PERGAMON

Computers & Graphics 27 (2003) 593–604

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Technical section

On comparing ray shooting algorithms

Vlastimil Havran[a,*,1], Werner Purgathofer[b]

[a] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
[b] Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9/186, A-1040 Wien, Austria

Accepted 15 April 2003

Abstract

In this paper we discuss a methodology for comparing various ray shooting algorithms through a set of experiments performed on a set of scenes. We develop a computational model for ray shooting algorithms, which allows us to map any particular ray shooting algorithm to the computational model. Further, we develop a performance model for ray shooting algorithms, which establishes the correspondence between the computational model and the running time of the ray shooting algorithm for a sequence of ray shooting queries. Based on these computational and performance models, we propose a set of parameters describing the use of a ray shooting algorithm in applications. These parameters allows us to make a fair comparison of various ray shooting algorithms for the same set of input data, i.e., the same scene and the same sequence of ray shooting queries, but virtually independently of hardware and implementation issues. Under certain conditions, the proposed comparison methodology enables cross-comparison of published research work without reimplementation of other ray shooting algorithms.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: Ray shooting; Benchmarking; Visibility; Ray tracing; Global illumination

1. Introduction

Shooting a ray is one of the fundamental geometric tasks in computer graphics. It is utilized by virtually all modern global illumination methods to sample three-dimensional space. Ray shooting is used not only for image synthesis in ray-tracing based methods, but also for form-factor computations in radiosity, for photon map construction, for visibility preprocessing, etc.

Ray shooting is a simple task: find out the first object intersected by a given ray for a given set of N objects, if such an object exists. In spite of this simple formulation, it is not easy to implement an efficient and fast *ray shooting algorithm* (*RSA*). The problem of finding an ultimately efficient *RSA* still remains open. Both computational geometry and computer graphics researchers have tried to develop a fast *RSA* with varying success. Computational geometry [1] aims its effort at improving the worst-case time complexity, however, the methods developed are often restricted to certain shapes of objects (convex polygons, triangles, spheres). Unfortunately, the space and preprocessing time complexity of these methods is unacceptable for real implementations in rendering frameworks. Szirmay-Kalos and Márton [2] proved that the lower bound on the worst-case complexity of ray shooting is $\Omega(\log N)$ in a computational model based on algebraic decision trees. Their next result is the lower bound of space complexity $\Omega(N^4)$ (and thus of preprocessing) for any worst-case *RSA* working in $O(\log N)$ time complexity,

*Corresponding author. Tel.: +49-681-9325415; fax: +49-681-9325499.

E-mail addresses: havran@mpi-sb.mpg.de (V. Havran), purgathofer@cg.tuwien.ac.at (W. Purgathofer).

[1] A major part of the text was written on leave from the Department of Computer Science, Czech Technical University, Karlovo nám. 13, CZ-12135 Prague 2, the Czech Republic.

0097-8493/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0097-8493(03)00103-1

which makes these *RSAs* practically inapplicable in computer graphics applications.

It is also possible to use a brute force method for solving the problem. A naive *RSA* tests all the scene objects in order to select the closest one, which gives the complexity of $O(N)$, where N is the number of objects. However, for a higher number of objects this is impractical since the time consumed by the method is unacceptable.

In the field of computer graphics various *heuristic RSAs* [3] aimed at average-case time complexity have been investigated since 1980. These *RSAs* usually assume that a ray-object intersection test is available for each shape of object, which allows us to use general shapes of objects. We have to recall that the time complexity of the heuristic algorithms for the worst case is no better than for the naive *RSA*. However, good average-case complexity is the reason why the heuristic *RSAs* are commonly used in rendering packages [4–6].

Some heuristic *RSAs* exhibit worst-case complexity $O(N)$ and average-case complexity $O(1)$ for scenes with uniformly distributed objects [7]. These complexities remain for non-random scenes as well, but unfortunately, the unknown multiplicative factor hidden behind the O-notation and random object distribution in the scene for average case analysis make theoretical complexity definition unusable in deciding which *RSA* should be used in practice. This is the reason why the performance of *RSAs* is commonly compared experimentally on a set of some test scenes. Within this context, Haines introduced the *Standard Procedural Database* (SPD) package [8] in order to test the performance of ray tracing algorithms. SPD defines a set of scenes and description of ray tracing algorithm to be used for testing. Recently, Smits and Jensen proposed another set of test scenes [9], which defines scenes suitable for testing global illumination algorithms. Even more recently, Lext et al. [10] provided a scene benchmark for animated ray tracing.

In spite of two decades of research on *RSA* in the computer graphics community, it is not yet clear whether some particular *RSA* is more convenient and/or more efficient than any other *RSA*. Some contradictory statements about *RSAs* have appeared with the introduction of new types of *RSAs* in published papers. Moreover, each paper that introduces a new *RSA* must, or at least should, compare the proposed algorithm with some reference algorithm. There is no common choice for the reference algorithm, but in most cases a uniform grid was used using the $\sqrt[3]{N}$ rule to set the uniform grid resolution. The quantitative comparison between a reference *RSA* and a newly proposed *RSA* always depends on the software implementation and on a particular hardware platform. For this reason any cross-comparison of the results presented in the different papers is problematic and usually impossible. A fair comparison of different *RSAs* has been possible only when they were implemented and tested within a uniform software framework, and when testing was performed on the same hardware. Only a few papers devoted to the comparison of various *RSAs* [11–14] have been published until now, which use running time for comparing *RSAs*.

Since *RSAs* are implemented and tested using different software and hardware, an important problem in research on *RSAs* is how to compare them qualitatively and quantitatively. This should be done on a technically sound basis, it should define time and memory complexity, suitability for various types of scenes, and particular features of an *RSA*.

Classical worst-case complexity measures are rather wrong for evaluating ray shooting since they rank those algorithms first, which are intuitively unacceptable due to their memory requirements. Worst-case complexity analysis may extract those input cases which happen very rarely, thus they cannot be accepted as typical inputs. Average-case complexity is better and can explain why heuristic *RSAs* are good, but cannot make a distinction between different heuristic *RSAs*. All complexity measures hide physical parameters behind O-notation, while different algorithms belonging to the same complexity class can have very different performance. The objective of this paper is to propose a model that solves all of these problems, also taking into account the parameters of the actual computer.

In this paper we will try to decrease the gap in understanding the functionality of various *RSAs* by finding out their commonalities. The commonalities found in all *RSAs* allow us to describe the *RSA* computational model in a general way that enables us to map a particular *RSA* to this computational model. Furthermore, we will describe a performance model which establishes a connection between the running time of the application and the different algorithmic operations that are the subject of the computational model. We will then define two procedures for an "ideal *RSA*" that provides the minimum time on which ray shooting can be solved on a given machine. We use the time consumed by the "ideal *RSA*" as a reference value for all the parts of the computation in a specific *RSA*. These parts cover the time needed to traverse the data structure on which an *RSA* is based, ray-object intersection tests, and the remaining time consumed by the application. The design of the computational model and performance model allows us to define a set of thirteen parameters referred to as the minimum test output that should be reported for one experiment, given a scene, a particular *RSA*, and a sequence of ray shooting queries. The definition of the two models and the minimum test output is the basis for

a methodology for making a fair comparison of various *RSAs*.

