Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Computational Geometry 34 (2006) 159–181

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

# Cost prediction for ray shooting in octrees [☆]

Boris Aronov, Hervé Brönnimann [*], Allen Y. Chang, Yi-Jen Chiang

*Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201, USA*

Received 7 January 2005; received in revised form 21 July 2005; accepted 27 September 2005

Available online 11 November 2005

Communicated by P. Agarwal

**Abstract**

The *ray shooting* problem arises in many different contexts and is a bottleneck of ray tracing in computer graphics. Unfortunately, theoretical solutions to the problem are not very practical, while practical methods offer few provable guarantees on performance.

Attempting to combine practicality with theoretical soundness, we show how to provably measure the average performance of any ray-shooting method based on traversing a bounded-degree spatial decomposition, where the average is taken to mean the expectation over a uniform ray distribution. An approximation yields a simple, easy-to-compute cost predictor that estimates the average performance of ray shooting without running the actual algorithm.

We experimentally show that this predictor provides an accurate estimate of the efficiency of executing ray-shooting queries in octree-induced decompositions, irrespective of whether or not the bounded-degree requirement is enforced, and of the criteria used to construct the octrees. We show similar guarantees for decompositions induced by *k*d-trees and uniform grids. We also confirm that the performance of an octree while ray tracing or running a radio-propagation simulation is accurately captured by our cost predictor, for ray distributions arising from realistic data.

© 2005 Elsevier B.V. All rights reserved.

*Keywords:* Ray shooting; Cost model; Cost prediction; Average performance; Octree; *k*d-tree; Uniform grid; Space decomposition

## 1. Introduction

Many practical problems encountered by algorithm designers and practitioners have the unfortunate property that, while the worst-case behavior of algorithms solving these problems is relatively easy to predict, the corresponding question for "real" or "typical" data sets is extremely difficult to answer. This gap between theory and practice is a well known source of problems, and some realistic input models have been developed to capture the geometric complexity of a scene; refer for instance to de Berg et al. [14] for an attempt to classify such models and their relationships in two dimensions. Some bounds have been obtained for geometric data structures and algorithms under these realistic

☆ Work on this paper has been supported by NSF ITR Grant CCR-0081964. Research of the first author has also been supported in part by NSF Grant CCR-9972568, the second author by NSF CAREER Grant CCR-0133599, and the fourth author by NSF CAREER Grant CCR-0093373 and NSF Grant ACI-0118915. A preliminary version of this paper appeared as B. Aronov, H. Brönnimann, A.Y. Chang, and Y.-J. Chiang, Cost prediction for ray tracing, in: Proc. 18th Annual ACM Symposium on Computational Geometry, 2002, pp. 293–302.

* Corresponding author.

*E-mail addresses:* aronov@cis.poly.edu (B. Aronov), hbr@poly.edu (H. Brönnimann), achang@cis.poly.edu (A.Y. Chang), yjc@poly.edu (Y.J. Chiang).

0925-7721/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.comgeo.2005.09.002

input assumptions [42] and references therein. The two main obstacles in this line of development, however, appear to be identifying the correct measures of problem complexity and designing algorithms which adapt naturally to the complexity of a problem instance.

*Ray shooting.* One such algorithmic problem is that of *ray shooting*. Given a set $\mathcal{S}$ of $n$ objects (the *scene*), we would like to ask queries of the following type: determine the first object of $\mathcal{S}$, if any, met by a query ray. The ray-shooting question lies at the heart of much of computer graphics (e.g., ray tracing [17] and radiosity techniques in global illumination [36] that produce photo-realistic images from three-dimensional models), as well as radio-propagation simulation [7] and other practical problems. In the context of ray tracing [17], rays are generated from the primary rays by reflections, refractions, visibility queries for light sources, etc. We are interested in the case where many ray queries will be asked for a fixed set of objects, so that it makes sense to preprocess the scene to facilitate queries. The goal is to beat the brute-force method that tests a ray against every object in the scene. Much work went into investigating the problem both in theory [30] and references therein, and in practice [6,11,20], and references therein.

On the one hand, most algorithms designed to improve the asymptotic worst-case performance usually introduce rather complicated data structures, and are either realistically unimplementable or too slow in practice. On the other hand, data structures used by practitioners to speed up ray shooting are not asymptotically faster than the brute-force method *in the worst case*. (Note that in this paper we do not consider attempts to analyze algorithms for "average" input scenes, as this approach represents a very different view of algorithm analysis.) Yet one observes that they behave much better in practice. The main problem in trying to explain this phenomenon is that the dependence of the algorithm behavior on the data set is not well understood. The "complexity" of a data set is difficult to define and evaluate. The total combinatorial object complexity (i.e., feature count) used traditionally in computational geometry is not a good parameter of the scene when it comes to ray shooting. Additionally, different data structures exhibit different behaviors on the same data set, and the selection of the right combination of data structure and algorithm for a given data set seems difficult.

*Our results.* A main theme of this paper is that the type of "optimization" that tries to make the worst case asymptotically efficient (in our case, in the sense of focusing on the best possible performance for a worst possible query ray in a worst possible scene) is actually irrelevant for the overall performance of the algorithm. Yet, this is no excuse to abandon the search for theoretically provable guarantees. Indeed, what matters in the overall performance is the *average* (as opposed to the worst-case) complexity of shooting a ray, in a *given* (as opposed to worst-case) scene.

One step towards understanding this average complexity would be the ability to accurately predict the performance of a specific data structure on a specific data set. In this approach, a cost function is attached to a data structure, that is intended to reflect the cost of shooting an "average" ray in a given scene using that data structure. The uses for such a predictor vary from simply estimating the running time on the given input, to choosing the right data structure, to fine-tuning the data structure to optimize its cost, to distributing the load in a parallel environment. There is a sizable body of work in the computer graphics literature about cost measures for some data structures. We attempt to summarize it in the next section.

Our starting point is a paper by Aronov and Fortune [4] that presents a proof that the average cost of traversing a bounded-degree space decomposition (they actually use a triangulation) compatible with the scene (see Section 2.1 for definitions) is proportional to the surface area of the decomposition. We note that minimizing the area of the decomposition is a heuristic (sometimes referred to as the "surface-area heuristic") that has been used in the graphics literature (as in [26], for example) and has its roots in integral geometry.

Following their approach, we propose a cost measure that is applicable to a large variety of data structures (i.e., to any "bounded-degree" spatial decomposition, not necessarily compatible; see Section 2.1) and we prove using integral geometry in Section 3.3 that this cost measure accurately predicts the average cost of shooting a ray by traversing the decomposition, assuming a "uniform" distribution of rays (see Section 3.1 for details about the ray distribution).

On the practical side, however, our theoretical guarantees do not quite meet all our requirements. For one, several data structures do not have bounded degree; this is for instance the case for decompositions induced by the leaves of an octree—although the degree is not bounded, it could still be shown that for the type of octrees we consider in this paper the average degree is constant; see, e.g., MacDonald and Booth [26]. Also, computing the theoretically provable cost measure, though not difficult, is still somewhat expensive, prompting us to propose a simplified cost measure, which we argue still models the traversal costs accurately in practice; see Section 4.3.1 for discussion. Finally, the distribution of rays in an actual ray-tracing process may differ from the assumed uniform distribution.

In this paper, we focus most of our experimental work on a somewhat old-fashioned data structure, namely an octree in which the split always occurs at the spatial median, since we felt that this provided a realistic if suboptimal data structure that had been widely used in practice in the past, which was at the same time easy to fit into our model and would give us an opportunity to evaluate the cost predictor in a relatively realistic context. Thus in Section 4, we concentrate on octrees. We experimentally confirm that ray traversal can be implemented directly on octrees in such a manner that its efficiency matches our cost predictions. A related cost measure was already proposed for octrees by MacDonald and Booth [26], with an additional term modeling the cost of using a hierarchy to traverse the octree-induced decomposition (see next section for a discussion of their approach and a comparison). Our verification is much more extensive, and thus complements and strengthens their result. In particular, we justify experimentally that their additional term is not necessary for an accurate prediction. We also extend this experimental confirmation to $k$d-trees and uniform grids in Sections 4.3.8 and 4.3.9.

In order to confirm that our cost predictor is independent of the manner in which the octree is built, we conduct experiments for various octree construction schemes. We further compare our predicted cost to the average cost of shooting a ray while rendering the image using a simple ray tracer. Our experiments indicate that the accuracy of our predictor does not depend too much on the ray distribution, for distributions produced by the ray-tracing and radio-propagation modeling processes.

The paper is organized as follows: After a review of relevant previous work in Section 2, we introduce our cost predictor and the main data structure investigated in this paper in Section 3, and in Section 4 we examine the validity of the assumptions we made to derive our predictor, and experimentally confirm its relevance to ray tracing and radio-propagation modeling.

## 2. Notation and previous work

### *2.1. Ray shooting using spatial decompositions*

In practice, a large number of variants of the same few data structures are used for storing a scene for ray shooting (read the surveys by Chang [11, Chapter 2] and Havran [20, Chapter 1]). None guarantees better than $\Theta(n)$ query time in the worst case (for the worst possible ray in the worst possible scene), which is clearly the cost of the brute-force algorithm. The data structures can be roughly classified into *space-partition*-based and *object-partition*-based. The former usually construct a space partition, whether hierarchical or flat, assign objects to those regions of the partition that they cross, and perform ray shooting by traversing the regions met by the ray. The latter enclose each object with an extent [12], i.e., a bounding volume, and organize them into a hierarchy which is not necessarily spatially separated. During ray-shooting queries, ray-object intersection tests are performed only if the extents are met by the ray. In this paper, we are only concerned with predicting the cost of ray shooting using space-partition-based structures; thus we will not elaborate on object-partition-based structures any further.

A *space decomposition* $\mathcal{T}$ is a partition of some volume (here the *bounding volume* $\mathcal{B}$ of the scene) into relatively open cells, faces, edges, and vertices. A space decomposition has *bounded degree* if each cell has a number of *neighbors*, i.e., cells sharing a two-dimensional portion of their boundary with it, that is bounded by a constant. Each cell is *simple* and *convex* if it is a convex polyhedron bounded by a small (fixed) number of planes. The bounded-degree requirement in fact implies "simplicity" if we insist that cells be convex. *Triangulations* are the quintessential examples of bounded-degree decompositions, as are (spatial) uniform grids. Decompositions of space induced by *octrees*, described in Section 3.4, do not necessarily have bounded degree, but those corresponding to "balanced" octrees do.

A cell $\mathcal{B}_i$ of a decomposition $\mathcal{T}$ can intersect a number of scene objects; let $\mathcal{S}_i$ denote the set of those objects and $|\mathcal{S}_i|$ denote their number. For a ray $r$ with origin $o(r)$, the algorithm for shooting $r$ is simple: first locate $o(r)$ in $\mathcal{T}$; that is, find the *originating cell* $\mathcal{B}_i$ that contains $o(r)$. If $o(r)$ falls on a cell boundary, pick any such cell. For rays originating outside the bounding volume $\mathcal{B}$, clip the ray to within $\mathcal{B}$, relocate the origin $o(r)$ to a new point on the boundary $\partial\mathcal{B}$ of $\mathcal{B}$, and proceed as above. If the ray misses $\mathcal{B}$ altogether, the problem is trivially solved. Otherwise, test $r$ against all the objects in $\mathcal{S}_i$. If $r$ intersects any object in $\mathcal{S}_i$ inside $\mathcal{B}_i$, then return the first such intersection along $r$. Otherwise, locate the cell of $\mathcal{T}$ that is next along $r$ after $\mathcal{B}_i$, and repeat until either an intersection has been found, or the ray leaves $\mathcal{B}$ (in which case the algorithm returns "no intersection found"). This process is called the *traversal* of $r$ through $\mathcal{T}$.

A decomposition is *compatible* with a set of object boundaries if every (open) cell avoids all object boundaries, and every (open) face or edge of the decomposition either lies completely in an object boundary or is completely disjoint from all object boundaries. This is the case for the triangulations considered by Aronov and Fortune [4]. The algorithm in that case is simplified: simply traverse $\mathcal{T}$ from cell to cell along $r$, starting from the cell that contains $o(r)$. While crossing a face or edge of a cell, check whether it belongs to an object boundary, if so return that object.

## 2.2. Cost measures for ray shooting in computer graphics

Previous work in studying the cost of ray shooting in computer graphics has been motivated mostly by its application to ray tracing. On the one hand, work within the computer graphics community has focused on improving the practical performance of ray shooting. On the other hand, work within the computational geometry community has focused on improving its theoretical performance. This research is discussed in this and the following sections.

We focus here on work which attempts to capture the performance of ray shooting via some cost measure. The potential uses of cost measures are many: one may simply want to estimate the rendering time for planning purposes (for instance, for load balancing in a parallel or distributed environment). One may also need to determine which data structure would be most efficient for processing a given scene. One may use cost-driven optimization when building the data structure; we describe such an application of our proposed cost measure in a companion paper [3].

