Ray Tracing with Rope Trees

Vlastimil Havran, Jiří Bittner, Jiří Žára

Czech Technical University, Faculty of Electrical Engineering, Department of Computer Science and Engineering, Prague, Czech Republic

e-mail: {havran, bittner, zara}@fel.cvut.cz

Abstract

In this paper an acceleration method for finding the nearest ray–object intersection for ray tracing purposes is presented. We use the concept of BSP trees enriched by *rope trees*. These are used to accelerate the traversal of the BSP tree. We give a comparison of experimental results between the technique based on BSP tree and uniform spatial subdivision.

Key words: spatial subdivision, BSP tree, spatial data structures, rope trees, ray tracing.

1 Introduction

Ray tracing is a well–known rendering technique for producing realistic images that simulates well specular surfaces. The main drawback of this algorithm is its rather big computational complexity, that disallows its interactive use. The problem has been focused a great deal of research interest in the past leading to some practical techniques to accelerate the basic algorithm.

The principal expense of ray tracing is the determination of the closest ray–object intersection for a given ray and a set of objects. This problem is known as *ray–casting*. The naive algorithm solves the ray–casting problem by testing all objects for intersection with a given ray. Another related problem is determining if two points in a scene are visible. In ray tracing the visibility of two points is used to determine shadows cast by point light sources. It can be solved by means of ray–casting or using specialized techniques.

In this paper we deal with an unusual method of ray–casting acceleration based on BSP trees. We present an analysis of the method and compare its performance with common acceleration schemes.

Related Work

The ray–casting problem has been dealt by many researches in the field of computer graphics and computational geometry. Techniques developed by these two groups differ in the approach used.

The first community mentioned is oriented to improve the average complexity of ray–casting. The algorithms developed are intended to be used in practice, even for large scale scenes. They are mostly based on the observations and heuristics. Their comprehensive overview can be found in [4]. More recent survey is presented in [19].

The computational geometry community solves the problem more theoretically focusing on the worst–case complexity. In order to obtain mathematically provable results the scene description is usually restricted to polygons. A recent algorithm published [6] reaches the time complexity $O(\log n)$ with $O(n^{4+\epsilon})$ preprocessing time and $O(n^{4+\epsilon})$ storage, where ϵ is an arbitrarily small positive constant, and n is a number of polygons. Although these results are important theoretically, the preprocessing and storage complexities restrict their practical use.

Overview

The approach presented in this paper focuses on improving the average case complexity of ray–casting. In particular, it is suitable for scenes containing large number of objects. We exploit the idea of spatial subdivision, which serves to determine a portion of the scene pierced by a ray efficiently.

A short overview of the properties and the construction of a uniform spatial subdivision and a BSP tree is given in Section 2. In Section 3 we present the concept of ropes and rope trees. A method further increasing the performance of ray–casting using spatial subdivision is dealt in Section 5. In Section 6 we present comprehensive results of measurements and comparisons. Section 7 surveys the contribution of the rope trees technique to ray tracing. Finally, Section 8 points out some topics for the future work.

2 Accelerating Data Structures

In this section we give a survey of two spatial subdivision techniques. These are the uniform spatial subdivision and the BSP tree.

Uniform Spatial Subdivision

A *Uniform spatial subdivision* (often called *grid*) is one of the first methods developed for spatial subdivision [7]. It involves the subdivision of the scene space into equally sized elementary *cells* regardless of the object distribution in the scene. The three–dimensional grid resembles the subdivision of a two–dimensional screen into pixels. Each cell is assigned a list of objects intersecting it.

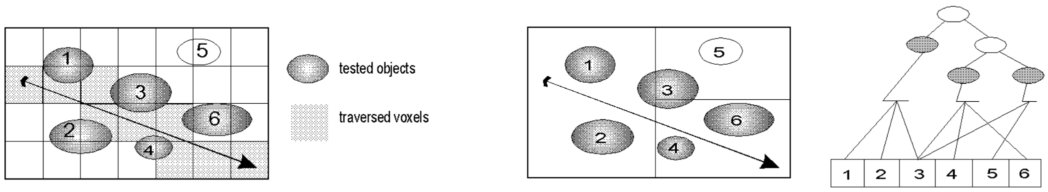

Figure 1: Uniform Spatial Subdivision (left) and BSP Tree (right)

Since the uniform grid is built regardless of occupancy by objects, it usually contains many cells and therefore it demands large storage space. Nevertheless, the traversal of the grid structure can be performed efficiently using a 3D–DDA algorithm [2][7]. A ray is tested only for intersection with objects referred in grid cells pierced by the ray.

The uniform grid can contain many empty cells (i.e., cells with no objects referred). Therefore a ray may traverse many empty cells before an intersection with an object is found. The method was analyzed in [5] by means of mathematical statistics.

