Uppsala Master's Theses in Computing Science
Examensarbete 179
2001-01-19
ISSN 1100-1836

Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering

Samuel Ranta-Eskola

Information Technology
Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

Supervisor: Erik Olofsson
Examiner:
Passed:

BSP-TREES AND POLYGON REMOVAL IN REAL TIME 3D RENDERING

By Samuel Ranta-Eskola

Uppsala University

Abstract

When the original design of the algorithm for Binary Space Partitioning (BSP)-trees was formulated the idea was to use it to sort the polygons in the world. The reason for this was there did not exist hardware accelerated Z-buffers, and software Z-buffering was too slow. Today that area of usage is obsolete, since hardware accelerated Z-buffers exist. Instead the usage is to optimise a wide variety of areas, such as radiosity calculations, drawing of the world, collision detection and networking.

We set out to examine the areas where one can draw advantages of the structure supplied and study the generating process.

As conclusion a BSP-tree is a very useful structure in most game engines. Although there are some drawbacks with it, such as that it is static and it is very expensive to modify during run-time. Hopefully some ideas can be taken from the BSP-tree algorithm to develop a more dynamic structure that has the same advantages as the BSP-tree.

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgments

The author wishes to thank Andreas Brinck for giving essential help that was needed to get started. Erik Olofsson at O3games was also an invaluable asset when it came to feedback and helping to solve difficult problems along the way.

GLOSSARY

FPS First person shooter, a game viewed in first person perspective, where the goal is to eliminate the opponents.

Map An object that contains the geometry of the world.

BSP-tree Binary Space Partioning tree, a tree structure used to divide a map into smaller parts.

Z-Value This is a measurement used to classify how close a polygon is to the viewer's position.

Frame rate The number of times the screen is updated per second. This has nothing to do with the refresh rate of the monitor. It is the number of times the world is drawn per second. Usually this should be above 30 times/sec to give a continuous feeling.

Pre-processing Calculations that are done before run-time, thus saving valuable CPU time at run time that can be used for other things.

Polygon A polygon is a many-sided planar figure composed of vertices and edges. Triangles, squares, hexagons, and pentagons are examples of polygons that have names, but any closed series of line segments forms a polygon. [Unknown Author, Basic Math FAQ]. To be meaningful in this area of usage all vertices must be on the same plane, i.e. 2-dimensional.

Plane equation The mathematic formula for a 3 dimensional plane. Ax+By+Cz+D, where A-D is the constant coefficients of the equation, A-C is the normal of the plane and D is the distance from origin in the direction of the normal to the plane.

Node A part of a tree. Each node consists of a left- and a right sub tree.

PVS Potentially Visible Set; given a position this is the set of objects/polygons/nodes that are potentially visible from that location.

Portal A hole through which two nodes are connected or a mirror on which a scene can be rendered.

Radiosity A lightning model commonly used in game engines. Main feature is so called color bleeding, where walls "bleed" its color to neighbor walls.

Avatar An object that represents a player in the virtual world.

Client A user that is connected to a server in a multiplayer application.

Viewing frustum The field of view for the camera. Often shaped as a pyramid with the top in the camera.

Target system The minimum system that a game should be able to run on.

Scene A set of objects with attribute from which one can render an image from any viewing position and angle.

LOD Level of Detail, when this technique is used objects is drawn with different amount of detail depending on distance from the viewer. The reason is to reduce the polygon count in the scene. The closer an object is to the viewer the more detailed it will be.

Bounding box Generally this is defined as the least box that encapsulates some set of objects, for example points, polygons, other bounding boxes etc. When we use the term as the bounding box of a BSP-node, we mean that this is the least box that encapsulates all polygons in the sub trees of this node.

Chapter 1

INTRODUCTION

Background

Binary Space Partioning (BSP)-trees were first described in 1969 by Shumacker et al.[1], it was hardly meant to be an algorithm used to develop entertainment products, but since the beginning of the 90's BSP-trees have been used in the gaming industry to improve performance and make it possible to use more details in the maps[2]. The first game ever to use the technology is Doom, created by John Carmack and John Romero, two legends in the gaming industry.[3] Since then almost all First Person Shooting (FPS[4]) games have been using that technique.

Problem Statement

Because of the tough competition in the gaming industry a lot of work has been done to improve the original design of the algorithm, but we believe improvements can be done. Our main focus has been on moving costly calculations from run time to pre processing time, thus building a structure that holds a lot of information that can be used during run time. Another thing we wish to do with our work is to find ways to improve and optimize surrounding areas in a gaming engine using the strengths of BSP-trees. As a side effect this report can also be considered as a tutorial on how to develop a gaming engine.

In this report we are going to show:

- What a BSP-tree is.
- How to create a BSP-tree.
- Advantages / Disadvantages.
- Similar techniques that can be used.
- The usability of BSP-trees.

---

1 [Shumacker, R., Brand, R., Gilliland, M., Sharp, W. Study for Applying Computer-Generated Images to Visual Simulation]

2 See the glossary for description.

3 [http://www.idsoftware.com/corporate/index.html]

4 See the glossary for description.

- Compare our methods with existing methods.

The research for this thesis has been done at a company called O3games [http://www.o3games.com], which is a Swedish game developer. The results is used in a product released by Cloop Systems, a sister company. We have implemented a fully working BSP-tree with all the surrounding benefits, i.e. the areas where one can use the strengths of a BSP-tree. To support our discussion we are going to use examples from the code we have written. We will use pseudo-code similar to C++, when it is not clear we will analyze the complexity of the algorithm in order to shed light on where optimizations have been done. Most of our graphical examples are illustrated in 2D, but they would work just as well in 3D. Throughout the report the author will assume that the reader have some knowledge about basic concepts in 3D-math and vector algebra.

Chapter 2

BSP-TREES

Background

A Binary Space Partitioning-tree is a structure that, as the name suggests, subdivides the space into smaller sets. These days, given hardware accelerated Z-buffers; the benefit of this is that one has a smaller amount of data to consider given a location in space. But in the beginning of the 90's the main reason BSP-trees were being used was that they sorted the polygons in the scene so that you always drew back-to-front, meaning that the polygon with the lowest Z-value[5] was drawn last. There are other ways to sort the polygons so that the closest polygon is drawn last, for example the Painter's algorithm[6], but few are as cheap as BSP-trees, because the sorting of the polygons is done during the pre-processing[7] of the map and not under run- time. The algorithm for generating a BSP-tree is actually an extension of Painter's algorithm.[8] Just as the original design of the BSP algorithm, the Painter's algorithm works by drawing the polygons in the scene in back-to-front order. However, there are some major drawbacks with Painter's algorithm:

- Polygons will not be drawn correctly if they pass through any other polygon.
- It is difficult and computationally expensive to calculate the order that the polygons should be drawn in for each frame.
- The algorithm cannot handle cases of cyclic overlap as shown in the figure below.

Figure 1. Cyclic overlap[9]

[5] See the glossary for description.

[6] [Sobey, Anthony. Software Engineering and Sunsted, Tod. 3D computer graphics: Moving from wire-frame drawings to solid, shaded models]

[7] See glossary for description.

[8] [Feldman, Mark. Introduction to Binary Space Partioning Trees, 1997]

[9] [Feldman, Mark. Introduction to Binary Space Partioning Trees, 1997]

The BSP algorithm

The original idea for the creation of a BSP-tree is that you take a set of polygons that is part of a scene and divide them into smaller sets, where each subset is a convex set of polygons. That is that each polygon in this subset is in front of every other polygon in the same set. Polygon 1 is in front of polygon 2 if each vertex in polygon 1 is on the positive side of the plane polygon 2 defines or in that plane that. A cube made of inward facing polygons is a convex set, whilst a cube made of outwards facing polygons is not.

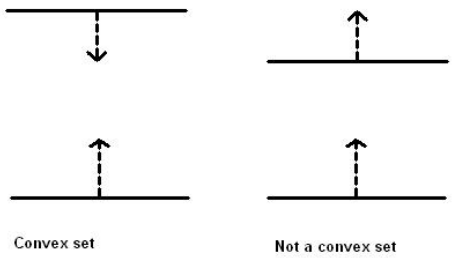


Figure 2. The difference between a convex set and a non-convex set.

The functions needed to determine whether a set of polygons is a convex set would look like this:

```
w CLASSIFY-POINT
w Indata:
w Polygon – The polygon to classify the point versus.
w Point   - 3D-point to classify versus the plane defined
w by the polygon.
w Outdata:
w Which side the point is of the polygon.
w Effect:
w Determines on which side of the plane defined by the polygon the
w point is located.

CLASSIFY-POINT (Polygon, Point)
1 SideValue  f  Polygon.Normal * Point
2 if (SideValue = Polygon.Distance)
3    then return COINCIDING
4 else if (SideValue < Polygon.Distance)
5    then return BEHIND
6 else return INFRONT
```

```
w POLYGON-INFRONT
w Indata:
w Polygon1 – The polygon to determine whether the other polygon is
w in front of or not
w Polygon2 – The polygon to check if it is in front of the first
w polygon or not
w Outdata:
w Whether the second is in front of the first polygon or not.
w Effect:
w Checks each point in the second polygon is in front of
w the first polygon. If so is the case it is considered
w to be in the front of it.
```

```
POLYGON-INFRONT (Polygon1, Polygon2)
1 for each point p in Polygon2
2    if (CLASSIFY-POINT (Polygon1, p) <> INFRONT)
3       then return false
4 return true
```

```
w IS-CONVEX-SET
w Indata:
w PolygonSet – The set of polygons to check for convexity
w Outdata:
w Whether the set is convex or not
w Effect:
w Checks each polygon against each other polygon, to see if they are
w in front of each other, if any two polygons doesn't fulfill that
w criteria the set isn't convex.
```

```
IS-CONVEX-SET (PolygonSet)
1 for i  f 0 to PolygonSet.Length ()
2    for j  f 0 to PolygonSet.Length ()
3    if(i <> j && not POLYGON-INFRONT(PolygonSet[i], PolygonSet[j]))
4           then return false
5 return true
```

The function POLYGON-INFRONT is a non-symmetric comparison, meaning that if Polygon2 is in front of Polygon1 it does not necessarily imply that Polygon1 is in front of Polygon2. This can easily be shown with the following example:

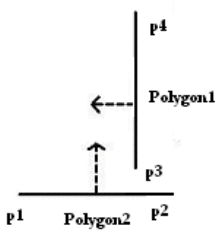


Figure 3. The non-symmetric nature of the comparison POLYGON-INFRONT

In Figure 3 Polygon1 is in front of Polygon2 since both p3 and p4 is on the positive side of Polygon2, but Polygon2 is not in front of Polygon1 since p2 is behind Polygon1.

The idea can be slightly modified as the need of convex sets is not as acute when you can use hardware accelerated Z-buffers. Later in this chapter it will be described how this was solved.

The structures needed for a BSP-tree can be defined as follows:

```
class BSPTree
{
     BSPTreeNode RootNode         w The root node of the tree.
}

class BSPTreeNode
{
     BSPTree Tree                 w The tree this node belongs to.
     BSPTreePolygon Divider       w The polygon that lies in middle
                                  w of the two sub trees.
     BSPTreeNode *RightChild      w The right sub tree of this node.
     BSPTreeNode *LeftChild       w The left sub tree of this node.
     BSPTreePolygon PolygonSet[] w The set of polygons in this node.
}

class BSPTreePolygon
{
     3DVector Point1              w Vertex 1 in the polygon.
     3DVector Point3              w Vertex 2 in the polygon.
     3DVector Point3              w Vertex 3 in the polygon.
}
```

As you can see each polygon is represented by only three points. This is because the hardware in graphic cards is designed to draw triangles. But the algorithm for generating BSP-trees is designed to take care of polygons with any number of points, as long as all points are in the same plane.

There are several ways to split up the set of polygons into smaller subsets. For example, you can choose an arbitrary plane in space and divide the polygons by putting the ones on the positive side of the plane in the right sub tree and the polygons on the negative side in the left sub tree. The problem with this approach is that it is very difficult to find a plane that divides the polygons into two approximately equally sized sets, since there are an infinite set of planes in space. So the most common way to do this is by taking one of the polygons in the scene and dividing the polygons according to the plane that polygon defines.

We have defined an algorithm, POLYGON-INFRONT, which can classify whether a polygon is on the positive side of another polygon. Now we need to modify that algorithm to be able to also determine whether the polygon is spanning the plane defined by the other polygon. The algorithm is defined as follows:

```
w CALCULATE-SIDE
w Indata  :
w Polygon1 – The polygon to classify the other polygon against
w Polygon2 – The polygon to classify
w Outdata :
w Which side of polygon1 polygon 2 is located on.
w Effect:
w Classifies each point in the second polygon versus the
w first polygon. If there are points on the positive side but no
w points on the negative side, Polygon2 is considered to be in front
w of Polygon1. If there are points on the negative side but no
w points on the positive side, Polygon2 is considered to be behind
w Polygon1. If all points are coinciding polygon2 is considered to
w be coinciding with Polygon1. The last possible case is that there
w are points on both the positive and the negative side, then
w polygon2 is considered to be spanning Polygon1.

CALCULATE-SIDE (Polygon1, Polygon2)
1  NumPositive f 0, NumNegative f 0
2  for each point p in Polygon2
3     if (CLASSIFY-POINT (Polygon1, p) = INFRONT)
4        then NumPositive = NumPositive + 1
5     if (CLASSIFY-POINT (Polygon1, p) = BEHIND)
6        then NumNegative = NumNegative + 1
7  if (NumPositive > 0 && NumNegative = 0)
8    then return INFRONT
9  else if(NumPositive = 0 && NumNegative > 0)
10    then return BEHIND
11 else if(NumPositive = 0 && NumNegative = 0)
12    then return COINCIDING
13 else return SPANNING
```

This gives us a problem when it comes to determining which subset a polygon that is spanning the plane should be placed in. The algorithm deals with this by splitting such a polygon into two polygons. This also solves two of the problems in Painter's algorithm, namely cyclic overlap and intersecting polygons. Below is example of how a polygon is splitted:

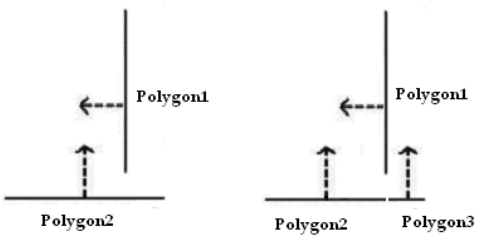


Figure 4. Splitting a polygon

In the figure above Polygon1 is the classifying polygon and Polygon2 is the polygon that is classified. Since Polygon2 is spanning the plane defined by Polygon1 it has to be splitted. The result is the picture to the right. Polygon2 is now completely in front of

Polygon1 and Polygon3 is completely behind. The glitch between Polygon2 and Polygon3 is just there to illustrate that it is two separate polygons, after a split the two resulting polygons will be adjacent to each other.

