Cost Prediction for Ray Shooting*

Boris Aronov aronov@poly.edu
Hervé Brönnimann hbr@poly.edu
Allen Y. Chang achang@cis.poly.edu
Yi-Jen Chiang yjc@poly.edu

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

Abstract

The *ray shooting* problem arises in many different contexts. For example, solving it efficiently would remove a bottleneck when images are ray-traced in computer graphics. Unfortunately, theoretical solutions to the problem are not very practical, while practical solutions offer few provable guarantees on performance. In particular, the running times of algorithms used in practice on different data sets vary so widely as to be almost unpredictable.

Since theoretical guarantees seem unavailable, we aim at obtaining a simple, easy to compute way of estimating the performance without running the actual algorithm. We propose a very simple cost predictor which can be used to measure the average performance of any ray shooting method based on traversing a bounded-degree space decomposition.

We experimentally show that this predictor is accurate for octree-induced decompositions, irrespective of whether or not the bounded-degree requirement is enforced. The predictor has been tested on octrees constructed using a variety of criteria.

This establishes a sound basis for comparison and optimization of octrees. It also raises a number of interesting and challenging questions such as how to construct an optimal octree for a given scene using our cost predictor.

Since the distribution of rays in a ray-tracing process may differ from the rigid-motion invariant distribution of lines and the corresponding distribution of rays assumed by our cost predictor, we also experimentally confirm that the performance of an octree for an actual ray-tracing computation is well captured by our cost predictor.

*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.

SoCG'02, June 5-7, 2002, Barcelona, Spain.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*; I.3.7 [**Computer graphics**]: Three-Dimensional Graphics and Realism—*raytracing*

General Terms

Algorithms, experimentation

Keywords

Ray shooting, cost model, cost prediction, octree, space decomposition, average performance

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," "typical," or "average" 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 try and capture the geometric complexity of a scene (see for instance [11] 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 input assumptions [30, and refs. therein]. It is much harder, however, to design algorithms which adapt naturally to the complexity of a problem.

One such algorithmic problem is that of *ray shooting*. Given a set $\mathcal{S}$ of n objects, 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 and radiosity techniques in global illumination to produce photo-realistic images from 3D models), as well as radio-propagation simulation and other practical problems. In the context of ray tracing [12], 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 given set of objects, so that it makes sense to preprocess them to facilitate queries. The goal is to beat the brute-force method that compares a ray to every object in the scene. Much work went into investigating the problem both in theory [22, and refs. therein] and in practice [5,8, and refs. therein].

Unfortunately, theoretically efficient algorithms are not practical, due to large constants hidden in the asymptotic analysis or very complicated data structures aiming at making the worst case efficient, which we argue is irrelevant. On the other hand, data structures used to speed up ray shooting by practitioners are not asymptotically faster than the brute-force method in the worst case. 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 behavior of the algorithms on the data set is not well understood. The "complexity" of a data set is difficult to define and evaluate. The total 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 data structure and the right algorithm for a given data set seems difficult.

One step towards understanding this behavior would be the ability to predict the performance of a specific data structure on a specific data set. Some heuristic models have been developed for some data structures; we summarize them in the next section. In this paper, we focus on models with theoretical guarantees. Our starting point is a paper by Aronov and Fortune [3], whose theoretical analysis supports a heuristic that has been used in the graphics literature, namely, that the average cost of traversing a subdivision compatible with the scene[1] is proportional to the surface area of the subdivision (also called its *weight*). They construct (an approximation of) the minimum-weight Steiner triangulation for a scene made up of triangles.

Even though the focus of [3] is on triangulation as the data structure for ray shooting, their triangulation is constructed by building an octree and subsequently triangulating it. The triangulation step introduces additional complexity (both in the constants and in the difficulty of implementation); we instead consider the octree directly. Here one difficulty arises, namely, octrees cannot be made compatible with non-axis-aligned objects, and hence we must allow scene objects to intersect subdivision-cell interiors. For this purpose, we extend the surface area criterion of [3] into a cost predictor that takes into account, in addition to the number of octree cells traversed, the fact that every object intersecting a cell must be tested against a ray as this ray enters the cell.

We attempt to confirm experimentally that ray traversal can be implemented directly on octrees in such a manner that its efficiency matches our cost predictions. In order to verify that our cost predictor is independent of the manner in which the octree is constructed, we conduct experiments for various octree construction schemes.

Although this cost predictor is based on a ray distribution induced by a rigid-motion invariant distribution for lines, we further compare our predicted cost to the average cost of 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 the distributions produced by the ray-tracing process.

