || !OcclusionNoVF.add(nodeDisplayList(current), current))
        {
            ReadyQNoVF.add(current);
        }
    }

    private void addReadyNode(Node* current, byte alreadyInside)
    {
        if (ReadyQ.contains() < BufferNodes)
        {
            OcclInfo infoAdd;
            infoAdd.Current = current;
            infoAdd.AlreadyInside = alreadyInside;
            ReadyQ.add(infoAdd);
        }
        else
        {
            addOccNode(current, alreadyInside);
        }
    }

    private void addReadyNodeNoVF(Node* current)
    {
        if (ReadyQNoVF.contains() < BufferNodes)
        {
            ReadyQNoVF.add(current);
        }
        else
        {
            addOccNodeNoVF(current);
        }
    }

    private void drawFrontNodeNoVF(Node* current)
    {
        if (current.Front.PlaneIndex >= 0) //If not leaf
            drawPVSNoVF(current.Front);
    }

    private void drawFrontNodeOccNoVF(Node* current)
    {
        if (current.Front.PlaneIndex >= 0) //If not leaf
            drawPVSOccNoVF(current.Front);
    }

    private void drawFrontNodeOcc(Node* current, byte alreadyInside)
    {
        if (current.Front.PlaneIndex >= 0) //If not leaf
            drawPVSOcc(current.Front, alreadyInside);
    }

```

```

private void drawFrontPVSOcc(Node* current, byte alreadyInside)
{
    if (current.Flag & FLAG_STOP_CULLING)
    {
        BBocclusionOff(); //Make sure occlusion is off
        drawFrontNodeNoVF(current);
        current.draw();
        if ( current.Back ) drawPVSN VF(current.Back);
    }
    else if ( alreadyInside == 63 ) //No more frust testing needed
    {
        drawFrontNodeOccNoVF(current);
        draw(current);
        if ( current.Back ) drawPVSOccNoVF(current.Back);
    }
    else
    {
        drawFrontNodeOcc(current, alreadyInside);
        draw(current);
        if ( current.Back ) drawPVSOcc(current.Back, alreadyInside);
    }
}

private void drawBackPVSOcc(Node* current, byte alreadyInside)
{
    if (current.Flag & FLAG_STOP_CULLING)
    {
        BBocclusionOff(); //Make sure occlusion is off
        if ( current.Back ) drawPVSN VF(current.Back);

        //current.draw();
        drawFrontNodeNoVF(current);
    }
    else if ( alreadyInside == 63 ) //No more frust testing needed
    {
        if ( current.Back ) drawPVSOccNoVF(current.Back);
        //draw(current);
        drawFrontNodeOccNoVF(current);
    }
    else
    {
        if ( current.Back ) drawPVSOcc(current.Back, alreadyInside);
        //draw(current);
        drawFrontNodeOcc(current, alreadyInside);
    }
}

private void drawFrontPVSOccNoVF(Node* current)
{
    if (current.Flag & FLAG_STOP_CULLING)
    {
        BBocclusionOff(); //Make sure occlusion is off
        drawFrontNodeNoVF(current);
        current.draw();
        if ( current.Back ) drawPVSN VF(current.Back);
    }
    else
    {
        drawFrontNodeOccNoVF(current);
        draw(current);
        if ( current.Back ) drawPVSOccNoVF(current.Back);
    }
}

```

```

    }
}

private void drawBackPVSOccNoVF(Node* current)
{
    if (current.Flag & FLAG_STOP_CULLING)
    {
        BBocclusionOff(); //Make sure occlusion is off
        if ( current.Back )    drawPVSNoVF(current.Back);
        //current.draw();
        drawFrontNodeNoVF(current);
    }
    else
    {
        if ( current.Back )    drawPVSOccNoVF(current.Back);
        //draw(current);
        drawFrontNodeOccNoVF(current);
    }
}

private void drawPVSChoose(Node* current, byte alreadyInside)
{
    if ( PlaneList[current.PlaneIndex].computeSide(OldCameraPos) > 0 ) //If in front
        drawFrontPVSOcc(current, alreadyInside);
    else
        drawBackPVSOcc(current, alreadyInside);
}

private void drawPVSChooseNoVF(Node* current)
{
    if ( PlaneList[current.PlaneIndex].computeSide(OldCameraPos) > 0 ) //If in front
        drawFrontPVSOccNoVF(current);
    else
        drawBackPVSOccNoVF(current);
}

protected void drawPVSOcc(Node* current, byte alreadyInside)
{
    if ( current.Mark == MarkedFrame && BSPPVS.View.isIn(current.BB, alreadyInside) )
    {
        addReadyNode(current, alreadyInside);
        checkReady();
    }
}

private void drawPVSOccNoVF(Node* current)
{
    if (current.Mark == MarkedFrame)
    {
        addReadyNodeNoVF(current);
        checkReady();
    }
}

private void checkReady()
{
    GLuint count;
    OcclInfo info;
    Node* node;

```



```

        if (Occlusion.checkAhead())
        {
            while (Occlusion.next(count, info))
            {
                if ( count != 0 )
                {
                    if (count > MostlyVisible) info.Current.Info.MarkOcc = MarkedFrame;
                    drawPVSChoose(info.Current, info.AlreadyInside);
                }
            }
        }

        if (OcclusionNoVF.checkAhead())
        {
            while (OcclusionNoVF.next(count, node))
            {
                if ( count != 0 )
                {
                    if (count > MostlyVisible) node.Info.MarkOcc = MarkedFrame;
                    drawPVSChooseNoVF(node);
                }
            }
        }
    }

    void drawPVSOcc(Node* current)
    {
        if ( PlaneList[current.PlanelIndex].computeSide(OldCameraPos) > 0 )
        {
            if ( current.Front.PlanelIndex >= 0 )
                drawPVSOcc(current.Front, 0);

            current.draw();

            if ( current.Back )
                drawPVSOcc(current.Back, 0);
        }
        else
        {
            if ( current.Back )
                drawPVSOcc(current.Back, 0);

            current.draw();

            if ( current.Front.PlanelIndex >= 0 )
                drawPVSOcc(current.Front, 0);
        }
    }

    OcclInfo info;
    GLuint count;
    Node *node;

    //Do anything that was missed
    while (!OcclusionNoVF.isEmpty() || !Occlusion.isEmpty() || !ReadyQ.isEmpty() || !ReadyQNoVF.isEmpty() )
    {
        while (ReadyQ.next(info))
        {
            drawPVSChoose(info.Current, info.AlreadyInside);
        }
    }

```

```

    }

    while (ReadyQNoVF.next(node) )
    {
        drawPVSChooseNoVF(node);
    }

    //Could do other things here!
    glFlush();

    while (Occlusion.next(count, info) )
    {
        if ( count != 0 )
        {
            if (count > MostlyVisible) info.Current.Info.MarkOcc = MarkedFrame;
            drawPVSChoose(info.Current, info.AlreadyInside);
        }
    }

    //Could do other things here!
    glFlush();

    while (OcclusionNoVF.next(count, node) )
    {
        if ( count != 0 )
        {
            if (count > MostlyVisible) node.Info.MarkOcc = MarkedFrame;
            drawPVSChooseNoVF(node);
        }
    }

    if (ReadyQ.isEmpty() && ReadyQNoVF.isEmpty()) //Try not to spend too much time
    {
        while (Occlusion.forceNext(count, info) )
        {
            if ( count != 0 )
            {
                if (count > MostlyVisible) info.Current.Info.MarkOcc = MarkedFrame;
                drawPVSChoose(info.Current, info.AlreadyInside);
            }
        }

        while (OcclusionNoVF.forceNext(count, node) )
        {
            if ( count != 0 )
            {
                if (count > MostlyVisible) node.Info.MarkOcc = MarkedFrame;
                drawPVSChooseNoVF(node);
            }
        }
    }
}

}

bool drawPVS()
{
    if ( CurrentLeaf ) //If in empty
    {

```

```

        //Mark each bit in the PVS up the the parent
        uncompressPVS();
        mark(NodeList[0]);
    }

    drawPVSOcc(&NodeList[0]);

    proccessDetail(BSPPVS.View);
    BBocclusionOff();

    return true;
}

protected:

    const uint BufferNodes = 2;        //How many nodes to correct before using occlusion
    const uint BufferOcclusion = 5000; //How many queries to have going at one time

    struct OcclInfo { Node* Current; byte AlreadyInside; };
    OcclusionQuery!(OcclInfo) Occlusion;
    OcclusionQuery!(Node*) OcclusionNoVF;

    circularQueue!(OcclInfo) ReadyQ;
    circularQueue!(Node*) ReadyQNoVF;

