Comparison Methodology for Ray Shooting Algorithms

Vlastimil Havran[†] **and Werner Purgathofer**[‡]

[†] Department of Computer Science, Czech Technical University
Karlovo nám. 13, CZ-12135 Prague 2, Czech Republic
e-mail: VHavran@seznam.cz
[‡] Institute of Computer Graphics and Algorithms, Vienna University of Technology
Favoritenstrasse 9/186, A-1040 Wien, Austria
e-mail: purgathofer@cg.tuwien.ac.at

Abstract

In this paper we deal with a methodology for comparing various ray shooting algorithms for a set of experiments performed on a set of scenes. We develop a computation model for ray shooting algorithms, which allows us to map of any particular ray shooting algorithm to the computation model. Further, we develop a performance model for ray shooting algorithms, which establishes the correspondence between the computation model and the execution time of the ray shooting algorithm for a sequence of ray shooting queries. Based on the computation and performance models, we propose a set of parameters describing the use of a ray shooting algorithm in applications. This 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 the hardware and the implementation issues. Under certain conditions, the proposed comparison methodology enables to perform cross-comparison of published research work without reimplementing other ray shooting algorithms.

1. Introduction

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

Ray shooting is a simply defined task: find out the first object intersected by a given ray for a given set of objects, if such an object exists. In spite of this simple formulation, it is not trivial to implement an efficient and fast ray shooting algorithm (abbreviated to *RSA*). The problem of finding an ultimately efficient *RSA* still remains open. Both computational geometers and computer graphics researchers have tried to develop a fast *RSA* with varying success. Computational geometers aimed their efforts at improving the worst-case time complexity. Unfortunately, the space and preprocessing time complexity of these methods is unacceptable for real implementations in rendering frameworks[6]. In computer graphics different heuristic *RSAs* improving the average-case time complexity were investigated. Even if the worst-case complexity of these heuristics algorithms is unfavourable, the good average-case complexity is the reason why these heuristic *RSAs* are commonly used in rendering packages[21, 5, 22].

It is also possible to use a brute force method for solving the problem. A naïve *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. Szirmay-Kalos and Márton[18] proved that the lower bound on the worst-case complexity of ray shooting is $\Omega(\log N)$. Some heuristics algorithms exhibit worst-case complexity $O(N)$ and average-case

complexity $O(1)$ for scenes with uniformly distributed objects[19]. These complexities remain for non-random scenes as well, but unfortunately, the unknown multiplicative factor hidden behind the scenes and random object distribution in the scene for average case analysis make theoretical complexity definition unusable to decide which *RSA* should be used in practice. This is the reason why the performance of *RSAs* is normally compared on a set of some test scenes. In order to test the performance of ray tracing algorithms Haines introduced *Standard Procedural Database* package (abbreviated to SPD[11]), which defines a set of scenes and a description of ray tracing algorithm.

Since the *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 type of scenes, and particular features of an *RSA*. In spite of two decades of research on *RSA* in the computer graphics community, it is not yet clear if 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 *RSA* 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. 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 sometimes rather impossible. A fair comparison of different *RSAs* has been possible only when they are implemented and tested within a uniform software framework, such as[12], and when the testing is performed on the same hardware. Only a few papers devoted to the comparison of various *RSAs* [13, 16, 9, 10] have been published until now, which uses as means for comparison the execution time.

In this paper we will try to decrease the gap in understanding of the functionality of various *RSAs* by finding out their commonalities. The commonalities found in all *RSAs* allow us to describe the *RSA* computation model in a general way that allows us to map a particular *RSA* to this computation model. Further, we will describe a performance model that establishes the connection between the execution time of the application and the different algorithmic operations that are the subject of the computation model. Then we will define two procedures for an "ideal *RSA*" that allow us to compute the answers to ray shooting queries in constant time. The "ideal *RSA*" results in the smallest possible time that can ever be achieved using a certain hardware and a set of ray-object intersection routines. We use the time consumed by the "ideal *RSA*" as a reference time 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 *RSA* is based, ray-object intersection tests, and the remaining time consumed by the application. The design of the computation model and performance model allow us to define a set of thirteen parameters referred to as the minimum testing output that should be reported for one experiment, given a scene, a particular *RSA*, and a sequence of ray shooting queries. Further, we describe how to get the minimum testing output. The definition of the two models and the minimum testing 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[14] for the kd-tree and further elaborated in[13]. The comparison methodology presented here is a generalised and extended version for any *RSA* virtually independently of the hardware issues.

