Statistical Comparison of Ray-Shooting Efficiency Schemes

Vlastimil Havran†, Jan Přikryl‡, and Werner Purgathofer*

† Department of Computer Science, Czech Technical University
Karlovo nám. 13, CZ-12135 Praha 2, Czech Republic
e-mail: VHavran@seznam.cz
‡ VrVis Center for Virtual Reality and Visualisation
e-mail: prikryl@vrvis.at
* Institute of Computer Graphics, Vienna University of Technology
Karlsplatz 13/186/2, A-1040 Wien, Austria
e-mail: purgathofer@cg.tuwien.ac.at

Abstract

In this report we present an experimental efficiency study of spatial subdivision schemes for ray-shooting acceleration. Presented results are part of our effort to find the long-discussed best efficiency scheme *from the statistical point of view. We propose four testing procedures evaluating the ray-shooting algorithm properties. We use these methods to produce hardware independent statistics for different ray-shooting acceleration schemes that have been reimplemented following the published literature. We define the minimal set of parameters to be reported as well as scene invariants, parameters that stay the same regardless of the acceleration scheme used. The main results of first 1440 measurements for 30 scenes from the SPD database and 12 acceleration methods are reported. We also attempt to outline a method that suggests the most suitable acceleration scheme based on the scene complexity analysis of a given scene.*

1. Introduction

Ray-shooting algorithm is one of fundamental algorithms in computer graphics. It is used by virtually all modern rendering 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.

The ray-shooting task is actually simple: find out the first object intersected by a given ray for a given set of objects. In spite of this simple definition, it is not trivial to implement an efficient and fast ray-shooting method and the problem of finding an ultimately efficient algorithm still remains open. Both computational geometers and computer graphics researches have tried to develop fast ray-shooting acceleration schemes (RAS) 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 implementation in rendering frameworks. In computer graphics different heuristic acceleration methods improving the average-case time complexity of ray-shooting were researched. Even if the worst-case complexity of these heuristics is unfavourable, the good average-case complexity is the reason why these heuristic RASs are commonly used in recent rendering packages.

A naïve ray-shooting implementation 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 Marton[25] proved that the lower bound on the worst-case complexity of ray-shooting is $\Omega(\log N)$. The practical solutions exhibit worst-case complexity $O(N)$ and average-case complexity $O(1)$ for scenes with uniformly distributed objects[26]. These complexities remain for non-random scenes as well, but unfortunately, the unknown multi-

© 2001 Institute of Computer Graphics, Vienna University of Technology

plicative factor hidden behind the scenes and random object distribution in the scene for average case analysis make theoretical complexity definition unusable to decide which ray shooting algorithm should be used in practice. This is the reason why the RAS performance is compared on a set of test scenes, using for example scenes from *Standard Procedural Database* (SPD)[12].

Looking at the literature addressing RASs we find out that many recent results on different acceleration schemes are either limited to a subset of SPD scenes, or have been measured on private scene sets. Further, different implementation of reference algorithms, and the use of hardware-dependent timing statistics rather than hardware independent characteristics make a fair comparison of the published work very difficult. Additionally, it is not known if the SPD—although being scalable—is a good representative set of scenes suitable for RAS. As a result of all the facts mentioned above, many papers published about RASs contain mutually contradictory statements.

Researchers involved in ray-shooting acceleration have tried long to find the *best efficiency RAS*, that is, such an acceleration technique that outperforms other algorithms for any kind of ray-shooting task. As one can suppose, no RAS has been found to be generally the best until now. We propose an alternative to the search of the globally best-performing RAS: Based on statistics provided by number of different RAS tests we will try to find the *statistically best RAS*. The project presented in this paper is an ongoing work announced to the globillum mailing list in October 1999[15]. We call this project *Best Efficiency Scheme*, abbreviated BES. This research report shows our first results.

The paper is further structured as follows: Section 2 gives more details on goals of the BES project. Section 3 recalls the known scene complexity measures. In Section 4 we describe the design of testing procedures used. Section 5 presents the results from measurements on 30 SPD scenes of different complexities and the possible interpretation of the obtained results with regard to scene complexity measures. Finally, Section 6 concludes the paper with ideas for future work.

2. Project Goals

The basic idea of the BES project is to collect a reasonable set of test scenes of different complexity, and use these scenes to measure hardware-independent characteristics of different RASs. Collected scenes shall not exhibit self-similar behaviour described by the algorithms in the SPD database. The scenes and measured results will be made available to the graphics community in a suitable form. The results of the project for collected scene set will enable us to evaluate the properties of SPD scenes for testing.

The project shall help in clarifying the following points:

BES Existence. The question whether the best efficiency RAS does or does not exist will be answered using a statistically relevant set of scenes. We suppose that the answer will be negative, but it still has to be verified.

Alternative BES formulation. The proposal of an alternative definition of best efficiency scheme, based on hardware-independent statistics for a relevant number of scenes, will be given.

BES Testing Procedures. Simple testing procedures with moderate time requirements will be defined. These procedures can be used by other researchers to present properties of a new RAS. These procedures shall not be directly global-illumination algorithms requiring other computation than ray-shooting—they shall just perform different ray-shooting tasks.

BES Comparison. The precise summary and comparison of currently used RASs will be provided. In order to minimise discrepancies, all RASs are implemented within a uniform framework (GOLEM rendering system[13]).

BES Repository. A collection of freely available test scenes of different complexity will be made available for the scientific community. This can make future research in global illumination and the visibility field easier and verifiable, as the lack of the commonly used scenes makes it impossible to verify the results when reimplementing previously published methods for reference purposes.

BES Prediction. We will check how the proposed definitions of scene complexity could help us predict which RAS should be optimally selected in advance for the given scene. If such a prediction does not exist, the way to define such predictors will be opened.

Hybrid Methods. The question if it pays off to construct hybrid spatial data structures for some scene regions could be answered. The concept of *Meta-Hierarchies* was defined ten years ago[2], but we are not aware that there would be any implementation in use at present.

3. Scene Complexity

In order to estimate the scene complexity we can analyse our scenes and attempt to characterise

© 2001 Institute of Computer Graphics, Vienna University of Technology

them taking into account, for example, the number of objects, scene sparseness, sparseness variance and standard deviation, non-uniformity, and so on. We have made use of several methods evaluating scene characteristics that were introduced for this purpose [20, 5, 7].

