About the Relation between Spatial Subdivisions and Object Hierarchies Used in Ray Tracing

Vlastimil Havran*
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

In this paper we study the relation between object hierarchies (such as bounding volume hierarchies) and spatial subdivisions (such as kd-trees and octrees) in the context of ray tracing for static scenes. First, we recall the principles used in efficient ray tracing algorithms and discuss the changes to the performance model more appropriate to current computer architectures. Second, we show how kd-trees can be emulated via bounding volume hierarchies. More importantly we show how bounding volume hierarchies can be emulated via kd-trees in six-dimensional space. Through emulation of one data structure via the second one we show that both data structures are computationally equivalent, assuming that their construction is carried out in top-down fashion.

CR Categories:

I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: ray shooting, ray tracing, hierarchical data structures, spatial sorting, performance model

1 Introduction

Ray tracing based image synthesis algorithms are becoming more and more popular thanks to several factors. First, the algorithmic progress allows us today ray tracing with expected logarithmic complexity in the number of objects. Even if there are no practical worst case complexity algorithms in use while they have been proposed [Szirmay-Kalos and Márton 1998], the algorithms aimed at average case complexity achieve remarkable performance [Wald 2004]. Second, a key behind the success of ray tracing algorithms is the performance increase of current CPUs compared to the end of seventies when the first ray tracing papers were published. For example, the performance gain can be estimated roughly by Moore's Law; for over last fifteen years a speedup factor is approximately one thousand. Third, the implementations of the algorithms are more efficient also thanks to the progress in compilers, computer languages, and better fitting of implemented algorithms to the computer hardware.

Unfortunately, there has been some confusion in the comprehension of concepts and performance of the ray tracing algorithms both among users and researchers. An efficient algorithm requires algorithmically efficient spatial data structures. Since there is a number of published algorithms and also data structures using different computer hardware, it is uneasy for non-expert to select an appropriate algorithm. In order to find some conclusion to this problem several experimental studies on the performance of ray tracing algorithms were conducted, however, often with contradictory conclusions. For example, Fujimoto et al. [Fujimoto et al. 1986] claims the superiority of uniform grids over octrees for a couple of example scenes. More recent studies [Endl and Sommer 1994; Havran et al. 2000; Szirmay-Kalos et al. 2002; Chang 2004] show the opposite for the skewed distribution of objects in the scene. There is a similar misunderstanding concerning the difference between the concept and performance of bounding volume hierarchies and spatial subdivisions such as kd-trees or octrees.

In this paper our concern is not to provide better or faster algorithm for ray tracing but to give better insight into the problem of spatial data structures used in this context. The paper is structured as follows. In Section 2 we recall the principles of efficient ray tracing algorithms. In Section 3 we discuss the properties of the hierarchical data structures in the dependence on arity of the hierarchy. In Section 4 we propose a novel performance model for ray tracing. In Section 5 we discuss the relation between spatial subdivisions and object hierarchies. In Section 6 we conclude the paper and propose possible future work.

2 Principles

In this section we describe several key principles behind the efficient ray tracing algorithms and how these relate to the used data structures. We focus on the comprehension of the basic ideas since most of them are known but often insufficiently understood in the context of ray tracing algorithms in computer graphics community.

2.1 Sorting and Searching

The key issue relevant to the complexity of studied algorithms is that *ray tracing is a searching problem* [Knuth 1998]. Given a set of primitives and a query, find out the result of this query. In our case, the set of primitives is a set of scene objects (such as triangles) and the query is a ray. The ray tracing belongs to the category of *geometric range searching* [Agarwal 2004]. In order to implement an efficient searching algorithm we first have to organize the input data by *sorting*. This paradigm is inherent to all searching problems including common life applications such as yellow pages that organize telephone numbers. If we do not preprocess the data (the preprocessing in our case equals to spatial sorting), the searching has to be implemented in a naive way with $O(N)$ time complexity for N objects.

*e-mail: havran@fel.cvut.cz

2.2 Hierarchical Organization: Divide and Conquer

One general way to sort objects is a hierarchical organization of the input data implementing powerful "Divide and Conquer" paradigm. The resulting data organization is represented by *hierarchical data structures* in form of a tree. The way of hierarchy creation is often referred to as *construction in a top-down fashion*. The hierarchical data structures are often distinguished according to the domain of organization. If we organize the data in a spatial domain, we call the resulting data structure a *spatial subdivision*. If we organize the data in an object domain, we call this data structure usually an *object hierarchy*. The link to the sorting is as follows: the creation of hierarchical data structures corresponds to quicksort [Knuth 1998, page 113] generalized to d-dimensional space. We pick up a pivot, either an object or position in the space, and sort all the objects either to left or to right. Then we recurse and continue the construction until the leaves are created.