[1] A space subdivision is a decomposition of some volume (here the bounding box of the scene) into relatively open cells, faces, edges, and vertices. A subdivision is *compatible* with a set of object boundaries (here modeled as a set of triangles) if every cell avoids all object boundaries, and every face or edge of the subdivision either lies completely in an object boundary or is completely disjoint from all object boundaries.

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 analyzed 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.

2. Previous Work

In practice, it appears that a large number of variants of the same few data structures are used for storing a scene for ray shooting [8]. None guarantees better than $\Theta(n)$ query time in the worst case, which is clearly the cost of the naïve 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 different regions of the partition, and perform ray shooting by traversing the regions met by the ray. The latter encloses each object with an *extent* [9], i.e., a bounding volume, and organizes them into a hierarchy which is not necessarily spatially separated. During ray shooting query, ray-object intersection tests are performed only if the extents are met by the ray.

Previous work in studying the cost of ray tracing was motivated by two major goals. Firstly, given a scene, one would like to know which data structure is the most efficient for this scene. Secondly, given a data structure, one would like to fine tune the it to adapt to the scene in the preprocessing phase, so as to run as fast as possible in the rendering phase. We distinguish between work with and without theoretical guarantees.

Without theoretical guarantees. Most of the work described in this subsection is heuristic and based on computing various parameters of the input which affect the running time of the ray-tracing process.

Global scene properties can be captured by three factors: the object count, the object size, and the object locality. Object count is used widely in theoretical complexity analysis [22], 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 research shows that it has more impact on ray-tracing time [10, 24, 27]. The size of an object can be measured either by its surface area or its volume.

Applying area heuristics to object-partition schemes, Goldsmith and Salmon [13] construct bounding volume hierarchies (BVH) with different types of extents in order to minimize the time of traversing the hierarchy. The trade-offs between the competing factors of BVH are studied and used in [5, 13, 28, 31].

For space-partition-based structures, Scherson and Caspary [27] discuss several properties of the object distribution such as the object density within a small region, in order to analyze the cost of a ray traversal. To examine the global scene, Cazals and Sbert [7] enumerate several integral geometry tools to analyze statistical properties of a scene. 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 [27]. This approach is closest to offering theoretical guarantees since it relies on proven mathematical results. 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. [29]. Cazals and Puech [6] demonstrate how to use these parameters to evaluate and optimize hierarchical uniform grids (HUGs).

As for estimating traversal costs, the surface area is used to estimate the traversal cost of octrees [32] and *bintrees* [18], which is a three-level octree similar to Kaplan's BSP-tree [17]. Recognizing that a provably optimal result is out of reach, in both papers an approximate solution to the optimal splitting plane is obtained by discretizing the search space, which leads to costly computations.

Another approach to predicting the cost is to run a low resolution ray-tracing phase before the fully functional ray tracing starts. The cost function is used to monitor the low resolution ray tracing phase, as proposed by Reinhard et al. [24]. They 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 in an octree cell [23].

Volume heuristics are also used to estimate the cost of BSP-trees [21] and uniform grids [10]. In the latter context, the *augmented volume* of an object is used, which is the sum of the volumes of grid cells where the object resides.

Recently, Havran et al. proposed a methodology (Best Efficiency Scheme—BES) to determine experimentally the most efficient method for a given set of scenes [15]. 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 [16]. The range of data structures and parameters tested is quite extensive, and the conclusion is that no single one 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. The BES scheme is based on many parameters of the scene, none of which dominates the others.

With theoretical guarantees. In this section, we focus on work that, in addition to being theoretically sound, proposes data structures that can be feasibly implemented in practice.

Various theoretical algorithms constructed over the years [2, 22] were successful in reducing 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, plus the constants hidden in the asymptotic notation are potentially large, so that the improvements in query time would 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 on ray shooting and worst-case analysis developed in the more theoretical algorithms community.

It appears difficult, however, to obtain lower bounds on the ray-shooting problem—no such bounds in a general context are known. 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. This paradigm essentially models many of the 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. The worst possible scene is of theoretical interest in that it shows how good a statement may be made about the decomposition without further assumptions on the scene. Nevertheless, scenes in computer graphics usually have some properties which make them easier to manipulate. See [11, 30] for a discussion of such properties. Thus the worst-case scene is usually totally irrelevant for the actual ray tracing. Concentrating on the worst ray is unrealistic as well, 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.