3.1. Count Approach

The prevalent way to characterise the scene complexity is to take the number of objects in the scene, often referred to as *scene size*. Although this is a very simplistic definition of scene complexity — N objects —, it raises the question, whether the use of this complexity measure is not exaggerated. To our best knowledge no answer to this problem was given.

3.2. Voxelisation Approach

A method of scene characterization based on the presence of objects in voxels of a uniform grid has been proposed by Klimaszewski[20]chapter 4. The resolution of the grid is selected according to the $O(\sqrt[3]{N})$ rule, more precisely:

$$\text{resolution}_x = \text{resolution}_y = \text{resolution}_z = \lfloor \sqrt[3]{d \cdot N} + 0.5 \rfloor, \quad (1)$$

where N is the number of objects and d is the scene density (it is usually assumed $d = 1.0$).

The mean $\tilde{n}$ gives the average number of objects in a voxel:

$$\tilde{n} = \frac{1}{V} \sum_{i=1}^{V} n_i, \quad (2)$$

where n_i is the number of objects in the i-th voxel of a grid comprising V voxels. Variance v, the variability of the data around the mean, and standard deviation, σ are given as:

$$v = \frac{1}{V-1} \sum_{i=1}^{V} (n_i - \tilde{n})^2, \ \sigma = \sqrt{v}. \quad (3)$$

To measure the uneveness of a distribution, the nonuniformity coefficient is used, the larger it is, the larger the disparities among the voxel occupancy. It is defined as:

$$\lambda = \sigma / \tilde{n} \quad (4)$$

Additionally, higher order moments reported are known as *skewness*:

$$s = 1/V. \sum_{i=1}^{V} (\frac{n_i - \tilde{n}}{\sigma})^3, \quad (5)$$

and *kurtosis*:

$$k = [1/V. \sum_{i=1}^{V} (\frac{n_i - \tilde{n}}{\sigma})^4] - 3, \quad (6)$$

The objects should be assigned to the voxels using the intersection of the object surface with the voxel, not the intersection of the objects' bounding box with the voxel.

3.3. Integral Geometry Approach

Cazals and Sbert [5] investigated several integral geometry tools that characterise average case scene properties. Their strategy consisted in probing the scene with random entities (lines and planes) paying special attention to those statistics that may reveal the spatial distribution of scene objects.

We have selected all the random line-based tests for our measurements, which allowed us to determine the following characteristics: average number of intersection points for a global line crossing the scene n^G_{int}, probability of not intersecting any object in the scene p_0, and relative average length of a line span that lays in free space s_{len}:

3.3.1. Average number of intersection points

When casting a global line through the whole scene, an average number of intersections with scene objects n^G_{int} may be determined a priori as:

$$n^G_{\text{int}} = \frac{2}{A_{\text{bb}}} \sum_{i=1}^{N} A_i, \quad (7)$$

where A_{bb} is the surface area of a tight scene bounding box and A_i is the surface area of the i-th object in the scene.

3.3.2. Probability of zero intersections

The probability of ray not intersecting anything in the scene, denoted p_0, is quite hard to determine analytically. As we have to cast global lines anyway in order to compute another complexity characteristics, we compute the probability simply as:

$$p_0 = \frac{n_0}{n_{\text{total}}}, \quad (8)$$

where n_{total} is the total number of global lines casted and n_0 is the number of global lines that did not intersect any object in the scene.

© 2001 Institute of Computer Graphics, Vienna University of Technology

3.3.3. Free path statistics

While tracing a global line trough a scene, every intersection adds additional "span" to the traced line. Average length of spans may give us insight into the spatial density of the scene in concern. For every global line n_i casted we sum up span lengths l_k and identify also the maximum span length $l_{\max}$. For the total of M spans the free path statistic is then given as:

$$s_{\text{len}} = \frac{1}{M \cdot l_{\max}} \sum_{i=1}^{M} l_i. \qquad (9)$$

3.4. Information Theory Approach

Feixas *et al.* [7] describe the task of determining the scene complexity as a task of determining the mutual information transfer. In their paper they present a number of complexity measures from information theory quantifying how difficult is to accurately compute visibility in the scene. While working with scene discretised into patches, the paper contains also a definition of scene continuous mutual information, that is mutual information independent of whatever discretisation of the scene. We have used the continuous mutual information I_S^C to characterise our scenes.

The scene continuous mutual visibility information can be easily determined using Monte-Carlo integration with global lines. The whole quite involved formula boils down to:

$$I_S^C \approx \frac{1}{M} \sum_{i=1} M \log \left(\frac{A_{\text{total}} \cos\theta_x \cos\theta_y}{\pi d(x,y)^2} \right), \qquad (10)$$

where M stands for total number of point pairs observed, A_{total} is the total scene surface area, (x, y) is the point pair in concern, $d(x, y)$ is the distance of those two points and θ_x, θ_y are the angles between the direction vector and corresponding normal vector. As objects in SPD scenes do not overlap, for the purposes of this paper we have $A_{\text{total}} = \sum_{i=1}^{N} A_i$. In case of more complex geometry (CSG objects), the stochastic area estimation method proposed by Wilkie *et al.* [28] can be used.

4. Testing Procedures

We designed the RAS testing procedures to emulate the use of ray-shooting in rendering algorithms. During the tests we take only surface geometry into account and do not perform any lighting or material calculations. This way the most computational time in the tests is really devoted to ray-shooting.

Rays shot during rendering can be classified as primary (leaving the camera), secondary (reflected and refracted rays), and shadow rays. As shadow rays are actually used just for light source visibility tests and these tests may be optimized regardless of the RAS used, we disregard shadow rays in our tests to make the testing procedures consistent.

We use four different testing procedures. The first three tests are general methods simulating ray–shooting in rendering algorithms. The last test is ray–tracing an image as defined in SPD database. We are not rendering it just for the purpose of getting a visually appealing image, it also allows for finding out errors when implementing a new RAS, and it also allows subjective evaluation of scene properties.

In case that a testing scene has quite a varying object distribution, performing ray-shooting tests for a single ray origin located somewhere in the scene may reveal only local properties of the tested RAS. In order to test the RAS behaviour for the whole scene, use of uniformly–distributed global rays is more appropriate. In order to obtain equal ray distributions for all tested RASs the same initial seeds for random number generators have to be used in every test procedure.

