Postgraduate Study Report DC-PSR-97-xx

Spatial Data Structures for Visibility Computation

Vlastimil Havran

Supervisor: *Pavel Slavík*

May 1997

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

email: havran@cs.felk.cvut.cz
WWW: http://sgi.felk.cvut.cz/~havran

This report was prepared as a part of the project

———————————

This research has not been supported by any grant.

. . . .
Vlastimil Havran
postgraduate student

. . . .
Pavel Slavík
supervisor

Table of contents

Spatial Data Structures for Visibility Computation

Vlastimil Havran
havran@cs.felk.cvut.cz
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Abstract

In this report we deal with two of the basic problems of computer graphics required for rendering: visibility of two-point computations and ray-casting. The first part of the report is devoted to the introduction of spatial data structures designed to decrease the time complexity of these problems. Most of the report presents our own ideas concerning spatial data structures developed in the past year. They concern the experimental evaluation of these spatial data structures, the positioning of a splitting plane for a BSP tree, and cache sensitive mapping for a BSP tree in the memory.

Keywords

computer graphics, rendering, spatial data structures, BSP tree.

1 Introduction

The principal goal of computer graphics is the image synthesis of a scene representing reality. The algorithms for image synthesis have a different time complexity and quality of their outputs. The main effort devoted to the research in the area of image generation is oriented to the synthesis of high quality and *photo-realistic* images. By a photo-realistic image, we mean an image indistinguishable from a photograph of the real world. The scene is modelled by geometric object primitives; it is not exceptional that their numbers may reach hundreds of thousands for one scene.

In this paper we will refrain from such problems of image synthesis concerning object modelling, shading models, and global illumination. We will focus on the problem of time and space complexity.

There have been developed two main classes of algorithms for photo-realistic rendering: *ray–tracing* and *radiosity*. These are sometimes combined together to overcome their opposite shortcomings. The common property of both algorithms is their high time and space complexity. The algorithms spend most of the time repeatedly computing *visibility* for pairs of points in the scene. The problem is more formally defined as follows: two points (x, y, z) and (x', y', z') are mutually visible if the abscissa connecting them does not intersect any object located in the scene. The computation of the visibility is indispensable to determine correctly the global illumination and shading of objects by light sources. The second important problem of image generation is *ray-casting*: for a ray given by its origin and direction vector, we want to find the closest object which is intersected by the ray if an intersection exists.

The time devoted to solve both of these problems is typically 95% or more of total rendering time. The rest of the time is consumed by specific computation and cannot be decreased. In case of ray–tracing it is necessary to determine the reflected and refracted rays, evaluate colour of pixels, etc. For these reasons the main effort concerning the algorithms for image synthesis is aimed to reduce the time and space complexity of visibility computation.

The report is organized as follows. Chapter 2 introduces basic definitions and terminology needed for comprehension of this report. Chapter 3 summarizes the previous results and work in the area of visibility computation for high-quality images. Chapter 4 describes our advances and approaches to the problem. Chapter 5 outlines the open problems and ideas that should be solved in our future work. Chapter 6 concludes the report.

2 Basic definitions

In this opening we define the basic terms required for better comprehension of further chapters.

Definition 1 Let *ray* in n-dimensional space be determined by its origin and direction vector. The ray corresponds geometrically to the half-line in n-dimensional space.

Definition 2 Let A_1 and A_2 be a pair of points in n-dimensional space. Then *pair of points visibility algorithm* solves the following problem: pair of points are mutually visible if the abscissa connecting them does not intersect any object located in n-dimensional space. Contrarily, the objects are not mutually visible, i.e., there is at least one intersection with object(s) and the abscissa connecting the pair of points.

Definition 3 Let *ray-casting* be a problem described as follows: For a given ray find the closest object which is intersected by the ray if such an object exists.

Definition 4 Let C^n denote a *cell* in n-dimensional space. C^n is defined as a n-dimensional region determined by continuous $(n-1)$-dimensional boundaries. The boundary of the cell does not cross itself. Let ∂C^n denote the boundary of cell C^n.

Definition 5 The cell C^n is *separating* if its boundary splits the n-dimensional space into two disjoint parts $\rho(C^n)$ and $\tau(C^n)$ with the following properties: for each pair of points if one is in ρ and the second in τ, the abscissa connecting the points intersects the boundary of the cell.

Definition 6 We call *ϵ-surrounding* of point A in n-dimensional space P_n the following set of points:

$$surrounding(P, \epsilon) = \{x \in P_n, \|A - x\| \leq \epsilon\}$$

Definition 7 The cell C^n is *closed* if it is separating and one of the parts $\rho(C^n)$ or $\tau(C^n)$ is finite. The finite part of the cell is called *inner* and the other one is *outer*.

Definition 8 The cell C^n is *elementary* if its inner part $\rho(C^n)$ does not contain any other cell or any part of other cell.

Definition 9 The cell C^n is *hierarchical* if it contains fully a set of elementary cells $SE(C^n)$ or a set of hierarchical cells $SH(C^n)$.

Definition 10 *Spatial subdivision* (SSD) of a cell G^n is such a finite ordered set K of cells, that for each point $P \in G^n$ exists a cell C^n, $C^n \in K$, $P \in C^n$.

Definition 11 *Elementary spatial subdivision* (ESSD) of a cell G^n is SSD, so that SSD is composed of a finite ordered set K of closed, disjoint, separating, and elementary cells with the property: $V(G^n) = \sum_{i=1}^{i=k} V(C_i^n)$, where $V(C^n)$ is the volume of the cell C^n.

Definition 12 Two cells C_1^n and C_2^n are *neighbours* for a particular ESSD if and only if

$$\partial C_1^n \bigcap \partial C_2^n \neq \emptyset$$

Note: Usually, the neighbouring information is formed during the process of constructing ESSD. The alternative is to construct the neighbouring information on the demand after constructing ESSD.

Definition 13 Let *hierarchical spatial subdivision* (HSSD) of a cell G^n be two finite sets E and H. Let E be a set of elementary cells and H a nonempty set of hierarchical cells.

3 Related work/Previous results

In this chapter we describe the application of visibility computations and the ray-casting problem to advanced rendering techniques. Further, we outline the principle and properties of basic accelerating methods for visibility computations developed in the past.

3.1 Ray-tracing

In this section we recall the fundamentals of the ray-tracing algorithm for better understanding of the derivation of time complexity given throughout this report. Let us suppose the objects in the scene are described by their geometrical and optical properties. The basic principle of the ray-tracing algorithm is the simulation of the real world by means of geometrical optics.

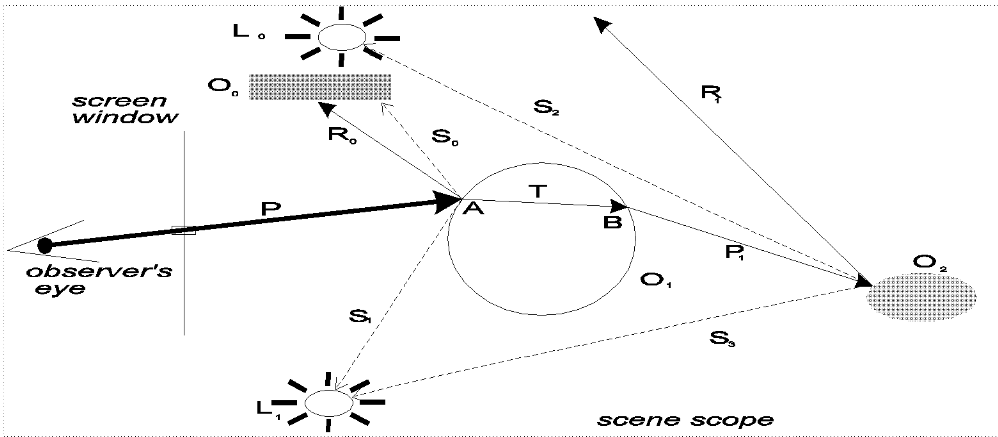

Figure 1: Basic concept of ray-tracing

The resulting image is created as follows (see Fig. 1): For each pixel in the screen window we cast a ray P towards the scene space and solve the problem of ray-casting. If this so-called *primary* ray does not intersect any object in the scene space, the colour of the corresponding pixel in the screen window is a background colour. Otherwise, there is an intersection point A, and we solve the visibility problems between A and all the point lights located in the scene by *shadow* rays (S_0, S_1, S_2, S_3). If the surface properties of the object primitives include the reflectiveness or refractiveness, then *reflected* (R_0) and *refracted* (T) rays (they are also called *secondary* rays) to solve the ray-casting problem are generated. The process of generation of secondary rays is performed recursively until some *depth of recursion* is reached. The ray generation process corresponds to the binary tree.

The light contributions corresponding to the direct illumination by light sources and indirect illumination by reflected/refracted rays are computed from the surface properties and the mutual geometrical position of the rays and the light sources.

3.2 Visibility computation

In this section we present the methods for acceleration of visibility computations. We derive the time complexity of a naive algorithm and then show the methods that reduce it. The time complexity will be described for the *ray-tracing* algorithm for ease of understanding. The number of visibility computations for radiosity is much higher than for ray-tracing, and therefore the question of time complexity of visibility computation is also very important.

3.2.1 Naive approach

The naive approach does not use any acceleration. The following pseudo-codes outline the naive approach for ray-casting and for visibility computations.

```
function visibility(point_A, point_B):boolean
  A:boolean;
  N,i:integer;
begin
  N:=number of objects;
  construct the abscissa L between point_A and point_B;

  for(i:=1;i<=N;i:=i+1)
  begin
    A:=exists the intersection with object[i] and the abscissa L;
    if (A=TRUE) then
      break;
  end
  visibility:=not A;
end


function ray_casting(ray_origin, ray_direction):object
  t,res_t:real;
  N,i,index:integer;
begin
  N:=number of objects;
  res_t:=infinity;
  index:=-1;

  for(i:=1;i<=N;i:=i+1)
  begin
    t:=calculate the closest positive
      intersection of ray with object[i];
    if (t<res_t) then
    begin
      t:=res_t;
      index:=i;
    end
  end
  ray_casting:=object(i);
end
```

3.2.2 Time complexity

Let us discuss the time complexity of the naive approach for ray-casting and visibility.

Lemma 1 *The pair of points visibility and ray-casting problem has $O(n)$ time complexity for the naive approach.*

PROOF: a) visibility problem: in the worst case the ray is checked with all objects in the scene space, the last test with object can occlude the points. b) ray-casting: the closest object to a ray-origin has to be selected, and in the worst case it is the last one.

It is evident the naive method is computationally complex. Most of the total rendering time (more than 95 percent) is devoted to the computation of the intersections of the ray with objects [71]. The time requirements depend particularly on the shape of the objects in the scene. The smallest time requirements to compute the intersection are for sphere; they are significantly higher for the objects as quadrics and NURBS.

Lemma 2 *Let us consider the allowed depth of recursion h, the number of pixels in the image $width \times height$, and the number of lights l_m. Then for n objects in the scene the upper limit of number of calculations of intersection of a ray with object is expressed as follows:*

$$I_{max} = O(R_{max}.n),$$

where R_{max} denotes the maximal number of rays generated:

$$R_{max} = O(width \times height \times 2^{(h-1)} \times (l_m + 1))$$

PROOF: The generation of secondary rays for one pixel is described by generation of a binary tree, which is expressed by term 2^{h-1}. It is necessary to compute the shadow ray for every intersection point; it is expressed by the term $(l_m + 1)$. The number of pixels to be computed is $width \times height$, and the worth time complexity of both ray-casting and the visibility naive algorithm is $O(n)$.

