FAST ROBUST BSP TREE TRAVERSAL ALGORITHM FOR RAY TRACING

Vlastimil Havran, Tomáš Kopal, Jiří Bittner, Jiří Žára
Czech Technical University, Faculty of Electrical Engineering
Department of Computer Science and Engineering
Karlovo nám. 13, 12135 Prague 2, Czech Republic
e-mail: {havran, kopal, bittner, zara}@fel.cvut.cz

ABSTRACT

An orthogonal BSP (binary space partitioning) tree is a commonly used spatial subdivision data structure for ray tracing acceleration. While the construction of a BSP tree takes a relatively short time, the efficiency of a traversal algorithm significantly influences the overall rendering time. We propose a new fast traversal algorithm based on statistical evaluation of all possible cases occurring during traversing a BSP tree. More frequent cases are handled simply, while less frequent ones are more computationally expensive. The proposed traversal algorithm handles all singularities correctly. The algorithm saves from 30% up to 50% of traversal time comparing with the commonly known Sung and Arvo algorithms.

Keywords: spatial subdivision, BSP tree, ray tracing, traversal algorithm.

1 INTRODUCTION

Spatial subdivision techniques are well–known methods used to speed up ray tracing calculations. There is a variety of schemes for spatial subdivision including uniform ones (grids), non–uniform (BSP tree, octree), and their combinations. The basic introduction to acceleration techniques is given in [Watt92]. Their more comprehensive classification can be found in [Glass89]. Generally, the time complexity of the ray tracing is determined by casting a large amount of rays. The computation of a first object in the scene, that is hit by a given ray, is known as the ray–casting problem.

The principle of all spatial subdivision techniques is the reduction of ray–object intersection computations by partitioning the scene space into disjunct cells. The ray–object intersection test is then computed only for cells located along the ray path. Although the number of intersection tests is reduced significantly, there are additional time requirements for *traversal* of spatial subdivision data structures. One way to reduce the overall time is to improve spatial data structures, the second method is to improve the traversal algorithm. This paper deals with the latter approach.

A Binary Space Partitioning (BSP) tree is simple and powerful data structure that can be used to solve a variety of geometrical problems. In case of the ray tracing a BSP tree is often constructed using splitting planes perpendicular to principal axes only. It is advantageous especially when rendered objects with complex shapes are enclosed by axis–aligned bounding boxes. Moreover, the orthogonality of BSP tree decreases the number of computations required for an intersection of a ray and a splitting plane.

2 TRAVERSAL CLASSIFICATION

When finding the nearest object pierced by a ray during rendering process, a BSP tree is traversed from the root to the leaves. Corrected sequence of leaves has to be identified for a given ray origin and direction. The nearest leaf has to be processed first.

The splitting plane subdivides a rectangular cell into two smaller ones. The test performed during the traversal has to determine, whether to visit both nodes and in which order or just one node and which one.

All possible cases are shown in Fig. 1, classified by the ray origin and its direction with respect to a given