The usual way of generating global lines is to generate two uniformly distributed points on the scene bounding sphere and shoot the ray between these two points. This method would be however unfair to those RASs that use directional acceleration schemes[3, 24]. Algorithm 1 shows an alternative uniform global ray generation scheme that also preserves ray coherency of the subsequent rays at the same time.

4.1. Definition of Testing Procedures

We have used the following testing procedures in our tests:

Testing procedure TP_A: Shoots only primary rays generated using Algorithm 1 on a sphere circumscribed to the scene bounding box. The bounding box is computed as a union of bounding boxes of all scene objects.

Testing procedure TP_B: Assigns a tight rectangular bounding box to each object in the scene. For each bounding box i computes its center point Q_i and computes the center of the sphere as $\vec{C} = 1/N \cdot \sum_{i=1}^{N} Q_i$. Using a binary search finds a minimum radius of a sphere containing 90% of all Q_i. Shoots only primary rays generated on this sphere using Algorithm 1.

Testing procedure TP_C: The same as TP_B, but

© 2001 Institute of Computer Graphics, Vienna University of Technology

Algorithm 1 Systematically casting $N \cdot (N-1)$ rays uniformly distributed on sphere

{Compute N points uniformly distributed on the unit sphere}
$B \leftarrow$ number of bands
Subdivide sphere by $(B-1)$ planes $z = const$
$\{z_k = 1 - 2(k+1)/B,\ k \in \{0, B-2\}\}$
{Create B bands having the same surface area on the sphere}
$b \leftarrow 0$ {current band index }
$S_{\text{one}} \leftarrow 4/3 \cdot \pi/N$ {required surface area of one region}
$\alpha \leftarrow 0$ {the end angle of the current region}
for all $i \in N$ points **do**
  $S_{\text{curr}} \leftarrow 0$ {Current region has zero area. Always generate point at the end of new region}
  Integrate bands by increasing α or/and the b until $S_{\text{curr}} = S_{\text{one}}$
  $\vec{P}_i.z = 0.5 \cdot (1 - (1-2i)/N)$
  $R = \sqrt{1.0 - (\vec{P}_i.z)^2}$, $\vec{P}_i.x = R\cos(\alpha_i)$, $\vec{P}_i.y = R\sin(\alpha_i)$
end for
Transform N points to world space given the sphere center $\vec{C}$ and radius R
for all $i \in N$ points **do**
  for all $j \in N$ points **do**
    if $i \neq j$ **then**
      Shoot (primary) ray between points i and j on the sphere in the world space.
    end if
  end for
end for

rays are randomly reflected using uniform distribution over the hemisphere given by the surface normal at the hit point. The bounces continue until the maximum depth of recursion $d_{\max} = 4$ is reached (primary rays have recursion depth 0) or until the ray leaves the scene.

Testing procedure TP_D: Recursive ray tracing exactly as defined in SPD. This task requires a camera to be set and surface materials to be defined. Depth of recursion is the same as for TP_C, the number of primary rays cast is 513×513. All other details can be found in `Readme.txt` in the SPD distribution[12].

As we can see, procedure TP_A uses the whole scene (rays are shot from the outside into the scene). Procedures TP_B and TP_C generate rays in the space where most objects are present. Procedure TP_C simulates a random walk. Results of the test TP_D will then show how the results of TP_A–TP_C correlate with the RAS performance for a common rendering task.

Obviously, it is always possible to construct an artificial scene where our testing procedures will not work as expected. However, scenes that are used for practical purposes, will not pose difficulties to our tests.

4.2. Invariants

For every tested scene we can determine a set of parameters that shall remain constant regardless of the RAS used. These *invariants* can be used to verify results of the acceleration scheme implementation. However, we shall be aware that this verification does not impose a globally correct RAS implementation, it merely proves that the results are correct for the particular scene.

The invariants are:

- $N^{\text{hit}}_{\text{bbox}}$, the number of primary rays hitting the scene box (applies for TP_A-TP_D),
- $N^{\text{hit}}_{\text{prim}}$, the number of primary rays hitting any object (applies for TP_A-TP_D),
- N_{sec}, the number of secondary rays (reflected rays for TP_C, reflected and refracted rays for TP_D),
- $N^{\text{hit}}_{\text{sec}}$, the number of secondary rays hitting any object (reflected rays for TP_C, reflected and refracted rays for TP_D),
- N_{shad}, the number of shadow rays (TP_D only), and
- $N^{\text{hit}}_{\text{shad}}$, the number of shadow rays hitting opaque objects (TP_D only).

In practice we observed that the invariants are equal or that they differ just very slightly due to the numerical precision problems. The relative error for shooting 10^6 rays using single-precision floating point arithmetic was always below $\epsilon = 10^{-4}$ in our experiments, which is acceptable under assumption no differences between images obtained as a result of TP_D for different RASs are visible. Given the finite precision arithmetic, we feel that tuning the RAS implementations to get exactly the same results can be even impossible to achieve.

4.3. Minimum Testing Output

Performance of a ray-shooting acceleration scheme can be described by some parameters related either to the RAS itself or to the particular ray-shooting task. In this Section we will define the minimum set of RAS parameters that shall be recorded during our tests.

All RASs can be considered to be different instances of range-search query[1]. This implies that every RAS can be separated into two parts: the

© 2001 Institute of Computer Graphics, Vienna University of Technology

data structure DT and the *algorithm* A working over DT. The DT contains references to objects that are to be tested for the intersection against a given ray. The DT is also composed of certain number of *nodes*.

Nodes of a DT can be divided into two groups: *elementary nodes* contain references to objects (and, if RAS 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. Special cases of an elementary nodes are *empty elementary nodes* that do not contain any object references and act as "free space containers" within the RAS. Given a ray, the algorithm A (usually called *traversal algorithm*) operates over the DT, traversing it from one node to another, possibly adding/deleting nodes to any part of DT on the fly, testing objects for intersection in the elementary nodes, and finally finding the nearest intersection.