When a BSP-tree is created, one has to decide whether the need is of a balanced tree, meaning that there should not be too big a difference in depth between the left and the right sub tree of each node, or try to limit the number of splits, since each split creates new polygons. If too many new polygons is created during the BSP-tree creation the graphic card will have a hard time rendering the map, thus reducing the frame rate, while a unbalanced tree will require more expensive traversal of the tree. We decided to accept a certain number of splits in order to get a more balanced tree. But the main concern was reducing the number of new polygons created. Below is our loop for choosing the best dividing polygon from a set of polygons:

```
w CHOOSE-DIVIDING-POLYGON
w Indata:
w PolygonSet – The set of polygons to search for the best dividing
w polygon.
w Outdata:
w The best dividing polygon
w Effect:
w Searches through the set of polygons and returns the polygons that
w splits the set into the two best resulting sets. If the set is
w convex no polygon can be returned.

CHOOSE-DIVIDING-POLYGON (PolygonSet)
1  if (IS-CONVEX-SET (PolygonSet))
2     then return NOPOLYGON
3  MinRelation f MINIMUMRELATION
4  BestPolygon f NOPOLYGON
5  LeastSplits f INFINITY
6  BestRelation f 0

w Loop to find the polygon that best divides the set.
7  while(BestPolygon = NOPOLYGON)
8     for each polygon P1 in PolygonSet
9        if (Polygon P1 has not been used as divider previously
             during the creation of the tree)

w Count the number of polygons on the positive side, negative side
w of and spanning the plane defined by the current polygon.
10          NumPositive f 0, NumNegative f 0, NumSpanning f 0
11          for each polygon P2 in PolygonSet except P1
12             Value = CALCULATE-SIDE(P1, P2)
13             if(Value = INFRONT)
14                NumPositive = NumPositive + 1
15             else if(Value = BEHIND)
16                NumNegative = NumNegative + 1
17             else if(Value = SPANNING)
18                NumSpanning = NumSpanning + 1
```

```
w Calculate the relation between the number of polygons in the two
w sets divided by the current polygon.
19          if (NumPositive < NumNegative)
20             Relation = NumPositive / NumNegative
21          else
22             Relation = NumNegative / NumPositive

w Compare the results given by the current polygon to the best this
w far. If the this polygon splits fewer polygons and the relation
w between the resulting sets is acceptable this is the new candidate
w polygon. If the current polygon splits the same amount of polygons
w as the best polygon this far and the relation between the two
w resulting sets is better -> this polygon is the new candidate
w polygon.
23          if (Relation > MinRelation &&
                (NumSpanning < LeastSplits ||
                 (NumSpanning = LeastSplits &&
                  Relation > BestRelation))
24             BestPolygon f P1
25             LeastSplits f NumSpanning
26             BestRelation f Relation

w Decrease the number least acceptable relation by dividing it with
w a predefined constant.
27    MinRelation f MinRelation / MINRELATIONSCALE
28 return BestPolygon
```

Complexity analysis:
Because of the while loop it is very hard to find a bound to this function. Depending of the structure of the scene the while loop might loop for a very long time. The **MINRELATIONSCALE** is what decides how much the acceptable relation decreases per iteration, thus how long it will take before the minimum relation will be small enough to accept the best possible solution. The worst case is that we have a set consisting of n polygons that is not a convex set and the best possible solution is a dividing polygon that splits the set into one part consisting of n-1 polygons and another set consisting of 1 polygon. This solution will only be acceptable when the minimal acceptable relation is less than 1/(n-1) (see line 19-23 in the algorithm). Meaning that MinRelation / $\mathbf{MINRELATIONSCALE}^i$ < 1/(n-1) where i is the number of iterations in the loop, this is due the division by **MINRELATIONSCALE** at line 27 in the algorithm. Let's assume that the initial value for MinRelation is 1, which is the highest possible value since the relation is always between 0 and 1 (see lines 19-22 in the algorithm). We have:

$1 / \mathbf{MINRELATIONSCALE}^i < 1/(n-1)$
$1 < \mathbf{MINRELATIONSCALE}^i/(n-1)$
$(n-1) < \mathbf{MINRELATIONSCALE}^i$
$\log_{\mathbf{MINRELATIONSCALE}}(n-1) < \mathbf{i}$

This is no upper bound for i, but since i will be very close to $\log_{\mathbf{MINRELATIONSCALE}}(n-1)$ we will, for simplicity assume they are equal. Another practical assumption to make is that **MINRELATIONSCALE** always should be greater than or equal to 2. Thus giving us:

$\log_{\texttt{MINRELATIONSCALE}}(n-1) = \texttt{i} \quad \texttt{MINRELATIONSCALE} >= 2$
$\texttt{i} = \log_{\texttt{MINRELATIONSCALE}}(n-1) < \lg(n-1) = \texttt{O}(\lg \texttt{n})$

Inside the while loop, there are two iterations over the set of polygons. Giving us that the worst case behavior of this algorithm is of order $O(n^2 \lg n)$, but the typical behavior is almost always closer to $O(n^2)$ as there tend to exist a polygon that will fulfill the requirements in the first iteration.

The loop in CHOOSE-DIVIDING-POLYGON might look as if there are cases where it will not terminate, but this is not true since if the set of polygons is a non-convex set there is always one polygon that can divide the polygons into two sets. CHOOSE-DIVIDING-POLYGON selects the polygon that splits the least number of polygons. To prevent from choosing polygons that would not divide the set, the relation between the sizes of the two resulting sets must be better than a threshold value. To better illustrate this we show an example where choosing the polygon that splits the fewest amount of polygons would render in an infinite loop:

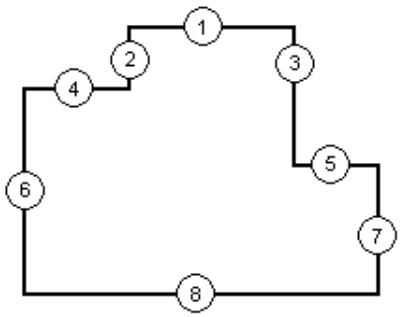


Figure 5. Problems when choosing dividing polygon.

In the above example choosing either polygon 1,6,7 or 8 would not render in the split of any polygon, but on the other hand each of the polygons in the set is on the positive side of those polygons, so in the next loop the same polygon would be chosen again, rendering in a infinite loop. As a matter of fact 1,2,3 and 4 is on the border of the least convex hull that can hold the polygon set, polygons for which this is true cannot be used as a dividing polygon since all other polygons in the set is on the positive side of them. Choosing polygon 2,3,4 or 5 would each cause one split but it would also divide the set into two smaller sets.

Another reason why a it is not always good to choose the polygon that splits the fewest polygons is that in most cases that heuristic will render in a unbalanced set. A balanced tree will perform better during runtime than an unbalanced one.

When the best polygon has been chosen the rest of the polygons is divided according to that polygon. There are two ways to do deal with the dividing polygon:

1. A leafy tree can be created, meaning that all polygons are put into the leaf nodes, thus the dividing polygons have to be categorized to be on one of the sides. In our example we count the polygons in the same plane as the dividing polygon as being on the positive side of the plane.

2. The other way is to store the dividing polygons in the internal nodes. This process is repeated for each sub tree until each leaf contains a convex set of polygons.

The algorithm for generating a leafy BSP-tree looks like this:

```
w GENERATE-BSP-TREE
w Indata:
w Node – The sub tree to build of the type BSPTreeNode.
w PolygonSet – The set of polygons to create a BSP-tree from.
w Outdata:
w A BSP-tree stored in the incoming node.
w Effect:
w Generates a BSP-tree out of a set of polygons.

GENERATE-BSP-TREE (Node, PolygonSet)
1  if (IS-CONVEX-SET (PolygonSet))
2     Tree f BSPTreeNode (PolygonSet)
3  Divider f CHOOSE-DIVIDING-POLYGON (PolygonSet)
4  PositiveSet f {}
5  NegativeSet f {}
6  for each polygon P1 in PolygonSet
7     Value f CALCULATE-SIDE (Divider, P1)
8     if(Value = INFRONT)
9        PositiveSet f PositiveSet U P1
10    else if (Value = BEHIND)
11       NegativeSet f NegativeSet U P1
12    else if (Value = SPANNING)
13       Split_Polygon[10] (P1, Divider, Front, Back)
14       PositiveSet f PositiveSet U Front
15        NegativeSet f NegativeSet U Back
16 GENERATE-BSP-TREE (Tree.RightChild, PositiveSet)
17 GENERATE-BSP-TREE (Tree.LeftChild, NegativeSet)
```

Complexity analysis:
The call to CHOOSE-DIVIDING-POLYGON is of order $O(n^2 \lg n)$, which dominates the rest of the function except for the recursive calls. If we assume that the division of the polygon set is fairly even we can formulate the following function to calculate the bounds of GENERATE-BSP-TREE:

$T(n) = 2T(n/2) + O(n^2 \lg n)$

Using Masters Theorem[11] we get that the order of complexity is $\Theta(n^2 \lg n)$, where n is the number of polygons in the incoming set.

[10] [Silicon Graphics. BSP Tree Frequently Asked Questions (FAQ)]

Following there is an example of how a BSP-tree is generated. The structure below is the original set of polygons, we have numbered them to make the example easier to follow. This set of polygons is going to be divided into a BSP-tree.

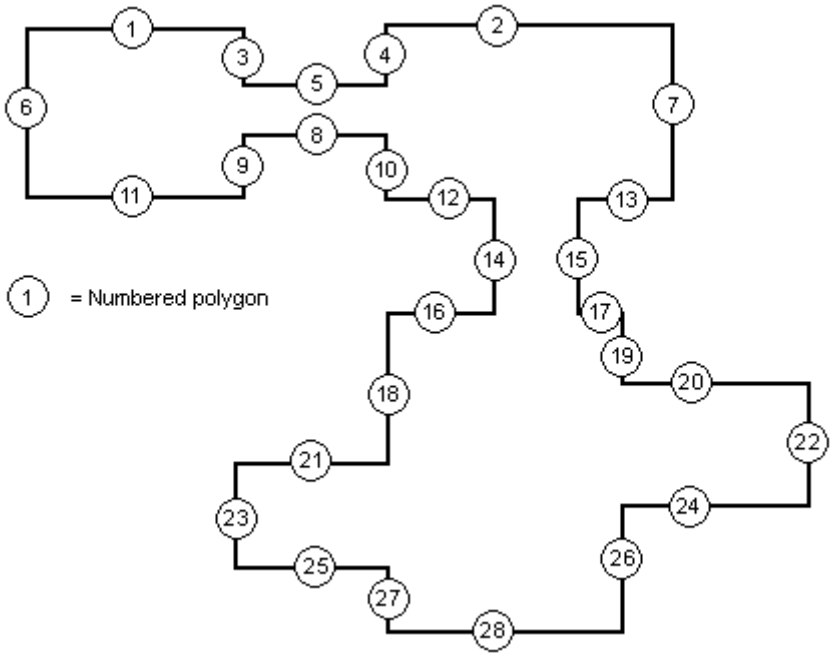


Figure 6. Example structure

To be able to run the algorithm we must choose a couple of constants, namely: **`MINIMUMRELATION`** and **`MINRELATIONSCALE.`** We found that choosing **`MINIMUMRELATION = 0.8`** and **`MINRELATIONSCALE = 2`** will give quite good result, but one can experiment we those numbers. The higher the **`MINIMUMRELATION`** is the better balanced the tree will be, but the number of splitted polygons will probably increase too.

The starting set of polygons is obviously not a convex set, so a dividing polygon will be chosen. After a quick glance at the structure we can see that polygons {1,2,6,22,28} cannot be used as dividing polygons since they define convex hull that contains the whole set of polygons. But all the other polygons are candidates for being dividing polygon. The polygons that split the fewest number of polygons and give the best relation between the sizes of the two resulting sets are 16 and 17, they lie on the same line and do not split any other polygon. The two resulting sets is almost equally sized namely |negative| = 15 and |positive| = 13 polygons in each of the resulting sets. Let us choose polygon 16 as the divider. The result will look as follows:

---

[11] [Cormen, Thomas H. Leiserson, Charles E. and Rivest, Ronald L.: Introduction to Algorithms]

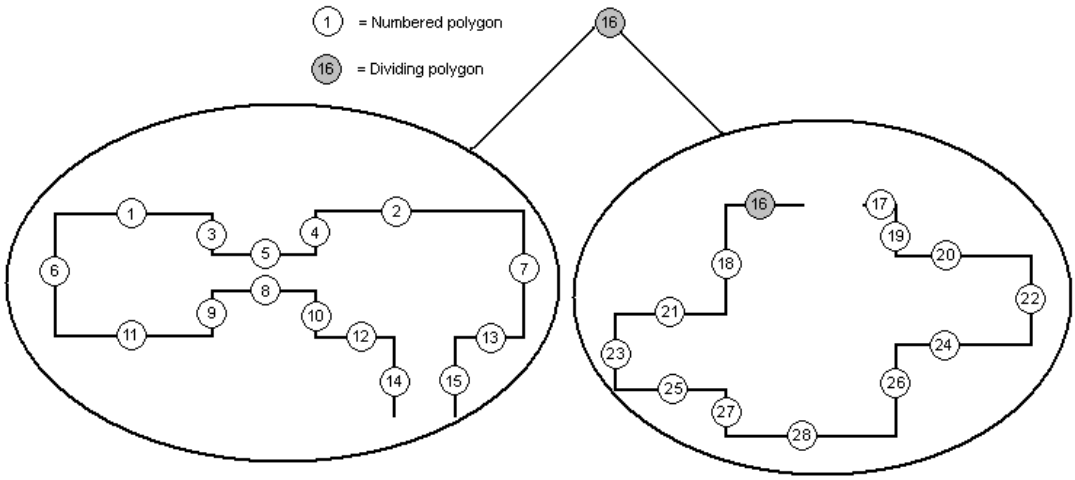


Figure 7. The result of a split at polygon 16

Neither the right nor the left sub tree contains a convex set of polygons so a new dividing polygon must be chosen in both.

In the left sub tree {1,2,6,7} is on the convex hull so they cannot be used as dividers. Polygon 4 and 10 is on the same line and they do not split any other polygon. The sizes of the resulting sets is |negative|= 7 and |positive| = 8 which is very balanced. We choose 4 as the divider.

{16,17,22,23,28} contains the right sub tree, so they will not be dividers. The polygons that will not split any other polygons are {18,19,27,26} but the sizes of the resulting sets for all of them will be |negative|= 3 and |positive| = 11, 3/11 is below the minimum relation(0.5) so we will have to check the other polygons to see if they can provide us with a more balanced solution. Each of {20,21,24,25} splits exactly one polygon, but the most balanced set is attained by polygon 21, which after splitting polygon 22 produces resulting sets of size |negative|= 6 and |positive| = 8.

On the next page the result after these operations is shown.