For instance, if the $width = 800$, $height = 600$, $n = 1000$, $l_m = 2$, and $h = 3$, the whole picture requires computing the huge number 10.08×10^9 of intersection calculations.

Lemma 3 *The visibility problem can be solved by ray-casting.*

PROOF: We proof the lemma by algorithm construction. For a pair of points A and B we construct a ray. Then we perform a ray-casting algorithm. There are three possibilities:
a) there is *no intersection* of a ray with any object - points are mutually visible
b) the closest intersection point lies *outside* the abscissa connecting the pair of points - the points are mutually visible
c) the closest intersection point lies *inside* the abscissa connecting the pair of points - the points are not mutually visible

It is obvious that all objects are not transparent and specular, but on the other hand, the number of the primitives in the scene is usually much higher than for example those given above. The computational complexity is also decreased if the reflected ray does not hit any object and leaves the scene space completely. The increase of the depth of recursion leads to the comparatively slight increase of the time complexity for real scenes. Nevertheless, the number of the intersection calculations is still very large. The time complexity $O(n)$ of this naive approach makes it practically unusable for real rendering applications.

3.3 Introduction to algorithmic improvements

This section gives a survey of algorithmic approaches to decrease the time complexity of visibility computation and ray-casting. We try to determine time complexity for all algorithms, but we must state beforehand, the derivation is cumbersome and it is made under simplifying circumstances. The researchers have done some attempts to determine time complexity of accelerating techniques [55] [13] in the past, but theoretical analysis is rather scarce and differs with the results of measurements. The problem is that we can determine time complexity under simplifying conditions for the worst case, but the average time complexity is not strongly connected with worst time complexity. Therefore the researchers often have recourse to the measurement on some benchmarking scenes and compare the times measured with some reference algorithms.

The algorithm to decrease the time complexity of visibility computation and ray-casting are based either on space subdivision schemes [68], the hierarchical clustering of objects [7], or the combination of these principles [31]. The space subdivision approach is more common and less computationally expensive than the hierarchical one.

3.4 Bounding volumes

A naive ray-tracing algorithm tests every object for intersection with a given ray. The intersection test itself is an expensive operation. Therefore it is advantageous to enclose the objects in a *bounding volume* (often called bounding box) with a simple ray intersection test. Then if the ray intersects the bounding volume, the ray intersection with the object is performed.

A simple method uses spheres as bounding volumes. The second possibility is to use rectangular parallepipeds parallel to coordinate axes. Another alternative uses arbitrarily oriented rectangular parallepipeds [21]. The mutually opposite requirements posed on the properties of bounding volume are as follows:

- the probability of intersection with the object if it intersects the bounding volume is high.
- the time complexity of intersection calculation of a ray with the bounding volume is small.

From these demands we can further deduce for polyhedron representation that the bounding volume should be a convex cell with a small number of polygon boundaries.

Hierarchical bounding volumes

A natural extension to bounding volumes is a hierarchy of bounding volumes (*HBV* in the following text). Bounding volume hierarchy takes advantage of hierarchical coherence. Given the bounding volumes of the objects, a n-ary tree of enclosing volumes is created with the bounding volumes of the objects at the leaves and at every intermediate node a bounding volume that encloses completely the volumes of the subtrees. The construction of HBV proceeds *bottom-up*.

The hierarchy gives naturally the method for testing the ray with the objects. If the ray does not intersect the enclosing volume at the root, it does not intersect any object. If the ray intersects the root, the tree is recursively descended to the leaf to test for ray intersections with the bounding volumes of the subtrees. The method for construction of HBV was first described in [22].

3.5 Spatial subdivision structures

Spatial subdivision is another very popular method to decrease the number of object-intersection tests. The basic method was developed independently by other authors ([29] [18] [19]). The common principle of all spatial subdivision structures is to divide the cell C^n into a set of cells

$S(C^n)$ by a set of $(n-1)$ dimensional boundaries $S_b(\partial C^n)$. In terms of definitions given in Chapter 2 to create ESSD or HSSD over initial cell, each elementary cell contains a list of objects fully or partially contained in the cell. The visibility and ray-casting problem are thus localized to the elementary cells. If any intersection exists with an object belonging to the elementary cell and if the intersection point lies in the elementary cell, then the intersection point is found. If we have more intersection points inside the subvolume, the closest one is selected. If the intersection does not exist or the intersection point lies outside currently processed elementary cell, the computation is proceeded to the next elementary cell along the path of the ray.

The elementary cells are either of the same shape, structure, and size or irregular. The common property of the spatial subdivision schemes as opposed to the hierarchical bounding volumes is that the elementary cells are disjointed (non-overlapping). The constructed subvolumes are either addressed directly or there is some hierarchy constructed over the elementary cells represented by hierarchical cells. The construction of spatial subdivision structures is created using a *top-down* approach. The spatial data structures are space oriented instead of object oriented for the hierarchical methods.

Let us describe the spatial subdivision schemes in more detail.

3.5.1 Binary space partitioning

A *Binary Space Partitioning* (BSP) tree is a spatial data structure that can be used to solve a variety of geometrical computational problems. It was initially developed as a means of solving the hidden surface problem in computer graphics [23].

It is the analogue to the search binary tree, but the data in the BSP tree represent n-dimensional data. The BSP tree hierarchically subdivides an initial cell C_p^n containing a collection of objects defined in n-space. The tree is formed by recursively subdividing the cell C^n in two cells C_{left}^n and C_{right}^n, usually halves. The resulting data structure is a binary tree in which each interior node represents a partitioning hyper-plane and its children represent convex sub cells determined by the partitioning. The leaf nodes of a BSP tree are convex non-overlapping elementary cells.

The leaves of a BSP tree are either occupied fully or partially by objects or vacant. The construction of the tree is done recursively by subdividing the space in the mid-point until the number of objects in a currently subdivided cell is smaller than a constant or the depth of the cell in the tree is equal to a constant. The algorithm as described above is currently the most commonly used. Typical threshold value for maximal number of objects in a leaf is about 3, the typical upper limit for depth is 30. The elementary cell is often called *leaf* and the hierarchical cell (inner), *node*. The partitioning process is depicted in a Fig. 2.

The important factor influencing the construction of a BSP tree and its resulting properties is the splitting criterion for positioning the splitting plane. The requirements posed on the BSP tree for visibility computations are:

- well balanced,
- low depth,
- memory efficient.

The following pseudo-codes outline the principle of construction of a BSP tree and traversing a ray through the tree [60].

The construction of a BSP tree over objects in the scene is done as follows:

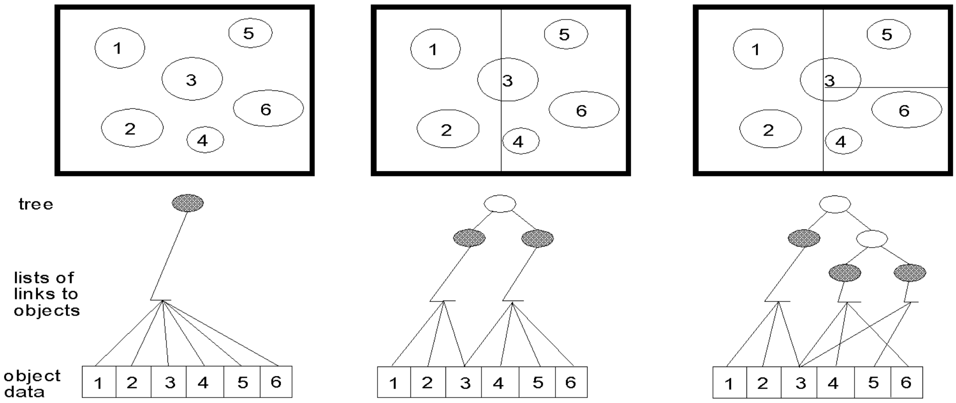

Figure 2: Partitioning of space by a BSP tree

```
procedure Subdivide(CurrentNode, CurrentTreeDepth, CurrentSubdividingAxis)
begin
  if ( (CurrentNode contains too many primitives) and
       (CurrentTreeDepth is not too deep) )
  begin
    Children of CurrentNode := CurrentNode's Bounding Volume;
    { Note that child[0].max.DividingAxis and
       child[1].min.DividingAxis are always equal. }
    if (CurrentSubdividingAxis = X) then
    begin
      child[0].max.x=child[1].min.x := mid-point of CurrentNodes's X-Bound
      NextSubdividingAxis := Y-Axis
    end
    else
      if (CurrentSubdividingAxis = Y) then
      begin
        child[0].max.y=child[1].min.y := mid-point of CurrentNodes's Y-Bound
        NextSubdividingAxis := Z-Axis
      end
      else
        if (CurrentSubdividingAxis = Z) then
        begin
          child[0].max.z=child[1].min.z := mid-point of CurrentNodes's Z-Bound
          NextSubdividingAxis := X-Axis
        end
    for ( each of the primitives in CurrentNode's object link list) do
      if ( the primitive is within children's bounding volume ) then
        add the primitive to the children's object link list
    Subdivide ( child[0], CurrentTreeDepth+1, NextSubdividingAxis )
    Subdivide ( child[1], CurrentTreeDepth+1, NextSubdividingAxis )
  end
end
```

The traversal of a ray through a BSP tree:

```
function RayTreeIntersect(Ray, Node, min, max):object
begin
  if ( Node is free ) then
    RayTreeIntersect:="no intersect";
  else
    if ( Node is a leaf) then
    begin
      intersect Ray with each primitive in the object link list
        discarding those farther away than "max";
      RayTreeIntersect:="object with closest intersection point";
    end
    else
    begin
      dist := signed distance along Ray to the cutting plane of the Node;
      near := child of Node for half-space containing the origin of Ray;
```

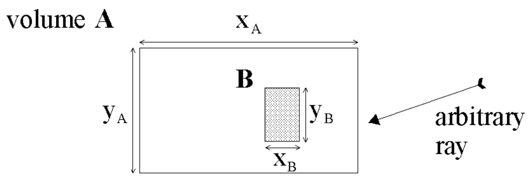

Figure 3: Computing conditional probability that the ray hits object B once it passes through volume A

```
        far := the "other" child of Node - i.e. not equal to near
        if ( (dist>max) and (dist<0) ) then { Whole interval is on near side }
            RayTreeIntersect := RayTreeIntersect(Ray, near, min, max); { recursion }
        else
          if (dist<min) then { Whole interval is on far side = recursion }
            RayTreeIntersect := RayTreeIntersect(Ray, far, min, max);
          else
          begin { the interval intersects the plane }
             { recursion }
            hit_data := RayTreeIntersect(Ray, near, min, dist); { test near side }
            if (hit_dat indicates that there was a hit) then
              RayTreeIntersect := [hit_data];
             { recursion }
            RayTreeIntersect := RayTreeIntersect(Ray, near, dist, max);  { test far side }
          end
      end
  end
```

Both pseudocodes are written recursively for ease of understanding. The efficient source code is organized in such a way that recursive calls are omitted by maintaining an explicit stack in the inner loop.

The whole process of construction of a BSP tree was modified and improved by MacDonald and Booth [37]. Let us discuss the improvements in more detail.