All RASs currently known fit in this general framework. This makes it possible to define the following hardware-independent parameters that can be reported for any RAS:

- parameters depending on the scene S only:
  - N_G, the number of generic nodes in DT,
  - N_E, the number of elementary nodes in DT,
  - N_{EE}, the number of empty elementary nodes in DT,
  - N_{ER}, the total number of references to objects in elementary nodes of DT.
- parameters dependent on the scene S and testing procedure TP:
  - N_{IT}, the number of intersection tests per one ray,
  - N_{TS}, the number of all nodes accessed per one ray,
  - N_{ETS}, the number of elementary nodes accessed per one ray,
  - N_{EETS} the number of empty elementary nodes accessed per one ray.

We will also record following hardware-dependent timings:

- T_B[s], time to build DT for given RAS (depends on the scene S),
- T_R[s], time consumed by given TP (depends on the scene S, and on the testing procedure TP).

For T_B and T_R the hardware should be precisely described together with operating system, compiler, and compiler switches used. If possible, the RAS should be given not only in as pseudo-code in the paper, but also the source code shall be made freely available as all implementation details are usually important.

The relationship between HW implementation parameters and independent parameters can be described by a general cost model, first introduced by Cleary and Wyvill[6]:

$$\mathrm{T} = (N_{IT}C_{IT} + N_{TS}C_{TS}).N^{\mathrm{rays}}_{\mathrm{total}} + T_{\mathrm{other}},$$

where $C_{IT}[s]$ is the average cost of intersection a ray with an object and $C_{TS}[s]$ is the average traversal cost of RAS among DT nodes. $N^{\mathrm{rays}}_{\mathrm{total}}$ is the total number of rays cast. Parameter $T_{\mathrm{other}}[s]$ covers other computations performed in the given TP, as the ray reflection for TP_C and lighting and material computation for TP_D, that should be always constant for a particular scene and TP.

5. Results and Discussion

In this section we describe the scenes used for the project, their scene complexity measures, and the results for all testing procedures.

5.1. Test Scenes

The process of collecting and preparing scenes has started in October 1999 and it still continues. We decided to group the collected scenes according to the number of objects, creating 7 groups: G^X, $X \in 0 \ldots 5$, 15 scenes containing $\langle 10^X+1, 10^{X+1} \rangle$, and G^6, 10 scenes with more than 10^6 objects.

There are many WWW sites offering 3D models usable for our purposes. We have however encountered two problems: First, models are usually available in proprietary formats and conversion into open formats (VRML in our case) does not usually work very well. This results in scenes having corrupted faces, invalid normals, missing textures, and so on. Second, one can never estimate the scene size before actually downloading the model. We observed that most of the 215 scenes downloaded until now typically contain $5 \cdot 10^3$–$5 \cdot 10^4$ primitives. Scenes having less than 100 or more then $5 \cdot 10^5$ objects primitives were not available at all. While small scenes can be modelled, composing meaningful large scenes is quite demanding task. As a result, groups supposed to contain scenes with higher numbers of primitives are still incomplete.

In experiments presented in this paper we have used three groups of SPD scenes with different object counts. Since SPD scenes are scalable, we decided to generate individual scenes with object counts as close as possible to maximum counts required scene size: group G^3_{SPD} (10^3 objects), G^4_{SPD}

© 2001 Institute of Computer Graphics, Vienna University of Technology

(10^4 objects), G^5_{SPD} (10^5 objects). The size of the generated SPD scene depends on an optional size factor S_F, where ratio of scene sizes for S_F and $S_F + 1$ varies from 1.2 for "teapot" to 8.0 for "jacks". Table 1 lists numbers of objects in different SPD scenes for $S_F = 1 \ldots 6$. The bold typeset numbers of objects denote scenes selected into groups G^3_{SPD}, G^4_{SPD}, and G^5_{SPD}. We decided not to use scene "shells" for our experiments, as this is the only SPD scene containing densely overlapped objects (that should not be the case of correctly modelled scene), and causes problems for any RAS that we tested.

Table 2 shows selected scene complexity measures for the 30 SPD scenes. Comparing the computed complexities with testing results presented below, it unfortunately seems that there is no direct correlation between existing complexity measures and RAS performance.

5.2. Results

We implemented the following acceleration schemes:

BSP: Binary space partitioning[19] using efficient hierarchical traversal algorithm[16]. The subdivision plane creates always equally-sized children. Maximum allowed depth was 16, maximally 2 objects were allowed in a node.

KD: *KD*-tree, similar to *BSP*, but the subdivision plane is put according to the surface area heuristics[22]. The same build criteria as for *BSP* were used.

UG: Uniform grid, also called uniform space subdivision[8], with resolution according to[17] (Woo's method) with density 3.0.

BVH: Bounding volume hierarchy built with cost function[11].

AG: Adaptive grid, *BVH* over uniform grids[21].

RG: Recursive grid, a grid recursively put in parent grid voxels again[18].

HUG: Hierarchy of uniform grids[4].

O84: Octree with sequential traversal build using midpoint subdivision[10].

O84A: Sequential traversal + Octree-R using surface area heuristics[10, 27].

O89: Octree using neighbour finding for traversal[23] using midpoint subdivision.

O93: Octree using recursive traversal algorithm[9].

O93A: Octree using recursive traversal algorithm[9] and surface area heuristics[27].

The detailed parameter settings of tested RASs are beside the scope of this paper. We have consistently used the best settings that we found during previous experiments[17, 14] with one exception—in case we ran out of memory, we allowed three iterations of modifying RAS build parameters to require less memory and testing again. This occurred for *RG*, *AG*, and *HUG*. Failures of this iterative setting are reported in Table 4. There are two cases when results are not reported: either the testing procedure did not finish in 10 hours, or the computer memory was exhausted even after three iterative parameter modifications described above. It should be clear that manual tuning of build parameters to construct failure-proof *DT*s for 12 given RASs and 30 scenes is quite impractical.

All the tests were conducted on PCs running Linux, kernel version 2.2.12-20, processor Intel Pentium II, 350 MHz, 128 MB RAM. Test program was compiled using egcs-1.1.2 with -O2 optimisation. The total number of measurements is 1440 (12 RASs by 30 scenes by 4 testing procedures). With 10 reportable parameters for every measurement (see Section 4.3) we have measured 11520 hardware-independent and 2880 hardware-dependent RAS parameters.

Due to space limitations it is not possible to report all the measurements in this paper. We have therefore selected main characteristics from all tests and we present a short summary. All measured statistics are available on the WWW site of the project[15].