There are other ways to create hierarchies often referred to as *construction in bottom-up fashion* or *insertion one-by-one*. The latter one has been used for bounding volume hierarchies (BVH) by Goldsmith and Salmon in [Goldsmith and Salmon 1987], but without comparison to top-down method. It has been shown experimentally by two independent studies that ray tracing with Goldsmith and Salmon's method of BVH construction results in strong penalty in performance compared to top-down methods [Masso and Lopez 2003; Havran et al. 2000]. Not only the performance of data structures constructed in bottom-up fashion is lower, but also the time required to construct these data structures can increase significantly.

We offer the following analogy between the data structures and the traditional sorting of numbers (in 1D). The top-down construction of spatial hierarchies corresponds to quicksort. The construction by insertion one-by-one corresponds to sorting by insertion [Knuth 1998, page 80]. The construction in bottom-up fashion corresponds to mergesort [Knuth 1998, page 158]. The major difference between 1D sorting and sorting in N-dimensional space is exactly the increase of data dimensionality. We make the following observation: only top-down construction algorithms keep during placement of any interior node v the data sorted with respect to already existing interior nodes and the "pivot" used for v. The data structures constructed this way keep the expected time complexity of their construction $O(N \log N)$ with high probability, similar to quicksort. This is very likely the reason why the hierarchical data structures constructed in top-down fashion outperform the other two construction methods. According to experiments, the difference in performance is significant in particular for a higher number of objects [Masso and Lopez 2003; Havran et al. 2000].

3 From Kd-trees to Uniform Grids via Arity

In this section we discuss the common properties of spatial subdivisions. Using any spatial subdivision we restrict the ray-object intersection computations along the path of the ray. This requires to identify the elements of the spatial data structures along the ray. These elements are often referred to as *spatial cells*, while the identification of these data elements is called a *ray traversal algorithm*.

The way of organization of these spatial data structures has strong impact on the performance of ray tracing with these data structures. In principle both spatial subdivisions and object hierarchies can be characterized by the number of children of a node that subdivides either a spatial region into subregions or the set of objects into subsets of objects. This is usually referred to as *arity* or *branching factor* or *fanout factor*. Below we list elementary spatial subdivisions according to their arity:

- kd-trees used in ray tracing [Kaplan 1985; MacDonald and Booth 1990] have nodes with the arity two; a box associated with a node is subdivided into two disjoint boxes. The splitting planes in the nodes are axis aligned. This allows for an efficient ray traversal algorithm, since there are only four cases how the two child nodes of a node are traversed. In addition, the positioning of the splitting planes represented in the interior nodes has high flexibility. It has been shown that use of surface area heuristics instead of spatial median positions results in higher performance [MacDonald and Booth 1990; Havran 2000].

- octrees proposed to the use in ray tracing in [Glassner 1984] have nodes with the arity eight. A spatial box associated with a node of octree is subdivided to eight boxes of the same size. This allows to represent more information about the subdivision in a single node, but it requires more complex ray traversal algorithm that gives the correct order of up to four child nodes to be traversed. Several ray traversal algorithms have been proposed, for a survey see [Havran 1999]. The extension to octrees based on surface are heuristics called Octree-R [Whang et al. 1995] with higher flexibility of splitting planes positioning inside the octree node was proposed.

- EN-trees proposed by Hsing and Thibadeau [Hsiung and Thibadeau 1992] have nodes with arity 64 or 512. The spatial box associated with an interior node is subdivided to $4 \times 4 \times 4$ or $8 \times 8 \times 8$ spatial cells of the same size. Then the process of tree creation recurses. The EN-trees offer an interesting tradeoff between uniform grids and octrees.

