Fast Rendering Using Front-to-back Binary Space Partitioning Trees

Neelam Aggarwal, Karthik Ginuga, Ramesh Manuvinakurike, Vinod Sharma
neelamag@usc.edu, ginuga@usc.edu, manuvina@usc.edu, vinodsha@usc.edu

Abstract

An efficient rendering algorithm a.k.a front-to-back, is developed using front-to-back traversal method of the BSP(Binary Space Partition) tree in this project. Front-to-back rendering algorithm performs much faster than the rendering algorithm which uses back-to-front traversal method of the BSP tree, a.k.a back-to-front. Front-to-back rendering algorithm also performs faster than the rendering algorithm a.k.a Z-buffer which uses Z-buffer for hidden surface removal. In this project we mainly focus on comparing front-to-back and back-to-front rendering algorithms but we also compare them with Z-buffer rendering algorithm.

1.0 Introduction

Real-time systems that can generate large 3-D environments efficiently has always been a goal of 3D graphics world[1][2]. When the number of polygons to be rendered are large, the scanline and LEE method used for rendering with z-buffer (for hidden surface removal) are time consuming. These algorithms go through every polygon of the model to be rendered. Every pixel lying within the polygon are rendered after checking for the z-buffer values of the previously rendered polygon. This would mean that the same pixel values are rewritten or checked multiple times. BSP trees(Binary Space Partition) have an embodied geometrical priorities which helps eliminate this redundancy[1] when used with appropriate data-structures.
The BSP trees help with hidden surface removal and generating many images with different viewpoints in fixed environments. In this work we show the advantages of BSP trees and compare the methods of traversal of BSP trees. The process of creating BSP trees is by repeatedly dividing the subspace using a hyperplane. The polygons lying to one side of the hyperplane are represented in a branch and the polygons lying on the other side on the other side of the tree. The detailed algorithm used to construct the BSP trees are shown in the later sections.

2.0 Related Work

The current work is an implementation of the algorithm discussed by Gordon et.al [1]. They[1] implemented the front-to back algorithm and have successfully shown that the front-to-back traversal of the BSP tree is more efficient than the back-to-front traversal. The key to their implementation is the Dynamic screen data structure, Edge tables and the merge operations. The dynamic screen data structure used are motivated by the works of Reynolds et.al [3]. The traversal of the BSP trees discussed are of two types: a. Front-to-back b. Back-to-front. In the back to front display of the polygons are traversed chronologically from the polygons which are farthest from the viewer to the polygons which are closest. In, front-to-back traversal the polygons are traversed from the closest to the farthest from the viewer in the viewing direction.

3.0 Methods

The overall architecture of our system is shown in figure 1. Our experiments will be conducted mainly for four different cases: small polygons, medium polygons, large polygons and Number of polygons covering each pixel.

```
For each case in (small, medium, large polygons, Number of polygons covering each pixel):
        1. Generate polygons using polygon generator module
        2. Construct the BSP tree using the polygons from step 1.
        3. Use front-to-back and back-to-front rendering algorithms on the BSP tree from step 2 for comparison.
```

Table 1: Pseudocode for the experiment

The polygon generation module is largely responsible for generating polygons for the above mentioned cases. These polygons are generated procedurally and are not loaded from specific models. This gives the freedom for testing large variety of cases. The main module controls the cases and generates the specific outputs for each of the cases. The outputs are generated as graphs. The BSP tree creation module creates the tree and is responsible for polygon splitting and other tree creation logic. Rendering module traverses the tree and writes to the framebuffer.

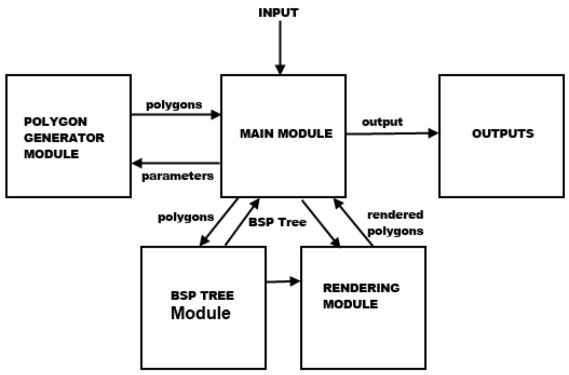


Figure 1: The architecture of the system built.