In [19], Mitchell, Mount, and Suri 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 complicated 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 [11]. 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 [3], who suggest that one should be concerned with the *average*, as opposed to worst-case, stabbing number of a triangulation. They start by considering "line-shooting" queries, in which a line is traced through a triangulation, stepping from tetrahedron to adjacent tetrahedron, and all line-object intersections are reported. The work expended is proportional to the number of triangles met by the line and the "useful" work is the number of objects encountered (for the purposes of this discussion the surface of a bounding volume such as the convex hull or a bounding box is considered an object surface). Both quantities are integrated over all lines, using the rigid-motion-invariant measure on the lines. Since the measure of the set of lines meeting a triangle in 3-space is proportional to its area, the two integrals yield the total area of the triangulation (its *weight*) and that of the objects, respectively. The ratio of the two areas is taken as a measure of the quality

of the triangulation—it is the expected amount of work required to report a single line-object intersection. (The analysis is carried out for lines but the result holds as well for a ray distribution that chooses the ray origin uniformly on the surface of the objects and biases the direction toward the surface normal, see the discussion below.) Thus minimizing the average traversal costs induced by the triangulation reduces to constructing a triangulation compatible with the scene objects which minimizes the surface area. The paper [3] then goes on to construct such a triangulation whose surface area is within a constant factor of the minimum possible, for any given scene. This solves both the problem of only considering the worst-case rays—average behavior is considered here—and the problem of aiming for worst-case scenes—the triangulation adapts to the scene and its surface area measures the expected overhead of traversing it, for an average line. 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 [3], to predict the average cost of a ray traversal in a space decomposition using surface areas and integral geometry. Their model is valid only for subdivisions compatible with the obstacles, and in this section we extend it to other subdivisions, especially octrees.

3.1 Distributions of lines and rays

Underlying our entire discussion is a distribution μ_ℓ of lines that corresponds to rigid-motion invariant measure on the lines that meet the *bounding surface* of the scene. For triangulations, the bounding surface is the convex hull of the scene. For simplicity or for an octree, a bounding box of the scene is also appropriate. The total measure of the line distribution is proportional to the surface area of the bounding surface [26].

As for rays, 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*, we mean a combination of *origin* (belonging to the union of the bounding surface and of the object surfaces) and of a *direction* (the oriented line supporting the ray). The distribution μ_ℓ of lines induces a distribution μ_r of rays as follows [26, section 12.7, eq. (12.60)]: the origin is chosen uniformly at random from the union of the bounding surface and the surfaces of the objects, and with probability density for the direction proportional to the cosine of the angle between the surface normal and the direction of the ray. The total measure of the distribution is proportional to the surface area of the objects plus that of the bounding surface. This measure depends solely on the scene, and not on the space decomposition.

For space decompositions *compatible* with the scene objects (such as triangulations considered by Aronov and Fortune [3]), there is no need to distinguish between μ_ℓ and μ_r, since indeed the average number of cells crossed by a line between two consecutive encounters with objects, for μ_ℓ, is the same as the average number of cells crossed by a ray until it meets the boundary of an object, for μ_r. Since we are now extending the cost model discussion to potentially *non-compatible* decompositions, it will be important to distinguish the distribution μ_r from μ_ℓ.

3.2 The predictor for bounded-degree decompositions

Consider a bounded-degree decomposition of a bounding box of the scene into simple convex cells. A space decomposition is *bounded-degree* if each cell has a bounded number of neighbors and each cell is *simple* and *convex* if it is a convex polyhedron bounded by a small number of planes (the bounded-degree requirement in fact implies "simplicity" if we insist that cells be convex). The above discussion suggests that, given such a decomposition $\mathcal{T}$, we should define the *weighted work* of the decomposition in the following manner:

$$W^*(\mathcal{T}) = \sum_{\mathcal{B}_i} (1 + |\mathcal{S}_i|) \times (A(\mathcal{B}_i) + A(\mathcal{S}_i \cap \mathcal{B}_i)) \qquad (1)$$

where $\mathcal{B}_i$ 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 $\mathcal{B}_i$ and $A(\mathcal{S}_i \cap \mathcal{B}_i)$ is the surface area of the objects within $\mathcal{B}_i$.