    GLuint NodeDisplayLists;
}
}

```

## 19 Appendix 6 - Examples

### 19.1 General BSP tree Compilation

The first step for compiling general BSP trees is to pre-compute the planes for all polygons. This technique will save time in the long run as polygon planes are reused often. The diagrams shown will be in 2D, however, the technique can easily be extended to 3D.

Step 1 - Pre-compute the polygon planes:

#### Compute Normals

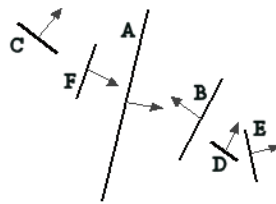


Figure 48. Step 1 (Compute Normals)

The arrows in Figure 48 illustrate the polygon planes normals which is also the side of the face (of that polygon) that can be seen. Code for calculating the plane normal can be found at 17.4.

The root of the tree begins its life pointing to nothing (NULL) tree until the BSP tree inserts the first polygons. Each polygon is placed into a new BSP tree node where the front and back children are initialised to NULL. In reality a programmer may choose to form these BSP tree nodes later in the BSP compilation stage when needed. Any other values (such as material) can be copied, and if a pointer to hold common data is used then it can reduce the amount of data being copied.

Step 2 - Picking a polygon from the list and making that the partitioning polygon:

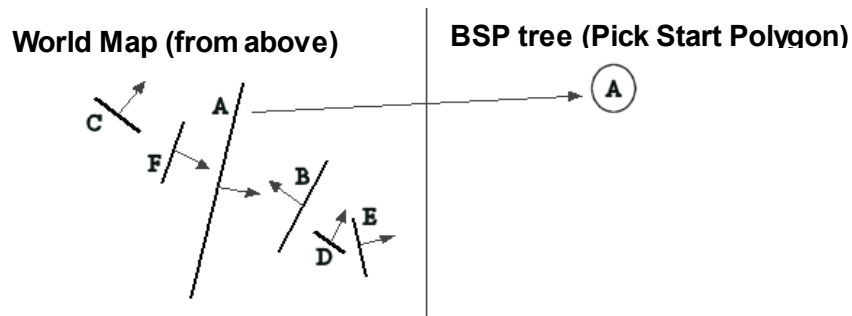
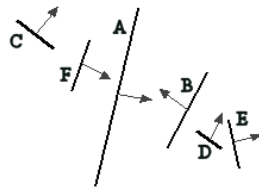


Figure 49. Step 2 (Pick a partitioning polygon)

In Figure 49 A was picked and made the root node (root = L(1)).

Step 3 - Split remaining polygons into two groups depending on which side they are on:

**World Map (from above)**



**BSP tree (Make Groups)**

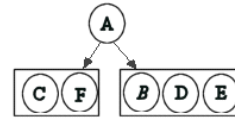


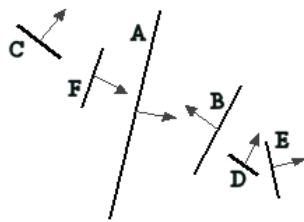
Figure 50. Step 3 (Split Remaining into groups)

In Figure 50, C and F are behind the normal so they go on the left, and B, D and E are in front so they go on the right.

The polygons' location in the tree can be maintained as a list (with divider) and can be kept on the program stack. For now these lists will be referred to as front (F) and back (B). Also the present node (which at this stage is the root) will be referred to as the "current node".

Step 4 - Repeat for every polygon in the child nodes (except for step 1):

**World Map (from above)**



**BSP tree (Make Groups)**

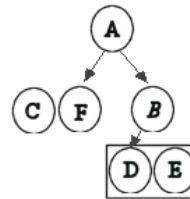
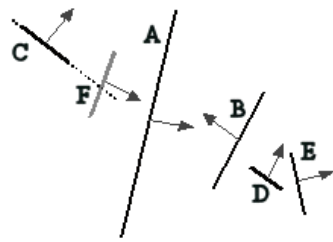


Figure 51. Step 4 (Recursively repeat)

The process is recursively repeated for each side of the tree. B is picked as the new sub-partition plane and attached to the right (front) side of A (Current Node = F[1]). D and E are behind B so they go on the left (back) node of B.

Step 4 continue...

### World Map (from above)



### BSP tree (Make Groups)

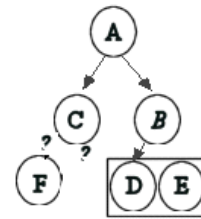
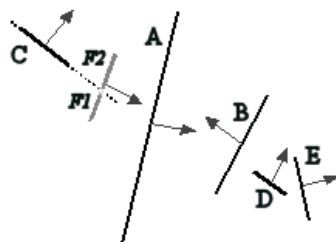


Figure 52. Step 4 con... (A problem)

In Figure 52 there is a problem with polygon F. Which side does polygon F belongs to (Left, right or both)? In this case the classifier (which has been used to determine the positions of all nodes so far) returns SPAN. The solution to this problem is to split the node using the intersection algorithm described previously in “Polygon Splitting”.

### World Map (from above)



### BSP tree (Make Groups)

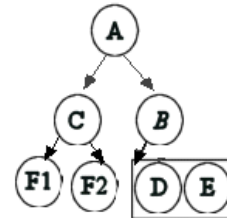
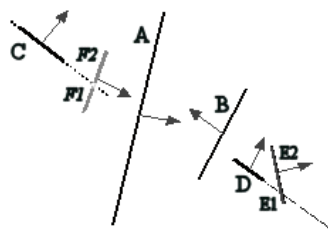


Figure 53. Step 4 con... (Splitting the node)

In Figure 53, F is split into F1 and F2 along the plane. Now F1 is behind so it should go on the left. F2 is in front so it should go on the right. Note that F no longer exists. F1 and F2 are placed into the left and right side of C because they are the final nodes for that part of the tree.

Finally... A complete BSP tree!

### World Map (from above)



### BSP tree (Make Groups)

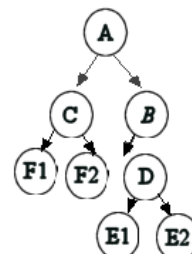


Figure 54. Completed General BSP tree

In Figure 54, E is split into E1 and E2 along the plane. E1 is behind D so it goes on the left. E2 is in front so it goes on the right.

## 19.2 General BSP tree at Runtime

This example will demonstrate the runtime traversal of a general BSP tree, for priority-ordering of polygons by their depth from the camera's point of view, using the tree that was generated in "General BSP tree Compilation".

The traversal starts from the root of the tree processing all the nodes that furthest from the camera first. For example: traverse behind node (if it exists), draw the current node then traverse the other child node.

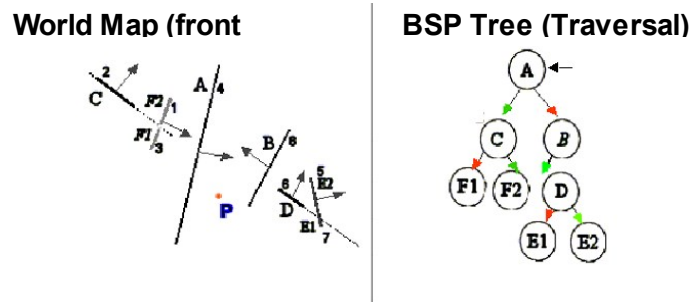


Figure 55. General BSP tree runtime (1)

In Figure 55, P is the Camera/Player's Position. The algorithm starts at A (the tree's root).

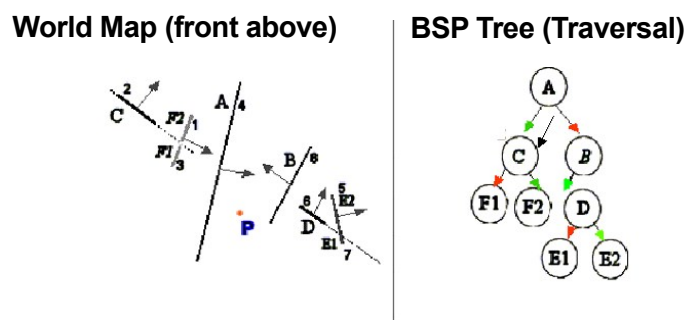
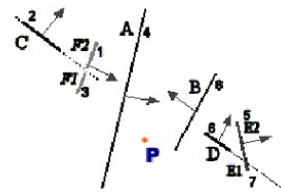


Figure 56. General BSP tree runtime (2)

The first task is to determine the direction of A plane in regards to the camera (the classify algorithm) and hence the path in the tree to take. Camera (P) is in front of A so that means that C is behind A and B is in front. So C is traversed first, as shown in Figure 56.

**World Map (front above)**

World Map (from above)



BSP Tree (Traversal)

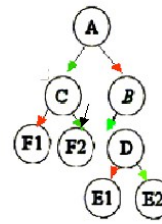
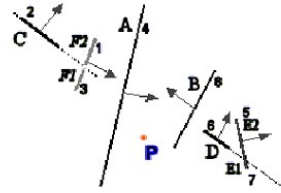


Figure 57. General BSP tree runtime (3)

In Figure 57, again C is classified (since it is not empty). This time P is behind C so F2 is traversed first.

World Map (front)



BSP Tree (Traversal)

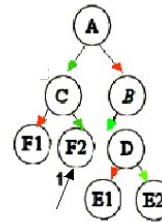
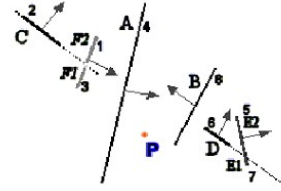


Figure 58. General BSP tree runtime (4)

In Figure 58, F2 has no children (it is a leaf) so F2 is drawn.

World Map (front)



BSP Tree (Traversal)

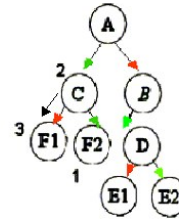
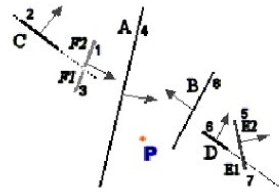
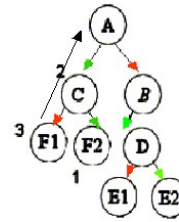


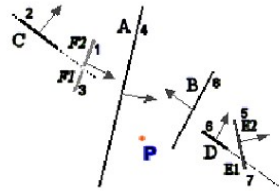
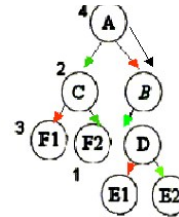
Figure 59. General BSP tree runtime (5)

In Figure 59, the algorithm is back at C, so C is drawn and F1 is traversed (because it is in front of C).

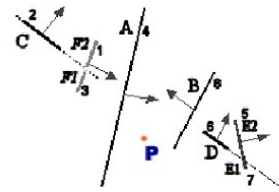
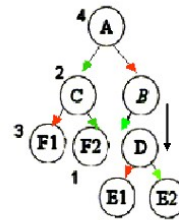


**World Map (front)****BSP Tree (Traversal)**Figure 60. General BSP tree runtime (6)

In Figure 60, F1 is drawn because it has no children and then because F1 and C have nothing more to do they are dropped of the program stack levelling the algorithm at A.

**World Map (front)****BSP Tree (Traversal)**Figure 61. General BSP tree runtime (7)

In Figure 61, A is drawn (because it has already traversed one child) and then B is traversed because it is in front (of A, C, F1 and F2) from the position of the camera.

**World Map (front above)****BSP Tree (Traversal)**Figure 62. General BSP tree runtime (8)

In Figure 62, P is now in front of B so that means D is behind, so D is traversed.

## World Map (front above)      BSP Tree (Traversal)

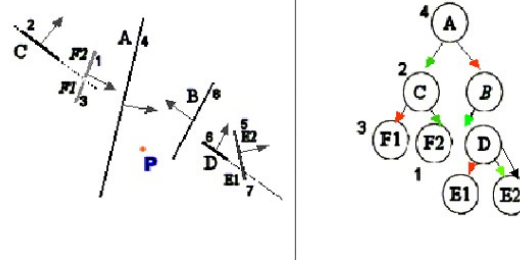


Figure 63. General BSP tree runtime (9)

In Figure 63, P is behind D so that means E2 is behind the camera also, so E2 is traversed.

## World Map (front above)      BSP Tree (Traversal)

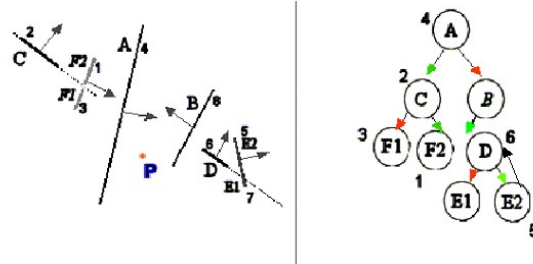


Figure 64. General BSP tree runtime (10)

In Figure 64, E2 has no children so it is drawn. Since E2 has nothing more to do so D is returned control.

## World Map (front)      BSP Tree (Traversal)

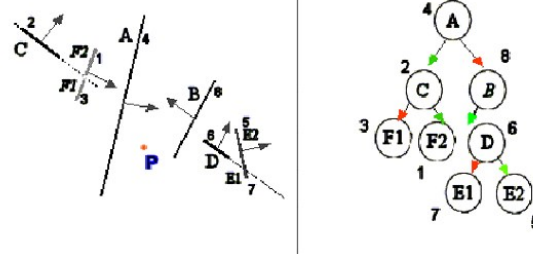


Figure 65. General BSP tree runtime (11)

In Figure 65, E1 is drawn as it is in front of D.

## World Map (front above)      BSP Tree (Traversal)

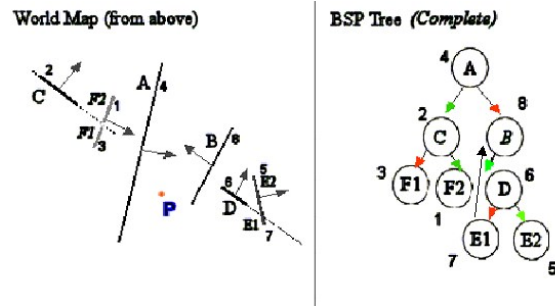


Figure 66. General BSP tree runtime (12)

In Figure 66, B has been drawn so now the algorithm heads to the top of the tree.

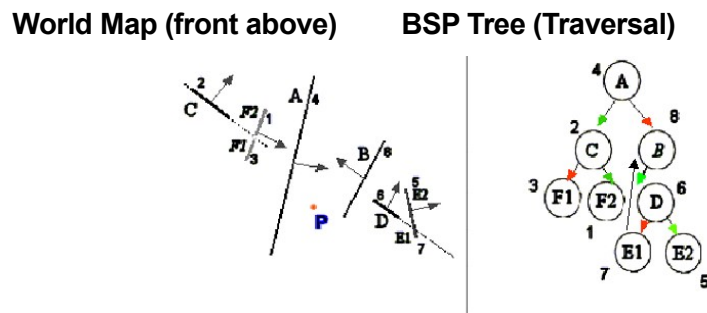


Figure 67. General BSP tree runtime (13)

In Figure 67, all polygons are now drawn in the correct priority-order, which proves that the algorithm worked for at least this simple test case example. However note that this example did not reduce polygons sent to the display.

### 19.3 BB BSP tree Compilation

Following is an example of adding AABB to the BSP tree produced in "General BSP tree Compilation". Once the two children nodes BB are known the parent nodes BB can be computed.

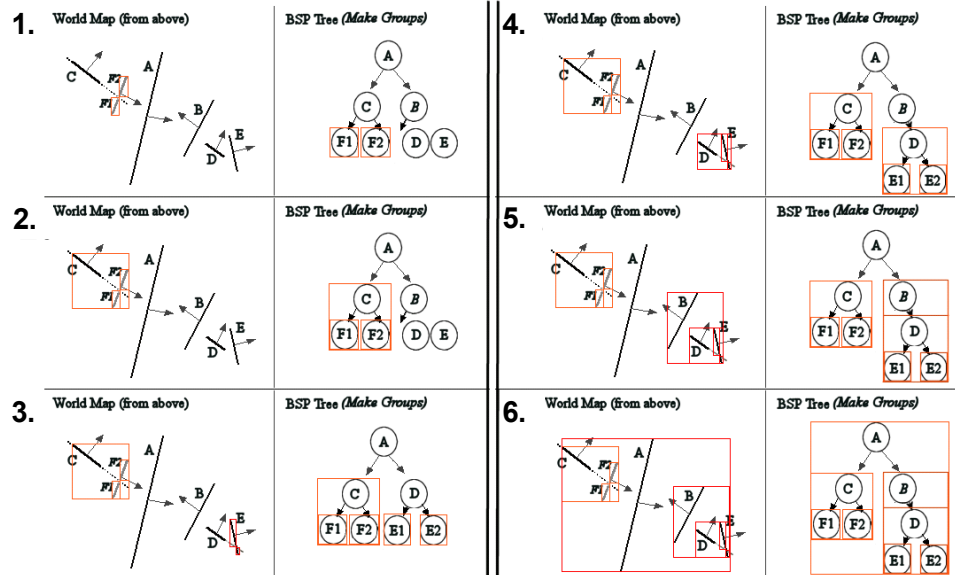


Figure 68. BB BSP tree compilation example