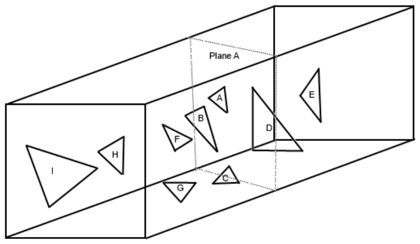
3.1 Main Module

The main module combines the the inputs from different modules. The function of the main module is to test the efficiency of the Front-to-back implementation compared to the back-to-front implementation. Either of these implementations are tested for the cases mentioned in Table 1. The parameters such as the size of polygons, number of polygons and number of polygons covered per each pixel are provided to the polygon generator. The list of polygons generated is input for the BSP tree creator module. The BSP tree returned by the BSP tree creation module is fed to the rendering module. The time taken for the implementations of back-to-front and front-to-back are recorded.

3.2 Polygon Generator

The purpose of the Polygon Generator module is to create wide variety of polygons to test the efficiency of the BSP tree creator module. Two separate methods were used for this purpose. The first method (a) takes a single parameter, “Number of triangles covered per pixel” (k) as input and outputs a list of polygons. The second method (b) takes two parameters “Number of polygons” and “Size of polygons”. The outputs for both these methods are list of polygons. The method (a) ensures that the each pixel is covered by k number of polygons. This is done by generating triangles with same xy coordinates for all vertices but with different Z depths. The method (b) generates the polygons by splitting the screen into subspaces and generating the polygons within these subspaces. The size of the subspace is determined by the size of the triangles desired for testing. For small sized polygons, the screen space is divided into 16 x 16 blocks. A polygon is then generated within the bounds of the randomly selected block. Similarly for medium and large sized polygons, the screen space is divided into 8 x 8 and 4 x 4 blocks respectively and polygons are generated within the bounds of randomly selected block.

3.3 BSP tree creator



Fast rendering using BinarySpace Partitioning Trees

Figure 2: Polygons in the 3D world. Hyperplane build using the triangle A, cuts the space into two parts.

The BSP tree creator module is responsible for creating the BSP tree which the Rendering module uses. This module is called by the Main Module only once. Once, the BSP tree is created the same BSP tree is reused for different viewer positions. The BSP tree module gets the input from the Polygon generation module. The Polygon generation module creates the polygons and the main module passes these polygons to the BSP tree creator. It is assumed that the polygons are all triangles. A random triangle is selected from the triangles passed to the BSP tree creator module. The plane in which this triangle lies forms the Hyperplane. The function of the hyperplane is to split the 3D world into two parts. This hyperplane acts as the current root of the BSP tree. The area to one side of the plane are termed positive and the other end negative. The positive or negative are named consistently depending on which side of the hyperplane the polygons lie in. It is important the positive and negative naming convention is held consistent. This is to make sure that one side of the hyperplane is consistently inserted into one side of the BSP tree. Once, the hyperplane is constructed, all the polygons are traversed and are assigned to the positive or negative branch of the tree. The positive and negative naming of the branches of the tree is same as those computed by the hyperplane. After classifying all the polygons into the positive or negative branch of the tree, a new hyperplane is selected for each of the branches and the process is repeated recursively until a tree is constructed like the one shown in Figure 4.

A few complexities arise during this process. It is possible that the hyperplane splits the triangle into two polygons. The resulting split may result in polygons of 3-4 vertices. These resulting polygons with 4 vertices are split into two intermediary triangles as shown in the figure 3. The BSP tree creator makes sure that the every leaf of the BSP tree is a triangle.

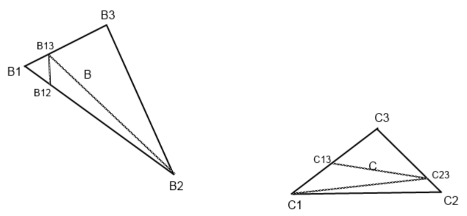


Figure 3: Shows polygon splitting by the plane. The resulting polygons are split into triangles and are added to the trees as the new polygons.