The cost of performing ray shooting clearly depends both on the particular scene itself and on the data used. Global scene properties are commonly described by three factors: the object count, the object size, and the object locality. Object count is used widely in theoretical complexity analysis [30], especially for stating the worst-case behavior of an algorithm. In the ray-tracing literature, the size of the objects receives more attention than the object count, since experience shows that it has more impact on ray-tracing time [13,32,35]. To analyze statistical properties of a scene, Cazals and Sbert [10] enumerate several integral geometry tools. The average number of intersection points between a transversal line and the scene objects is used to measure the sparseness of the scene or the percentage of screen coverage [35]. For instance, the distribution of the lengths of rays in free space indicates where most of the objects are located, and was already proposed as the *depth complexity* by Sutherland et al. [41]. Cazals and Puech [9] demonstrate how to use integral geometry parameters of the scene to evaluate and optimize hierarchical uniform grids.

Integral geometry plays an important role in cost prediction for ray shooting. Its main use is through a surface area heuristic which, assuming a uniform line distribution, measures the quantity of lines crossing a region as proportional to the area of that region's boundary. (This assertion is mathematically valid if the region is convex.) Volume heuristics are also used (sometimes profitably), but do not rely on solid mathematical foundations. They have been used, for instance, to estimate the cost of BSP-trees [29] and uniform grids [13].

Applying area heuristics to object-partition schemes, Goldsmith and Salmon [18] construct bounding volume hierarchies with different types of extents in order to minimize the time of traversing the hierarchy. The trade-offs between competing factors of such hierarchies are studied and used in [6,18,40,43].

For space-partition schemes, the cost of ray shooting can be measured by the total amount of work while traversing a ray through the decomposition until the first object is hit. This total work is comprised of computing ray-object intersections and finding the next cell along a ray in the decomposition. Scherson and Caspary [35] discuss using several properties of object distributions such as object density within a small region, to analyze the cost of a ray traversal. The surface area has been used by Whang et al. [44] to estimate the traversal cost of octrees, and by MacDonald and Booth [26] for bintrees, which is an equivalent to octrees where every octree subdivision is done in three levels, in a way similar to Kaplan's BSP-tree [24].

In particular, the work of MacDonald and Booth is especially relevant here since their cost predictor is in essence the same as ours, for octrees. We note a few differences, though. While their cost measure is specifically developed for bintrees, we do show how to derive it as a simplification of a theoretically provable cost measure for a much larger category of spatial decompositions. They attempt to model the cost of neighbor-finding as non-constant, and use a heuristic approximation for this term, while we argue that this term is not necessary for an accurate cost prediction. Finally, our experimental study, while reproducing all of their experiments, is more extensive (although as mentioned above, we do not perform cost-based optimization in this paper; instead, we address this issue in a companion paper [3]). They do, however, investigate the optimization of the cost of the data structure by moving the splitting point.

Using a proposed cost function for optimizing the spatial decomposition is the next logical step, once the predictive power of the cost measure has been confirmed; in this paper we will focus on deriving and experimentally testing the accuracy of our cost function, while some applications are addressed in [3]. Both Whang et al. [44] and MacDonald and Booth [26] use their cost function to exploit the freedom in choosing the splitting plane, by discretizing the search space for the "optimal" split. In another approach, Reinhard et al. [32] propose to run a low-resolution ray-tracing preprocessing phase before the fully functional ray tracing starts. The cost function is used in the preprocessing to construct an octree whose leaf cells are further divided only if the cost keeps decreasing. Their cost function can also be used to estimate the number of rays that intersect an octree cell (as shown in [31]).

Recently, Havran et al. proposed a methodology (Best Efficiency Scheme—BES) to determine experimentally the most efficient method for a given set of scenes [20,21]. They construct a database such that one can render a given scene by finding a scene in the database with similar characteristics, and using the acceleration method that has been identified as the best one for that scene. This proposal rests on a few premises, which are tested in a companion paper [22]. The BES scheme is based on a large number of parameters of the scene, none of which dominates the others. The greatest merit of BES is that their cost function is able to compare data structures of difference kinds. The downside is that it relies on empirical tuning among a representative table of experiments. The range of data structures and parameters tested is quite extensive, and the conclusion is that no single data structure is best in all cases, although hierarchical ones tend to win over non-hierarchical ones, and that the surface area heuristic works very well for octrees and BSP trees. In his thesis, Havran [20] also introduces a fairly involved cost function for $k$d-trees, building on the work of MacDonald and Booth [26] with the same mathematical justification for the area heuristic. In particular, he introduces conditional probabilities based on ratios of areas, and uses them to weigh terms modeling such refinements as blocking factors, empty spaces, and preferred ray sets. He then attempts to reduce the cost of the constructed $k$d-tree, by choosing the position of the splitting plane and termination criteria; no optimality is claimed. This experimental study identifies, for each variant of the cost measure and automatic criterion, the deviation in runtime of ray tracing from a reference algorithm. The missing element of his (otherwise very complete) study is the relation of the cost model to the actual running time; this omission is justified as an attempt to gain machine- and compiler-independence. By contrast, we validate our cost model for octrees, no matter how good (or bad) the octree is.

### 2.3. *Improving the theoretical performance of ray shooting*

In this section, we focus on work that, in addition to being theoretically sound, with provable performance guarantees, proposes data structures that can be feasibly implemented in practice. This rules out a variety of theoretical algorithms constructed over the years (read surveys by Agarwal and Erickson [2], Pellegrini [30]) which successfully reduced the *worst-case* behavior of ray shooting, but at a cost: Ignoring the significant space requirements, these data structures are complicated and thus not too likely to be implemented. The constants hidden in the asymptotic notation are potentially large, so that the improvements in query time might only be visible in huge data sets (and even then, I/O issues and other phenomena would likely overshadow the algorithmic improvements). Hence we do not discuss further the sizeable literature aiming at reducing the worst-case analysis of ray shooting developed in the more theoretical geometric algorithms community.

It appears difficult, however, to obtain non-trivial (i.e., slower than $\Omega(\log n)$ query time) lower bounds on the ray-shooting problem—no such bounds in a general context are known. Recently, Szirmay-Kalos and Márton [39] established an $\Omega(n^4)$ lower bound in the decision tree model for the space needed by algorithms that perform ray shooting in worst-case logarithmic time. This contrasts sharply with results by Szirmay-Kalos et al. [38], which prove a constant expected time for several acceleration schemes, including $k$d-trees, octrees, regular grids, and ray classification, in a probabilistic scene model. Again, this is a very different view of algorithm analysis, and we do not consider it further. Instead, we focus on approaches that have provable performance guarantees (either worst-case or expected) but where the analysis is restricted to a single scene.

A conceptually simple (but realistic and implementable) paradigm for ray shooting considers the scene objects as obstacles, and triangulates a bounding volume thereof in a manner compatible with the obstacles. (In two dimensions, this paradigm essentially models many of the best proposed solutions to ray shooting in the computational geometry literature.) After the origin of the ray is located in the triangulation, the ray is traversed from tetrahedron to adjacent tetrahedron until a scene object is encountered and reported. As the triangulation is assumed to be face-to-face, i.e., a tetrahedron has only one neighbor across a face, the cost of traversal is simply proportional to the number of tetrahedra traversed by the ray before encountering an object of the scene. The maximum of this number over all rays is the *stabbing number* of the triangulation.

Agarwal, Aronov, and Suri [1], in an attempt to gauge the computational complexity of ray shooting, analyzed the maximum stabbing number of polyhedral scenes. They construct examples of scenes for which any triangulation has a high stabbing number, and present algorithms for building triangulations that are not too large and do not have a stabbing number much larger than the worst possible. This mostly settles the question of *worst-case* efficiency of triangulations for ray shooting.

The analysis in [1] is unsatisfying in that it focuses on the worst possible ray in the worst possible scene. In practice, worst-case scenes are of limited interest; for instance, scenes that occur in computer graphics usually have some properties which make them easier to manipulate. See the papers by de Berg et al. [14] and Vleugels [42] for a discussion of such properties. Perhaps more importantly, concentrating on the worst ray is even less realistic, as most applications of ray shooting and particularly ray tracing generate a huge number of rays, and it is highly unlikely that each ray will turn out to be worst possible.

Mitchell et al. [27] used a different approach to obtain theoretical guarantees. Instead of relying on the *size* of the scene as the main parameter to determine its complexity, they defined a different measure (*simple cover complexity*) which measures both how complicated the scene is and how difficult a particular ray query is, and present an algorithm that builds a hierarchical space partition with the property that any region is incident to a bounded number of other regions (cf. the discussion of bounded-degree decompositions below), and that the number of regions met by any ray is bounded by the simple cover complexity of the ray. Attempts to measure the simple cover complexity and to relate it to other parameters of a scene are reported in [14]. Since the simple cover complexity depends both on the scene and on the ray, this approach solves both problems mentioned above.

A different attempt to address these issues was made by Aronov and Fortune [4], who suggest that one should be concerned with the *average*, as opposed to worst-case, stabbing number of a triangulation. Using integral geometry, they prove that minimizing the average traversal costs induced by the triangulation reduces to constructing a triangulation compatible with the scene objects which minimizes the surface area. They then go on to construct a triangulation whose surface area is within a constant factor of the minimum possible, for any given scene. This solves both problems mentioned above: instead of only considering worst-case rays they focus on the average behavior, and the triangulation adapts to the particular scene at hand (instead of a worst-case scene) and its surface area measures the expected overhead of traversing it. In addition, unlike the simple cover complexity which applies to every ray individually, this measure is global and does not constrain any particular ray: this offers the data structure some freedom to be sub-optimal for a given ray if it improves the average behavior.

## 3. Cost model

In this section we explain the cost measure that we use for predicting the run-time behavior of our ray tracer. Our starting point is that it is possible, as shown by Aronov and Fortune [4], to predict the average cost of a ray traversal in a spatial decomposition using surface areas and integral geometry. Their model was given only for triangulations and is valid only for bounded-degree decompositions compatible with the obstacles. In this section we extend it to other, not necessarily compatible, decompositions. In order to discuss "average" behavior of a ray query, we need to agree on a ray distribution. This is the subject of the next section.

### *3.1. Distributions of lines and rays*

Underlying our entire discussion is a distribution $\mu_\ell$ of lines that corresponds to the rigid-motion-invariant (also known as the *uniform*) measure on the lines in space, restricted to the set $\mathcal{L}(\mathcal{B})$ of lines that meet the *bounding volume* $\mathcal{B}$ of the scene. For triangulations, the bounding volume is the convex hull of the scene. For an octree (or for simplicity, for any data structure or scene), a bounding box of the scene is also appropriate. In any case, we will assume that the bounding volume is convex.

The basic property of the uniform distribution $\mu_\ell$ is that the measure of the set of lines meeting a convex object $\sigma$ is proportional to the surface area $A(\sigma)$ of this object [34]. Moreover this measure is uniquely defined up to a constant of proportionality, which we choose such that the total measure of the line distribution equals the area of the bounding surface:

$$\int_{\mathcal{L}(\mathcal{B})} \mathrm{d}\mu_\ell = A(\mathcal{B}).$$

As for rays, we wish to consider only rays that originate and/or terminate on the objects (bounding box included). These are the rays of interest in most applications of ray shooting. For instance, primary rays originate from the camera and terminate either on an object or on the bounding surface; shadow rays are cast from points on the surface of an object towards light sources to determine whether they are visible or not.

Given a scene, each line is partitioned by the objects into several segments, each originating either on the bounding surface or on an object surface. By a *ray* $r$, we mean a combination of *origin* $o(r)$ (belonging to the union of the bounding surface and of the object surfaces) and *direction* $\vec{\ell}(r)$ (the oriented line supporting the ray). If the origin lies on $\partial\mathcal{B}$, the orientation is constrained so that the ray enters $\mathcal{B}$. If the origin lies on the surface of an object, there are two possibilities: the object can be *transparent* (we could also say *hollow*, or in the case of a two-dimensional object, such as a triangle, *two-sided*), or *opaque*. In the first case, both orientations are possible. In the second case, no ray originates inside or otherwise penetrates an opaque object, and so only one orientation is possible. The set of those rays, originating on the objects of the scene $\mathcal{S}$ (occasionally with opacity constraints) or on the bounding surface $\mathcal{B}$, is denoted by $\mathcal{R}$. We disregard rays or lines intersecting object edges, as they form a set of measure zero.

This motivates the following definition of *unified object area*: $A$ is the standard area for an opaque object (we treat $\partial\mathcal{B}$ as an inverted opaque object) and twice that area for a transparent object. Formally, this can be achieved by considering the origin $o(r)$ not as a point, but as a pair of a point and a vector normal to the surface at that point; for a transparent object, there are two opposite such normal vectors, and for an opaque object or for the bounding surface, there is only one choice of orientation for the normal vector. This allows us to treat a mix of opaque and transparent objects in a single scene with a single unified notation. From now on, we only consider rays in $\mathcal{R}$ (hence forbidding rays from entering opaque objects) and use the notion of unified object area when we refer to surface area.