Table 3 reports for every tested scene time T_B needed to build DT for the fastest RAS and minimum time T_R over all RAS needed to run the given testing procedure.

Table 4 reports the average running time T^{avg}_R for given RAS and TP for the all scenes and summary times for columns and rows. The parameter m is the number of tasks where measurements failed due to the memory limits, l denotes the number of cases when tests were not finished within the time limit. RASs are sorted in T^{avg}_R for total sum including all TPs. We can see that the winner on the tests on SPD scenes is *KD*-tree, while *BVH* has the worst average running time, being in some tests even more than two orders of magnitude slower than *KD*-tree.

The total running time of the whole experiment was about 400 hours on a single processor. Tests TP_A–TP_C used 10^6 primary rays.

Graphs in Figures 1 and 2 show the summary of hardware independent parameters. Parameters for every RAS are summed over all tested scenes and testing procedures. Graph 3 shows for every RAS its total running time T_R and construction

© 2001 Institute of Computer Graphics, Vienna University of Technology

S_F	Scene									
	balls	gears	jacks	lattice	mount	rings	sombrero	teapot	tetra	tree
1	11	147	9	20	12	61	**1922**	57	4	7
2	92	**1169**	81	81	36	301	**7938**	244	16	15
3	**821**	3943	**657**	208	132	**841**	32258	561	64	31
4	**7382**	**9345**	**5265**	425	**516**	1801	**130050**	**1008**	256	63
5	**66341**	18251	**42129**	756	2052	3301	522242	1585	**1024**	127
6	597876	31537	337041	**1255**	**8196**	5461	–	2292	**4096**	255

Table 1: *Number of objects of SPD scenes according to the size factor S_F.*

Scene(S_F)	Group	N	A_{total}	$\tilde{n}$	σ	s	k	n^G_{int}	p_0	s_{len}	I^C_S
balls(3)	G^3_{SPD}	821	588	0.764	5.41	14.3	228	0.920	0.995	0.035	9.384
balls(4)	G^4_{SPD}	7382	591	0.584	7.50	23.1	593	0.924	0.995	0.038	9.957
balls(5)	G^5_{SPD}	66341	594	0.481	10.2	32.5	1170	0.929	0.995	0.035	10.242
gears(2)	G^3_{SPD}	1169	32.6	0.760	2.53	4.97	27.2	1.356	0.788	0.120	4.333
gears(4)	G^4_{SPD}	9345	59.6	1.25	3.52	3.38	11.3	2.478	0.723	0.066	7.792
gears(9)	G^5_{SPD}	106435	126	1.25	3.82	3.68	13.1	5.245	0.690	0.032	10.735
jacks(3)	G^3_{SPD}	657	26.7	1.29	1.77	1.11	-2.90e-2	1.317	0.750	0.108	4.398
jacks(4)	G^4_{SPD}	5265	57.3	1.20	1.87	1.28	0.264	2.498	0.662	0.092	6.769
jacks(5)	G^5_{SPD}	42129	118	1.20	1.96	1.37	0.464	4.869	0.607	0.076	9.365
lattice(6)	G^3_{SPD}	1255	14.7	2.09	1.66	0.57	-4.46e-1	4.166	0.300	0.101	5.165
lattice(12)	G^4_{SPD}	8281	24.5	1.87	1.52	0.77	0.090	7.488	0.174	0.081	7.279
lattice(29)	G^5_{SPD}	105307	52.7	1.76	1.54	1.11	0.830	16.919	0.090	0.061	10.757
mount(4)	G^3_{SPD}	516	9.25	1.57	3.20	2.14	3.93	0.676	0.867	0.149	5.189
mount(6)	G^4_{SPD}	8196	10.1	0.986	3.40	3.88	15.4	0.743	0.851	0.131	5.808
mount(8)	G^5_{SPD}	131076	10.5	0.773	4.49	6.75	48.9	0.792	0.851	0.108	5.808
rings(3)	G^3_{SPD}	841	362	1.14	2.97	2.97	8.40	1.162	0.867	0.125	3.747
rings(7)	G^4_{SPD}	8401	2.82e4	1.15	2.78	2.62	6.57	2.461	0.749	0.078	6.722
rings(17)	G^5_{SPD}	10701	3.27e5	1.10	2.55	2.41	5.19	5.952	0.642	0.043	10.381
sombrero(1)	G^3_{SPD}	1922	72.9	1.31	2.60	1.71	1.32	0.812	0.966	0.105	5.152
sombrero(2)	G^4_{SPD}	7938	73.5	1.08	2.88	2.67	6.35	0.819	0.963	0.120	4.786
sombrero(4)	G^5_{SPD}	130050	73.7	0.772	3.40	4.46	18.8	0.820	0.962	0.123	4.628
teapot(4)	G^3_{SPD}	1008	115	1.26	3.88	5.49	45.2	1.009	0.795	0.267	6.647
teapot(12)	G^4_{SPD}	9264	116	0.967	4.74	10.2	159	1.020	0.791	0.272	6.595
teapot(40)	G^5_{SPD}	103680	117	0.743	5.72	16.6	421	1.021	0.791	0.272	6.600
tetra(5)	G^3_{SPD}	1024	13.9	1.40	2.68	1.70	1.78	1.152	0.641	0.065	5.027
tetra(6)	G^4_{SPD}	4096	13.9	1.26	3.12	2.62	6.46	1.152	0.690	0.051	6.244
tetra(8)	G^5_{SPD}	65536	13.9	0.896	3.64	4.61	22.5	1.152	0.815	0.062	7.738
tree(8)	G^3_{SPD}	1023	1.00e5	0.839	18.4	55.0	3030	0.943	1.000	0.138	5.807
tree(11)	G^4_{SPD}	8191	1.00e5	0.791	28.5	81.2	7040	0.941	1.000	0.136	7.006
tree(15)	G^5_{SPD}	131071	1.00e5	0.626	46.1	121	1.69e5	0.941	1.000	0.154	6.275

Table 2: *Scene complexity according to the Section 3 for G^3_{SPD}, G^4_{SPD}, and G^5_{SPD} scenes*

time T_B and sum of both summed over all testing procedures and scenes (120 measurements in case all the tests completed successfully).