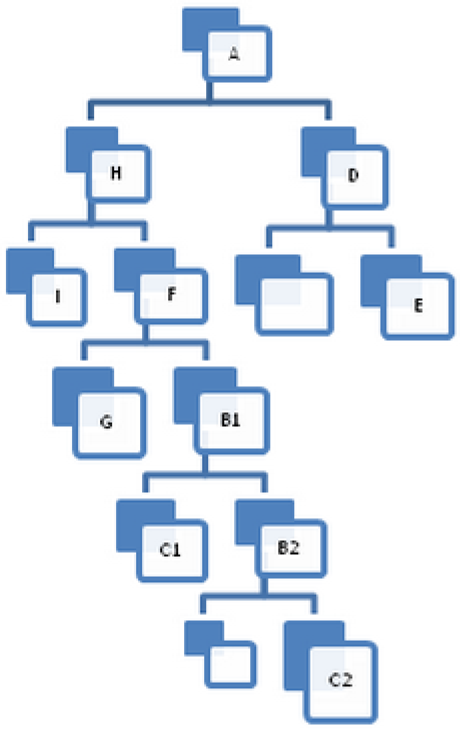


Figure 4: BSP tree representation of Figure 2

3.3.1 Triangle splitting

It is possible that the triangle is split by the hyperplane into a multiple polygons. The algorithm for splitting is shown below:

```
1. For every edge of the triangle:
2.          Find the intersection point of the plane with the edge.
3.          Record the intersection points
4. Convert Resulting polygons into triangles.
5. For each triangle
```

```
6.      If the triangle lies to the front of hyperplane:
7.              add to the front list
8.      else
9.              add to the back list
```

3.3.2 Implementation details

The BSP tree structure used is shown below:

```
struct  BSP_tree {
   Plane     partition;
   Triangle*  polygons;
   Triangle*  front_list;
   Triangle*  back_list;
   BSP_tree  *front;
   BSP_tree  *back;
   int front_index;
   int back_index;
   int total_polygons;
};
```

Plane is another struct defined by the coefficients of the equations of the plane. The Triangle struct is defined by the coordinates of the vertices of the triangle. The added integer index values of front and back trees increases the implementation time of the algorithm. These values help faster insertions of the polygons into the tree.

3.4 Rendering module

The rendering module consists of the tree traversal logic. In this work we consider two tree traversal methods namely Back-to-Front and Front-to-Back with flat-shading. The Back-to-Front method traverses the the object farthest from the viewer to the object closest to the viewer. While, the Front-to-back method traverses the objects in the order of closest polygons to the viewer to the object farthest polygons from the viewer. In Back-to-Front rendering method, each pixel is written multiple times if covered by multiple polygons, which results in poor performance. If we plainly use the Front-to-Back rendering method to write pixels, the resulting image would be incorrect, as pixel written by closer polygons will be overwritten by farther polygons. One way to correctly implement front-to-back rendering algorithm is to perform a check before a writing a pixel. If pixel is written at-least once, then avoid writing this pixel. This way of implementing front-to-back rendering algorithm does not improve performance, as we are still checking if pixel is set for all polygons covering it. Efficient implementation of front-to-back rendering method needs special data-structures to avoid checking a pixel if it's already set. The two data-structures used for this purpose are Edge tables and Dynamic Screen Data-structures (DSD). The Dynamic screen data-structure is shown below in figure 5. Each row of the DSD will hold the linked lists containing nodes and each node can be viewed as a part of row of an image that is not painted yet(image segment). The Edge tables hold the scan line pixel values of the current polygon being rendered. The merge operation modifies the DSD structure to paint the matching screen segments and remove these matching image segments from DSD. The time saving in front-to-back rendering happens due to this step. Only those pixels which are covered by the new polygons and those haven't been painted previously are painted. This avoids the overwriting of the pixel values by multiple polygons and hence, achieves time saving. An example operation is shown in figure 6. where only those pixels not covered by the earlier polygons (DSD) are printed to the screen.