Figure 68 shows the inclusion of axis aligned BB, which was built in 19.1. As each leaf is found, the algorithm returns the current nodes bounding box to that node's parent, allowing the parent to produce its own BB. The AABB on the "world map" indicates the area the BB consumes on the map; where as the BBs on the tree represent BBs that have been computed.

## 19.4 Runtime BB BSP tree

This example continues from the tree created in 19.3. In the following example, camera P has a view frustum showing what the camera can see. Nodes determined renderable are shaded.

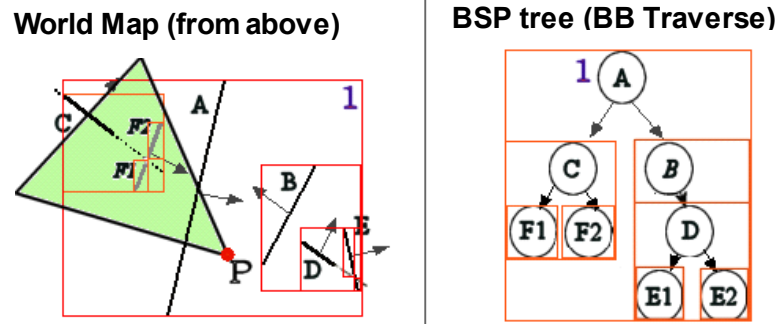


Figure 69. Runtime BSP Example 1

As Figure 69 shows, the traversal has started at A, the root of the tree.

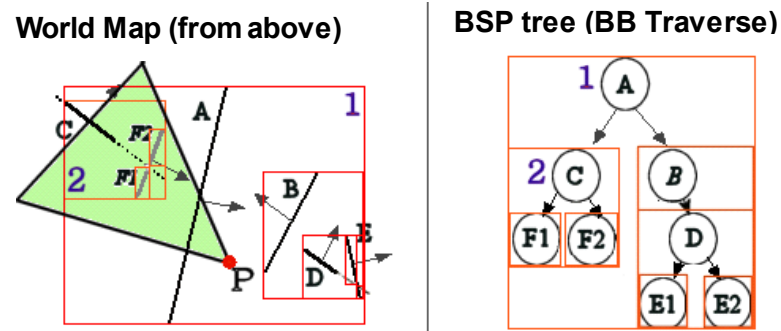


Figure 70. Runtime BSP Example 2

As Figure 70 shows, A's BB is partly within of the view frustum so the traversal has continued down C.

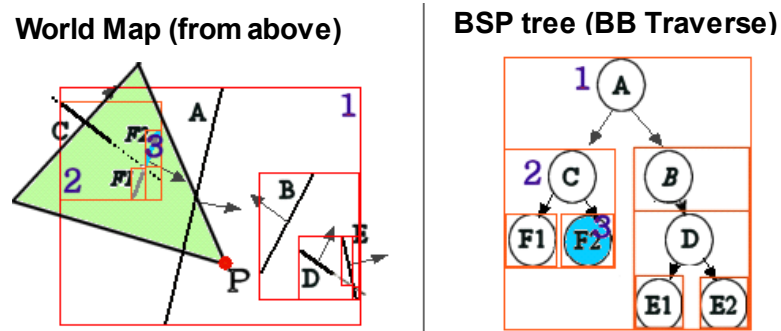


Figure 71. Runtime BSP Example 3

As Figure 71 shows, C's BB is within the view frustum so the traversal has continued down to F2. F2's BB is completely inside; therefore test has stopped at this point, besides F2 is a leaf node anyway. F2 is sent down the rendering pipeline (drawn).

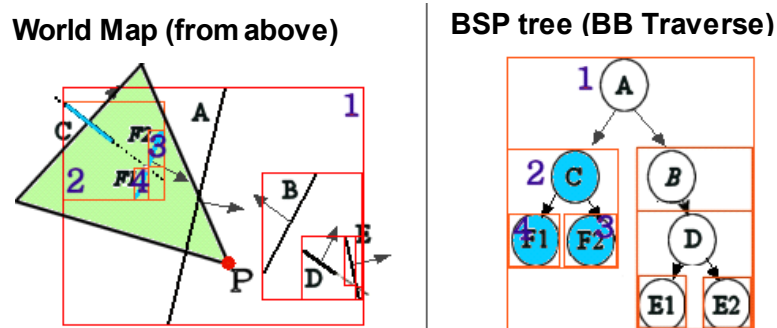


Figure 72. Runtime BSP Example 4

As Figure 72 shows, the traversal has returned to C, which is drawn and then traversed F1. F1's BB is completely inside the frustum. Therefore testing has stopped at this point, besides F1 is a leaf node anyway. F1 is sent down the rendering pipeline (drawn).

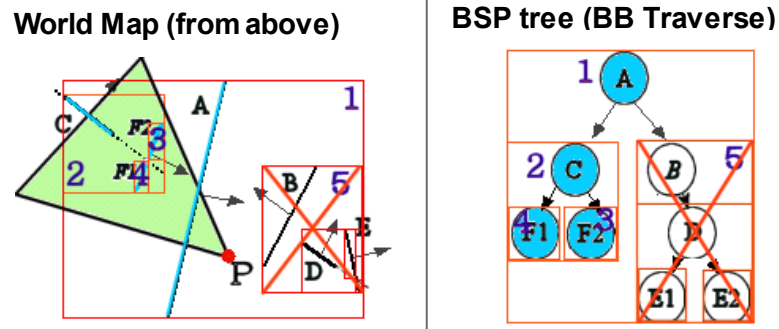


Figure 73. Runtime BSP Example 5

As Figure 73 shows, the traversal has returned back to A, which is then drawn. The algorithm then tested B to see if it was within the frustum. B is not within the frustum and there are no more nodes to process so traversal has ended (for this frame). D and E were not even tested which is indicated in the Figure 73 by the crossing out. On a larger scale maps many more nodes could have been discarded which can be a quite significant performance improvement when compared to simply rendering every polygon.

## 19.5 Solid BSP tree Compilation

This example shows the construction of a solid leaf BSP tree. The diagram represents a 2D room with a brick wall (pillar) inside it.

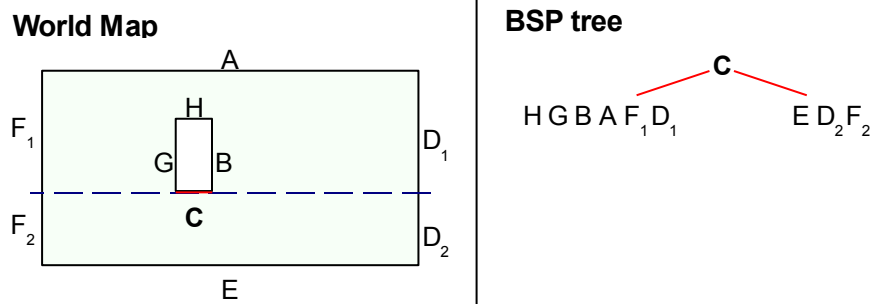


Figure 74. Solid BSP Compilation 1

In Figure 74, E is on in front C and G, B, A and H are behind. F and D needed to be split into  $F_1$ ,  $F_2$ ,  $D_1$  and  $D_2$  respectfully.

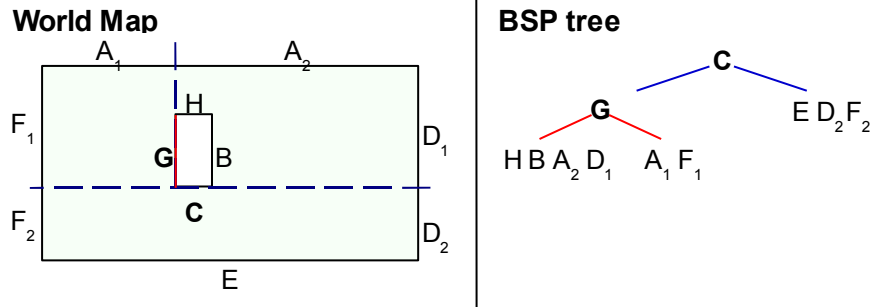


Figure 75. Solid BSP Compilation 2

In Figure 75, G is the new partitioning plane therefore A needs to be split. The planes B, D<sub>1</sub>, A<sub>2</sub> and H are placed behind and A<sub>1</sub> and F<sub>1</sub> in front.

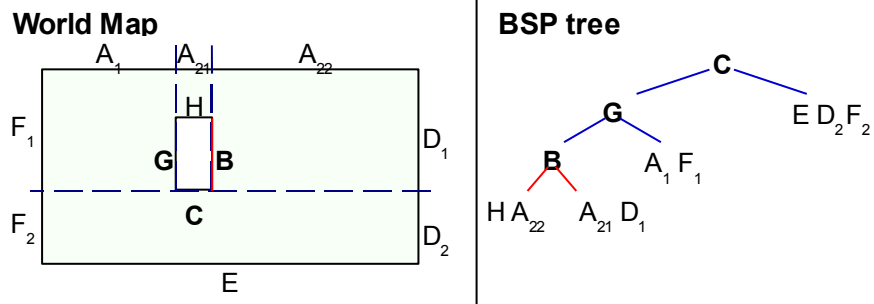


Figure 76. Solid BSP Compilation 3

In Figure 76, B is chosen as the splitter. A<sub>2</sub> needs to be split again into A<sub>21</sub> and A<sub>22</sub>. D<sub>1</sub> and A<sub>21</sub> are in front and A<sub>22</sub>, H is behind.

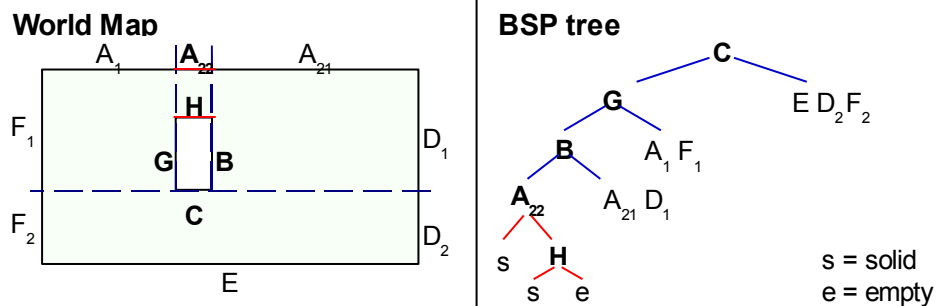


Figure 77. Solid BSP Compilation 4

In Figure 77, A<sub>22</sub> is chosen as the splitter. The node H is on the front of A<sub>22</sub>. Has H is the only polygon left so H becomes a node with a solid back and empty front leaf.



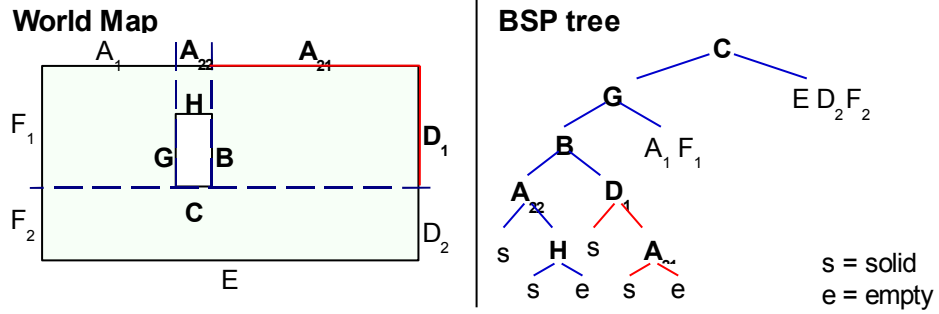


Figure 78. Solid BSP Compilation 5

In Figure 78,  $D_1$  is now the partitioning plane.  $A_{21}$  ends up being on the front side and is made into a node.

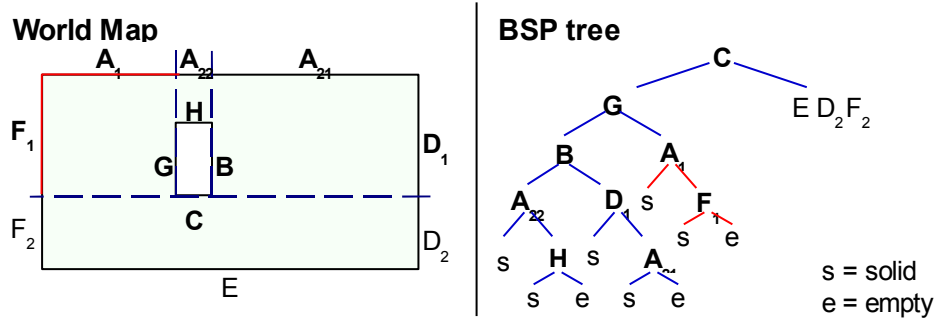


Figure 79. Solid BSP Compilation 6

In Figure 79,  $A_{21}$  is chosen as the splitting plane.

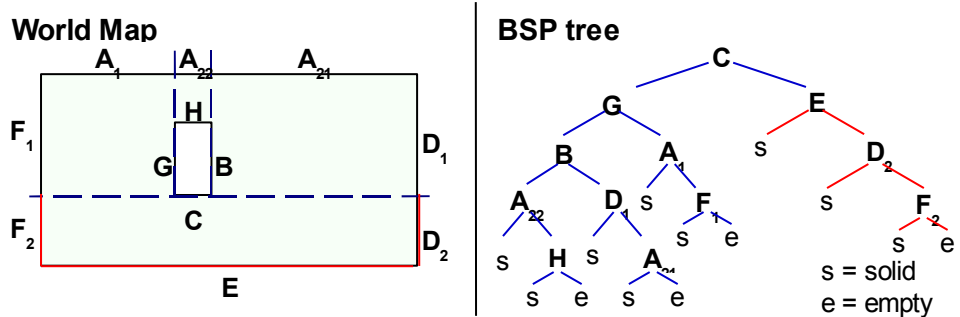


Figure 80. Solid BSP Compilation 7

In Figure 80 E then  $D_2$  then  $F_2$  (in succession) become the partitioning planes to form the final tree.

## 19.6 Portal Creation

This example shows how portals could be generated using the 2D solid BSP tree built in 19.5. While planes (nodes) are represented by capital letters, portals are represented by lowercase letters. The algorithm traverses each node in the tree to find the portals shapes and how portals link empty leaves together.

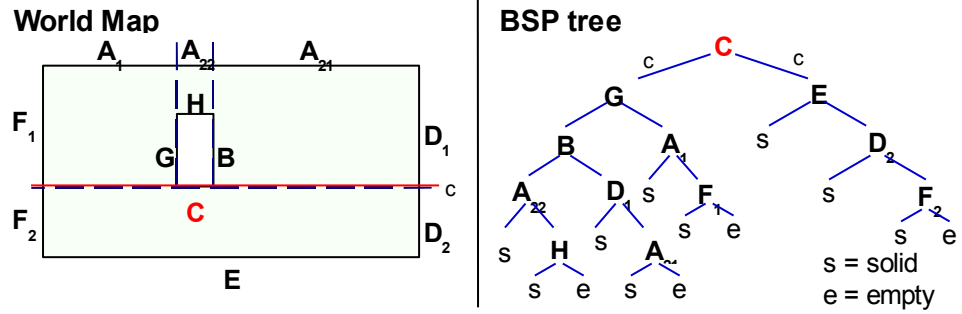


Figure 81. Portal generation 1

In Figure 81, a large polygon  $c$  is created along the plane  $C$ .  $c$  is pushed down to nodes  $G$  and  $E$  with  $G$  on the back and  $E$  on the front side of  $c$ . Note that  $c$  represented in the tree is actually a link to  $c$ , therefore there is only one instance of  $c$ .

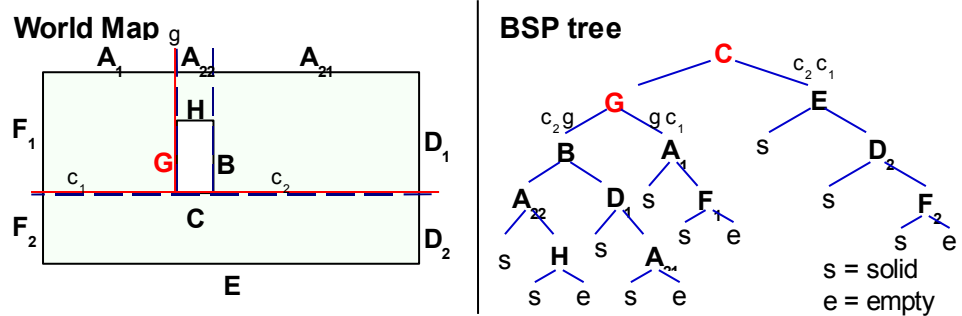


Figure 82. Portal generation 2

In Figure 82, a large polygon  $g$  is created along the plane  $G$ .  $g$  is trimmed by its parent node  $C$ .  $g$  is pushed down to nodes  $A_1$  (front) and  $B$  (back).  $c$  is on both sides of  $G$  so it is split into  $c_1$  (front) and  $c_2$  (back). Note that the arrows between portals (ie  $c$ ) in the diagram are used to indicate that links are maintained between them.

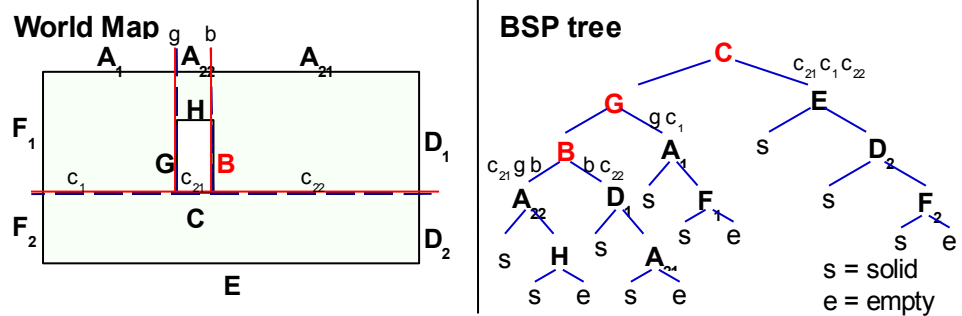


Figure 83. Portal generation 3

In Figure 83, a large polygon  $b$  is created along the plane  $B$ .  $b$  is trimmed by its parent nodes  $G$  (which has no effect) and  $C$ .  $b$  is pushed down to nodes  $A_{22}$  (front) and  $D_1$  (back).  $g$  is behind  $B$  so  $g$  is pushed to the back of  $B$ .  $c_2$  is split by  $B$  so  $c_{21}$  and  $c_{22}$  is pushed down the back and front of  $B$ , respectively.

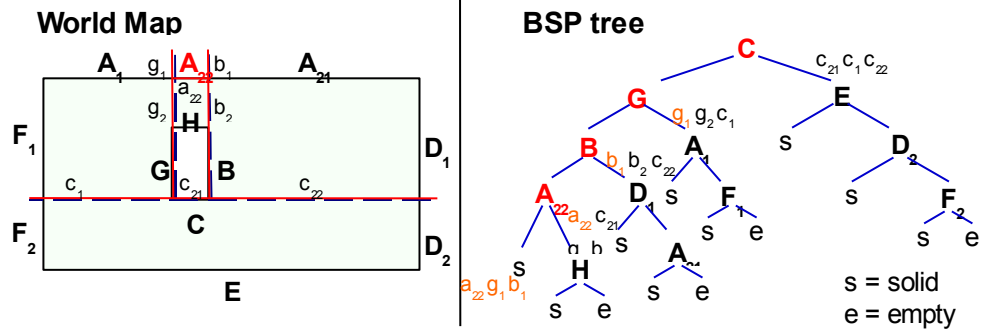


Figure 84. Portal generation 4

In Figure 84, a large polygon  $a_{22}$  is created along the plane  $A_{22}$ .  $a_{22}$  is trimmed by its parent nodes B, G and C (which has no effect).  $a_{22}$  is pushed down to a solid node (back) and H (front).  $g$  and  $b$  are split by  $A_{22}$  into  $g_1$ ,  $g_2$ ,  $b_1$  and  $b_2$ .  $c_{21}$  is pushed down the front side of  $A_{22}$ .  $a_{22}$ ,  $b_1$  and  $g_1$  can be removed as they are in a solid.

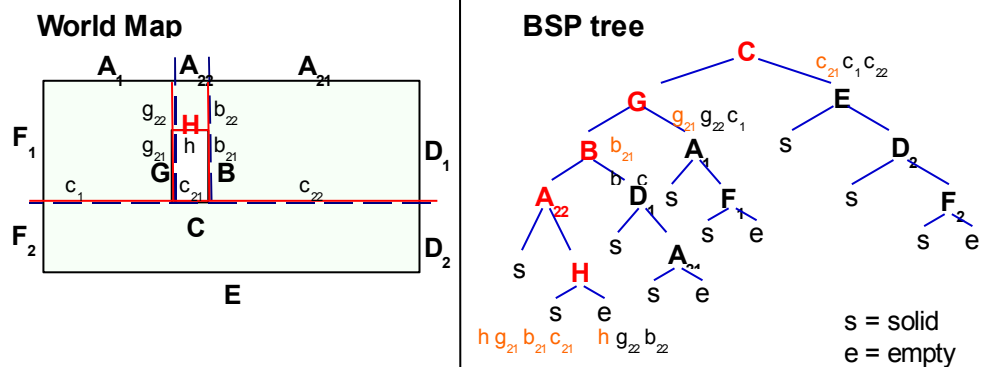


Figure 85. Portal generation 5

In Figure 85, a large polygon  $h$  is created along the plane  $H$ .  $h$  is trimmed by its parent nodes  $A_{22}$  (which has no effect), B, G and C (which has no effect).  $h$  is pushed down to a solid node (back) and an empty node (front).  $g_2$  and  $b_2$  are split by  $H$  into  $g_{21}$ ,  $g_{22}$ ,  $b_{21}$  and  $b_{22}$  and pushed down the relevant side.  $c_{21}$ ,  $h$ ,  $g_{21}$  and  $b_{21}$  can be removed as they are in solids.

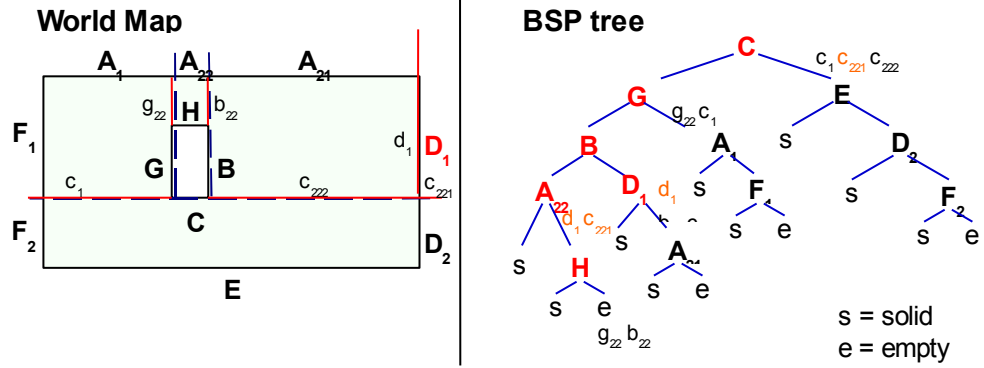


Figure 86. Portal generation 6

In Figure 86, a large polygon  $d_1$  is created along the plane  $D_1$ .  $d_1$  is trimmed by its parent nodes B (which has no effect) and G (which has no effect) and C.  $d_1$  is pushed down to a solid node (back) and  $A_{21}$  (front).  $c_{22}$  is split into  $c_{221}$  and  $c_{222}$ .  $b_{22}$  is pushed down the front ( $A_{21}$ ) of  $d_1$ .  $d_1$  and  $c_{221}$  can be removed as they are in a solid.

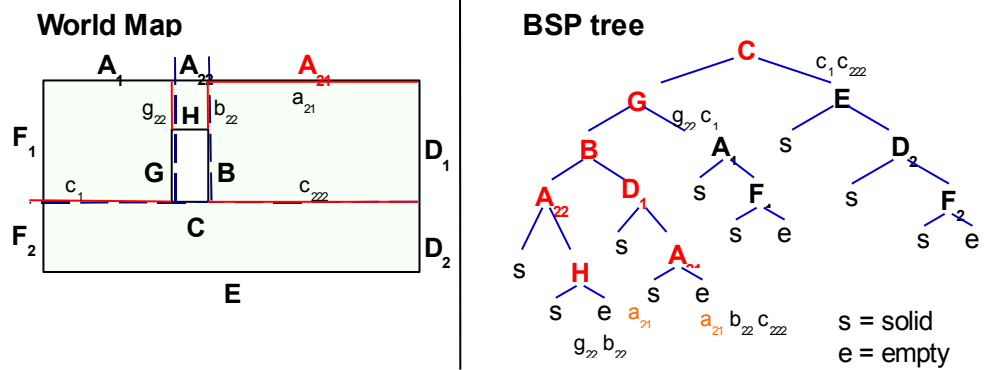


Figure 87. Portal generation 7

In Figure 87, a large polygon  $a_{21}$  is created along the plane  $A_{21}$ .  $a_{21}$  is trimmed by its parent nodes  $D_1$ , B and G (which has no effect as the portal was already trimmed by B) and C (which has no effect).  $a_{21}$  is pushed down to a solid node (back) and empty node (front).  $b_{22}$  and  $c_{22}$  are both in front of  $A_{21}$ .  $a_{21}$  be removed as it is in a solid.

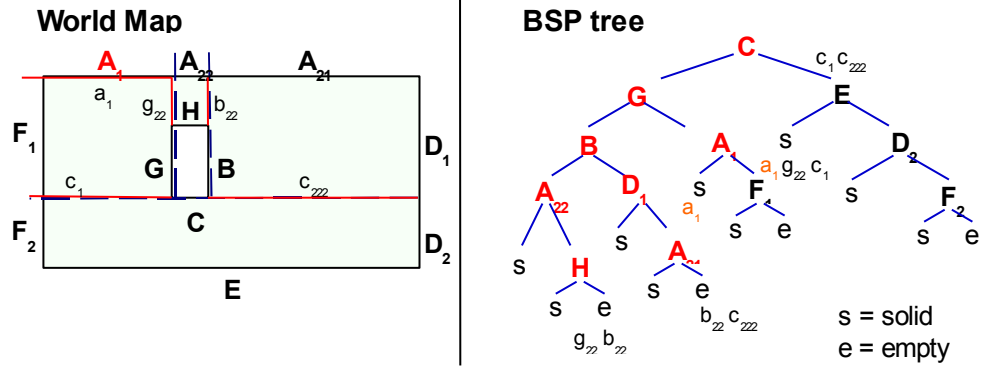


Figure 88. Portal generation 8

In Figure 88, a large polygon  $a_1$  is created along the plane  $A_1$ .  $a_1$  is trimmed by its parent nodes  $G$  and  $C$  (which has no effect).  $G_{22}$  and  $c_1$  are both in front of  $A_1$ .  $a_1$  can be removed as it ended up in solid space.

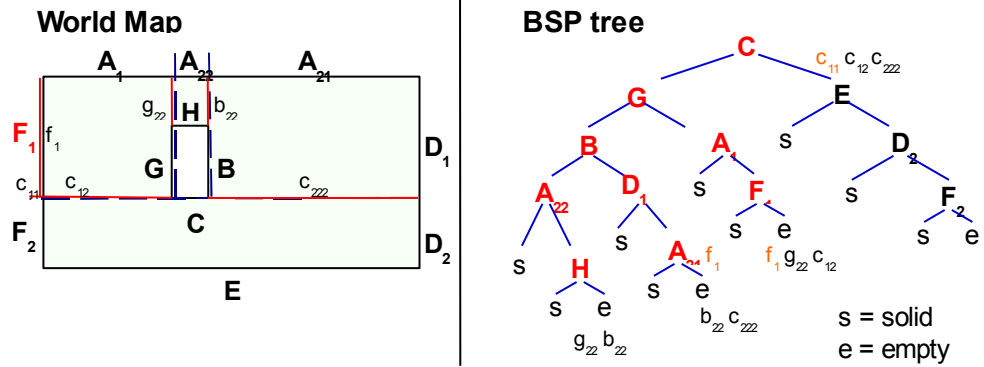


Figure 89. Portal generation 9

In Figure 89, a large polygon  $f_1$  is created along the plane  $F_1$ .  $f_1$  is trimmed by its parent nodes  $A_1$ ,  $G$  (which has no effect) and  $C$ .  $g_{22}$  is in front of  $F_1$ .  $c_1$  is split into  $c_{11}$  and  $c_{12}$  along the plane  $F_1$ .  $f_1$  and  $c_{11}$  can be removed as they are in solid space.

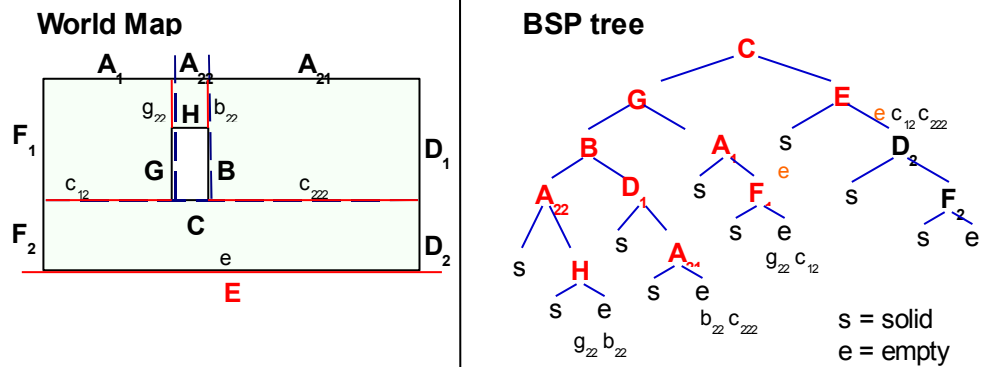


Figure 90. Portal generation 10

