CACHE SENSITIVE REPRESENTATION FOR BSP TREES

Vlastimil HAVRAN
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
12135 Prague
CZECH REPUBLIC
havran@fel.cvut.cz

ABSTRACT

A binary tree is a common data structure used in many branches of computer science. It is often used for solving various types of searching problems. In complexity analysis we usually abstract from the real implementation and derive easily the time complexity of traversal from the root of a balanced tree to any leaf in asymptotic time $O(\log(m))$, where m is the number of leaves. In this paper we propose a new method for memory mapping of a binary tree aiming to improve spatial locality of data represented by binary tree and to decrease traversal complexity.

Key Words: Ray-tracing, Binary tree, BSP tree, Cache, Spatial Locality.

MOTIVATION

The basic motivation for this research comes from binary trees in computer graphics applications. In this paper we will refrain from details of image synthesis concerning object modeling and from the classification of rendering algorithms. The significant part of image synthesis is devoted to repeatedly computing the ray-casting problem and visibility between two points to determine the global illumination of the resulting image. The spatial data structures are used in order to reduce the time complexity of ray-casting or the visibility problem. They are usually based on space subdivision schemes [Watt-Watt92].

One scheme for space subdivision is based on *a binary space partitioning tree* (*BSP* tree in the following text). It was initially developed as a means of solving the hidden surface problem in computer graphics [Fuchs et al. 80].

It is the analogue to the search binary tree, but it works for three-dimensional data. A BSP tree hierarchically subdivides a volume of n-dimensional space scene containing a collection of objects. The tree is formed by recursively subdividing a space volume in two sub-volumes, 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 subspaces determined by the partitioning. The leaf nodes of the tree are convex non-overlapping volumes.

The leaves of the tree are either occupied by objects or vacant. The construction of the

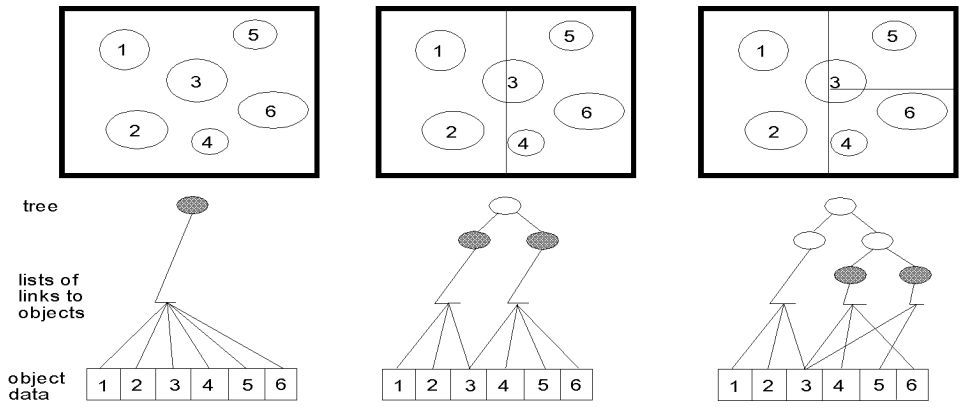

Figure 1: Partitioning of space by BSP tree

tree is done by recursively subdividing node volumes until the number of objects in nodes is smaller than a given constant or the depth of nodes in the tree is equal to a given constant. The partitioning process is depicted in Fig. 1.

An efficient algorithm for traversing a BSP tree exists for ray-tracing [Sung-Shirley92]. The rendering consists of traversing the BSP tree and performing the intersection tests. Our measurements have shown that the time of a ray-tracing algorithm devoted to BSP traversing varies from 50% to 80% [Havran-Žára97]. The total time for rendering a picture with resolution 512×512 for a scene with hundreds of thousands of objects can reach several hours. One way to decrease the time is to optimize the BSP tree construction using *surface area heuristics* [MacDonald-Booth90], which on average decreases the total time by up to 60%, whereas the time portion for BSP traversal even increases. In this paper, we propose a new technique to decrease the time of traversal by representing the BSP tree in a special way. It increases spatial locality and improves the processor cache hit ratio during the code execution. The technique is also applicable to binary trees used outside the area of computer graphics.