3.5.2 Statistical optimization of a BSP tree

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

MacDonald and Booth [37] used simple heuristics for finding the optimal position of a splitting plane. The plane remains perpendicular to one of the main coordinate axes, because of simple computations performed later during traversal phase. The plane position is determined by minimizing a **cost function**. The cost function is based on the probability that a ray hits the object placed inside a certain volume once it passes through that volume as shown in Fig. 3.

Let us suppose that both object B and a volume A are of convex shape. It is mostly fulfilled because objects are often temporarily replaced by their bounding volumes during construction of a binary tree. 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 a volume A ([58] [21]):

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

The cost function can be expressed by the conditional probability. It expresses the estimated time needed for traversing one ray through the specified tree. During the building of a binary tree, the cost function helps to decide when and where to split certain subspace, i.e., to replace one leaf node by a new internal node with two children – leaves. Let us assume the situation

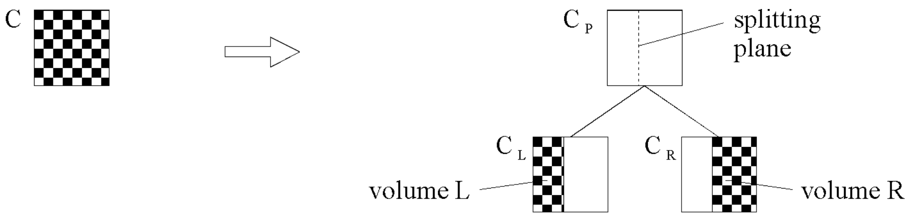

Figure 4: New costs after one subdivision step

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 scene 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 internal nodes. The cost has to incorporate time needed for traversing all nodes visited by a ray. Let us suppose the splitting plane is perpendicular to x axis. Figure 4 shows the geometrical factors that influence the change of the cost for one space subdivision step.

The node on the left side in Fig. 4 has been replaced by a new tree structure on the right side in Fig. 4. Original cost C is changed to a new 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 conditional probability that a ray hits the node L or R once it visits the parent node P. 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 (internal) node
	S_L, S_R	is a surface area of left subspace and right subspace respectively
	S	is the surface are of the node to be subdivided
	n_L, n_R	is the number of objects belonging to the volume L and volume R respectively

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

The aim is to build the optimal binary tree with minimal 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 volume 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 volume).

The minimum of the cost function can be roughly estimated using a few sample splitting planes. MacDonald and Booth showed that there are two important positions of the splitting planes. One position is in the geometrical center of the volume. Let us call it **spatial median**. The second position is in the middle of an ordered list of objects*, 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.

Although almost every splitting plane intersects some objects (especially for complex scenes), this simple condition quickly gives relatively good estimation of its optimal position.

3.5.3 Octree

Octree (Octal tree) is recursive data structure that is similar to the quadtree in two-dimensional space. In the opposite to the BSP tree the initial cell C_p^n is not split into halves but into 2^n cubic cells. The cubic cells, are often called *voxels* and they can vary in size. The general definition defines the octree for $n \in N$, but the usual convention for an octree is restricted to the three-dimensional case. The octree itself can serve as the representation of a three-dimensional model; the leaves of the octree are either denoted empty or full. Another usage is the acceleration technique for computing visibility and the ray-tracing. The use of an octree for this purpose was introduced by Glassner [19]. The cells with high object complexity can be recursively subdivided into smaller and smaller cells, generating new nodes in the octree.

The addressing of child nodes is provided by direct pointing or hash table. In case of hashing the denotation of the child node is usually done by postfixing or prefixing the parent denotation by ciphers from 1 to 8 corresponding to the geometrical position of the child node (see Fig. 5). The numbering the nodes this way (instead of from 0 to 7) looses the octal purity of the original scheme and improves the hashing itself.

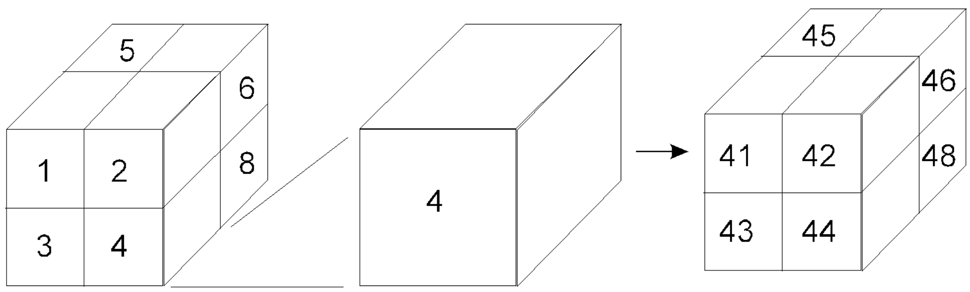

Figure 5: Octree subdivision

We should note that the traversal of an octree in sorted order along the ray path is more complicated than for a BSP tree or SEADS (see subsection 3.5.4). It is due to the more complex decision computation for each traversal step. The octree naturally exploits the spatial coherence because objects that are close to each other in space are represented by leaves that are close to each other in the octree. The termination criteria for octree construction is the same as for a BSP tree: the maximal allowed depth and the minimal number of objects in leaves.

The properties of an octree from the point of view of visibility computations are rather worse. Each node can cover a small object and the intersection calculation has to be performed each time entering the node, which can be very large. The intersection calculation is also performed if the probability of a successful intersection with an object in the leaf of an octree is quite small. The next disadvantage can be considered small occupancy of leaves. The empty neighbouring

*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 leaves have to be determined and traversed. The small occupancy of leaves implies relatively large memory requirements for octree representation.

There is the modification of surface area heuristics for octree data structures called Octree-R [70]. If the surface area heuristics is applied to the octree structure to each subdividing plane independently, we can decrease the rendering time from 4% up to 47% depending on the scene characteristics.

3.5.4 SEADS

Spatially Enumerated Auxiliary Data Structure (SEADS in the following text) is another method of spatial subdivision. It involves the subdivision of initial cell C_p^n into equally sized elementary cells regardless of the occupancy of objects. The n-dimensional grid resembles the subdivision of a two-dimensional screen into pixels. The list of objects that are partially or fully contained in the cell is assigned to each parallepiped cell (also called *voxel* for three-dimensional space). The method was first introduced in [18] and the traversal algorithm of this data structure was improved by Hsiung [26] and Endl [17].

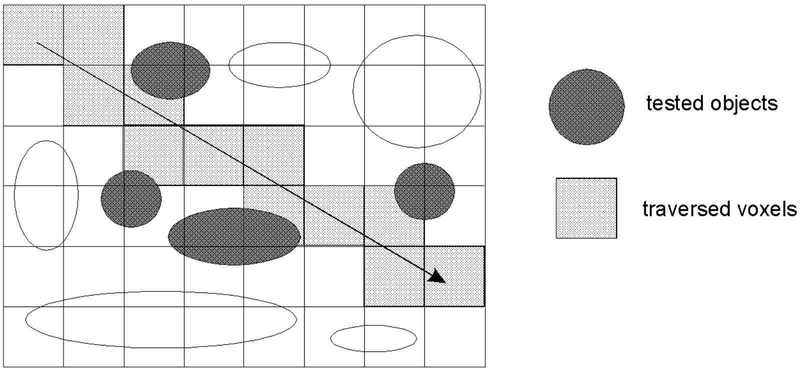

Figure 6: SEADS - grid structure

Since the grid is created regardless of occupancy by objects, a SEADS subdivision forms many more voxels and therefore it demands necessary storage space. Nevertheless, the traversal of SEADS structure can be performed very efficiently by a 3D-DDA algorithm. It is analogous to the algorithm for drawing a straight line in two-dimensional space and requires thus a simple operation for each traversal step (addition, subtraction, and comparison). A ray is only tested with the few grid elements that are traversed by the ray.

The problem with the SEADS is the occupancy of most voxels is very small, and therefore the ray traverse a lot of empty voxels before hitting the full voxel. The method is the only one from all accelerating methods that was analyzed well enough in [13]. The analysis shows the optimal division for one axis is for $k = const.N^{\frac{1}{3}}$ and the minimal total time per ray corresponds to $t_{min} = const.N^{\frac{1}{3}}$ for the N objects of the same shape and size in the scene.

3.6 Ray-space subdivision methods

The method is based on the ray-space subdivision by 5-dimensional coordinates. The first three coordinates of the ray-space are Euclidean and they express the origin of the ray. The next two coordinates are spherical and determine the direction of the ray. The space subdivision is thus as follows:

$$R5 = E^3 \times \sigma^2$$

Let us suppose we have N objects in the cell, which are indexed from 0 to $(N-1)$.

Definition 14 The assignment function $f_A(x,y,z,\varphi,\rho)$ is the discrete function $f : E^3 \times \sigma^2 \rightarrow Z_0^+ \bigcup\{-1\}$ that solves the ray-casting problem and is defined as follows:

$$f_A(x,y,z,\varphi,\rho) = i \begin{cases} i = k, k \in< 0, N-1 > & \text{corresponds to the closest object intersected} \\ i = -1 & \text{if the ray does not intersect any object in the cell} \end{cases}$$

The space, over which function f_A is defined, has to be discretized for practical use. In the case the direction of rays in discretized positions creates generalized rays, and the f_A should return the *candidate list* of indexes instead of one index to object primitives that are visible from the origin point. The origin of the ray is also discretized and corresponds to the cell, and the total volume which can be reached by the ray is called *hyper-cubic region*.

The ray-acceleration scheme was suggested by Arvo and Kirk [3] and further elaborated by Simiakakis [57]. The ray-space can be achieved by binary partitioning. Whatever the improvements of the algorithm, the method suffers from an algorithmic paradox: the construction of the candidate lists is much more difficult than with the space subdivision schemes. Arvo and Kirk report the high time complexity of detecting polyhedral intersections and suggest the approximation where hypercubic regions are bound by cones [21]. Nevertheless the time complexity is still higher than for space subdivision techniques.

4 Our work and results

4.1 Methods for experimental evaluation and comparison of ASDS

The problem of *auxiliary spatial data structure* (ASDS in the following text) is the evaluation and the comparison of time and space complexity with regard to input data. It is rather impossible to determine the time and space complexity by means of asymptotic comparisons. It is due to the relatively high complexity of traversal algorithms and the input data dependency. This property of different acceleration schemes was used to demonstrate the advantages of some methods proposed by scientists on particular scenes that fitted well their accelerating method.

Standard Procedural Data method

The way to encounter this problem partially was introduced by Haines [24] who proposed a *Standard Procedural Data* method (SPD in the following text). It is a collection of the scenes which are generated by a program. The description of scenes consists of basic primitives, and it can be converted easily to the format used by different rendering packages. The idea is promising, but the requirements posed on performance of current rendering systems are much higher than the ones in the time of introduction of the SPD in 1987. The scenes processed today contain up to hundreds of thousands of object primitives. SPD models are constructed by fractals and the number of primitives is rather small. That is why scientists in research papers use some scenes which are more complex and correspond to the time of paper issue. For the same reasons we have selected some SPD data (see Fig. 7) and other scene data (see Fig. 8) to measure the performance of different accelerating techniques. The images presented in this report were computed by a ray-tracing algorithm.

We have perused the ways for spatial data construction and their traversal for visibility computation. The time complexity is highly connected with the number and distribution of objects in the scene. The properties of algorithms for visibility computation can be evaluated for some specific algorithms. We selected the ray-tracing algorithm for the evaluation because of its simpler implementation. Moreover, the ray-tracing actually solves only the visibility algorithm. In case of radiosity it is necessary to compute some additional operations (matrix solving, meshing) to get the resulting image.