5.3. Discussion

Comparing results presented in Tables 3 and 4, and Figures 1, 2, and 3 together with all the extra data[15], we can comment on the tested data structures for ray-shooting acceleration:

© 2001 Institute of Computer Graphics, Vienna University of Technology

Scene(S_F)	Testing procedures											
	TP_A			TP_B			TP_C			TP_D		
	RAS	$T_B[s]$	$T_R[s]$	RAS	$T_B[s]$	$T_R[s]$	RAS	$T_B[s]$	$T_R[s]$	RAS	$T_B[s]$	$T_R[s]$
balls(3)	KD	0.30	3.72	KD	0.30	13.63	KD	0.30	50.25	KD	0.30	21.4
balls(4)	KD	1.75	3.51	KD	1.75	17.64	KD	1.75	62.87	KD	1.75	27.06
balls(5)	KD	16.2	3.82	KD	16.2	31.41	KD	16.2	102.2	KD	16.2	42.0
gears(2)	KD	0.6	5.01	KD	0.6	10.1	KD	0.6	34.24	KD	0.6	38.96
gears(4)	KD	2.45	5.71	KD	2.45	12.94	KD	2.45	57.49	KD	2.45	36.24
gears(9)	KD	22.79	7.91	UG	7.24	18.36	KD	21.89	91.01	KD	22.97	40.59
jacks(3)	UG	0.04	12.59	UG	0.04	30.51	UG	0.04	74.05	UG	0.04	10.46
jacks(4)	UG	0.3	16.38	UG	0.3	38.87	UG	0.3	118.8	UG	0.3	19.82
jacks(5)	KD	12.68	21.76	UG	2.9	46.45	UG	2.9	167.4	UG	2.9	30.69
lattice(6)	UG	0.1	9.47	UG	0.1	13.36	UG	0.1	48.56	UG	0.1	33.24
lattice(12)	UG	0.6	12.02	UG	0.6	16.88	UG	0.6	72.96	UG	0.6	39.95
lattice(29)	UG	8.6	14.53	UG	8.6	19.35	UG	8.6	93.45	UG	8.6	43.65
mount(4)	KD	0.09	6.83	KD	0.09	11.18	KD	0.09	26.65	KD	0.09	18.88
mount(6)	KD	1.49	9.11	KD	1.49	15.71	KD	1.49	39.3	KD	1.49	21.06
mount(8)	KD	25.9	18.45	KD	25.9	37.07	KD	26.9	114.7	KD	25.82	25.14
rings(3)	KD	0.47	8.14	KD	0.47	30.01	KD	0.47	80.93	KD	0.47	40.39
rings(7)	KD	2.36	12.61	KD	2.36	38.09	KD	2.36	139.5	KD	2.36	64.76
rings(17)	KD	23.55	22.23	KD	23.55	61.13	KD	23.55	260.4	KD	23.55	106.6
sombrero(1)	KD	0.32	6.18	KD	0.3	8.00	KD	0.31	18.21	KD	0.33	3.82
sombrero(2)	KD	1.37	6.82	KD	1.37	9.16	KD	1.37	20.99	KD	1.37	4.0
sombrero(4)	KD	26.39	11.83	KD	26.39	16.88	KD	26.39	40.61	KD	26.39	6.9
teapot(4)	KD	0.38	5.65	KD	0.38	11.56	KD	0.38	35.25	KD	0.38	13.94
teapot(12)	KD	2.18	6.65	KD	2.18	13.89	KD	2.18	43.12	KD	2.18	15.66
teapot(40)	KD	22.4	9.54	KD	22.4	23.46	KD	22.4	74.58	KD	22.39	23.85
tetra(5)	KD	0.1	6.6	KD	0.1	9.33	KD	0.1	19.6	KD	0.1	2.48
tetra(6)	KD	0.49	7.74	KD	0.49	10.9	KD	0.47	21.96	KD	0.47	2.66
tetra(8)	KD	10.9	13.69	KD	10.9	19.47	KD	10.9	36.65	KD	10.9	3.57
tree(8)	RG	0.13	3.72	KD	0.34	20.66	KD	0.34	34.23	KD	0.34	18.39
tree(11)	AG	1.75	3.82	AG	1.8	29.96	KD	2.04	47.28	AG	1.8	20.61
tree(15)	RG	358.5	3.72	HUG	10.4	76.37	AG	380.0	68.71	AG	380.0	43.38

Table 3: *The RASs with minimum $T_R[s]$ for $TP_{[A,B,C,D]}$*

BVH: Has rather poor results for all TPs compared to other RASs. We see the main problem in the nature of *BVH* construction. This scheme does not keep track of spatial coherency–when inserting a new object to the hierarchy, there is no global information about other still uninserted objects.

O84: Octree with sequential traversal algorithm requires quite a lot of traversal steps from the root node. Subdividing in midpoints does not work particularly well for sparse scenes ("tree").

O89: Has slightly better traversal algorithm which outperforms *O84* especially for scenes with higher numbers of objects.

BSP: Although conceptually the same structure as *KD*-tree, subdividing in midpoints results again in poor performance for sparse scenes. For densely occupied scenes *BSP* performs comparably to *KD*-tree.

O93: Due to the most efficient traversal algorithm outperforms the *O84* and *O89* even if constructed using the midpoint subdivision. We can see that for all midpoint subdivision octrees, shooting rays inside the octree (TP_B, TP_C, TP_D) is very time demanding. This is the price we pay for