- Uniform grid [Fujimoto et al. 1986] (regular subdivision) can also be viewed just as a node with high arity. The arity is proportional to the number of objects. The regularity of the subdivision results in two contradictory impacts on the ray tracing performance. Positively, the regularity of uniform grids allows us to implement more efficient ray traversal algorithm that steps from a cell to cell along the ray without necessity to traverse any parental hierarchical nodes. The ray traversal algorithm exploits the regularity of subdivision via 3D discrete differential analyzer algorithm referred to as 3DDDA [Fujimoto et al. 1986]. On the negative side, the regularity of the data structures does not allow any adaptability of the data structure to distribution of objects in the scene. Therefore uniform grids are rather inefficient for scenes with moderately and highly skewed object distributions. To reduces this problem several algorithms exploiting recursion and resulting in recursive grids were proposed, however it is difficult to control the memory consumptions of the recursive grids [Chang 2004; Havran 2000].

The arity and flexibility of positioning subdivision elements such as splitting planes has high impact on the performance of the data structures. It is easy to show that the geometry of spatial subdivisions based on axis aligned splitting planes with arity greater than two can be emulated by kd-trees. The possibility of emulation raises the question which data structures are the most efficient and why. We discuss below the impact of arity on performance.

Low arity. Kd-trees are highly adaptive to the distribution of objects in a scene and the tree has the depth in order of $O(\log N)$ for N objects. Initially, a ray traversal algorithm has to descend to the first leaf according to an input ray. The computation is finished after traversing several leaves when the ray intersects an object. A relatively small number of traversed leaves on average is achieved thanks to the high adaptability of splitting planes; the large spatial regions without objects are covered by large empty cells and these spatial regions are traversed quickly. The distribution of objects in a scene has only a low impact on the algorithm performance. Since an efficient ray traversal algorithm has to descend to the first leaf, a ray tracing with kd-trees shows a logarithmic behavior.

High arity. Uniform grids are highly regular and an initial cell where a ray meets the first leaf is computed in $O(1)$ time unlike $O(\log N)$ time required for kd-trees. However, the uniform grids are very inefficient for scenes with skewed distributions of objects. First, a ray has to traverse on average many empty cells before entering a cell with objects, where the intersection can be found. If the number of cells is $O(N)$, the number of traversed cells is of order $O(\sqrt[3]{N})$. Second, there are on average many references to objects in a single non-empty cell. The ray has to be tested against all objects in the cell to find out the closest intersection. As a result, the performance of ray shooting with uniform grids is rather low for skewed object distributions, in practice sometimes by order(s) of magnitude lower than the one achieved for kd-trees. In this case the factor $O(\sqrt[3]{N})$ (the number of traversed leaf cells in uniform grids) outweighs the factor $O(\log N)$. On the other hand, for uniform distribution of objects the performance of uniform grids is in practice slightly higher than for kd-trees. The reason for that is that rays in such scenes are short. In this case the location of the first cell along the ray path computed in $O(1)$ for uniform grid is faster than the location of the first leaf in kd-trees which is computed in $O(\log N)$ time.

We summarize the impact of arity on performance as follows. The low arity allows for high flexibility and adaptability of data structures to the input data. On the other hand, this induces a logarithmic time complexity. In the opposite the high arity yields the regularity of data structures which allows us to implement a very efficient ray traversal algorithm. On the other hand, the resulting highly regular data structures are not suited for irregular distributions of objects. There is a clear correlation between the performance, uniformity of data distribution, and the arity of used data structures.

4 Performance Model

In order to better understand and enumerate the performance of ray tracing algorithms and hence data structures discussed above we need a *performance model*. Recently, in years 2004-2006, the performance of ray tracing is still often referred in the numbers of rays that can be shot per second in some scenes to generate an image. Optionally, the performance is given in frames per second. However, this quantification of performance is rather application dependent and it does not document properly the qualitative properties of the used data structures and/or ray traversal algorithm as we discuss below.

The problem of an appropriate performance model is rather complex since the model has to cover a number of issues. For example, the primary rays induced by a perspective camera show high coherence of rays(similar origin and direction of rays). Therefore, it is more likely that the computation will be faster thanks to the data coherence and branch prediction in CPUs. The performance gain thanks to coherence of queries when ray tracing individual incoherent rays achieves factor between 3 to 7 on current processors according to our experiments. In our tests coherent rays were formed by primary rays of a perspective camera directed towards the scene. For incoherent rays we generated for each ray randomly one starting point and one ending point on a sphere enclosing the scene. The ray origin was at the starting point and the ray was directed towards the ending point.