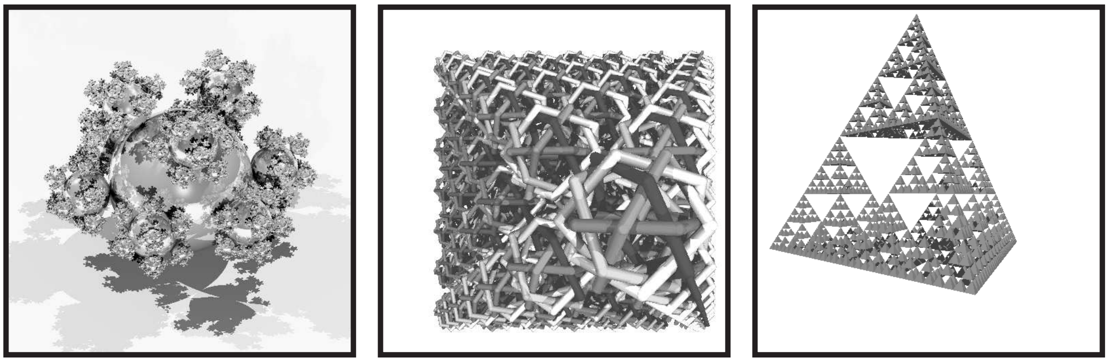

Figure 7: Standard (SPD) scenes – *balls*, *rings*, and *tetra* (resolution 512×512)

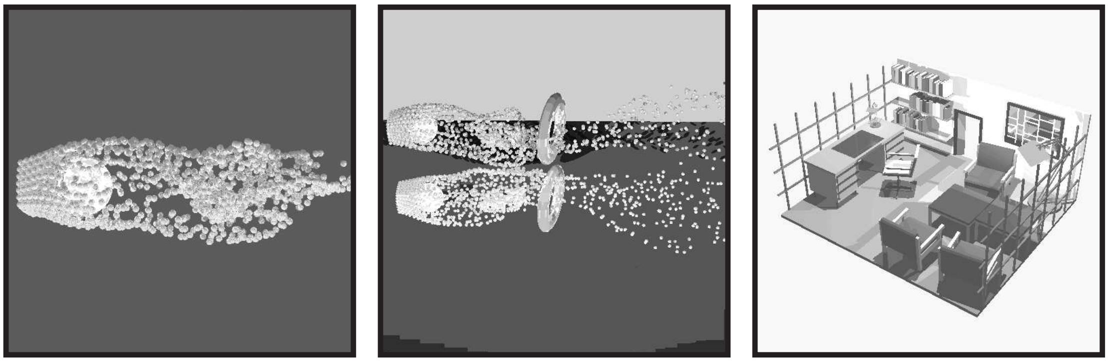

Figure 8: Specific scenes – *fluid*, *m-fluid*, and *room* (resolution 512×512)

The evaluation of the algorithm is based on two sets of tasks to be computed. The first set $\Omega(N_{RC})$ specifies N_{RC} of ray-casting tasks, the second set $\Upsilon(N_{VC})$ specifies N_{VC} of visibility ones. These sets are always the same to obtain a given image for all acceleration techniques, but each algorithm handles the sets with different time and space complexity.

The ASDS properties can be grouped into two parts. The first one includes *static* properties corresponding to the ASDS construction and the second one reflects the *dynamic* behaviour of ASDS during the visibility computation. These properties are mutually connected with the *scene characteristics* over which ASDS is built.

Let us classify the properties concerning ASDSs into some groups. Each parameter belongs to one or more groups. The parameters express both maximal and average values. The groups are marked by following letters:

- B .. specified *before* computation, independent of ASDS properties and the implementation itself. They depend on the distribution and geometrical properties of the objects in the scene and other required parameters.
- D .. *derived* from the B parameters, but they are not specified by the user.
- C .. computed from the *construction* of ASDS, independent of $\Omega(N_{VC})$ and $\Upsilon(N_{RC})$ tasks. They describe how the ASDS meets with the object distribution in the scene.
- R .. computed from *rendering* , they are dependent on B, D, and C. They reflect dynamic behaviour of ASDS with respect to $\Omega(N_{RC})$ and $\Upsilon(N_{VC})$.

- T .. *time* parameters dependent on implementation, compiler and architecture used. They correspond to the real time consumed to obtain the resulting image.

N_X,N_Y	B	the resolution of the image
N_O	B	the number of scene objects and their distribution in the scene
N_L	B	the number and position of lights in the scene
OP_{CAMERA}	B	the position, the orientation, and other settings of camera
S_{COV}	B,D	the percentage of screen coverage
D_{COMPX}	D	depth complexity, i.e., the average number of object primitives that are hit by an arbitrary ray from the viewpoint (see [55])
V_{SCENE}	B,D	the volume taken by the bounding box of the whole scene
V_{BB}	B	the sum of volumes specified by bounding boxes of object primitives
$R_{BBSC} = \frac{V_{BB}}{V_{SCENE}}$	B	the ratio of volumes for bounding boxes to the volume of whole scene
D_{AREC}	B	maximal depth allowed for secondary rays

Table 1 Scene and image characteristics

TC_{MAX}	B	termination criteria for construction of ASDS
$D_{REACH} \leq D_{MDEPTH}$	C	the maximal depth reached if any
N_C	C	the number of cells
N_{EC}	C	the number of empty cells (without objects)
$R_{ETNC} = \frac{N_{EC}}{N_C}$	C	the ratio of empty cells to all cells
N_{RO}	C	the total number of references to objects from all (non-empty) elementary cells
$N_{ADC} = N_{RO}/N_O - 1.0(\geq 0.0)$	C	the average number of objects' duplication for one object in elementary cells
$N_{AOIC} = \frac{N_{RO}}{n_C}$	C	the average number of objects in all cells
$N_{AOIFC} = \frac{N_{RO}}{n_C - n_{EC}}$	C	the average number of objects in full cells
V_{EMPTY}	C	the sum of volumes taken by empty cells
$R_{FVWV} = \frac{V_{SCENE} - V_{EMPTY}}{V_{SCENE}}$	C	the ratio of volumes of full cells to V_{SCENE}
T_{CB}	T	time required for construction of ASDS

Table 2 Static properties of ASDS

Tables 1-3 give the lists of some parameters describing some properties of ASDSs. It is even possible to enlarge the set of parameters in Tables 1-3 by other mean and maximal values or to relate the parameters to specific groups of rays, but these extensions are rather useless for the purposes of evaluation.

Definition 15 The *traversal step* is the elementary operation to make a pass between two cells. These two cells are either neighbouring and elementary or they have a hierarchical relationship.

$N_{PR} = N_X \times N_Y$	D	the number of primary rays
N_{PRIT}	R	number of intersections tests for all primary rays and object primitives
$N_{HPR} = N_{PR}.S_{COVERAGE}$	D	the number of primary rays hitting the objects
N_{SR}	D	the number of shadow rays
N_{SRIT}	R	the number of intersection tests carried out for all shadow rays
N_{HSR}	D	the number of shadow rays hitting objects
N_{SECR}	D	the number of secondary (reflected + refracted) rays
N_{SECRIT}	R	the number of intersection tests performed for all secondary rays
N_{HSECR}	R	the number of secondary rays hitting the objects
$N_{RPRT} = \frac{N_{PRIT}+N_{SRIT}+N_{SECRIT}}{N_{HPR}+N_{HSR}+N_{HSECR}}$	R	the ratio of all intersection tests performed to minimal intersection tests
N_{TEC}	R	the number of traversed elementary cells
N_{TEEC}	R	the number of traversed empty elementary cells
$N_{TEFC} = N_{TEC} - N_{TEEC}$	R	the number of traversed non-empty elementary cells
N_{ITHC}	R	the number of traversed hierarchical cells
$N_{TAC} = N_{TEC} + N_{ITHC}$	R	the number of all traversed elementary and hierarchical cells
$N_{AT} = \frac{N_{TAC}}{N_{PR}+N_{SR}+N_{SECR}}$	R	the average number of traversed cells per one ray (primary, secondary, shadow)
T_{TR}	T	time required for the rendering itself of the image for a specific ASDS
T_{TT}	T	time devoted only to traversing ASDS ($T_{TT} < T_{TR}$) during image synthesis

Table 3 Dynamic properties of ASDS

Then the overall time required for ray tracing algorithm itself can be expressed (terms are described in Tables 1-3) in a simplified way as follows:

$$\begin{aligned} T_{TR} &\doteq (T_T + T_P).N_{NODES} + T_I.(N_{HPR} + N_{HSECR}) + \\ &+T_{IUN}.(N_{PRIT} + N_{SECRIT} + N_{SRIT} - N_{HPR} - N_{HSECR} - N_{HSR}) + const. \end{aligned} \tag{4}$$

where T_I is the time for performing one ray-object succeeded intersection test
T_{IUN} is the time for performing one ray-object failed intersection test
T_T is the time for performing one traversal step
T_P is the time for a decision step in parent node

Total time T_{TR} can be minimized by all R parameters in this equation including the volume of non-empty cells and by other C parameters, e.g., by the average number of objects in a cell.

4.1.1 Comparison methodology for ASDS

The comparison between different ASDSs is usually performed only by times for rendering. There are in general only two methods, a new one and the old one, which are mutually compared.

It is impossible to compare the results obtained by different researches in different papers due to the different dependencies (hardware, compiler, implementation etc.).

The intent of this subsection is to give a new method for comparison of ASDS techniques for visibility and ray-casting computation. The important assumption for comparison of some ASDS methods for construction or implementation is that the image synthesis is performed under the same conditions. In other words, the parameters denoted by B and thus also by D in Tables 1-3 have to be equal, i.e., the result of both rendering processes has to be the same image. The difference of some parameters indicates some errors in the algorithm implementation. The time (T_{TR}) and the space (N_C) requirements are the most important aspects of practical applicability of rendering software using ASDSs.

The first possibility is to compare some ASDS properties for a given scene by C and R parameters so the comparison is independent of the architecture, implementation, and compiler used. The set of parameters shown in the tables determines well the properties of the ASDSs. Unfortunately, their number is rather high to be dealt with, and therefore we have tried to restrict them reasonably. The restriction should be done so that each selected parameter expresses some specific meaning. Moreover, selected parameters should express different and important properties of ASDS for image synthesis.

We have divided the parameters expressing properties of ASDS for image synthesis into two n-tuples. The first n-tuple concerns the properties of accelerating method and is independent of the implementation. We propose to use the following septet Δ of the parameters above for this type of comparison:

$$\Delta =< N_C, R_{ETNC}, N_{ADC}, N_{AOIFC}, R_{FVWV}, N_{RPRT}, N_{TAC} > \tag{5}$$

The second type of comparison concerns the implementation of ray-tracing source code. In this case the parameters denoted by B, D, C, and R remain unchanged, parameters denoted by T reflect the quality of implementation, or/and the optimization efficiency of a compiler, or/and the performance of the architecture used. For the overall evaluation of ray-tracer performance by triplet Λ other parameters from S, D, C, and R should also be taken into consideration.

$$\Lambda =< T_{CB}, T_{TR}, \frac{T_{TT}}{T_{TR}} >, \tag{6}$$

The practical use of septet Δ and triplet Λ is demonstrated in the following section.