Binary Space Partitioning Tree

A *Binary Space Partitioning* (BSP) tree is an analogue to the binary search tree. The BSP tree represents a scene containing a set of objects hierarchically. The BSP is formed by recursively subdividing the scene space in two parts. Each interior node contains a splitting plane. All nodes of the BSP tree correspond to convex polyhedral cells. In each leaf of the BSP tree a list of objects intersecting the corresponding cell is stored.

The *axis-aligned BSP* trees are usually used for ray tracing purposes since they enable efficient traversal by a ray. These are BSP trees with all splitting planes perpendicular to the main coordinate axes. In such case all cells corresponding to the nodes of the BSP tree are rectangular parallelepipeds.

An important factor influencing the construction of a BSP tree and its resulting properties is the criterion for positioning the splitting plane. The construction of BSP trees for ray tracing was studied by Kaplan [15] and improved by MacDonald and Booth [17]. There are recent publications in computational geometry (e.g. [1]), that show a persistent research interest in BSP trees.

We adopted the subdivision method of MacDonald and Booth, which is described in the next section in detail.

Statistical Optimization of a BSP Tree

The time needed for construction of a BSP tree is typically insignificant compared with the computation time spent traversing the tree to determine ray–object intersections. Hence it is advantageous to devote a greater effort to create an efficient tree, under the assumption, that the extra time would then be recovered during traversal.

In [17] a simple heuristics for finding the optimal position of a splitting plane is used. These planes are perpendicular to the main coordinate axes, which simplifies both construction and traversal algorithms. The plane position is determined by minimizing a *cost function*. The cost function is based on the probability that a ray hits an object placed inside a certain volume once it passes through that volume as shown in Fig. 2.

Suppose that both object B and a volume A are of convex shape. Then the conditional probability $Pr(B|A)$ is expressed as a ratio of the surface area of the object B to the surface area of the volume A (see [4]):

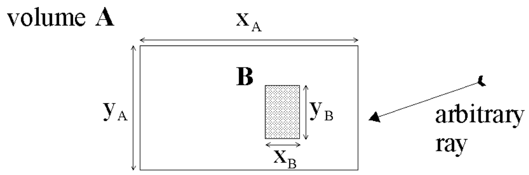

Figure 2: Surface area heuristics.

$$Pr(B|A) = \frac{S_B}{S_A} = \frac{2(x_B.y_B + x_B.z_B + y_B.z_B)}{2(x_A.y_A + x_A.z_A + y_A.z_A)} \tag{1}$$

The cost function is defined using conditional probability and expresses the estimated time needed for traversing a ray through the BSP tree. During the building of a BSP tree the cost function helps to decide when and where to split a certain cell, i.e., to replace a leaf node by a new interior node with two children (*sub–cells*).

Let us assume the situation at the beginning of a tree construction. One node contains n objects. All of them have to be tested for intersection with a ray passing through the scene. The intersection test for i-th object takes computation time T_i. The cost for such non-subdivided node is given as follows:

$$C = \sum_{i=1}^{n} T_i \tag{2}$$

A space subdivision helps to decrease the number of intersection tests, but increases the number of interior nodes. The cost has to incorporate the time needed for traversing all nodes visited by a ray. Without loss of generality let us suppose the splitting plane is perpendicular to x-axis. Figure 3 shows the geometrical factors that influence the change of the cost for one space subdivision step.

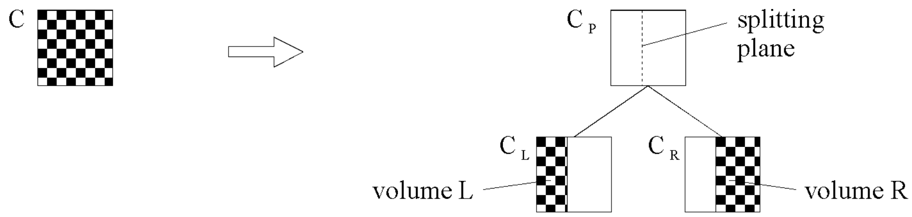

Figure 3: New costs after one subdivision step

The node on the left side in Fig. 3 has been replaced by a new tree structure on the right side in Fig. 3. The original cost C is changed to the cost C_{new} given as the sum of three terms - C_P, C_L, and C_R. The term C_P is the cost of traversing the parent node only. It does not incorporate any ray–object intersection tests. Costs for left and right child nodes, C_L and C_R, contain a factor with the conditional probability that a ray hits the node L or R once it visits the parent node P. The new cost C_{new} is given as follows:

$$C_{new} = C_P + C_L + C_R = T_P + \frac{S_L}{S}\sum_{j=1}^{n_L} T_j + \frac{S_R}{S}\sum_{k=1}^{n_R} T_k + T_T \tag{3}$$

where	T_j, T_k	is the time for intersection test with j-th and k-th object respectively
	T_T	is the time for performing one traversal step
	T_P	is the time for decision step in parent (interior) node
	S_L, S_R	is a surface area of left sub–cell and right sub–cell respectively
	S	is the surface area of the node to be subdivided
	n_L, n_R	is the number of objects belonging to the sub–cell L and sub–cell R respectively