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

2. *RSA* Computation Model

In this section we introduce the *RSA computation model*. We will show that any *RSA* currently known or developed in the future can be mapped to this computation model. The computation model is based on the definition of algorithmic operations in an *RSA*. These operations must

always be performed due to the nature of the ray shooting problem. We then use the computation 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[1], which implies that some data structure is built to answer the specific query. The definition of the ray shooting problem implies that every *RSA* contains somewhere 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 all algorithms[2]): the data structure (further abbreviated to *DS*) containing at least pointers to the scene objects and the ray traversal algorithm working over *DS*. The lifetime of an *RSA* is composed of two phases, the first one is called *preprocessing phase* and it 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.

More theoretically, an *RSA* can be described a special case of a general *RAM* model[2, 20], where any memory cell can be accessed in constant time or through a series of pointers. A *DS* is composed of some data entries, here referred to as *nodes*, which contain some data. It usually involves the pointers to the scene objects, the pointers to the other nodes of *DS*, the size of the cells etc. Nodes of a *DS* can be divided into two groups: *elementary nodes* contain only pointers to objects (and, if *RSA* requires it, to some other data), whereas *generic nodes* do not refer to any objects, but rather 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*.

When answering a ray shooting query in a particular *RSA*, the computation proceeds as follows. Given a ray R, a ray traversal algorithm begins at a special starting node of a *DS* and performs a sequence of the following operations:

TRAVERSAL STEP:	visit a new node of the *DS* using a pointer from a previously visited node,
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 the intersection with the ray R,

finally finding the closest intersected object if such an object exists. There are two possibilities: a *DS* is or is not changed by a ray traversal algorithm. 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. These *RSAs* are referred to as *RSAs* 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[3]. The operations "NEW NODE" and "DELETE NODE" can be used within the preprocessing phase to build up some initial *DS*, however, this *DS* is modified during the execution phase. These *RSAs* are referred to as *RSAs* based on a *dynamic data structure*.

Since every *RSA* can be mapped to this general *RSA* computation model, 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. The sequence of ray shooting queries induced by the application for an input scene S is associated with a *testing procedure* (the symbol for the testing procedure is *TP*). The testing procedure is an algorithm in the application that generates a sequence of ray shooting queries to be answered by a particular *RSA*. 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.

We propose to organise 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 independent:

- *RSA* parameters related to static properties of data structure *DS*:
  - If an *RSA* is based on a static data structure, they 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, they 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 DS,
$N_E[-]$ – maximum number of elementary nodes in DS,
$N_{EE}[-]$ – maximum number of empty elementary nodes in DS ($N_{EE} < N_E$),
$N_{ER}[-]$ – maximum number of the pointers to objects in all the elementary nodes of DS ($N_{ER} \geq N$).

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

  $r_{ITM}[-]$ – ratio of ray-object intersection tests performed to minimum number of intersection tests ($r_{ITM} \geq 1.0$, assuming at least one object is hit given S and TP),
  $\tilde{N}_{TS}[-]$ – average number of all DS nodes accessed per ray ($\tilde{N}_{TS} \geq 1.0$),
  $\tilde{N}_{ETS}[-]$ – average number of elementary DS nodes accessed per ray ($\tilde{N}_{ETS} \leq \tilde{N}_{TS}$),
  $\tilde{N}_{EETS}[-]$ – average number of empty elementary DS nodes accessed per ray ($\tilde{N}_{EETS} \leq \tilde{N}_{ETS}$).

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

  $T_B[s]$ – time consumed to build DS for the *RSA* in the preprocessing phase (depends on S and hardware/implementation),
  $T_R[s]$ – time consumed to perform given TP in the application, which involves the execution phase of the *RSA*.

We consider the parameters in the first two subsets as the minimum hardware/implementation independent parameters to be reported. Principally, it is possible to extend the set of parameters by others (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 *RSA* via the computation 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 optimisation switches used for compilation etc. For this reason, all 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 a solution to the problem in more detail below.