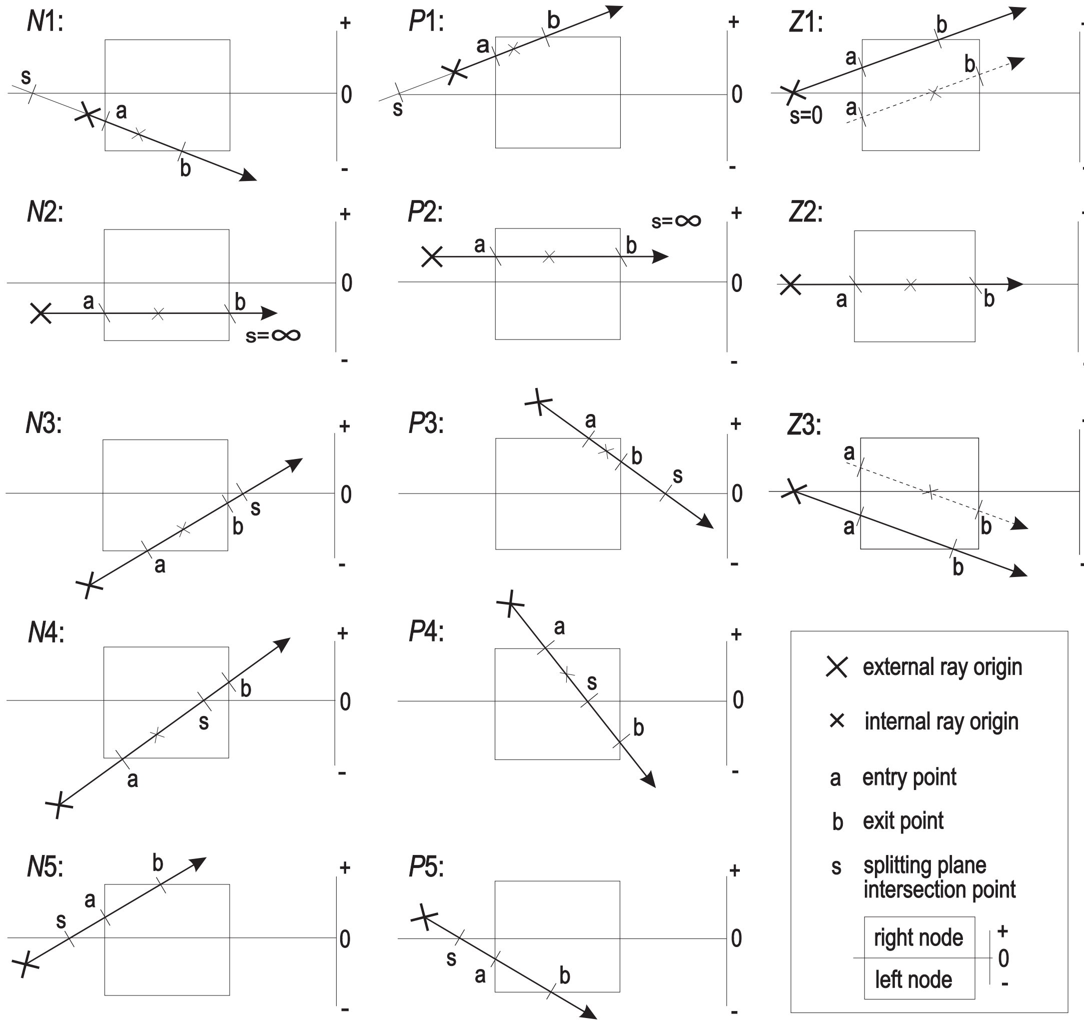

Figure 1: Classification of mutual positions between a ray and a BSP tree node

splitting plane. Since the splitting plane is oriented, we can distinguish between *negative* location of the origin (below the splitting plane) and the *positive* location (above the plane). These cases are marked as N and P respectively. Three remaining cases Z show the situation when the ray origin is embedded in the splitting plane. For the purpose of the following text, we call the BSP node representing subvolume below the splitting plane as a *left* node. The other is the *right* node.

Important geometrical points of the classification scheme are characterized by their signed distance measured from the ray origin along the ray direction. The ray enters the node at the *entry* point denoted by signed distance a and exits the node at the *exit* point denoted by signed distance b. It is obvious that $a \leq b$. The signed distance from the ray origin to the splitting plane is denoted as s. The classification includes all possible mutual relations of a, b, and s assuming the ray hits the bounding box of given node.

The classification basically depicts the cases when ray origins are placed outside the node. Such rays have *external* origins. All cases can also incorporate *internal* origins located inside the node. Possible internal origins are distinguished by thin crosses in Fig. 1. The only exceptions are cases $N5$ and $P5$ respectively, which are equivalent to cases $P1$ and $N1$ respectively from the point of view of internal origins.

3 COMMON TRAVERSAL ALGORITHM

A commonly used traversal algorithm has been described several times in the literature ([Janse85, Arvo88, Sung92]). We call it TA–A algorithm in this paper.

The algorithm uses signed distances a, b, and s measured from the ray origin. The value a is the signed distance to the entry point, similarly the signed distance b corresponds to the exit point.

The sequence of comparisons determines which child node will be processed in the next step. Child nodes are called *near* and *far* in Sung algorithm. The near node is placed on the same side as the ray origin with respect to the splitting plane. If $s < 0$ or $s > b$, then only the near node will be visited by the ray (cases $N1, P1$ or $N3, P3$ in Fig. 1).

Otherwise, if $s < a$ (cases $N5$ and $P5$), then the ray hits the far node only. If s is between a and b, the ray first hits the near and then the far node (cases $N4$ and $P4$).