Considering the interaction between rays and lines again, there are two processes for generating a random ray in $\mathcal{R}$. The first process consists of picking a line at random in $\mu_\ell$ and selecting with uniform probability any of the rays defined by this line and originating either on the objects or on the bounding surface, as long as it does not exit the bounding volume or enter an opaque object. Another process consists of picking an origin on an object or on the bounding surface with a probability proportional to its (unified) area, which also entails randomly picking an orientation for the normal vector at this point, and then a direction with a probability density given by

$$\mathrm{d}\mu_r = \cos\theta \, \mathrm{d}A_s \, \mathrm{d}\theta,$$

where $\theta$ is the angle between the normal vector to the surface $s$ at $o(r)$ with the direction $\vec{\ell}(r)$. Observe that $\theta$ is less than $\pi/2$ since $r$ belongs to $\mathcal{R}$. Note also that $dA_s$ is the Euclidean surface density, but because it will be summed for both orientations of the normal vector for transparent objects, its summation over $\mathcal{R}$ will yield the unified area of the object. Because of the presence of $\cos\theta$, this processes leads to a distribution for the supporting line of $r$ which is identical to $\mu_\ell$, while the distribution of $o(r)$ is the same as with the previous process. For a proof of these facts, the reader is referred to Santaló's book [34, Section 12.7, Eq. (12.60)].

Thus both processes induce the same ray distribution $\mu_r$, whose total measure is

$$\int_{\mathcal{R}} \mathrm{d}\mu_r = A(\mathcal{B}) + \sum_{s\in S} A(s),$$

where $A(s)$ is the unified area of object $s$. (This last equality is most easily seen with the second definition.) In other words, the total measure of the ray distribution equals the Euclidean surface area of the bounding surface, plus that of the opaque objects, plus twice that of the transparent objects. This measure depends solely on the scene, and not on the spatial decomposition.

## *3.2. Ray shooting in bounded-degree decompositions*

The cost of the ray-shooting algorithm outlined in the introduction, which traverses a spatial decomposition, depends naturally on the ray $r$. If we ignore the origin location cost (often, the originating cell is known), the cost of shooting $r$ is the sum of $|\mathcal{S}_i|$ over all the cells $\mathcal{B}_i$ of the decomposition encountered by the ray plus the traversal costs in $\mathcal{T}$; recall that $\mathcal{S}_i$ is the set of scene objects meeting a cell $\mathcal{B}_i$ of our decomposition. For bounded-degree decompositions, that latter cost is a constant per cell visited. Hence the cost of shooting a single ray $r$ by traversing $\mathcal{T}$ is proportional to

$$W_{\mathcal{T}}(r) = \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr),$$

where the summation is taken over all the cells $\mathcal{B}_i$ visited by the traversal along $r$ until the first intersection is found.

The cost of shooting a ray in a *compatible* decomposition is simply proportional to the number of cells crossed by a ray until it meets the boundary of an object. Over all the rays supported by a given line, it sums up as the number of object boundaries intersected by a line (transparent boundaries need to be counted twice). Since the probability distribution $\mathrm{d}\mu_r$ of all those rays sum up exactly as the probability density $\mathrm{d}\mu_\ell$ of the supporting line (according to our first random ray generation process), for compatible decompositions there is no need to distinguish between $\mu_\ell$ and $\mu_r$. Indeed the above discussion shows that the average number of cells crossed by a line between two consecutive encounters with objects, for $\mu_\ell$, is the same as the average number of cells crossed by a ray until it meets the boundary of an object, for $\mu_r$.

Since we are now extending the cost model discussion to potentially *non-compatible* decompositions, it will be important to distinguish the distributions $\mu_r$ and $\mu_\ell$.

### *3.3. The predictor for bounded-degree decompositions*

Consider a bounded-degree decomposition of a bounding box $\mathcal{B}$ of the scene into simple convex cells. The above discussion suggests that, given such a decomposition $\mathcal{T}$, we should define the *weighted work* of the decomposition in the following manner:

$$W^*(\mathcal{T}) = \int_{\mathcal{R}} W_{\mathcal{T}}(r)\,\mathrm{d}\mu_r = \int_{\mathcal{R}} \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr)\,\mathrm{d}\mu_r, \tag{1}$$

where the inner sum is over all the cells $\mathcal{B}_i$ traversed by $r$ until it meets the boundary of an object and $\mathcal{S}_i$ is the set of scene objects meeting $\mathcal{B}_i$. If we instead consider a single cell $\mathcal{B}_i$, the rays for which that cell will be involved in the inner sum belong to a subset $\mathcal{R}_i$ of $\mathcal{R}$, namely the set of rays entering $\mathcal{B}_i$ from outside or emanating from the surface of an object within $\mathcal{B}_i$. Exchanging the order of summation and integration in (1), we arrive at

$$W^*(\mathcal{T}) = \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr) \int_{\mathcal{R}_i} \mathrm{d}\mu_r = \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr)\,\mu_r(\mathcal{R}_i),$$

where the outer sum is over all the cells $\mathcal{B}_i$ of the decomposition. Note that only objects in $\mathcal{S}_i$ can be involved within $\mathcal{R}_i$. Thus $\mu_r(\mathcal{R}_i) = A(\mathcal{B}_i) + A(\mathcal{S}_i \cap \mathcal{B}_i)$ is the measure of rays entering $\mathcal{B}_i$ from outside or emanating from the surface of an object within $\mathcal{B}_i$. We obtain the expression for the weighted work

$$W^*(\mathcal{T}) = \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr)\bigl(A(\mathcal{B}_i) + A(\mathcal{S}_i \cap \mathcal{B}_i)\bigr), \tag{2}$$

where, as above, $\mathcal{B}_i$ is a cell in the decomposition, $\mathcal{S}_i$ is the set of scene objects (say, triangles) meeting $\mathcal{B}_i$, $A(\mathcal{B}_i)$ is the surface area of $\mathcal{B}_i$, $A(\mathcal{S}_i \cap \mathcal{B}_i)$ is the (unified) surface area of (the portions of) the objects within $\mathcal{B}_i$, and the summation is taken over all $\mathcal{B}_i \in \mathcal{T}$.

Dividing $W^*(\mathcal{T})$ in Eq. (2) by the total measure of the ray distribution, we get the *efficiency* of the decomposition, defined as the average of $W_{\mathcal{T}}(r)$ over $\mu_r$ by

$$E^*(\mathcal{T}) = \frac{W^*(\mathcal{T})}{A(\mathcal{B}) + \sum_{s \in \mathcal{S}} A(s)}, \tag{3}$$

where $A(\mathcal{B})$ is again the surface area of the bounding volume $\mathcal{B}$, and $A(s)$ is the (unified) surface area of object $s$.

To summarize, $E^*(\mathcal{T})$ measures the *quality* of the decomposition for the purpose of ray shooting (assuming the distribution of rays indeed follows $\mu_r$), namely, the expected amount of work required to trace an average ray until its first intersection with an object of $\mathcal{S}$ or with the boundary of $\mathcal{B}$. In practice, it fails to meet our requirement for simplicity due to the term $A(\mathcal{S}_i \cap \mathcal{B}_i)$, which while not unreasonably difficult to compute nonetheless is surprisingly expensive to evaluate and not much more precise than its simplification proposed below; see the discussion in Section 4.3.1. (Note that if we were tracing lines instead of rays, this term would disappear.) Instead, we introduce a *simplified* version of the *weighted work* $W(\mathcal{T})$, defined as

$$W(\mathcal{T}) = \sum_{\mathcal{B}_i} \bigl(1 + |\mathcal{S}_i|\bigr) A(\mathcal{B}_i). \tag{4}$$

In a sense, this simplified measure disregards rays that emanate from interiors of cells. In order to normalize the work we have computed, we still divide by $A(\mathcal{B}) + \sum_{s\in\mathcal{S}} A(s)$. This may appear a little odd, but is justified as follows: $W(\mathcal{T})$ measures the work performed during the line traversal (traversing through both cells and objects), while $A(\mathcal{B}) + \sum_{s\in\mathcal{S}} A(s)$ accounts for the "useful" portion of the work, namely the number of line-object intersections reported, integrated over $\mu_\ell$. In this sense, the ratio

$$E(\mathcal{T}) = \frac{W(\mathcal{T})}{A(\mathcal{B}) + \sum_{s\in\mathcal{S}} A(s)} \tag{5}$$

is a meaningful measure of the *quality* of the decomposition for the purpose of line-shooting. This is true in the sense that it measures the amount of work required for reporting a single "useful" intersection of an average *line* with the scene.

In the remainder of this paper, we aim to argue that $E(\mathcal{T})$ is a surprisingly good estimator of the behavior of *ray*-shooting algorithm, despite the following issues:

(i) The omission of the term $A(\mathcal{S}_i \cap \mathcal{B}_i)$ in the simplification from $E^*$ to $E$ leaves the rays originating on the objects unaccounted for. In particular, the secondary rays generated by refractions and reflections are not necessarily accounted for. We investigate this discrepancy in Section 4.3.1.

(ii) The ray distribution generated by an actual ray-tracing process can be quite different from the distribution $\mu_r$ we have postulated. Clearly, much depends on the scene, and it is possible to construct contrived scenes where the actual distribution will lead to quite different traversal costs from those predicted by $E(\mathcal{T})$ (see also our last remark below). Nevertheless, this does not seem to have much impact on the predictor for realistic scenes, as we argue in Section 4.3.2.

*Remarks.* The cost measure we have derived here has been kept at its simplest expression. In practice, especially when relying on the cost predictor to optimize a data structure, it might have to be fine-tuned. Firstly, the "unit" costs $\alpha$ of stepping from one cell of the decomposition to its neighbor, and $\beta$ of computing a ray-object intersection, may differ significantly, especially in scene with complex objects, and should be incorporated into the definition of work as follows:

$$W_{\alpha,\beta}(\mathcal{T}) = \sum_{\mathcal{B}_i} \bigl(\alpha + \beta|\mathcal{S}_i|\bigr) A(\mathcal{B}_i),$$

$$W^*_{\alpha,\beta}(\mathcal{T}) = \sum_{\mathcal{B}_i} \bigl(\alpha + \beta|\mathcal{S}_i|\bigr)\bigl(A(\mathcal{B}_i) + A(\mathcal{S}_i \cap \mathcal{B}_i)\bigr).$$

The parameter $\beta$ could even be extended to a function $\beta(\mathcal{S}_i)$ to take into account the differing costs of computing ray-object intersections for various types of objects. We take this into consideration in the companion paper [3].

Secondly, we have modeled the decomposition $\mathcal{T}$ as a flat, rather than hierarchical structure. Yet, to traverse the main decomposition that we focus on, namely, that induced by the leaves of the octree (cf. Section 3.4), we implement a uniform traversal algorithm. This algorithm does use the hierarchy, which would appear to violate our model; namely, neighbor-finding may no longer be a constant-time operation. In fact, the discussion of Section 4.3.3 shows that neighbor-finding via the hierarchy still takes constant time when the octree is balanced (i.e., the decomposition has bounded degree, cf. Section 3.4), and still does in an amortized sense if the octree is not balanced.

Lastly, addressing point (ii) above, for an actual ray distribution $\mu'_r$ differing from $\mu_r$, the analogous cost estimator could be derived: $E(\mathcal{T}) = \sum_{\mathcal{B}_i} (1 + |\mathcal{S}_i|)\mu'_r(\mathcal{B}_i)/\mu'_r(\mathcal{B})$, where $\mu'_r(\mathcal{B}_i)$ (resp. $\mu'_r(\mathcal{B})$) represents the measure of the set of rays involved in $\mathcal{B}_i$ (resp. the total measure of all rays). This would introduce additional calculation, as well as the problem of computing and representing the distribution $\mu'_r$. (Perhaps the ray classification of Arvo and Kirk [5] could be useful here.) The resulting model would be more accurate, especially if irregularities in ray distribution render the approximation by $\mu_r$ useless.

## *3.4. Octrees*

The above framework can be applied to any well-behaved decomposition of the scene, but in this paper we confine our attention mostly to decompositions that are induced by leaves of octrees, for various termination criteria. A *(split-at-the-spatial-median) octree* [33] is a hierarchical spatial decomposition data structure that begins with an axis-parallel bounding box of the scene—the root of the tree—and proceeds to construct a tree. A node (box) that does not meet the *termination criterion* is subdivided into eight congruent child sub-boxes by planes parallel to the axis planes and passing through the box center; hence the “split-at-the-spatial-median” name. The scene surface is modeled as a collection of triangles.

The triangulation of Aronov and Fortune [4] is constructed by first building an octree starting with a *cube* as a root (as opposed to, say, a minimal axis-parallel bounding box) with the following termination criterion: in that paper, a cut-off size is determined to make sure that the tree does not grow arbitrarily deep, and a tree box is subdivided if it is bigger than the cut-off size and if it meets any of the edges or vertices of the scene triangles. This octree is further refined to be *balanced* [28], i.e., so that no two adjacent leaf boxes are at leaves whose tree depths differ by more than one, where, as in a general space decomposition, two tree-node boxes are *adjacent to* or *neighboring* each other if two of their faces overlap. There are of course many other ways to construct octrees, and the aim is to quantify them (using our cost measure) for the purposes of cost prediction, comparison, and optimization.

To trace a ray, one descends the tree from the root to locate the ray origin among the leaves and then steps from leaf to leaf, checking all objects stored in the current leaf and proceeding to the next leaf using Samet’s table look-up for neighbor links [33, pp. 57–110]; we use the version of neighbor links as described by MacDonald And Booth [26].