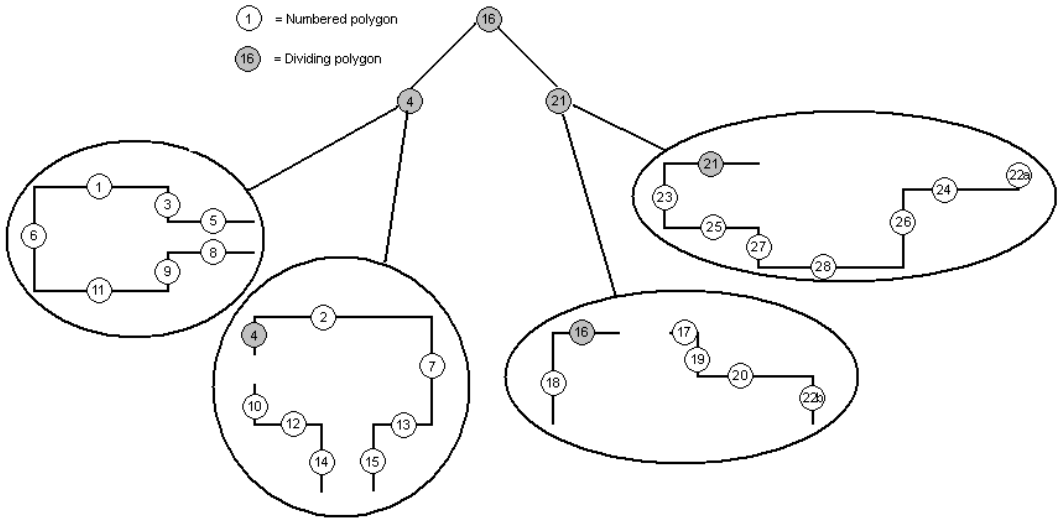


Figure 8. The second step.

None of the sub trees contain a convex set of polygons so the algorithm will move on in the same manner; the resulting tree will look like this:

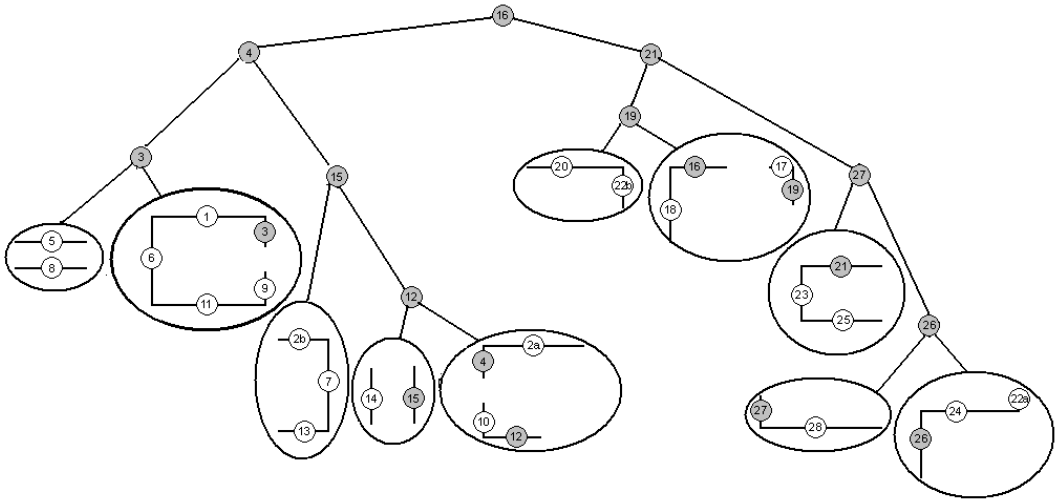


Figure 9. The final tree.

Even though it is not the optimal solution it is quite close to it and it does not take that long time.

Drawing the BSP-tree

Now that the BSP-tree is created it is very easy to draw the polygons the tree, with zero chance of failure in the drawing. Below the algorithm of that process is described. We assume there is a function `IS-LEAF` that given a BSP-node it returns true if that node is a leaf otherwise false.

```
w DRAW-BSP-TREE
w Indata:
w Node – The node to draw.
w Position – The viewer's position.
w Outdata:
w None
w Effect:
w Draws the polygons contained in the node and its sub trees.

DRAW-BSP-TREE (Node, Position)
1  if (IS-LEAF(Node))
2     DRAW-POLYGONS (Node.PolygonSet)
3     return

w Calculate which sub tree the viewer is in.
4  Side f CLASSIFY-POINT (Node.Divider, Position)

w If the viewer is in the right sub tree draw that tree before the
w left.
5  if (Side = INFRONT || Side = COINCIDING)
6     DRAW-BSP-TREE (Node.RightChild, Position)
7     DRAW-BSP-TREE (Node.LeftChild, Position)

w Otherwise draw the left first.
8  else if(Value = BEHIND)
9     DRAW-BSP-TREE (Node.LeftChild, Position)
10    DRAW-BSP-TREE (Node.RightChild, Position)
```

This way of drawing gives us no reduction in number of polygons that is drawn to the screen. Since a map can consist of hundreds of thousands of polygons, it is no good solution. In some way nodes that are not visible or even polygons that are not visible should be discarded. This is called hidden surface removal. There are several ways to do this; we will explain some of them in the next chapter.

Related reading:

[Sunsted, Tod. 3D computer graphics: Moving from wire-frame drawings to solid, shaded models]
[Sobey, Anthony. Software Engineering]
[Southwick, Andrew R. Quake Rendering Tutorial]
[Meanie, Mr.. Binary Space Partitioning Trees]
[Royer, Dan. Dan's Programming Tutorials]
[Feldman, Mark. Introduction to Binary Space Partioning Trees]

Chapter 3

HIDDEN SURFACE REMOVAL

Background

The need of removing what is not visible has been and always will be extremely high in the gaming industry, even though graphic cards evolve at gigantic rates and things that were true a couple years ago are not even remotely true these days. When a game is created a goal frame rate[12] is set. The lowest acceptable rate on a target system[13] use to be around 30 frames/second. A couple of years ago this meant putting out over 5000 textured polygons per frame could be too much. These days there are graphic cards in the market with the ability to draw hundreds of million of polygons per second during optimal conditions. Still there is a need of removing hidden parts. Why? Each hidden polygon that is drawn could be replaced by a polygon that is visible, hence increasing the detail in the scenes[14], making the game visually more attractive. The question is how far one should go to remove hidden polygons. To remove a hidden part heavy computations are needed to be done, such as view frustum[15] culling and portal[16] rendering.[17] The CPU-time used to do these computations could be used to enhance other effects in the game, such as AI and collision detection. Hence there is a lot to take in to consideration when developing algorithms for removal of hidden surfaces. There are almost no games that go so far as to remove each polygon that is hidden. Most games are satisfied with the removal of whole sets of polygons, such as nodes, objects etc. They do not consider individual polygons, so it seems like the correct way to go is to accept some overdraw to limit the computations when removing hidden surfaces.

The most common technique to remove hidden surfaces when creating a FPS is portal rendering. This technique is very well suited to utilize the benefits of BSP-trees, though the use of BSP-trees is not necessary. We considered to use this but thought that a more static representation could give a speedier rendering of the BSP-tree. The portal rendering has some nice side effects such as mirrors and surveillance cameras that we cannot do with our technique, but on the other hand, our technique require much less computations during run-time.

---

12 See the glossary for definition.

13 See the glossary for definition.

14 See glossary for definition.

15 See the glossary for description.

16 See glossary for description

17 [Hoff III, Kenneth E. Faster 3D Game Graphics by Not Drawing What Is Not Seen]

Portal Rendering

The world can be described as several sectors that are connected to each other through portals. A sector is a convex and closed set of polygons, where closed means that there is no way for a line drawn in the sector to get out of the sector without encountering a polygon.[18] This means that each hole in each node must be filled with a portal polygon. The placement of portal polygons can either be done manually or automatically. As we described before the need of convex sectors has disappeared with hardware-accelerated Z-buffers, so many game engines skip that criteria. But we are going to describe how to do it the old fashioned way.

The basic idea with a portal engine is that when you render a scene from a viewer's position with a viewing frustum and encounter a portal polygon, the portal clips the viewing frustum. Then the adjacent sector is rendered from the same viewer's position but with the new viewing frustum. This is a very simple approach and is very well suited for a recursive function. Many objects that are visible can easily be culled away since the viewing frustum is limited only to be exactly what you see. Below is a picture of how a viewing frustum can be clipped in a portal engine:

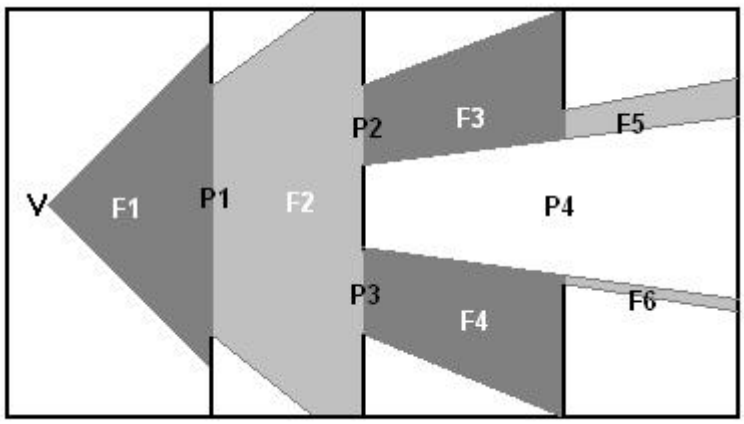


Figure 10. View frustum clipping

In the figure above the viewer is positioned at V, the original view frustum is F1. When F1 encounters portal polygon P1 it is clipped and renamed to F2. Later on F2 encounters portals P2 and P3 and is clipped to F3 and F4. When encountering portal P4, F3 is trimmed to F5 and F4 is trimmed to F6. This process is well suited for a recursive function.

[18] [Tyberghein, Jorrit. The Portal Technique for Real-time 3D Engines].

To cull away an object in a portal-rendering engine, as matter of a fact any 3D-engine, there are a series of steps that can be done to speed up the process. First, compute a bounding sphere for that object; a bounding sphere is the smallest sphere that can hold each vertex in an object. Optimally this is done once and for all during the creation of the object. Then, that sphere is tested against each plane in the viewing frustum. If it is on the completely negative side of any one of those planes the object is not visible and is not drawn. The figure below describes a situation where one object is culled, thus not drawn, while the other is drawn.

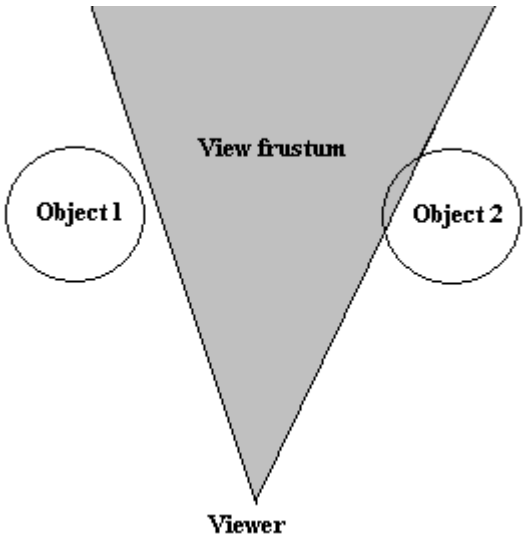


Figure 11. Culling of objects

Object 1 in the figure is on the positive side of the right plane in the view frustum, but it is on the negative side of the left plane so it is culled. The other object (2), is completely on the positive side of the left plane while a part of it is on the positive side of the right plane so it cannot be culled.

The original idea was that portal engines would have zero overdraw by clipping the polygons so that only the visible area would be drawn. These days this is an awful waste of processor time. But since a polygon can be encountered several times in the recursive loop that draws the scene we need to know if a polygon has been drawn or not. A good way to do this is to tag the polygons with a frame counter which indicates which was the last frame the polygon was drawn. That is the case for the right wall in figure 4, which should be drawn in frustum F5 and frustum F6, polygons has to be tagged to tell if they have been drawn this frame or not. Otherwise there will be Z-buffering errors.

In order to be able to render in portal engine we need to define what a view frustum consist of. A view frustum is a structure that holds n number of planes, each of these planes' normals faces inwards the view frustum, thus enclosing a volume referred to as inside the frustum. Below there is an algorithm on how to calculate whether a polygon is

inside a frustum or not, in this algorithm we use the function CLASSIFY-POINT as if it takes a plane and a point as parameters.

```
w INSIDE-FRUSTUM
w Indata:
w Frustum – The frustum to check whether the polygon is inside or
w not.
w Polygon – The polygon to check.
w Outdata:
w Whether the polygon was inside the frustum or not.
w Effect:
w Checks each point in the polygon versus each plane in the view
w frustum. If any point is on the positive side of all planes the
w polygon is counted as inside.

INSIDE-FRUSTUM (Frustum, Polygon)
1 for each point Pt in Polygon
2    Inside f true
3    for each plane Pl in Frustum
4       if (CLASSIFY-POINT (Pl, Pt) <> INFRONT)
5          Inside f false
6    if (Inside)
7       return true
8 return false
```

The main rendering function in a portal engine would look something like this:

```
w RENDER-PORTAL-ENGINE
w Indata:
w Sector – The sector the viewer is in.
w ViewFrustum – The current viewing frustum.
w Outdata:
w None
w Effect:
w Renders the polygons in a portal engine. Where the world is
w represented as sectors connected by portals.

RENDER-PORTAL-ENGINE (Sector, ViewFrustum)
1 for each polygon P1 in Sector
2    if (P1 is a portal and INSIDE-FRUSTUM (ViewFrustum, P1))
3       NewFrustum f CLIP-FRUSTUM (ViewFrustum, P1)
4       NewSector f get the sector that is connected with the
                    current sector through portal P1
5       RENDER-PORTAL-ENGINE (NewSector, NewFrustum)
6    else if (P1 has not been drawn yet)
7        draw P1
8  return
```

Placing the portals

As we mentioned before, one of the big problems in a portal engine is the placement of the portals. It is a very time consuming process to manually place the portals, not to mention the skill required of the map designer. As with many other things, that time could be better used in other places. So a good algorithm for automatic portal placement is needed. A colleague of mine, Andreas Brinck, has come up with a good solution to this problem. To use his solution a BSP-tree will have to be used.

The general idea is that each portal in the tree must be coinciding with a plane defined by a dividing polygon in the tree. Out of each of these planes a portal polygon is created, that portal polygon is initially a four-sided polygon that exceeds the bounding box[19] of the node it is located in. Then each portal polygon is pushed down the sub trees of the node it is in. When a portal polygon passes through a node in one of its sub trees the plane defined by the dividing polygon in that node clips it, it is also clipped by the polygons in that node if the node is a leaf. If a polygon is clipped, the two resulting parts are sent down from the top of the tree. When a portal polygon is not in need of any clipping, it is sent down to the sub trees of the node currently visiting. This means that if it is on the positive side of the plane it will be sent down the right sub tree, and if it is on the negative side it will be sent down the left sub tree. But if it is coinciding with the plane defined by the dividing polygon in the current node it will be sent down both sub trees.