The concepts proposed below form a comparison methodology that allows us to compare various *RSAs* virtually independently of the implementation and the hardware used. We follow the comparison methodology for experimental measurement that was described in Havran and Žára [15] for the kd-tree and further elaborated in [11].

To make the scope of this paper clearer, we have to say that *ray tracing* is a particular global illumination method. Ray tracing, similarly to other global illumination algorithms, uses some *RSA* to evaluate pixel illumination. Even if it is possible to define other generalized rays (beam, cone, and pencil tracing, see [3]), the computation of these is usually considered time consuming [3]. Practically all global illumination algorithms that can produce general lighting effects such as Monte Carlo techniques stick with the concept of a ray as a half-line, as presented here. Further, we do not consider these generalized rays, nor refer to different ray tracing methods. Instead, we limit ourselves to ray shooting methods only. Consequently, we from now on assume the existence of an application that deterministically generates a set of rays for a given input set of objects and uses an *RSA* to compute the result. However, the necessary condition and limitation for any comparison methodology is that the ray-object intersection test is required within an *RSA*. We do not include the algorithms, in which results of the *RSA* are approximated, e.g., with the use of an OpenGL z-buffer for eye rays in ray casting etc. All the requirements we pose are common in applications using *RSAs*, particularly in global illumination algorithms.

This paper is further structured as follows: In Section $\sim$2 we introduce an *RSA* computational model. In Section $\sim$3 we describe an *RSA* performance model. In Section $\sim$4 we develop an "ideal *RSA*" that allows to compute the answers to ray shooting queries in constant time under certain conditions. In Section 5 we describe a minimum test output and in Section 6 the method of getting values for minimum test output. In Section 7 we present a comparison methodology which describes how to compare two or more *RSAs*, and in Section 8 we discuss various properties of our method. Section 9 concludes the paper.

2. A computational model of ray shooting algorithms (RSAs)

In this section we describe the *RSA computational model* for *RSAs* based on ray-object intersection tests. We will show that any such *RSA* currently known or developed in the future can be mapped to this computational model. The computational model is based on the definition of algorithmic operations in an *RSA*. These algorithmic operations must always be performed due to the nature of the ray shooting problem. We then use the computational model to describe the set of parameters to be reported, when an *RSA* is tested experimentally.

The ray shooting problem can be understood as an instance of geometric range-searching [16], which implies that some data structure is built to answer the specific query. The definition of ray shooting implies that every *RSA* somewhere contains pointers to objects that are to be tested for intersection against a given ray. This means that each *RSA* is separated into two parts (as are all algorithms [17]): the *data structure* (*DS*) at least containing pointers to the scene objects and the *ray traversal algorithm* working over the *DS*. The lifetime of an *RSA* is composed of two phases, the first one is called *preprocessing phase* and involves the construction of the initial *DS*. The second phase of an *RSA* is called the *execution phase*. Within the execution phase the *RSA* answers given ray shooting queries.

A *DS* contains some data entries, here referred to as *nodes*. The nodes include the pointers to the scene objects, the pointers to the other nodes of the *DS*, the size of the spatial cells, etc. Nodes of a *DS* can be divided into two groups: *elementary nodes* are intended to contain only pointers to objects (and, if the *RSA* requires it, some other data), whereas *generic nodes* are all other nodes, which serve to point to other generic and/or elementary nodes. A special case of elementary nodes are *empty elementary nodes* that do not contain any pointers to objects and act as "free space containers" within the *DS*.

For any answer to a ray shooting query in a particular *RSA*, computation proceeds as follows. Given a ray R, a ray traversal algorithm performs a sequence of the following operations:

TRAVERSAL STEP:	visit a new node of the *DS*,
NEW NODE:	create a new node of the *DS*,
DELETE NODE:	delete a node from the *DS* and unlink all pointers to the node from the remaining nodes of the *DS*,
TEST OBJECTS:	when accessing an elementary node of the *DS*, test objects pointed to in this node for intersection with the ray R,

finally finding the closest intersected object if such an object exists. There are two cases: a *DS* is changed by a ray traversal algorithm or it may not. If a *DS* underlying the *RSA* is not changed by the ray traversal algorithm, then the operations "NEW NODE" and "DELETE

NODE'' are not performed during the execution phase. Such an *RSA* is referred to as the *RSA* based on a *static data structure*.

There are several *RSAs* that modify the underlying *DS* on the fly within the execution phase, for example, ray space subdivision techniques [18]. The operations ''NEW NODE'' and ''DELETE NODE'' are usually used within the preprocessing phase to build up some initial *DS*, however, in this case the *DS* is modified during the execution phase. Such an *RSA* is referred to as the *RSA* based on a *dynamic data structure*.

As we can see, each *RSA* based on ray-object intersection tests must contain a data structure where the pointers to objects are stored, with at least one pointer for each object. From all this it follows that through this underlying data structure it is possible to map any *RSA* based on ray-object intersection tests to the general *RSA* computational model above. This enables us to define a common set of parameters to be reported when any *RSA* is performed on an input scene S containing N objects over an input sequence of ray shooting queries. Further, we call the *testing procedure* an algorithm in the application that generates a sequence of ray shooting queries to be answered by a particular *RSA*. (The symbol for the testing procedure is *TP*.) A particular testing procedure *TP* can be the result of a global illumination algorithm such as ray tracing, etc., or just an artificial algorithm shooting rays to obtain some required distribution of rays in space [11].

We propose to organize the set of parameters resulting from the use of a particular *RSA* on the input scene, and given a *TP* into three subsets, the first two of them hardware/implementation/compiler independent:

- *RSA* parameters related to static properties of data structure *DS*:
  - If an *RSA* is based on a static data structure, the parameters depend on the scene S only, and they are evaluated at the end of the preprocessing phase.
  - If an *RSA* is based on a dynamic data structure, the parameters depend on the scene S and the testing procedure *TP*, and they are evaluated during the execution phase as maximum values reached.

    N_G [−]—maximum number of generic nodes in the *DS*,
    N_E [−]—maximum number of elementary nodes in the *DS*,
    N_{EE} [−]—maximum number of empty elementary nodes in the *DS* ($N_{EE} < N_E$),
    N_{ER} [−]—maximum number of pointers to objects in all the elementary nodes of the *DS* (N_{ER} is greater than or equal to the number of objects N).