There exist algorithms that explicitly use the coherence of primary or shadow rays [Reshetov et al. 2005; Wald 2004; Havran 2000]. They can be viewed as *offline searching* where we know the queries in advance while for *online searching* we process queries one by one without any knowledge of queries in advance.

The input data is given by geometric objects in the scene and rays. In order to describe the performance model more accurately, we have to consider an *ordered sequence* of rays with a particular scene. We also need to distinguish if an algorithm may access the rays in the sequence in online or offline mode of computation. The strict equivalence of the input data is a necessary condition to compare the algorithms in a fair way. An appropriate performance model of ray tracing was sketched already by Kay and Kajiya [Kay and Kajiya 1986] in the context of bounding volume hierarchies:

$$C = C_{IT} \cdot N_{IT} + C_T \cdot N_T, \qquad (1)$$

where C is the expected cost for ray tracing, C_{IT} is the expected cost of ray-object intersection, N_{IT} is the average number of ray-object intersections, C_T is the expected traversal cost among the data structure elements, and N_T is the average number of traversal steps per ray. The costs include the access time for the data stored in the cache or the main memory. The performance model was further elaborated in [Havran 2000, Chapter 2].

Since there has been an increasing bottleneck between performance of a CPU core and latency of the main memory, we propose to extend the above described performance model as follows:

$$C = C'_{IT} \cdot N_{IT} + C'_T \cdot N_T + C'_R \cdot N'_R, \qquad (2)$$

where C'_R is the cost of moving a data block from the main memory to the CPU registers and N'_R is the average number of data block moves per ray. C'_{IT} is an expected cost of ray-object intersection assuming that the data is available in the cache. Similarly, C'_T is a cost of traversal step assuming the data available in the cache.

The data block is typically a cache line in L1 and L2 cache of size between 64 and 256 Bytes on current CPUs. The data structure layout in the main memory has already been used to optimize the performance of the ray tracing algorithms [Havran 2000; Wald 2004].

The performance model proposed here has already been sketched in [Havran et al. 2006; Yoon and Manocha 2006]. Note that the last term in Eq. 2 is included in both two terms of Eq. 1. As we can distinguish the cost of reading the data from main memory in the cost model of Eq. 2, the new model is more accurate.

Sometimes it is also important to include the initial cost of precomputation required for each ray when this part cannot be considered negligible. For already proposed data structures the precomputation time for each ray is not negligible for uniform and hierarchical grids. The cost model should be then extended by precomputation cost for each ray C_{PREP}:

$$C = C'_{IT} \cdot N_{IT} + C'_T \cdot N_T + C'_R \cdot N'_R + C_{PREP}, \qquad (3)$$

From the performance model it can be seen that the search for the efficient ray tracing algorithm is implementation and hardware dependent. We cannot minimize only one term in Eq. 3, but we want to minimize the total cost to get an efficient ray tracing algorithm. A particular algorithm has the same N_{IT} and N_T independent of the implementation and computer architecture for a particular data. Obviously, the algorithm itself has some intrinsic algorithmic properties (documented by N_{IT} and N_T). However, the actual performance is significantly influenced by constants C'_{IT}, C'_T, C'_R, C_{PREP}, and N'_R that highly depend on the particular implementation and computer architecture used. Therefore there is no winning algorithm when we abstract from the implementation, computer architecture, and the input data. However, the proper documentation of algorithmic properties of published algorithms makes the experimental results reproducible and verifiable.

5 Relation between Object Hierarchies and Spatial Subdivisions

In Section 3 we have discussed the properties of spatial subdivisions in dependence on their arity. Object hierarchies in ray tracing known as bounding volume hierarchies (BVHs) are different in principle. They do not organize the space into disjoint regions as spatial subdivisions, but they hierarchically organize objects. Since each object resides in a spatial region, the BVHs also organize the space, however only indirectly. There are three major differences if we compare BVHs to spatial subdivisions. First, the spatial regions induced by child nodes and interior nodes of a BVH node can overlap in contrast to spatial subdivisions. Second, some empty spatial regions need not be covered by elementary nodes of the BVH. Third, the objects in a BVH are referenced only once in its leaves. Practically, the representation of an interior node in the BVH has higher memory consumption than spatial subdivisions, since the geometry of spatial regions associated with BVH nodes has to be described completely. The algorithm by [Kay and Kajiya 1986] creating BVHs using a "Divide and Conquer" paradigm has only two children in interior nodes, similarly to kd-trees. Papers have been recently published in which the light-weight version of BVH have been studied [Havran et al. 2006; Woop et al. 2006]. Those contain only a subset of six planes defining the box associated with a BVH node.