In order to be able to define the algorithm that places all the portals in the tree we need to define how to clip a polygon, for this we need to assume there is a function called `INTERSECTION-POINT` that returns a intersection point between a plane and a line between two 3D points.

[19] See the glossary for definition.

```
w CLIP-POLYGON
w Indata:
w Clipper – The plane/polygon to clip the other polygon versus.
w Polygon – The polygon to clip.
w Outdata:
w The two resulting pieces after the clipping.
w Effect:
w Clips the polygon by the plane defined by the clipper polygon. If
w the polygon isn't spanning the clipper one of the resulting parts
w will be an empty polygon

CLIP-POLYGON (Clipper, Polygon)
1  RightPart f {}
2  LeftPart f {}
3  for each point edge E in Polygon
4     Side1 f CLASSIFY-POINT (Clipper, E.Point1)
5     Side2 f CLASSIFY-POINT (Clipper, E.Point2)
6     if (Side1 <> Side2 and
          Side1 <> COINCIDING and
          Side2 <> COINCIDING)
7        Ip f INTERSECTION-POINT (Clipper, E)
8        if (Side1 = INFRONT)
9           RightPart f RightPart U E.Point1
10          RightPart f RightPart U Ip
11          LeftPart f LeftPart U Ip
12          LeftPart f LeftPart U E.Point2
13       if (Side1 = BEHIND)
14          LeftPart f LeftPart U E.Point1
15          LeftPart f LeftPart U Ip
16          RightPart f RightPart U Ip
17          RightPart f RightPart U E.Point2
18    else
19       if (Side1 = INFRONT or Side2 = INFRONT or
            Side1 = COINCIDING and Side2 = COINCIDING)
20          RightPart f RightPart U E.Point1
21          RightPart f RightPart U E.Point2
22       if (Side1 = BEHIND or Side2 = BEHIND)
23          LeftPart f LeftPart U E.Point1
24          LeftPart f LeftPart U E.Point2
25 return (RightPart, LeftPart)
```

Now we can define how to distribute the portals in a BSP-tree. The algorithm is initially called with a portal polygon that is larger than the bounding box[20] of root node of the tree. We got the design to this function from a fellow game programmer, Andreas Brinck, currently employed at DICE, Sweden.

[20] See the glossary for definition.

```
w PLACE-PORTALS
w Indata:
w PortalPolygon – Polygon to push down the tree
w Node – The node that we are currently visiting.
w Outdata:
w None
w Effect:
w Pushes a portal polygon down through the tree clipping when it
w needs it. The output of this function will be that each node
w contains a list of portal polygons where each portal connects
w exactly two nodes.

PLACE-PORTALS (PortalPolygon, Node)
1  if (IS-LEAF (Node))

w The portal is checked against every polygon in the node. When the
w portal polygon is spanning the plane defined by a polygon it will
w be clipped against that plane. The two resulting parts will be
w sent down from the top of the tree again.
2     for (each polygon P2 in Node)
3        IsClipped f false
4        if (CALCULATE-SIDE (P2, PortalPolygon) = SPANNING)
5           IsClipped f true
6           (RightPart, LeftPart) f CLIP-POLYGON (P2, PortalPolygon)
7           PLACE-PORTALS (RightPart, RootNode)
8           PLACE-PORTALS (LeftPart, RootNode)
9     if (not IsClipped)
10       Remove the parts of the portal polygon that coincide with
         other polygons in this node. w see description below
11       Add this node to the set of connected nodes in this
         portal polygon.
12 else
13    if (the dividing polygon of this node hasn't been pushed down
          the tree)
14       Create a polygon P that is larger than the bounding box that
         contains all polygons in the sub trees of this node that
         lies in the same plane as the dividing polygon.
15       PLACE-PORTALS (P, Node.LeftChild)
16       PLACE-PORTALS (P, Node.RightChild)
17    Side f CALCULATE-SIDE (Node.Divider, PortalPolygon)
18    if (Side = POSITIVE)
19       (RightPart, LeftPart) f CLIP-POLYGON(P2, PortalPolygon)
20       PLACE-PORTALS (RightPart, RootNode)
21       PLACE-PORTALS (LeftPart, RootNode)
22    if (Side = POSITIVE or COINCIDING)
23       PLACE-PORTALS (PortalPolygon, Node.RightChild)
24    if (Side = NEGATIVE or COINCIDING)
25       PLACE-PORTALS (PortalPolygon, Node.LeftChild)
```

Complexity analysis:
This function is extremely complex to analyze and since it is not our design we will not analyze it.

To clarify line 10 we need to show what happens when we remove the coinciding parts between the portal polygon and other polygons in the node. See the image on the next page:

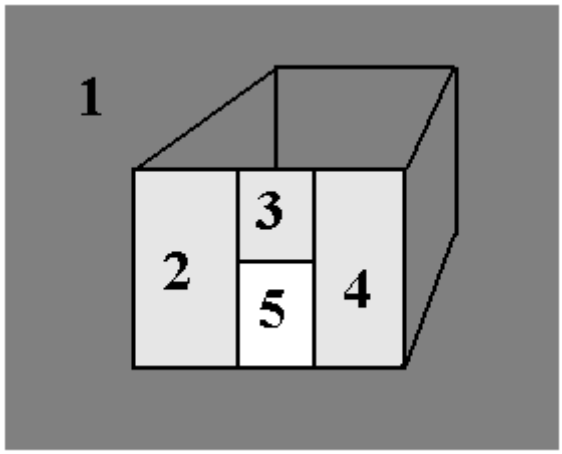


Figure 12. Removing coinciding parts.

In Figure 12 a portal polygon has reached a leaf. The dark gray areas marked as 1 is removed during the pushing down the tree. Parts 2, 3 and 4 which is painted in light gray is coinciding with polygons in the end node thus they are removed. The only remaining part is the part marked as 5; this is going to be used as a portal.

The algorithm on the previous page might look very complex at first sight but it is in fact very simple and intuitive. In the end every portal polygon will end up in exactly two nodes. These are the two portals that will be visible from each other. On the next page there is an example of the algorithm implemented.

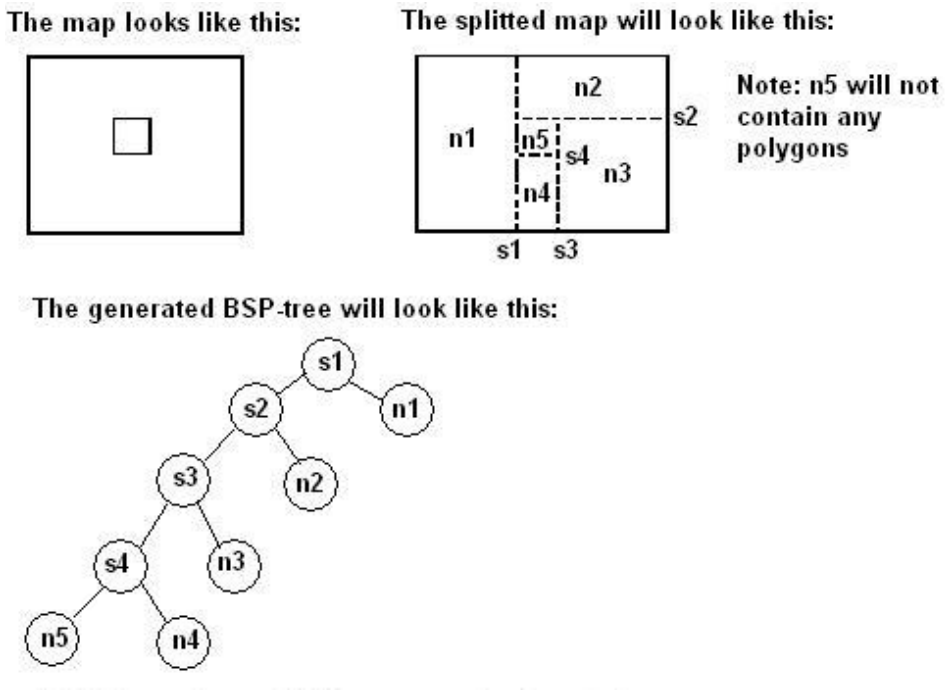
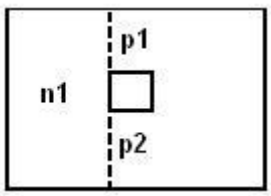


Figure 13. An example map for automatic portal placement.

1. Portal polygon 1 (s1) enters node n1.

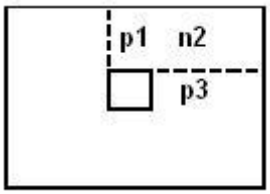


In n1 the splitting polygon will be clipped to fit and one part will be removed since it coincides with one of the polygons in the pillar. This leaves us with two polygons, namely p1 and p2. These two polygons replace s1.

2. p1 and p2 enters node s2

In node s2 p1 since it is on the positive side of s2 together with splitting polygon s2 will be sent to node n2. p2 (because it is on the negative side of s2) together with s2 will be sent further down to s3, none of them will be clipped since they do not cross splitting polygon s2.

3. p1 and s2 enters node n2

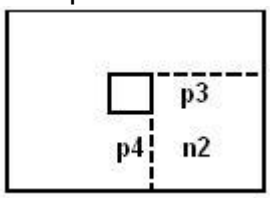


In n2 p1 is accepted as a portal, so it is not changed in node n1 either, and a part of p3 is removed since it coincides with a polygon in the pillar. Polygon s2 that was sent down to s3 in the previous step is now called p3.

4. p3 and s3 enters node n3.

Since neither of p2 or p3 is clipped they are pushed downwards together with s3. p3 and s3 goes down to node n3 and p2 and s3 is pushed down to node s4.

5. p3 and s3 enters node n3

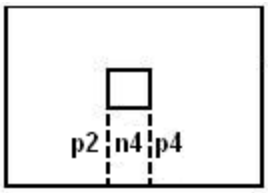


In n2 p3 is accepted as a portal and a part of s3 is removed by the same reasons as before. s3 is now named p4.

6. p2 and p4 enters node s4

None of the polygons need clipping, both p2 and p4 are sent down to n4 together with s4. Only s4 is sent to n5.

7. p2, p4 and s4 enters node n4

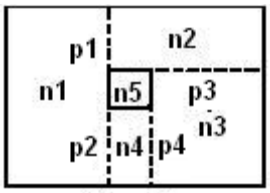


Neither of p2 or p4 need clipping, except for that to fit the node. But s4 is completely coinciding with a polygon in the pillar so it is removed.

8. Nothing enters node n5.

This node will have no portals, since it is not visible from any node and cannot see any other node.

9. The result



Portal p1 is in both n1 and n2.
Portal p2 is in both n1 and n4.
Portal p3 is in both n2 and n3.
Portal p4 is in both n3 and n4.

This is everything we need for building a simple portal engine that will give a relatively good frame rate.

Our Solution

A portal engine has a very flexible structure that provides some nice features. When we started designing our system we considered doing it as a portal engine, but there are some problems with portal engines, especially all the clipping that occurs when you are drawing the scene. So we decided to do a more static solution to avoid expensive calculations during run-time. The idea is somewhat similar to a portal engine but instead of calculating what needs to be drawn during run-time it is done in the pre-rendering of the map. For each leaf in the BSP-tree a Potentially Visible Set (PVS[21]) is created. This PVS is the set of leaves that is visible from the first leaf; it is not only of use during the drawing phase. It can also be used when radiosity[22] is calculated and networking is optimised for example.

The PVS is calculated during the pre-rendering of the map. In each leaf a set of visible leaves is stored. When a scene should be drawn, first the leaf where the camera is in is drawn, and then each leaf in the PVS is drawn. This requires that you have some kind of algorithm that takes care of overdraw. As we have mentioned before, graphic cards of today has hardware accelerated Z-buffers, which is enough.

Calculating the PVS

To calculate the PVS we need to do standard ray tracing between the leaves, to see if any point in a leaf is visible from another leaf. Each leaf has to have some sample points, between which visibility can be traced. These sample points have to be as few as possible to avoid massive calculations. The problem is how to distribute them.

As with the portals in a portal engine the sample points can be distributed along the splitting planes in the tree, because only the openings between the leaves have to be checked for visibility. If a point that lies in the centre of a leaf is considered visible by a ray that came from another leaf, the ray must have passed through an opening in the leaf. See the next page for a clarifying picture.

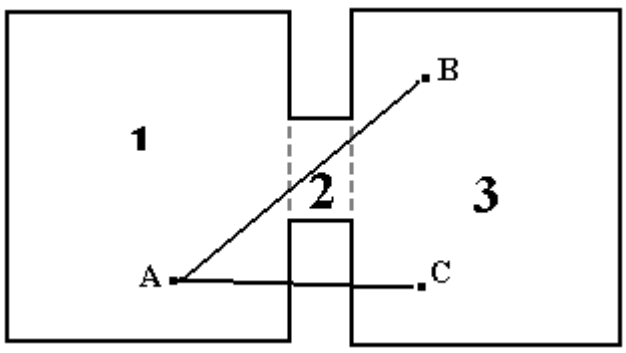


Figure 14. Visibility between nodes.

[21] See the glossary for description.

[22] See the glossary for description.

In the figure above we clearly see that for a point to be visible from another node the visibility line must pass through an opening in the node. This is very obvious since if the passed somewhere else the line would be obstructed, thus there would be no visibility between the two points. Hence distributing the sample points in the openings of the nodes is adequate. Below we have described an algorithm on how to distribute sample points in a BSP-tree. For this function we need a couple of helper function that distributes points in a node. These are:

- `DISTRIBUTE-POINTS (Node)` This function distributes points with a certain interval along the splitting plane of the incoming node, within the boundaries of the bounding box of the node[23]. It returns a set of points. Complexity: O(xy), where x is the width of the dividing plane in the bounding box and y is the height.

- `CLEANUP-POINTS (Node, PointSet)` Removes points from the point set that are either coinciding with a polygon in the node or outside the bounding box of the node. Complexity: O(np), where n is the number of polygons in the node and p is the number of points in the set.

[23] See the glossary for definition.

```
w Function: DISTRIBUTE-SAMPLE-POINTS
w Indata:
w Node     - The current we are visiting.
w PointSet – A set of points to distribute in the sub tree of the
w node.
w Outdata:
w None
w Effect:
w Distributes points along the splitting plane of this node. Then it
w divides the incoming points according to the splitting plane and
w removes the points that are coinciding with a polygon in this node
w or is outside the  bounding box of this node. The newly created
w points  will be added to both the positive and negative set.  When
w a set of points reaches a leaf node the points are the sample
w points of this leaf.

DISTRIBUTE-SAMPLE-POINTS (Node, PointSet)
1  CLEANUP-POINTS (Node, PointSet)
2  if (IS-LEAF (Node))
3     Set the point set to be the sample points of this node
4  else
5     RightPart  f  NewPoints
6     NewPoints  f  DISTRIBUTE-POINTS (Node)
7     RightPart  f  NewPoints
8     LeftPart   f  NewPoints
9     for each point P in PointSet
10       Side  f  CLASSIFY-POINT (Node.Divider, P)
11       if (Side = COINCIDING)
12           RightPart  f  RightPart U P
13           LeftPart  f  LeftPart U P
14       if (Side = INFRONT)
15           RightPart  f  RightPart U P
16       if (Side = BEHIND)
17           LeftPart  f  LeftPart U P
18    DISTRIBUTE-SAMPLE-POINTS (Node.LeftChild, LeftPart)
19    DISTRIBUTE-SAMPLE-POINTS (Node.RightChild, RightPart)
```

Complexity analysis:
Each call to this function is of order O(np + xy) (see CLEANUP-POINTS and DISTRIBUTE-POINTS). To calculate the full complexity we can formulate the following function (we will assume that the set of points are evenly distributed in the both sets):

$T(n) = 2T(n/2) + O(np + xy)$

Using Masters Theorem[24] we get that the order of complexity is Θ (np + xy).