- *RSA* parameters related to dynamic properties of a data structure *DS*, i.e., the use of the *DS* within an *RSA*. They depend on the scene S and the testing procedure *TP*, and they are evaluated at the end of the execution phase:

  r_{ITM} [−]—ratio of ray-object intersection tests performed to minimum number of ray-object intersection tests ($r_{ITM} \geqslant 1.0$, assuming at least one object is intersected given S and *TP*. The minimum number of ray-object intersection tests corresponds to the number of rays hitting objects.),
  $\tilde{N}_{TS}$ [−]—average number of (all) *DS* nodes accessed per ray ($\tilde{N}_{TS} \geqslant 1.0$),
  $\tilde{N}_{ETS}$ [−]—average number of elementary *DS* nodes accessed per ray ($\tilde{N}_{ETS} \leqslant \tilde{N}_{TS}$),
  $\tilde{N}_{EETS}$ [−]—average number of empty elementary *DS* nodes accessed per ray ($\tilde{N}_{EETS} \leqslant \tilde{N}_{ETS}$).

- *RSA* hardware/implementation/compiler dependent parameters. Obviously, these parameters also depend on the scene S and testing procedure *TP*:

  T_B [*s*]—time consumed to *build* an initial *DS* for the *RSA* in the preprocessing phase (it does not depend on a *TP*),
  T_R [*s*]—*running* time of the application that uses the *RSA*. It involves the execution phase of the *RSA*, which also covers the time devoted to other computations than the *RSA*. (Obviously, T_B is not included in T_R.)

We consider the parameters in the first two subsets as the minimum hardware/implementation independent parameters to be reported. It is certainly possible to extend the set of parameters even further (for example, the variance of number of objects in leaves), but we want to keep this set of the smallest possible size that still characterizes an *RSA* through the computational model.

The parameters T_B and T_R depend not only on the hardware used, but also on the quality of implementation (and programming language), the compiler used and its version, the optimization switches used for compilation, etc. For this reason, all of these experimental conditions should be described in detail. The treatment of these parameters related to the implementation makes the problem of comparing various *RSAs* rather difficult; we describe our solution to the problem in more detail below.

3. *RSA* performance model

The *RSA* computational model enables us to count the number of basic algorithmic operations performed

on average in an *RSA*. The *RSA* computational model does not define any cost of these operations in terms of running time, it only describes the time T_B and T_R. For the sake of convenience, we further use the term *cost*[*s*] as the running time required to perform some particular algorithmic operation.

In order to establish the relationship between hardware/implementation dependent and independent parameters, we further develop an *RSA performance model*. The concept of the performance model for an *RSA* was first introduced by Cleary and Wyvill [19] in the context of uniform grid analysis. Here we present a more general performance model for any *RSA* that is derived from the *RSA* computational model described above. The *RSA* performance model is based on the decomposition of the total running time T_R of the application that uses an *RSA* into three parts:

- computing ray-object intersection tests,
- traversing nodes of the *DS*, and
- the remaining computation effort required by the application.

For any application that uses an *RSA* based on ray-object intersection tests we can divide the time T_R into the parts mentioned above. We put into relation the time-dependent and independent characteristics by means of cost consumed by specific algorithmic operations as follows:

$$T_R = (r_{ITM} \cdot r_{SI} \cdot \tilde{C}_{IT} + \tilde{N}_{TS} \cdot \tilde{C}_{TS}) \cdot N_{RAYS} + T_{APP}, \quad (1)$$

where r_{SI} is the ratio of the number of rays intersecting objects to the number of all rays ($r_{SI} < 1.0$). We can remark that r_{SI} does not depend on an *RSA*, it is given by *TP* and *S*. Then the average number of ray-object intersection tests per ray is $\tilde{N}_{IT} = r_{ITM} \cdot r_{SI}$. Further, $\tilde{C}_{IT}[s]$ is the average cost of a ray-object intersection test, $\tilde{C}_{TS}[s]$ is the average cost of the traversal step of a ray traversal algorithm among the nodes of the *DS*, N_{RAYS} is the total number of rays induced by a testing procedure *TP*, and $T_{APP}[s]$ is the *remaining application time*. The time T_{APP} covers other computational effort performed in the application, for example, in a rendering application T_{APP} might cover the time consumed to compute the ray reflection, lighting, texturing, and other material calculations. Thus T_{APP} is always constant for a particular scene *S* and testing procedure *TP*, provided the same implementation and hardware is used.

We can refine the performance model if we consider the ratio of successful ray-object intersection tests to all intersection tests computed:

$$T_R = [(\tilde{N}_{IT}^{succ} \cdot \tilde{C}_{IT}^{succ} + \tilde{N}_{IT}^{fail} \cdot \tilde{C}_{IT}^{fail}) \cdot r_{SI} + \tilde{N}_{TS} \cdot \tilde{C}_{TS}] \cdot \tilde{N}_{RAYS} + T_{APP}, \quad (2)$$

where $\tilde{N}_{IT}^{succ}$ is the average number of successful ray-object intersection tests per ray, $\tilde{C}_{IT}^{succ}[s]$ is the average cost of successful ray-object intersection tests, $\tilde{N}_{IT}^{fail}$ is the average number of failed ray-object intersection tests per ray, and $\tilde{C}_{IT}^{fail}[s]$ is the average cost of failed ray-object intersection tests.

4. Ideal *RSA*

Having described the refined performance model, we can now introduce the "*ideal RSA*" as an *RSA* that has the best possible performance. The concept of the "ideal *RSA*" serves us as the ultimate but in practice unachievable goal. However, it is important since the running time of the "ideal *RSA*" is used as the reference time value for comparing various *RSAs*.

Definition. An "*ideal RSA*" is an *RSA* that for any ray shooting query computes the answer in $O(1)$ time independently of whether an intersected object exists or not. The multiplicative factor hidden behind the O-notation is very small. (For example, it is close to reading a single element of an array from the memory.)

We should recall that each *RSA* is in fact a query of a data structure, which is described by the parameters of the ray. The "*ideal RSA*" aims at minimizing the time required by the query to the smallest possible value provided the ray-object intersection is computed at most once.

Since Szirmay-Kalos and Márton [7] proved that any *RSA* works at least at time $\Omega(\log N)$ for the worst-case input, we can ask whether the definition of an "ideal *RSA*" makes sense. Inspired by the idea of *Parametrized Ray Tracing* [20], we can construct the "ideal *RSA*" provided the same testing procedure *TP* is repeatedly performed for the same scene *S*. Further, it is required that the application code be deterministic in the sense that the testing procedure *TP* in the application always generates *the same sequence of ray shooting queries* for a given scene. This can require the setting of initial seeds in pseudo-random generators to the same value in the application, etc.

Further, we describe the two procedures that form the "ideal *RSA*". The first assumption that enables us to execute the "ideal *RSA*" is that the application is run at least twice using the same *TP* and *S*. Each object is assigned an identification tag ID (integer) in the range $\langle 0, N-1 \rangle$. Then we construct the array A_T of pointers to objects. An object is then accessed directly in A_T using IDs of the object in $O(1)$ time.