© 2001 Institute of Computer Graphics, Vienna University of Technology

RAS	Testing procedures												Total		
	TP_A			TP_B			TP_C			TP_D			$\sum_{TP=A,B,C,D}$		
	T_R^{avg}	m/l	F	T_R^{avg}	m/l	F	T_R^{avg}	m/l	F	T_R^{avg}	m/l	F	T_R^{avg}	m/l	F
KD	9.98	0/0	22	26.29	0/0	21	76.27	0/0	23	29.42	0/0	22	142.0	0/0	88
O93A	15.01	0/0	0	40.91	0/0	0	106.8	0/0	0	38.59	0/0	0	201.3	0/0	0
O84A	15.41	0/0	0	41.72	0/0	0	109.0	0/0	0	39.08	0/0	0	205.2	0/0	0
RG	35.09	3/0	2	64.44	3/3	0	134.4	0/2	0	47.84	0/0	0	281.8	6/5	2
HUG	39.63	0/0	0	99.52	0/0	1	268.2	0/0	0	77.38	0/0	1	484.7	0/0	1
AG	63.31	3/0	1	108.6	3/0	1	278.9	3/1	1	136.5	0/0	2	587.3	9/1	5
UG	11.74	0/0	5	372.7	0/0	7	525.5	0/0	6	145.4	0/0	6	1055	0/0	24
O93	16.86	0/0	0	1114	0/0	0	257.5	0/0	0	392.8	0/0	0	1781	0/0	0
BSP	28.63	0/0	0	1291	0/0	0	325.9	0/0	0	560.3	0/1	0	2206	0/1	0
O89	14.49	0/0	0	1127	0/0	0	1421	0/0	0	381.7	0/0	0	2944	0/0	0
O84	17.44	0/0	0	1132	0/0	0	1437	0/0	0	400.2	0/0	0	2987	0/0	0
BVH	1903	0/0	0	4569	0/2	0	5111	0/6	0	3376	0/0	0	14960	0/8	0
$\sum_{allRAS}$	2170	6/0	12	9986	6/5	12	10050	3/9	12	5625	0/1	12	27832	15/15	120

Table 4: *Average tuning time T_R^{avg}, memory (m) and time limit (l) failures, and number of "wins" F for all tested acceleration algorithms and testing procedures.*

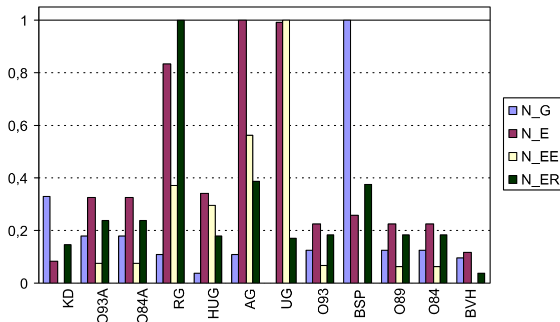

Figure 1: *Parameters N_G, N_E, N_{EE}, N_{ER} summed for all scenes and every RAS, normalized to the worst RAS.*

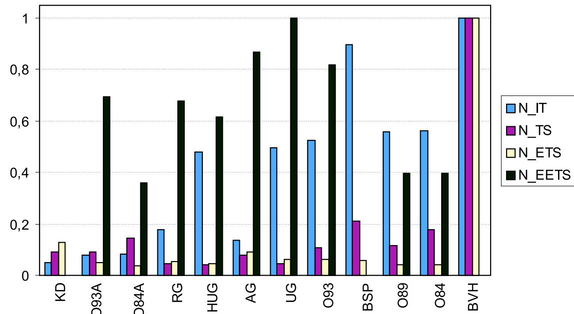

Figure 2: *Parameters N_{IT}, N_{TS}, N_{ETS}, N_{EETS} summed for all scenes for every RAS and normalized to the worst RAS.*

traversal down to the leaf when the intersected object is in this node or very close.

UG: Classical acceleration scheme. The employed smart algorithm for heterogeneous grid resolution setting results in best performance for several scenes. These scenes are densely occupied with mostly regular structure ("jacks", "lattice", and "mount"). In this kind of scenes the down traversal phase for hierarchical spatial data structures is expensive, since the ray intersects object very close to the ray origin. For sparsely occupied scenes *UG* has rather poor performance as it lacks a sense of hierarchy.

AG: as combination of *BVH* with *UG* has reasonable performance, but the prediction of memory needed to construct *DT* is difficult for G^5_{SPD} scenes, for one scene the computation failed time limit due to swapping. Tweaking of build settings was necessary to get some G^5_{SPD} scenes to work on available memory.

HUG: Consists of *UG*s arbitrarily positioned in other *UG*s. Has not only a slightly better performance than *AG*, but also much smaller and predictable memory usage.

RG: *UG* inserted in voxels of *UG* recursively shows negligible performance improvement especially for TP_C. Its performance varies, five tests failed on the time limit. The tuning of parameters to keep the test within the memory limit was difficult as memory consumption was rather unpredictable.

© 2001 Institute of Computer Graphics, Vienna University of Technology

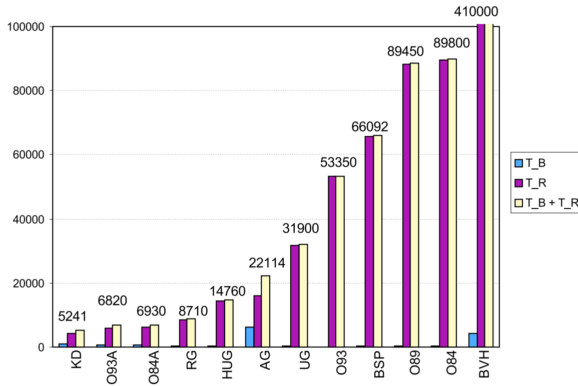

Figure 3: *Total times $T_B[s]$, $T_R[s]$, $T_B+T_R[s]$ for every RAS, the last number gives the total running+construction time (T_B+T_R) for all tests for particular RAS.*

O84A: Octree built using the surface area heuristics. We can see that although a simple traversal algorithm was used, the more appropriate subdivision process improved the total performance by one order of magnitude. The improvement is particularly apparent on sparse scenes in G^5_{SPD}. Unfortunately, the build time T_B rises rapidly for G^5_{SPD} as well.

O93A: The same improvement as between *O84* and *O93* can be observed due to the more efficient traversal algorithm. Again, the build time rises rapidly with the number of objects in the scene.

KD: Although being principally a *BSP* tree, the subdivision using the surface area heuristics and fast traversal algorithm makes from this hierarchal spatial data structure a winner even if improvements from *O93A* are not dramatic. It has been beaten in several cases: for regular artificial scenes ("jacks", "lattice", and "mount") by *UG*, for G^5_{SPD} sparse scene ("tree") by *AG* and *HUG*. However, the differences in performance in all these cases are small. The only disadvantage is that the build time for G^5_{SPD} scenes, which is comparable to build times *O93A* and *O84A*, can be rather high in comparison with *UG*. Therefore, if the number of rays to be cast is low, using *KD*-tree probably does not pay off.

In general, we observe that using the surface area heuristics[22] pays off for both Octree and BSP tree to get Octree-R and *KD*-tree. Also, hierarchical data structures win over non-hierarchical ones in most cases, especially for sparse scenes.