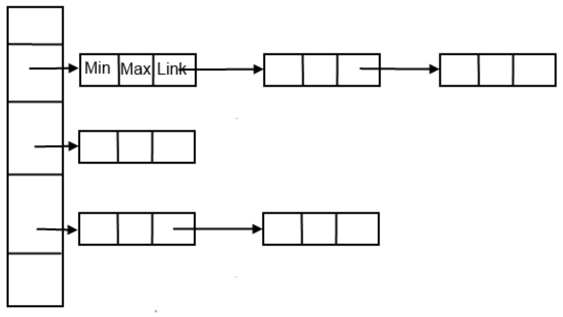


Figure 5: Dynamic Screen data-structure

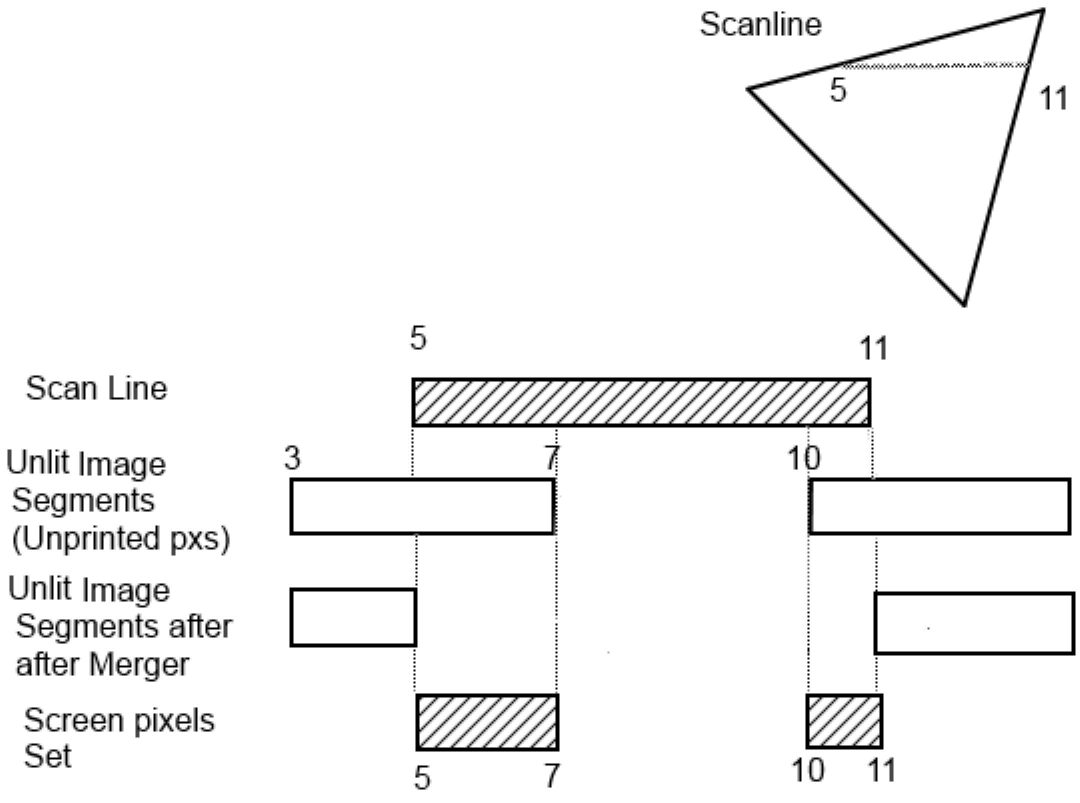
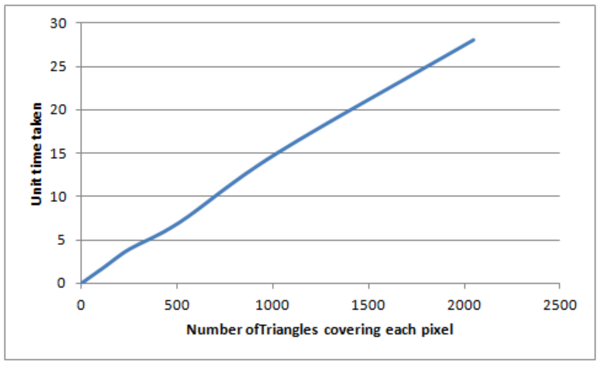

Scanline
5
11
Scan Line
5
11
Unlit Image Segments (Unprinted pxs)
3
7
10
Unlit Image Segments after after Merger
Screen pixels Set
5
7
10
11

Figure 6: Merge operation of the scanline starting from y=5 to y = 11 with the DSD to produce the new DSD and the pixels to be printed.