Below we show that spatial subdivisions and BVHs are *computationally equivalent* since axis-aligned spatial subdivisions can be emulated by BVHs and the other way round. By computationally equivalent in this context we mean that the both data structures have the same level of expressiveness for spatial sorting required by ray tracing traversal algorithm. Since commonly used spatial subdivisions (grids, octrees) can be emulated by kd-trees, we restrict our discussion only to kd-trees.

5.1 Emulation of Kd-trees via BVHs

The emulation of kd-trees via BVHs is easy to describe and implement. In general, BVH interior nodes can represent any shape, for example boxes, discrete orientation polytopes (k-DOPs), spheres, ellipsoids. If we restrict BVH nodes to be represented by boxes, then in both kd-trees and BVHs the interior nodes and leaves correspond to axis aligned boxes. Obviously, the BVHs need more memory than the kd-trees (increasing term $C_R \cdot N_R$ in Eq. 2).

The leaves of BVH can contain references to only a single object, whereas the leaves of kd-trees can contain references possibly to several objects. Another possibility is that a leaf of a kd-tree is empty, i.e., it does not contain any object. All the cases are easy to implement in BVHs. First, empty leaves of a kd-tree need not be included, since empty spaces are cut away already in parental nodes. Second, the kd-tree leaves with a single object are directly mapped to BVH leaves. Third, the kd-tree leaves with multiple references (say n) to objects are emulated by a small linear structure of completely overlapping $n-1$ interior BVH nodes. The structure is depicted in Fig. 1. Each interior node has at least one leaf with a reference to an object. Since the boxes associated with interior BVH nodes completely overlap, we have to traverse them all and compute ray-object intersections with all n objects.

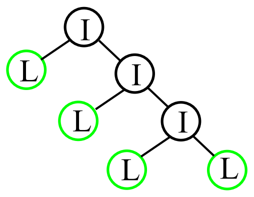

Figure 1: Emulation of kd-tree leaf with four objects in BVH - a linear tree with four child nodes, each child node contains a reference to a single object. The BVH interior nodes overlap completely.

5.2 Emulation of BVHs via Kd-trees

Much less obvious is the feasibility of emulation BVHs via kd-trees, since the spatial regions of two child/interior nodes of BVHs can overlap. This is clearly impossible if we stick to common kd-trees used for ray tracing in 3D space [Havran 2000; Wald 2004] with three types of interior nodes where a splitting plane is perpendicular to one of x, y, and z axis. The nodes of the kd-tree form a spatial subdivision in 3D and therefore BVHs cannot be emulated by the 3D kd-trees.

However, we can increase the dimensionality of the kd-trees and solve the problem in a different way. In order to emulate BVH via kd-trees we propose to use *six-dimensional kd-trees* and change the meaning associated with the dimensions. Recall that a 3D axis aligned box is described by min and max values for all three axes, in total by six values. We can then interpret the box as a point in six-dimensional space and construct a kd-tree in this "six-dimensional min-max space". A *min–A* node ($A \in \{x,y,z\}$) of the 6D kd-tree with a single splitting plane at V says that a minimum value in axis A is at most V in the left child and at least V in the right child. Similarly, a *max–A* node ($A \in \{x,y,z\}$) of the 6D kd-tree with a single splitting plane at V says that a maximum value in axis A is at most V in the left child and at least V in the right child. The geometrical interpretation of the *min–A* and *max–A* nodes is depicted in Figure 2. In total, the proposed 6D kd-trees have seven types of nodes; three nodes to limit minimum in either x, y, or z axis, three nodes to limit maximum in either x, y, or z axis, and a single leaf node. Any BVH node can be emulated by at most 6 interior nodes of the 6D kd-tree.

Changing the dimensionality of kd-trees requires only small changes to the ray traversal algorithm. Upon accessing an interior node there are four cases as for traditional 3D kd-trees:

1. we traverse only the left child,
2. we traverse only the right child,
3. we traverse first the left child and then the right child,
4. we traverse first the right child and then the left child.

These four cases are distinguished in the ray traversal algorithm using two subsequent *IF* commands. A difference is that for *min–A* nodes we limit the value of signed distance from below ("tmin" value) and for *max–A* nodes we limit the value of signed distance from above ("tmax" value).