The formula (3) represents the worst–case when the ray visits both left and right sub–cells. Another improvement could be achieved by incorporating conditional probability expressing that the ray visits only one sub–cell. Such a situation occurs when either a ray is directed into one sub–cell only or a ray hits any object in the first sub–cell and does not continue to the second one. The probability would depend on the area obtained by projecting objects from one sub–cell on the surface of the other sub–cell (see [18]). In the following text, we are dealing with the "worst–case" probability only.

The aim is to build the binary tree with minimized (suboptimal) global cost. This can be achieved by minimizing values of the cost function (3) depending on the position of a splitting plane. The plane can intersect some objects in the original cell and such objects have to be included into both costs C_L and C_R. That is the reason why $n_L + n_R \geq n$ (n is the number of objects in the original cell).

The minimum of the cost function can be roughly estimated using a few sample splitting planes. MacDonald and Booth [17] showed that there are two important positions of the splitting plane. One position is in the geometrical center of the cell. Let us call it *spatial median*. The second position is in the middle of an ordered list of objects[1], which is called an *object median*. The interval specified by spatial and object median marks the boundaries of possible positions of the optimal splitting plane upon the condition that this plane does not intersect any object. If objects inside the median interval are overlapping, then the optimal splitting plane can lie outside this interval. In the case the use of the planes lying outside the spatial–object median interval can lead to further increase of efficiency (see [12]).

Although the ray–traversal algorithm through the BSP tree deserves a special attention, we do not describe it here in detail. The first algorithm published in [15] is called the *sequential* algorithm. A more efficient *recursive* algorithm was introduced by Jansen [14] and described in more detail in [3] and [20].

In the following section we deal with another efficient ray–traversal algorithm based on the BSP tree, that was first introduced in [17].

3 Rope Trees for BSP Tree

In this section we present the concept of *ropes* and *rope trees* for BSP tree. We outline rope trees construction and ray–traversal algorithm using the data structure.

Motivation

The inner representation of the BSP is very important for the efficiency of its ray–traversal algorithm. The usual way is to represent BSP as a binary tree with the splitting plane referred in the interior nodes of the tree. Each node of the tree corresponds to a polyhedral cell. In the case of axis-aligned splitting planes these cells are rectangular boxes. A leaf contains a list of links to the objects intersecting the corresponding cell.

An efficient recursive algorithm for traversal of this structure was published by Jansen in [14]. It is more elaborated theoretically in [3]. If N denotes the number of leaves in BSP tree, then the average number of traversed interior nodes and leaves ray is $2.N^{\frac{1}{3}}$ supposing the ray does not hit any object and the leaf cells form a structure similar to a grid.

The recursive ray–traversal for all types of rays (primary, secondary, shadow) always starts from the root node of the tree and continues downwards. It proceeds identifying pierced nodes until it hits an object. It is often the case, that a ray intersects objects very close to its origin and thus the down-traversal from the root can form a large portion of the whole traversal time. We observed experimentally that this portion can be quite significant, as we show in Section 6. This property makes the potential time to be eliminated and thus to speed up the rendering process. The concept of ropes and rope trees was outlined in [17], however, no clear results based on experimental measurements or an analysis were presented.

Ropes

Each leaf in the BSP tree corresponds to the axis-aligned *leaf–cell*. Each such a leaf–cell has six faces. The faces of a leaf–cell will be called *leaf–faces* further in the paper. The cells intersecting a face of a leaf are called *neighbour* cells.

There are two mutual relations for a given leaf-face and its neighbour leaf–cells. Firstly, a leaf–face can be contained in the neighbour leaf–cell completely. In the case there is only one neighbour corresponding to a given leaf–face. Secondly, a leaf–face has intersection with more leaf–cells. It means, that there are more neighbour leaf–cells corresponding to a given leaf–face.

For a given leaf–face we call a *neighbour–node* a node, which corresponds to a smallest cell which contains the leaf–face completely. The neighbour–node is either a leaf or an interior node of the BSP tree.

A *rope* is a link from a leaf–face to its neighbour–node. Using ropes in ray–traversal algorithm we can avoid most of the traversal steps of interior nodes of the BSP tree.