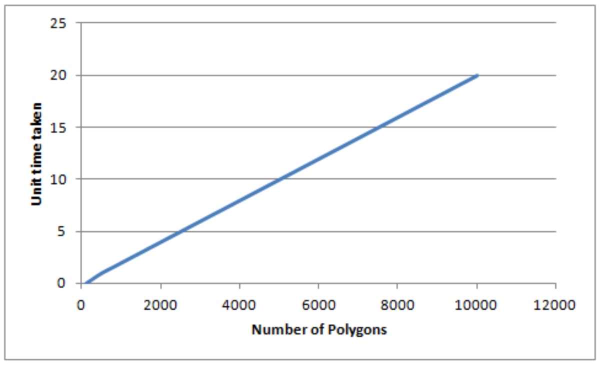
4.0 Results

In this section we analyze the performance of the Front-to-back algorithm with scanline algorithm implemented in the homeworks and with the back-to-front algorithm implemented as a part of this work. The performance of the scanline algorithm using z-buffer is shown below in figure 7. The number of polygons per pixel is the number of polygons that occupy every pixel of the screen. The other measurement parameters were the number of polygons and the size of polygons. The increase in the number of polygons per pixel increases the number of times a pixel value in the frame-buffer is overwritten. This causes the increase in the running times. A similar argument can be made for number of polygons in the model. With the increase in the number of polygons, more polygons need to be rendered to the screen and also some of these polygons are overwritten resulting in the increase in the running times of the algorithm. The size of the polygons is an interesting parameter to observe. The increase in the size of the polygons will mean that there are more number of pixels to be written onto the frame-buffer and also increases the intersection of the polygons(i.e. more triangles covering the same pixel). As number of polygons increases with larger polygon sizes, more number of pixels need to be written resulting in the exponential increase in the running times.

A.



B.



C.

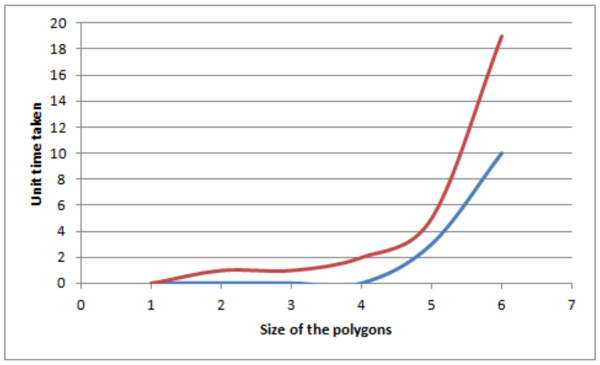
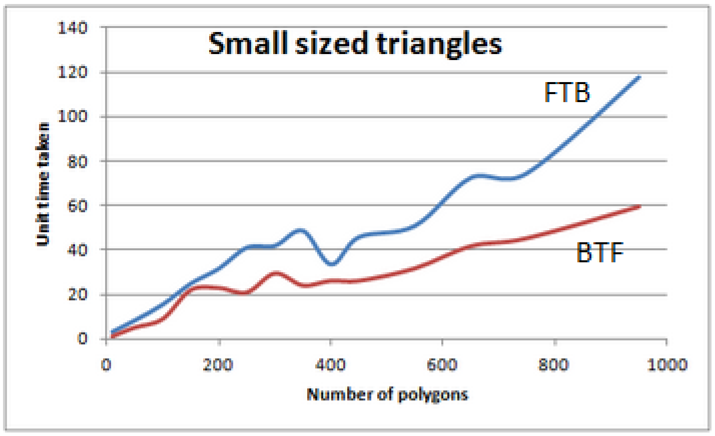


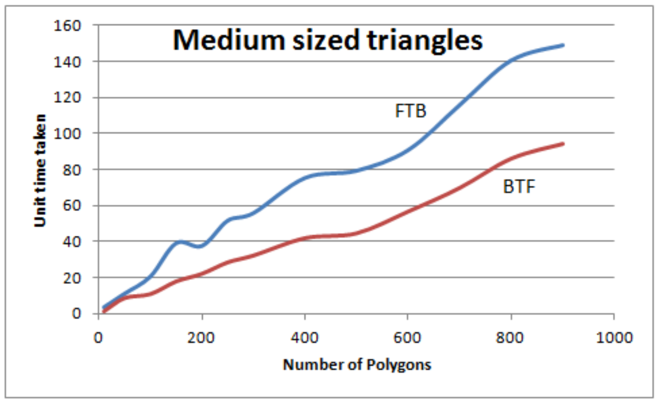
Figure 7: Scanline algorithm performances. A. shows the increase in running times with the increasing number of polygons per pixel. B. shows the increase in running times with the increase in number of polygons in the model. C. shows the exponential increase in running times with the increase in size of polygons.