3. *RSA* Performance Model

The *RSA* computation model enables us to count the number of basic algorithmic operations performed on average in an *RSA*. The *RSA* computation model does not define any cost of these operations in terms of execution 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 execution time 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*, which separates the cost of a ray traversal algorithm and the cost of ray-object intersection tests. The concept of the performance model for an *RSA* was first introduced by Cleary and Wyvill[7] in the context of uniform grid analysis. We present here a more general performance model for any *RSA* that is derived from the *RSA* computation model described above. The *RSA* performance model is based on the decomposition of the total execution time T_R of the application that uses an *RSA* into three parts:

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

We bind the time-dependent and independent characteristics by means of cost consumed by specific algorithmic operations. Then we can express T_R as:

$$T_R = (r_{ITM}.r_{SI}.\tilde{C}_{IT} + \tilde{N}_{TS}.\tilde{C}_{TS}).N_{rays} + T_{app}, \quad (1)$$

where r_{SI} is the ratio of the number of rays hitting the objects to the number of all rays ($r_{SI} < 1.0$), thus the average number of ray-object intersection tests per ray is $\tilde{N}_{IT} = r_{ITM}.r_{SI}$. 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 DS, N_{rays} is the total number of rays induced by a testing procedure TP, and $T_{app}[s]$ is the remaining time of the application. The time T_{app} covers another computation 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 hit ratio of ray-object intersection tests to all intersection tests:

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

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 execution 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 a given ray computes any ray shooting query in $O(1)$ time independently of whether an intersected object exists or not. The multiplicative factor hidden behind the O-notation is very small.

Since Szirmay-Kalos and Márton[19] proved that any *RSA* works at least at time $\Omega(logN)$ in the worst-case, then we can ask if the definition of an "ideal *RSA*" makes sense. Inspired by the idea of *Parametrized Ray Tracing*[17], 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 is 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. In the preprocessing phase each object is assigned an identification tag ID (integer) in the range $\{0, N-1\}$. Then we construct the array A_T where objects are addressed directly using IDs of objects in $O(1)$ time.

In the first application run we use some traditional *RSA*. The results obtained by the *RSA* for the sequence of input ray shooting queries generated by TP are saved linearly to a temporary array A_S using the object's IDs. When no object is intersected, the array entry is set to a special ID value ($ID_{spec} = -1$). Since the number of ray shooting queries can be high, it may be necessary to save the results of ray shooting queries to external memory. The procedure that must be used in the first application run and at the interface between the application and the traditional *RSA* is outlined in the pseudocode, Algorithm 1.

Algorithm 1 The first phase of "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 number of entries in 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 RayShoot(ray R): object
{compute the result of the i-th ray shooting query by some other specific RSA.}
Compute the result for R using some specific RSA
Object O ← the result of the specific 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
RayShoot ← O
{Final phase of RSA }
Possibly save A_S to external memory
```

In the second (repetitive) application run, in-

stead 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 results in the same testing procedure TP and uses the same scene S. If we get the object's valid ID, we get the address of the object through the array A_T and compute the ray-object intersection point exactly by one ray-object intersection test. This computation is required to get 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 phase of "ideal RSA" that reads the results for ray shooting queries.