---

[1] Objects are ordered by the x coordinate in the case of a splitting plane perpendicular to the x axis; similarly for the other two orientations of the splitting plane

Rope Construction Algorithm

The construction of ropes is straightforward. For each leaf–cell of the BSP tree a rope is set up. For a given face and leaf the hierarchy is searched starting from the root node. In each step the search continues in the subtree, which corresponds to a cell intersecting the face. If the face is split by the plane referred in the currently reached node (i.e., it intersects both subtrees), the search is terminated. A rope to the neighbour–node obtained by the search is stored within the leaf–face the algorithm was applied for.

Assume that the BSP tree has n leaves and its average depth is $O(\log n)$. For each face the down traversal takes $O(\log n)$ steps on average. It is applied on $6n$ faces. Hence the complexity of the algorithm is $O(n \log n)$.

Rope Trees

As it will be shown later, the ropes are used to locate a neighbour–leaf for a ray leaving the current leaf during the ray–traversal algorithm. Ropes point either to leaves or interior nodes of BSP tree. Denote a rope that points to an interior node of BSP tree an *indirect* rope. In such a case it can be replaced by the rope tree.

The *rope trees* are additional data structures exploited for the neighbour–leaf identification, leading to a faster ray–traversal algorithm. Let us discuss the motivation for building the rope trees.

The first drawback of indirect ropes with respect to the ray–traversal algorithm is that BSP subtree pointed to by the rope is a three–dimensional data structure. Therefore the splitting planes in the neighbour–node, that are parallel to the face where a ray exits the current leaf cell have to be traversed downwards. This is inefficient, because for a given leaf-cell we actually search for a neighbour–leaf on the leaf-face, which is a two–dimensional searching problem. A *Rope tree* is a two–dimensional binary tree, that is formed by pruning of splitting planes in neighbour–node. The pruned splitting planes are either parallel to the plane supporting a given leaf–face or they do not intersect the leaf–face supposing they are restricted to the extent of a current node.

The projection of the neighbour cell to the plane supporting a given leaf–face gives the second way to decrease the traversal complexity. Only splitting planes (projected as lines to the plane) intersecting the leaf–face are used in the rope tree. This pruning corresponds to the clipping of the two-dimensional BSP tree against a rectangle (leaf–face).

The algorithm of rope trees construction replaces indirect ropes by a corresponding rope trees. Starting from the node the indirect rope points to, we perform a constrained *depth–first–search* (DFS) on the BSP tree. Only subtrees corresponding to cells intersecting the face are visited during the DFS. Only the nodes whose faces intersect a given leaf–face are added to the rope tree.

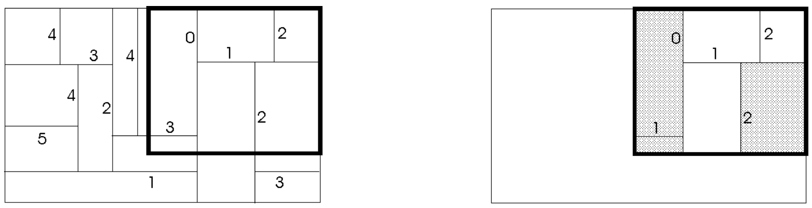

Figure 4: Building of Rope Tree: left - unclipped, right - clipped

The two–dimensional clipping of a rope tree is depicted in Figure 4. On the left side a leaf–face (smaller rectangle with thicker edges) the rope tree is built for is shown. The large rectangle is a projection of corresponding neighbour node subtree on the plane supporting a given leaf–face. The numbers marking the splitting planes denote the depth of the node in the subtree (the root node of the subtree is in the depth zero). Note that the first splitting plane (root node of the subtree) always goes through the face of the cell which the rope tree is built for. On the right side of the figure a rope tree is constructed by clipping with minimal number of splitting planes. Three grey areas corresponding to neighbour leaf–cells depict parts of the leaf–face, where the traversal is accelerated owing to the clipping of the rope tree.

We have observed (see Section 4) that the average number of neighbour cells per face is bound by a small constant. Then the algorithm takes $O(n)$ time, requiring $O(n)$ memory for rope trees. See Sections 4 and 6 for further details.

4 Traversal Algorithm

In order to exploit rope trees during ray–casting the traversal algorithm for the BSP has to be modified. The algorithm consists of two components: the *exit–face determination* and the *point location*.