4.2 Position of optimal splitting plane for BSP tree

As we described in Section 3.5.2, the minimal value of the cost function (3) depends on the position of a splitting plane. MacDonald and Booth [37] estimated the optimal position to be between spatial and object median, but it is truth only upon the condition that no objects are intersected by a splitting plane.

To minimize the cost function (3) correctly, full range of the volume should be searched for the optimal plane. Although the range is continuous, certain discrete points can be used to simplify computations. Let us consider only one possible orientation of the splitting plane, for instance perpendicular to the x axis as shown in Fig. 9. Bounding volumes can also be used instead of real objects. Figure 9 shows an example of a scene with four objects and corresponding graph of the cost function for unit size of the whole scene. The formula (3) has been simplified to $C = (S_L/S)n_L + (S_R/S)n_R$. Terms T_j and T_k have been set to one, because all intersection tests are supposed to be of the same time complexity. Terms T_P and T_T have

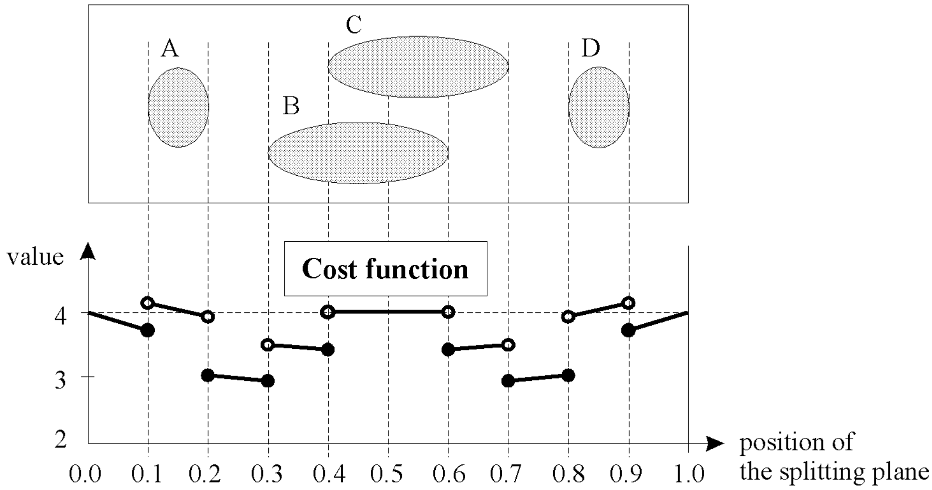

Figure 9: Key positions for selecting the optimal splitting plane

been set to zero, since their values do not change the shape of the graph, i.e., they do not influence extremes of the cost function.

It is obvious that the cost function can be linearly interpolated between two adjacent key positions as shown in Fig. 9. The number of objects between two adjacent key positions remains constant and cost function depends only on the projected surface area. The cost function is discontinuous and linear piecewise. Minimal value of the cost function can be found just at key positions, i.e., using limited number of sample splitting planes.

The set of key positions within the x range is derived from min–max x coordinates of all objects (their bounding volumes). In the sample scene in Fig. 9, four objects determine eight key positions, two of them are optimal (for $x = 0.3$ and $x = 0.7$).

The example also shows that the optimal position does not have to be always inside the interval specified by spatial and object median. Here the value of spatial median is 0.5. The value of object median would be somewhere between 0.4 and 0.6, because in that range the plane subdivides objects into two groups with the same cardinality. In spite of both optimal positions are outside the interval.

Table 4 shows the statistics how often the optimal plane has been found inside and outside the median interval for all six test scenes (for scenes see Fig 7 and 8).

Scene	Number of splitting planes	Planes outside median interval [%]
balls	25514	46.3
rings	88995	70.0
tetra	7270	7.8
fluid	15978	50.5
m–fluid	16030	48.1
room	16060	19.5

Table 4 Positions of optimal splitting planes

The number of splitting planes outside the median interval is surprisingly high. It sometimes represents more than 50% of all cases.

Implementation of the concept of key positions for setting the optimal splitting plane is not too difficult. Since planes are perpendicular to main coordinate axes, min–max values of bounding volumes for all objects are sorted into three lists, for each axis separately. A sorting is usually computed with $O(n \log n)$ complexity, but in this special case it is more convenient to use radix sort algorithm with O(n) complexity only. Those three lists are prepared during preprocessing phase and they are sorted only once. Whenever a node is split, new lists for child nodes are created by selection of objects from the current node. This operation preserves the order of key positions.

4.2.1 Orientation of the splitting plane

Whilst an octree structure can be built by evaluating optimal splitting planes in all three directions (Whang et al. [70]), the only one plane from three candidates has to be selected for the BSP tree. Two basic approaches can be recognized. First, the orientation of the plane is changed in cyclic order on the path from the root of a BSP tree to the leaves. Second, minimal value of the cost function taken from all three main directions always determines the splitting plane orientation.

We have implemented both approaches and measured Λ and Δ for them. In all cases the second method (arbitrary orientation) gives better results.

4.2.2 Cutting off empty space

The BSP tree building process is usually terminated when either the predefined depth of a tree is reached or the number of objects inside a subspace decreases under certain limit, typically 1–4 objects. It does not seem to be interesting to split a leaf node containing one object only. Still this situation should also be investigated to ensure that the current cost function for a certain tree is really minimal. We call this approach **cutting off empty space**.

Let A be a volume representing the whole scene consisting of one object only. Cost function (2) is then expressed as $C_A = \sum_{i=1}^{1} T_i = T_1$. Let us suppose the volume is split by a plane in such a way that the object B stays in the right subspace, whereas the left subspace is empty. Then using formula (3) we get the following new cost:

$$C_{new} = T_P + \frac{S_L}{S}(0 + T_T) + \frac{S_R}{S}(\sum_{k=1}^{1} T_k + T_T) = T_P + \frac{S_L + S_R}{S} T_T + \frac{S_R}{S} T_1 \tag{7}$$

The term $(S_L + S_R)/S$ can be further evaluated and the final formula is as follows:

$$C_{new} = T_P + \frac{x_A y_A + x_A z_A + 2 y_A z_A}{x_A y_A + x_A z_A + y_A z_A} T_T + \frac{S_R}{S} T_1 = T_P + \text{Const.} T_T + \frac{S_R}{S} T_1 \tag{8}$$

The position of the splitting plane influences only the last term in the cost function (8). Since constant coefficient $(S_L + S_R)/S$ is always bigger than 1, cost C_{new} could be also higher than the original cost C_A. Minimum of the function (8) is thus sensitive to computation times T_P, T_T, and T_1.

Real values of those computing times depend on the implementation of traversal algorithm. One efficient implementation has been published in Graphics Gems III by Sung and Shirley [60]. A BSP tree is traversed recursively from the root and a stack is used for storing nodes that should be visited on a path of a ray. The selection of nodes on the path (equal to term T_P) needs much more computing time than simple pop operation (equal to term T_T) performing a traversal step from a node to another one. Both values T_P and T_T can be precomputed for given implementation, and they stay constant for the whole computing process.

The time T_1 needed for one intersection test depends on the object geometry. Simple geometrical objects like spheres and triangles can be tested in time comparable with time T_P. Complex objects like NURBS require more computing time [68].

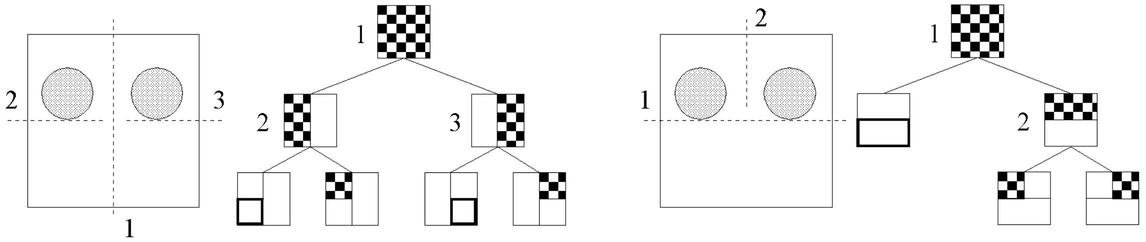

Figure 10: A volume with empty space and a corresponding BSP tree. Late cutting (left) is less effective than early cutting (right).

Our test scenes consist of simple geometrical objects only. In this case, the cutting off empty space does not improve the efficiency considerably. Comparing times T_P and T_1 for our implementation, cutting off empty space is meaningful when the ratio of the object surface area to volume surface area is smaller than 25% for spheres and 40% for triangles.

Empty space can be cut off not only from leaves, but also sooner, before splitting larger volumes. Figure 10 shows that the second approach saves memory space and decreases computational cost.

properties	test scenes								
	balls			*rings*			*tetra*		
	1	2	$\frac{2}{1}$[%]	1	2	$\frac{2}{1}$[%]	1	2	$\frac{2}{1}$[%]
N_C	3244	19892	613	153456	78677	51	96056	11322	11.8
R_{ETNC}[%]	32.3	25.4	−6.9	15.9	25.5	+9.7	28.6	25.9	−2.7
N_{ADC}[%]	130.7	284.7	+154.0	4505	1337	−3168	7400	1052	−6348
N_{AOIFC}	7.75	1.91	24.6	16.48	11.33	68.8	4.48	5.62	12.5
R_{FVWV}[%]	43.3	28.8	−14.5	23.4	8.10	−15.3	6.54	3.12	−3.42
N_{RPRT}	34.8	22.3	63.3	117.8	86.1	73.1	21.8	13.33	61.1
N_{AT}	26.2	4.35	16.6	56.51	34.5	61.0	36.3	17.4	47.9
$T_{CB}[sec]$	0.86	2.46	286	44.0	24.7	56.0	5.31	1.17	22.0
$T_{TR}[sec]$	128.7	43.8	34.0	612.0	340.6	55.7	21.0	8.31	39.6
T_{TT}/T_{TR}[%]	51.6	69.9	+18.3	17.8	16.4	−1.4	49.5	60.5	+11.0

Table 5 The n-tuple Δ and Λ for SPD scenes

4.2.3 Experimental results and discussion

Our improvements that decreases the time complexity of rendering using a BSP tree are shown in Tables 5 and 6. The first method (1) is the algorithm [60] with splitting plane orientation changing in cyclic order; its position always lies in the mid-point. In addition to the surface area heuristics, the second method (2) uses the cutting off the empty spaces in both on the outside and the inside of currently processed node introduced in subsection 4.2.2. Three columns are reported in the Tables 5 and 6 for each scene. The parameters Δ and Λ for method (1) are in the first column, the parameters for method (2) are in the second one. The third column describes their mutual position. In case the parameters Δ and Λ express absolute values (N_C, N_{AOIFC}, N_{RPRT}, N_{AT}, T_{CB}, T_{TR}), the third column expresses the ratio of (2) to (1). In case of relative parameters (R_{ETNC}, N_{AC}, R_{FVWV}), the value in the third column is computed as

properties	test scenes								
	fluid			*m-fluid*			*room*		
	1	2	$\frac{2}{1}[\%]$	1	2	$\frac{2}{1}[\%]$	1	2	$\frac{2}{1}[\%]$
N_C	28619	15912	55.6	18565	16727	90.1	167834	27925	16.6
$R_{ETNC}[\%]$	11.3	27.7	+16.4	16.4	28.0	+11.6	27.5	18.4	−9.1
$N_{ADC}[\%]$	2709	1486	−1223	1631	1121	−510	13172	1483	−11689
N_{AOIFC}	2.78	3.47	124.8	3.23	2.94	91.0	8.49	5.41	63.7
$R_{FVWV}[\%]$	37.2	3.70	−33.5	42.8	1.21	−41.6	12.2	40.6	+28.4
N_{RPRT}	10.0	4.0	40.0	10.1	3.3	32.7	63.0	19.3	30.6
N_{AT}	21.1	16.1	76.9	24.3	16.7	68.7	64.7	32.3	49.9
$T_{CB}[sec]$	1.16	1.57	135.3	1.11	1.64	148.8	14.56	3.06	21.0
$T_{TR}[sec]$	20.6	14.1	68.1	63.8	36.6	57.4	277.0	71.9	26.0
$T_{TT}/T_{TR}[\%]$	75.1	82.3	+7.2	61.8	80.2	18.4	26.8	41.1	+14.3

Table 6 The n-tuple Δ and Λ for specific scenes

the difference (2) and (1).