In the first application run we use some "conventional" *RSA* to compute the results of given ray

shooting queries. The results obtained by the "conventional" *RSA* for the sequence of input ray shooting queries generated by *TP* are saved one by one to a temporary array A_S using the objects' ID. When no object is intersected, the array entry is set to a special ID value ($\mathrm{ID}_{spec} = -1$). The procedure that is used in the first application run and at the interface between the application and the "conventional" *RSA* is outlined in the pseudocode, Algorithm 1.

Algorithm 1. The first run of an "ideal *RSA*" that saves the results of ray shooting queries.

```
{Preprocessing phase}
Assign each object a unique ID in the range ⟨0, N − 1⟩.
Allocate the array A_S to store IDs of objects, the size of
A_S must be greater than or equal to the number of all ray
shooting queries generated by TP.
{the pointer to the array—order of ray shooting query}
i ← 0
{Execution phase}
function ShootRay(ray R): object
{Compute the result of the ith ray shooting query by
some "conventional" RSA.}
Compute the result for R using some "conventional"
RSA.
Object O ← the result of the "conventional" RSA for the
given R.
if object was found then
   A_S[i] ← ID of object O
else
   A_S[i] ← ID_spec
end if
i ← i + 1
ShootRay ← object O
{Postprocessing phase—possibly save A_S to external
memory}.
```

In the second (repetitive) application run, instead of calling a specific *RSA*, we read the correct answer to the ray shooting query from the array A_S provided that repetitive run(s) of the application result in the same testing procedure *TP* and use the same scene *S*. If we obtain the object's valid ID, we obtain the address of the object through the array A_T and compute the exact intersection point by one ray-object intersection test. This computation is required to obtain the correct signed distance for the current ray shooting query. If the object's ID has the value ID_{spec}, then the answer to the ray shooting query is "no object", and no signed ray-object intersection test is computed. Since the ray-object intersection test is computed at most once for each ray shooting query, the "ideal *RSA*" runs in $O(1)$ time. The "ideal *RSA*" performed in a repetitive run of the application is outlined in the pseudocode, Algorithm 2.

Algorithm 2. The second run of an "ideal *RSA*" reading the results of ray shooting queries.

```
{Preprocessing phase}
Assign each object its unique ID in the range ⟨0, N −
1⟩. These ID correspond to the first run of an "ideal
RSA".
Allocate the array A_S to store ID of objects.
{Possibly read A_S from the external memory.}
Allocate the array A_T to store the pointers to objects,
size of A_T is the number of objects.
for each object O specified by its ID do
   A_T[ID] ← address of the object O
end for
{the pointer to the array—order of ray shooting query}
i ← 0
{Execution phase}
function ShootRay(ray R): object
ID of object ← A_S[i]
i ← i + 1
if ID ≠ ID_spec then
   Object O ← A_T[ID]
   Compute the signed distance for the ray R and the
     object O.
else
   Object O ← "no object"
end if
ShootRay ← object O
```

For the repetitive run(s) of the "ideal *RSA*" the time T_R becomes the *minimum application running time* T_R^{MIN}:

$$T_R^{MIN}[s] = T_{RSA}^{MIN} + T_{APP}, \tag{3}$$

where T_{RSA}^{MIN} is the minimum time devoted to ray shooting only, further called the *ideal ray shooting time*:

$$T_{RSA}^{MIN}[s] = \tilde{C}_{IT}^{succ} \cdot N_{RAYS} \cdot r_{SI}. \tag{4}$$

As we can see, the "ideal *RSA*" is also a kind of average since it is different for different object types according to different ray-object intersection tests.

5. Minimum test output

The results of experiments published in the papers introducing new *RSAs* were often restricted only to times T_B and T_R and some other parameters. Mostly based on these hardware dependent parameters, we could not fairly compare newly introduced *RSAs* with those published in the past. It follows from the description of the computational and performance model that experiments allowing us to fairly compare various *RSAs* must be performed for the same scene *S* and testing procedure *TP*. For this purpose Haines

introduced the *Standard Procedural Database* SPD [8], which enables us to procedurally generate various scenes with various numbers of objects. It also defines some standard sizes of the scenes that should preferably be used for testing *RSAs*. The ray tracing as an algorithm generating rays corresponds to the *TP*.

However, the use of SPD scenes for testing *RSAs* has also been violated, and research papers often show results for testing performed on private scenes, or on only a small subset of SPD scenes. The selection of scenes is important, since it might be biased to get better results for the algorithmic technique presented in the research paper than for other scenes. For this reason it is highly desirable to use either standard test scenes or to provide the tested scenes used in a research paper for public use. We would like to recall that the nature of scientific work is that every research paper should describe new techniques and experiments with results that are *reproducible* and *verifiable* by all following researchers [21]. Therefore, whenever possible, qualitative properties of algorithms obtained via experiments should be tested on non-private input data.

Let us discuss why the comparison of various *RSAs* based only on time T_R consumed by the whole application is not fair. The first reason is that T_R also includes T_{APP}, which is constant. If we want to compare the ratio of performances of various *RSAs* on the same hardware and with the same implementation, instead of comparing T_R^1 for RSA^1 and T_R^2 for RSA^2 it is more correct to compare $(T_R^1 - T_{APP})$ with $(T_R^2 - T_{APP})$, since this considers the time devoted to ray shooting only. The value of T_R can be used correctly only for ranking the *RSAs*, but it cannot be used to express by how much a given *RSA* is faster than another *RSA*.

The SPD package [8] also recommends that some time-independent characteristics should be reported: N_{RAYS}, $\tilde{N}_{IT}^{succ} \cdot N_{RAYS}$, $\tilde{N}_{TS} \cdot N_{RAYS}$. We follow this approach by extending this set of hardware/implementation/compiler independent characteristics.

In order to avoid mutually contradictory statements in further papers concerning *RSAs*, we define a set of parameters to be reported from the experiments. We call the set of parameters the *minimum test output*. This consists of three subsets as already presented: *RSA* parameters that relate to static properties of the *DS*, *RSA* parameters that relate to dynamic use of the *DS*, and *RSA* parameters dependent on hardware/implementation. The hardware/implementation dependent characteristics T_B and T_R are supplied by three other parameters. We normalize the time portions devoted to the particular phases to the ideal ray shooting time T_{RSA}^{MIN} that allow us to make a fair comparison among different implementations and different hardware used for testing. Our main goal is that the parameters in the minimum test output should allow us to compare the performance of various *RSAs* independently of hardware and implementation.