[24] [Cormen, Thomas H. Leiserson, Charles E. and Rivest, Ronald L.: Introduction to Algorithms]

The function is first called with the root node of the tree and an empty set as parameters. In words the function does the following. It starts by distributing points in the plane defined by the splitting polygon in the root node of the BSP-tree. Since a plane is an infinite shape this would generate an infinite number of sample points, so there has to be some boundaries in which the points are distributed. These boundaries are the bounding box of the root node.

After the points have been distributed all of them are sent down to both sub trees. When a set of sample points enter a node, they are divided into two sets, one set for the points on the positive side of the dividing plane in the node and one set for the points on the negative side. The points that are exactly on the plane are put in both sets. Then points are distributed along this nodes dividing plane with this nodes bounding box as boundaries. The newly distributed points are put in both sets. Now the positive set is sent down the right sub tree and the negative set is sent down the left sub tree. The process is repeated until a set of points enters a leaf. After these operations each leaf contains a set of sample points that are distributed in the openings of the node.

If we would ray trace between each node at this stage it would take quite long time. But if we knew which leaves are connected to each other it would be much easier, since this could be used to skip tracing between some leaves. It is very simple to find out which leaves that are connected to each other; just check the sample points in each leaf against each other leaf's sample points. If two nodes share one sample point these two nodes are connected to each other, because during the distribution of the sample points through the tree every point will end in zero or two leaves. When we know which leaves are connected, the algorithm for tracing visibility can be defined. But first we need to define some helper functions.

In order to trace visibility some basic ray tracing is needed. BSP-trees is a very good structure to ray trace in, since you can discard huge parts of the world, at a very little cost. The set of functions needed for our solution is:

- `POLYGON-IS-HIT (Polygon, Ray)` returns whether the ray interests the polygon or not. [25]

- `RAY-INTERSECTS-SOMETHING-IN-TREE (Node, Ray)` returns whether the ray intersects something in the sub tree of the node or not.

- `INTERSECTS-SPHERE (Sphere, Ray)` returns whether the ray interests the sphere or not.[26]

- `CREATE-RAY (Point1, Point2)` returns the ray between the two points.

[25] .[Åhs, Cons T and Bevemyr, Johan. Inlämningsuppgift I Programmeringsmetodik 2]

[26] .[Åhs, Cons T and Bevemyr, Johan. Inlämningsuppgift I Programmeringsmetodik 2]

RAY-INTERSECTS-SOMETHING-IN-TREE is the most interesting function of those above, since it shows some of the advantages with BSP-trees, and how BSP-trees can be used to optimize ordinary ray tracing. This is a recursive function that first is applied to the root node of the tree. The algorithm is formulated as follows:

```
w RAY-INTERSECTS-SOMETHING-IN-TREE
w Indata:
w Node – The node to trace through
w Ray – The ray to test for intersection.
w Outdata:
w Whether the ray intersected something or not.
w Effect:
w Checks if the ray intersects something in this node or any of this
w node's sub trees.

RAY-INTERSECTS-SOMETHING-IN-TREE (Node, Ray)
1  for each polygon P in Node
2     POLYGON-IS-HIT (P, Ray)
3  startSide f CLASSIFY-POINT (Ray.StartPoint, Node.Divider)
4  endSide f CLASSIFY-POINT (Node.EndPoint, Node.Divider)
w If the ray spans the splitting plane of this node or if the ray is
w coinciding with the plane, send it down to both children
5  if ((startSide = COINCIDING and endSide = COINCIDING) or
        startSide <> endSide and  startSide <> COINCIDING and
        endSide <> COINCIDING)
6     if (RAY-INTERSECTS-SOMETHING-IN-TREE (Node.LeftChild, Ray))
7        return true
8     if (RAY-INTERSECTS-SOMETHING-IN-TREE (Node.RightChild, Ray))
9        return true
w If the ray is only on the positive side of the splitting plane
w send the ray down the right child. The or in the if statement is
w because one of the points might be coinciding with the plane.
10    if (startSide = INFRONT or endSide = INFRONT)
11       if(RAY-INTERSECTS-SOMETHING-IN-TREE (Node.RightChild, Ray))
12          return true
w If the ray is only on the positive side of the splitting plane
w send the ray down the right child. The or in the if statement is
w because one of the points might be coinciding with the plane.
13    if (startSide = BEHIND or endSide = BEHIND)
14    if (RAY-INTERSECTS-SOMETHING-IN-TREE (Node.LeftChild, Ray))
15       return true
w There was no intersection anywhere, pass that upwards
16 return false
```

Complexity analysis:
Worst case is that the ray passes through exactly every node in the tree in which case it has to be tested against every single polygon. Giving us an order of O(n), where n is the number of polygons in the tree. Typically a ray will not pass through every node in the tree, thus reducing the number of polygons to check versus. The best case is if the ray is limited to only one node, in which case the order of the function will be somewhere around O(lg n), depending on the structure of the tree.

```
w CHECK-VISIBILITY
w Indata:
w Node1 – The starting node
w Node2 – The end node.
w Outdata:
w Whether Node2 is visible from node 1 or not.
w Effect:
w Traces between the sample points in the both leaves to see if
w there is visibility between the two nodes.

CHECK-VISIBILITY (Node1, Node2)
1 Visible f false
2 for each sample point P1 in Node1
3    for each sample point P2 in Node2
4      Ray f CREATE-RAY (P1, P2)
5      if(not RAY-INTERSECTS-SOMETHING-IN-TREE(Node1.Tree.RootNode,
                                                Ray)
6         Visible f true
7 return Visible
```

Complexity analysis:

The function CHECK-VISIBILITY is computationally extremely expensive. When we trace between to leaves between which there is no visibility, a trace from every sample point in node 1 to every sample point in node 2 has to be done. In worst case each of these traces has to be checked towards every polygon in the tree, hence the function would be $O(s_1 s_2 p)$, where s_1 is the number of sample points in node 1, s_2 is the number of sample points in node 2 and p is the number of polygons in the tree. Generally the behavior is much better, closer to $O(s_1 s_2 \lg p)$ because of the reduction of polygons that are needed to check versus in the ray tracing through the tree.

```
w TRACE-VISIBILITY
w Indata:
w Tree – The BSP-tree to trace visibility in.
w Outdata :
w None
w Effect:
w For each leaf in the tree it traces visibility to that leaf's
w connected nodes. Every node that is found visible is added to the
w PVS of that node. When a visible leaf is found we have to trace
w for  visibility to the visible nodes connected nodes.

TRACE-VISIBILITY (Tree)
1 for (each leaf L in Tree)
2    for (each leaf C that is connected to L)
3       Add C to L's PVS
4 for (each leaf L1 in Tree)
5    while (there exist a leaf L2 in L's PVS which's connected nodes
           hasn't been checked for visibility yet)
5       for (each leaf C that is connected to L2)
6          if (C isn't in L1's PVS already and
              CHECK-VISIBILITY (L1, C))
7             Add C to L1's PVS
7             Add L1 to C's PVS
```

Complexity analysis:

If we would not draw usage of the optimization that the connected leaves strategy gives us we would need to trace visibility between each leaf hence $O(n^2)$, where n is the number of leaves in the tree. It is very hard to give an approximation of how much the strategy speeds up the process since it is very much dependent of how the level is constructed. In a level where each leaf is visible from every other leaf it wouldn't optimize anything, while a level where only one or two leaves is visible from every other leaf it would optimize a great deal, almost down to O(n).

The structure that is generated now will discard large amounts of polygons each frame in a good map. A good map is built considering the visibility aspect, meaning that sight-blocking objects should be inserted, such as walls that prevent sight. If a map contains large room with enormous amounts of detail there is nothing our engine (or for that matter a portal engine) can do to remove hidden surfaces. In those bad cases there is another technique can be used to remove polygons; it is called level of detail (LOD[27]).

[27] See glossary for definition

Static Objects

Consider a map that consists of a sphere that lies in the middle of a box. When we try to render a BSP-tree from this map we would end up with a terrible amount of nodes, and great number of splitted polygons, since each of the polygons in the sphere would end up in separate leaves. So if the sphere consists of 200 polygons, a simple scene as this would render a BSP-tree of 200 leaves. A tree with such depth (in the mentioned case the depth would be as much as 200) would be very cumbersome during run time, not to mention all the extra polygons created because of the splitting. Certainly there is a need of taking care of such cases.

To solve this problem the map designer chooses which objects are defining the geometry of the map, in the example above the box would be such an object. The rest of the objects are classified as static objects, these are objects that will not be used to render the BSP-tree or during the visibility testing, but they will cast shadows during the lightning phase of the map. Each of the static objects will be added to the BSP-tree when the visibility calculations are done. This is done by taking each polygon in a static object and pushing it down the tree. The algorithm that pushes a polygon down the tree will need further description. It looks like this:

```
w PUSH-POLYGON
w Indata:
w Node – The node the polygon is currently in
w Polygon – The polygon to push down
w Outdata :
w None
w Effect:
w Pushes the polygon down through the tree. If the polygon at some
w point spans the dividing plane of a node it must be
w clipped. The resulting parts will be pushed downwards in the tree.
w When a polygon enters a leaf it will be added to the set of
w polygons in that leaf.

PUSH-POLYGON (Node, Polygon)
1  if (IS-LEAF (Node))
2     Node.PolygonSet  f  Node.PolygonSet U Polygon
3  else
4     Value  f  CALCULATE-SIDE (Node.Divider, Polygon)
5     if (Value = INFRONT or Value = SPANNING)
6        PUSH-POLYGON (Node.RightChild, Polygon)
7     else if (Value = BEHIND)
8        PUSH-POLYGON (Node.LeftChild, Polygon)
9     else if (Value = SPANNING)
10       Split_Polygon[28] (P1, Divider, Front, Back)
11       PUSH-POLYGON (Node.RightChild, Front)
12       PUSH-POLYGON (Node.LeftChild, Front)
```

[28] [Silicon Graphics. BSP Tree Frequently Asked Questions (FAQ)]

PUSH-POLYGON is a neat recursive function that adds a polygon to the tree. The function will be called once for every polygon in every static object together with the root node of the tree to add the object to.

After this process our leaves are no longer convex sets, this will render some problems when doing collision detection, which will be described in the chapter Physics in BSP-trees.

Related reading:

[Hoff III, Kenneth E. Faster 3D Game Graphics by Not Drawing What Is Not Seen]
[Tyberghein, Jorrit. The Portal Technique for Real-time 3D Engines]
[Bikker, Jacco. Building a 3D Portal Engine]
[Nuydens, Tom. 3D Engine Column, Delphi3D]
[Chalfin, Alex. Cells and Portals]
[Hoff, Kenny. The Warnock Area Subdivision Algorithm for Hidden Surface Removal]

Chapter 4

RADIOSITY

Background

The original idea for radiosity was formulated by a set of writers called Goral, Torrance, Battaile & Greenberg.[29] They suggested that radiosity would simulate energy transference between diffuse surfaces. That is surfaces that reflect light equally in all directions, as opposite to shiny surfaces. The result of such a simulation would give a view independent result, meaning that the illumination on surface would look the same from all viewing angles. This is very well suited for 3D games since the calculations only needs to be done once, during the pre rendering of the map.[30]

We will only give a quick brief in how the radiosity algorithm works and focus on how BSP-trees can be used to optimize the calculations. For more knowledge about the algorithm we suggest that you read some of the related reading in this chapter.

The radiosity algorithm is designed so that the lightning of a scene will be smooth and natural. If we would use a straightforward lightning model where each light sends out rays to the world and illuminates it without any further reflecting of light, shadows would we be very sharp and things would look very unnatural. To use the radiosity algorithm the world has to be divided into patches, where each patch represent a small part of the world. Each of these patches has an initial energy level, normally zero if it is not an emitting source such as lights, glowing walls or something like that.

There are several ways of distributing energy over the world. We chose to use so called iterative radiosity. This means that we start by sending out energy from the patch with the highest level of unsent energy in the scene, after which that patch unsent energy is set to zero. This process is repeated until it doesn't exist a patch which as energy above a certain threshold value.

When sending out energy from one patch (j) to another (i) the following formula is used:

$$B_i = B_i + B_j * F_{ij} * A_i / A_j$$

B_i = the level of energy level of patch I B_j = the level of energy level of patch j
F_{ij} = form factor between patches i and j (described later) A_i = area of patch i
A_j = area of patch j

---

[29] [Goral, Cindy M., Torrance, Kenneth E., Greenberg, Donald P., Battaile , Bennett. Modelling the interaction of light between diffuse surfaces]

[30] [Tettle, Paul. Radiosity in English]

In the formula above form factor needs further description.

$$F_{ij} = (\cos \theta_i * \cos \theta_j) / d^2 * H_{ij}$$

F_{ij} = form factor between patches i and j

d = Distance between the two patches

θ_I, θ_j = angles between the normal of the patch and the ray between the to patches

H_{ij} = Visibility between the two patches. If only one ray is traced between the two patches this is 0 or 1. Typically more than ray is used to get better approximations since patches are not just single points, but areas.

As you see above it is extremely expensive to do the radiosity calculations on a scene. This function is of order $O(n^3)$ where n is the number of patches in the scene. Since you for every patch will send at least one ray to every other patch in the scene, thus tracing through the scene potentially towards every polygon (it is safe to assume that the number of patches in the scene is greater than the number of polygons). The H (visibility) part of the form factor is the most expensive part to calculate and it is here we can draw usage of the strengths in a BSP-tree.

Radiosity in BSP-trees

Before the actual lightning of the scene can be calculated the surfaces has to be subdivided into patches. One idea is that the patches is of a certain size from the beginning and when the energy is calculated in that patch, it could be divided into smaller patches if the energy varies too much over the patch. Due to lack of time we discarded this idea and continued on what we thought was more important, namely using the BSP-tree to optimize the calculations.

The creation of the patches turned out to be quite a challenging problem, but it is not related to the BSP-trees nor can it draw any use of the BSP-tree, so we will not go further into that.