MEMORY REQUIREMENTS

It is obvious, that an arbitrary BSP tree cannot be efficiently represented by a heap structure. A common solution is to represent each node in a specially allocated variable. Let S_I denote the size of memory for the information in a node and S_P denote the size of a pointer. Then the size required to represent one node of a binary tree is $S_N = S_I + 2.S_P$.

Unfortunately, this is not true if we take into account the strategy of allocation libraries of current compilers. The data structures for a memory allocation manager are depicted in Fig. 2.

The memory is composed of allocated and free blocks. These blocks are connected by bidirectional links. Moreover, free blocks of size belonging to a specific range $< 2^k + 1, 2^{k+1} > (k \in N)$ are connected by other bidirectional links that are linked up with the table of constant size. These data structures enable us to allocate and deallocate the variables in $O(1)$ time.

The disadvantage of this allocation strategy is the additional memory consumed by every allocated variable and by memory fragmentation. The memory taken by a variable of given size S_V is $S_T = S_V + 2.S_P$. For one node of the binary tree ($S_V = S_N$) $S_T = S_I + 4.S_P$.

MEMORY HIERARCHY

The time complexity of the traversal algorithm is connected with the hardware where the algorithm is executed. For analysis we

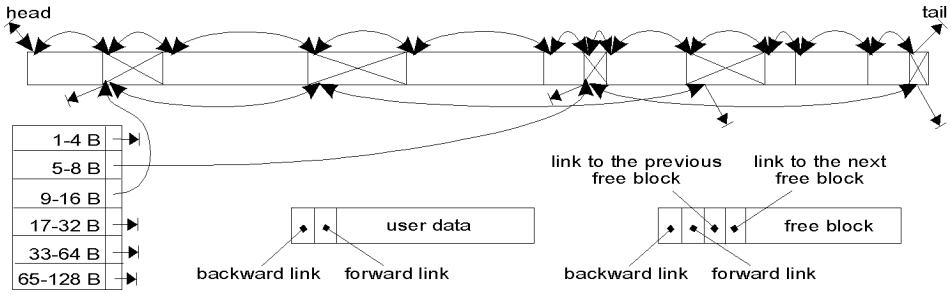

Figure 2: Allocation data structure

suppose *Harvard* architecture with separated caches for instructions and data. Let T_{MM} denote latency of main memory (time to read/write one word of data).

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 the 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 in blocks corresponding to *cache line size* S_{CL}.

In this article, 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 [SGI96]). Note that $T_W \ll T_{MM}$.

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 us to perform operations with nodes, e.g., to insert or delete a node. 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, an *incomplete* tree is the tree, which is not complete. The depth of the root node is zero.

Let h_C define *complete height* of a tree A as the maximal depth, for which the tree B constructed by the nodes of A with depth smaller than or equal to h_C is complete.

In this paper we deal only with static trees, the BSP tree is its typical example. Let us give the overview for the binary tree representation.

Random Representation

A common way to store the arbitrary binary tree in the main memory is to represent each node as a special variable. The disadvantage of the method is 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. 3 (a). The addresses of the nodes in the memory have no connection with their location in the tree. It corresponds to the pseudo code given in [Sung-Shirley92].

Depth-First-Search (DFS) Representation

We have not found any references to the DFS method in bibliographies, but it can be the result of using some allocation techniques for size-equal objects, see [Stroustrup91]. The nodes are put in the memory in the DFS order they are constructed (see Fig. 3 (b)). To alleviate the problem of the memory size consumed by pointers required for each allocated variable, firstly a 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 $S_O = (2+2.N_{NO}).S_P+S_I.N_{NO}$, where N_{NO} is