Indeed, the measure of rays entering $\mathcal{B}_i$ from outside or emanating from the surface of an object within $\mathcal{B}_i$ is $\mu_r(\mathcal{B}_i) = A(\mathcal{B}_i) + A(\mathcal{S}_i \cap \mathcal{B}_i)$. For each such ray, besides the unit cost of crossing $\mathcal{B}_i$, we need to test the intersection of the ray with all the objects in $\mathcal{S}_i$, thus contributing an additional cost of $|\mathcal{S}_i|$. So the weighted work contributed by $\mathcal{B}_i$ is $(1 + |\mathcal{S}_i|)\mu_r(\mathcal{B}_i)$. Dividing $W^*(\mathcal{T})$ in Equation (1) by the total measure of the ray distribution, we get the *efficiency* of the decomposition, defined by

$$E^*(\mathcal{T}) = \frac{W^*(\mathcal{T})}{A(\mathcal{B}) + \sum_{s \in \mathcal{S}} A(s)}, \qquad (2)$$

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

In theory, $E^*(\mathcal{T})$ provably and accurately measures the *quality* of the decomposition for the purpose of ray shooting (assuming the distribution of rays indeed follows μ_r), namely the average amount of work required to trace a ray until its first intersection with an object 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 is complex to compute. (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

$$W(\mathcal{T}) = \sum_{\mathcal{B}_i} (1 + |\mathcal{S}_i|) \times A(\mathcal{B}_i).$$

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 easily 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 μ_ℓ. In this sense, the ratio

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

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 simplification from E^* to E leaves some rays unaccounted for. In an actual ray-tracing process, the primary rays originating from the camera are all accounted for, as are the rays from the light source to the objects. However, the secondary rays generated by refractions and Lambertian 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 rigid-motion-invariant one we have assumed. Clearly, much depends on the scene, and it is not hard to present contrived scenes where the actual distribution will lead to quite different traversal costs from those predicted by $E(\mathcal{T})$. Yet we find in Section 4.3.2 that there is little discrepancy between the two measures in our experiments.

Remarks. The cost measure we have derived here is kept at a minimum for simplicity. 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 α of navigating through the data structure, and β of computing a ray-object intersection, may differ, and should be incorporated into the definition of work as follows:

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

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

The parameter β 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 kinds of objects.

Secondly, we completely ignore the costs of traversing across different levels in a hierarchical decomposition (*vertical* motions). These costs could also be taken into account, although it is not needed in this analysis, because bounded-degree decompositions permit direct traversal from neighbor to neighbor without making use of the hierarchy; see the discussion below.

Lastly, for an actual ray distribution μ'_r differing from μ_r, the analogous cost estimator could be derived: $E(\mathcal{T}) = \sum_{\mathcal{B}_i}(1+|S_i|)\mu'_r(\mathcal{B}_i)/\mu'_r(\mathcal{B})$, 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 μ'_r. (Perhaps the ray classification of Arvo and Kirk [4] could be useful here.) The resulting model would be more accurate, especially if irregularities in ray distribution renders the approximation by μ_r inaccurate.

3.3 Octrees

The above framework can be applied to any well-behaved decomposition of the scene, but in this paper we confine our attention to decompositions that are induced by leaves of octrees, for various termination criteria. An *octree* is a hierarchical spatial subdivision 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 criteria* is subdivided into eight congruent child sub-boxes by planes parallel to the axis planes and passing through the box center. The scene surface is modeled as a collection of triangles.

The octree of [3] is constructed starting with a *cube* as a root (as opposed to, say, a minimal axis-parallel bounding box) with the following termination criteria: A cut-off size is determined, to make sure that the tree does not grow arbitrarily 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. The tree is further refined to be *balanced* [20], i.e., is such that no two adjacent leaf boxes are at leaves whose tree depths differ by more than 1, where two tree-node boxes are *adjacent to* or *neighboring* each other if two of their faces overlap.

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 [25, pp. 57–110].

In Section 4.1 we discuss a few variants of the construction of [3] that we consider to test the validity of our cost predictor. We also describe how to balance an octree and how to perform ray traversal in both balanced and unbalanced octrees[2] in a uniform way, in Section 4.1. We experimentally show in Section 4.3 that our cost predictor is accurate for both cases.