In the original idea of radiosity each light source in the scene should be considered as one or several patches. We chose to do it another way. Each light source is stored in the BSP-leaf it is located in. The first thing that happens is that each light sends out its energy to all patches. When this is done the radiosity calculations could be ended and the scene would look quite good. To make it look even better we used a technique called progressive refinement[31] slightly modified. In every iteration of the refinement, the patch with highest energy in each of the leaves will reflect its energy to all other patches. This will result the energy spreading from the heavily lighted patches to the patches in shadows. As in real life, where nothing is really black since everything reflects light more or less.

[31] [Nuydens, Tom. 3D Engine Column, Delphi3D]

Because of the expensive nature of the radiosity calculations we need to do some optimizations. Using the PVS that was built during the rendering of the BSP- tree when choosing which patches should receive energy can cut a lot of unnecessary calculations. The ray tracing is performed in the same way as when the PVS was calculated.

Our version of the algorithm for distribution of energy through the scene is as follows:

```
w RADIOSITY
w Indata:
w Tree – Tree to apply the radiosity in
w Outdata:
w None
w Effect:
w Sends energy between the patches in the scene.

RADIOSITY (Tree)
1 for(each leaf L in Tree)
2    for(each light S in L)
3      for(each leaf V that is in L's PVS)
4         Send S's energy to the patches in V
w At this stage we chose to do so that the level designer can at any
w point check how the scene looks and break the energy sending when
w he feels it looks good enough
5 while(not looks good enough)
6    for(each leaf L in Tree)
7      for(each leaf V that is in L's PVS)
8         Send energy from the patch with the most unsent energy in L
          to all patches in V.
```

Complexity analysis:
This is a real killer in computational cost. The worst case is that every ray has to be checked versus every polygon in the scene, which is of order $O(n^3)$ where n is the number of patches in the tree. Fortunately the optimizations we have done will in most cases reduce the cost a great deal, but it is almost impossible to say how much since it is very dependent of the structure of the tree.

This gives a very speedy lightning of the scene where the advantages of BSP-trees come in handy. Especially the work done during the ray tracing can be cut down significantly. Since the map designer can decide when the rendering of a scene is done, by breaking the loop at any time to see if the result is good enough. It is very easy to pre render the map a couple of times to se an approximate of how it will look, instead of doing a costly full render for each change.

Below is a screenshot from a sample rendering done with our radiosity algorithm:

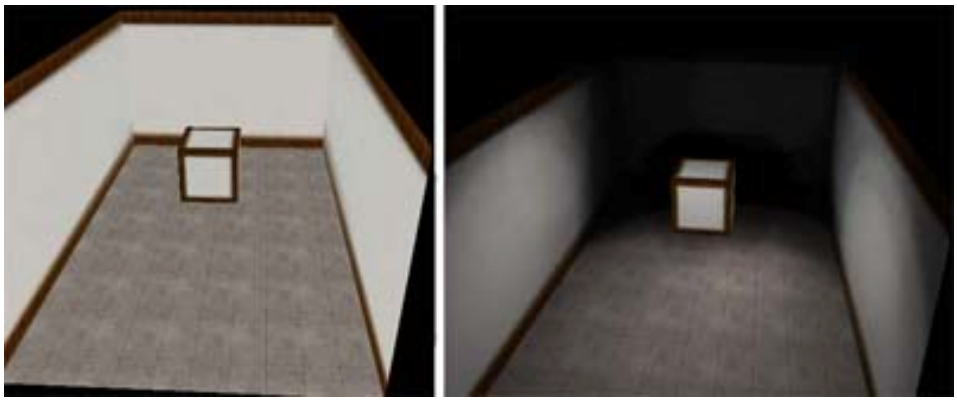


Figure 15. Sample of Radiosity.

Above is a sample of a scene rendered with our technique. The left part of the image is the unrendered version of the scene, to the right the scene has been rendered with a light approximately in front of the camera.

Related reading:

[Saykol, Ediz and Kirimer, Burak. Progressive Refinement of Radiosity]
[Teller, Seth. Application Challenges to Computational Geometry]
[Firebaugh, M. Three-Dimensional Graphics – Realistic Rendering]
[Nettle, Paul. Radiosity in English]
[Nuydens, Tom. 3D Engine Column, Delphi3D]

Chapter 5

SUMMARY OF BSP-TREE RENDERING

Now we have described the steps needed to complete the pre processing part of a BSP engine. Following is the algorithm for rendering a scene into a BSP-tree:

```
w RENDER-SCENE
w Indata:
w Scene – The scene to render as a BSP-tree
w Outdata:
w A BSP-tree
w Effect:
w Renders a BSP-tree out of the information stored in the scene.

RENDER-SCENE (Scene)
w Render the BSP-tree using the objects that describes the geometry
w of the scene
1  GeometryPolygons f {}
2  for (each object O that belongs to the geometry of Scene)
3     GeometryPolygons f GeometryPolygons U O.PolygonSet
4  GENERATE-BSP-TREE (Tree.RootNode, GeometryPolygons)
w Distribute the sample points in the leaves of the tree.
5  DISTRIBUTE-SAMPLE-POINTS (Tree.RootNode, {})
6  TRACE-VISIBILITY (Tree)
7  for each object O that is a static object in Scene
8     for each polygon P in O
9        PUSH-POLYGON (Node, P)
w CREATE-PATCHES is an undefined function that needs serious
w consideration. Our solution of this problem was not good enough, so
w we choose not to present it.
10 CREATE-PATCHES (Tree)
11 RADIOSITY (Tree)
```

Complexity

The complexity of the function calls in RENDER-SCENE is as follows:

Function	Worst Case	Typical Case	Description
GENERATE-BSP-TREE	$O(n^2 \lg n)$	$O(n^2)$	n is the number of polygons in the geometry of the scene
DISTRIBUTE-SAMPLE-POINTS	Θ (np + xy)	Θ (np + xy)	n is the number of polygons in the tree, p is the number of sample points in the tree, x and y is the widths of the dividing planes in the bounding boxes of the corresponding node.
TRACE-VISIBILITY	$O(n^2)$	$O(n \lg n)$,	n is the number of polygons in the tree
RADIOSITY	$O(n^3)$	$O(n^2 \lg n)$	n is the number of patches in the tree

The column typical case is our estimation of the general running time of that algorithm, but as we have mentioned before it varies a great deal from scene to scene. It is clear that the dominant function is RADIOSITY which leads to that the order of the whole rendering of a scene is $O(n^3)$ in worst case.

Chapter 6

PHYSICS IN BSP-TREES

One of the most intriguing problems when creating a BSP-tree based 3D-engine is collision detection. It is not as hard to solve as to do it fast. In the vast majority of FPS games most of the processor time is consumed when doing collision detection. Consider an object or avatar[32] (avatars is considered as an object from now on) that is moving through the world. It has to be checked against the geometry and all other objects in the world to see that it does not pass through or get too close to any of them. For one avatar or object this can be done with a slow algorithm at an acceptable frame rate. The problem gets quite more complex when several objects and avatars have to be handled each frame. The rendering of the world to the screen has to be done only once each frame, whilst collision detection might need to be done hundreds of times each frame, depending on the number of objects currently moving in the world. So it is of utter most importance that the algorithm used is very fast.

There are several decisions that are needed to make before starting to design an algorithm to handle collisions. The objects must be encapsulated in one or more simple geometric shapes, since it is not be possible to check every single polygon in an object for collisions against everything and still have an acceptable frame rate. I chose to encapsulate each object in an ellipsoid, with one collision radius in the xz-plane and one collision radius in the y-axis. If several shapes are chosen there is a need of making them interact with each other, which is quite a complex problem. Below is an example of how to encapsulate a human.

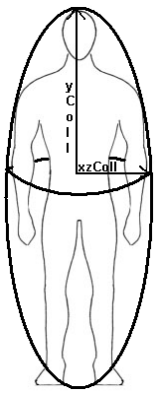


Figure 16. A human encapsulated in an ellipsoid.

[32] See the glossary for description.

Then you need to decide what should happen when an object collides with something. One variant is that if a collision is detected the move is prohibited and the object will stay in the original position, this will give a very bad behavior such as bouncing against the walls. In the figure below two moves of equal length are shown, the move to position a will be prohibited and the object will remain in the same position, but the move to position b will take place. This means that you can get closer to walls if you move along them, while a move straight towards a wall will be prohibited much earlier. Hence objects will bounce against the walls.

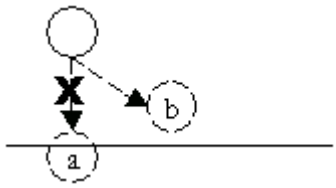


Figure 17. Prohibited and allowed move.

In the figure above the move to position a will not be allowed, but the move to b is allowed. Another drawback with this idea is that the walls will behave as if they have "glue" on them, since the objects will get stuck to them when trying to move along them.

A better way to do it is to let the objects slide against walls and other objects. This will be a less efficient way, but will give a much smoother result. This is the way I chose to do it.

Movement of an object can be divided in to three parts:[33]

1. Future Position Calculation
2. Collision Detection
3. Collision Handling

For each frame an object that moves in the world has to pass each of these steps.

[33] [Magarshak, Greg. Theory and Practice]

Future Position Calculation

The easiest part is calculating which position the object will end up in given the original location, speed, acceleration, direction, current friction coefficient and time passed during that frame. We chose to use meters and seconds as units in our calculation, further more that we chose to discard the mass of objects and assume that all objects has a mass of 1. To simplify things we assume that the gravity always is applied downwards and that a user only can apply force in the xz-plane. In our solution the following steps are taken to calculate an objects future position:

1. Deduct the speed reduction caused by friction. The friction coefficient (0.0-1.0) is subtracted from the acceleration in the xz-plane. Since the mass is assumed to be 1, no more calculations need to be done at this point.

   Formula:
   ```
   Acceleration(x,z) -= Friction
   ```

2. Add the force input from the user to the acceleration vector in x- and z-direction. Multiplying the force added by the user with the frame time and then adding the resulting vector in x- and z-direction to the acceleration vector calculate this. We consider the mass to be 1, so that the force applied is in the unit Newton/kg.

   Formula:
   ```
   Acceleration(x,z) += Force * Normalize (Direction(x,z))
   ```

3. Set gravity to the acceleration in y-direction. The traditional gravity can be used (9.82) but in games this will be quite boring, since the falls will be too fast, so it's quite common to use lower gravity.

   Formula:
   ```
   Acceleration(y) = -Gravity
   ```

4. Add the acceleration to the movement vector, this is done by adding the acceleration in x and z – direction multiplied with the frame time. It's a good idea to limit the movement speed in x- and z-direction so that the objects will reach a maximum speed at sometime, otherwise a constant input of force will lead to an infinitely accelerating object.

   Formula:
   ```
   Movement = Movement + Acceleration * FrameTime
   ```

5. Now the position modifier for this frame has to be calculated. Multiply the movement vector by the frame time and the resulting vector is the distance the object travels this frame.

   Formula:
   ```
   Distance=Movement * FrameTime
   ```

6. The desired new position can now be calculated. Add the distance traveled to the current position and the result is the position the object will end up in this frame if it passes all collision checks.

   Formula:
   ```
   NewPosition = Position + Distance
   ```

Now it is time to check if the desired position is a valid position.

Collision Detection and Collision Handling

This is one of the trickiest parts in a BSP-tree engine. There are several aspects to consider when designing this step in a physics engine. First of all the efficiency is of great importance, since this is the part that hogs most of the processor time. Secondly the accuracy of the detection is of great concern, it is hard to get good accuracy without having to reduce the efficiency of the algorithm. So there is a trade-off between correctness and efficiency. Collision handling is not as difficult though but it is of great concern, since this is what will give the right "feeling", a poor and jumpy collision handling will lower the overall appearance of any game.

This is where BSP-trees show their strength. Most engines do not use the BSP-tree to speed up the drawing, such as portal engines, but still almost all of them have built a BSP-tree out of the geometry. This is because of the great advantage when calculating the collision detection, namely that it is very cheap to position the user in the BSP-tree. When that is done, the only polygons needed to be checked; are the polygons in the leaves the object passed through that frame.

One of the other strengths of the BSP-tree is that each leaf contains a convex set of polygons in the original design of BSP-trees. This means that the polygons can be tried for collision in any order. If the sets are not convex, as in our BSP-tree, the polygons have to be checked against in order of facing. We use a term called facing value to describe the value that tells how a polygon is directed compared to an object; this is calculated by taking the dot product of the normalized movement vector for the object and the normal for the polygon, which will return a value between –1 and 1. Where –1 means that the polygon is facing straight towards the movement direction of the object and 1 means that it is facing in the same direction as the object is moving.

The order of testing is; first test the polygon that has the lowest facing value, and then the polygons are checked in order up to the polygon with the highest facing value. Below is a set of figures that show why the polygons must be taken in that order.

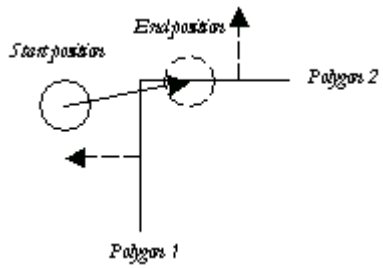


Figure 18. Object move.

In the figure above the facing value is lower for polygon 1, since the dot product between the objects movement vector and polygon 1´s normal is less than the same value for polygon 2. So if we were to check collisions versus polygon 2 first, the result would look as follows.

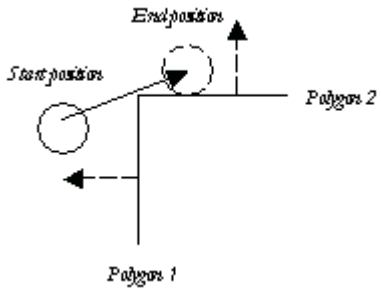


Figure 19. Incorrect collision.

In the figure above the object avoided collision with polygon 2. The end position has been corrected so that the object does not collide with polygon 2. The feeling of the result would be that the object passed through polygon 1.If the check for collision was made versus polygon 1 first the result would have looked like this:

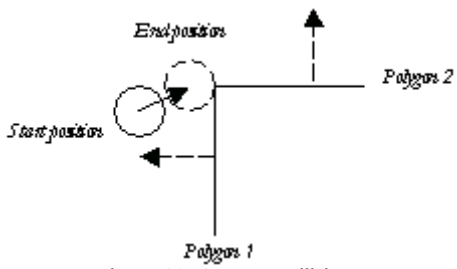


Figure 20. Correct collision

In Figure 20 the movement has been corrected more accurately, since the object avoided collision with polygon 1 first. The end position has been corrected so that the object does not collide with polygon 1.

When the BSP-tree contains only convex set of polygons there is no need to sort the polygons in the same node, but still the nodes have to be traversed in the same order as they were passed through.

In order to make a fast collision detection algorithm, a cheap test that can discard a large number of polygons from further testing is needed. This is done by calculating so called side values. The side value is a value for the distance between the objects center and the plane that the polygon is in; see the figure for better description.

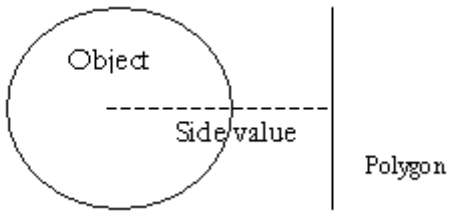


Figure 21. Side value.