```
{Preprocessing phase}
Assign each object its unique ID in the range
{0, N − 1}. These IDs correspond to the first
phase of "ideal RSA".
Allocate the array A_S to store IDs 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 RayShoot(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 t for R and O
else
  Object O ← "no object"
end if
RayShoot ← object O
```

For the repetitive run(s) of the "ideal RSA" the time T_R becomes the minimum possible application execution 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*:

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

If external memory is used to save the array A_S, we should avoid the time consumed to transfer the data from this external memory to internal memory to minimise the repetitive application execution time T_R. Practically, 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_R^{MIN}. From the implementation point of view, the "ideal RSA" is fairly easy to implement in the application.

5. Minimum Testing Output

The results of experiments published in the papers introducing new $RSAs$ were often restricted to only T_B and T_R and some other parameters. 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 computation 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 a *Standard Procedural Database*[11] was introduced. This database 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 $RSAs$. 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. Such a researcher's behaviour is a direct violation of research etiquette, since the nature of science is that every research paper should describe new techniques and experiments that will be *reproducible* and *verifiable* by all following researchers[8]. Therefore, whenever possible, qualitative properties of algorithms should always 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 rather incorrect. The first reason is that T_R also includes T_{app}, which is constant. If we want to compare the performance 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 the ray shooting only only. Obtaining the value of T_{app} can be

difficult, as it usually requires profiling of the application by some software tool. We propose a way avoiding the use of a profiler in the section below. The value of T_R can be used correctly only for ranking of *RSAs*, but it cannot be used to express how much an *RSA* is faster than another *RSA*.

The SPD package[11] also recommends that some time-independent characteristics should be reported: N_{rays}, $\tilde{N}_{IT}^{succ}.N_{rays}$, $\tilde{N}_{TS}.N_{rays}$. We follow this approach by extending this set of hardware/implementation 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 testing output*. This consists of three subsets as already presented: *RSA* parameters that relates to static properties of *DS*, *RSA* parameters that relates to dynamic use of *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 normalise the time portions devoted to the particular phases to the ideal ray shooting time T_{RSA}^{MIN} to 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 testing output should allow us to compare the performance of various *RSAs* independently of hardware and implementation.

We define the minimum testing output for an *RSA* as:

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 *DS*, which also depend on the input scene S and testing procedure *TP*:
$\Delta = \{r_{ITM}, \tilde{N}_{TS}, \tilde{N}_{ETS}, \tilde{N}_{EETS}\}$,

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

$$\Theta = \{\Theta_A, \Theta_{IT}, \Theta_{TS}, T_B, T_R\} = \{\frac{T_{app}}{T_{RSA}^{MIN}}, \frac{N_{rays}.(\tilde{N}_{IT}^{succ}.\tilde{C}_{IT}^{succ} + \tilde{N}_{IT}^{fail}.\tilde{C}_{IT}^{fail})}{T_{RSA}^{MIN}}, \frac{N_{rays}.\tilde{N}_{TS}.\tilde{C}_{TS}}{T_{RSA}^{MIN}}, T_B, T_R\} \quad (5)$$

The parameter Θ_A 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*, however, suitable for other reasons. The parameter Θ_{IT} is the ratio of time required for computing the ray-object intersection tests to T_{RSA}^{MIN}. Similarly, the parameter Θ_{TS} gives the ratio of time consumed by traversing the *DS* 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:

$$T_{RSA}^{MIN} = \frac{T_R}{\Theta_A + \Theta_{IT} + \Theta_{TS}} \quad (6)$$

6. Measuring the Minimum Testing Output

The minimum testing output allow us to make a fair comparison of various *RSAs*. We pay for it by additional effort needed to get this set of thirteen parameters for one experiment. The counters to get 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 error of 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 considered as an independent input quantity, it is often the case that the number of rays generated is connected with the use of *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 naïve *RSA* even if the naïve *RSA* is inefficient.

To obtain subset Θ we need the total execution time T_R to be decomposed into the 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}. There are two