In Figure 90, a large polygon  $e$  is created along the plane  $E$ .  $e$  is trimmed by its parent node  $C$  (which has no effect).  $C_{12}$  and  $C_{222}$  are in front of  $E$ .  $e$  can be removed as it is in solid space.

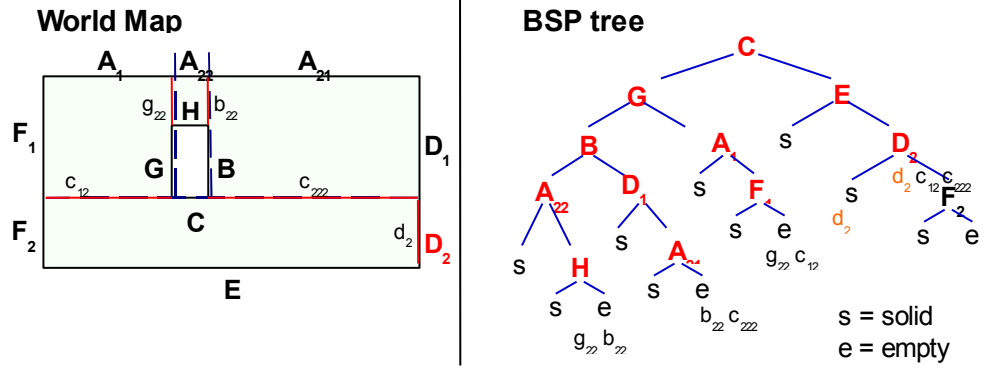


Figure 91. Portal generation 11

In Figure 91, a large polygon  $d_2$  is created along the plane  $D_2$ .  $d_2$  is trimmed by its parent nodes  $E$  and  $C$ .  $C_{12}$  and  $C_{222}$  are in front of  $D_2$ .  $d_2$  can be removed as it is in solid space.

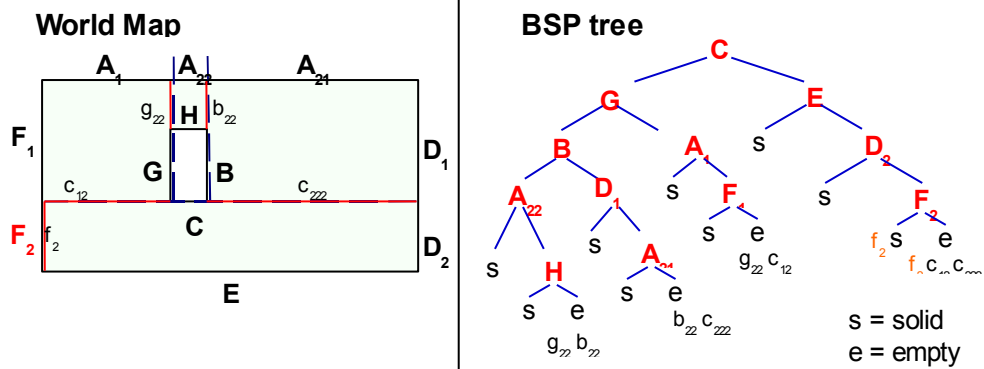


Figure 92. Portal generation 12

In Figure 92, a large polygon  $f_2$  is created along the plane  $F_2$ .  $f_2$  is trimmed by its parent nodes  $D_2$ ,  $E$  and  $C$ .  $C_{12}$  and  $C_{222}$  are in front of  $F_2$ .  $f_2$  can be removed as it is in solid space. The portals would then be filtered to remove any bad portals. The filtering stage is not shown as there are no bad portals to remove, therefore portal generation is complete.

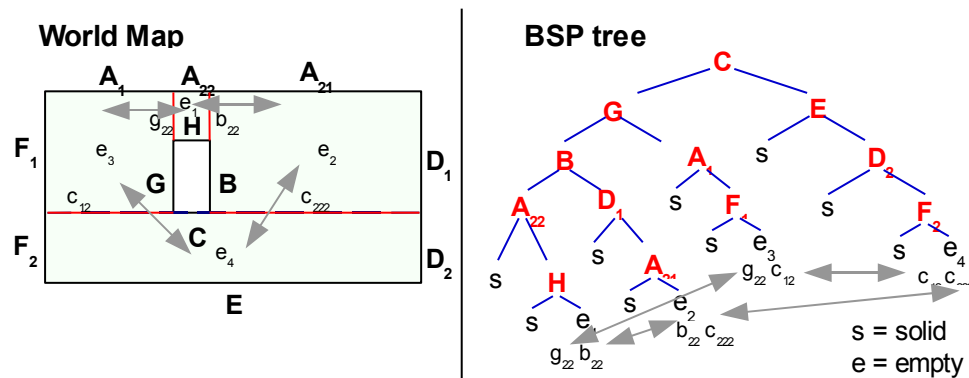


Figure 93. Portal generation 13

Figure 93 shows the portals found for the world map and the linkages (arrows) between each portal. The four empty leaves (sectors) have been labelled  $e_1$ ,  $e_2$ ,  $e_3$  and  $e_4$ .  $e_1$  is connected to  $e_2$  and  $e_3$  by portals  $g_{22}$  and  $b_{22}$  respectively.  $e_2$  is connected to  $e_1$  and  $e_4$  by portals  $b_{22}$  and  $c_{222}$ .  $e_3$  is connected to  $e_1$  and  $e_4$  by portals  $g_{22}$  and  $c_{12}$  respectively.  $e_4$  is connected to  $e_3$  and  $e_1$  by portals  $c_{12}$  and  $c_{222}$  respectively.

## 19.7 Anti-penumbra Portal Culling

A simple example of the anti-penumbra portal culling algorithm:

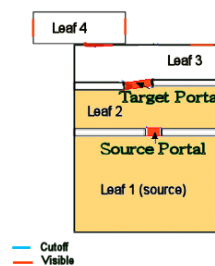


Figure 94. A leaf is chosen

In Figure 94, the shaded area represents rooms known to be visible to A. Leaf 1 is chosen as the source leaf. Two areas that leaf 1 definitely knows it can see are leaf 2 and itself. Leaf 2 should be appended into leaf 1's PVS.

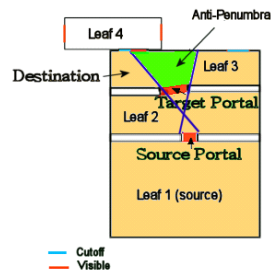


Figure 95. Anti-penumbra is created

In Figure 95, an anti-penumbra is created using the source portal and the target portal. Part of the portal connecting to leaf 4 needs to be clipped away (shown in lighter shade). Also one of the doors in leaf 3 is not within the anti-penumbra so that is also clipped. Leaf 3 (the destination) is added to leaf 1's PVS.

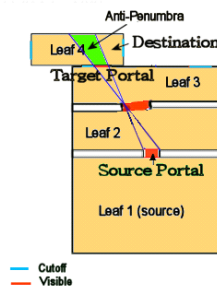


Figure 96. Sub-anti-penumbra is created

In Figure 96, a new anti-penumbra is created using the remains of the portal connecting leaf 3 and 4. The two portals in leaf 4 are not visible within the anti-penumbra so they are discarded. Leaf 4 (destination) are added to leaf 1's PVS. Leaf 4, leaf 3, leaf 2 and leaf 1 (respectively) have no more doors to process so this process is starts over again for another leaf in the tree until all PVS is known.



## 20 Appendix 7 – Using the demo

### 20.1 The Compiler

The BSP compiler requires 3Dsmax 5 to be installed. Although a binary file (BSPexport.dle) is provided the source code can be compiled using visual studios (net) as described below.

#### 20.1.1 Source Compilation

To compile the binary file:

- 1) Select Release mode
- 2) Open "BSPexport.sln" in Microsoft Visual Studios net
- 3) Select: solution explorer → export1 → right button → properties
- 4) Select: General
- 5) In output directory, change "F:\3DsMax5\" to the folder location of 3Dsmax 5 (most commonly "C:\3DsMax5\")
- 6) Press F5 (compile and build)

#### 20.1.2 Installation (with the binary file)

To simply use the plug-in without compiling, copy the "BSPexport.dle" file into the 3Dsmax "plugins" folder located in the 3Dsmax folder.

#### 20.1.3 Exporting

Before exporting a map into an mbs file, a map needs to be loaded into 3Dsmax. A couple of max maps are included on the CD for to try. Here are some important points to check before compiling the map:

- The map must be solid. A map can be checked to make sure it is solid by selecting each object and running the 3Dsmax STL modifier on them. If there are no errors then the map is solid.
- A start location for the camera must be assigned. A start location is specified using a dummy object with the term "[PLAYER POS]" within the object name. Dummy objects can be found under the helpers section in the create panel. These object must be placed inside the solid structure, otherwise the camera will have no where to land when the program starts. More then one of these cameras can be placed in a map.
- There must be at least one object greater then the size of detail which is by default 512x512x512units.
- Curved objects will be converted into mesh data. This may result in slow tree builds. That is not to say that the algorithm couldn't be modified to take advantage of curved objects in the future. Curved objects would be simplified into structural planes and only rendered as curved objects once they are known to be visible.

Once the map has met the above requirements it can be exported by:

- 1) Selecting File → Export
- 2) Save as Type → My BSP format (\*.MBS)
- 3) Choose the name of the file to save and where to save it
- 4) Click Save

Then a BSP compilation dialog like Figure 97 will popup which allows the user to specify how to build the BSP tree.

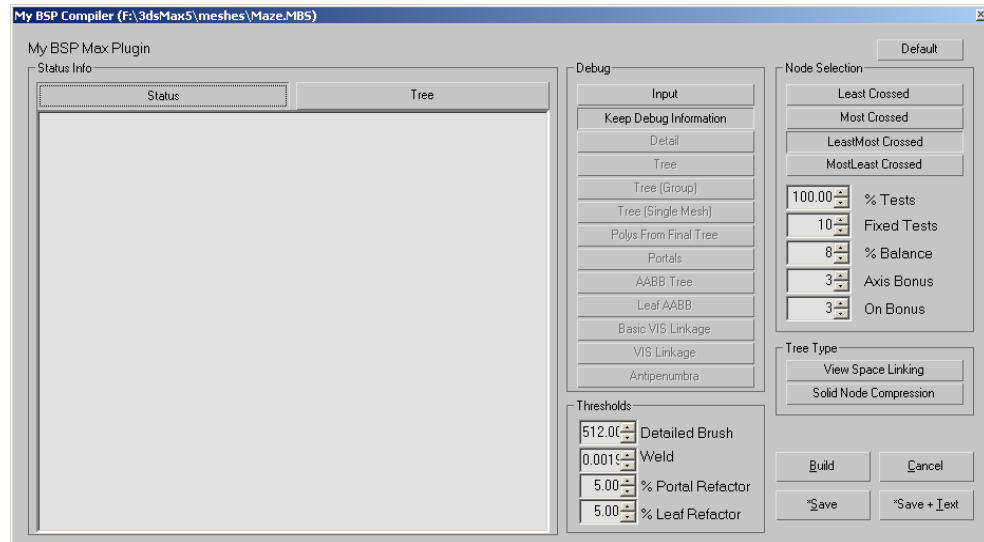


Figure 97. Screen shot of the BSP compiler IDE

The BSP compilation dialog is setup with good defaults to make compilation easy. If save is selected the BSP tree will be saved to the output mbs file which can be tested in the engine. A detailed description of all the commands follows:

### Status Window

The status window shows information about what occurred when building. The status window can be in two views which can be toggled by either clicking on the “Status” button or the “Tree” button.

### Status

Status mode outlines what the compiler is doing including:

- the stage being performed
- errors that may occur
- details about the data being used
- the amount of time a particular stage takes

Figure 98 shows the status view after a BSP tree has been built.

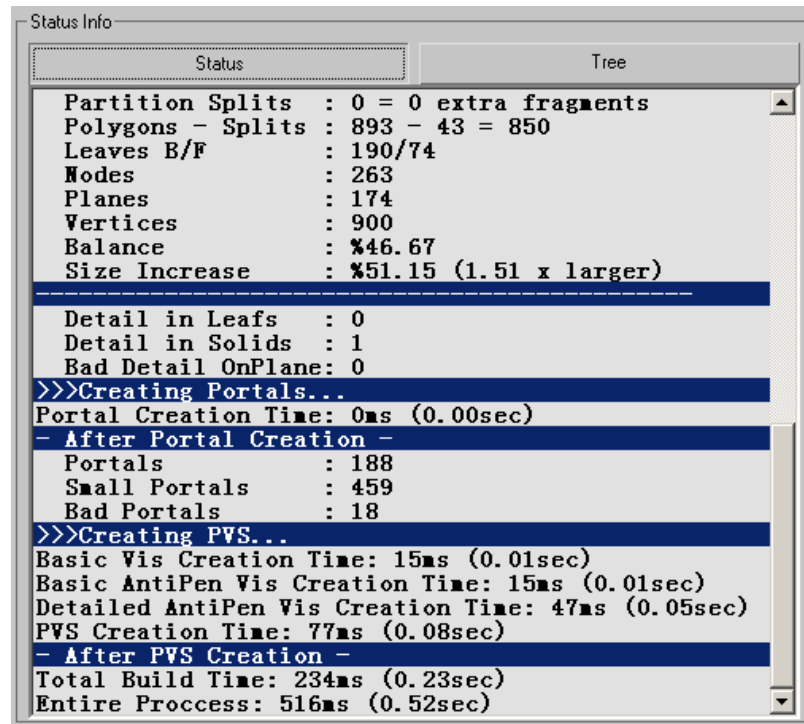


Figure 98. Status View

### Tree

Tree mode shows the resulting BSP tree and information about each node and leaf. Only standard BSP tree information is shown, that is, compact and node link tree details are not represented. Tree mode is not recommended for large BSP trees as these will most probably be too large for the tree view to display in any reasonable time.

Figure 99 shows the tree view once a bsp has been built.

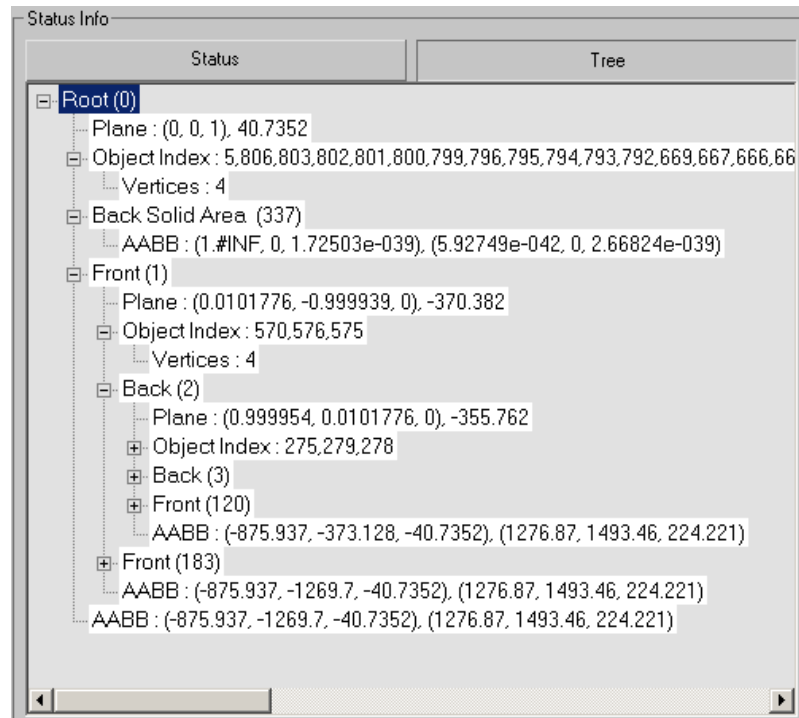


Figure 99. Tree View

## Control Commands

These are the major commands that will cause the plugin to react. Figure 100 shows the available control commands.



Figure 100. Control Commands

### **Build**

Builds the BSP tree but does not save it or exit the dialog. After a BSP has been built more of the debug options become available (see debug). Also the status info window can be examined.

### **Cancel**

Exits without saving, all changes will be lost.

### **Save**

Save will build (if the tree has not already been) and save the result to an mbs file. If there is no error then the dialog will exit returning focus back to 3Dsmx.

### **Save + Text**

Save + Text will build an mbs file and also a text file (.txt) which is in the same structure as the binary file except in a readable format with comments. The text file is primary used for debugging.

## Build Modifiers

Build modifiers are anything that affects how a tree is built. Therefore they need to be before building the bsp tree.

### ***Tree Type***

The tree type specifies which type of tree to build. In no tree type is selected then the standard (comparison) tree is built. View Space Linking and Solid Node Compression trees can be selected singularly or together to build a tree with the combination. An occlusion tree type is not provided because occlusion bsp trees do not require any pre-compilation. Figure 101 shows the tree type selection menu.

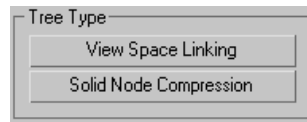


Figure 101. Tree Type

### ***Thresholds***

Thresholds define the point at which the state in question is too large to continue applying. Figure 102 shows the thresholds value panel.

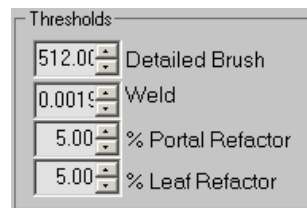


Figure 102. Thresholds

#### ***Detailed Brush***

Defines the maximum cubed size an object can be to be considered detail.