In Section 4.1 we discuss a few variants of the octree construction that we consider to test the validity of our cost predictor. We demonstrate experimentally in Section 4.3 that our cost predictor is accurate.

## 4. Experimental evaluation

In order to evaluate the accuracy of our predictor in practice, we implemented an octree-construction algorithm based on the one described in Section 3.4. Our implementation allows us to build variations of the octree by incorporating various construction schemes. Once an octree is built, we estimate the ray-shooting cost per ray associated with that octree by computing our predictor; we also compare the cost of computing $E(\mathcal{T})$ and $E^*(\mathcal{T})$ and the relative accuracy of $E(\mathcal{T})$ and $E^*(\mathcal{T})$. We do this for three distribution of rays: random rays induced by the distribution $\mu_r$ (Section 4.3.1), quite extensively for a distribution induced by a ray tracing process (Sections 4.3.2 to 4.3.6), and another one arising from a radio propagation simulation (Section 4.3.7). The intent behind the random rays experiment is to investigate the relationship between $E^*(\mathcal{T})$ and $E(\mathcal{T})$, while the intent behind the other distributions is to validate our model for those particularly relevant applications, whose underlying distributions might differ from what we have assumed. More details are given in the beginning of Section 4.3. But first, we describe the data structures and test datasets.

### *4.1. Octree construction and traversal*

Our octree construction implementation is based on the scheme described in Section 3.4, with the flexibility of producing different variants of octrees by adjusting its construction criteria. These criteria include: a choice of a balanced or unbalanced octree and a choice of subdivision termination conditions. The top-down construction procedure of a subtree terminates when one of the following conditions is true: (1) the number of objects stored in a leaf node is less than or equal to a preset threshold value $j$; (2) the depth of the leaf node reaches a preset threshold value $l$; (3) the size of the leaf node is less than or equal to a fixed *cut-off size*, defined as the minimum size of a cell in the octree. Conditions (2) and (3) are equivalent and ensure that the octree construction always terminates, but are used differently: for (2), $l$ is a parameter adjusted to produce different octrees in the experiments for the same scene, but for (3), the cut-off size is *fixed*, to ensure that the octree does not become unreasonably large. Following Aronov and Fortune [4], we set the fixed cut-off size to be $b/m$, where $b$ is the size (i.e., maximum edge length) of the root bounding box, n is the number of objects, and $m = 2^{\lceil \log_2 n \rceil}$ is the first power of 2 greater than or equal to $n$. The threshold values, $j$ for (1) and $l$ for (2), are parameters to be adjusted in order to produce different octrees in the experiments. Note that in contrast with the octree constructed by Aronov and Fortune [4], in this paper we focus not on the number of *edges* and *vertices* meeting a cell, but on the number of *triangles* meeting a cell.

In addition, since the scheme described in [4] starts by enclosing the scene in an axis-aligned bounding *cube*, we want to test whether enclosing the scene in a minimal axis-aligned bounding *box* rather than a cube affects the accuracy of our predictor. Thus we also provide a choice between these two options. In the case of a box, the conditions (2) and (3) above are no longer equivalent.

Recall that a balanced octree is such that any two adjacent leaf boxes are at leaves whose tree depths differ by at most one. We first construct the octree without imposing the balancing requirement, and then perform an additional step to balance the octree, by employing the algorithm of Moore [28]. An octree is called "unbalanced" if we do not perform the additional step to balance it.

For any leaf node $b$, we store a pointer to its neighboring leaf node $t$ *only if* that leaf node has size greater than or equal to that of $b$; the same scheme is used for example by de Berg et al. [15] and by MacDonald and Booth [26]. If $t$ is smaller than $b$, we maintain in $b$ a pointer to the ancestor node of $t$ that is of the same size as $b$.

To traverse from a large leaf node to a small adjacent leaf neighbor, we follow the pointer to the neighbor's ancestor, and then descend the octree to the desired leaf node along a root-to-leaf path. If the octree is balanced, then we only go through at most one internal node before reaching the leaf neighbor (since the depths of neighboring leaf nodes differ by at most one), using O(1) steps. In this case, even though we do not explicitly store pointers to every adjacent leaf node, constant-time access to a neighbor effectively satisfies the bounded-degree assumption we used when formulating the definitions of $E(\mathcal{T})$ and $E^*(\mathcal{T})$ (see Section 3.3). On the other hand, if the octree is unbalanced, then such traversal costs may not be bounded, and thus the bounded-degree requirement is not enforced. By experimenting on both balanced and unbalanced octrees, we evaluate our cost predictor with and without imposing the bounded-degree requirement.

## 4.2. Test datasets

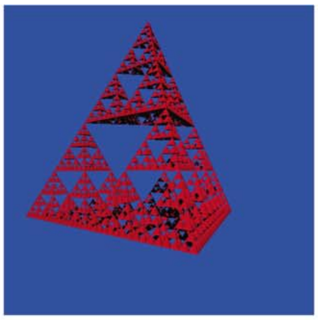
We evaluate our cost predictor using scenes drawn from the Standard Procedural Databases (SPD) [19] and the Stanford 3D Scanning Repository [37]. The SPD scenes include ten `tetra`, forty `teapot`, and seven `gear` scenes of various complexities. To these, we add nine `sphere` scenes, each an approximation of a mathematical sphere derived from recursive subdivision of a regular tetrahedron, with number of faces ranging from 16 to 1,048,576. The intent is that within a scene family, surface geometry stays the same with only the level of refinement of the subdivision being different, so we will be able to view the effect of varying the object count. Comparing among different scene families will indicate how the geometry affects our measurements. For the `tetra` (Fig. 1(b)) and `gears` (Fig. 1(c)) scenes, the geometry actually changes with the number of objects. Here we measure the effect of creating holes in the scene. The `horse` (Fig. 1(d)) scene obtained from Stanford University [37] provides an additional type of geometry for measuring the accuracy of our predictor.

In addition to these scenes, we study our cost predictor on *wall* models of architectural nature: the models of `lower-manhattan` (Fig. 1(h)) and `mid-manhattan` (not displayed), both of modest size less than 10,000 polygons, as well as of `rosslyn` (Fig. 1(e)), either with and without the underlying terrain, and two wall models (`crawford-hill` and `middletown`; Figs. 1(f) and 1(g)) of Bell Laboratories buildings, communicated to us by Steven Fortune. Each of the architectural scenes consists of a set of simple polygons such as rectangles, placed either horizontally or vertically, except for the terrain.

## 4.3. Experimental results

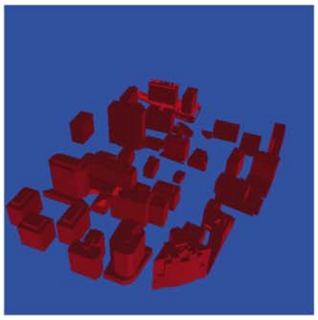
We performed our experiments on test datasets described in Section 4.2 where the number of triangles ranges from 4 to 1,087,716. We used various Sun Blade 1000 workstations running Solaris 8 and having 750MHz UltraSPARC III CPUs and up to 4GB of main memory, and HP workstations running Redhat 6.2 Linux with two processors and up to 2GB of main memory. For each dataset, we build an octree for every possible combination of the following

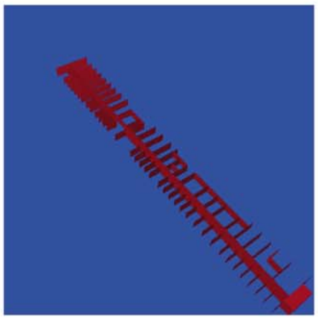
(a) `teapot` (2,292)

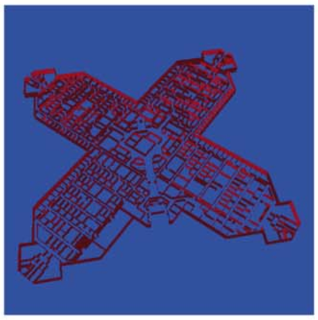

(b) `tetra` (4,096)

(c) `gears` (32,130)

(d) `horse` (96,966)


(e) `rosslyn` (2,467)


(f) `crawford-hill` (154)


(g) `middletown` (2,722)

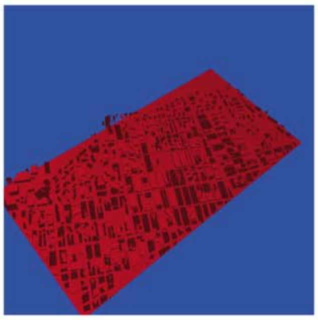

(h) `lower-manhattan` (6,826)

Fig. 1. Sample datasets (with number of triangles in the set).

options: (1) balanced vs. unbalanced (recall from Section 4.1 that "unbalanced" simply means that we do not perform the balancing step), (2) maximum number $j$ of objects allowed to reside in a leaf node ranging from 2 to 26, (3) maximum level $l$ of octree ranging from 1 to 17, and (4) the root box of the octree being a cube (*cube octree*) vs. being a minimum axis-aligned bounding box (*box octree*). The total number of combinations of datasets and octree variants we have tested is more than a thousand.

For each of the dataset-tree combinations, we compute the following quantities:

- $C_P$ (the *predicted cost*) is the average ray-shooting cost per ray using our predictor $E(\mathcal{T})$ given in Eq. (5);
- $C_E$ (the *explicit cost*) is the theoretically sound cost $E^*(\mathcal{T})$, computed explicitly from Eq. (3);
- $C_R$ (the *actual cost for random rays*) is the average ray-shooting cost per ray (i.e., the total number of ray-box intersection tests plus the total number of ray-object intersection tests, divided by the total number of rays) for random rays following the distribution $\mu_r$ of Section 3.1; and
- $C_A$ (the *actual cost for ray tracing*) is the average ray-shooting cost per ray computed as in $C_R$, but with rays arising from a ray tracing process.

We would like to see how close the predicted and the actual costs are, both in the setting of random rays and of ray tracing. We offer our analysis in Sections 4.3.1 to 4.3.6. In addition, we also ran a series of simulations on the architectural models for the *radio-propagation paths*. We report on these experiments in Section 4.3.7. For all these experiments, the complete results are too extensive to report in detail here, consult Chang's Ph.D. thesis [11] for the extensive study and further plots.

Finally, in addition to octrees, we also repeated some of the experiments on other spatial subdivisions, namely, $k$d-trees and uniform grids. We include a short report of these experiments in Sections 4.3.8 and 4.3.9. Further details can be found in Chang's Ph.D. thesis [11].

### *4.3.1. Comparison of $E$ and $E^*$ for uniform ray distribution*

The first and most important question we have to answer is the relation between the theoretically provable cost measure $E^*(\mathcal{T})$ given in Eq. (3) and our simplification $E(\mathcal{T})$ given in Eq. (5).

The computation of $E(\mathcal{T})$ is easy and we denote its result by $C_P$ (the *predicted cost*). There are two ways to compute $E^*(\mathcal{T})$. The first one is an explicit computation of the term $A(\mathcal{S}_i \cap \mathcal{B}_i)$ in Eq. (3). We apply Heckbert's polygon clipping algorithm [23]. We denote the result by $C_E$ (the *explicit cost*). The second one is a Monte Carlo approximation of Eq. (1), obtained by summing $W_{\mathcal{T}}(r)$ for many rays $r$ chosen randomly according to the distribution $\mu_r$; for each ray, we simulate the actual ray traversal and count the number of visited octree leaves as well as the number of the ray-object intersection tests. The total count divided by the number of rays is denoted by $C_R$ (the *actual cost for random rays*). The number of rays varies from 250 to 250,000.

We contrast these experiments with those of MacDonald and Booth[26, Section 3]—they work with random trees, random rays, and random scenes, which by their own admission are too sparse. In our case, the scenes are fixed and realistic, trees vary according to different construction criteria, while random rays are used to produce a Monte Carlo approximation to $E^*(\mathcal{T})$.

Perhaps the first thing that should be mentioned is that the actual cost for random rays $C_R$ converges towards the explicit cost $C_E$ when the number of rays increases. This was validated on all the scenes. Thus we may use either value in our experiments, whichever is more convenient.

The ratio of the time $t_E$ for computing $C_E$ over the time $t_P$ for computing $C_P$ ultimately depends on the implementation and many system parameters, and so is less precise than our subsequent cost measurements which involve only counting intersections and traversal costs. For this reason, we do not plot those quantities. Nevertheless, we found in our implementation that the ratio $t_E/t_P$ is stable and ranges between 90 and 230. In fact, it can be much higher for smaller scenes, up to almost a thousand. Some improvements may be achievable and may lower the computational cost of estimating $C_E$, yet it seems unlikely that the ratio can be brought to much lower than what we found. We can thus say that computing the simplified predictor $E(\mathcal{T})$ via $C_P$ is around two orders of magnitude faster than computing the theoretically accurate predictor $E^*(\mathcal{T})$ via $C_E$.

Due to the relatively high computational cost of computing $C_E$, we use $C_R$ instead of $C_E$ from now on. In fact, it is interesting to do so since in actual ray-tracing or radio propagation modeling processes, the actual cost is akin to a Monte Carlo computation, where the underlying ray distribution is dictated by the process.