ways to obtain subset Θ, which are two profiling methods described below.

6.1. Software Tool Profiling

One way to get 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 the times consumed within particular software functional units, such as functions, procedures, or even the lines of a source code. Then we can sum the time devoted to ray-object intersection test routines, the time consumed by traversing the nodes of a *DS*, and the remaining application time.

Software tool profiling should be preferred for getting Θ, since it provides precise values. However, under certain conditions this is not possible, for one of the following reasons: the profiler is not available, the profiler does not work correctly, the profiler cannot determine the time portions of the required *RSA* parts within a given implementation, the profiler needs some compiler switches to be used, which influences T_R (a debugging switch is usually required, and this can increase the application time considerably) and the different time portions of T_R. Therefore we propose an alternative to obtain subset Θ without using a software profiler tool below.

6.2. Multiple Run Profiling

This profiling method involves running the application several times and computing the unknown variables in Eq. 2 from linear equations. Eq. 2 contains four unknown variables that express the costs of distinct algorithmic operations in some *RSA* application: $\tilde{C}_{IT}^{succ}$, $\tilde{C}_{IT}^{fail}$, $\tilde{C}_{TS}$, and T_{app},

In order to obtain the four unknown variables we need four different application runs. These have to use the same sequence of ray shooting queries, but they have to result in different total execution times. For this purpose we utilise the concept of the "ideal *RSA*", and modify the ray-object intersection tests to be performed K-times. The first used equation comes from the common application run, described by Eq. 2. The second used equation is for the application run, when the ray-object intersection test is performed K-times, resulting in the execution time:

$$T_R(K)[s] = [K.(\tilde{N}_{IT}^{succ}.\tilde{C}_{IT}^{succ} + \tilde{N}_{IT}^{fail}.\tilde{C}_{IT}^{fail}) + \tilde{N}_{TS}.\tilde{C}_{TS}].N_{rays} + T_{app} \quad (7)$$

The third used equation is the time of the "ideal *RSA*", Eq. 3. The fourth used equation is for the case when the ray-object intersection test in the "ideal *RSA*" is performed K-times, resulting in the execution time:

$$T_R^{MIN}(K)[s] = K.\tilde{C}_{IT}^{succ}.\tilde{N}_{rays}.r_{SI} + T_{app} \quad (8)$$

Assuming $\tilde{C}_{IT}^{succ}$, $\tilde{C}_{IT}^{fail}$, $\tilde{C}_{TS}$, and T_{app} are of the same value in these four application runs, we can compute these unknown variables by solving a system of linear equations. From Eqs. 3 and 8, we get $\tilde{C}_{IT}^{succ}$ and T_{app} as follows:

$$\tilde{C}_{IT}^{succ} = \frac{T_R^{MIN}(K) - T_R^{MIN}}{N_{rays}.(K-1).r_{SI}} \quad (9)$$

$$T_{app} = T_R^{MIN} - N_{rays}.r_{SI}.\tilde{C}_{IT}^{succ} \quad (10)$$

From Eqs. 2 and 7 we derive the $\tilde{C}_{IT}^{fail}$ and $\tilde{C}_{TS}$:

$$\tilde{C}_{IT}^{fail} = [\frac{T_R(K)-T_R}{N_{rays}.(K-1)} - \tilde{C}_{IT}^{succ}.\tilde{N}_{IT}^{succ}].\frac{1}{\tilde{N}_{IT}^{succ}} \quad (11)$$

$$\tilde{C}_{TS} = [\frac{T_R(K) - T_R}{N_{rays}} - \tilde{C}_{IT}^{succ}.\tilde{N}_{IT}^{succ} - \tilde{C}_{IT}^{fail}.\tilde{N}_{IT}^{fail}].\frac{1}{\tilde{N}_{TS}} \quad (12)$$

We call the profiling method based on the four equations *multiple run profiling*. It is oriented only to the software applications that use *RSA*.

6.2.1. Properties

Multiple run profiling has one big advantage, it does not require any profiling. However, it suffers from several disadvantages. First, it requires the multiple ray-object intersection test to be implemented in routines for all the object types in the application. Second, the time of the multiple ray-object intersection test is affected by the cache behaviour of the processor used during experiments. Even if the ray-object intersection test is performed K-times, instead of being K-times slower it is only K' times slower, where $0 < K' < K$. Further, we have found out during the testing of an "ideal *RSA*" that caching and branch prediction within the processor also influences the time of ray-object intersection tests within the "ideal *RSA*". Since in this case the ray-object intersection test is always positive (Eqs. 3 and 8), the branches are always well predicted, resulting in lower cost $\tilde{C}_{IT}^{succ}$. Our experiments had best matching with the software tool profiling using $K = 2$. Third, the application must be run at least

five times over the same input scene. In addition to the four measurement runs described above, one run is required to save the results of *RSA* into the visibility array A_S for the "ideal *RSA*" not to influence T_R in the measurement run corresponding to Eq. 2.