Special nodes can be used to optimize the kd-trees traversal if one child of interior node is empty. For *min–A* nodes then the left child

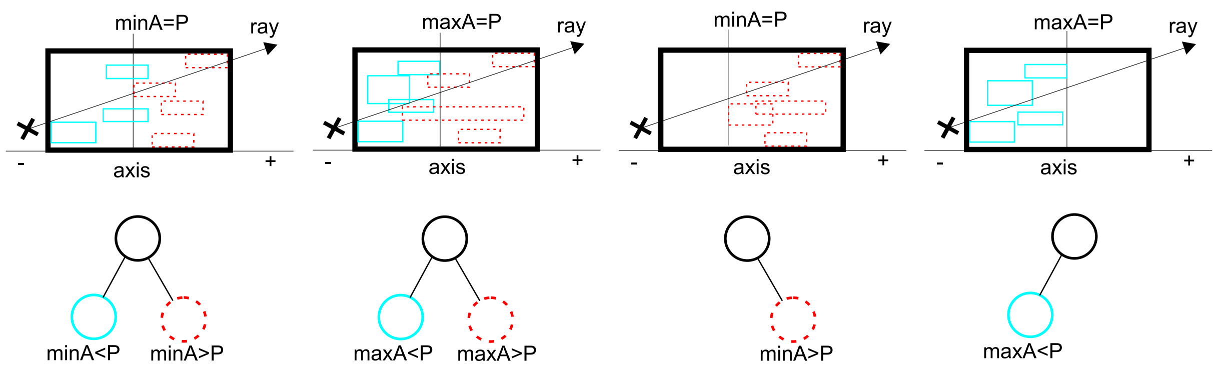

Figure 2: Visualization of six-dimensional kd-tree interior nodes for a single axis with a splitting plane placed at P. From left to right: *min–A* node with two children, *max–A* node with two children, *min–A* node with one child, *max–A* node with one child.

is empty and for *max–A* nodes the right child is empty. The proposed optimized 6D kd-trees have then thirteen types of nodes:

- three *min–A* nodes with both children (one for each axis x, y, z),
- three *max–A* nodes with both children (one for each axis x, y, z),
- three *min–A* nodes with only the right child (one for each axis x, y, z),
- three *max–A* nodes with only the left child (one for each axis x, y, z),
- leaf node containing reference to the object.

We have implemented the optimized 6D kd-trees with the thirteen types of nodes and tested their performance for 42 scenes of various complexity and object distributions. We only present the main result here; the cost overhead of emulating BVHs by 6D kd-trees yields from 20 to 40% for our implementation. Clearly, 6D kd-trees are slower, but by a constant factor compared with BVHs, as the memory requirements of 6D kd-trees are higher than for BVHs, where the information about the 6D spatial extent is more compactly represented in every node.

Further we note that the construction of the 6D kd-trees and BVHs is faster than the construction of traditional 3D kd-trees used for ray tracing, since the number of references to objects equals to the number of objects.

Obviously, we cannot claim that kd-trees are (algorithmically) equivalent to BVHs in the context of ray tracing, since there are clear differences between these structures such as the number of references to objects in the data structure. However, we can say that kd-trees and BVHs are *computationally equivalent*, as they have the same ability to address the searching problem with respect to their time complexity. The difference in performance of the data structures can be expressed by constant multiplicative factor regardless the number of objects in the scene.

6 Conclusions and Future Work

In this paper we have discussed the key principles behind the performance of data structures for ray tracing. First, we have unified spatial subdivisions such as uniform grids and kd-trees via arity. Second, we have discussed the dependence of spatial data structures properties on their arity. Third, we have proposed a modification of the performance model. Fourth, we have shown how kd-trees can be emulated via BVHs and more importantly, how BVHs can be emulated via kd-trees in a six-dimensional space. Even if clearly kd-trees and BVHs are not algorithmically equivalent data structures, the feasibility of their mutual emulation shows the computational equivalence of these data structures.