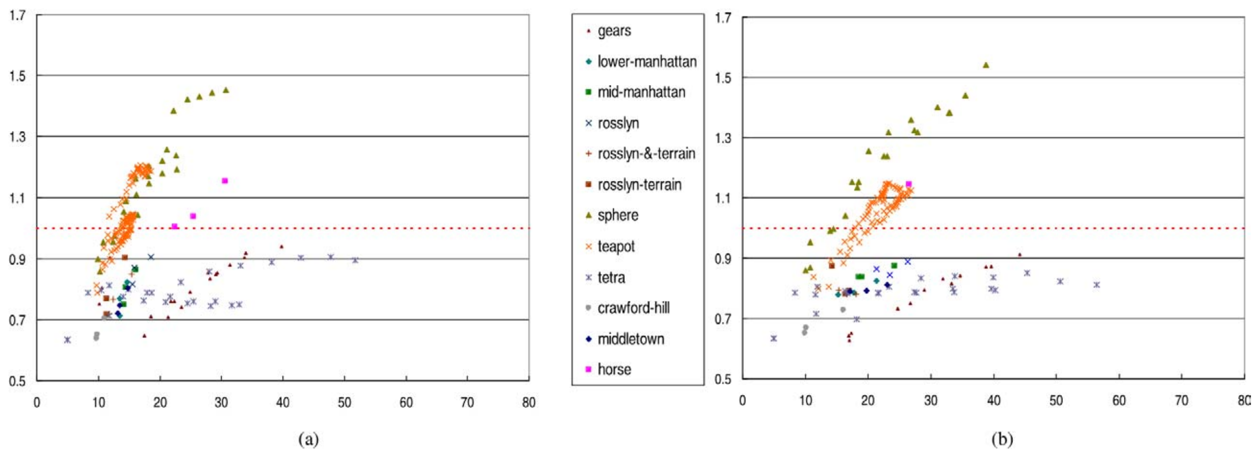


Fig. 2. Results for random rays: (a) Ratio $C_P/C_R$ ($y$-axis) for unbalanced octrees. The $x$-axis is the actual cost $C_R$. (b) The same quantities for balanced octrees.

Next, we address the question of accuracy. Having verified that $C_R$ converges towards $C_E$, the ratio $C_P/C_R$ of the cost $C_P$ given by our predictor to the actual cost $C_R$ for random rays should converge towards $E(\mathcal{T})/E^*(\mathcal{T}) = C_P/C_E$. If the ratio $C_P/C_R$ is close to one, then our simplified predictor $E(\mathcal{T})$ is close to the more complicated theoretical predictor $E^*(\mathcal{T})$. In Fig. 2(a), we plot this ratio $C_P/C_R$ as a function of the actual cost of random rays $C_R$ for unbalanced octrees. Fig. 2(b) shows this ratio for balanced octrees. We observe that the ratio lies between $0.625^{\pm 0.05}$ to $1.49^{\pm 0.04}$ for both balanced and unbalanced octrees. Thus the simplified predictor $E(\mathcal{T})$ is close to the theoretical predictor $E^*(\mathcal{T})$ no matter whether the octree is balanced or not.

We would also like to know how fast $C_P/C_R$ converges towards $E(\mathcal{T})/E^*(\mathcal{T})$, and how the underlying data structure affects such convergence. In Fig. 3, we plot $C_P/C_R$ as a function of the total number of random rays cast for a selection of scenes. For each scene, we compare box octree vs. cube octree, and balanced octree vs. unbalanced octree. We observe that the ratio depends on the scene and on how we construct the octree. For the `teapot` scene (Fig. 3(a)), the ratios of four different tree variants converge to four different limits. For `gears` (Fig. 3(b)), where many triangles are horizontal and the entire scene is flat, subdivision using different data structures does not affect the ratio too much. Most of the triangles in `middletown` (Fig. 3(c)) are vertical and the entire scene is flat; the ratios converge at the same pace irrespective of whether the octrees are balanced. The ratios for cube and box octrees differ distinctly, however. For that scene, we observe that all of the objects are vertical and the entire scene is flat. Using box octrees as a decomposition scheme, each cell contains many vertical objects, therefore forcing an excessive subdivision until the cut-off size is reached. This does not happen in cube octrees since the resulting octrees are quite balanced due to the placement of the splitting planes. For the `horse` scene (Fig. 3(d)), many triangles are vertical but the scene is tall. Although the ratios of box and cube octrees are different, the differences are not as much as those in the `middletown` scene. For all of these scenes, no matter how they converge, we observe a rather slow and steady convergence of the process in terms of the number of random rays cast in our experiments.

In conclusion, we have demonstrated that (i) $E(\mathcal{T})$ is an accurate approximation to $E^*(\mathcal{T})$ and to a Monte Carlo estimation of Eq. (1) when the random rays follow the distribution $\mu_r$ of Section 3.1, for a wide variety of scenes and data structures, and that (ii) $E(\mathcal{T})$ is easier to compute, by about two orders of magnitudes, than $E^*(\mathcal{T})$.

### 4.3.2. *Actual vs. predicted cost for ray tracing distributions*

For each of the dataset-octree combinations that we consider, we compute the predicted cost $C_P$ and run our ray tracer on the octree to render the scene and compute the actual cost $C_A$. We would like to evaluate the accuracy of our predictor by examining how close the predicted cost $C_P$ and the actual cost $C_A$ are under various circumstances. Recall that $j$ denotes the maximum number of objects allowed to reside in an octree leaf node, $l$ the maximum depth of an octree, and that we call an octree *cube octree* if its root bounding box $\mathcal{B}$ is a cube and *box octree* if $\mathcal{B}$ is a minimum axis-aligned bounding box.

In Fig. 4(a), we plot the ratio $C_P/C_A$ of the predicted cost $C_P$ to the actual cost $C_A$ as a function of $C_A$, for each family of the scenes and unbalanced box octrees with $j$ being 2, 5 and 10. The $y$-values measure how good our

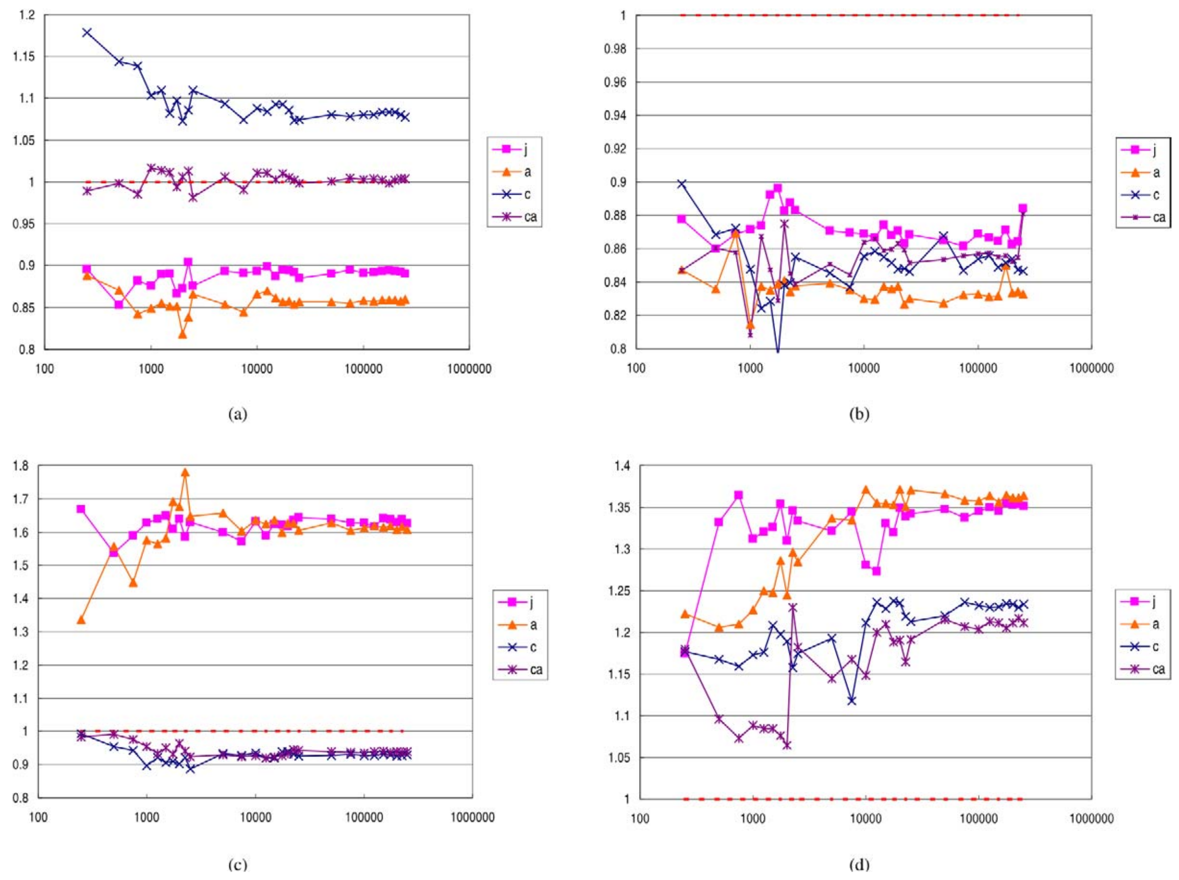


Fig. 3. Evolution of the ratio $C_P/C_R$ as a function of the number of random rays generated: (a) `teapot8`, (b) `gears2`, (c) `middletown`, and (d) `horse`. Explanation of the legends: `j`: box octrees constructed by restricting the maximum number $j$ of objects allowed in a cell; `a`: the same termination criteria as `j` on balanced box octrees; `c`: the same termination criteria as `j` on cube octrees; `ca`: the same termination criteria as `c` on balanced cube octrees.

predictor is, a ratio of one corresponding to a perfect prediction. Except for `crawford-hill`, for which the ratio lies between 4.26 and 4.37, we get predictions between 0.55 and 1.61 for all of the data sets. In Fig. 4(b) we plot the ratios for the same datasets but the underlying octrees are *balanced* box octrees. The distribution of the ratios is similar to that in Fig. 4(a), so octree balancing does not seem to affect the accuracy of our cost predictor. For some of the scenes such as `tetra` and `gears`, the underlying geometry of the object surfaces changes when the input size changes, while for the other scenes the geometry stays the same. In all these plots, we do not distinguish among different variations (i.e., different values of $j$ and $l$) of the underlying octrees for a given input, and the results show that the corresponding ratios $C_P/C_R$ do not vary much. In fact, for a single scene (e.g., `rosslyn`, `middletown`, or `teapot`), the accuracy of our cost predictor is remarkably independent of the underlying octree variations and of the scene sizes, so that the ratio seems to only depend on the scene geometry. For instance, the ratio is $0.94^{\pm 0.01}$ for `middletown`, it is $1.182^{\pm 0.007}$ for `rosslyn`, and in the range $0.81^{\pm 0.06}$ for `teapot1` to `teapot40`. The dataset `crawford-hill` seems to be less well-behaved for box octrees in terms of the ratio, but in fact the ratio for `crawford-hill` is $4.31^{\pm 0.06}$, which is consistent for a single scene even though the underlying octree structures are different. This indicates that the behavior of our predictor is very robust with respect to different octree structures.

In Figs. 4(c) and (d) we plot the same ratio for unbalanced and balanced *cube* octrees, respectively. To show their similarity, we use the same vertical scale as in the previous two figures, even though the high values of `crawford-hill` have disappeared in the cube version. As shown, there is relatively little difference between unbalanced and balanced octrees. Comparing with Figs. 4(a) and (b), we notice that the general trend is that the actual cost is usually a bit smaller in a cube octree, compared to the same scene with a box octree. This is true for most scenes, and is especially dramatic for `crawford-hill`: the ratio for *cube* octrees in (c) now becomes $0.403^{\pm 0.002}$, which is much