The second way is to get directly some other estimates of $\tilde{C}_{TS}$, $\tilde{C}_{IT}^{succ}$, $\tilde{C}_{IT}^{fail}$, and T_{APP}. Some of these may be known or well estimated for given hardware/implementation independent of TP and S for some previous runs of the application. Provided $T_{APP} \gg T_{RSA}^{MIN}$, we can also assume $T_{APP} \doteq T_R^{MIN}$. Third, we can obtain the estimate for $\tilde{C}_{IT}^{succ}$ and $\tilde{C}_{IT}^{fail}$ when we use the same *RSA* with a different setting used for the construction of data structure underlying the *RSA*. For example, if a kd-tree is used, we can set the maximum depth allowed to various constants and then we get a set of equations of type 2, which allows us to compute $\tilde{C}_{TS}$.

Although multiple run profiling has several disadvantages, it remains the only known way when software tool profiling is not possible for some reason. Below, we improve this method using some correction parameters.

6.2.2. Corrected Measuring of Subset Θ

To obtain more precise profiling results we can modify the equations of application runs to model the behaviour of caching and branch prediction to some extent. We propose to use three correction parameters in this modified multiple run profiling. They express the time between the operation that is expected to be cached and the time of uncached operation: r_{corr}^{rep}, r_{corr}^{hit}, r_{corr}^{fail}, all of them in the range $(0.0, 1.0)$. First, we correct $\tilde{C}_{IT}$ provided that the ray-object intersection test always succeeds:

$$\tilde{C}'_{IT} = \tilde{C}_{IT}.r_{corr}^{rep} \tag{13}$$

Second, we correct $\tilde{C}_{IT}$ of the repetitive successful ray-object intersection test:

$$\tilde{C}'^{succ}_{IT}(K) = \tilde{C}_{IT}.[1 + (K-1).r_{corr}^{hit}] \tag{14}$$

Third, we correct $\tilde{C}_{IT}$ of the repetitive failed ray-object intersection test:

$$\tilde{C}'^{fail}_{IT}(K) = \tilde{C}_{IT}.[1 + (K-1).r_{corr}^{fail}] \tag{15}$$

Then we can express the corrected three equations as follows:

$$T_R^{MIN} = \tilde{C}_{IT}^{succ}.N_{rays}.r_{SI}.r_{corr}^{rep} + T_{app}, \tag{16}$$

$$\begin{aligned} T_R(K)[s] \;=\; & \{[\tilde{N}_{IT}^{succ}.\tilde{C}_{IT}^{succ}.(1 + (K-1).r_{corr}^{hit})] + \\ & [\tilde{N}_{IT}^{fail}.\tilde{C}_{IT}^{fail}.(1 + (K-1).r_{corr}^{fail})] + \\ & \tilde{N}_{TS}.\tilde{C}_{TS}\}.N_{rays} + T_{app}, \end{aligned} \tag{17}$$

and

$$\begin{aligned} T_R^{MIN}(K)[s] \;=\; & \tilde{C}_{IT}^{succ}.r_{corr}^{rep}.(1 + (K-1).r_{corr}^{hit}). \\ & N_{rays}.\tilde{n}_{IT} + T_{app}. \end{aligned} \tag{18}$$

Similarly to Eqs. 9–12 we can derive the formulas to obtain $\tilde{C}_{TS}$, $\tilde{C}_{IT}^{succ}$, $\tilde{C}_{IT}^{fail}$, and T_{APP}. Corrected measuring is more precise, but it requires us to set the correction parameters. These can be estimated by using a software profiler when compiling without using optimisation switches, or for a different setting of K. Another way is to use various ray traversal algorithms, since the correct setting of the correction parameters Θ_{IT} remains the same, because the number of ray-objects does not differ and Θ_{TS} depends on the ray traversal algorithm used.

7. Comparison Methodology

The establishment of the subsets Σ, Δ, and Θ of the minimum testing output enables us to compare different features of *RSAs*. For the use of an *RSA* in an application there are several different features for us to distinguish:

S: the complexity of input scene S is important, for example, some *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 Σ, Δ, and Θ.

RSA: the idea behind *RSA* has a major impact on performance. *RSA* influences Σ, Δ, and Θ.

TP: the testing procedure is specific to the application used, and the use of *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 optimisation switch -O2 of the C++ compiler in the UNIX operating system can decrease the execution time by half.

IMPL: implementation – the actual coding of the algorithm also has a great impact on

performance, depending on the programmer's experience, etc. Various implementations of the same ideas can exhibit significant differences in performance. It influences only subset Θ. When the *RSA* is (re)implemented correctly, the parameters in subsets Σ and Δ are not influenced.

We note that *HW*, *COMP*, and *IMPL* can be intertwined to some extent, since a certain implementation can better fit to 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 fix as many other possible dimensions as possible, in this case S, *TP*, *HW*, *COMP*, and *IMPL*. As the minimum requirement, we can require the same set of scenes and the testing procedure within the application to 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 such cases, due to the generality of the underlying *RSA* computation 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):

- memory complexity we compare $(N_G^1 + N_E^1)$ with $(N_G^2 + N_E^2)$. To a constant factor given by implementation of a particular *RSA*, it expresses the different memory requirements.
- use of hierarchy, we compare N_G^1/N_E^1 and N_G^2/N_E^2.