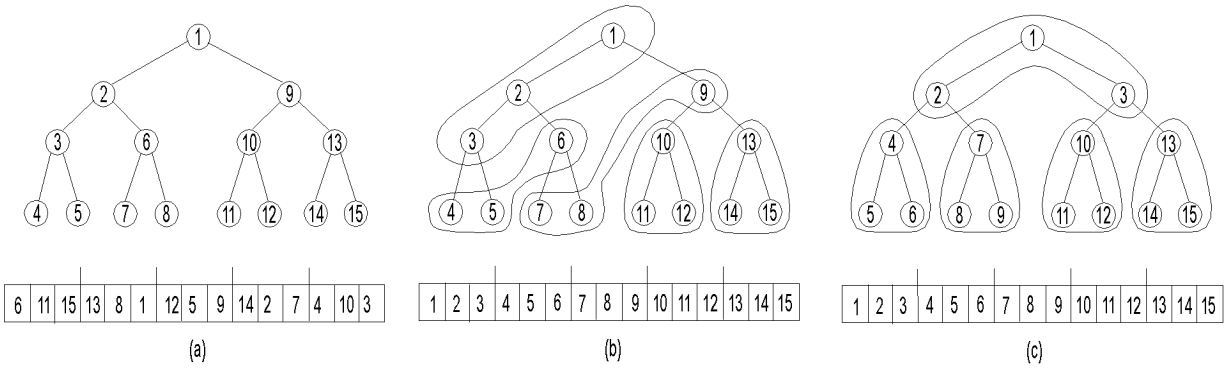

Figure 3: Binary tree representations ($S_{CL} = 3.size(node)$) (a) Random (b) DFS (c) Subtree

the number of nodes to be stored in the block. For big N_{NO} nearly up to $2.N_{NO}.S_P$ of memory taken by pointers are saved in comparison with random representation.

Subtree Representation

We propose the following data structure in order to reduce the complexity of DFS order traversal for binary tree. 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 than or equal to S_{CL}, see Fig. 3 (c). Once the subtree is read to the cache, the access time to some of its descendant nodes is equal to T_C. The subtree needs not be complete.

We distinguish between two ways for representing a subtree (Fig. 4). An *ordinary* subtree has all the nodes of the same size, with two pointers to two descendants, regardless of whether the descendant lies in the subtree or not. A *Compact subtree* has no pointers among the nodes inside the subtree because their addressing can be provided explicitly by a traversal program. The leaves of an incomplete binary subtree have to be marked in a special data variable (one bit for each node).

TIME COMPLEXITY AND CACHE HIT RATIO ANALYSIS

In this section we are going to analyze the behavior of all the mentioned representations of BSP trees. The analysis is performed under the assumption that the data are in the main memory and none is located in the cache, i.e., cache hit ratio $C_{HR} = 0.0$ and 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 depth of a tree h_A and substitute it for h.

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

If some data are already located in the cache ($C_{HR} > 0.0$), the analysis can be very difficult or even infeasible, the interested reader should follow [Arnold90]. Since the cache has asynchronous behavior, we analyzed the case by means of simulation.

Random Representation

As 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:

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

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

DFS Representation

The DFS representation increases the cache hit ratio by reading the nodes for the next traversal step if we continue the traversal to the left descendant. Assuming the size of the

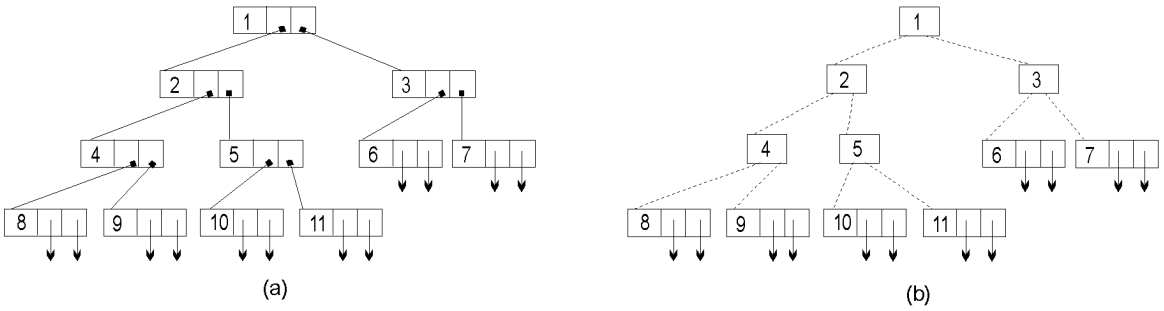

Figure 4: Subtree representation: (a) Ordinary (b) Compact

node is S_{IN}, we derive the average access time to a leaf T_A as follows:

$$T_A \doteq (l+1).(p_L.T_{MM}.S_{IN}/S_{CL} + T_W + T_C.(1 - S_{IN}/S_{CL}) + (1-p_L).T_{MM}) \quad (2)$$