In Figure 21 the length of the dotted line is the so-called side value. The further away the object is from the polygon, the greater the side value is.

When the side values are calculated, it can be decided whether an object could have passed through the polygon. To calculate a side value the objects position is put into the plane equation of the polygon, but the distance value in the plane equation is subtracted instead of added. In this way we will have a value that is the distance between the object and the plane. By calculating the side value for the start position and the end position we can easily decide whether the object passed the plane or not.

There is one problem though; since we chose to represent objects with a bounding ellipsoid the collision radius of the object in the polygon's normal's direction has to be calculated. To do this, the x - and z - component of the polygons normal is multiplied with the xz – collision radius of the object and the y– component of the polygons normal is multiplied with the y– collision radius of the object. The length of the resulting vector is the collision radius for the object versus that polygon. Following is a figure and a formula to better describe the collision radius.

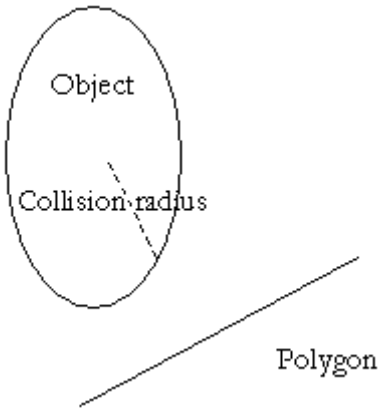


Figure 22. Collision radius.

In the figure above the dotted line represents the effective collision radius of the object towards the polygon. Observe that the line is perpendicular to the polygon.

```
CollisionRadius(Object, Polygon) =
Sqrt((Object.xzColl * Polygon.Normal.x)^2 +
     (Object.yColl * Polygon.Normal.y)^2 +
     (Object.xzColl * Polygon.Normal.z)^2)
```

Now we can decide whether the object actually passed through the plane. Following is the algorithm to perform this test, together with a helper function to calculate the collision radius for an object in a given direction:

```
w CALCULATE-COLLISIONRADIUS
w Indata:
w Object – The object to get the collision radius for.
w Direction – The direction to calculate the collision radius in.
w Outdata:
w The collision radius of the object in the given direction.
w Effect:
w Calculates the objects collision radius in the given direction.

CALCULATE-COLLISIONRADIUS (Object, Direction)
1 return Sqrt((Object.xzColl * Direction.x)^2 +
              (Object.yColl * Direction.y)^2 +
              (Object.xzColl * Direction.z)^2)
```

```
w PRE-CHECK-COLLISION
w Indata:
w Object – The object to check collision for
w Polygon – The polygon to check the object towards
w Outdata:
w Whether the object passed through the plane defined by the polygon
w Effect:
w Checks if the object passed trough the plane defined by the
w polygon.

PRE-CHECK-COLLISION (Object, Polygon)
w Calculate the effective collision radius of the object towards
w this polygon.
1 CollisionRadius f CALCULATE-COLLISIONRADIUS (Object,
                                                Polygon.Normal)

w Calculate distance between the plane and the objects start
w position.
2 StartSide f Object.StartPosition.x * Polygon.Normal.x +
              Object.StartPosition.y * Polygon.Normal.y +
              Object.StartPosition.z * Polygon.Normal.z -
               Polygon.Distance

w Calculate distance between the plane and the objects end
w position.
3 EndSide f Object.EndPosition.x * Polygon.Normal.x +
            Object.EndPosition.y * Polygon.Normal.y +
            Object.EndPosition.z * Polygon.Normal.z -
             Polygon.Distance

w If the two points is on different sides of the plane, multiplying
w the two values will give a negative result, indicating that the
w object passed through the plane.
4 if ((StartSide – CollisionRadius) *
      (EndSide – CollisionRadius) < 0)
5    return true
6 else
7    return false
```

PRE-CHECK-COLLISION determines whether an object passed through the plane defined by a polygon, but it does not consider the boundaries of the polygon. We need further testing for the polygons that passed this test, namely a test that determines if the object passes within the boundaries of the polygon. This test is quite expensive so the more polygons that are discarded in the earlier stage the better. First we have to check if the polygon is inside the cylinder created by the object's move. See Figure 23.

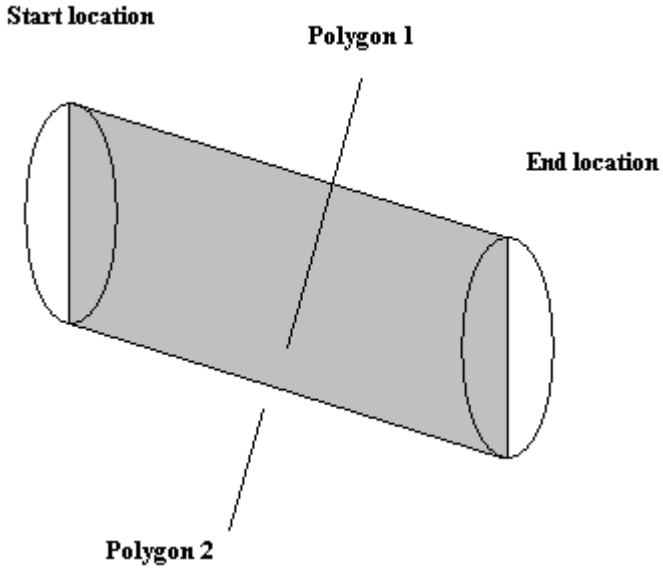


Figure 23. Testing if a object passes through a polygon.

In the figure above the object moves from the start location to the end location. Both locations are on different sides of both polygons. But the cylinder (the gray area in the figure) created by the move of the object only passes through polygon 1.

In order to perform this test, we need to calculate perpendicular planes for the polygon. They can be calculated once and for all when a polygon is created to save time. Perpendicular planes are planes with a normal perpendicular to the normal of the polygon and facing inwards. There are three perpendicular planes for a triangle, one for each edge. We have illustrated how it looks in the figure below.

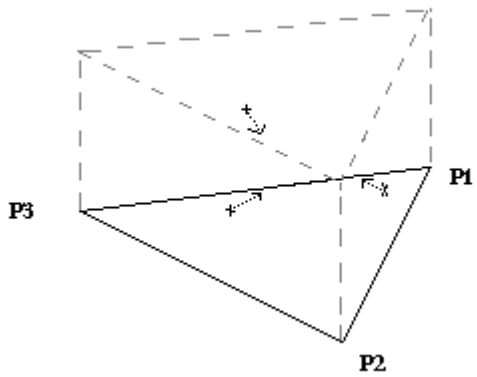


Figure 24. Perpendicular planes for a triangle.

Each perpendicular plane is calculated by the cross product of the direction vector of the edge it is aligned on and the normal of the polygon, and normalizing the resulting vector. If the result gives an outward facing normal, it is inverted. The distance of the perpendicular planes is calculated by the dot product between the normal of the perpendicular plane and one of the vertices on the edge the plane is aligned on. For example the perpendicular plane between p1 and p2 in the figure is calculated as follows.

```
1. PerPlane.Normal = Normalize (Direction (p1, p2) x Polygon.Normal)
2. if(Perplane.Normal * p3 < 0) invert(PerPlane.Normal)
3. PerPlane.Distance = PerPlane.Normal*p2
```

If the object were a single a point we would only need to check it is on the positive side of each of the perpendicular planes, thus indicating that the object is within the polygon. But since the object has a collision radius we cannot just take the planes that are along the edges, we need to move them outwards a bit. So each of the perpendicular planes are moved a collision radius from the center of the polygon, calculated by deducting the collision radius from perpendicular plane's distance value. This will still leave us one problem, illustrated in the following figure:

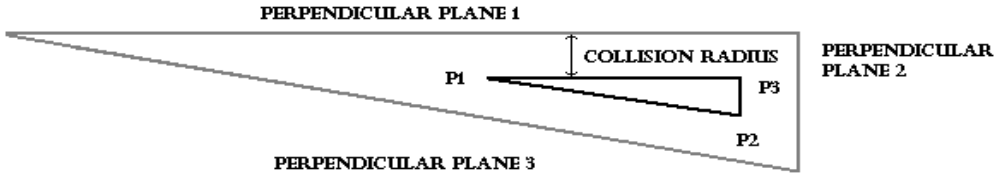


Figure 25. Moved perpendicular planes.

Obviously the planes are encapsulating a too big area. The distance between p1 and the intersection of perpendicular plane 1 and 3 is too long. The same is true for each of the other corners. To correct this the objects position needs too be checked so that it inside three more perpendicular planes. Let us copy perpendicular plane 2 in the figure above and move it to p1. Then move it the collision radius of the object further away, and invert the normal of that plane. That corrects the problems. This is done for each the vertices opposing plane. The figure would look like this then:

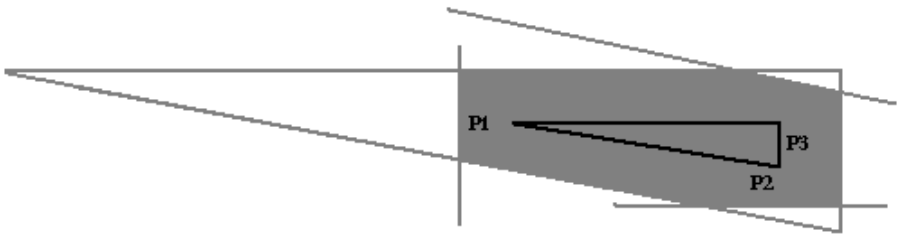


Figure 26. The effective collision area of a triangle.

The gray area in the figure above is the area in which an object will be considered as inside of the polygon. Even this area is not exactly correct, where as the corners will be a bit too far away, but the result is good enough. Using the exact area would be much too expensive since we would have to use an infinite number of planes in the corners of the area. Below is figure that illustrates the correct appearance of the area.

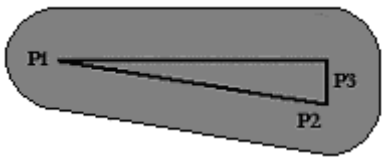


Figure 27. The correct collision area of a triangle.

Now we can detect whether an object collides with a polygon or not. The algorithm for doing this can shortly be put down as this:

```
w GET-COLLIDING-POLYGON
w Indata:
w Object – The object to check collisions for
w PolygonSet – The polygons to check collisions versus
w Outdata:
w The polygon that the object collides with.
w Effect:
w Loops through all polygons in the polygon set and checks if the
w object's movement is obstructed by a polygon or not.

GET-COLLIDING-POLYGON (Object, PolygonSet)
1 for each polygon P in PolygonSet in order of increasing facing
      value

w If the object passes through the plane defined by the polygon,
w  further testing is needed.
2    if (PRE-CHECK-COLLISION (Object, P))

w Test each of the six perpendicular planes.
3       for each perpendicular plane Pl in P

w The effective collision radius the object towards the plane is
w the width of the object in the direction of the planes normal.
4          CollisionRadius  f
            CALCULATE-COLLISIONRADIUS (Object, Pl.Normal)

w Move the plane backwards the collision radius of the object.
5          Pl.Distance  f  Pl.Distance - CollisionRadius

w Calculate distance between the plane and the objects
6          Side  f  Object.Position.x * Pl.Normal.x +
                    Object.Position.y * Pl.Normal.y +
                    Object.Position.z * Pl.Normal.z -
                     Pl.Distance

w If the object is on the negative side of this plane, the object is
w not within the polygon and we can skip further testing on this
w polygon.
7          if ( Side < 0 )
8             goto step 1

w If we reach this point the object was within all perpendicular
w planes and we can return this polygon as the polygon the object
w collided with.
9       return P

w We couldn't find a polygon that the object collide with so we
w return no polygon.
10 return NOPOLYGON
```

Complexity analysis:

Since this is a very frequently used function during run time it is of utter most importance that it is effective. As it is now it is not effective enough, the order of the function is dominated by the sorting of the polygons needed on line 1, using an effective sorting algorithm such as quick sort, the order will be O(n lg n), where n is the number of

polygons in the incoming set. We need to be closer n. We will have to re-write this function later on.

Every frame GET-COLLIDING-POLYGON will be called until there is no colliding polygon left in every node the object passes through.

Now that we can detect whether an object collides with a polygon or not, we need to handle objects colliding with objects. This is a much simpler task and it can be done with just a few calculations. First a direction vector between the centers of the two objects is calculated and normalized. Then the collision radius for both objects is calculated in that direction, in the same way as it was calculated in the polygon collision case. If the distance between the two object's centers is less than the sum of the two collision radii the objects collide and collision handling needs to be done. Below is an algorithm to determine if two objects collide.

```
w OBJECTS-COLLIDE
w Indata:
w Object1 – The object that moves.
w Object2 – The object to check collision towards
w Outdata:
w Whether the first object collide with the second object or not.
w Effect:
w Checks if the two objects collide.

OBJECTS-COLLIDE (Object1, Object2)

w Calculate the direction vector between the two objects
1 Direction f GET-DIRECTION (Object1.EndPosition, Object2.Position)

w Calculate the both collision radius in that direction.
2 CR1 f CALCULATE-COLLISIONRADIUS(Object1, Direction)
3 CR2 f CALCULATE-COLLISIONRADIUS(Object2, Direction)

w Calculate the distance between the two objects
4 Distance f GET-DISTANCE (Object1.EndPosition, Object2.Position)
5 if (Distance < CR1 + CR2)
6    return true
5 else
6    return false
```

OBJECTS-COLLIDE will be called once for every object in the nodes the moving object passes through.

When collision between an object and a polygon or an object and another object is detected, the end position of the moving object needs to be corrected.
In the case of collision versus a polygon, the corrected end position that is calculated with the following formula:

```
EndPosition += Polygon.Normal*(CollRadius-EndSideValue)
```

The effect of this formula will be that objects "slide" against the walls as opposite to get stuck against walls which would be the case if the end position was set to the start position every time a collision was detected.

In the case of collision between two objects the end position for the moving object is corrected according to the following formula:

```
EndPosition += Direction * (CollRadius1 + (CollRadius2-Distance))
```

In the above formula, Direction is the direction between the moving object's end position and the other object's current position, CollRadius1 is the moving object's collision radius in that direction, while CollRadius2 is the other object's collision radius in the direction and Distance is the distance between the end position of the moving object and the other object's current position.

When the position is corrected it might be that the object is moved so that it collides with a previously passed polygon or object. So each time a collision is detected the collision detection needs to be restarted from the beginning. This can become very expensive in complex environments, so it is recommended to put some upper limit for the number of iterations. When that number of iteration has passed and collisions are still detected the end position will be set to the object's start position.