We define the minimum test output for an *RSA* as (see Section 2 for the description of most of the parameters used below):

Subset Σ of parameters describing the *static* properties of a *DS* within the *RSA*:

$$\Sigma = \{N_G, N_E, N_{EE}, N_{ER}\},$$

subset Δ of parameters describing the *dynamic* use of the data structure the *DS*, which also depend on the input scene S and on the testing procedure *TP*:

$$\Delta = \{r_{ITM}, \tilde{N}_{TS}, \tilde{N}_{ETS}, \tilde{N}_{EETS}\},$$

and subset Θ of hardware/implementation dependent parameters of the *RSA* related to *timing*, which also depend on the input scene S and on the testing procedure *TP*:

$$\begin{aligned}\Theta &= \{T_B, T_R, \Theta_{APP}, \Theta_{RAT}, \Theta_{RUN}\} \\ &= \left\{ T_B, T_R, \frac{T_{APP}}{T_{RSA}^{MIN}}, \right. \\ &\quad \frac{N_{RAYS} \cdot (\tilde{N}_{IT}^{succ} \cdot \tilde{C}_{IT}^{succ} + \tilde{N}_{IT}^{fail} \cdot \tilde{C}_{IT}^{fail})}{T_R - T_{APP}}, \\ &\quad \left. \frac{T_R - T_{APP}}{T_{RSA}^{MIN}} \right\}. \end{aligned} \tag{5}$$

The parameter Θ_{APP} expresses the ratio of the remaining application time to the ideal ray shooting time T_{RSA}^{MIN}, which is not necessary for the comparison of *RSAs*, but, suitable for other reasons as we will show later. The parameter $\Theta_{RAT}(\Theta_{RAT} \in \langle 0; 1 \rangle)$ is the ratio of time required for computing the ray-object intersection tests to the time consumed by an *RSA* only. The parameter Θ_{RUN} gives the ratio of time consumed by an *RSA* to T_{RSA}^{MIN}.

The time portions related to the ideal ray shooting time T_{RSA}^{MIN} can be difficult to measure. In the next section we deal further with this problem. The value of T_{RSA}^{MIN} enables us to compare different hardware/implementation dependent characteristics. Subset Θ contains the value of the ideal ray shooting time T_{RSA}^{MIN} only indirectly, since it can be computed as

$$T_{RSA}^{MIN} = \frac{T_R}{\Theta_{APP} + \Theta_{RUN}}. \tag{6}$$

6. Measuring the minimum test output

The minimum test output allows us to make a fair comparison of various *RSAs*. We pay for it through the additional effort needed to get this set of thirteen parameters for one experiment. The counters to obtain the subsets Σ and Δ must be coded inside the *RSA* in its preprocessing and execution phase, which is fairly easy to implement. It is advantageous to check these counters

for verification purposes as well, since they can indicate to us an implementation bug in a particular *RSA*. In order to have a correct implementation of a particular *RSA* given some testing procedures *TP* and scene *S*, the parameters N_{RAYS} and r_{SI} must have correct values when the application run is over. Although N_{RAYS} can be regarded as an independent input quantity, it is often the case that the number of rays generated is connected with the use of an *RSA* and thus N_{RAYS} is dependent on the correctness of the *RSA*. For example, this is the case for higher order rays in various global illumination algorithms. Reference values of N_{RAYS} and r_{SI} can be obtained by running another *RSA* that is known to be correct. The simplest way is to implement naive *RSA*, although the naive *RSA* is extremely slow for higher numbers of objects in the scene.

In order to obtain the subset Θ we need the total running time T_R to be decomposed into the following three portions: the time for the ray traversal algorithm performed within the *RSA*, the time of the ray-object intersection tests performed within the *RSA*, and the remaining application time T_{APP}. One way to obtain subset Θ is to use a software *profiler tool*. This is a common method for solving performance issues in software applications. It enables us to distinguish between the time portions consumed within particular software functional units, such as functions, procedures, or even the lines of source code. Then we can sum up the time devoted to ray-object intersection test routines, the time consumed by traversing the nodes of a *DS*, and the remaining application time. In order to obtain subset Θ we thus profile the normal application run (Eq. (2)) and the application run that uses the "ideal *RSA*" (Eq. (3)). Subset Θ could also be measured by instrumenting the code with appropriate timing calls.

We also require to profile the run of the application with the "ideal *RSA*". In order to get T_{RSA}^{MIN} we only need to sum up the time consumed by ray-object intersection test procedures/functions, which gives us the minimum time possible.

7. Comparison methodology

The establishment of the subsets Σ, Δ, and Θ of the minimum test output enables us to compare different characteristics of tested *RSAs*, we are mainly concerned with time and space complexity. For the use of an *RSA* in an application there are several different input features to distinguish:

S: the input scene *S* is important, for example, one kind of *RSA* can be efficient for scenes with a small number of objects, although slow for scenes with a higher number of objects. The scene influences all parameters in subsets Σ, Δ, and Θ.

RSA: the idea behind an *RSA* has a major impact on time and space complexity, which influences Σ, Δ, and Θ.

TP: the testing procedure is specific to the application used, and the use of the *RSA* can vary greatly. It only influences Δ and Θ for an *RSA* based on a static data structure, otherwise it also influences Σ.

HW: type of hardware used—this influences all parameters in subset Θ, particularly T_B and T_R.

COMP: the compiler, its version, and the switches used can influence T_B and T_R significantly, and thus all parameters in subset Θ. For example, setting optimization switch -O2 of the C++ compiler in the UNIX operating system can decrease the running time by half for some programs.

IMPL: implementation—the actual coding of the algorithm also has a great impact on its performance, depending on the programmer's experience, etc. Various implementations of the same ideas can exhibit significant differences in performance, which influences only subset Θ. When the *RSA* is (re)implemented correctly, the parameters in subsets Σ and Δ will not be influenced.

We note that *HW*, *COMP*, and *IMPL* can be intertwined to some extent, since a certain implementation can better fit a certain hardware, etc. It is obvious that so many dimensions of freedom make the comparison of various *RSAs* rather difficult in general, especially for subset Θ. For example, if we want to compare two different *RSAs*, we have to keep fixed as many other possible dimensions as possible, in this case *S*, *TP*, *HW*, *COMP*, and *IMPL*. As the minimum requirement, we demand that the same set of scenes and the same testing procedure within the application be used. The existence of dimensions *HW*, *COMP*, and *IMPL* disable the direct use of T_B and T_R for comparing various *RSAs*. Some parameters in subset Σ, Δ, and Θ allow us to compare even those cases, due to the generality of the underlying *RSA* computational and performance model.

In general, we can perform the following comparisons for one measurement using the same *TP* and scene *S* for two ray shooting algorithms RSA^1 and RSA^2 (values for RSA^1 are denoted by superscript[1], for RSA^2 by superscript[2]):

- for *space complexity*, we compare $(N_G^1 + N_E^1)$ with $(N_G^2 + N_E^2)$. To a constant factor given by an implementation of a particular *RSA*, it expresses the different memory requirements for various *DSs*.
- for *use of hierarchy* inside the *DS*, we compare N_G^1/N_E^1 and N_G^2/N_E^2.