Assume the ray–traversal starts in a certain leaf–cell. Supposing the ray is not terminated here, we require to locate the next leaf–cell pierced by the ray. First we determine the *exit–face* of the leaf–cell, that is intersected by the ray in its positive direction. Knowing the exit–face the *exit–point* on the face is computed. We follow the rope corresponding to the exit–face. If it is an indirect rope, we apply the point location using the rope tree to determine the next leaf–cell pierced by the ray. The point location is a simple down traversal of the rope tree starting from its root comparing the coordinates of a given exit–point to splitting planes of the interior nodes of the rope tree.

```
Object*
RayCasting( ray, BSP)
{
   if (a leaf of ray origin is not known) {
      entryPoint = intersection of ray and BSP bounding box;
      if (no entry point exists) return NULL;
      currentLeaf = LocateLeaf( BSP, entryPoint);
   }

   // traverse through whole BSP tree
   while (currentLeaf != NULL) {
      // exit-face determination
      nextExitFace = GetExitFace( currNode, ray, exitPoint);

      if (currentLeaf is not empty)
         if (ray intersects an object in currentLeaf) {
            ray = terminationLeaf = currentLeaf;
            return object;
         }

      if (currentLeaf.ropes[nextExitFace] == NULL)
        return NULL; // the ray is leaving the BSP

      if (currentLeaf.ropes[nextExitFace] is leaf)
         currentLeaf = currentLeaf.ropes[nextExitFace];
      else
         currentLeaf = LocateLeaf( currentLeaf.ropes[nextExitFace], exitPoint);
   }
}
```

Figure 5: The pseudo-code of the Traversal Algorithm of the BSP Tree with Rope Trees

If the ray–traversal is terminated in a leaf–cell, the corresponding leaf is used to start the ray–traversal for all rays spawned from the terminated ray. These are the reflected, refracted, and shadow rays. If the origin of a ray is outside the parallelepiped (axis–aligned cell) corresponding to the root of the BSP, the intersection of the ray and the parallelepiped is computed. The intersection point (if any) is used to locate the first leaf–cell pierced by the ray. The leaf–cell is found using the point location starting at the root node of the whole BSP tree.

Algorithm Analysis

The rope trees technique requires an additional memory to store the rope trees, but on the other hand, it eliminates the long (often called vertical) traversal steps from the root downwards. The elimination of the traversal steps is especially remarkable for secondary and shadow rays because we know exactly in which leaf–cell the origin of the ray is located. The same situation occurs when the viewpoint lies inside the axis–aligned bounding box of the scene. The long traversal is also performed after the ray traverses upwards from the leaf to the root node (or nearly to the root node) and then backward to the neighbour leaf. For this traversal the intersections of a ray with many splitting planes must be evaluated which is costly and it can be useless. These computations are eliminated by rope trees in a natural way.

In [17] it was shown that the average number of neighbour leaf–cells per leaf–face is always lower than two for an octree. It seems difficult to analyze the case of the general axis–aligned BSP. Nevertheless, some preliminary results supported by a simulation show this number is on average limited by a small constant (smaller than four) independently of the depth of the BSP tree.

This observation is very important, since it bounds the average size of the rope trees as well as the average number of the point location steps (down traversal of the rope tree).

5 Cleaning Up the Spatial Subdivision

In this section we deal with a simple idea introduced in [8], but mostly not exploited in spatial subdivisions. It further increases the ray tracing efficiency by reducing the number of intersections to be computed.

Usually, the objects are assigned axis–aligned bounding boxes. During the construction of a spatial subdivision the bounding box is repeatedly used to determine if the object belongs to a given axis–aligned cell. It is often the case especially for oblong and rod objects, since their axis–aligned boxes are very loose. In many cases the cells of the spatial subdivision contain the links to the objects, that actually do not belong to these cells. The situation can be solved using better bounding boxes for objects, but the solution is always imprecise. Some valuable ideas on tighter bounding boxes and their use were published by Weghorst et al. in [21]. There are two disadvantages of using tighter bounding boxes to determine if a given object and a cell intersects. These are the time and memory complexity to construct these tighter bounding boxes and the time complexity during the intersection test itself.

For the reasons mentioned above we propose a following technique. The spatial subdivision is constructed using the axis–aligned bounding boxes of objects, that is always quicker than to use the object–cell intersection test itself. In post–processing, all objects referred to in a given cell are checked if they have really intersection with the cell. If an object does not intersect the cell, it is removed from the list of objects. The ray–objects intersection tests are performed for all (axis–aligned) non–empty cells of the spatial subdivision.

The intersection of an object and an axis–aligned box is performed using precise intersection tests based on the geometry of the object. The example of such a test can be found in [9]. Robust intersection algorithms of an object and an axis–aligned box are simple for most of basic primitives. Due to the lack of space we do not discuss the intersection tests for the particular primitives in this paper. Unfortunately, we have not succeeded to find any survey of the box–object intersection algorithms.

The results achieved using the cleaning up of spatial subdivisions for ray tracing are also presented in the next section.