#### ***Weld***

Defines how close two vertices need to be, to be considered as the same vertex.

#### ***Portal Refactor***

Defines the percentage of portal PVS refactors to perform. 100% being one refactor for every node.

#### ***Leaf Refactor***

Defines the percentage of leaf PVS refactors to perform. 100% being one refactor for every node.

### ***Node Selection***

Node selection is used to minimise splits and improve the balance of the tree. Figure 103 shows the selection panel.

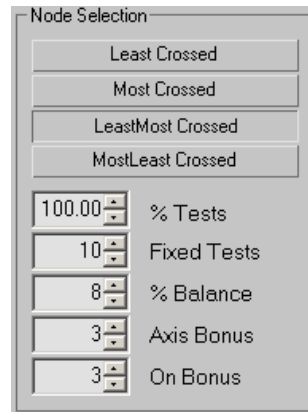


Figure 103. Node Selection

### *Least Crossed*

When selected least crossed selection will be performed.

### *Most Crossed*

When selected most crossed selection will be performed.

### *Least Most Crossed*

When selected least crossed will be performed first and any ties will be broken by most crossed. This is the default option as it appears to give the most better result when compared to the other selection choices.

### *Most Least Crossed*

When selected most crossed will be performed first and any ties will be broken by least crossed.

### *Tests*

The test percentage is the percentage of polygons to scan when picking the next node using the picking algorithm. Lower values will improve performance but may result in a more badly-organized tree. The number of actual tested polygons is:

$$[\text{Number of polygons to test}] = [\text{Remaining Polygons}] * \text{Tests} + [\text{Fixed Tests}]$$

Which is clipped to [Remaining Polygons].

### *Fixed Tests*

Fixed tests are the fixed number of polygons to scan when picking the next node using the picking algorithm. Lower values will improve performance but may result in a more badly-organized tree. The number of actual tested polygons is:

$$[\text{Number of polygons to test}] = [\text{Remaining Polygons}] * \text{Tests} + [\text{Fixed Tests}]$$

Which is clipped at [Remaining Polygons].

### *Balance*

Balance is a percentage representing the amount of preference to give to balancing over splitting. Generally a small percentage (ie 8%) to balancing will actually help reduce splitting as opposed to having no balance favouring at all.

### *Axis Bonus*

The amount of bonus score to give to axis aligned polygons.

### *On Bonus*

The amount of bonus score to give to on plane polygons.

## **Debug**

Most debug information is enabled after the map has been built. Debugging information is created within the 3Dsmx as objects. In order to prevent debug information from contributing to map builds all debug objects includes the term "[IGNORE]" in the name. Figure 104 shows the debug panel after the map has been built.

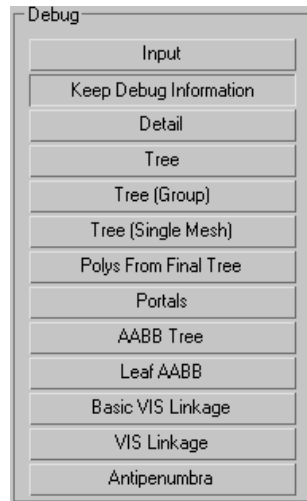


Figure 104. Debug

### ***Input***

As Figure 105 illustrates, input will output all the polygons (without material information) that were passed into the BSP tree compiler. Note that planes have been grouped together and curved objects have been converted into meshes.

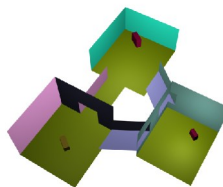


Figure 105. Input

### ***Keep Debug Information***

If pressed in, debugging information will be kept when the BSP tree is built (the user presses build). Otherwise the program will be more memory conservative and delete memory that is held for debugging purposes after a build.

### ***Detail***

As Figure 106 illustrates, the detail button will show the detail found in the scene.



Figure 106. Detail

### ***Tree***

As Figure 107 illustrates, the tree button will display the polygon groups that are sliced up into the tree. Each plane is an individual object. Note that tree will also show how objects are split.

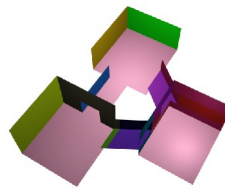


Figure 107. Tree

### ***Tree (Group)***

As Figure 108 illustrates, tree group is similar to the tree button but rather than show individual surfaces, they are combined into a tree of groups so the actual form of the tree can be seen. Each node in the tree group has a splitter and the two sub-groups of polygons. For complex maps, tree group can take much longer to create than tree.

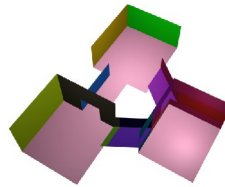


Figure 108. Tree (group)

### ***Tree (Single Mesh)***

As Figure 109 illustrates, the tree single mesh button will create a mesh from what is generated in the BSP tree, including polygons that are split.



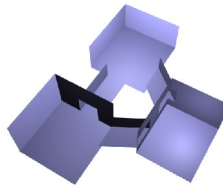


Figure 109. Tree (Single Mesh)

### ***Polys From Final Tree***

As Figure 110 illustrates, the polys from final tree button will generate a mesh from the result of the BSP generation and polygon reformation. Note that there will be no split polygons in this tree because the final tree uses original polygons.

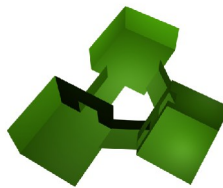


Figure 110. Final Tree

### ***Portals***

As Figure 111 illustrates, the portals button will show the portals used to create the PVS data.

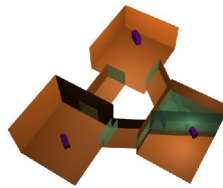


Figure 111. Portals

### ***AABB Tree***

As Figure 112 illustrates, the AABB tree button will show the hierarchical axis aligned bounding boxes of the tree.

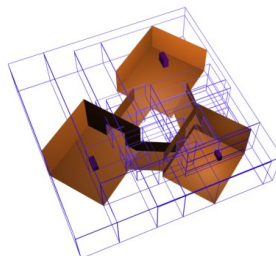


Figure 112. AABB Tree

### **Leaf AABB**

As Figure 113 illustrates, the leaf AABB button will show the axis aligned leaf bounding boxes of the tree. Note that a leaf bounding box is not the bounding box that surrounds the leaf but rather the bounding box that surrounds the portals the leaf contains.

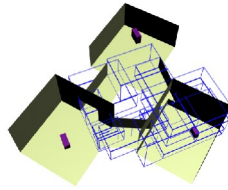


Figure 113. Leaf AABB

### **Basic Vis Linkage**

As Figure 113 illustrates, the basic vis linkage button will show the linkages that have been determined between portals in stage 2 PVS generation. Each linkage is shown by a line with an arrow on the end which indicates the direction of the visibility. It is not recommend that this button be used for complex maps as they will generate too many linkages and slow down 3Dsmx to much to be useable.

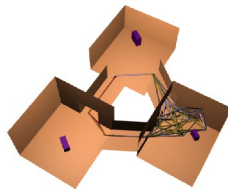


Figure 114. Basic Vis Linkage

### **Vis Linkage**

As Figure 114 illustrates, the vis linkage button will show the visibility between leaves determined in the final stage (stage 3) of PVS generation. Note that the arrows point out from the centre of the leaf's BB to the leaf that is visible from that leaf. It is not recommend that this button be used for complex maps as they will generate too many linkages and slow down 3Dsmx to much to be useable.

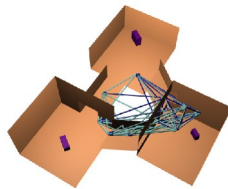


Figure 115. Vis Linkage

### **Anti-penumbra**

As Figure 116 illustrates, the anti-penumbra button allows the user to select which leaf and destination they wish to see the anti-penumbra for. Note that if it is impossible to generate an anti-penumbra for the given leaves then none will be generated.

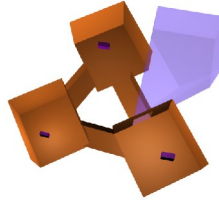


Figure 116. Anti-penumbra

On clicking the anti-penumbra button, Figure 117 shows the anti-penumbra dialog that will show up.

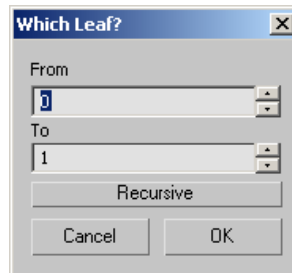


Figure 117. Anti-Penumbra Selection Dialog

#### *From*

From is the source leaf to generate the portal from.

#### *To*

To is the destination leaf to generate the portal from.

#### *Recursive*

Specifies whether subsequent leaves in the destination leaf are traversed, creating sub-anti-penumbras.

### **Default**

The default button will reset all the values to their defaults. Figure 118 shows an image of the default button.

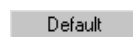


Figure 118. Default

## **20.2 The Engine**

A binary (Engine.exe) is provided on the CD however it can be useful to compile the engine.

### **20.2.1 Compiling**

Needed API's and programs to compile the engine:

- Digital mars D (originally developed by Walter Bright) compiler from <http://www.digitalmars.com/d/>
- Digital mars C compiler (originally developed by Walter Bright) from <http://www.digitalmars.com/>
- unDig (originally developed by Burton Radons) from <http://www.dsource.org>
- GLee (ported from Ben Woodhouse's GLee) from <http://elf-stone.com/>.

Optional tools to make things easier:

- DIDE is a D interface development tool. DIDE project file export has been provided so the engine can be compiled using DIDE.

These API's and programs have also been included on the CD with permission from the authors where relevant.

Once these files have been installed the engine can be compiled using this command:

```
C:\dmd\bin\dmd.exe -O -inline -release -c -IC:\dmd\include -IC:\dig -IC:\glee -I.-I.. -odC:\PROGRA~1\DIDE\Projects\Engine "[Code Location]\netjoelanderson\engine3D\bsptree.d" "[Code Location]\netjoelanderson\engine3D\bsptreecompact.d" "[Code Location]\netjoelanderson\engine3D\bsptreecompactlink.d" "[Code Location]\netjoelanderson\engine3D\bsptreelink.d" "[Code Location]\netjoelanderson\engine3D\common.d" "[Code Location]\netjoelanderson\engine3D\elements.d" "[Code Location]\Engine.d" "[Code Location]\netjoelanderson\common\investigator.d" "[Code Location]\netjoelanderson\engine3D\material.d" "[Code Location]\netjoelanderson\engine3D\occlusion.d" "[Code Location]\netjoelanderson\engine3D\polygon.d" "[Code Location]\netjoelanderson\common\queue.d" "[Code Location]\netjoelanderson\common\read.d" "[Code Location]\netjoelanderson\engine3D\texture.d" "[Code Location]\netjoelanderson\common\tools.d" "[Code Location]\netjoelanderson\engine3D\tools3D.d"
```

And linked with this command:

```
C:\dmd\bin\link.exe [Obj Code Location]\bsptree.obj+[Obj Code Location]\bsptreecompact.obj+[Obj Code Location]\bsptreecompactlink.obj+[Obj Code Location]\bsptreelink.obj+[Obj Code Location]\common.obj+[Obj Code Location]\elements.obj+[Obj Code Location]\Engine.obj+[Obj Code Location]\investigator.obj+[Obj Code Location]\material.obj+[Obj Code Location]\occlusion.obj+[Obj Code Location]\polygon.obj+[Obj Code Location]\queue.obj+[Obj Code Location]\read.obj+[Obj Code Location]\texture.obj+[Obj Code Location]\tools.obj+[Obj Code Location]\tools3D.obj,Engine.exe, GLee.lib+dig.lib+digglib+OPENGL32.LIB+ADVAPI32.LIB+SHELL32.LIB+GDI32.LIB+COMCTL32.LIB+noi;
```

Where [Code Location] is the location of the engine code and [Obj Code Location] is the location where the obj files were created which is probably the same folder as the engine code. It is assumed that dmd, dig, glee are all stored on the c drive.

For troubleshooting the best place for advice is the D newsgroup on [news.digitalmars.com](http://news.digitalmars.com).

## 20.2.2 Running

The engine should be run from the command prompt or from a bat file. The parameter format is:

Engine [mbs file] [record file] [log file] [-f] [-no]

- If the map (.mbs) file isn't specified then the maze.mbs file will be used, otherwise the one specified will be.
- If the log (.log) file isn't specified then that the log.txt file will be used to log information, otherwise the one specified will be used.
- If a record file (.rec) is specified then that file will be played.
- -f (full screen) will make the program start up in full screen mode
- -no (no occlusion) will make force the program to use the BSP trees that have been optimised **not** to use occlusion otherwise the occlusion trees will be used.

## 20.2.3 Control

### Movement controls

Key	Action
(↓) Down or S	Walk backwards
(↑) Up or W or Right Mouse Button	Walk forwards
(→) Left or A	Turn Left
(←) Right or D	Turn Right
Q or <	Strafe Left
E or >	Strafe Right
Space	Float Jump
Mouse Down + Drag	Turns/Pitches
Page Down	Jump (when gravity is on), move up when gravity is off.
Page Up	Down when gravity is off
Home	Look up
End	Look down
Insert	Roll Left
Delete	Roll Right
Left + Right Mouse Button + Drag.	In overview mode zooms in/ out.

### Toggle controls

Keys	Action
O	Over view mode, useful to see exactly what polygons are being culled.
Z	Allows free movement in overview mode so the camera can rotate around and keep the camera in the same place.
R	Renders everything without Hidden Surface Removal (except individual polygon occlusion).
X	Turns off hardware occlusion. Note that even with this off the BSP trees are still structured for hardware occlusion so they will not run as fast as the engine started with the –no command.
G	Disables gravity so the movement switches to fly mode.
C	Disables collision detection. With gravity on the camera will fall through the floor.
N	Goes to the next camera
F	View Frustum
V	Changes background colour to white which is useful for printouts.

### Bounding Boxes

Keys	Action
B	Shows the trees bounding boxes.
L	Shows the leaf bounding boxes.
U	Shows the details bounding boxes.

## Recording

Keys	Action
F1	Starts recording the cameras movement as a path. Press F1 again to save and stop recording.
F2	Replays the last recorded camera movement. Press F2 again to stop playing.
F3	Loads a camera movement from a file and starts playing it.

## Stats

Leaf	Action
1	Toggles Frames per Second. In this mode FPS are computed at there real time, that is frames are rendered even if the scene doesn't change.
2	Writes out the number of polygons
3	Writes out information to a log file and displays the average FPS.

## Other

Leaf	Action
Escape	Exits the program

## 21 Appendix 8 – Definition of terms

The following definitions are the main constructs used in building most HSR algorithms:

### 21.1.1 Vector

In the context of this document, a vector  $A$  is any variable that is made up of three components  $(x, y, z)$ . If not otherwise stated, a vector may also be indicated by the  $_{xyz}$  suffix as in  $A_{xyz}$ , indicating the variable  $A$  is a vector. Individual components  $(x, y, z)$  of a vector are specified by using that component as the suffix; as in  $A_x$  meaning the  $x$  component of vector  $A$ .

### 21.1.2 Dot Product

Let  $a_{xyz} \cdot b_{xyz}$  represent the dot product with  $a_{xyz} \cdot b_{xyz} = a_x * b_x + a_y * b_y + a_z * b_z$  where  $x, y$  and  $z$  are the coordinates and  $a$  and  $b$  are vectors.

### 21.1.3 Surface Normal

A surface normal is a vector that indicates the direction the surface (polygon) is facing.

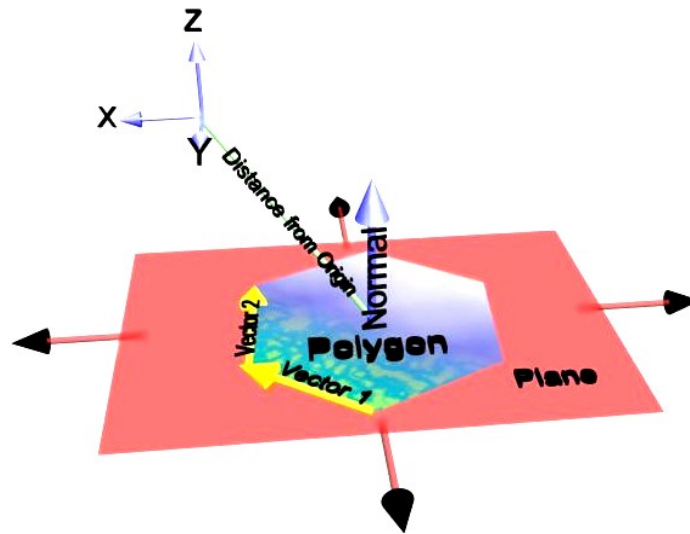


Figure 119. Polygon's Plane and Normal

The normal for a surface (polygon), as shown in Figure 119, is determined by computing the cross product  $(a_{xyz} \times b_{xyz})$  from two consecutive edges, of the surface,  $i_{xyz}$  and  $j_{xyz}$ , and then normalizing the result between  $\pm 1$ .

The cross product computation of the vectors:

$$a_x = i_y j_z - i_z j_y$$

$$a_y = i_z j_x - i_x j_z$$

$$a_z = i_x j_y - i_y j_x$$

Normalization is performed by dividing the result  $a_{xyz}$  by the magnitude  $|a|$  of the vector  $a$  as in:

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

$$b_x = a_x / |a|$$

$$b_y = a_y / |a|$$

$$b_z = a_z / |a|$$

Where  $b_{xyz}$  is the normal for the surface  $i_{xyz}$  and  $j_{xyz}$ .

### 21.1.4 The plane

The 3D plane  $E_{xyzd}$  is represented by four coefficients  $A$ ,  $B$ ,  $C$  and  $D$ , and 3 coordinates  $(x, y, z)$  such that the plane equation  $A_x + B_y + C_z + D = 0$  holds true. Therefore,  $E$  is made up of a vector  $E_{xyz}$  and a constant  $E_d$ . Components  $ABC$  represent the normal vector of the plane.  $D$  is the distance (magnitude) from the origin. Any point  $R_{xyz}$  can be determined to be on a plane if  $R \cdot E_{xyz} + E_d = 0$  holds true. A point  $R_{xyz}$  can be determined to be in front or behind a surface if the plane equation  $R \cdot E_{xyz} + E_d < 0$  or  $R \cdot E_{xyz} + E_d > 0$  holds true, respectfully.

An example of a plane is shown in Figure 119. Note that the plane's surface is infinite, which is represented by the arrows.

Source code for computing the plane can be found at 17.4.



### 21.1.5 The Z-buffer

Z-buffers provide a way to draw polygons so they appear priority-ordered to the viewer without the need for sorting them. Moreover, they do not suffer from crossover problems. A Z-buffer works by providing a 2D array (buffer) of depth (Z) values. At the beginning of a frame, the Z-buffer is reset to the furthest Z-value possible. As each pixel of a polygon is drawn, it tests its Z value against the Z-buffer value at that pixel location. If that pixel's Z value is closer than the Z-buffers pixel value at that position, it draws that pixel to the screen buffer and updates the Z-buffer with the new pixels Z value. Otherwise, it does not update the Z-buffer or draw the pixel. Depth values are calculated by interpolating between the depth values at the corners (vertices) of the polygon. Most Z-buffering today is done in hardware, parallel to texturing (after clipping), so it requires no extra processing time.

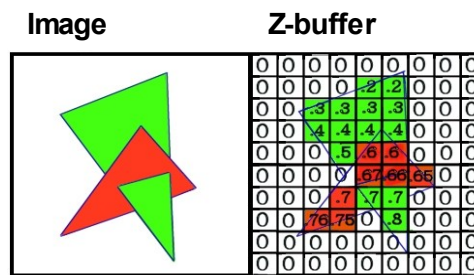


Figure 120. Z-buffer

Figure 120 shows how a polygon may be interpreted into the Z-buffer on the screen. The image is a diagram of the polygons that are to be rendered. The Z-buffer image shows the outcome of Z-buffering that polygon. Note that the Z-buffer in this example is low resolution so the effect can be seen. 0 represents the furthest value and 1 the closest. This diagram also demonstrates how the crossover problem is handled with a Z-buffer.

Z-buffers can be quite speedy when compared to a sorting algorithm, as they only require an extra calculation per polygon pixel. It is hard to compare a Z-buffer to a sort algorithm as they come into effect at different points in the rendering pipeline. While sort algorithms deal with an entire polygon Z-buffers deal with each individual pixel drawn to the screen. Z-buffers generally outperform depth sorts when there are a lot of polygons.

Things that come after Z-buffering in the pipeline such as lighting (which is notoriously slow) can be ignored if the pixel isn't visible. However, there are situations where Z-buffers do not perform well. Z-buffers perform worst when the scene is already depth sorted (closest item last). In these cases there will never be a pixel discarded and performance will be worse than painters' algorithm.

Like Z-buffers, BSP trees also provide a sort-free method of priority-ordering polygons. Many versions of BSP trees trade off the ability to PPO for efficiency and use Z-buffers as a replacement for PPO. With today's 3D hardware Z-buffers generally are done in parallel with the rest of the hardware pipeline incurring virtually no cost at all. There can sometimes be a small improvement using BSP trees to sort polygons in backwards priority-order to removing overdraw, however in many cases the extra CPU time for the BSP tree PPO is not worth it.

### 21.1.6 Anti-penumbra

A penumbra means the shape caused by a shadow, therefore an anti-penumbra (anti-shadow) creates the shape light has on the shadow.

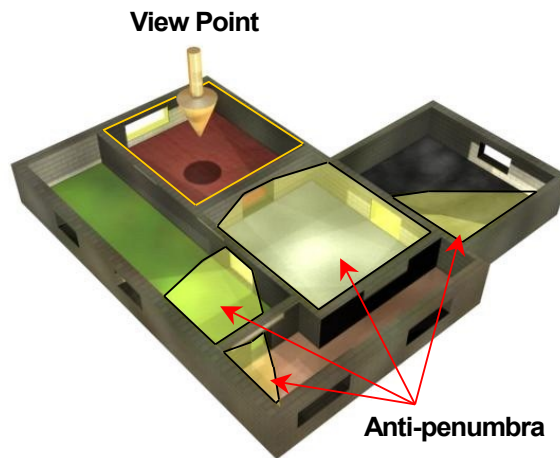


Figure 121. Anti-penumbra

In Figure 121 the arrow represents the light source/view point. Notice how the light spreads out as it passes further through the rooms.

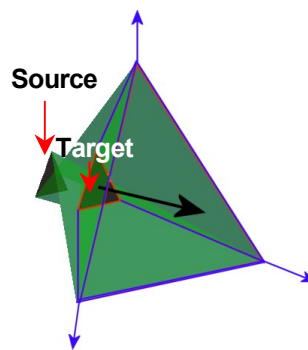


Figure 122. 3D Anti-penumbra

illustrates a 3D anti-penumbra made out of triangular source and target portals. The source portal is the portal that the viewer is looking through and the target portal is one that can be seen through the source portal. Details on anti-penumbra construction and culling can be found in Teller and Séquin (1991) & Teller (1992a; 1992b).

### 21.1.7 Convex Hulls

A convex hull (also known as a convex volume) is an area (volume) where any point within that area can see every other point, without being obstructed by any surface that surrounds the volume.

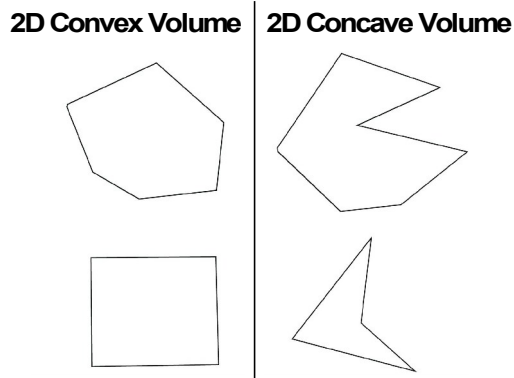


Figure 123. 2D Convex/Concave Volumes

shows the difference between convex and concave volumes. Although the example is shown in 2D, the concept is analogous for 3D.

### 21.1.8 Binary Space Partition (BSP) trees

The standard BSP tree approach is to divide a group of polygons into two sub-trees, recursively, using a partitioning plane. A polygon's plane is normally selected at each level to build a binary tree with a polygon at each node. Note that a polygon determined to be on the plane can be placed on either side. A polygon that is between a plane needs to be split down that plane into two polygons (nodes). At runtime on each node, the side of the partition that faces the camera is visited first in a recursive traversal of the tree, resulting in a depth ordered list of polygons.

A BSP tree divides the scene up into convex spaces by partitioning planes into a binary tree. The steps to creating a BSP tree are:

1. Choose a partitioning plane.
2. Place all polygons that are on the back side of the partitioning plane on the back (left) node, and all those on the front side of the plane on the front (right) node, of the tree. Polygons that are on both sides are split into two along the partitioning plane and placed on their respective sides.
3. For each sub tree repeat step 1, until there are no polygons left to use as partitions.

BSP trees are traditionally used to speed up sorting. To sort back to front use these steps

2. Start at the root.
3. Traverse down the side that is facing away from the camera first, and then traverse the other side.

The BSP tree can successfully be used to sort a scene in  $O(n)$  time, where  $n$  is the number of polygons (including polygons that are split in two) in the scene. However, all objects within these trees must remain static for this approach to work. In 1983, Fuchs, Abram, & Grant explored dynamic objects within BSP trees. The premise was that most dynamic objects are likely to stay within the same ancestral convex hulls in the tree and, therefore, the tree would not be required to be rebuilt, often. Each object in the tree can itself be a BSP tree with child objects that are also BSP trees. If an object crosses outside of a convex hull, then its ancestor trees, and perhaps their parent sub trees, would need to be re-built, recursively.

### 21.1.9 Portals Culling

In portal culling (Bikker, n.d.) each sector is connected to neighbouring sectors by portals (for example windows or doors) as shown in Figure 124. “The rationale for all portal-culling algorithms is that walls often act as large occluders in such cases.” (Moller and Haines, 1999, p. 200). The advantage of portals is that they provide an efficient and easy technique for hidden surface removal of dynamic scenes.

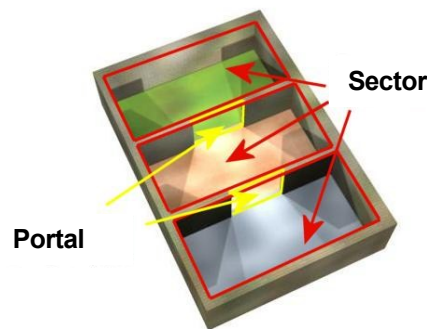


Figure 124. Portals and Sectors

It is possible to have one-way portals such as doors that close shut behind a player. Portals can be used for special effects such as,

- doorways between different worlds that do not need to be logically adjacent to each other
- mirrors (for instance a portal could lead into itself)

A major problem with portals is in determining where to place them. One approach is to leave it to the artists to decide, including portals in the door and window primitives in the editor. Another solution is to use BSP trees to auto-generate portals.

The initial step in the BSP tree technique for portal determination (which is detailed in “BB BSP trees”) is to clip large polygons along potential planes by the BSP tree (which holds the entire world). A particular polygon (portal being created) is sent down the BSP tree and is trimmed by all the planes it intersects. After being culled, smaller polygons that perfectly fits the gaps (between the convex sectors) results, known as portals. An advantage of using BSP trees for portal generation is that all polygons are guaranteed to be convex. The problem with using BSP trees is that they may not always pick the best places for portals, and generally create more portals than is necessary.

*For more information on portal culling see 16.1.*

### 21.1.10 Quadtrees / Octrees

Octrees and quadtrees as explained by Nuydens (2000) divides the scene into a hierarchy of parallel boxes are a reasonable solution for hidden surface removal (HSR) in indoor and outdoor scenes. Quadtrees use 2D rectangles for the boxes. Octrees are the 3D form of quadtrees using bounding cubes (boxes) instead. Octrees provide a quick way to determine what polygons are contained in the view frustum. Octrees also provide a means to cull away objects that are hidden behind other objects (occlusion culling). This occlusion culling provides a mechanism for making polygons appear correctly depth priority-ordered although octrees do not naturally provide this capability.

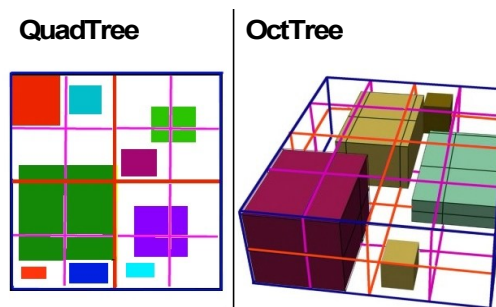


Figure 125. Quadtrees and Octrees

Figure 125 show the difference between quadtrees and octrees. Notice how each box can contain sub boxes. The black lines the octree show how polygons are divided into sub polygons (if needed) into the quadtree. Some of the boxes do not need to be divided because they fit into the quad.

Octrees (quadrees) are a logical step to BSP trees. In fact, an octree (quadtree) can be stored in a BSP tree. “BSP trees are more general than quadrees and Octrees because there is no constraint on the orientation of the planes.” (Eberly, 2000) The 2D (3D) scene is divided horizontally, vertically (and though the middle for octrees), to create four (eight) sub cubes in the BSP tree. Therefore 2 (3) nodes represent one node in a conventional quadtree (octree). A partition stops when there are no polygons on the other side of that partition. This process enables BSP octrees to have more optimal storage and smaller depth. Standard BSP trees, however, use the polygon’s plane to partition polygons, which makes for an even more optimal tree than most adaptive octree, for indoor scenes.

*For more information on Quadrees/Octrees see 16.2.*

### 21.1.11 KD-trees

KD-trees (Csabai, n.d.; Kmett, 1999b) are balanced multidimensional binary trees that divide space recursively into two halfspaces along an axis. BSP trees are sometimes known as enhanced KD-trees. KD-trees can be more optimal than an octree or a quadtree by allowing data volumes to be more balanced (Csabai, n.d.; Kmett, 1999b). The HSR algorithm is similar to the octree method so it won’t be discussed here. Figure 126 shows an example of a 2D and 3D KD-tree.

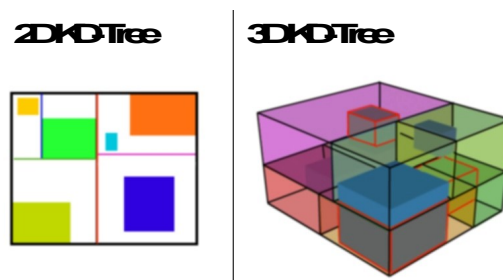


Figure 126. KD-trees

*Information of KD-trees such as how they can be used for occlusion culling, constructive solid geometry, hidden surface removal and collision detection can be found in .*

### 21.1.12 Problems solved by BSP trees

BSP trees are used to solve many algorithmic problems in interactive 3D worlds namely: polygon priority-ordering, hidden Surface removal, collision detection and constructive solid geometry. In many cases these problems are interrelated, for example:

- Most first person shooter games need fast HSR and CD as both HSR and CD.
- Some HSR algorithms, for example the one used in this thesis, require CSG to build solid worlds.

More information on how algorithms such as portals, octrees and KD-trees provide solutions to the problems mentioned below see “Appendix 3”.

## Polygon Priority Ordering

In the past, one of the major bottlenecks of 3D renders was polygon priority-ordering (polygon depth ordering). Polygon depth ordering is used to make sure objects that are closer appear to be in front of objects that are further away, such as the case below:

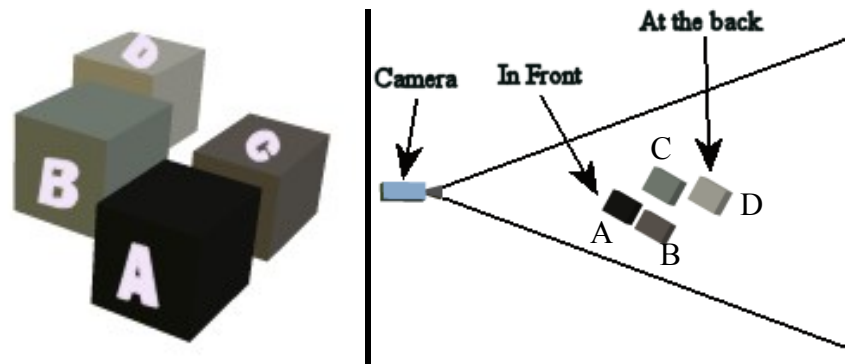


Figure 127. Polygon Depth Ordering

In Figure 127, d is the furthest box away, followed by b then c then a. The lines coming from the camera show what the camera can see known as the cameras field of view (FOV). Using a traditional sort (for example quick sort, merge sort, binary tree sorting, AVL sort) to priority-order these polygons would result in the cost of  $O(n \log(n))$  time spent sorting polygons (surfaces in the object); where  $n$  is number of polygons. Using traditional sorting to order polygons can result in some polygons being placed in the wrong order, such as in “the crossover problem”. However, provided the objects remain static or don’t move much, BSP trees only require  $O(n)$  to order the scene and don’t suffer from “the crossover problem”.

### ***The crossover problem***

One problem with polygon priority-ordering is that it is sometimes difficult to work out which polygon to draw first, such as in the crossover problem shown below:

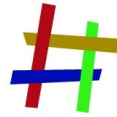


Figure 128. The Crossover Problem

One solution is to test each pixel as it is being drawn using a Z-buffer (see 21.1.5). Another solution is to divide some of the offending polygons into two pieces; as in:

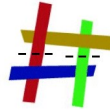


Figure 129. Fixing Crossover by splitting

At the cost of extra polygons Figure 129 shows how the crossover problem may be fixed using division, for the particular problem in Figure 128. These polygons can now be depth ordered without the crossover problem.

A problem with the division solution is that in general cases it is difficult and computationally expensive to determine which polygons to split. Sébastien Loisel (1996, pp. 77) discusses some more solutions to the crossover problem. BSP trees provide a reasonably effective solution for the crossover problem.

### ***Using BSP trees for Polygon Priority Ordering***

Once compiled, BSP trees simply rely on the direction of the plane to determine priority-order. The crossover problem is solved because standard BSP trees require nodes that exist over two planes to be split along that plane. Therefore, it is impossible for a static (non-moving) plane to exist both behind and in front at the same time.

“When the BSP tree is complete, we process the tree by selecting the surfaces for display in the priority-order back to front, so that foreground objects are painted over the background objects.” (Hearn and Baker, 1986, pp. 482).