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

Ordinary Subtree Representation

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

$$M(h) = (2^{h+1} - 1).S_{IN} + S_{ST} \leq S_{CL} \quad (3)$$

From Eq. 3 we can derive the complete height of the subtree h_C as follows:

$$h_C = \lfloor -1 + \log_2[(S_{CL} - S_{ST})/S_{IN} + 1] \rfloor \quad (4)$$

The number of nodes in the ordinary incomplete subtree with depth $d = h_C + 1$ is then:

$$N_{ODK} = \lfloor [S_{CL} - (2^{h_C+1} - 1).S_{IN} - S_{ST}]/S_{IN} \rfloor \quad (5)$$

The average height of the subtree $h_A \geq h_C$ for $N_{ODK} > 0$ is computed as follows:

$$h_A = -1 + \log_2(2^{h_C+1} + N_{ODK}) \quad (6)$$

Finally, the total traversal time of the whole tree with the depth l is:

$$T_A = (l+1).\ (T_W + T_{MM}/(h_A+1)\ +\ T_C.h_A/(h_A+1)) \quad (7)$$

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 S_I denote the portion of the memory for representation of the information inside the node, S_P the memory occupied by one pointer S_P. The size of memory taken by a complete subtree with the height h is expressed as follows:

$$M(h) = (2^{h+1} - 1).S_I + 2^{h+1}.S_P + S_{ST}$$
$$M(h) \leq S_{CL} \quad (8)$$

A complete height h_C of the compact subtree is from Eq. 8 derived similarly to Eq. 4 as follows:

$$h_C = -1 + \lfloor (S_{CL} + S_I - S_{ST})/(S_I + S_P) \rfloor \quad (9)$$

In the same way as for the ordinary subtree we derive the number of node N_{ODK} with the depth $d = h_C + 1$:

$$N_{ODK} = \lfloor (S_{CL} - 2^{h_C+1}.(S_I + S_P) + S_I - S_{ST})/(S_I + S_P) \rfloor \quad (10)$$

The average height of the subtree h_A and the total access time T_A is computed using Eq. 6 and Eq. 7. 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.

The functions h_C, N_{ODK}, h_A for the ordinary and compact subtree and T_A for all types of representation in the dependence on the cache line size are depicted in Fig 5.