- for *use of empty space inside the DS*, we compare N^1_{EE}/N^1_E and N^2_{EE}/N^2_E.
- for *time complexity*, we have several choices depending on the conditions for comparison:
  - T^1_B with T^2_B to compare the time required for the preprocessing phase for the same *HW*, *COMP*, and *IMPL*. If we use different *HW*, *COMP*, and *IMPL*, we have to normalize T_B to T^{MIN}_{RSA} using Eq. (6), and thus compare $T^1_B/T^{MIN,1}_{RSA}$ with $T^2_B/T^{MIN,2}_{RSA}$.
  - $T^1_R - T^1_{APP}$ with $T^2_R - T^2_{APP}$ for performance ratio, T^1_R with T^2_R for ranking only—time can be used directly for comparison, when *HW*, *COMP*, and *IMPL* attributes are the same. The *HW*/*COMP*/*IMPL* attributes must always be stated explicitly for the experiments.
  - Θ^1_{RUN} with Θ^2_{RUN}—concerns the time required for ray shooting in the application related to the ideal ray shooting time. Assuming that the implementation of ray-object intersection tests is practically the same, this enables a fair comparison independent of *HW*, *COMP*, and *IMPL* attributes. The parameter Θ_{RUN} defines how far the tested *RSA* is from the "ideal *RSA*", and thus the maximum portion of the time that could possibly be reduced by some *RSA* with a higher performance. It can be regarded as the main index of performance. Unlike comparing $T^1_R - T^1_{APP}$ with $T^2_R - T^2_{APP}$, it enables us to compare various different *RSAs* virtually independently of *HW*, *IMPL*, and *COMP*.
  - Θ^1_{RAT} with Θ^2_{RAT}—we can compare how much of the time for an *RSA* is devoted to computing ray-object intersection tests.
  - $\Theta^1_{RUN} \cdot \Theta^1_{RAT}$ with $\Theta^2_{RUN} \cdot \Theta^2_{RAT}$—concerns the portion of time for ray-object intersection tests. It can be used virtually independently of *HW*, *COMP*, and *IMPL*.
  - $\Theta^1_{RUN} \cdot (1 - \Theta^1_{RAT})$ with $\Theta^2_{RUN} \cdot (1 - \Theta^2_{RAT})$—concerns the portion of time for traversing and manipulating data structures. It can be used virtually independently of *HW*, *COMP*, and *IMPL*.
  - r^1_{ITM} with r^2_{ITM}—an efficient *RSA* should have this ratio (of ray-object intersection tests performed to the minimum number of intersection tests) which is as small as possible (as close to 1.0 as possible).
  - $\tilde{N}^1_{TS}$ with $\tilde{N}^2_{TS}$—an efficient *RSA* has a number of traversal steps per ray which is as small as possible (as close to 0.0 as possible).
  - $\tilde{N}^1_{EETS}/\tilde{N}^1_{ETS}$ with $\tilde{N}^2_{EETS}/\tilde{N}^2_{ETS}$—this shows us the utilization of empty space within the execution phase. The level of empty space utilization can have a great impact on *RSA* performance.

Based on these developments, we can formulate a *comparison methodology* for two or more *RSAs*. First, we map each tested *RSA* to the *RSA* computational model described in Section 2. Second, we measure the minimum test output for each *RSA* and a set of scenes (the scenes should be publicly available, like the SPD package [8]). The testing procedure used within the application must be the same for each scene and each *RSA* and it must be well documented. For example, four testing procedures are described in [11]. The same testing procedure guarantees the same sequence of ray shooting queries and thus the correctness and reproducibility of experiments. Then, we can compare various features of tested *RSAs* as described above, for each scene used and also as a whole set of scenes using basic statistics tools (e.g., minimum, maximum, average, variance). The minimum test output for each scene and each experiment considered as research work must be fully published. For example, when introducing a new *RSA*, for SPD scenes, a table of numbers with 10 rows (scenes) and with at least 13 columns (minimum test output) is required.

8. Discussion

From the implementation point of view, the "ideal *RSA*" is fairly easy to implement in the application. Since the number of ray shooting queries can be high, it can be necessary to save the results of ray shooting queries to external memory. In this case for the second (repetitive) run we should avoid counting the time consumed to transfer the data from external memory to internal memory to minimize T^{MIN}_{RSA}. In practice, array A_S is read from a file by blocks to internal memory, and the time for reading the blocks should not be included in T^{MIN}_{RSA}.

One might argue that any *RSA* is influenced by hardware issues, such as caching etc. However, the "ideal *RSA*" as presented here has practically the best behavior for caching that can ever be achieved. First, we access the minimum number of objects for one ray, so the hit ratio in the cache is the best one possible for a given sequence of rays. Second, the references for array A_T are linearly arranged, having also the best possible memory coherence and properties for caching of subsequent rays with respect to coherence. Third, the branching within ray-object intersection tests is predicted with 100% accuracy, since all the objects are intersected. It is hard to imagine an *RSA* based on a ray-object intersection test with better cache coherence for data access and code execution than the "ideal *RSA*". The second possible objection is that the data coherence access can be improved, for example, if the application uses space-filling curves in ray tracing. We do not want

to change the testing procedure *TP* within an application to preserve the generality of our methodology, we regard the application as a black-box producing the sequence of rays for the *RSA*. The issue of how rays are generated within an application with respect to the data access of the *RSA* into the memory hierarchy is an inherent issue of the application. A better ordering of rays can decrease the time for both the "ideal *RSA*" and the tested *RSA*.

The proposed minimum test output organized into three subsets has a total of thirteen parameters, which can be considered a high number. Nonetheless, we regard this set as the minimum output that shows different features of an *RSA*, since it is based on the general computational model that fits any *RSA*. The minimum test output contains both hardware/implementation/compiler independent and hardware/implementation/compiler dependent characteristics that allow us to make mutual comparisons of various *RSAs* under certain conditions. The disadvantage of proposed comparison methodology is the underlying assumption that the costs of the ray-object intersection tests are of equal efficiency for various shapes of objects on different implementations. Fortunately, the ray-object intersection tests for shapes of objects (at least for objects in the SPD package) are more or less standardized [22,23]. There is a set of standard scenes, and a well-defined testing procedure, namely ray tracing in the SPD package. However, we show that at least the same scene S and the same testing procedure *TP* must be used to validate the comparison.

Let us now discuss whether it is possible to manipulate the minimum test output by changing the quality of the implementation. It is not possible to influence the parameters in subsets Σ and Δ, assuming that the implementation of statistics counters and the *RSA* itself is correct. If less efficient or more efficient ray-object intersection tests are applied, then practically all the parameters in Θ are influenced, excluding T_B. However, it is virtually impossible to influence the value of $\Theta_{RUN} \cdot \Theta_{RAT}$.