We can also compare the results which have been achieved by better positioning of the splitting plane (see 4.2) instead of the median interval [37]. The gain has been from 1% (scene *balls*) to 12% (scene *room*).

The improvement achieved by the method with empty space cutting off (called *newf*) in leaves is not so significant as we expected. The gain has been from 1% (scene *room*) to 7% (scene *m-fluid*) to the method (2). The small gain is due to the low time complexity of intersection calculation of a ray with objects in the scene comparing with the time complexity of traversal step.

Preliminary results
Very recently we have implemented the method that performs cutting off empty space in both inner nodes and leaves of BSP tree. The decision whether to cut off the empty space in a leaf is used in the method described here in 4.2.2. For cutting off the empty space in inner nodes we use another decision step based on surface area heuristics. Its disadvantage is that it has to take into account the ratio of the real time complexity of traversal step and intersection calculation. The new method (called *newsf*) decreases the time complexity from 1% (scene *m-fluid*) to 14% (scene *rings*) compared with the method *newf*.

To conclude the comparisons, we have decreased the time complexity of rendering by our method *newsf* from 30% up to 87% of time complexity required for uniform BSP tree.

4.3 Cache sensitive representation of BSP tree

In this part of the report we propose a new representation of the BSP tree data structure in the memory. The proposed memory mapping is designed to improve the spatial locality of data represented by the BSP tree to decrease the traversal complexity. The septet Δ remains the same, the triplet Λ changes.

4.3.1 Motivation: memory hierarchy

The time complexity of a traversal algorithm is connected with the hardware where the algorithm is executed. For analysis we suppose *Harvard* architecture with separated caches for instructions and data. Let T_{MM} denote memory latency (time to read/write one word of data from main memory to processor/cache).

The larger the memory and the lower the access time, the higher the cost of the memory. Since the instruction latency of processors is smaller than T_{MM}, between the memory and the processor is placed cache: smaller memory with lower access time T_C. This solution is economical; it uses temporal and spatial locality for accessing the data. The data between the cache and the main memory are transferred by blocks corresponding to the *cache line size* S_{CL}.

In this part of the report concerning cache sensitive representation of BSP tree we denote the time consumed by operations in terms of cycles. Let T_W denote the time of the operation performed in a node during traversing. Typical values for today's superscalar processors are $T_{MM} = 55, T_C = 4, T_W = 5, S_{CL} = 128 Bytes$ for MIPS R8000 (taken from [51]). Note that $T_W \ll T_{MM}$.

4.3.2 Standard methods of representation for binary trees

Binary trees can be either static or dynamic. A *static* tree once constructed remains unchanged during its use until its destruction. A *dynamic* tree enables to perform operations with nodes, e.g., to insert or delete a node.

Definition 16 We call a binary tree *complete* if all leaves are positioned in the same depth d from the root node and the number of leaves is 2^d, *incomplete* tree is the tree, which is not complete.

Definition 17 We call a *subtree* F of binary tree T arbitrarily connected subgraph of T.

Definition 18 We call a *rooted subtree* RF of a binary tree T such a subtree of T that contains the root of T.

Definition 19 Let h_C define *complete height* of a binary tree T as the depth of the maximal complete rooted subtree of binary tree T.

In this report we deal with static binary trees only, the BSP tree is its typical example. Let us review for the binary tree representations.

Random representation

A usual way to store the arbitrary binary tree is to represent each node as a special variable. The disadvantage of the method is the additional memory consumed by pointers, which can be, e.g., four times greater than the actual information stored in the node. The only advantage is that it is simple to implement. The situation is depicted in Fig. 11 (a). The addresses of the nodes in the memory have no connection with the position in the tree. It corresponds to the pseudo-code given in [60] by Sung.

Depth-first-search (DFS) representation

The nodes are put in the memory in the DFS order they are constructed (see Fig. 11 (b)). To alleviate the problem of the memory size consumed by pointers required for each allocated variable, large block of memory can be allocated and the nodes are then allocated subsequently from such a memory block. The size of the allocated block is expressed as $S_O = (2+2.N).S_P + S_I.N$, where N is the number of nodes to be stored in the block. For large N nearly up to $2.N.S_P$ of memory taken by pointers are saved in comparison with random representation.

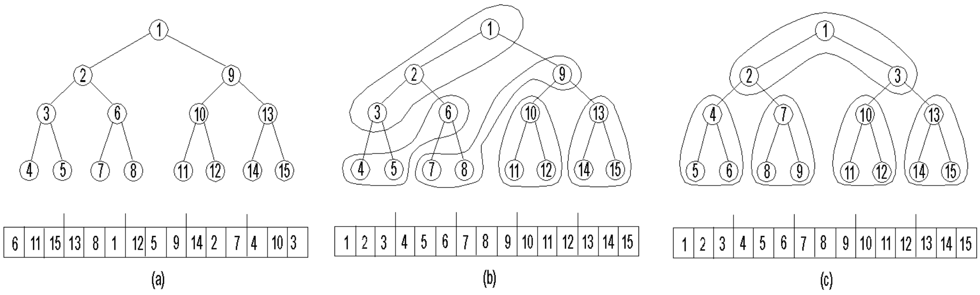

Figure 11: Binary tree representations ($S_{CL} = 3.size(node)$) (a) Random (b) DFS (c) Subtree

4.3.3 Subtree representation

We propose the following data structure in order to reduce the traversal time of the binary tree in DFS order. We suppose that we allocate one big block of memory and then we occupy it by the nodes - organized into smaller subtrees with the size smaller or equal to S_{CL}; see Fig. 11 (c). Once the subtree is read to the cache, the access time to some child nodes is equal to T_C. The subtree needs not be complete.

There are two ways for representing a subtree (Fig. 12). *Ordinary* subtrees have all the nodes of the same size, with two pointers to two descendants, regardless whether the descendant lies in the subtree or not. *Compact* subtrees have no pointers among the nodes inside the subtree because their addressing can be provided explicitly by a traversal program. The leaves of incomplete binary subtrees have to be marked in a special data variable (one bit for each node).

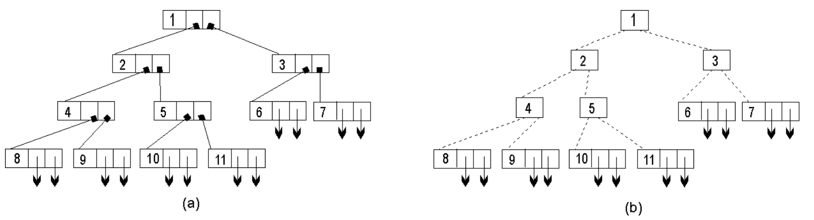

Figure 12: Subtree representation: (a) Ordinary (b) Compact

4.3.4 Time complexity and cache hit ratio analysis

In this subsection we are going to analyze the behaviour of the various representations during binary tree traversal. We assume the traversal is performed in depth-first-search order for simplicity. Another simplifying assumption is the data are in the main memory and none of them are located in the cache, i.e., cache hit ratio $C_{HR} = 0.0$. The analysis is provided assuming the binary tree is complete. The analysis is based on the height h of the complete binary tree, for an incomplete binary tree we can compute average the depth of a tree h_A and substitute it for h.

This analysis enables us to compute the average time T_A for performing a traversal on an binary tree of depth l from the root to a leaf. We suppose that in the each node the probability that we turn left is equal to $p_L = 0.5$.

If some data are already located in the cache ($C_{HR} > 0$), it is very difficult to analyze [2]. Since the cache has an asynchronous behaviour we analyze it by means of simulation.

Random representation

Since we suppose $C_{HR} = 0.0$ during the whole traversing, i.e., the access time to each node during traversing is T_{MM}, we can express T_A as follows:

$$T_A \doteq (T_{MM} + T_W).(l + 1) \tag{9}$$

For $T_{MM} = 53, T_W = 5, l = 23$ we obtain the time $T_A = 1392.0$ cycles.

DFS representation

This storage is influenced by reading the nodes for the next traversal step if we continue the traversal to the left descendant. Assuming the size of the node is S_{IN}, thus the number of nodes in one cache line is S_{CL}/S_{IN}, we can derive T_A as follows:

$$T_A \doteq (l + 1).(p_L.T_{MM}.S_{IN}/S_{CL} + T_C.(1 - S_{IN}/S_{CL}) + T_W + (1 - p_L).T_{MM}) \tag{10}$$

For $T_{MM} = 53, T_C = 4, T_W = 5, l = 23, S_{IN} = 12, S_{CL} = 128$, we get time $T_A = 859.1$.

Ordinary subtree representation

Assume that S_{CL} and S_{IN} are given. For each subtree, we are required to store S_{ST} bytes additionally, that are used as the identification of the type of the subtree. Let us express the memory part taken by a complete subtree with the height h:

$$M(h) = (2^{h+1} - 1).S_{IN} + S_T \leq S_{CL} \tag{11}$$

From (11) we can derive the complete height of the subtree h_C as follows:

$$h_C = \lfloor -1 + \log_2[(S_{CL} - S_{ST})/S_{IN} + 1] \rfloor \tag{12}$$

The subtree of height h_C is stored fully in one cache line. The rest of the cache line can be used to store N_{ODK} nodes in the depth $d = h_C + 1$ in the subtree:

$$N_{ODK} = \lfloor [S_{CL} - (2^{h_C+1} - 1).S_{IN} - S_{ST}]/S_{IN} \rfloor \tag{13}$$

The average height of the subtree to be used in the formula should be computed expressing the time complexity of binary tree traversal. The average height of the subtree $h_A \geq h_C$ is computed as follows:

$$h_A = -1 + \log_2(2^{h_C+1} + N_{ODK}) \tag{14}$$

Finally, the total time to traverse the tree for the depth l from root to arbitrary leaf is:

$$T_A = (l + 1).(T_W + T_{MM}.1/(h_A + 1) + T_C.D_A/(D_A + 1)) \tag{15}$$

For $T_{MM} = 53, T_C = 4, T_W = 5, l = 23, S_{IN} = 12, S_{ST} = 4, S_{CL} = 128$, we get $h_C = 2$, $N_{ODK} = 3$, $h_A = 2.46$, and $T_A = 555.9$ cycles.