It is interesting that cases $Z1$ and $Z3$ are not solved correctly, because the origin placed in the splitting plane does not give any hint to distinguish which node is *near* and which is *far*. The problem can be overcome by additional tests of the ray direction. Robust implementation has also to solve the cases $N2$, $P2$, and $Z2$, where the signed distance s cannot be directly computed. It must be assigned to certain constant if the division by zero is detected when computing s.

```
// a = signed distance from origin to entry point a
// b = signed distance from origin to exit point b
// s = signed distance from origin to splitting plane
// LeftNode  = the child node in negative halfspace
// RightNode = the child node in positive halfspace

Pick up a and b from memory
Compute and store s
if (origin in negative halfspace) {          // cases N
    NearNode = LeftNode
    FarNode  = RightNode
}
else {                                       // cases P
    NearNode = RightNode
    FarNode  = LeftNode
}
if (s < 0) or (s > b)
    Visit NearNode                 // N1, P1 or N3, P3
else
    if (s < a)
        Visit FarNode                        // N5, P5
    else
        Visit NearNode and FarNode           // N4, P4
```

Algorithm 1: Selection of node(s) in algorithm TA–A

Note, that signed distances a and b used in algorithm TA–A are not recomputed in each traversal step. They are inherited from parent nodes during traversing the BSP tree. Actually, the distance s replaces either a or b for child nodes. A use of a stack is convenient for storing a and b values.

The C implementation of this algorithm, which is used as a reference one for the following text, can be found on the web [GEMS].

4 NEW TRAVERSAL ALGORITHM

In this section we propose a new algorithm TA–B, which is more efficient than algorithm TA–A. Moreover, it solves correctly all the cases depicted in Fig. 1. The algorithm compares *coordinates* instead of *distances*. This is allowed, since the BSP tree is orthogonal.

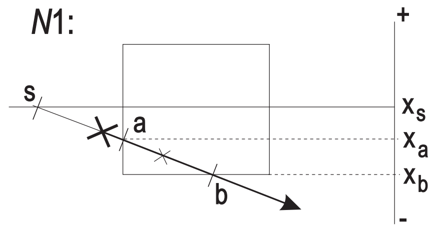

Figure 2: Example of traversal case $N1$

For the explanation, see Fig. 2. It depicts the case $N1$. Let us suppose that the splitting plane is perpendicular to the x axis. Then x coordinates of points s, a, and b are sufficient enough to distinguish among all traversal cases (Fig. 1). The case $N1$ is recognized when $x_a \le x_s$ and $x_b \le x_s$.

```
// a[coord] = proper coordinate of entry point a
// b[coord] = proper coordinate of exit point b
// s        = splitting plane offset

Pick up a[coord], b[coord] and s from memory
if (a[coord] <= s) {
    if (b[coord] < s)
        Visit LeftNode     // N1, N2, N3, P5, Z3
    else
        if (b[coord] == s)
            Visit arbitrary child node        // Z2
        else {
            Compute and store location of "s"
            Visit LeftNode and RightNode      // N4
        }
}
else {   /* i.e. (a[coord] > s) */
    if (b[coord] > s)
        Visit RightNode    // P1, P2, P3, N5, Z1
    else {
        Compute and store location of "s"
        Visit RightNode and LeftNode          // P4
    }
}
```

Algorithm 2: Selection of node(s) in algorithm TA–B

The algorithm TA–B needs exactly two comparisons to find certain traversal case. Appropriate coordinates x, y, or z are selected according to orientation of the given splitting plane. The coordinate s is not recomputed because it is already stored in the splitting plane data.

When both nodes are hit by the ray (cases $N4, P4$), complete location of point s has to be computed. It is the most time consuming part of the algorithm. Fortunately, coordinates of point s are reused later, because this point alters to points a and b in child nodes. This is implemented in the same way by a stack as in the algorithm TA–A.

The statistical occurrence of individual traversal cases plays important role in overall efficiency of the TA–B algorithm. While the signed distance s is repeatedly evaluated in each traversal step in algorithm TA–A, new algorithm TA–B has been designed with the aim to process costly computations only when needed.

5 STATISTICAL POINT OF VIEW

The idea behind the efficiency improvement is based on the statistical analysis. The traversal cases occurring less frequently are dealt with higher constant time complexity, while the cases occurring more frequently are handled as much as efficiently.

In terms of algorithm TA–B, the piece of code with the highest cost belongs to the cases $N4$ and $P4$. The computation of all three coordinates of point s represents several algebraic operations including multiplication and division. In this particular case, the algorithm TA–B is less efficient than TA–A, where the computation of signed distance to the splitting plane takes lower number of algebraic operations.

The question is how frequent is the situation in which the ray crosses the splitting plane and visits both child nodes. The answer can be obtained from the theoretical analysis based on the *surface area heuristics* (see [Glass89] or [MacDo90a]). The probability that a ray of arbitrary direction hits certain object A (a splitting plane) once it passes through a given volume B (a node bounding box) is proportional to the surface areas.

We have derived a formula for conditional probability regarding cases $N4$ and $P4$. It depends on dimensions of the bounding box and positions of splitting planes inside this box. The worst case occurs, when all dimensions are the same (the volume is of a cubic shape) and splitting planes are placed just in the middle of the volume. Then the probability has theoretically the highest value $47/180 \doteq 26.1$ %. The proof of the formula is out of scope of this paper.