Since sorting the polygons by facing value every time a node is checked for collision would take much too long time, another solution is better. In every iteration: loop through all polygons and remember the one with the lowest facing value that was in the way for the moving object. If a collision was detected when all polygons in a leaf has been checked against, collision handling will be done towards that polygon. Then the loop will be restarted from the beginning. This will give that the collisions will be taken in order of facing as was mentioned earlier in this chapter. So our re-written GET-COLLIDING-POLYGON will look like this:

```
w GET-COLLIDING-POLYGON
w Indata:
w Object – The object to check collisions for
w PolygonSet – The polygons to check collisions versus
w Outdata:
w The polygon that the object collides with.
w Effect:
w Loops through all polygons in the polygon set and checks if the
w object's movement is obstructed by a polygon or not.

GET-COLLIDING-POLYGON (Object, PolygonSet)
1 LowestFacing f INFINITY
2 CollidingPolygon f NOPOLYGON
3 for each polygon P in PolygonSet

w If the object passes through the plane defined by the polygon,
w further testing is needed.
4    if (PRE-CHECK-COLLISION (Object, P))

w Test each of the six perpendicular planes.
5       for each perpendicular plane Pl in P

w The effective collision radius the object towards the plane is
w the width of the object in the direction of the planes normal.
6          CollisionRadius f
            CALCULATE-COLLISIONRADIUS(Object, Pl.Normal)

w Move the plane backwards the collision radius of the object.
7          Pl.Distance f Pl.Distance - CollisionRadius

w Calculate distance between the plane and the objects
8          Side f Object.Position.x * Pl.Normal.x +
                  Object.Position.y * Pl.Normal.y +
                  Object.Position.z * Pl.Normal.z -
                   Pl.Distance

w If the object is on the negative side of this plane, the object is
w not within the polygon and we can skip further testing on this
w polygon.
9          if ( Side < 0 )
10            goto step 1

w If we reach this point the object was within all perpendicular
w planes, so if this polygon has lower facing value than the lowest
w facing value this far, we will remember this polygon.
11      FacingValue f P.Normal.x * Object.MovementDirection.x +
                      P.Normal.y * Object.MovementDirection.y +
                      P.Normal.z * Object.MovementDirection.z
12      if (FacingValue < LowestFacing)
13         CollidingPolygon f P
14         LowestFacing f FacingValue
w Return the remembered polygon, might be no polygon if no colliding
w polygon was found.
14 return CollidingPolygon
```

Complexity analysis:
Now we only loop through all polygons once, so the order of this algorithm is O(n)), where n is the number of polygons in the incoming set..

In our solution we set a maximum of 5 iterations for the polygons, but of course this is very dependent of how complex the scene is, where a more complex scene could require more iterations. Following on the next page is the collision loop that is done once every frame for every moving object.

```
w COLLISION-HANDLING
w Indata:
w Object – The moving object
w Outdata:
w None
w Effect:
w Checks the objects movement versus every polygon and object in the
w nodes the object passes through. When collision is detected, it
w will be handled.

COLLISION-HANDLING (Object)
1 PolygonSet f {}
2 for each node N the object passes through
3    PolygonSet f PolygonSet U N.PolygonSet
4 Iterations f 0
5 while ( Iterations <= MAXITERATIONS)
6    Polygon f GET-COLLIDING-POLYGON (Object, N)
7    if (Polygon f NOPOLYGON)
8       goto step 14
9    CollisionRadius f
          CALCULATE-COLLISIONRADIUS(Object, Polygon.Normal)
10   EndSide f Object.EndPosition.x * Polygon.Normal.x +
               Object.EndPosition.y * Polygon.Normal.y +
               Object.EndPosition.z * Polygon.Normal.z -
                Polygon.Distance

w Move the end position of the object so that it will not collide.
11   Object.EndPosition f Object.EndPosition + Polygon.Normal *
                          (CollisionRadius - EndSide)
12   Iterations f Iterations + 1

w The only way to reach this step is if there were more collisions
w than then maximum number of iterations. This move is considered
w illegal so we set the end position to be the original position.
13 Object.EndPosition f Object.StartPosition
14 for each object O in the nodes the object passes through
15   if (OBJECTS-COLLIDE (Object, 0))

w Calculate the direction vector between the two objects
16      Direction f GET-DIRECTION (Object1.EndPosition,
                                   Object2.Position)

w Calculate the both collision radius in that direction.
17      CR1 f CALCULATE-COLLISIONRADIUS (Object1, Direction)
18      CR2 f CALCULATE-COLLISIONRADIUS (Object2, Direction)

w Move the end position of the object so that it will not collide.
19      Object.EndPosition f Object.EndPosition + Direction *
                            (CollRadius1 + (CollRadius2-Distance))
```

Complexity analysis:
The collision loop is of order O(i n), where n is the number of polygons in the nodes the object passes through and i is the maximal number of iterations (which could vary).

There is at least one obvious optimization that is very easy to implement, that is to only calculate the collision radii once per polygon-object and object-object pair. But we chose to present this way, unoptimized, out of clarity reasons. Probably there are some more obvious things that can speed up the process, but we leave them to you.

Related reading:
[Nuydens, Tom. 3D Engine Column, Delphi3D]
[Magarshak, Greg. Theory and Practice]
[Lin, Ming C. Fast Collision Detection for Interactive games]
[UNC Collide Research Group, Collision Detection]
[Bikker, Jacco. Building a 3D Portal Engine]

Chapter 7

NETWORK OPTIMIZATION USING BSP-TREES

Today we have passed the limit where the computers processing and graphical ability can be considered as a great limitation. Instead networking is where the troubles exist, as many users still are connected with modems. To be able to reach as many people as possible, multiplayer games must be designed to run on a modem connection. To illustrate why this can be a problem, consider the following problem. We have a multiplayer game that runs on a server designed to take care of 15 clients[34]. Let's say that the server updates the world 20 times per second (called ticks[35]). This means that if all clients were to get information about all other players every update there would be a quite substantial amount of data to transfer to each client every second. To be able to count how much a client will receive each frame we need to know how much information about each user that is sent every frame. If we consider the minimum case there is a vector of movement, a position vector, and a rotation vector that each consists of 3 floats, which take 4 bytes of memory. Then we would probably have some packet overhead, let's say six byte. Then every packet would be 3*3*4+6= 42 byte. Now every client will have to get information about every other client 20 times per second. Giving us that every client will receive 20*15*42 = 12600 bytes every second. Since a 55.6 modem only can receive 55600/8 = 6950 bytes/second at optimal conditions (which never happens) this will clearly overflow all clients that are using modems. Not to mention the bandwidth needed on the server. It is quite obvious that these numbers need to be cut down quite a bit.

Again the structure of the BSP-tree comes in handy. As with drawing, where the principle is "What is not seen, is not drawn", the network can be optimized by only sending information about the visible objects to the user. This can be done with a portal engine, by sending information about the objects that are visible to each user. When you have a static PVS you just send the objects that are in the visible leaves from the leaf the user is currently occupying. In good maps this will reduce the amount of data sent significantly.

Related reading:
[Sweeney, Tim. Unreal Networking Architecture]

---

[34] See the glossary for description.

[35] [Sweeney, Tim.Unreal Networking Architecture]

Chapter 8

FUTURE WORK

There are millions of things that can be done to improve the solutions presented in this report. Some examples are improving the collision detection algorithms, better removal of none visible objects and the creation of a portal engine with a static PVS. Other things are not even mentioned in this report such as prediction, which have not even been implemented.

Even though the collision detection algorithms provided in this report is giving satisfactory performance, there is still some things that can be done better. Perhaps a box is a better bounding form for an object. The drawback with a bounding box is that there are much more calculations needed than with a bounding sphere, but the visible result would be better.

Objects are often more complex than the geometry of the world, i.e. consists of more polygons. Hence, it would be good to remove as many non-visible objects as possible. In our solution all objects that are in a visible leaf are drawn. If a fast algorithm could be developed to remove hidden objects, it would improve performance. It is very tricky though to create an algorithm to remove objects that is cheaper than to draw them to the screen.

If you create a portal engine with a static PVS, you would get all the benefits from a portal engine such as mirrors and easy removal of objects, but you could draw usage of the strengths of a static PVS, meaning that it is cheap to find out which sectors to draw and cheap lightning of the world.

Prediction is an important thing in multiplayer games. The goal is that the clients should have as correct image of the scene as possible, i.e. not differ too much from the server. In our solution the objects are simply put on the position received from the server. This results in that if it takes 200 ms for the data to get from the server to the client, the image the client will have of the world is 200 ms seconds old. If you could predict where the object is now or at least at a later stage than the server data, based on previous movement of the object, the client would get a much more accurate view of the world.

Chapter 9

CONCLUSIONS

BSP-trees are very useful structures that have many advantages when it comes to creating a 3D-engine. Even though the original purpose of usage, i.e. sorting the polygons to be able to draw them in correct order onto the screen, is obsolete, many areas of usage remain, such as faster collision detection, removal of hidden surfaces and network optimisation. There is still space for much improvement in the area though. The following is some of the advantages and disadvantages with BSP-trees.

Advantages

- Fast collision detection, a big part of the map can be discarded easily since it is cheap to position the object into a leaf. When that is done only the polygons in that leaf is needed to check for collision.
- With a PVS it is very easy to remove not visible parts of the map.
- Can be used to optimise networking.
- Can be used to optimise lightning calculations of the map.

Disadvantages

- BSP trees are really only suitable for static worlds. It is possible to add and remove polygons in the world, but to do this you need to recompute part of the BSP tree. Using local BSP trees, which are intersected, with the main BSP tree is also a possible optimization technique, but the fact remains that BSP trees are better suited for static worlds.[36]
- To make the most efficient use of a BSP tree, you still need to add a PVS (Potentially Visible Set) or other similar techniques. If you don't, you will probably end up considering too many polygons (especially with large worlds). [29]
- The BSP tree technique is rather complicated. [29]

---

[36] [Tyberghein, Jorrit. The Portal Technique for Real-time 3D Engines]

The BSP-trees will probably still exist in the gaming industry the next five to ten years, but hopefully somebody will invent a smarter more dynamic structure with the same advantages as the BSP-trees have. One of the biggest problems with BSP-trees is the complexity of them. There are too many parts needed to make everything work efficiently, so an ideal solution must be much more simple and intuitive.

An alternative to BSP-trees could be some kind of scene-graph, where everything is an object that derives from the same parent. Each object knows in which object it is located. All objects know how they should be displayed, how collision takes place inside of them, which neighbors they have, where you can see out of them. This would make it much easier to insert and remove parts of the world without having to re-render the whole world. It would also be a much more elegant way to solve the complexity that occurs with BSP-trees, since no object type needs to know anything about any other object types. They all implement the same interface and the only communication between the objects would go through that interface.

APPENDIX

Here are some screenshots from the product released with the developed technology:

BIBLIOGRAPHY

Feldman, Mark. Introduction to Binary Space Partioning Trees, http://www.geocities.com/SiliconValley/2151/bsp.html, 1997.

Unknown Author, Basic Math FAQ, http://www.cc.gatech.edu/gvu/multimedia/nsfmmedia/graphics/elabor/math/mathfaq.html

Hoff III, Kenneth E. Faster 3D Game Graphics by Not Drawing What Is Not Seen, http://www.cs.unc.edu/~hoff/papers/vfc/vfc.html, ACM Crossroads, 1997.

Tyberghein, Jorrit. The Portal Technique for Real-time 3D Engines, http://crystal.linuxgames.com/docs/portal.html, 1998.

Bikker, Jacco. Building a 3D Portal Engine, http://www.flipcode.com/portal/, 1999.

Royer, Dan. Dan's Programming Tutorials, http://members.home.com/droyer/tutorials/, 1997-1999.

Åhs, Cons T and Bevemyr, Johan. Inlämningsuppgift I Programmeringsmetodik 2, 2000.

Nettle, Paul. Radiosity in English, http://www.flipcode.com/tutorials/tut_rad.htm,1999.

Nuydens, Tom. 3D Engine Column, Delphi3D, http://www.gamedeveloper.org/delphi3d/3de.shtml, 2000.

Firebaugh, M. Three-Dimensional Graphics – Realistic Rendering, http://www.uwp.edu/academic/computer.science/morris.csci/CS.320/Week.10/Ch10.html.

Teller, Seth. Application Challenges to Computational Geometry, http://graphics.lcs.mit.edu/~seth/pubs/taskforce/paragraph3_3_0_0_1.html, 1996.

Saykol, Ediz and Kirimer, Burak. Progressive Refinement of Radiosity, http://www.ug.bcc.bilkent.edu.tr/~saykol/radiosity/sld001.htm.

Mr. Meanie. Binary Space Partitioning Trees, http://easyweb.easynet.co.uk/~mrmeanie/bsp/bsp.htm.

Chalfin, Alex. Cells and Portals, http://www.netmagic.net/~achalfin/Graphics/portal.htm, 2000.

Magarshak, Greg. Theory and Practice, http://www.flipcode.com/tpractice/, 2000.

Cormen, Thomas H. Leiserson, Charles E. and Rivest, Ronald L.: Introduction to Algorithms (MIT Press, 1990)

Hoff, Kenny. The Warnock Area Subdivision Algorithm for Hidden Surface Removal, http://www.cs.unc.edu/~hoff/techrep/warnock.html, 1996.

Southwick, Andrew R. Quake Rendering Tutorial, http://www.quakerant.com/qw/rendering.html, 1998.

Warren,Mike. Useful Algorithms, http://www.quakeworld.com/mikebot/useful-algorithms.html, 1998.

Lin, Ming C. Fast Collision Detection for Interactive games, Game Developer Conference 1999, Conference Proceedings, 1999.

UNC Collide Research Group, Collision Detection, http://www.cs.unc.edu/~geom/collide/index.shtml, 2000.

Sobey, Anthony. Software Engineering, http://louisa.levels.unisa.edu.au/1996/se/project/proj_bak.htm - HD_NM_48, 1996.

Sunsted, Tod. 3D computer graphics: Moving from wire-frame drawings to solid, shaded models, http://www.javaworld.com/javaworld/jw-07-1997/jw-07-howto.html, 2001.

Sweeney, Tim.Unreal Networking Architecture, http://unreal.epicgames.com/Network.htm,1999.

Goral, Cindy M., Torrance, Kenneth E., Greenberg, Donald P., Battaile , Bennett. Modelling the interaction of light between diffuse surfaces, Computer Graphics (ACM SIGGRAPH '84 Proceedings, 1984.

Shumacker, R., Brand, R., Gilliland, M., Sharp, W. Study for Applying Computer-Generated Images to Visual Simulation. AFHRL-TR-69-14. U.S. Air Force Human Resources Laboratory, 1969.

Silicon Graphics. BSP Tree Frequently Asked Questions (FAQ), http://reality.sgi.com.com/cgi-bin/bspfaq/bsp?8.txt, 1997.

Luebke, David P., Georges, Chris. Portals and Mirrors, http://www.cs.virginia.edu/~luebke/publications/portals.html,1998.

Bittner, Jiri, Havran, Vlastimil, Slavik, Pavel. Hierarchical Visibility Culling with Occlusion Trees, http://sgi.felk.cvut.cz/~bittner/publications/cgi98-copy.pdf, 1998.

Zielinski, Michal. Radiosity: physical bases of this computer graphics method, http://www.phys.uni.torun.pl/~mzielin/radio_htm.html, 2000.