6 Results

In this section we present and discuss the efficiency for spatial subdivisions given in previous sections obtained by practical implementation. We follow the experimental measure $< \Lambda, \Delta >$ to compare the efficiency of spatial subdivisions given in [12] for presentation of our results. The meaning of the variables for this measure is shortly described in Table 1 (see also [11]).

	N_C	number of (non–hierarchical) cells of spatial data structure
	$R_{ETNC}[\%]$	(number of empty cells)/(number of all cells)
	$R_{EVWV}[\%]$	(volume of empty cells)/(volume of whole scene)
Δ	N_{ADC}	(number of references to objects)/(number of objects) - 1
	N_{AOIFC}	average number of objects in non-empty cells
	N_{RPRT}	the ratio of all intersection tests performed to minimal intersection tests
	N_{AT}	the average number of traversal steps per ray
	T_{CB}	the time required to build up spatial data structure
Λ	T_{TR}	the time of rendering itself (only intersections + traversal)

Table 1 Efficiency Measure: n-tuple Δ and Λ

In order to obtain some comparable results with other papers published the measure should be performed on a particular set of scenes. Since we support the proposal of *Standard Procedural Database* introduced by Haines in [10], our test scenes are chosen accordingly (balls, gears, mountain, rings, tetra, tree).

In papers published (e.g. [13]) there is some indication that the BSP tree is less efficient than the grid for some test scenes. We observed this is usually true for Kaplan's uniformly constructed BSP tree [15]. We implemented the 3D–DDA algorithm for grids [2], the BSP tree recursive traversal algorithm [14] [20], and the traversal algorithm for the BSP tree with rope trees. The BSP tree is built up using Formula 3 taking the minimum cost of all three coordinates. Splitting plane can lie outside the object–spatial median since the projections of objects onto axes may overlap.

It is worth mentioning that there are some invariants of ray tracing algorithm for a given scene regardless of acceleration techniques: the number of primary rays hitting the objects in the scene, the number of shadow and secondary rays etc. These should remain equal, otherwise, this fact reveal an incorrect implementation of an acceleration technique. Some invariants can be found in [10] and also in the SPD package distribution itself.

All images were rendered in 513×513 resolution. The maximal ray-recursion depth was set to 4. The results for tested scenes and methods are given in Table 2 in means of the experimental measure $< \Lambda, \Delta >$. In order to decrease further the time complexity of ray–casting the concept of *mailboxes* [2] was used for testing of all acceleration methods.

All BSP trees were constructed with following termination criteria: maximal depth was 18 and the number of primitives for a node to become a leaf was 2.

We have tried to select such a grid resolution, that is efficient for each particular scene. The grid resolution was based on a set of measurements with different strategies (regular, the heterogeneous setting presented by Klimaszewski in [16], $resolution = const.n^{\frac{1}{3}}$ for each side for n objects). The grid resolution producing the best performance results for each particular scene was chosen for the measurements presented.

The measurements were conducted on Pentium-MMX based PC, 166 MHz, 32 MBytes RAM, running under *Linux* operating system.

Discussion

The results in Table 2 show that the subdivision tree built up using the surface area heuristics is always quicker than Kaplan's BSP tree (constructed by splitting in the spatial median and with regular change of splitting planes orientation). This difference can be very significant, when the objects are non–uniformly distributed in the scene (scene balls and tree).

The grid performance is comparable to the BSP tree based on surface area heuristics, when the bounding box of a scene is of cubic shape and the objects are uniformly distributed (gears, mount, rings). Nevertheless, the surface area heuristics based BSP tree has better performance even in these cases. It should be mentioned that traversal algorithms for both spatial subdivisions (BSP trees, grids) were implemented as much as efficiently. The better efficiency of the BSP tree follows also from comparisons N_{RPRT} and N_{AT} parameters. The grid performance looses clearly if the scene contains large object(s) and most of the scene primitives are localized at one place (tree, balls).

The performance of Kaplan's method of BSP tree construction is rather unbalanced compared to the grid. It is due to the positioning of splitting plane in spatial median during its construction.

The cleaning up method does not improve the results very much for most of scenes. Its impact can be clearly identified on scene rings (8%), that contains a set of cylinders and spheres, whose bounding boxes are very loose and therefore they overlap. Parameters N_{ADC} and N_{AOIFC} are reduced correspondingly.

The traversal algorithm for BSP trees using ropes decreased the total rendering time in all cases from 10 to 20 %. There is apparent reduction of traversal steps (N_{AT}). The ratio of the number of interior nodes of all rope trees to the interior nodes of BSP tree varies between 3.77 and 6.27. Since the size of interior node of rope tree is much smaller than the size of a leaf of BSP tree, the total memory consumed is increased by a factor less than two at maximum.