Compact subtree representation

Let us denote the portion of the memory for representation of the information inside the node S_I, the memory occupied by one pointer S_P. The memory part taken by a complete subtree with the height h is then:

$$M(h) = (2^{d_k+1} - 1).S_I + 2^{d_k+1}.S_P + S_T \leq S_{CL} \quad (16)$$

Complete height of the subtree h_C is from (16) derived similarly to (12) as follows:

$$h_C = -1 + \lfloor((S_{CL} + S_I - S_ST)/(S_I + S_P))\rfloor \quad (17)$$

Similarly as for an ordinary subtree, we derive the number of nodes in the depth $d = h_C + 1$ as follows:

$$N_{ODK} = \lfloor(S_{CL} - 2^{h_C+1}.(S_I + S_P) + S_I - S_{ST})/(S_I + S_P)\rfloor \quad (18)$$

The h_A and t_A is computed by equations (14) and (15). For $T_{MM} = 53, T_C = 4, T_W = 5, l = 23, S_P = 4, S_I = 4, S_{ST} = 4, S_{CL} = 128$, we compute $h_C = 3$, $N_{ODK} = 0$, $h_A = 3.0$, and $T_A = 510.0$ cycles.

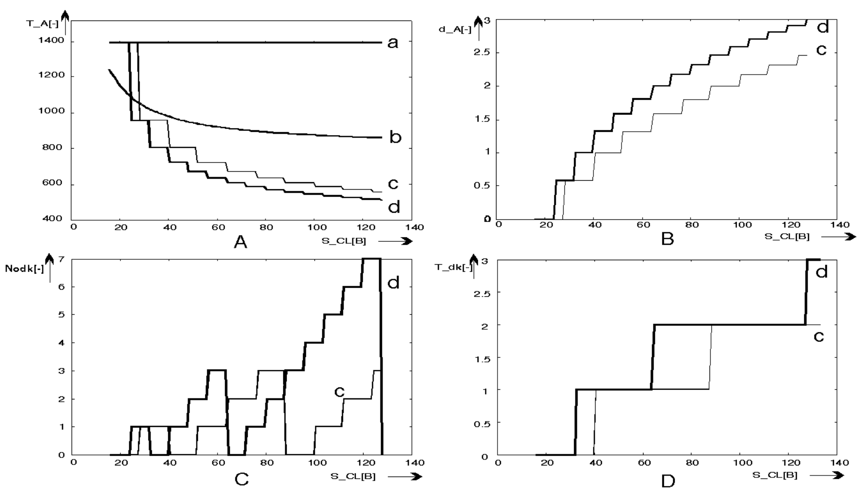

Figure 13: The analysis A: $T_A = f_1(S_{CL})$, B: $h_A = f_2(S_{CL})$, C: $N_{ODK} = f_3(S_{CL})$, D: $N_{dk} = f_4(S_{CL})$; Representation (a) Random (b) DFS, (c) Ordinal subtree, (d) Compact subtree

The functions h_C, N_{ODK}, h_A for ordinary and compact subtree and T_A for all types of storage in dependence on the cache line size are depicted in Fig 13.

4.3.5 Results of simulation

The simulation was done for the same times as in the subsection above: $T_{MM} = 53$, $T_C = 4$, $T_W = 5$, $l = 23$, $S_P = 4$ Bytes, $S_I = 4$ Bytes, $S_{ST} = 4$ Bytes. The simulation of traversing was performed in depth-first-search order, the same as for theoretical analysis. The times obtained by simulation correlate surprisingly well with the ones computed theoretically.

The cache was four–way set associative, cache line size $S_{CL} = 2^7$ Bytes, the size of the cache was 2^{20} Bytes (1 MB). It corresponds to the number of cache lines 2^{13} (8192). The cache organization corresponds to that found in current superscalar processors, e.g., MIPS R8000 or MIPS R10000 (see [51]). In Table 7 we can see the theoretical, simulated times, and their ratio. The C_{HR} reflects average hit ratio for any node during the traversal. The cache hit ratio for the node as the function of its depth is in Table 8.

	Representation			
	Random	DFS	Ordinary subtree	Compact subtree
t_A (theoretical)	1392.0	859.1	555.9	510.0
t'_A (simulated)	987.1	629.4	445.6	379.3
$ratio = t_A/t'_A$	1.41	1.36	1.24	1.34
C_{HR}[%]	35.8	69.8	83.5	90.3

Table 7 The times computed theoretically and obtained by the simulation

	Depth											
	0	1	2	3	4	5	6	7	8	9	10	11
C_{HR} (Random)	100	100	100	100	97	91	62	52	39	25	21	18
C_{HR} (DFS)	100	100	100	100	100	93	79	84	58	56	63	51
C_{HR} (Ordinary subtree)	100	100	100	100	100	100	97	73	90	85	53	79
C_{HR} (Compact subtree)	100	100	100	100	100	100	100	100	69	100	100	100
	Depth											
	12	13	14	15	16	17	18	19	20	21	22	23
C_{HR} (Random)	21	19	19	0	0	0	0	0	0	0	0	0
C_{HR} (DFS)	57	59	47	59	54	48	51	49	47	54	43	54
C_{HR} (Ordinary subtree)	80	64	66	79	66	70	72	74	61	75	74	62
C_{HR} (Compact subtree)	7	100	100	100	1	100	100	100	0	100	100	100

Table 8 The cache hit ratio for the node as the function of its depth

Here we conclude the part of the report concerning the cache sensitive representation. The proposed methods of subtree representation for binary tree in main memory decrease the traversal time by 62% and increase the hit ratio from 30% to 90%. Moreover, the memory required for representation of the binary tree is decreased by 57%.

5 Future work

In this chapter of the report we show the shortcomings of currently used ASDSs. Further, we propose some new ideas and problems that should be solved and algorithmized to improve the efficiency of ASDSs.

The shortcomings of current ASDSs can be divided into two groups. The first group of ASDSs is noted for "reasonable" memory complexity, but rather high time complexity. It includes the adaptive, nonuniform data structures as *octree*, *BSP* tree, and *bounding-volume* hierarchy. These ASDSs try to adapt the local distribution of objects in the scene, but the result is mostly a too deep hierarchy that is costly to traverse. The second group includes the regular ASDSs. They are actually represented by SEADS (grid) and its modifications (see [14] and [49]). In the opposite to the original algorithm, the hierarchical and recursive modifications are nonuniform. The time complexity of regular ASDSs is lower, but the memory requirements are $O(n^3)$ and the time complexity of preprocessing phase is also high especially for recursive/hierarchical modifications.

Let us outline some ideas and problems that should be investigated during our future research.

Open problem 1 *Let $\Phi(C^n)$ define a set of rectangular parallepipeds representing the objects bounding volumes for initial cell C^n. The parallepipeds of $\Phi(C^n)$ can overlap arbitrarily. Find* *set of parallepipeds Ψ with the properties:*

a) cells of $\Phi(C^n)$ and $\Psi(C^n)$ create the space subdivision SSD, i.e., arbitrary two cells of $\Phi(C^n)$ do not overlap mutually and arbitrary cell of $\Phi(C^n)$ and arbitrary cell of $\Psi(C^n)$ do not overlap as well.

b) the sum[*] of surface areas of $\Psi(C^n)$ is minimal:

$$\begin{aligned} SSD_\Psi(C^n)opt = \quad & \{SSD_\Psi(C^n)opt \in \Omega^*(SSD_\Psi(C^n)) : f(SSD_\Psi(C^n)opt) \leq f(SSD_\Psi), \\ & \forall SSD_\Psi(C^n)opt \in SSD^*_\Psi(C^n)\}, \end{aligned}$$

where

$$f(SSD_\Psi(C^n)) = \sum_{i=0}^{i=k(SSD_\Psi(C^n))} surface\_of(cell)_i$$

Open problem 2 *Find the neighbouring mapping of cells of Φ and Ψ for SSD obtained by solution of Open problem 1.*

Conjecture 1 Open problem 1 is NP-complete.

Task 1 *Find an arbitrary suboptimal algorithm for Open problem 1.*

Open problem 3 *Find some set Γ of non-overlapping rectangular parallepipeds for set of $\Phi(C^n)$ of bounding volumes. The surface area of each parallepiped Γ is greater than $p.S_{SCENE}$, where S_{SCENE} is surface area of bounding volume of scene and $p \in (0, 1>$.*

Task 2 *Find the algorithm for building a BSP tree that uses set Γ of solved Open problem 3 for given set Φ. Built BSP tree covers the surfaces of all parallepipeds in Γ.*

Open problem 4 *Find set of overlapping rectangular parallepipeds $\Theta(C^n)$ for given set of bounding volumes of objects $\Phi(C^n)$. It means that arbitrary parallepipeds of $\Theta(C^n)$ and Φ do not overlap and two arbitrary parallepipeds of Φ can overlap. Each point of bounding volume of scene is covered at least by one parallepiped Φ or Θ.*

Open problem 5 *Find neighbouring information for Open problem 4.*

Conjecture 2 Open problem 4 is NP-complete.

Task 3 *Find an arbitrary suboptimal algorithm for Open problem 4.*

Open problem 6 *Find a* time estimation *algorithm for ASDSs and given time based on the distribution and geometrical properties of objects in the scene. The algorithm estimates the cost of rendering of given scene for set of ASDSs with given reliability.*

Open problem 7 *Find such set S_C of ASDSs [†] that every ASDS in S_C works for particular group of object primitives. These groups are distinguished by the different geometrical properties of objects that the groups contain. Moreover, find the efficient traversal algorithm for this set.*

In the current state of our research, we can define more problems and ideas that should be solved and verified by implementation. The problem of ASDSs is very complex and very challenging, and no optimal solution is known for arbitrary objects distribution.

[*] This requirement utilizes the surface area heuristics.

[†] This schemes minimizes the duplication of objects in elementary cells.

6 Conclusions

In the first part of the report we have reviewed well-known ASDSs. In the second part we outlined the problems that we have addressed in the past: a generalized method for experimental evaluation of ASDSs, improvements of cost function for a BSP tree, and the cache-sensitive representation for a BSP tree. Recently, we have been working with ideas concerning mostly improvements of a BSP tree for ray-casting and visibility computations. Currently, we are working with ideas towards more general and specific ASDSs. In the preceding chapter we gave some challenging problems concerning ASDSs. We are going to solve our problems, implement, and evaluate them both theoretically and experimentally.

Acknowledgements

I appreciate very much a help of my colleagues who helped me during the research that is presented here. For formal issues and appearance of this report, I would like to thank Bořivoj Melichar, Pavel Slavík, and Patricia Hymson for their comments and proofreading of the previous versions of the report.

7 References

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

[2] Arnold, O.A.:*Probability, statistics, and queuing theory with computer science applications.* Second editions, Academic Press, Inc., San Diego, 1990.

[3] James Arvo and David Kirk. Fast ray tracing by ray classification. volume 21, pages 55–64, July 1987.

[4] Kadi Bouatouch, M. O. Madani, Thierry Priol, and Bruno Arnaldi. A new algorithm of space tracing using a CSG model. In G. Marechal, editor, *Eurographics '87*, pages 65–78. North-Holland, August 1987.

[5] Wiley C, Campbell A, Szygenda S, Fussel D, and Hudson F. Multiresolution bsp trees. May, 1997, Proceedings of Graphics Interface'97 University of Texas.

[6] A. T. Campbell, III and Donald S. Fussell. Adaptive mesh generation for global diffuse illumination. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 155–164, August 1990.

[7] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: A new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):C/371–382, 1995.

[8] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Graphics Interface*, pages 132–141, May 1996.

[9] Mark J. Charney and Isaac D. Scherson. Efficient traversal of well-behaved hierarchical trees of extents for ray-tracing complex scenes. *The Visual Computer*, 6(3):167–178, June 1990.

[10] Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 99–106, July 1989.