The comparative performances of the Front-to-back(FTB) traversal using DSD and Back to Front (BTF) are shown in the figure 8. When the number of triangles are smaller the FTB and BTF perform relatively at the same rate for smaller number of triangles. The larger differences can be observed in the Medium, Large and Multiple triangles per pixel case. In these cases number polygons covering the same pixel keeps increasing resulting in collisions. Due to this, time taken by BTF started increasing at high rate. On the other side, FTB (using DSD) which avoids the repainting of the pixels provide significant run time performance improvement.

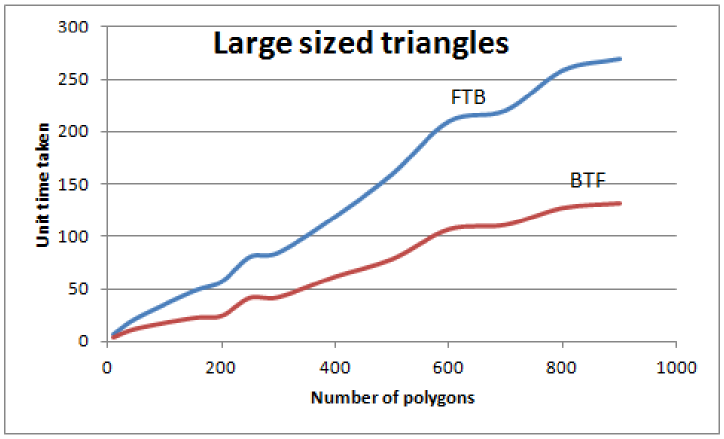
A.



B.



C.



D.

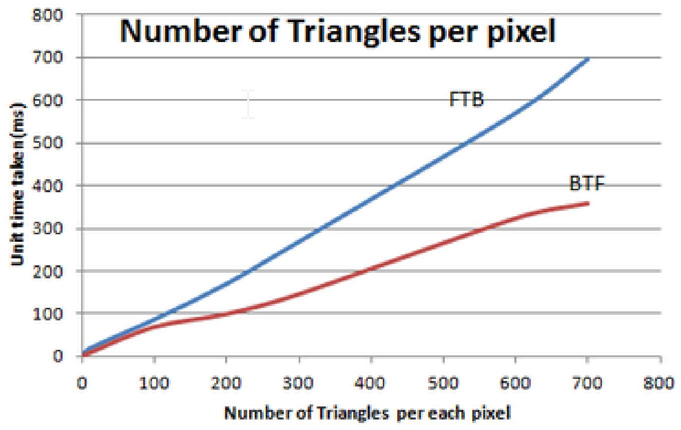


Figure 8: Shows the results of the Front to Back vs Back to front algorithm implementation. Legend: Red is Back to front(BTF) and Blue is Front to Back(FTB) results.

5.0 Discussion

We have observed that the BSP trees with the scanlines lead to lesser running times than scanline using z-buffer. We have also seen that the BSP trees with front-to-back implementations(using DSD) are more efficient than the back-to-front traversal of the BSP trees. The back-to-front are effectively the same as the scanline with z-buffer implementation as the pixels colliding with multiple polygons get overwritten to the framebuffer. In the case of front-to-back this is avoided as a result of which only those pixels which haven't been printed are added to the framebuffer and others are ignored. This saves time and results in a comparatively faster implementation. This is evident in the case of large number of polygons and also when the pixels per polygons are huge.

References

[1] Gordon, Dan, and Shuhong Chen. "Front-to-back display of BSP trees." IEEE Computer Graphics and Applications 11.5 (1991): 79-85.
[2] Fuchs, Henry, Zvi M. Kedem, and Bruce F. Naylor. "On visible surface generation by a priori tree structures." ACM Siggraph Computer Graphics. Vol. 14. No. 3. ACM, 1980.
[3] Reynolds, R. Anthony, Dan Gordon, and Lih-Shyang Chen. "A dynamic screen technique for shaded graphics

display of slice-represented objects." Computer Vision, Graphics, and Image Processing 38.3 (1987): 275-298.
[4] ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html
[5] Havran, Vlastimil. *Heuristic ray shooting algorithms*. Diss. Faculty of Electrical Engineering, Czech Technical University, 2000.