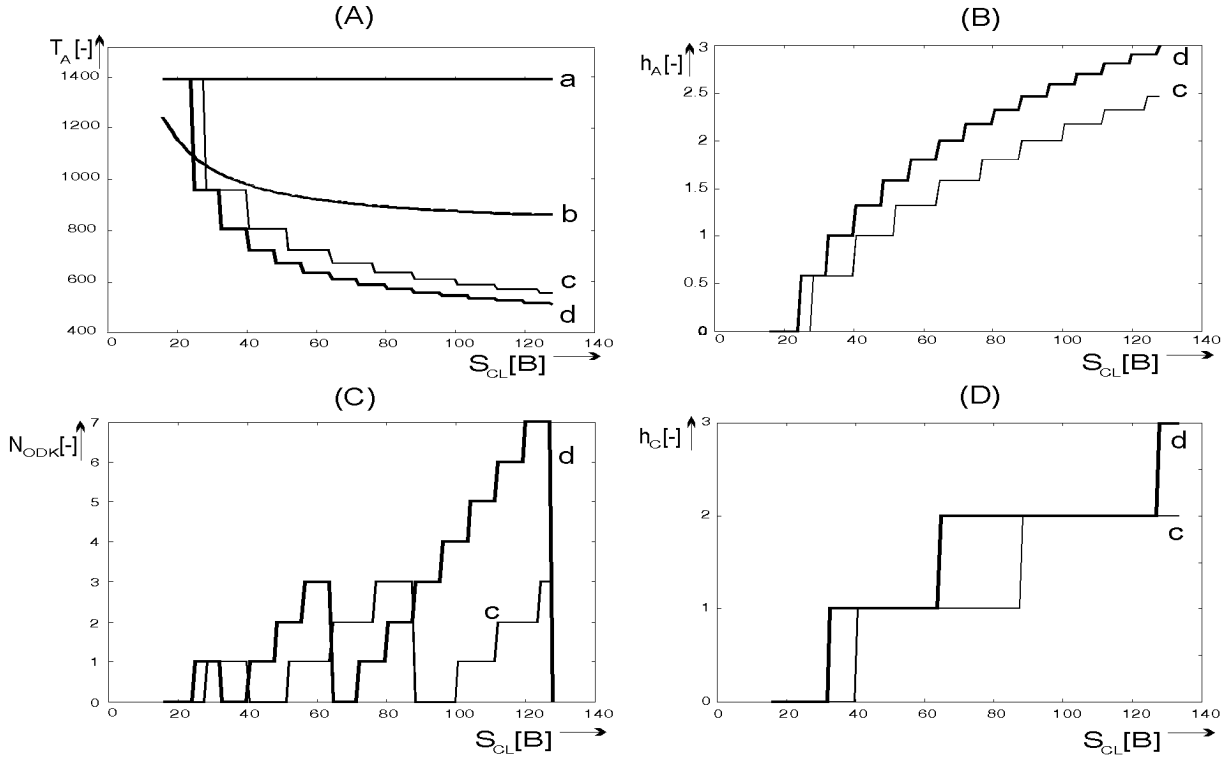

Figure 5: The analysis: (A) $T_A = f_1(S_{CL})$, (B) $h_A = f_2(S_{CL})$, (C) $N_{ODK} = f_3(CL)$, (D) $h_C = f_4(CL)$; Representation (a) Random (b) DFS, (c) Ordinary subtree, (d) Compact subtree

	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 1: The times computed theoretically and obtained by the simulation

RESULTS OF THE SIMULATION

We performed simulation in order to verify the validity of theoretical derivations. The simulation was realized as a special program simulating the memory hierarchy of a computer and the DFS traversal of a complete binary tree in its main memory. The algorithm for replacing the content of the cache was equivalent to the algorithm used in today's computer systems. The simulation was carried out for the same time values as in the previous section: $T_{MM} = 53$, $T_C = 4$, $T_W = 5$, $l = 23$, $S_P = 4$ Bytes, $S_I = 4$ Bytes, $S_{ST} = 4$ Bytes.

We chose the four–way set associative cache with cache line size $S_{CL} = 128$ Bytes, the size of the cache was 2^{20} Bytes. It corresponds to the number of cache lines 2^{13}. The cache organization corresponds to that found in current superscalar processors, e.g., MIPS R8000 or MIPS R10000 (see [SGI96]).

The theoretical, simulated times, and their ratio are summarized in Table 1. The parameter C_{HR} is the average cache hit ratio for any node during the traversal. The cache hit ratio for the node as the function of depth of the node is in Table 2. Note that for the compact subtree the values of C_{HR} for depth 12, 16, and 20 are quite different from other values, because the node must be always read from main memory into the cache.

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 2: The cache hit ratio for the node as the function of its depth

	Scenes					
	balls	gears	mount	rings	tetra	tree
#of objects	7382	9345	8196	8401	4096	8191
#of nodes	8469	20541	13369	23455	6011	5675
#of tr. steps [$\times 10^6$]	53.8	77.4	68.1	47.4	5.6	37.2
t_{total} (Random) [sec]	87.0	314.6	53.0	102.4	7.41	92.6
$t_{traversal}$ (Random) [sec]	42.1	59.8	22.9	21.5	4.09	18.6
$t_{traversal}$ (DFS) [sec]	35.6	51.8	17.4	14.6	3.76	14.6
$t_{traversal}$ (Ordinary subtree) [sec]	32.5	46.9	15.1	13.1	3.75	12.3
$\frac{t_{traversal}(Random)}{t_{traversal}(DFS)}$ [-]	1.18	1.15	1.31	1.47	1.09	1.27
$\frac{t_{traversal}(Random)}{t_{traversal}(Ordinarysubtree)}$ [-]	1.30	1.28	1.52	1.64	1.09	1.51

Table 3: The times obtained from ray-tracing on SPD scenes

The times obtained by simulation correlate well with the times computed theoretically, but are not equal, since the theoretical analysis supposes in each step an initial value of $C_{HR} = 0.0$. That is why the access times obtained by simulation are smaller than the theoretical ones.