If these results and the ranking of algorithms according to $\sum_{TP=A,B,C,D} T_R$ will remain for the collection of 100 downloaded scenes is not currently known.

Examining the results of Tables 2 and 3 we would like to present a preliminary proposal of an algorithm for selecting an RAS to be used given a scene: First, construct an uniform grid $G_{d=1.0}$ over the bounding boxes of objects using the heterogeneous resolution setting, and voxel density 1.0. Then compute sparseness parameters δ, s, and k. When the δ, s, and k are low (see Table 2), then construct uniform grid. If these three parameters are in the middle range, construct a *KD*-tree using the surface area heuristics. If these parameters are very high, then consider either *KD*-tree or *RG*/*AG* for even better performance, but be aware of difficult predictance of memory requirements for these hierarchical grids. If nothing from rules above is applicable, construct *KD*-tree. Here, we should stress out, that the preliminary algorithm is derived from the measurements of 30 SPD scenes with fractal nature. Its validity has to be verified on larger set of scenes. The knowledge if the most rays will be shot inside the scene will be helpful for RAS selection as well.

6. Conclusion and Future Work

In this paper we outlined the goals of BES project and its status in late March 2000. We have defined testing procedures for ray-shooting acceleration schemes, RAS invariants, and an algorithm for systematically shooting uniformly distributed rays. We have shown which parameters describing computation using RAS can be recorded. Based on the collected data we also outlined a heuristics for selection of suitable RAS given a statistics based on scene sparseness. Even if the heuristics predicts reasonably well for tested SPD scenes, we do not know if the algorithm in its present form will be applicable to general scenes and its success ratio. It is only clear, that for small number of rays to be shot the construction of any *DT* does not pay off at all.

Testing 12 RAS over 30 SPD scenes of different complexities and 4 testing procedures we can conclude that using hierarchical spatial data structures for ray-shooting acceleration, particularly *KD*-tree, definitely pay off—with exception of densely occupied scenes. This observation supports our opinion that only very unlikely there will be a single optimal RAS for general use in ray-shooting.

This project is still not completed. In order to provide a sound basis that would help us to

© 2001 Institute of Computer Graphics, Vienna University of Technology

avoid further speculations about the design and use of different ray-shooting algorithms, several tasks have to be completed: First, we have to make our collection of 100 practical scenes complete and run all the tests presented in this paper again over this set. This will provide us with a vast amount of statistical data that will have to be analyzed together with measurements presented in this paper. Having the results ready, we can conclude if the result presented in this paper correlate somehow with the results obtained for the practical scenes. This will also reveal how well is the SPD distribution actually simulating the real scenes.

Acknowledgements

The authors would like to thank Mateu Sbert from Universitat de Girona for his advises to reimplementation of the complexity measures and Algorithm 1. JiříŽára from CTU Prague gently allowed us to use the hardware of Computer Graphics Lab overnights for our measurements. We would also like to thank to all who commented the BES proposal, particularly to Eric Haines. This work has been supported by the joint Czech-Austrian scientific collaboration funding under project number 1999/17.

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. J. Arvo. Ray tracing with meta-hierarchies. In *SIGGRAPH '90 Advanced Topics in Ray Tracing course notes*. ACM Press, August 1990.
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. F. Cazals and C. Puech. Bucket-like space partitioning data-structures with applications to ray-tracing. In *13th ACM Symposium on Computational Geometry*, pages 11–20, Nice, 1997.
5. F. Cazals and M. Sbert. Some integral geometry tools to estimate the complexity of 3d scenes. Technical Report RR-3204, INRIA, 1997.
6. J.G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4(2):65–83, 1988.
7. M. Feixas, E. del Acebo, P. Bekaert, and M. Sbert. An information theory framework for the analysis of scene complexity. In P. Brunet and R. Scopigno, editors, *Eurographics '97 Conference Proceedings*, volume 18 of *Computer Graphics Forum*, pages 95–106, September 1999.
8. A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.
9. I. Gargantini and H. H. Atkinson. Ray tracing an octree: numerical evaluation of the first intersection. *Computer Graphics Forum*, 12(4):199–210, October 1993.
10. A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.
11. J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.
12. E. A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987. available from `http://www.acm.org/pubs/tog/resources/SPD/overview.html`.
13. V. Havran. GOLEM Rendering System, 1997–2000, `http://www.cgg.cvut.cz/GOLEM`.
14. V. Havran. A summary of octree ray traversal algorithms. *Ray Tracing News*, 12(2):cca 10 pages, Dec. 1999. `http://www.acm.org/tog/resources/RTNews/html/rtnv12n2.html`.
15. V. Havran, J. Bittner, and J. Přikryl. Best efficiency scheme project proposal. `http://www.cgg.cvut.cz/GOLEM/bes.html`, October 1999.
16. V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast robust bsp tree traversal algorithm for ray tracing. *Journal of Graphics Tools*, 2(4):15–23, December 1998.
17. V. Havran and F. Sixta. Comparison of hierarchical grids. *Ray Tracing News*, 12(1):cca 4 pages, June 1999. `http://www.acm.org/tog/resources/RTNews/html/rtnv12n1.html`.
18. D. Jevans and B. Wyvill. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89*, pages 164–172, June 1989.
19. M. Kaplan. *Space-Tracing: A Constant Time Ray-Tracer*, pages 149–158. July 1985.

© 2001 Institute of Computer Graphics, Vienna University of Technology

20. K. S. Klimaszewski. *Faster Ray Tracing Using Adaptive Grids and Area Sampling.* PhD thesis, Dept. of Civil and Environmental Engineering, Brigham Young University, Provo, Utah, 1994.

21. K. S. Klimaszewski and T. W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, Jan./Feb. 1997.

22. J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

23. H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–60, 1989.

24. G. Simiakakis. *Accelerating RayTracing with Directional Subdivision and Parallel Processing.* PhD thesis, University of East Anglia, October 1995.

25. 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. University of West Bohemia, Plzen, Czech Republic, February 1997.

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

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

28. A. Wilkie, R. F. Tobler, and W. Purgathofer. Orientation Lightmaps for Photon Radiosity in Complex Environments. To appear in *Proceedings of CGI 2000*, June 2000.

© 2001 Institute of Computer Graphics, Vienna University of Technology