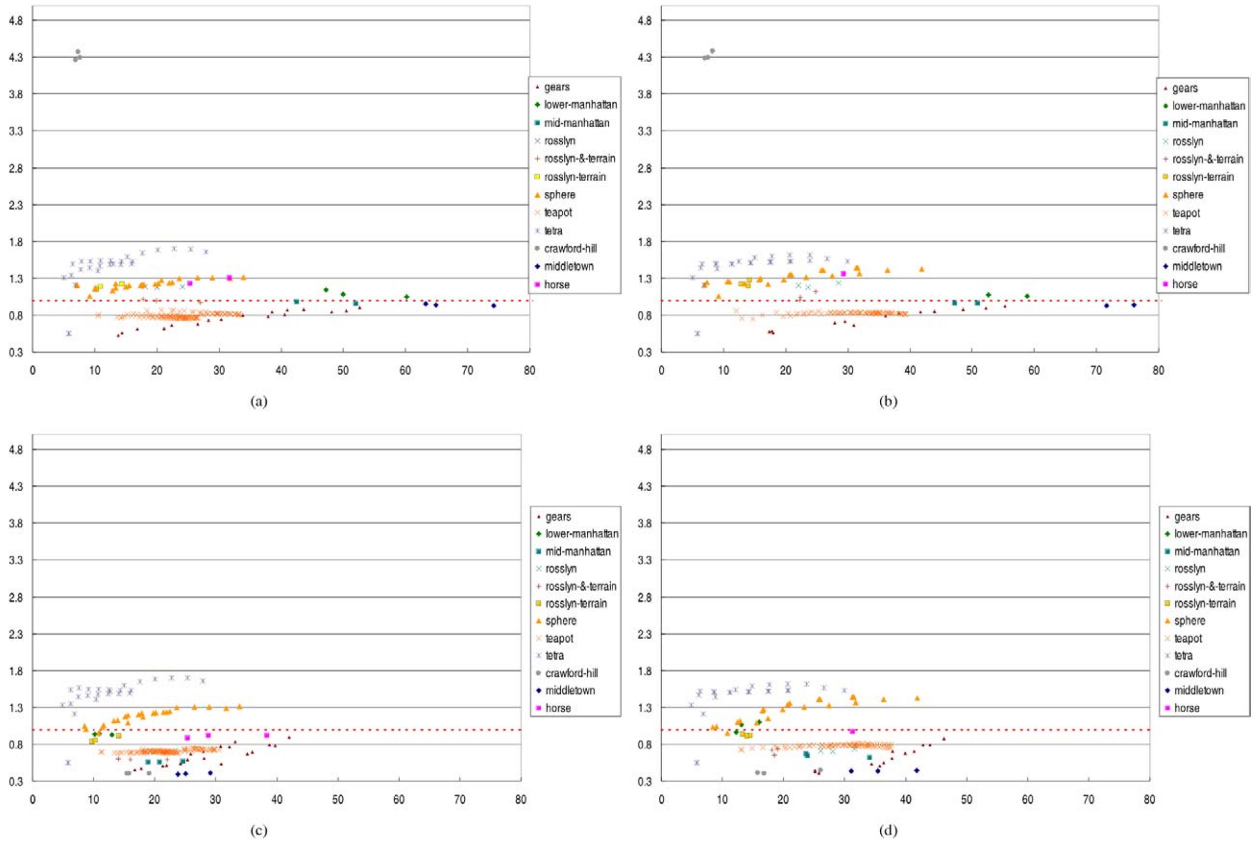


Fig. 4. Results for ray tracing: Ratio $C_P/C_A$ ($y$-axis). The $x$-axis is the actual cost $C_A$. (a) Unbalanced box octrees. (b) Balanced box octrees. (c) Unbalanced cube octrees. (d) Balanced cube octrees.

smaller than that for *box* octrees in (a) for the same dataset, thus the predictor becomes over-optimistic in such a scene with a cube octree. For scenes with similar bounding boxes, such as `middletown` and `gears`, the predictor becomes only a little too optimistic. For the other scenes, the bounding box has an aspect ratio closer to a cube, and the difference seems minimal.

The discrepancy between box vs. cube octrees in `crawford-hill` can perhaps be explained by the fact that, by the time the leaves adjacent to the ground have a height comparable with the scene, they already subdivide the horizontal space fairly well; thus the benefits of subdivision to the cost measure are mostly felt there. In contrast, when subdividing the (fairly flat) bounding box, we tend to get many objects in the same cell; indeed the horizontal subdivisions will cut almost all the vertical walls.

Beyond these simple observations, it is not really clear what happens here, except that the cube seems to produce lower and somewhat more accurate cost prediction, especially for scenes of an architectural nature. It could become overly optimistic for scenes such as `crawford-hill`. For other scenes, as we have said, the distinction between cube and box is anecdotal.

In conclusion, we observe that the fact that the rays come from a ray tracing process, as opposed to the uniform distribution $\mu_r$ as in the previous section, has little impact on the ability of our predictor to reflect the actual cost of the octree data structure, no matter what criteria are used to build it.

### 4.3.3. *Effects of balancing*

Since for a given scene, the ratios in Figs. 4(a) and (b) as well as in Figs. 4(c) and (d) do not vary much for different octree construction schemes, we want to see how much the trees constructed differ in structure; in particular, we want to see if balancing actually has any effect on the tree (after all, the tree could start off being nearly balanced) for our test data. Indeed, upon closer consideration we conclude that the effects of balancing on tree size vary widely, from not

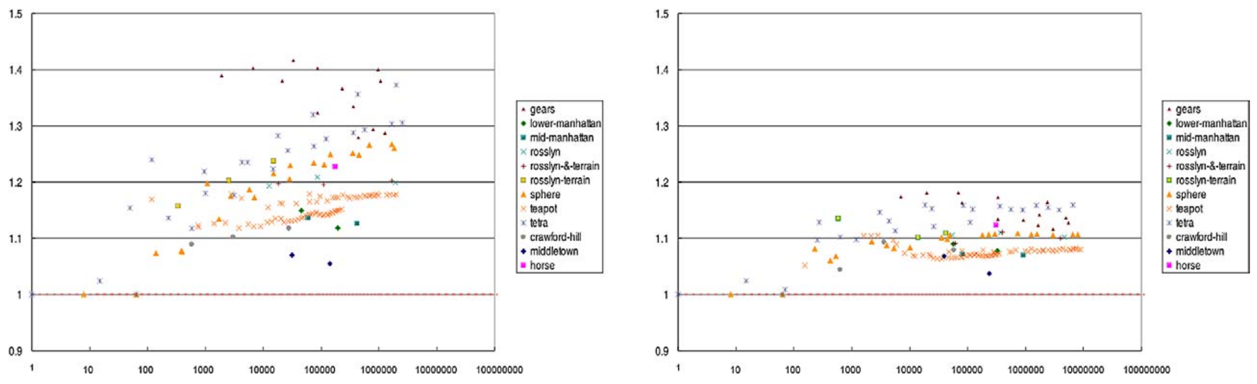


Fig. 5. Overhead of vertical motions for unbalanced (left) and balanced (right) octrees. The $y$-axis is the ratio of the total number of nodes traversed to the total number of leaf nodes traversed in running the ray tracer. The $x$-axis is the number of octree nodes in logarithmic scale.

affecting the tree size to increasing the number of leaves by a factor of 9.86. More specifically, out of the 152 box tree experiments covered by the plots in Fig. 5, we observed that the number of leaves in the tree has grown by less than a factor of two in 75 cases, by a factor of two to four in 47 cases, and by a factor of four to almost ten in the remaining 30. This seems to indicate that in many cases the tree structure is quite different after balancing. (Note in passing that Fig. 5 itself does not plot the number of nodes or leaves, but only the number of such nodes and leaves *traversed*, see discussion below.) As for the effect of balancing on the actual cost, out of 303 experiments (both with box and cube trees), the actual cost ratio between a balanced and its corresponding unbalanced tree ranged from 0.8 to 1.78, with 194 experiments having a ratio less than 1.27. Thus we conclude that, although balancing has the effect of increasing the tree size, it does not seem to change the actual cost significantly, let alone improve it. Since Figs. 4(b) and (d) show that the predicted cost reflects the actual cost quite well in the balanced case, we conclude that balancing does not seem to affect the predicted cost either.

Recall that in our ray-shooting queries, moving from a leaf to its neighboring leaf along the ray may require visiting some internal nodes of the hierarchy. We refer to such internal-node traversal as the *overhead of vertical motions*. In Fig. 5, we plot the ratio of the total number of nodes traversed, including internal and leaf nodes, to the total number of leaf nodes traversed, while running our ray tracer, against the number of octree leaves. We see that the ratio ranges between 1 and 1.41. Intuitively, the ratio should be upper-bounded by 1.5 for *balanced* octrees. Indeed, consider two adjacent leaf boxes. If they are at the same tree depth, then the ratio is one. If they are at different depths—necessarily differing by exactly one, going from the smaller to the larger box has unit cost, and going in the opposite direction has cost two since we need to go through one additional internal node. If rays of opposite directions are equally likely to occur, then two leaf neighbors of different depths contribute to a ratio value of 1.5, and therefore the overall ratio value is at most 1.5. As shown in Fig. 5, although the overhead for balanced octrees is noticeably lower than that for unbalanced octrees, the overheads for *both* cases are all between 1 and 1.5, i.e., all bounded by the intuitive upper bound of 1.5 for the *balanced* octrees, for a large set of sample rays from a typical ray-tracing computation in practice. This also experimentally justifies our assertion that one need not include this overhead into the cost predictor in this setting.

### *4.3.4. Random octrees*

To further verify that the quality of our prediction is not affected by the actual tree constructed, we took an extreme approach. For each given input, we built a *random* octree using the following subdivision-decision scheme. At the first two tree levels (levels 1 and 2), we always subdivide the node; for levels 3 and beyond, we compute the value $v = \mathit{rand} \cdot \mathit{level}$, and subdivide the node if $v > 2$, where *rand* is a random number chosen uniformly between 0 and 1. Finally, we always stop subdividing at level 8. Observe that the process does not depend on the objects in the scene! In particular, it is possible to subdivide an empty box. Again, we computed the predicted cost $C_P$ using our predictor, and ran our ray tracer to obtain the actual cost $C_A$, for each such random octree built. Since each run on the same input resulted in a different octree structure, we conducted 10 runs for each of the input datasets thus producing 10 data points per dataset.

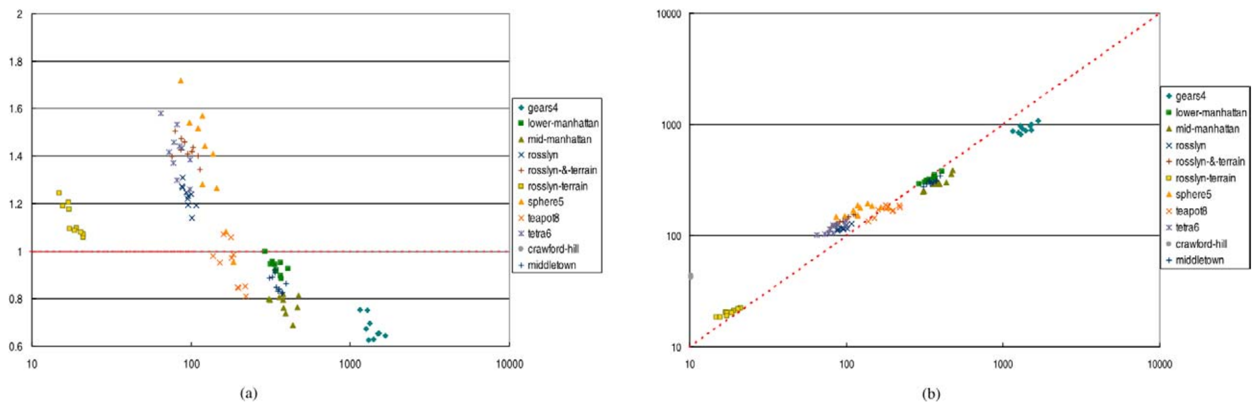


Fig. 6. Results for random octrees: (a) the ratio $C_P/C_A$ (the $y$-axis) vs. the actual cost $C_A$ (the $x$-axis); (b) the predicted cost $C_P$ (the $y$-axis) vs. the actual cost $C_A$ (the $x$-axis).

In Fig. 6(a), we plot the ratio $C_P/C_A$ of the predicted cost $C_P$ to the actual cost $C_A$, against the actual cost $C_A$, for the random octrees for some representative scenes. We see that the ratio values range from 0.62 to 1.71. Fig. 6(b) shows another view of how close the predicted cost (the $x$-axis) is to the actual cost (the $y$-axis). This shows that our predictor performs quite well even for random octrees. Notice that the actual cost for a randomly generated tree can be as high as nearly 1669, as opposed to less than 76 for "well-built" octrees shown in Figs. 4(a) and (b). Nevertheless, no matter how good or bad the actual cost is, we predict the actual cost quite accurately.

### 4.3.5. *Effects of the termination criteria*

As discussed above, constructing an octree for the *same scene* with different termination criteria results in different octrees as well as different costs for ray-shooting queries. We would like to know how the predictor reacts when the termination criteria change for the same scene. In Fig. 7 we compare the predicted cost $C_P$, the actual cost for random rays $C_R$, and the actual cost for ray tracing $C_A$, for a fixed dataset. The number of random rays is 10% of that of primary rays. Recall that $j$ is the maximum number of objects allowed to reside in a leaf node.

Fig. 7(a) shows the statistics of `teapot8` for unbalanced box octrees with $j$ ranging from 2 to 26. The actual cost $C_A$ increases smoothly when $j$ increases. Although the predicted cost $C_P$ is not exactly the same as $C_A$, the shapes of their curves match each other. Similarly, the curves of $C_P$ and of $C_R$ differ slightly in $y$-value but match in shape.

Fig. 7(b) shows the results of the same experiments for the `rosslyn` dataset using balanced box octrees. As $j$ increases, the actual cost $C_A$ first decreases to reach its minimum at $j = 9$ and then increases. The curve of the predicted cost $C_P$ matches the curve of $C_A$ exactly in shape. Fig. 7(c) shows another example of the same experiments, on `tetra6`, where both $C_A$ and $C_R$ increase in a staircase manner as $j$ increases. Again the curve of $C_P$ matches both curves of $C_A$ and $C_R$ in shape. The results of the same experiments on `crawford-hill` using balanced cube octrees are shown in Fig. 7(d). As we mentioned before, the predictor seems to be overly optimistic when the underlying structure is a balanced cube octree. Even though the curves of the predicted cost $C_P$ and of the actual cost $C_A$ are quite far apart, they behave in the same manner, i.e., $C_P$ and $C_A$ both go up or both go down at the same time as $j$ increases; so does the curve of $C_R$. All of the costs reach their minimum simultaneously, at $j = 6$.