9. Conclusion

In this paper we demonstrated a concept that can be applied to all *RSAs*: a computational model and performance model for the *RSA*. We described the "ideal *RSA*" that provides us with a reference value for comparing *RSAs*. Further, we presented a methodology for comparing various *RSAs*. However, after the analysis performed here it is clear that an experimental comparison of various *RSAs* still remains a difficult problem in full generality. The presented comparison methodology enables us to compare various *RSAs*, assuming that each application uses the same sequence of ray shooting queries for the same set of objects. The construction of an "ideal *RSA*" and thus the measurement of the ideal ray shooting time also shows us the time in the best possible and ideal case given hardware and an implementation. The minimum application running time expresses the minimum time of a particular application that uses an *RSA* for a given scene and testing procedure.

A by-product of this development is that we can measure how far we are from the minimum application running time ever achievable in dependence on a ray shooting algorithm, given the application implementation that computes the set of ray shooting queries on the tested hardware. For example, it can then be shown whether or not it is possible to compute a particular global illumination task such as ray tracing in real time, given a certain hardware and a certain software implementation.

The comparison methodology as described is not application-dependent with respect to the generation of rays and use of the results inside the application, which can be of a different nature. It is valid not only for applications working over static scenes computing still images, but also for applications computing animations. These methods seem to play an important role in the ongoing future of interactive rendering techniques [10,24].

Acknowledgements

The authors would like to thank Jan Přikryl, Jiří Bittner, Connie Simon and anonymous reviewers for comments on previous versions of the paper. This work has been partially supported by the joint Czech–Austrian scientific collaboration funding under project number 1999/17 and the research project IST-2001-34744.

Appendix

Here, we present an example of reporting the results using the minimum test output for a ray shooting algorithm for a set of scenes for two *RSAs*. The scenes were generated using the SPD package [8], where *X* for "sceneX" denotes the scaling factor when generating the scene. Table 1 reports the results for an *RSA* based on a kd-tree. Similarly, Table 2 reports the result for an *RSA* based on a uniform grid, where the required ratio of number of voxels and objects was set to one. For the sake of clarity and completeness, in both tables the first column refers to the name of the scene, the second column denotes the number of objects in the scene. (When introducing a new *RSA*, the description of the

Table 1
Example of reporting the results using the minimum test output in tabular form for scenes from the SPD package. Parameter N_{OBJ} is the number of objects in the scene. An *RSA* based on a kd-tree was used

Scene	N_{OBJ}	Minimum test output												
		Σ				Δ				Θ				
		N_G	N_E	N_{EE}	N_{ER}	r_{ITM}	$\tilde{N}_{TS}$	$\tilde{N}_{ETS}$	$\tilde{N}_{EETS}$	T_B	T_R	Θ_{APP}	Θ_{RAT}	Θ_{RUN}
balls4	7382	7891	7892	1484	17313	12.79	27.13	5.34	1.61	1.97	30.08	14.47	0.21	28.71
gears4	9345	27343	27344	940	53627	7.54	21.89	3.30	0.70	2.86	40.08	6.83	0.31	10.83
jacks4	5265	17859	17860	4883	25876	21.00	39.48	7.21	3.09	2.21	24.77	7.72	0.56	35.06
lattice12	8281	42868	42869	5203	49834	4.93	42.69	6.85	3.12	2.89	52.04	7.74	0.32	16.59
mount6	8196	13190	13191	5648	12239	6.57	20.72	3.71	1.69	1.41	24.23	14.20	0.20	21.69
rings7	8401	14943	14944	2523	35876	19.27	37.40	6.67	2.90	2.57	74.55	4.98	0.61	22.37
sombrero2	7938	13926	13927	7676	10234	6.05	18.88	3.51	2.39	1.28	5.34	11.19	0.18	20.22
teapot12	9264	23507	23508	6338	38225	12.01	28.97	5.50	3.33	2.54	19.01	16.71	0.21	27.20
tetra6	4096	2961	2962	1938	4096	10.47	14.81	2.78	2.30	0.46	3.77	16.80	0.20	29.13
tree11	8191	4369	4370	1747	11323	22.24	14.95	3.70	0.91	2.14	23.06	16.09	0.35	34.81
Average	7636	16886	16887	3838	25864	12.29	26.69	4.86	2.20	2.03	29.69	11.67	0.32	24.66

Table 2
Example of reporting results similarly to Table 1, but an *RSA* based on a uniform grid was used

Scene	N_{OBJ}	Minimum test output												
		Σ				Δ				Θ				
		N_G	N_E	N_{EE}	N_{ER}	r_{ITM}	$\tilde{N}_{TS}$	$\tilde{N}_{ETS}$	$\tilde{N}_{EETS}$	T_B	T_R	Θ_{APP}	Θ_{RAT}	Θ_{RUN}
balls4	7382	0	8648	6444	10542	607.08	4.94	4.94	1.14	0.25	238.63	14.47	0.67	327.90
gears4	9345	0	8976	6018	19682	43.60	6.20	6.20	1.70	0.48	58.85	6.80	0.58	19.13
jacks4	5265	0	5472	3110	18734	56.74	6.66	6.66	3.06	0.27	35.09	7.72	0.80	52.89
lattice12	8281	0	9261	0	21936	14.30	5.47	5.47	0.00	0.39	59.81	7.74	0.65	20.25
mount6	8196	0	8820	7204	21890	33.84	8.15	8.15	3.87	0.40	38.84	14.20	0.57	42.53
rings7	8401	0	8000	5635	28149	86.11	10.06	10.06	2.77	0.42	182.43	4.99	0.84	61.95
sombrero2	7938	0	7854	6154	21673	60.24	6.02	6.02	2.78	0.40	9.54	11.16	0.65	44.92
teapot12	9264	0	9251	7483	22771	112.69	13.90	13.90	6.99	0.43	38.70	16.71	0.70	72.70
tetra6	4096	0	4096	3104	12556	115.51	7.13	7.13	5.19	0.21	8.44	16.84	0.68	86.04
tree11	8191	0	8112	5400	11096	4583.10	5.38	5.38	1.13	0.25	1463.30	16.16	0.88	3214.08
Average	7636	0	7849	5055	18903	571.32	7.39	7.39	2.86	0.35	213.36	11.68	0.70	394.24

new *RSA* must precisely describe the setting used for the construction, here we just show how to use the minimum test output for comparison given any two *RSAs*.)

For testing we used a ray tracing algorithm as defined in the `Readme.txt` file in the SPD package distribution [8] (the number of primary rays cast is 513×513, depth of recursion is 4). All the experiments were conducted on an SGI O^2 running Irix 6.2, processor MIPS PRO R8000, 180 MHz, 128 MB RAM. The test program in the GOLEM rendering system [25] was compiled using MIPS PRO with -O2 optimization. The parameters for subset Θ of the minimum test output were obtained using a software profiler tool. (''Ideal pixie time'' was measured.)