[11] Y. Chrysanthou and M. Slater. Computing dynamic changes to BSP trees. volume 11, pages 321–332, September 1992.

[12] Jung-Hong Chuang and Weun-Jier Hwang. A new space subdivision for ray tracing CSG solids. *IEEE Computer Graphics and Applications*, 15(6):56–62, November 1995.

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

[14] D.Cohen and Z.Sheffer. Proximity clouds - an acceleration technique for 3d grid traversal. *The Visual Computer*, 11:27–38, 1994.

[15] David Elliott Dauenhauer and Sudhanshu Kumar Semwal. Approximate ray tracing. pages 75–82, May 1990.

[16] Olivier Devillers. The macro-regions: an efficient space subdivision structure for ray tracing. In W. Hansmann, F. R. A. Hopgood, and W. Strasser, editors, *Eurographics '89*, pages 27–38. Elsevier / North-Holland, September 1989.

[17] R. Endl and M. Sommer. Classification of ray-generators in uniform subdivisions and octrees for ray tracing. *Computer Graphics Forum*, 13(1):3–19, March 1994.

[18] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

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

[20] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

[21] Glassner, A.S.:*An Introduction to Ray Tracing*. Academic Press, London 1991.

[22] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[23] Dan Gordon and Shuhong Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, September 1991.

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

[25] Eric A. Haines and John R. Wallace. Shaft culling for efficient ray-traced radiosity. In P. Brunet and F. W. Jansen, editors, *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, New York, NY, 1994. Springer-Verlag. also available via FTP from princeton.edu:/pub/Graphics/Papers.

[26] Ping-Kang Hsiung and Robert H. Thibadeau. Accelerating ARTS. *The Visual Computer*, 8(3):181–190, March 1992. nested grid subdivision structures.

[27] Veysi Isler, Cevdet Aykanat, and Bulent Ozguc. An efficient parallel spatial subdivision algorithm for parallel ray tracing complex scenes. In *First Bilkent Computer Graphics Conference, ATARV-93*, Ankara, Turkey, July 1993.

[28] Sig Badt Jr. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, September 1988.

[29] Michael R. Kaplan. The use of spatial coherence in ray tracing. In David E. Rogers and Ray A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 173–193. Springer Verlag, 1987.

[30] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. volume 20, pages 269–278, August 1986.

[31] Kryzsztof S. Klimansezewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January/February 1997.

[32] Arjan J. F. Kok and Frederik Jansen. Source selection for the direct lighting computation in global illumination. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, pages 75–82, New York, 1994. Springer-Verlag.

[33] Arjan J. F. Kok and Frederik W. Jansen. Adaptive sampling of area light sources in ray tracing including diffuse inter-reflection. *Computer Graphics Forum (Eurographics '92)*, 11(3):289–298, September 1992.

[34] Arjan J. F. Kok, Frederik W. Jansen, and C. Woodward. Efficient, complete radiosity ray tracing using a shadow-coherence method. *The Visual Computer*, 10:19–33, oct 1993.

[35] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, November 1992.

[36] Nathan Loofbourrow. Optimizing ray tracing with visual coherence. Technical Report CMU-CS-93-209, Carnegie-Mellon University, Department of Computer Science, 1993.

[37] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. In *Proceedings of Graphics Interface '89*, pages 152–63, Toronto, Ontario, June 1989. Canadian Information Processing Society. criteria for building octree (actually BSP) efficiency structures.

[38] P. Morer, A. M. Garcia-Alonso, and J. Flaquer. Optimization of a priority list algorithm for 3-D rendering of buildings. *Computer Graphics Forum*, 14(4):217–227, October 1995.

[39] Isabel Navazo, Josep Fontdecaba, and Pere Brunet. Extended octtrees, between CSG trees and boundary representations. In G. Marechal, editor, *Eurographics '87*, pages 239–247. North-Holland, August 1987.

[40] B. Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer–Aided Design*, pages 250–252, 1990.

[41] Bruce Naylor. Constructing good partition trees. In *Proceedings of Graphics Interface '93*, pages 181–191, Toronto, Ontario, Canada, May 1993. Canadian Information Processing Society.

[42] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, August 1990.

[43] Bruce F. Naylor. Interactive solid geometry via partitioning trees. pages 11–18, May 1992.

[44] K. Nechvile and J. Sochor. Form-factor evaluation with regional BSP trees. In *Winter School of Computer Graphics 1996*, February 1996. held at University of West Bohemia, Plzen, Czech Republic, 12-16 February 1996.

[45] Qunsheng Peng, Yining Zhu, and Youdong Liang. A fast ray tracing algorithm using space indexing techniques. In G. Marechal, editor, *Eurographics '87*, pages 11–23. North-Holland, August 1987.

[46] B. S. S. Pradhan and A. Mukhopadhyay. Adaptive cell division for ray tracing. *Computers and Graphics*, 15(4):549–552, 1991.

[47] Gunter Raidl and Wilhelm Barth. Fast adaptive previewing by ray tracing. In *Summer school in computer graphics in Bratislava (SCCG96)*, page (do not know), June 1996.

[48] Erik Reinhard, Arjan J. F. Kok, and Frederik 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.

[49] Semwal S and Kvarnstrom H. Directional safe zones & dual extent algorithms for efficient grid traversal. page to appear, 1997. University of Colorado.

[50] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics. part I: Fundamentals. *IEEE Computer Graphics and Applications*, 8(5):48–68, 1988.

[51] Sillicon Graphics:*Power Challenge*. Technical Report, 1996.

[52] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–60, 1989. includes code.

[53] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. *IEEE Computer Graphics and Applications*, 8(4):59–75, July 1988.

[54] Sauer, C.H., Chandy, K.M.:*Computer systems performance modelling*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[55] Isaac D. Scherson and Elisha Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, December 1987.

[56] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[57] G. Simiakakis and A. M. Day. Five-dimensional adaptive subdivision for ray tracing. *Computer Graphics Forum*, 13(2):133–140, June 1994.

[58] Stone, L.:*Theory of Optimal Search*. Academic Press, New York, 1975.

[59] K. R. Subramanian and Donald S. Fussell. Automatic termination criteria for ray tracing hierarchies. In *Proceedings of Graphics Interface '91*, pages 93–100, June 1991.

[60] Kelvin Sung and Peter Shirley. Ray tracing with the BSP tree. In David Kirk, editor, *Graphics Gems III*, pages 271–274. Academic Press, San Diego, 1992. includes code.

[61] Cassen T., Subramanian K.R., and Michalewicz Z. Near-optimal construction of partitioning trees by evolutionary techniques. In *Proceedings of Graphics Interface '95*, pages 263–271, Canada, June 1995.

[62] Jean-Philippe Thirion. TRIES: Data structures based on binary representation for ray tracing. In C. E. Vandoni and D. A. Duce, editors, *Eurographics '90*, pages 531–541. North-Holland, September 1990.

[63] Karl Kelvin Thompson. *Ray tracing with amalgams.* Ph.d. thesis, University of Texas at Austin, May 1991.

[64] Enric Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In C. E. Vandoni and D. A. Duce, editors, *Eurographics '90*, pages 507–518. North-Holland, September 1990.

[65] Maurice van der Zwaan, Erik Reinhard, and Frederik W. Jansen. Pyramid clipping for efficient ray traversal. In *Proceedings of the Sixth Eurographics Rendering Workshop*, Dublin, Ireland, 1995.

[66] Voorhies, D.:*Space-filling Curves and a Measure of Coherence.* Graphics Gems II, pp. 197-201, 1992.

[67] Gregory Ward. Adaptive shadow testing for ray tracing. In *Eurographics Workshop on Rendering*, 1991.

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

[69] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[70] K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Cho, C. M. Park, and I. Y. Song. Octree-R: an adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995. ISSN 1077-2626.

[71] T. Whitted. An improved illumination model for shaded display. *CACM*, pages 343–349, June 1980.

[72] G. Wyvill, T. L. Kunii, and Y. Shirai. Space division for ray tracing in CSG (constructive solid geometry). *IEEE Computer Graphics and Applications*, 6(4):28–34, April 1986.

[73] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–23, September 1992.

Dissertation thesis

Title: Adaptive data structures for image synthesis

Abstract

In the dissertation thesis we are going to address the problem of pair-of-points visibility computation and the ray-casting problem. We will propose new data structure(s) with better or even optimal time complexity and with reasonable memory complexity with respect to the number of objects in the scene. Our method will also take into account the local distribution of objects and their geometrical properties with regard to the visibility problem. We want to solve the problem of scene coverage by both sets of non-overlapping rectangular parallepipeds and overlapping rectangular parallepipeds. We want to attack the problem of estimation of time complexity of rendering for different spatial data structures. We also want to find such combinations of overlapping data structures that perform with smaller time complexity than the original ones. Furthermore, we are going to analyze the spatial data structures theoretically and verify the results of the analysis experimentally by implementation.

Keywords

computer graphics, rendering, spatial data structures, ray-casting, visibility.

A References

[1] Havran, V.: *Cache Sensitive Representation for the Binary Tree.* Submitted to SOFSEM'97 in May 97, International Conference on Computer Science, 8 pages, Czech Republic, held in Milovy in November 97.

[2] Havran, V.: *Evaluation of BSP trees for ray-tracing.* Proceedings of Poster'97, 1st International Student Conference on Electrical Engineering, held in Prague, 2 pages, May 1997.

[3] Havran, V., Žára, J.: *Further Improvements of Space Subdivision Techniques for Ray Tracing.* Rejected from Eurographics Workshop on Rendering'97, held in Saint'Etienne in France, July 1997.

[4] Havran, V.:*Evaluation of BSP properties for ray-tracing.* In Proceedings of 12th Spring School on Computer Graphics, 8 pages, Bratislava, Slovak Republic, held in June 1997.

[5] Havran, V.: *Parallel Ray-tracer on Shared Memory Multiprocessor Machines Connected via Internet.* In A.Strejc, editor, Proceedings of Workshop'97, CTU-FEE Prague, January 20-22, 1997, pp.321-322, Czech Republic.

[6] Havran, V., Šoch, M., Šimek, P., Tvrdík, P.: *Parallel Merge Sort on Shared Memory Multicomputer.* In Proceedings of Workshop97, CTU-FEE Prague, January 20-22, 1997, Czech Republic.

[7] Havran, V.: *Parallel Implementation of Ray Tracing on Shared Memory Architecture.* Poster Section at SOFSEM'96, 23.11.96-30.11.96, Milovy, Czech Republic.

[8] Havran, V., Žára, J.: *Some Practical Aspects of Ray Tracing on Shared Memory Machine.* Eurographics Workshop on Parallel Rendering in Bristol at Poster Section, 26-27 August 1996.

[9] Havran,V.: *Load Balancing Techniques for Ray-tracer on Shared Memory Machine.* Proceedings of Poster'96, CTU-FEE Prague, pages 74-74, May 1996.

[10] Havran, V.: *The simulation of Optical Effects.* Master Thesis in February 1996, CTU-FEE Prague, Department of Computer Science and Engineering.

[11] Havran, V., Žára, J.: *Effective Homogenous Transformation of Large Raster Images.* Poster at SOFSEM'95, 25.11.95 - 1.12.1995, Milovy, Czech Republic.

[12] Havran, V., Žára, J.: *The simulation of the Real Camera for Rendering.* Proceedings of Workshop95, CTU-FEE Prague, January 23-26, 1995, pp.183-184, Czech Republic.