Our experiments on other scenes show the same behavior as that in Fig. 7. Although the shape of the curves are scene-dependent, the predicted cost $C_P$ always moves along with the actual cost $C_A$, going up or down together. Comparing the predicted cost $C_P$ with the actual cost of random rays $C_R$, the movements of the two curves match for most of the time, with only a few exceptions. Notice that in Fig. 7(b), the value of $C_R$ at $j = 20$ is larger than that at $j = 21$, which does not match the behaviors of $C_P$ or of $C_A$.

Another termination criterion is the maximum level $l$ of an octree. Fig. 8 shows the results on how different values of $l$ affect the predicted and actual costs. Taking Fig. 8(a) (unbalanced box octrees for `sphere3`) as an example, in the very beginning, when $l = 1$, the actual cost $C_A$ is huge. As $l$ increases, $C_A$ drops dramatically until at some point $C_A$ becomes stable, and stays stable for the remainder of the plot. The curve shape of the predicted cost $C_P$ matches exactly with that of $C_A$, with the ratio $C_P/C_A$ ranging from 1.08 to 1.34. Fig. 8(b) shows results of balanced box octrees for `rosslyn-and-terrain`. The costs drop very fast and reach their minimum at $l = 6$, then increase

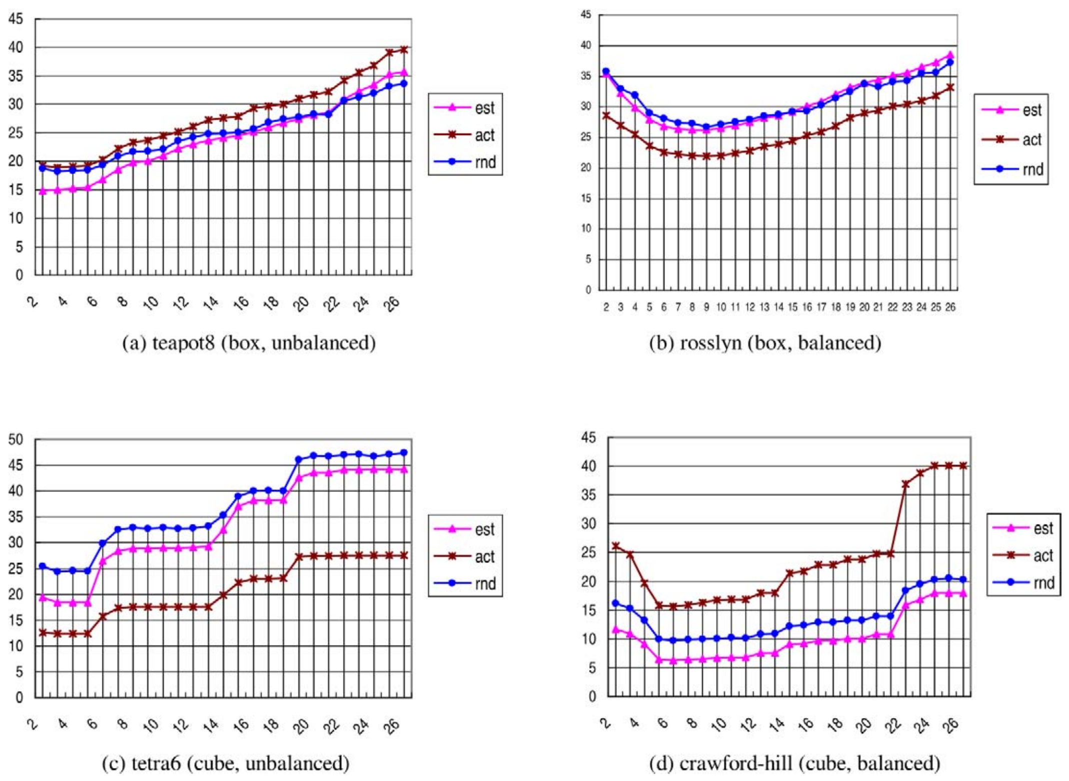


Fig. 7. Comparisons among the costs (the $y$-axis) $C_P$ (est), $C_R$ (rnd) and $C_A$ (act) for a fixed dataset with $j$ (the $x$-axis) ranging from 2 to 26: (a) `teapot8` using unbalanced box octrees; (b) `rosslyn` using balanced box octrees; (c) `tetra6` using unbalanced cube octrees; (d) `crawford-hill` using balanced cube octrees.

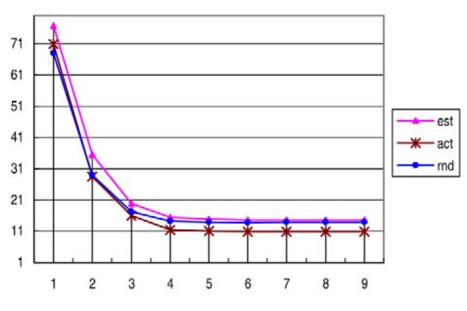
very slowly until the end of the plot. Figs. 8(c) and (d) show similar results for `horse` and `gears1` using unbalanced and balanced cube octrees, respectively. The results of Fig. 8 suggest a simple greedy approach to determine whether an octree cell is worth subdividing or not in an attempt to build a cost-optimal octree: if the predicted cost, as measured by $E(\mathcal{T})$, does not decrease enough or shows signs of increasing, then one could conclude that further subdivision is unnecessary and stop splitting the current cell immediately. Some results in this direction have been obtained in [3,8].

In conclusion, as long as the parameters used to build an octree remain in a reasonable range, we see little variation in the predicted or actual costs, and whatever variation there is always correlated. A maximum depth of at least 8 and a separation criterion $j = 6$ seem the best general-purpose choices.

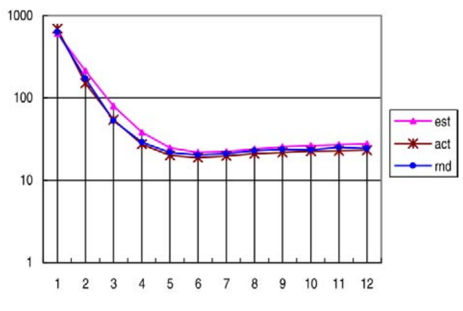
### 4.3.6. *Effects of the geometric resolution*

All of the above experiments confirm that our cost predictor is accurate irrespective of the different octree structures resulting from various termination conditions. There is still one more question to investigate: How does the predictor react for scenes of the *same* underlying geometric object surfaces but approximated with different number of triangles, using octrees constructed with *fixed* termination conditions?

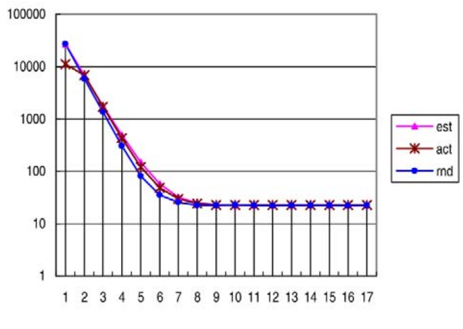
In Fig. 9(a), we plot the results of the `teapot` scene with the number of triangles ranging from 58 to 105,280, using unbalanced cube octrees with $j = 10$. The overall trend is that the actual cost increases as the number of triangles increases, but the cost seems to oscillate in the middle section of the plot. Notice how the predicted cost oscillates simultaneously with the actual cost. The curve of the actual cost for random rays $C_R$ almost matches the other two curves in shape except for the smallest scene; this is because $C_R$ is calculated based on random rays, and thus it is not expected to work well on very small samples. Fig. 9(b) shows the results of the same tests using balanced cube octrees. The curve of the actual cost is smoother than that in (a), and so is the curve of the predicted cost. These results show that the accuracy of our cost predictor is not sensitive to different levels of the scene complexity.



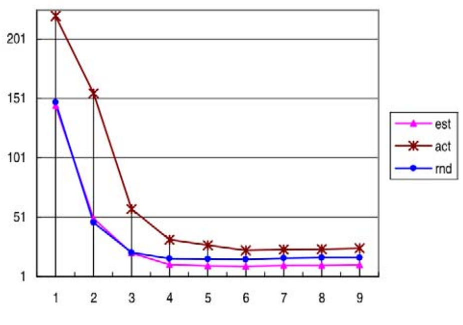
(a) sphere3 (box, unbalanced)



(b) rosslyn-and-terrain (box, balanced)

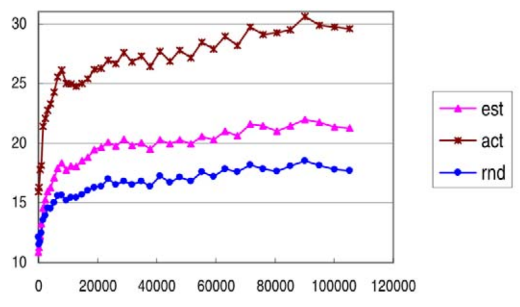


(c) horse (cube, unbalanced)

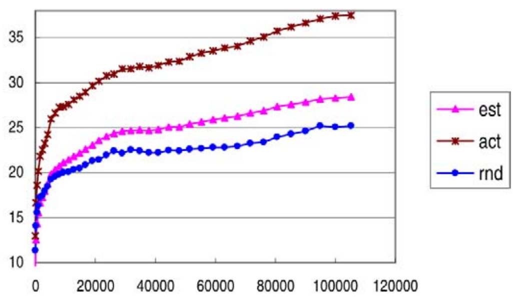


(d) gears1(cube, balanced)

Fig. 8. Comparisons among the costs (the $y$-axis) $C_P$ (est), $C_R$ (rnd) and $C_A$ (act) for a fixed dataset with $l$ (the $x$-axis) ranging from 1 to up to 17: (a) `sphere3` using unbalanced box octrees; (b) `rosslyn-and-terrain` using balanced box octrees; (c) `horse` using unbalanced cube octrees; (d) `gears1` using balanced cube octrees.



(a) teapot (cube, unbalanced)



(b) teapot (cube, balanced)

Fig. 9. Comparisons among the costs (the $y$-axis) $C_P$ (est), $C_R$ (rnd) and $C_A$ (act) for the `teapot` scene with the number of triangles (the $x$-axis) ranging from 58 to 105,280: (a) unbalanced cube octrees with $j = 10$; (b) balanced cube octrees with $j = 10$.

### *4.3.7. Another application: radio-propagation simulation*

A radio propagation path, or path for short, is a directed polygonal chain consisting of straight-line *segments*. We perform ray-shooting queries and line-segment intersection queries arising from paths generated by realistic experiments using a real-world radio-propagation prediction system (WISE [25]; the data are courtesy of Steven Fortune). We treat the path data the same as ray-tracing data, i.e., each path is a sequence of reflections of the primary ray. As we do not record the signal strength, power and path loss are not considered, as they would in an actual simulation run. When a path encounters the surface of a wall, it may pass through the wall or rebound along the direction determined by the law of mirror reflection. Each path terminates at one of the receivers, which are placed uniformly at grid positions in the scene.

We tested our cost predictor on four architectural models (`lower-manhattan`, `rosslyn`, and `crawford-hill`, as well as a similar `learning-center`). The results in Allen Chang's thesis [11] show that the predicted costs $C_P$ is consistently between 1 and 2.5 times the actual cost per ray, and between 0.2 and 2 times the actual cost per segment. These results indicate that our predictor appears to be accurate for ray-shooting queries arising in ratio-propagation simulations.

### *4.3.8. Additional experimental results with kd-trees*

A $k$d-tree is a spatial subdivision similar to an octree [33], in which each node's sub-box is obtained from that of its parent by a single subdivision along one of the directions $x$, $y$, or $z$. Our $k$d-tree is constructed in a top-down recursive manner similar to the octree construction described in Section 4.1, except that we never perform balancing. The termination criteria for the $k$d-tree are the same as for the octree, i.e., the top-down construction stops when the number of objects in the leaf is less than some threshold, or the depth of the leaf is larger than some other threshold. The cut-off size is defined the same as for the octree, i.e., $b/m$ where $b$ is the maximum side length of the root bounding box, $m = 2^{\lceil \log_2 n \rceil}$, and $n$ the number of objects in the scene.

There are two more factors to be considered when constructing a $k$d-tree. The first is the position of the splitting plane, and the second is its direction. As with the octree, we choose a split at the spatial median. For the direction, we always split along the longest dimension of a cell. If the bounding box is a cube, this results in the cyclic split, alternating between $x \to y \to z$ axes.

As for traversal, we associate a neighbor link to each face of a cell, as outlined in [20, Appendix D], and use it much like for the octree. Note that, as with octrees, the neighbor links may not point to a leaf and so there may be an effect due to what we dubbed the overhead of vertical motions.

As with octrees, we study the ratio $C_P/C_A$ of the predicted cost $C_P$ to the actual cost $C_A$ in a ray tracing process. For most of the test sets and box $k$d-tree, the ratio lies between 0.5 to 1.5 except for `crawford-hill` whose ratio lies between 3.24 to 3.62. With a cube $k$d-tree, all test sets have a ratio in the range 0.25 to 1.5 (including `crawford-hill`). Thus the behavior of our predictor is almost identical in the context of octrees and $k$d-trees. For more discussion, refer to the discussion in Section 4.3.2.