- use of empty space, we compare N_{EE}^1/N_E^1 and N_{EE}^2/N_E^2
- time complexity, we have several choices depending on the conditions for comparison:
  - ${T_R}^1 - T_{app}^1$ with ${T_R}^2 - T_{app}^2$ for performance ratio, ${T_R}^1$ with ${T_R}^2$ for ranking only – time can be used directly for comparison, when *HW*, *COMP*, and *IMPL* attributes are the same or very similar. The conditions must be stated explicitly.
  - Θ_{IT}^1 with Θ_{IT}^2 – concerns the portion of time for ray-object intersection tests. It can be used even when any of *HW*, *COMP*, and *IMPL* attributes differ.
  - Θ_{TS}^1 with Θ_{TS}^2 – concerns the portion of traversal time. It can be used even when any of *HW*, *COMP*, and *IMPL* attributes differ.
  - $\Theta_{IT}^1 + \Theta_{TS}^1$ with $\Theta_{IT}^2 + \Theta_{TS}^2$ – concerns the time required for ray shooting in the application related to ideal ray shooting time. Assuming that the implementation of ray-object intersection tests is practically the same, this enables a really fair comparison independent of *HW*, *COMP*, and *IMPL* attributes. The sum $\Theta_{IT} + \Theta_{TS}$ 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 higher performance. Unlike comparing ${T_R}^1 - T_{app}^1$ with ${T_R}^2 - T_{app}^2$, it enables us to compare various different *RSAs* virtually independently of *HW*, *IMPL*, and *COMP*.
  - r_{ITM}^1 with r_{ITM}^2 – an efficient *RSA* should have a ratio of ray-object intersection tests performed to the minimum number of intersection tests, as close to 1.0 as possible.
  - $\tilde{N}_{TS}^1$ with $\tilde{N}_{TS}^2$ – an efficient *RSA* has the number of traversal steps per ray as small as possible.
  - $\tilde{N}_{ETS}^1/\tilde{N}_{ETS}^1$ or $\tilde{N}_{EETS}^1/\tilde{N}_{EETS}^2$ – this shows us the utilisation of empty space within the execution phase. Empty space 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* computation model described in Section 2. Second, we have to measure the minimum testing output for each *RSA* and a set of scenes (the scenes should be publicly available, SPD scenes are suitable). The testing procedure used within the application has to be the same for one scene and any *RSA* and must be well described. This 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 testing output for each scene and each experiment considered as a research work must be fully published. For example, when introducing a new *RSA*, then for SPD scenes it is required a table with 10 rows (scenes) and with at least 13 columns (minimum testing output).

8. Discussion

The proposed minimum testing output organised into three subsets has a total of thirteen parameters, which can be considered a high number. Nonetheless, we consider this set as the minimum output that shows different features of an *RSA*,

since it is based on the general computation model that fits any *RSA*. The minimum testing output contains both hardware/implementation independent and hardware/implementation dependent characteristics that allow us to make mutual comparison of various *RSAs* under certain conditions. The disadvantage of this comparison methodology is the underlying assumption that the costs of the ray-object intersection tests are of the equal efficiency for various shapes of objects on different implementations. Fortunately, the ray-object intersection tests for objects' shapes in the SPD package are more or less standardised[4, 15]. 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 if it is possible to manipulate the minimum testing output by the 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 *RSA* itself is correct. If less efficient or more efficient ray-object intersection tests are applied, then the parameters Θ_A, Θ_{TS}, and T_R are influenced. However, it is virtually impossible to influence the parameter Θ_{IT}.

9. Conclusion

In this paper we have shown the concept that is common for all *RSAs*, *i.e.*, an *RSA* computation model and performance model. We have described the "ideal *RSA*" that provides us with the reference value for comparing two *RSAs*. Further, we have 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 general. The comparison methodology presented here enables us to compare various *RSAs*, assuming that the same application uses the same testing procedure for the same input scene. The construction of an "ideal *RSA*" and thus the measurement of T_{RSA}^{MIN} also shows us the time in the best possible and ideal case. The minimum application time T_R^{MIN} expresses the minimum time of a particular application that uses an *RSA* for a given scene and testing procedure.

A byproduct of this development is that we can measure how far we are from the minimum application execution time T_R^{MIN} ever achievable, 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.

Acknowledgements

The authors would like to thank Jan Přikryl and Jiří Bittner for comments on previous version of the paper. This work has been supported by the joint Czech-Austrian scientific collaboration funding under project number 1999/17.