A BSP tree requires  $O(n)$  to get priority-order because each node in the tree is only visited once; where  $n$  is the amount of polygons (nodes). Furthermore, BSP trees do not need to perform expensive distance calculations to work out polygon depth order.



One disadvantage of BSP trees over other sorting methods is that the updating algorithm, used when objects move, is expensive computationally (Foley, Dam, Feiner, Hughes, and Phillips, 1994, pp. 197). “The BSP tree is particularly useful when the view reference point changes, but the object in a scene are at fixed positions.” (Hearn and Baker, 1986, pp. 481) However, dynamic objects can still take advantage of some aspects of the BSP tree by being placed into the tree’s leaves instead of at the nodes. Therefore, BSP are a viable option for real-time applications that require a small amount of dynamic components in comparison to static ones, namely first person shooter games.

## Hidden Surface Removal (HSR)

Imagine a painter drawing a scene from the inside of an intergalactic space ship. First the painter draws a space scene full of stars and cosmic clouds. Then, he paints over the entire scene, the colour grey, as the spacecraft he is in has no windows. This technique, known as “painters algorithm” highlights a rationale for HSR algorithms. Human convention would imply simply to draw the grey wall, however the painters algorithm has no idea about occlusion (which object hides another). Furthermore, as Eberly (2000, p. 425) stated “Determining exactly what is visible dynamically is usually an expensive process. Most systems attempt to get an approximation and minimize the number of objects that are sent to the renderer but are unknowingly invisible.” Thus, engines that do not employ HSR draw everything, whether visible or invisible.

Hidden surface removal is a problem faced by many 3D software developers. Why spend time rendering what cannot be seen? Polygons can either be too small, too far away, hidden by other polygons or not within the FOV. The solution to this problem comes in many forms but generally it involves removing large clusters of polygons while minimising tests (selection statements). Generally these strategies sacrifice RAM (which is becoming larger and cheaper all the time) for increased CPU efficiency.

HSR algorithms often process many more polygons than the optimal amount or spend too much time finding for polygons to remove. With today’s 3D hardware it is generally faster to let the 3D card focus on removing individual or small groups of polygons that are out of view and let the CPU focus on the bigger picture, removing collections of polygons. The key is to provide healthy balance between number of polygons being removed from the pipeline and the amount of tests required at each stage. Often a hybrid solution tailored to suit the particular application is the best choice for HSR.

Types of hidden polygons that can be removed by HSR are:

- polygons that are facing away from the viewer are removed using back face culling.
- polygons that aren’t within the viewer’s vision are removed using field of view culling or view frustum culling.
- polygons that are obstructed by other polygons are known as occlusion.

### **Back face culling**

In general, back face culling reduces around half of all polygons in the scene. In discussing back face culling, Moller and Haines (1999, pp. 192) stated, "All back-facing polygons that are part of an opaque object can be culled away from further processing." Consider a cube such as made of six polygons. If the camera is inside the cube, it will never see the outside part of the cube. If the camera is outside the cube, it will never see the inside. Furthermore, only as much as three sides of the cube can ever be seen at one time.

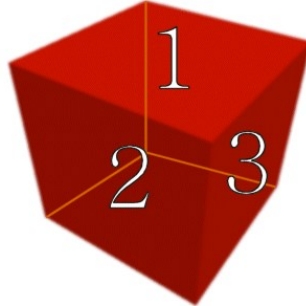


Figure 130. Back faced cube

Back facing works by considering polygons as one sided. A polygon's face direction can be worked out using the dot product of the polygon's plane with the camera's position vector:

Calculate Back-face: Dot product of a plane and the camera:

$$Ax + By + Cz + D = 0$$

Therefore for  $i$  plane it can be said,

$$D_i = (CP_i) \bullet N_i$$

For a 3D vector list that means:

$$D_i = (C_x P_{ix})N_{ix} + (C_y P_{iy})N_{iy} + (C_z P_{iz})N_{iz}$$

Where,

Let  $D$  be the resulting direction polygon  $i$  is facing (ranges from 1.0 to -1.0).

Let  $C$  be the camera, which is a 3D vector  $(x, y, z)$ .

Let  $P$  be one vertex on the plane (i.e. the first vertex), which is a 3D vector  $(x, y, z)$ .

Let  $N$  be the plane normal which is a 3D vector  $(x, y, z)$ .

Let  $i$  be the plane's (polygon's) index number.

Let  $\bullet$  be dot product

Listing 110. Back face culling formula

In back-face calculation, the plane (polygon) is moved so that the camera becomes the origin. The dot product is applied to the polygon's plane, which results in a real number between 1.0 and -1.0. The direction of the sign determines which way, the polygon is facing. If the result is zero then the polygon is directly side on to the camera, and displaying this face is optional. Winding determines which polygon side is which. Switching the face direction around simply involves reversing the polygon's vertex direction (the winding direction).

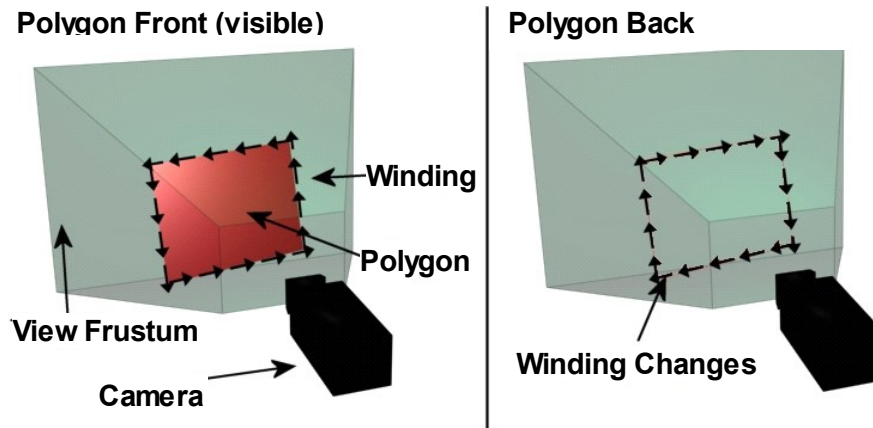


Figure 131. Polygon Winding

Figure 131 shows what happens when a polygon's winding is changed in back face culling. The arrows represent the direction of the winding. When the winding is switched around the polygon is no longer visible to the camera.

Back face culling is considered a low-level and simple form of polygon reduction because it is based on individual polygon attributes and is reasonably simple to determine. Being low-level means that back face culling is often the last step in the graphics pipeline stage before polygon clipping. Therefore, it is often built into hardware as a first stage of the clipping process.

In BSP trees, back-facing is performed at each node in the tree. Therefore back face culling can be performed with no additional coding instructions.

### ***Field of View Culling***

Objects that sit behind the camera obviously cannot be seen, and therefore shouldn't need to be sent down the rendering pipeline. The FOV is used to remove objects that are not within the camera's vision area. Generally a view frustum made up of six planes (left, right, top, bottom, near, far) is used to cull away objects that are definitely not visible to the camera.

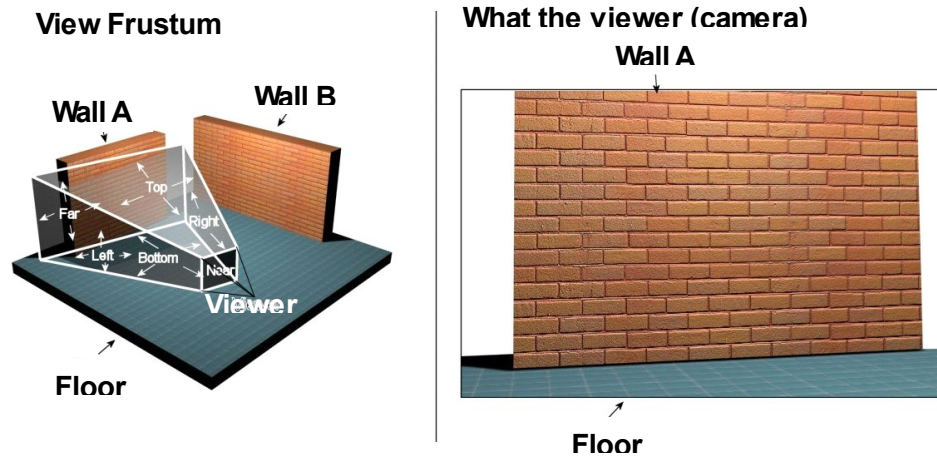


Figure 132. The View Frustum

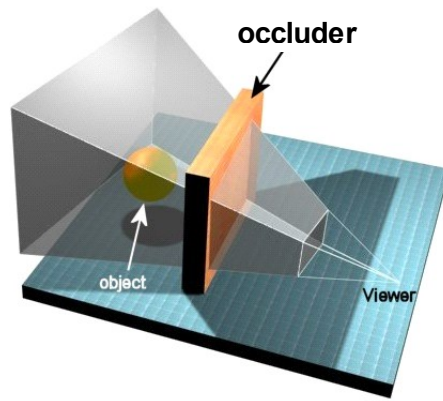
Figure 132 gives an example a view frustum in a particular scene and what the camera can see. The white arrows are used to indicate which plane is which, for the six planes. In this diagram, the viewer can only see wall A and the floor. Therefore rather than try to draw B which is not visible to the viewer B should be culled.

Instead of removing one object at a time, objects can be clustered together and be removed in groups. Quadrees, KD-trees and octrees are specifically structured into clusters of clusters of objects to take advantage of culling objects by groups. BSP trees can be modified to also gain this ability, namely using Bounding Volumes (see “BB BSP trees”).

### **Occlusion**

Occlusion occurs in a scene when one object prevents another or part of another object from being visible to the camera. Occlusion detection can be one of the most computational demanding types of HSR. At the time of writing (2004) the manufactures of consumer 3D hardware have begun to bring occlusion support into the consumer market for 3D hardware, namely the Geforce 4 and the Radeon 9700 video cards.

### The View Frustum



### What the viewer (camera) sees

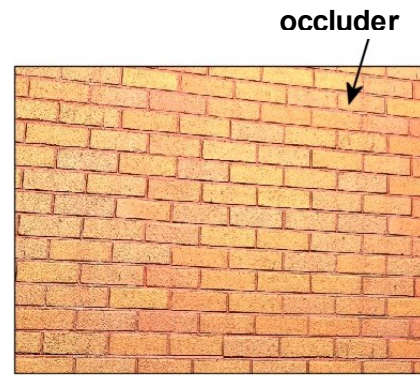


Figure 133. Occlusion

Figure 133 shows the effect of occlusion on an object. The object (the sphere) is behind the occluder (the wall) and so it does not need to be drawn. The way BSP trees curve up space into convex volumes and sort the scene make them ideal in optimising occlusion culling. Some ways BSP trees can help with occlusion culling are by:

- traversing the tree front to back (the reverse of painters algorithm) and testing bounding volumes at each tree node for occlusion from other polygons that have already been rendered (see chapter 9).
- creating portals between sectors in the world and only rendering the portals that can be seen from the camera's sector (see 16.1.1)
- determining PVS at the leaves of the tree eliminating objects that are unquestionably occulted (see "Potentially Visible Sets (PVS)")

This thesis investigates combining 1 and 3 to produce a hybrid advantage.

## Collision Detection (CD)

Collision detection (CD) is a process of detecting when a collision (is going to) occurs and which objects are involved. Many applications such as games require collision detection to prevent players walking through objects such as walls. CD involves informing the program of collisions, which then results in a collision reaction of some form. Collision reaction will not be discussed. For scenes with small amounts of objects, simple CD (comparing  $n$  objects against  $n$  objects) proves effective. However, for larger scenes BSP trees provide a more efficient means for collision detection.

It is important to note that on most consumer computer systems there is no hardware collision support. Therefore this task is still primarily a CPU task. Nonetheless, in the general case, collision detection is much faster than HSR because the amount of moving objects needing to be tested is less than the amount of objects that need to be found within the FOV.

There are several libraries/algorithms such as RAPID, I-Collide, and V-Collide Solid, Deep, Swift, Swift++, H-Collide, Impact, Pivot that handle collision detection reasonably efficiently. A description of these libraries/algorithms can be found at <http://www.cs.unc.edu/~geom/DEEP/>. However, in many cases it is a good idea to combine the HSR and the collision detection algorithm together due to efficiency considerations.

### ***Simple CD***

In a simple collision detection algorithm each object is tested against the other, requiring  $O(N^2)$ ; where  $n$  is the amount of objects (sometimes polygons). Furthermore, if once a collision is found (and due to collision responses applied), the collision algorithm needs to be repeated until no more relevant collisions are found. Simple collision detection may be unacceptable when considering the amount of objects in a game level (or scene) can be millions. Some techniques applied to reduce the amount of tests required are as follows:

#### *Static and Dynamic Objects*

Dynamic objects are tested against the other static and dynamic objects. Static objects do not need to be tested against other static objects because their collision status remains constant. Furthermore, usually only a small proportion of a scene is dynamic at any one time, objects being static most of the time. Using dynamic objects in with simple collision detection results in  $O(d(s+d))$  tests; where  $d$  represents the dynamic objects and  $s$  represents the static objects.

#### *Object simplification*

Objects can be grouped into larger simplified objects and tested in groups, often referred to as simplification. If the group is rejected in a CD then all the polygons in the group can also be rejected. Conversely if they are found to have collided then sub-objects of that group can be further tested to determine which (if) any individual objects have collided. For example if there is a large room with one hundred human models then the humans would be simplified into bounding boxes. Instead of testing the million x million polygon in the scene, only 100x100 tests need to be performed to determine which (if) human a human collision has occurred.

This grouping technique can be improved by placing object groups into a hierarchical tree structure. Each node in the tree holds a group of objects, which are sub-divided into smaller groups, which are stored in that node's children. For instance, taking that human room scene again, after two humans have been determined to collide, those humans may be sub-divided into body parts (i.e. legs, head, arms and torso) and tested again in more detail. If a foot was determined to have collided then it could be subdivided into smaller groups yet again to determine which part of the foot (ankle or toe) was involved in the collision.

With the hierarchical approach of polygon grouping it can be difficult to determine optimal object groups, that are not too large and do not overlap. For example if a simplified group of objects (a volume) happened to contain most of the world then the camera would generally be found within that volume and it wouldn't be worth performing that test. If groups overlap too much then there is a greater likelihood of re-testing the same areas, increasing the amount of hits (the amount of groups found to collide). Portals (16.1), quadrees/octrees (16.2), KD-trees ( ) and BSP trees provide ways to divide up the scene's objects in a reasonably balanced way, without overlaps, which is generally close to optimal.

### ***Using BSP trees with CD***

BSP trees provide ways of making collision detection more efficient including the use of bounding volumes (BV) or solid trees.

BSP trees with BV partition a scene into small sectors into a form of hierarchical tree. BV's are attached to each node in the tree. Each BV fits around the node it is placed at including all that node's children. Bounding volumes have maximum search time of  $O(n)$  and a minimum of  $O(1)$  ; where  $n$  is the number of polygons in the collision set.

Solid BSP trees can generally outperform BSP trees with BV for collision detection. Solid BSP trees are made up of solid (left) and empty (right) leaves. When the camera ends up in a solid leaf, it is within a wall (which is impossible in real life). When an object ends up in an empty leaf, there is no collision. Therefore, it only takes  $O(\log(n))$  to determine a collision in a solid tree; where  $n$  is the number of polygons in the collision set. Note that this result does not include the fact that solid trees require extra nodes to fill gaps in the tree. For implementation details on solid trees see 4.3.2.

BSP trees are most efficient when the majority of the scene is static. Dynamic objects still must be tested against one another in a separate algorithm. Nevertheless, dynamic objects can be placed in the leaves of trees to take on some of its advantages. Objects that are not within the same leaf cannot be in collision. The insertion of a dynamic object into a tree takes  $O(\log(n))$  ; where  $n$  is the amount of polygons in the tree. However, it can be placed into the tree at the same time as collision detection which requires no extra processing. The dynamic object CD algorithm is only run for an object when its position in the tree changes (it moves). When an object is moved to a new leaf it is then tested against any dynamic objects contained in that leaf.

## **Constructive solid geometry (CSG)**

### ***What is CSG?***

Constructive solid geometry (CSG) is solid geometry that is used to create other solid objects. A solid object is an object that does not have polygonal holes in it (as shown in Figure 134). To illustrate this concept, imagine the object was physical, and filled with gas. In a solid object the gas would not be able to escape.

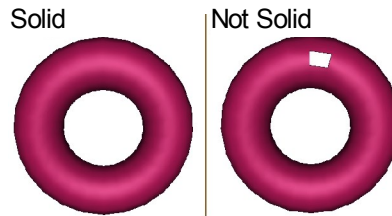


Figure 134. Solid/Not Solid

Solid objects normals all point away from the solid mass they encapsulate as shown by .

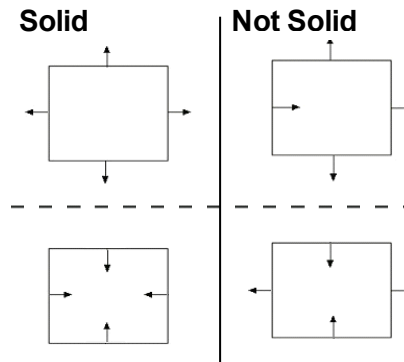


Figure 135. Direction of normals in solids

shows the differences between solid and non-solid (not solid) objects. The arrows in the diagram represent the normals of the polygons, which is the direction the polygon (surface) is facing. The sides of the polygon facing away from the arrow are invisible.

Constructive Solid Geometry involves the use of Boolean operation (union, subtraction and intersection) to create new objects. CSG objects begin by using some base CSG objects such as boxes, spheres, prisms, pyramids and cones (all simple solid shapes). Operations are applied to two objects to create one new object for example:

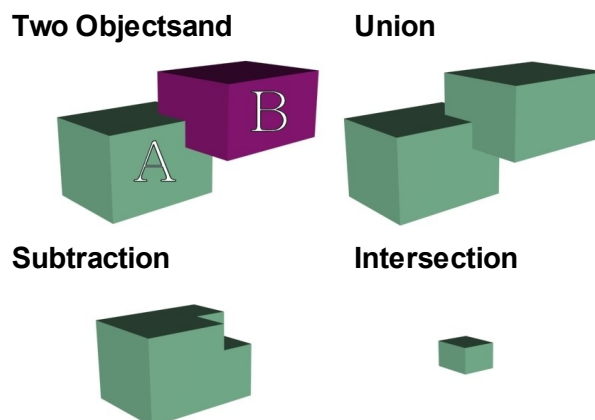


Figure 136. CSG Operations



Figure 136 shows the results of different Boolean actions on the object A. Notice that the resulting objects are always solid. In the above situation A is called the target object and B is called the modifier object. Subtraction is the only non-commutative operation of the three types of operations. Therefore, the target and modifier object can be switched for union and intersection with no change to output.

Solid BSP trees require a solid scene as input. Therefore, CSG is an ideal tool for solid BSP map editors as they are guaranteed to produce solid objects. The map editor Worldcraft is a prime example of this where CSG (with BSP trees) is used for the Quake BSP engine. Worldcraft only uses solid primitives to build the world and everything is done by CSG. A disadvantage of CSG is that it makes it difficult (not impossible) to model curved shapes. Quake 3 does allow for curved shapes but they are mainly available in the form of arcs. Without CSG, keeping a world solid can be an extremely difficult process.

The first step to applying a Boolean operation on two objects is to work out which surfaces (polygons) are inside and which are outside. There are several means of determining if a side is inside or outside namely using:

- Ray Casting;
- Solid BSP trees;
- Octrees or KD-trees (see 16.2.5).

Once all objects polygon/plane have been determined as inside or outside, it is simply a matter of applying the Boolean operation. To create an intersection between A and B simply take all the inside planes. To create a union, take all the outside planes. To subtract B from A, remove all B's outside planes and all A's inside planes, leaving behind a solid object. The determination whether a polygon inside/outside only needs to be performed once for static objects. Once inside/outside information is known, Boolean operations only require  $O(n)$  as it is a simple matter of checking each flag on each surface ( $n$ ) to determine whether to keep it or not.

#### *Using Ray Casting for CSG*

The initial step in the ray casting method is to find the point of intersection (if any). The intersections will form new surfaces within the object. To differentiate inside planes from the outside a ray (in the opposite direction to the planes normal) can be sent through the centre of the plane that is being tested. As the ray passes through planes (in priority-order), a count is incremented/decremented depending on whether the plane faces the ray or not, respectively. If the plane is found to have a count of 2 then it is deemed inside, otherwise outside. Not including the extra cost of ray casting, this algorithm normally requires  $O(n^2)$ ; where  $n$  is the number of polygons involved.

### *Using Solid BSP trees for CSG*

Solid BSP trees are particularly efficient at finding polygon intersections and also at determining inside/outside polygons. A solid tree (as described in 4.3.1) is created for both A and B. Object A is sent down object B's tree and Object B is sent down object A's tree. As the polygons are sent down the BSP trees they are automatically sliced up into fragments, as is the case with most BSP trees. When a plane reaches a solid leaf, it is deemed inside and is marked as inside, otherwise it is marked as outside. The fragments of A and B that have passed through the tree are then used to determine the resulting new object. Generally this algorithm requires  $O(n \log(n))$  and does not require ray casting; where  $n$  is the number of polygons involved.

### **21.1.13 Hierarchical Volumes (for BSP trees)**

BSP trees priority-order polygons into convex volumes at each node. Convex volumes of polygons tend to be clustered together and can be exploited using a hierarchy of Bounding Volumes (BV) at each node in the tree.

In discussing BV view frustum testing to improve rendering times Moller and Haines (1999, pp. 195) stated:

“If the BV is outside the frustum, then the geometry it encloses can be omitted from rendering ... If the BV is fully inside the frustum, its contents must all be inside the frustum. Traversal continues, but no further frustum testing is needed for the rest of such a subtree.”

Entire nodes (and their children) can be pruned from the tree resulting in fewer nodes to process. Unless the entire scene is within the FOV, the amount of nodes that need to be processed is generally much less than  $O(n)$  but greater than  $O(\log(n))$ ; where  $n$  is the number of polygons (or nodes) in the tree.

Due to BSP trees dividing up space into convex volumes and being able to render the scene front to back, the hierarchy of BV of the tree can be used to perform occlusion. That is BV (generally BB) are visibly rendered to the screen and tested to see if they are obstructed by polygons that have already been drawn. If a BV is obstructed then all that BV's contents can't be seen, otherwise that node is traversed. This thesis investigates the advantages of using hardware occlusion with BSP trees. More details about the BB BSP tree algorithm are given in 4.2.