We did not implement balanced $k$d-tree to enforce the bounded degree decomposition. Nevertheless, our experiments show that using neighbor links in $k$d-trees results in the same traversal costs as in a bounded degree decomposition, i.e., a ray traversing the $k$d-tree from one leaf node to its neighbor node visits a constant number of internal nodes on average.

### *4.3.9. Additional experimental results with uniform grids*

A uniform grid is a very simple example bounded degree decomposition. Our cost model should be applicable to predicting the cost of ray shooting on a uniform grid. We implemented a uniform grid following the details given by Fujimoto and Iwata [16]. The implementation is straightforward, with an option to choose the grid size manually. If one does not specify the grid size, the program tries to find the best grid size by itself using Eq. (3.18) of [11, p. 108]. Traversing the uniform grid is based on 3DDDA incremental method which uses only integer operations [16].

As in our experiments on octrees and $k$d-trees, we would like to confirm that the ratio $C_P/C_A$ of the predicted cost to the actual cost is bounded by a constant. Our experiments include all of the test scene families for octree and $k$d-tree tests. For each of the scenes, we set the number of grid cells on each side to be 0, 2, 4, 8, 16, and 32. The value 0 is for automatic grid generation, as mentioned previously. As for octrees and $k$d-trees, `crawford-hill` gets the highest ratio $C_P/C_A$, between 4.26 to 4.83, on box uniform grid. There is no such highest ratio for cube uniform grid. In any case, the ratios of the predicted cost and the actual cost are in the range of 0.24 to 2.38, except for `crawford-hill`, which shows the correctness of our predictor on uniform grids.

## 5. Conclusion

In this paper, we have devised a cost measure which predicts the average cost for ray shooting. The predictor assumes that the algorithm traverses a bounded-degree spatial decomposition (i.e., the number of neighbors of a cell in the decomposition should be bounded by a constant). The first cost measure we propose provably reflects the traversal costs for a random ray following a certain ray distribution, but is expensive to compute. The cost measure we advocate is a particularly simple and appealing adaptation which, although it does not provably relate to the cost of the traversal in general, intuitively should be close to the provable measure. In particular it is provably within a constant factor of it for decompositions whose leaves intersect only a constant number of objects (a common termination criterion).

We have experimentally verified the accuracy of our cost predictor for a set of scenes on a several types of spatial decompositions, mainly induced by octrees. There are many construction schemes for octrees, involving various termination criteria, and these octrees can be made into bounded-degree decompositions by a balancing process. Unbalanced octrees are used commonly in ray-tracing applications. They do not necessarily have bounded degree. Yet we experimentally confirm the accuracy of our predictor for those data structures as well. We also extended our experiments to consider other decompositions ($k$d-trees and uniform grids), and found again very similar results. Finally, in our experiments we have observed that the bias in the ray distribution introduced by an actual ray-tracing process does not substantially affect the precision of our cost predictor.

We conclude by mentioning a few directions for further research. We begin with the problem of computing data structure that is guaranteed to have cost within a constant factor of the optimal cost over all possible octrees. This was achieved for compatible triangulations by Aronov and Fortune [4] (where the cost measure is simply the surface area), but the octree underlying their construction, built with the criterion mentioned in Section 3.4, is not known to have a cost within a constant factor of the optimal with respect to our cost measure. Very recently, Brönnimann and Glisse [8] have given such an algorithm, based on a greedy termination criterion with a certain amount of lookahead. Some results and experiments are given by Aronov et al. [3] and Brönnimann and Glisse [8].

Secondly, although we concentrate on ray shooting, and assume a "uniform" distribution of rays derived from a rigid-motion-invariant distribution on lines, in an actual ray-tracing process these assumptions clearly do not hold due to the geometry of ray tracing and various illumination phenomena. We have experimentally verified the accuracy of our predictor for various commonly encountered scenes, but some contrived scenes may force our predictor to differ wildly from the actual cost of ray tracing. Therefore it would be an interesting direction for future research to (i) capture and represent a distribution for a given scene and illumination parameters, and (ii) build an octree which is optimized for a particular distribution. (See last remark of Section 3.3.) Lastly, it is intriguing that our cost predictor is insensitive to whether or not the octree has been balanced. While Brönnimann and Glisse [8] have recently shown that balancing cannot increase the cost by more than a constant factor, the experimental results indicate little or no change to the cost. There may be some scenes (constructed with the knowledge of the unbalanced octree) for which the balancing process substantially changes the cost, but we conjecture it would take very artificial scenes to achieve that.

## Acknowledgements

We wish to thank Steven Fortune of Bell Laboratories for providing some of the test scenes, as well as for several fruitful discussions. We thank Eric Haines for the Standard Procedural Databases (SPD). The `horse` dataset is courtesy of Stanford Computer Graphics Laboratory. Many thanks also go to an anonymous reviewer of a preliminary version of this article who greatly helped in improving it.

## References

[1] P.K. Agarwal, B. Aronov, S. Suri, Stabbing triangulations by line 3D, in: Proc. 11th Annu. ACM Symp. Comput. Geom., ACM, New York, 1995, pp. 267–276.

[2] P.K. Agarwal, J. Erickson, Geometric range searching and its relatives, in: B. Chazelle, J.E. Goodman, R. Pollack (Eds.), in: Advances in Discrete and Computational Geometry, Contemporary Mathematics, vol. 223, American Mathematical Society, Providence, RI, 1999, pp. 1–56.

[3] B. Aronov, H. Brönnimann, A. Chang, Y.-J. Chiang, Cost-driven octree construction schemes: an experimental study, Computational Geometry 31 (2005) 127–148.

[4] B. Aronov, S. Fortune, Approximating minimum weight triangulations in three dimensions, Discrete Comput. Geom. 21 (4) (1999) 527–549.

[5] J. Arvo, D. Kirk, Fast ray tracing by ray classification, in: Proc. Computer Graphics (SIGGRAPH '87), ACM, New York, 1987, pp. 55–64.

[6] J. Arvo, D. Kirk, A survey of ray tracing acceleration techniques, in: A. Glassner (Ed.), An Introduction to Ray Tracing, Morgan Kaufmann, San Francisco, CA, 1989, pp. 201–262.

[7] H.L. Bertoni, Radio Propagation for Modern Wireless Systems, Prentice-Hall, Upper Saddle River, NJ, 2000.

[8] H. Brönnimann, M. Glisse, Cost-optimal trees for ray shooting, in: Proc. Latin American Symp. Theoretical Informatics, in: Lecture Notes in Computer Science, vol. 2976, Springer, Berlin, 2004, pp. 349–358, Computational Geometry, in press.

[9] F. Cazals, C. Puech, Bucket-like space partitioning data structures with applications to ray tracing, in: Proc. 13th Annu. ACM Symp. Comput. Geom., ACM, New York, 1997, pp. 11–20.

[10] F. Cazals, M. Sbert, Some integral geometry tools to estimate the complexity of 3d scenes, Research Report no. 3204, INRIA, July 1997.

[11] A.Y. Chang, Theoretical and experimental aspects of ray shooting, Ph.D. Thesis, Computer and Information Science Department, Polytechnic University, Brooklyn, NY, 2004. Available at http://photon.poly.edu/~hbr/publi/ayc_thesis.pdf.

[12] J. Clark, Hierarchical geometric models for visible surface algorithms, Comm. ACM (October 1996) 547–554.

[13] J. Cleary, G. Wyvill, Analysis of an algorithm for fast ray tracing using uniform space subdivision, Visual Comput. 4 (1988) 65–83.

[14] M. de Berg, M. Katz, F. van der Stappen, J. Vleugels, Realistic input models for geometric algorithms, in: Proc. 13th ACM Symp. on Comput. Geom., ACM, New York, 1997, pp. 294–303.

[15] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, Springer, Berlin, 1997.

[16] A. Fujimoto, K. Iwata, Accelerated ray tracing, in: Computer Graphics: Visual Technology and Art (Proc. Computer Graphics, Tokyo'85), Springer, New York, 1985, pp. 41–65.

[17] A. Glassner (Ed.), An Introduction to Ray Tracing, Morgan Kaufmann, San Francisco, CA, 1989.

[18] J. Goldsmith, J. Salmon, Automatic creation of object hierarchies for ray tracing, IEEE Comput. Graph. Appl. 7 (5) (1987) 14–20.

[19] E. Haines, Standard procedural database (SPD), Version 3.13, 3D/Eye, 1992. Home page at http://www.acm.org/tog/resources/SPD/overview.html.

[20] V. Havran, Heuristic ray shooting algorithms, Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, November 2000. Available at http://www.cgg.cvut.cz/~havran/phdthesis.html.

[21] V. Havran, J. Bittner, J. Prikryl, The best efficiency scheme project proposal—version 1.90jp, April 1st 2000. Available at http://www.cgg.cvut.cz/GOLEM/proposal2.html.

[22] V. Havran, J. Prikryl, W. Purgathofer, Statistical comparison of ray-shooting efficiency schemes, Technical Report TR-186-2-00-14, Czech Technical University in Prague, Czech Republic, 2000.

[23] P. Heckbert, Generic convex polygon scan conversion and clipping, in: A. Glassner (Ed.), Graphics Gems, Academic Press, Boston, 1990, pp. 84–86.

[24] M.R. Kaplan, The use of spatial coherence in ray tracing, in: R.A. Earnshaw, D.E. Rogers (Eds.), Techniques for Computer Graphics, Springer, Berlin, 1987, pp. 173–193.

[25] S. Kim, B. Guarino, T. Willis, V. Erceg, S. Fortune, R. Valenzuela, L. Thomas, J. Ling, J. Moore, Radio propagation measurements and prediction using three-dimensional ray tracing in urban environments at 908 MHz and 1.9 GHz, IEEE Trans. Vehicular Technol. 48 (1999) 931–946.

[26] J.D. MacDonald, K.S. Booth, Heuristics for ray tracing using space subdivision, Visual Comput. 6 (1990) 153–166.

[27] J. Mitchell, D. Mount, S. Suri, Query-sensitive ray shooting, Int. J. Comput. Geom. Appl. 7 (3) (1997) 317–347.

[28] D. Moore, Simplicial mesh generation with applications, Ph.D. Thesis, Department of Computer Science, Cornell University Ithaca, NY, 1992.

[29] B. Naylor, Constructing good partitioning trees, in: Proc. Graphics Interface '93 Canad. Inf. Proc. Soc., Toronto, 1993, pp. 181–191.

[30] M. Pellegrini, Ray shooting and lines in space, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, CRC Press LLC, New York, 1997, pp. 599–614.

[31] E. Reinhard, A. Kok, A. Chalmers, Cost distribution prediction for parallel ray tracing, in: Proc. 2nd Eurographics Workshop Parallel Graphics and Visualization, Eurographics, 1998, pp. 77–90.

[32] E. Reinhard, A. Kok, F. Jansen, Cost prediction in ray tracing, in: X. Pueyo, P. Schröder (Eds.), Proc. Eurographics Workshop, Rendering Techniques '96, Porto, Portugal, Springer, Berlin, 1996, pp. 41–50.

[33] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, Cambridge, MA, 1990.

[34] L. Santaló, Integral Probability and Geometric Probability, Encyclopedia of Mathematics and its Applications, vol. 1, Addison-Wesley, Reading, MA, 1979.

[35] I. Scherson, E. Caspary, Data structures and the time complexity of ray tracing, Visual Comput. 3 (4) (1987) 201–213.

[36] F.X. Sillion, C. Puech, Radiosity and Global Illumination, Morgan Kaufmann, San Francisco, CA, 1994.

[37] Stanford University, Stanford 3D Scanning Repository, 2002. Home page at http://www-graphics.stanford.edu/data/3Dscanrep.

[38] L. Szirmay-Kalos, V. Havran, B. Balász, L. Szécsi, On the efficiency of ray shooting acceleration schemes, in: Proc. Spring Conf. Computer Graphics, Comenius University, Bratislava, Slovakia, 2002, pp. 89–98.

[39] L. Szirmay-Kalos, G. Márton, Worst-case versus average case complexity of ray-shooting, J. Computing 61 (2) (1998) 103–133.

[40] K. Subramanian, D. Fussell, Automatic termination criteria for ray tracing hierarchies, in: Proc. of Graphics Interface '91, Canad. Inf. Proc. Soc., Toronto, 1991, pp. 3–7.

[41] I. Sutherland, R. Sproul, R. Schumacker, A characterization of ten hidden surface algorithms, ACM Comput. Surv. 6 (5) (1974) 1–55.

[42] J. Vleugels, On fatness and fitness—realistic input models for geometric algorithms, Ph.D. Thesis, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1997.

[43] H. Weghorst, G. Hooper, D. Greenberg, Improved computational methods for ray tracing, ACM Trans. Graph. 3 (1) (1984) 52–69.

[44] K.Y. Whang, J.W. Song, J.W. Chang, J.Y. Kim, W.S. Choand, C.M. Park, I.Y. Song, Octree-R: An adaptive octree for efficient ray tracing, IEEE Trans. Visual. Comput. Graph. 1 (1995) 343–349.