Based on the results in the tables we can compare various aspects of two *RSAs* tested on average and also for a single scene. Concerning static properties, i.e., subset Σ of the minimum test output, the *RSA* based on a kd-tree used about 4 times more nodes ($N_G + N_E$) within the data structure on average than the *RSA* based on the uniform grid. Obviously, nearly half of the nodes for the kd-tree were elementary; for the uniform grid all the nodes were elementary. However, on average the ratio of empty elementary nodes to all elementary nodes (N_{EE}/N_E) for the *RSA* based on a kd-tree was 2.8-times smaller than for the *RSA* based on a uniform grid. The number of pointers to the objects (N_{ER}) was quite similar on average for both *RSAs*.

Concerning dynamic properties, i.e., subset Δ, the ratio of number of ray-object intersection tests to minimum number of ray-object intersection tests (r_{ITM}), was 46-times smaller on average for the *RSA* based on a kd-tree than for the *RSA* based on a uniform grid. The deficiency on performance of the *RSA* based

on a uniform grid is particularly apparent for sparsely occupied scenes ("tree11"). However, parameter r_{ITM} for the RSA based on a kd-tree also achieved lower values for all densely occupied scenes. The number of traversal steps per ray ($\tilde{N}_{TS}$) for the *RSA* based on a uniform grid was 4.4-times smaller than for the *RSA* based on a kd-tree on average. In the kd-tree, most of the traversed nodes were generic (81%), and nearly half of the traversed elementary nodes were empty (45%), on average. Obviously, in the uniform grid only elementary nodes were traversed, and nearly 39% of them were empty on average.

Concerning timing, i.e., subset Θ, we can compare the time for preprocessing directly (parameter T_B), since we used the same hardware, implementation, and compiler. The time consumed building up the kd-tree was on average about 6 times higher than the time for building up the uniform grid. The time portion devoted to the ray-object intersection tests in contrast to the time consumed by the *RSA* (Θ_{RAT}) was on average only 0.3 for the kd-tree, but still relatively high for the uniform grid (0.7). It means that in the kd-tree most of the time is devoted to traversing the data structure, in contrast to the uniform grid, where most of the time is needed for ray-object intersection tests. Concerning the parameter Θ_{RUN}, which is the main index of performance, we can see that the *RSA* based on a kd-tree performs significantly better (nearly 16-times faster on average) than the *RSA* based on a uniform grid for a given set of scenes. This is caused by degradation in the performance of uniform grids when used for sparsely occupied scenes as we have mentioned above (scene "tree11"). However, even for other scenes the performance of the *RSA* based on a kd-tree is typically twice as high than for the *RSA* based on a uniform grid. The closest difference of Θ_{RUN} is for the scene "lattice12", where the viewpoint (and thus origin of primary rays) is located inside the scene.

References

[1] de Berg M. Ray shooting, depth orders and hidden surface removal. Lecture Notes in Computer Science, vol. 703. New York: Springer, 1993.

[2] Szirmay-Kalos L, Márton G. Analysis and construction of worst-case optimal ray shooting algorithms. Computers and Graphics 1998;22(2–3):167–74.

[3] Arvo J, Kirk D. A survey of ray tracing acceleration techniques. In: Glassner A, editor. An introduction to ray tracing. New York: Academic Press, 1989. p. 201–62.

[4] Tobler RF, Wilkie A, Přikryl J. Advanced rendering toolkit, 2000. Available from http://www.cg.tuwien.ac.at/research/rendering/ART/.

[5] Bekaert P, Suykens F, Dutré P, Přikryl J. RenderPark—a photorealistic rendering tool, 2000. Available from http://www.cs.kuleuven.ac.be/~graphics/RENDERPARK/.

[6] Ward G. Radiance rendering package, 1995. Available from http://radsite.lbl.gov/radiance/HOME.html.

[7] Szirmay-Kalos L, Márton G. Worst-case versus average case complexity of ray-shooting. Computing 1998;61(2):103–31.

[8] Haines EA. A proposal for standard graphics environments. IEEE Computer Graphics and Applications 1987;7(11):3–5. Available from http://www.acm.org/pubs/tog/resources/SPD/overview.html.

[9] Smits B, Jensen HW. Global illumination test scenes. Technical Report UUCS-00-013, University of Utah, http://www.cs.utah.edu/~bes/papers/scenes/.

[10] Lext J, Assarsson U, Möller T. A benchmark for animated ray tracing. IEEE Computer Graphics and Applications 2001;21(2):22–31.

[11] Havran V, Přikryl J, Purgathofer W. Statistical comparison of ray-shooting efficiency schemes. Technical Report TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, Favoritenstrasse 9/186, A-1040 Vienna, Austria, May 2000. Available from ftp://ftp.cg.tuwien.ac.at/pub/TR/00/TR-186-2-00-14Paper.ps.gz. Submitted.

[12] Formella A, Gill C. Ray tracing: a quantitative analysis and a new practical algorithm. The Visual Computer 1995;11(9):465–76, ISSN 0178-2789.

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

[14] McNeill MDJ, Shah BC, Hebert M-P, Lister PF, Grimsdale RL. Performance of space subdivision techniques in ray tracing. Computer Graphics Forum 1992;11(4):213–20.

[15] Havran V, Žára J. Evaluation of BSP properties for ray-tracing. In: Proceedings of SCCG'97 (Spring Conference on Computer Graphics), Budmerice, June 1997. p. 155–62.

[16] Agarwal PK, Erickson J. Geometric range searching and its relatives. Technical Report CS-1997-11, Department of Computer Science, Duke University, 1997.

[17] Aho AV, Hopcroft JE, Ullman JD. The design and analysis of computer algorithms. Reading, MA: Addison-Wesley, 1974.

[18] Arvo J, Kirk D. Fast ray tracing by ray classification. In: Stone MC, editor. SIGGRAPH '87 Proceedings, vol. 21, July 1987; p. 55–64.

[19] Cleary JG, Wyvill G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. The Visual Computer 1988;4(2):65–83.

[20] Sequin CH, Smyrl EK. Parameterized ray tracing. In: Lane J, editor. SIGGRAPH '89 Proceedings, vol. 23, July 1989; p. 307–14.

[21] Day RA. How to write & publish a scientific paper. New York: Academic Press, 1997.

[22] Held M. Erit—a collection of efficient and reliable intersection tests. Journal of Graphics Tools 1997;2(4):25–44.

[23] 3D Object Intersection Home Page. Maintained by Foscari P, and Haines EA, 2000, http://www.realtimerendering.com/int/.

[24] Wald I, Slusallek P, Benthin C, Wagner M. Interactive rendering with coherent ray tracing. Eurographics '2001 Conference, 2001. p. 153–64.

[25] Havran V. Golem rendering system, 2000. HOME page at http://www.cgg.cvut.cz/GOLEM.