Appendix

Here, we present an example of reporting the results using the minimum testing output for a ray shooting algorithm for a set of scenes. The scenes were generated using the SPD package [11], 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, which were obtained using multiple run profiling. The first column of the table refers to the name of the scene, the second column denotes the number of objects.

For testing we used a ray tracing algorithm as defined in `Readme.txt` file in the SPD package distribution[11] (the number of primary rays cast is 513×513, depth of recursion 4). All the experiments were conducted on a PC running Linux, kernel version 2.2.12-20, processor Intel Pentium II, 466 MHz, 128 MB RAM. The test program in the GOLEM rendering system[12] was compiled using egcs-1.1.2 with -O2 optimisation.

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. Tech. Report CS-1997-11, Department of Computer Science, Duke University, 1997.
2. A. Aho, J. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
3. J. Arvo and D. Kirk. Fast ray tracing by ray classification. In M. C. Stone, editor, *(SIGGRAPH '87 Proceedings)*, volume 21, pages 55–64, July 1987.
4. J. Arvo and D. Kirk. *A survey of ray tracing acceleration techniques*, pages 201–262. Academic Press, 1989.
5. P. Bekaert, F. Suykens, P. Dutré, and J. Přikryl. RenderPark - a photorealistic rendering tool. Available from `http://www.cs.kuleuven.ac.be/~graphics/RENDERPARK/`.

Scene	Σ						Δ			Θ				
	N_{OBJ}	N_G	N_E	N_{EE}	N_{ER}	r_{ITM}	$\tilde{N}_{TS}$	$\tilde{N}_{ETS}$	$\tilde{N}_{EETS}$	Θ_A	Θ_{IT}	Θ_{TS}	T_B	T_R
balls4	7382	7892	7893	1479	17323	12.79	27.13	5.34	1.61	5.67	2.83	8.00	1.22	14.69
gears4	9345	27471	27472	964	53681	7.55	21.85	3.29	0.68	5.70	2.48	5.97	1.67	20.10
jacks4	5265	17878	17879	4883	25902	21.04	39.51	7.20	3.08	2.33	4.46	7.40	1.13	13.05
lattice12	8281	42904	42905	5197	49876	4.94	42.78	6.85	3.12	4.80	3.17	5.82	1.58	27.03
mount6	8196	13191	13192	5649	12237	6.57	20.73	3.71	1.69	8.19	4.17	6.51	0.93	11.89
rings7	8401	14951	14952	2527	35894	19.32	37.40	6.67	2.90	3.06	7.44	6.50	1.45	40.29
sombrero2	7938	13927	13928	7677	10234	6.05	18.88	3.51	2.39	4.17	1.18	5.21	0.87	2.43
teapot12	9264	23502	23503	6337	38220	12.01	28.97	5.50	3.33	9.33	3.18	10.97	1.50	9.16
tetra6	4096	2971	2972	1948	4096	10.47	14.83	2.79	2.31	29.00	6.37	22.30	0.31	1.73
tree11	8191	4369	4370	1743	11327	22.24	14.94	3.70	0.91	31.38	41.02	9.98	1.38	10.71

Table 1: *Example of reporting the results using the minimum testing output in tabular form for scenes from the SPD package. Parameter N_{OBJ} is the number of objects in the scene.*

6. M. D. Berg. Ray shooting, depth orders and hidden surface removal. In *Lecture Notes in Computer Science*, volume 703. Springer Verlag, New York, 1993.

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

8. R. Day. *How to write & Publish a Scientific Paper.* Academic Press, 1997.

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

10. A. Formella and C. Gill. Ray tracing: a quantitative analysis and a new practical algorithm. *The Visual Computer*, 11(9):465–476, 1995. ISSN 0178-2789.

11. E. A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, Nov. 1987. Available from `http://www.acm.org/pubs/tog/resources/SPD/overview.html`.

12. V. Havran. Golem rendering system. HOME page at `http://www.cgg.cvut.cz/GOLEM`.

13. V. Havran, J. Přikryl, and W. Purgathofer. Statistical comparison of ray-shooting efficiency schemes. Technical Report TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, Karlsplatz 13/186/2, A-1040 Vienna, Austria, May 2000. human contact: technical-report@cg.tuwien.ac.at.

14. V. Havran and J. Žára. Evaluation of bsp properties for ray–tracing. In *Proceedings of 12th Spring Conference on Computer Graphics*, pages 155–162, Budmerice, June 1997.

15. M. Held. Erit–a collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, Dec. 1997.

16. M. D. J. McNeill, B. C. Shah, M.-P. Hebert, P. F. Lister, and R. L. Grimsdale. Performance of space subdivision techniques in ray tracing. *Computer Graphics Forum*, 11(4):213–220, Oct. 1992.

17. C. H. Sequin and E. K. Smyrl. Parameterized ray tracing. In J. Lane, editor, *SIGGRAPH '89 Proceedings*, volume 23, pages 307–314, July 1989.

18. L. Szirmay-Kalos and G. Marton. On the limitations of worst–case optimal ray shooting algorithms. In *Winter School of Computer Graphics 1997*, pages 562–571, Feb. 1997. held at University of West Bohemia, Plzen, Czech Republic, 14-18 February 1997.

19. L. Szirmay-Kalos and G. Márton. Worst-case versus average case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998.

20. R. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1987.

21. R. F. Tobler, A. Wilkie, and J. Přikryl. Advanced Rendering Toolkit. Available from `http://www.cg.tuwien.ac.at/research/rendering/ART/`.

22. G. Ward. Radiance rendering package. Available from `http://radsite.lbl.gov/radiance/HOME.html`.