It is important to follow theoretical considerations by practical measurements. Several experiments have been performed on the SPD scenes described in [Haine87]. The BSP trees were built up using *surface area heuristics* [MacDo90a]. Table 1 shows the statistics of the traversal cases. The critical cases $N4$ and $P4$ are highlighted.

Scene	balls	tetra	mount	gears	average
$N1 + P1$ [%]	29.6	18.0	43.4	43.9	33.7
$N2 + P2$ [%]	0.04	0.0	0.02	0.04	0.03
$N3 + P3$ [%]	32.8	39.0	29.2	34.5	33.9
N4 + P4 [%]	**24.4**	**20.8**	**20.5**	**15.8**	**20.4**
$N5 + P5$ [%]	13.3	22.1	7.0	5.7	12.0
$Z1, 2, 3$ [%]	0.0	0.0	0.0	0.0	0.0

Table 1: Traversal cases statistics for SPD scenes

Real measurements show that occurrence of cases $N4$ and $P4$ is lower than theoretical value 26.1 %. We see that the only piece of code, where the computations in algorithm TA–B are of higher cost than in algorithm TA–A, is performed with reasonably low probability.

6 RESULTS

We have tested both algorithms discussed above by measurements performed on the SPD scenes. Table 2 shows running times for four selected SPD scenes. Total running time for the algorithm TA–A and TA–B is in the first and the second line. The times devoted to the traversal only (not including the intersection tests with the objects in the scene) are enumerated in the following lines.

Next two lines show how much the algorithm TA–B saves time comparing with the algorithm TA–A. The fifth line expresses saving for traversal part only. The last line shows total time reduction.

All the experiments were conducted on PC based computer equipped with *Pentium*, 166 MHz, running under *Linux* operating system.

The results depicted in Table 2 show evident improvement achieved by the new algorithm. Time saving for traversal part of ray tracing is certainly remarkable, the speedup is about two in average. The impact to the overall rendering time depends on the scene geometry and complexity. The range of possible total time saving is thus quite wide –

Scene	balls	tetra	mount	gears
t_{total}(TA–A) [sec]	94.21	7.86	122.4	307.7
t_{total}(TA–B) [sec]	70.34	6.33	94.26	281.4
t_{trav}(TA–A) [sec]	45.58	4.34	52.97	58.52
t_{trav}(TA–B) [sec]	21.71	2.81	24.83	32.22
saving (trav.) [%]	52.37	35.25	53.12	44.94
saving (total) [%]	25.34	19.47	22.99	8.54

Table 2: Experimental comparison between TA–A and TA–B algorithm

from 9 to 25 %.

7 CONCLUSION

In this paper we have described and analyzed a new algorithm for the orthogonal BSP tree traversal with theoretically the smallest number of condition operations. We have shown statistically based method for improving efficiency of traversal algorithms. The costly operations performed in each traversal step in commonly used Arvo and Sung algorithm have been left and replaced by combination of simple tests and one occasionally performed computation with a higher cost.

Theoretical expectations have been proved by practical measurements. The speedup reaches from 1.5 up to 2 depending on the input data. Additionally, the new algorithm correctly handles all singular cases, in which the commonly used algorithm can fail.

Acknowledgment

This research was partially supported by Grant Agency of the Ministry of Education of the Czech Republic number 1252/1998 and by Internal Grant Agency of Czech Technical University in Prague number 309810103.

References

[Arvo88] Arvo, J.: *Linear–time voxel walking for octrees*, Ray Tracing News, E–mail edition, March 26, 1988.

[Glass89] Glassner, A.,S.: *An Introduction to Ray Tracing*, Academic Press, London 1989.

[GEMS] http://www.acm.org/tog/GraphicsGems

[Haine87] Haines, E.: *A proposal for standard graphics environments.* In IEEE Computer Graphics and Applications, Vol. 7, No. 11, pp. 3–5, 1987.

[Janse85] Jansen, F.W.: Data structures for ray tracing, In book *Data Structures for Raster Graphics*, Springer Verlag, 1985.

[MacDo90a] MacDonald, J., D., Booth, K., S.: *Heuristics for ray tracing using space subdivision*, The Visual Computer, Vol. 6, No. 3, pp. 153–166, 1990.

[Sung92] Sung, K., Shirley, P.: Ray Tracing with BSP Tree, *Graphics Gems III*, pp. 271–274, 1992.

[Watt92] Watt, A., Watt, M.: *Advanced Animation and Rendering Techniques*, ACM–PRESS, Addison–Wesley, 1992.