### **21.1.14 Potentially Visible Sets (PVS)**

A PVS is a list (set) of groups of surfaces (polygons) that are potentially visible from the current location of the camera. Any group of polygons that are definitely not visible from that location are not contained in the PVS. PVS are generally bound to convex sub-region (sector) in the world, because producing one for every location is virtually impossible, and costly in terms of memory. Furthermore, PVS does not take the viewer's orientation, such as the view frustum, into account.

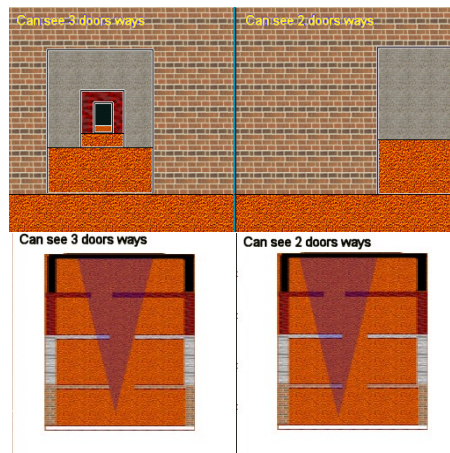


Figure 137. Estimate of what is visible

In Figure 137, the top diagram shows a room from the two camera points of view and the bottom shows that camera's view frustum. From one location in a room, the camera may be able to see though 3 doors ways however from another location, the camera may only be able to see though two (or one). In a PVS all 3 doorways would be accounted for even though only 2 can be seen some of the time. Some of the walls in the third room (furthest room) could be considered never visible from the current viewpoint therefore they would not be added to the PVS for the sector.

Several schemes have been adapted for PVS determination which are discussed by Teller et al. (1991) & Teller (1992a; 1992b). Determining a good PVS for each sector can be a costly process (Abrash, 1997, p. 1278), therefore, often it's only approximated. Thus, some polygons that are never visible from the viewpoint can be still included in the set, due to calculations being too expensive to eliminate these polygons.

## Using PVS with portals

Portals are the original inspiration for potentially visible sets (PVS). PVS can also be combined with algorithms such as octrees and BSP trees. PVS is a list of potentially visible polygons from a camera's position. Instead of traversing the indirectly linking polygons to determine if they can be seen, a PVS (actually two, one for each side) is placed at each local portal. This type of PVS contains all portals (or sectors) that are visible if looking through that portal from a local sector. However, some portals will still be in the PVS that are not actually visible from the camera position. Nevertheless this small error is generally considered negligible when compared to the amount of polygons PVS is able to discard.

As with most algorithms, it is a matter of determining the best performer for the target application. Portals with PVS require extra memory as each portal requires two potentially visible sets. Each PVS can potentially contain a link to all sectors (minus itself) on the map. A partial solution is to compress the PVS using an algorithm such as zero-run-length-compression; however, extra memory is still required. Sometimes portals can under perform more conventional methods. For example, portals cannot handle outdoor scenes well. In this situation, practically every sector can see every other sector meaning one portal per polygon. Portals can be efficiently combined with outdoor scenes for things such as caves, which are not always visible. Therefore, some attention should be given to analysing the nature of the problem before deciding on a particular algorithm such as portals.

## Using PVS with BSP trees

As each convex volume in a BSP tree cannot cover itself, they automatically become occluders. Furthermore, every leaf in a solid BSP tree represents a volume where a camera can exist anywhere within, without changing the list of potentially visible polygons.

This solid leaf property can be taken advantage of by the potentially visible sets (PVS) algorithm where a static list of visible polygons can be kept at each leaf. The location of the camera, and therefore the PVS can be found in  $O(\log(n))$  which is a huge saving when compared to  $O(n)$ ; where  $n$  represents the number of polygons (or nodes) in the tree. Then again, the PVS can be quite large (depending on the scene) and may result in a maximum of  $O(n)$  processing time.

## 22 Glossary

### **3D (Accelerated) Hardware**

3D hardware is the hardware that is specialized to handle 3D geometry algorithms, allowing for more realistic and faster 3D rendering. Most 3D hardware is part of the video card and motherboard of a computer.

### **Algorithm**

An algorithm is the method by which some programming task is performed. An algorithm provides instruction about how to go about programming of a particular task in the chosen language.

### **Application Programmers Interface (API)**

An API is a library of related functions that perform some task. In windows systems header files (.h) files with either a .DLL, .LIB, .c or .cpp source file are API's.

### **Axis Aligned Bounding Box (AABB)**

AABB are bounding boxes (BB) that sides are aligned to some global axis.

### **Bottleneck**

A bottleneck is a point in the program where the application spends most of its time processing. A program can be made considerably more efficient by optimising bottlenecks or removing them altogether. A program will always have bottlenecks because when a bottleneck is removed other parts of the program become new bottlenecks.

### **Bounding Boxes (BB)**

A Bounding box is the box (usually minimum) that a group of polygons will fit into. Bounding boxes can be axis-aligned (AABB) or object oriented (OBB) and represented in either 2D or 3D. Although oriented boxes are slightly more computationally expensive, they are generally considered better, because they can have a tighter fit. Generally, BB provides an excellent fit for the average polygon with minimal calculations.

See Bounding Volumes, Bounding Spheres, axis aligned bounding boxes and orientated bounding boxes.

### **Bounding Spheres (BS)**

A Bounding sphere is the sphere (usually minimum) that a group of polygons will fit into. A bounding circle, instead of a sphere can be used in 2D cases. Although BS is less computationally expensive than a BB, on average it has a looser fit. The worst case fit occurs with thin polygons. In exchange for computational efficiency, a sphere can be converted into an elliptic sphere, which allows tighter fittings.

See Bounding Volumes and Bounding Boxes.

**Bounding Volumes (BV)**

A bounding volume is an area that contains one or more objects. Any enclosed shape can act as a bounding volume. The most popular are BV's are spheres/circles and boxes. See Bounding Boxes and Bounding Spheres.

**Brushes**

Brushes are any geometric entity within the game level including boxes, staircases and rooms. Brushes are normally static. See detailed brushes, structural brushes.

**Camera**

A camera is the viewer of the visible area at particular location in a 3D scene. The camera normally has two properties: orientation and location.

**Clipping**

Clipping is a process of slicing up polygons in the scene and removing the pieces that are not wanted. For example, polygons that exceed the screen's boundaries need to be clipped to the screens boundary because that part cannot be seen.

**Collision Detection**

Collision detection is used to determine when two or more objects will collide. It is used in 3D room-based shoot-em-ups like Quake to help stop players going through walls. The collision detection does not actually prevent objects from going through one another. It simply informs the program to take an action known as collision reaction. It is generally Unrealistic to test every moving object in the scene against every other object. A particular scheme such as BSP trees, portals, quadrees or octrees can help reduce the amount of tests needed per object.

See Collision Reaction.

**Collision Reaction**

Collision reaction is the effect that happens to an object when collision occurs. Reaction comes in many forms. For example, the object may brake up, change colour, stop instantly, stick to the collision object, apply a force to collision object, bounce back or slide along the surface of the collision object. Collision Reaction enhances the realism of 3D graphic interaction.

See Collision Detection.

**Constructive Solid Geometry (CSG)**

CSG is base geometry such as a cube that is completely connected and is used to create objects that are more complex. Every edge in the geometry must belong to two (and only two) polygons. CSG uses Boolean operations to combine objects into other more complex solid objects. Quake 1, 2 and 3 calls CSG, brushes.

Central processing unit (CPU)

CPU is the central processing unit that computes and issues commands to the computer. The more computationally expensive an algorithm is the more time it'll take to process. In the context to this report, CPU means the impact on algorithm performance.

### **Depth-Buffer**

An array the same dimensions as the screen-buffer used to draw pixel in the priority-order according to the camera. As a polygon is drawn, each pixel is tested against the depth buffer at its current location. If it is closer to the camera (has a lower depth value) then it is drawn and the depth-value in the depth-buffer is updated with that pixel's depth. Otherwise the pixel is not drawn. This technique does not work for translucent surfaces. Another name of Depth-Buffer is Z-buffer.

### **Depth Sort(ed)**

Depth sort is used to sort polygons before they are rendered, so that they appear correctly on screen. Depth sort can be either back-to-front or front-to-back depending on the application. Nearly any type of sort can be used with a depth sort including bubble sort, insertion sort, KD-tree sort, merge sort, heap sort, selection sort, radix sort, bin sort, sequential sort, shell sort, quadratic selection sort, bucket sort and quick sort. Quick sort is the general choice in these situations as it performs reasonably most of the time.

Depth sort can be used with an algorithm like painter's algorithm, which need back-to-front priority-ordering. This sorting prevents objects that are far away overlapping objects that are near in the final render. Sometimes polygons exist both in front and behind one another at the same time. This special case can to be handled by the depth sort algorithm by simply splitting the polygon in two or switching to a Z-buffer.

See Painter's Algorithm.

### **Detailed Brushes**

Detailed brushes are brushes that are too detailed and too small to be considered for brake up into smaller groups. See brushes, structural brushes.

### **DirectX**

DirectX is Microsoft's competing product with OpenGL. Actually, direct3D is the competing product because DirectX does a lot of other stuff besides 3D. DirectX keeps changing so much that it is difficult to keep up with the terms. DirectX 7 was divided up into DirectPlay, DirectInput, DirectSound, DirectMusic, Direct3D and DirectDraw components. DirectX 8 has combined DirectDraw and Direct3D under the name DirectX Graphics. DirectSound and DirectMusic have also combined to create DirectX Audio. It also includes the new components Direct Show (original DirectMedia) and DirectX Application Manager.

### **Doom (1/2/3)**

Doom (1/2/3) is a well-known first person shooter game developed by the ID™ computer game software company. At the time of writing, Doom 3 is still under-development.

**Dynamic**

Something that is dynamic can change or moved around.

**Extensions**

Extensions are extras that are not included in the standard API. For example at time of writing vertex shaders are an extension and not part of OpenGL. Extensions are driver specific and therefore are not all the same on every platform. Extensions are normally written by third parties to help provide a better interface to their product.

**Field of View (FOV)**

The field of view is the area that fits into the cameras field of vision. Often a view frustum is used to represent a FOV. See View Frustum.

**Frame Number**

Frame number is the number of times the map has been redrawn. Generally frame number is used for calculating FPS and as a unique number for marking.

**Frames per Second (FPS)**

FPS is the amount of frames (on average) that can be displayed within a second.

**Frustum**

See View Frustum.

**Hidden Surface Removal (HSR)**

HSR is the removal of polygons that cannot be seen. Similar to "Polygon Removal Techniques" but does not include Simplification.

Here are some techniques that are used to efficiently remove hidden surfaces:

BSP trees

Portal Engines

Octrees/Quad trees

**Indoor**

An indoor scene is a world that exists mostly inside, such as a cave or a room. Indoor scenes provide good opportunities for occluders though the use of portals.

**Inside**

An object A is considered inside a plane if it is on the front (drawn) side of the plane. Therefore A is considered inside object B (for example a view frustum) if all B's planes face A, and does not split A.



**Leaf**

A leaf is a node on a tree data structure that has no child nodes.

**Node**

A node is an element in a tree or any graph structure that contains data and one or more pointers to other nodes (children). Nodes that do point other nodes (have children) are called branches. Nodes that point to no other nodes (have no children) are called leaves.

**Nondeterministic Polynomial (NP) - complete (completeness)**

A NP-complete problem is a problem where the time to solve it is nondeterministic and varies with the amount of input. Problems such as these are generally partially solved by guessing at the best answer and picking the best one in some fixed amount of time. For a more complete description of NP-completeness see "NP-Completeness, Cryptology, and Knapsacks" (Picciotto, 1995).

**Occlusion**

Occlusion is the effect of one 3D object covering up part of or all of another object. Sometimes programs are optimised by not rendering objects that are completely covered up.

**Octant**

An octant is a cube with eight neighbours. Octrees are made from octants, each octree dividing into eight more octants, until some threshold is met.

**Overdraw**

Overdraw occurs when more is rendered to the buffer than necessary. For example in Z-buffering if the scene is sorted back to front, then some pixels will be rendered several times to the same location in the buffer.

**Outdoor**

An outdoor scene is like a landscape scene where there are no portals, and many viewpoints can see most of the nodes on the scene.

**Outside**

An object A is considered outside a plane if it is on the back (not drawn) side of the plane. Therefore A is considered outside object B (for example a view frustum) if it is on the back of at least one of B's planes, and that plane doesn't split A.

**OpenGL**

OpenGL is 3D API that makes use of hardware to accelerate 3D programs. The manufactures of 3D cards write the OpenGL libraries (drivers) according to OpenGL specifications. This way all 3D cards have the same interface. OpenGL programmers only need to write one program that should work on all computers with OpenGL support.

OpenGL has the added advantage of being multi-platform. Multi-platform means that a program written for MS-Windows can be easily ported to Linux or Mac.

At present current OpenGL version is 1.4 however OpenGL 2.0 prototypes have been released.

### **Orientated Bounding Boxes (OBB)**

OBB are bounding boxes (BB) that need not be axis-aligned. Therefore they can be rotated in any direction. They can have a tighter fit around their contents than axis-aligned BB (AABB). OBB can have more than one best solution for a tight fit around an object.

### **Painter's Algorithm**

Each polygon is placed on top of the display regardless of other polygons Z-position (depth position). Painter's algorithm works similar to the way a painter draws to canvas (hence the name). A painter always draws on top of what they have already painted. To get polygons to appear in the proper priority-order, algorithms such as depth sort or BSP trees can be used to sort polygons into a back-to-front.

See BSP tree and Depth Sort.

### **Polygon Depth Ordering**

Polygon Depth Ordering is the sorting the polygons by depth so that the closest polygons appear to be closer than polygons at the further away.

### **Polygon Priority Ordering**

Polygon Priority Ordering is the sorting of polygons by some priority. An example of Polygon Priority Ordering is Polygon Depth Ordering.

### **Pre-calculation**

Pre-calculation is the process of determining the answer once so that it does not need to be continually re-computed many times. Generally the pre-computed results are stored in memory (RAM) for quick access.

### **Priority Order**

Priority order is the order by which polygons are sent down the rendering pipeline. The best order of polygons can depend on many things including the rendering attributes of each polygon and which polygon is closest to the camera.

### **Plug-in**

A plug-in, is a piece of software that extends the functionality of its target program. For example a 3D studio max plug-in allows the programmer to write (among other things) import and export plug-in, to export formats that were not originally programmed for 3D studio max.

**Quake (1/2/3)**

Quake (1/2/3) is a well-known first person shooter game developed by the ID™ software computer games company.

**Quadtrees**

Quadtrees are the 2D version of octrees the only difference being that quadtrees divide by four instead of eight. See octrees.

**Real-Time**

From the users' point of view, a real-time program calculates results immediately. An analogy of real-time is TV programs. Live broad casts are real-time, while movies take months to produce before the results can be seen. Many-times in computer graphics approximations are used instead of the more accurate calculations to achieve real-time results.

**Rendering**

Rendering is the process of converting 3D data into a 2D picture on the screen.

**Runtime**

Run-time is the time when the program is running in memory (RAM) performing the task it has been programmed to do.

**Sector**

A sector is a sub section of a map, such as a room or a hallway.

**Simplification**

Simplification approximates an object using less polygons then the original object. It is normally used to reduce polygon count, as things get further away.

**Slabs**

A slab is a 1-D slice of a 3D-grid like a slice from a loaf of bread.

**Software Rendering**

Something is software rendered when no 3D hardware acceleration is used.

**Solid**

A solid object is an object that does not have polygonal holes in it (as shown in Figure 138). To illustrate this concept, imagine the object was physical, and filled with gas. In a solid object the gas would not be able to escape.

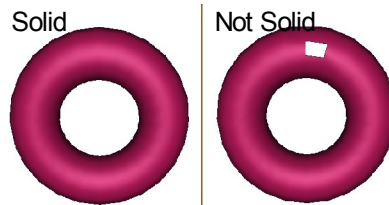


Figure 138. Solid verse not solid

### Sorting

The process of rearranging all elements (pieces of data) so that each can be considered priority-ordered consecutively in memory. In this context, it is not to be confused with priority-ordering. See priority-ordering.

### Static

Something that is static cannot change nor move around.

### Structural brushes

Structural brushes are brushes that are large enough and sparse enough to be considered for break up into smaller clusters. For example a few rooms where the camera can not see all of them at once. See brushes, detailed brushes.

### Texture

A texture is an image or picture such as a bitmap. Textures are normally 2D but can also be 3D. See Texture Map.

### Texture Map

When a texture's coordinates is mapped to a surface. Texture coordinates are normally given as  $(u, v)$  as shown in Figure 139.

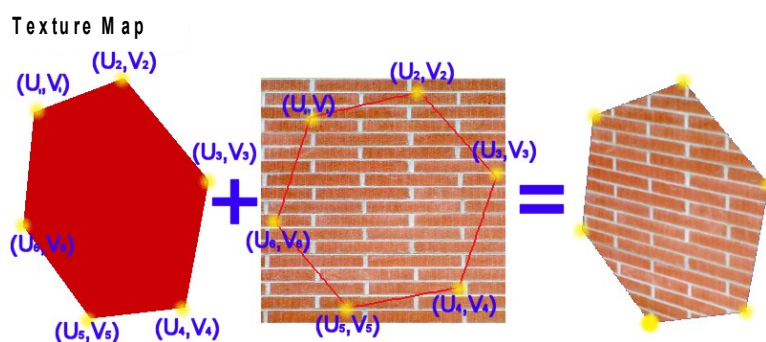


Figure 139. Texture map

### Traversal

Traversal is the process of visiting all the nodes with in a data structure by following the path establish by its links. In many cases there is more then one path to follow, so each node must take a turn at being visited. At each node visit in the structure traversal some action with the node contained there may be taken.

**Tree**

A tree is a data structure that is made up of nodes. Each node contains two or more pointers to other nodes (children) making hierarchical structure that can be traversed. Therefore each tree points to a sub-tree (recursively), which can be dealt with in the same way the parent can. See node.

**Tree Balancing**

Tree balancing is the process of making the nodes in a tree as evenly balanced as possible. Therefore each sub-tree (recursively) should result in having approximately the same amount of nodes as its siblings. Balancing is used to average out tree traversal times.

**Viewport**

A viewport is the window on the screen that the image is rendered to. A screen may have many view ports at one time. When a scene is drawn into a viewport all parts of the image that do not fit within are clipped.

**Viewpoint**

A viewpoint is the position at which the camera sits in the world.

**Visible Buffer**

A buffer that is eventually made visible to the user by the screen.

**World**

In the context of this report a world means the entire simulated scene area that the camera can move around in a view.

**Z-buffer**

See Depth-Buffer.