As far as the principles behind the construction of data structures for ray tracing follow “Divide and Conquer” paradigm, the question which data structures are the best is a matter of their practicality and implementation on a particular hardware. We cannot claim that kd-trees are more algorithmically efficient than BVHs, since BVHs can be (with some constant overhead) emulated by kd-trees and similarly kd-trees (also with some constant overhead) can be emulated by BVHs. We can only say that this particular implementation of BVHs is more efficient or less efficient than this implementation of kd-trees. The algorithmic properties should be documented by number of traversal steps and ray-object intersection tests and the implementation quality by timings. We can assume that the publicly available scene data are used and the sequence of rays (hence their distribution) is fully described. In this way we yield reproducibility and verifiability of results on a different hardware and/or using a different implementation.

There is in principle no need to restrict hierarchical data structures for ray tracing only to spatial subdivisions or object hierarchies. We can combine in our hierarchical data structures the nodes of traditional 3D kd-tree, 6D kd-tree, and bounding volume nodes, or any other types of hierarchical nodes. An interesting research direction assuming top-down construction is to combine traditional 3D kd-tree nodes with 6D kd-tree nodes, bounding volume primitives using the local greedy heuristics based on the cost model with the surface area heuristics. A particular node to be constructed is selected based on the estimated cost of traversing this node, selecting such a node with minimum estimated cost. Such a hybrid hierarchy can be tuned to either minimize memory consumption or performance on that particular hardware or the time required for the construction of data structures or any other objectives required by an application.

Acknowledgments

We would like to thank anonymous reviewers for their comments. This research work has been partially supported by MŠMT under research program MSM 6840770014 and LC-06008 (Center for Computer Graphics).

References

AGARWAL, P. 2004. Range Searching. In *CRC Handbook of Discrete and Computational Geometry (J. Goodman and J. O'Rourke, eds.)*, CRC Press, New York.

CHANG, A. Y.-H. 2004. *Theoretical and Experimental Aspects of Ray Shooting*. PhD thesis, Politechnic University, USA.

ENDL, R., AND SOMMER, M. 1994. Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing. *Computer Graphics Forum 13*, 1 (Mar.), 3–19.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications 6*, 4, 16–26.

GLASSNER, A. S. 1984. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications 4*, 10 (Oct.), 15–22.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (May), 14–20.

HAVRAN, V., PRIKRYL, J., AND PURGATHOFER, W. 2000. Statistical comparison of ray-shooting efficiency schemes. Tech. Rep. TR-186-2-00-14, May.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2006. On the fast construction of spatial data structures for ray tracing. 71–80.

HAVRAN, V. 1999. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News 12*, 2 (Dec.), cca 10 pages.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague.

HSIUNG, P.-K., AND THIBADEAU, R. H. 1992. Accelerating ARTS. *The Visual Computer 8*, 3 (Mar.), 181–190.

KAPLAN, M. R. 1985. The Uses of Spatial Coherence in Ray Tracing. In *ACM SIGGRAPH '85 Course Notes 11*.

KAY, T. L., AND KAJIYA, J. T. 1986. Ray Tracing Complex Scenes. *Computer Graphics (Proceedings of ACM SIGGRAPH) 20*, 4, 269–278.

KNUTH, D. E. 1998. *The Art of Computer Programming, Volume 3 Sorting and Searching*. Addison-Wesley.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for Ray Tracing using Space Subdivision. *Visual Computer 6*, 6, 153–65.

MASSO, J. P. M., AND LOPEZ, P. G. 2003. Automatic Hybrid Hierarchy Creation: a Cost-model Based Approach. *Computer Graphics Forum 22*, 1, 5–13.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3, 1176–1185. (Proceedings of ACM SIGGRAPH).

SZIRMAY-KALOS, L., AND MÁRTON, G. 1998. Worst-Case Versus Average Case Complexity of Ray-Shooting. *Computing 61*, 2, 103–131.

SZIRMAY-KALOS, L., HAVRAN, V., BALÁZS, B., AND SZÉCSI, L. 2002. On the Efficiency of Ray-shooting Acceleration Schemes. In *Proceedings of SCCG*, ACM Siggraph, A. Chalmers, Ed., 89–98.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

WHANG, K. Y., SONG, J. W., CHANG, J. W., KIM, J. Y., CHO, W. S., PARK, C. M., AND SONG, I. Y. 1995. Octree-R: an Adaptive Octree for Efficient Ray Tracing. *IEEE TVCG 1*, 4 (Dec.), 343–349.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware (2006)*, 67–77.

YOON, S.-E., AND MANOCHA, D. 2006. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Proceedings of the 2006 Eurographics Conference*, 507–516.