MEASUREMENTS

We verified the contribution to the acceleration on real implementation of a ray-tracer for Random, DFS, and Ordinary Subtree representation recently. The algorithm for building-up of Compact Subtree representation is more complex and requires also a special algorithm for its traversal. We suppose on the basis of simulation, that it should not bring a significant improvement of performance in comparison with Ordinary Subtree representation. The tests were performed on scenes taken from *Standard Procedural Database* [Haines87] and the results of measurement are summarized in Table 3.

The contribution to the performance of the ray-tracer is not so significant as expected from simulation. It is because the simulation was performed the traversal down the tree with probability 50% to turn left in each node of the binary tree. It is true no more if we realize the subsequent primary rays generated in scanline order are likely to hit the same nodes of the BSP tree (rays coherence, see [Groeller 93]). The second reason for smaller improvement of performance is the rather smaller number of nodes of the BSP tree for given test scenes.

The properties of BSP tree representations

for such operations as range-search queries [Samet90] are guaranteed to stay in the range given by theoretical derivation and thus the simulation given in the previous section.

CONCLUSIONS

We improved time complexity of traversing of a binary tree by organizing its inner representation so that it matches the memory hierarchy. We showed and analyzed four methods for binary tree memory mapping - one traditional and three non-traditional. The proposed methods decrease the traversal time for traversal by 62% and increase hit ratio from 30% to 90%. Moreover, the memory required for storage is decreased by 57%. The properties of tree representation proposed by us cannot be for n-dimensional data ($n > 1$) replaced by one used in the context of external-memory data structures, for example B-tree [Cormen et al. 90]. It is because B-trees cannot represent n-dimensional data.

We plan in the future to use a cache-sensitive approach for other recursive data types and to analyze the methods for dynamic cache sensitive data structures.

ACKNOWLEDGMENTS

I would like to thank Jan Hlavička for delivering the subject of *Advanced Computer Architectures* and enabling me to write a report on this topic within this subject. Further, I thank Pavel Tvrdík and all the anonymous reviewers for their remarks on the previous version of this paper.

REFERENCES

Arnold, O.A. 1990 *Probability, statistics, and queuing theory with computer science applications*, Second edition, Academic Press, San Diego.

Cormen, T.H., Leiserson, C.H., Rivest, R.L. 1990 *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts.

Fuchs, H., Kedem, M.Z., Naylor, B. 1980 On Visible Surface Generation by A Priori Tree Structures, *Proceedings of SIGGRAPH'80*, Vol. 14, No. 3, July, pp. 124-133.

Groeller, E. 1993 *Coherence in Computer Graphics*, Dissertation Thesis, Technical University in Vienna.

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

Havran, V., Žára, J. 1997 Evaluation of BSP properties for ray-tracing, *Spring Conference on Computer Graphics 1997*, Slovakia, June 5-8, pp 155–162.

MacDonald, J.D., Booth, K.S. 1990 Heuristics for ray tracing using space subdivision, *The Visual Computer*, Vol. 6, Toronto, June, pp. 153–166.

Samet, H. 1990 *The Design and Analysis of Spatial Data Structures*, reprinted with corrections in 1994, Addison-Wesley.

Sillicon Graphics 1996 *Power Challenge*, Technical Report.

Sung, K., Shirley, P. 1992 Ray Tracing with the BSP Tree, *Graphics GEMS III*, in David Kirk editors, ACM PRESS, pp 271-274.

Stroustrup, B. 1991 *The C++ Programming Language, 2nd ed.*, Addison-Wesley, pp. 176–178.

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