The comparison with other methods published recently is rather impossible. Nevertheless, there is an indirect indication on scene *balls*, that the performance of surface area heuristics based BSP tree with cleaning up and rope trees is comparable to one of hierarchical grid published recently [16].

7 Conclusions

In this paper we have presented the concept of building and using ropes and rope trees for non–uniform spatial data structures. We gave comparisons of the efficiency for the BSP tree and the uniform spatial subdivision based on the experimental measure $< \Lambda, \Delta >$ for ray tracing. The results obtained indicate better efficiency of the adaptive BSP tree over the uniform spatial subdivision. Further, we have shown the efficiency of the ray–traversal algorithm using rope trees and the cleaning up technique for spatial subdivisions based on the real implementation.

8 Future Work

We currently work on another technique to accelerate ray tracing using a spatial pseudo–cover instead of the BSP tree. Although the splitting planes of a BSP tree (especially the ones close to root) are placed in a globally good position using the surface area heuristics, locally their influence can vary. Hence these planes can have rather unfavorable impact on the spatial subdivision locally. The behaviour may be improved by a spatial data structure based on the BSP tree, where some of the leaf–cells are joined together, if it is advantageous. The leaf–cells create a pseudo-cover of the whole scene space and therefore they cannot be traversed using the traditional recursive algorithm.

Acknowledgement

The work has been supported by Grant Agency of the Ministry of Education of the Czech Republic number 1252/1998.

References

[1]P. Agarwal, T. Murali, and J. Vitter. Practical techniques for constructing binary space partitions for orthogonal rectangles. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 382–384, 1997.

[2]J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10. North-Holland, Aug. 1987.

[3]J. Arvo. Linear-time voxel walking for octrees. *Ray Tracing News (available at htpp://www.acm.org/tog/resources/RTNews/html/rtnews2d.html)*, 1(5), 1988.

[4]J. Arvo and D. Kirk. *A survey of ray tracing acceleration techniques*, pages 201–262. Academic Press, 1989. In Book An Introdution to Ray Tracing, A. S. Glassner editor.

[5]J. G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, July 1988.

[6]M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.

[7]A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[8]A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, Oct. 1984.

[9]D. Green and D. Hatch. *Fast Polygon-Cube Intersection Testing*, pages 375–379. Academic Press, 1995. In Book Graphics Gems V, A.S. Paeth editor.

[10]E. A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, Nov. 1987. also in SIGGRAPH '87, '88, '89 Introduction to Ray Tracing course notes, code available via FTP from princeton.edu:/pub/Graphics.

[11]V. Havran. Evaluation of bsp properties for ray-tracing. In *Proceedings of 12th Spring School on Computer Graphics*, pages 155–162, Budmeřice, 1997.

[12]V. Havran. Spatial data structures for visibility computation, May 1997. Postgraduate Study Report, 34 pages, available at ftp://sgi.felk.cvut.cz/outgoing/havran/minimum/dc-psr-97-05.ps.gz.

[13]E. Jansen and W. de Leeuw. Recursive ray traversal. *Ray Tracing News (available at nic.funet.fi:/pub/graphics/misc/RTNews)*, 5(1), 1992.

[14]F. W. Jansen. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, editors, *Data Structures for Raster Graphics*, pages 57–73. Springer-Verlag, New York, 1986. Eurographic Seminar.

[15]M. Kaplan. Space-tracing: A constant time ray-tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, pages 149–158. Addison Wesley, July 1985.

[16]K. S. Klimaszewski and T. W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, Jan./Feb. 1997.

[17]J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990. criteria for building octree (actually BSP) efficiency structures.

[18]E. Reinhard, A. J. F. Kok, and F. W. Jansen. Cost prediction in ray tracing. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 41–50, New York, NY, 1996. Springer-Verlag/Wien.

[19]G. Simiakakis. Accelerating raytracing with directional subdivision and parallel processing, october 1995. available at ftp://ftp.sys.uea.ac.uk/pub/ah/G.Simiakakis_PhD.ps.

[20]K. Sung and P. Shirley. Ray tracing with the BSP tree. In D. Kirk, editor, *Graphics Gems III*, pages 271–274. Academic Press, San Diego, 1992. includes code.

[21]H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, Jan. 1984.

scene	method	n-tuples Δ and Λ								
		N_C	R_{ETNC}	R_{EVWV}	N_{ADC}	N_{AOIFC}	N_{RPRT}	N_{AT}	T_{CB}	T_{TR}
		$[-]$	[%]	[%]	$[-]$	$[-]$	$[-]$	$[-]$	$[s]$	$[s]$
balls	A	1245	29.6	56.6	1.04	17.2	207.0	65.7	1.17	315.6
(7382)	B	8950	16.7	0.2	1.63	2.61	13.0	25.0	3.53	68.4
	B_C	8950	16.7	0.2	1.17	2.14	11.1	25.3	5.32	68.2
bbox=	$C(2.02)$	8950	16.7	0.2	1.63	2.61	13.0	13.3	4.46	56.7
(24,24,	$C_C(2.02)$	8950	16.7	0.2	1.17	2.14	11.1	13.6	6.42	54.5
1.35)	$G_9^{152}(152)$	207936	88.6	88.6	3.59	1.43	84.7	17.4	1.45	159.5
gears	A	21484	21.5	48.3	10.7	6.48	24.1	31.6	2.94	373.0
(9345)	B	22341	4.1	9.8	4.90	2.58	7.89	22.4	3.5	337.6
	B_C	22341	4.1	9.8	4.5	2.41	7.75	22.4	13.9	327.1
bbox=	$C(1.59)$	22341	4.1	9.8	4.5	2.41	9.62	8.09	5.84	309.5
(4,4,1)	$C_C(1.59)$	22341	4.2	9.8	4.5	2.41	9.37	8.12	16.8	296.0
	$G_{25}^{98}(99)$	242550	85.7	85.7	8.17	2.47	12.6	19.1	2.61	369.4
mount	A	23981	20.1	76.0	9.37	4.44	17.8	30.1	2.3	84.2
(8196)	B	6999	18.9	85.8	0.27	1.83	6.79	22.3	1.58	60.6
	B_C	6999	18.9	85.8	0.22	1.77	6.57	22.3	1.76	60.3
bbox=	$C(1.62)$	6999	18.9	85.8	0.27	1.83	7.23	10.1	2.29	48.4
(2.2,2.2,	$C_C(1.62)$	6999	18.9	85.8	0.22	1.77	7.01	10.1	2.56	47.6
2.1)	$G_{35}^{35}(35)$	42875	87.3	87.3	3.59	6.89	23.6	15.2	1.33	91.5
rings	A	36696	7.7	68.4	14.2	3.77	36.8	60.4	3.3	253.0
(8401)	B	24102	10.8	70.0	5.68	2.61	16.5	39.7	4.37	167.3
	B_C	24102	10.8	69.9	3.76	1.86	13.3	39.7	13.3	157.6
bbox=	$C(1.81)$	24102	10.8	70.0	5.68	2.61	16.5	18.8	7.03	136.0
(19.7,19.3,	$C_C(1.81)$	24102	10.8	69.9	3.76	1.86	13.3	19.0	16.1	126.4
19.7)	$G_{61}^{61}(59)$	219539	84.4	84.4	12.6	3.32	31.7	33.4	2.92	207.5
tetra	A	34552	35.7	91.5	26.0	4.98	48.1	34.2	2.25	17.0
(4096)	B	2991	65.8	97.0	0.0	4.0	10.5	15.0	0.59	8.92
	B_C	2991	65.8	97.0	0.0	4.0	10.5	15.0	0.58	8.74
bbox=	$C(1.36)$	2991	65.8	97.0	0.0	4.0	10.5	10.5	0.72	8.19
(2,2,2)	$C_C(1.36)$	2991	65.8	97.0	0.0	4.0	10.5	10.5	0.75	8.25
	$G_{39}^{38}(39)$	57798	88.1	88.2	9.31	6.16	54.8	18.0	1.11	16.4
tree	A	458	64.9	55.7	0.268	64.5	2009	71.0	1.07	2090
(8191)	B	5514	32.9	0.0	0.531	3.39	24.1	14.4	4.11	54.2
	B_C	5514	32.9	0.0	0.482	3.28	23.7	14.4	5.33	54.2
bbox=	$C(2.01)$	5514	32.9	0.0	0.531	3.39	24.1	8.41	4.67	49.1
(100,100,	$C_C(2.01$	5514	32.9	0.0	0.482	3.28	23.7	8.42	5.73	48.6
3.2)	$G_{25}^{25}(25)$	15625	95.9	95.9	0.167	14.7	8916	23.5	0.81	10160

Table 2 The n-tuple Δ and Λ for SPD scenes: Methods: A - Kaplan's BSP tree with cyclic change of splitting plane ($x \mapsto y \mapsto z$) placed in spatial median with recursive ray–traversal, B - surface area heuristics BSP tree with recursive ray–traversal, B_C - surface area heuristics BSP tree with recursive ray–traversal and cleaning up, $C(x)$ - surface area heuristics BSP tree with rope trees ray–traversal (x - average number of neighbour leaf–cells per face), $C_C(x)$ - surface area heuristics BSP tree with rope trees ray–traversal and cleaning up (x as for $C(x)$), $G_y^x(z)$ - grid with 3D–DDA traversal, (x, y, z) is the grid resolution for all axes.