4. Experimental Evaluation

In order to evaluate the accuracy of our predictor in practice, we need to perform careful experiments. For the preprocessing phase, we implemented an octree-construction algorithm based on the one described in Section 3.3. Our implementation allows us to build variations of the octree by incorporating various construction schemes. Once an octree is built, we can estimate the ray-shooting cost per ray associated with that octree by computing our predictor. For the run-time phase to perform ray-shooting queries, we have two classes of tests.
(1) Since our cost predictor is based on a ray distribution induced by a rigid-motion invariant distribution on lines, we generated random rays accordingly, and performed ray-shooting queries for these rays using the various octrees built. The average cost per random ray shot is to be compared with the estimated cost per ray obtained from our predictor. The intent is to confirm that the discrepancy between $E(\mathcal{T})$ (given in Equation (3)) and $E^*(\mathcal{T})$ (given in Equation (2)) is small, so that our choice of the much simpler $E(\mathcal{T})$ as the cost predictor is well justified.
(2) We also implemented a simple ray tracer, which performs the ray-shooting queries necessary to render an image using the underlying octree built. We ran the ray tracer to render the image, and collected the statistics of the average ray-shooting cost per ray, to be compared with the estimated cost per ray obtained from our predictor. The intent is to evaluate the accuracy of our predictor in the context of the important ray tracing applications in practice, whose un-

[2] We call an octree *unbalanced* if we do not perform an additional step to balance it, but it may happen to be already balanced.

derlying ray distributions might differ from what we have assumed.

4.1 Octree construction

Our octree construction implementation is based on the scheme described in Section 3.3, 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 (maximum number of objects that are allowed to meet any leaf node and maximum octree depth allowed). Recall that an octree is called "unbalanced" if we do not perform an additional step to balance it; the balancing step is described below. In addition, since the scheme described in [3] 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.

It is an interesting algorithmic question to efficiently balance an octree. 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 [20] described next.

While constructing the initial octree, we maintain a global table of leaf vertices; such a vertex may be shared by at most eight leaf boxes, but only appears once in the table, and each box has eight indices into the vertex table pointing to its eight vertex entries. Associated with each vertex entry in the table, we also maintain the maximum depth value among all leaf boxes sharing this vertex. In the octree we use an arbitrary but *globally fixed* order to arrange the child nodes of each internal node from left to right, so that it is well-defined to talk about the left-most child, the second child from the left, and so on. The balancing algorithm consists of three passes. In the first pass, we perform a depth-first traversal of the tree *from left to right* (i.e., always visiting the left-most child first); in the second pass, we perform a depth-first traversal of the tree *from right to left* (i.e., always visiting the right-most child first); finally, in the last pass, we perform a depth-first, left-to-right traversal again. In each traversal, we do the following. When we visit a leaf node, we remember the depth ℓ of the current leaf, and look at the eight vertices of its bounding box from the global table. If all such vertices in the table have maximum leaf-depth values no more than $\ell + 1$, then we proceed without additional actions. Otherwise, the current leaf box is adjacent to some other leaf box whose leaf depth is larger than allowed, and thus we subdivide the current leaf to increase the leaf depth, by replacing the current leaf with an internal node plus eight child leaves, and proceed to the child leaves.[3]

[3] In fact, this balancing algorithm produces a stronger version of balanced octrees, namely, any two leaf boxes sharing a common *vertex* (rather than just a common *face*) are at leaves with tree-depth difference no more than one. In general, we call an octree *k-balanced* if, in the balancing requirement, two leaf boxes are considered adjacent when two of their faces of dimension k overlap, for $k = 0, 1$, or 2. The balanced octrees defined in Section 3.3 are 2-balanced, and the balancing algorithm here makes an octree 0-balanced (also called *smooth* in [20]), which in particular is 2-balanced. Throughout the paper we use the term *balanced* to mean *2-balanced* for simplicity, and use the balancing algorithm here to obtain (2-)balanced octrees.

It is interesting to observe that running a single pass of the above process is not enough to guarantee global balancing, but performing three passes is. The correctness of the algorithm is proved in [20]. However, the algorithm may subdivide more than necessary, and hence it is not guaranteed to produce a space-optimal balancing [20].

There is another important issue regarding the implementation of our octrees. Recall that we consider both balanced and unbalanced versions of octrees. For an unbalanced octree, it is possible to have a large leaf node with many small adjacent leaf nodes. To traverse along a ray through adjacent leaf nodes, we could maintain bidirectional pointers between adjacent leaf nodes. However, this would cause two problems for an unbalanced octree: (1) we may need to store an *arbitrarily large* number of pointers from a large node to its neighboring small nodes; (2) traversing from a large node to a small node would need an extra step to determine which small node the ray goes to.

We use a simple scheme that works for both balanced and unbalanced octrees and resolves the above problems. For any leaf node b, we store a pointer to its neighboring leaf node *only if* that leaf node has size greater than or equal to that of b. For a neighboring leaf node t that is smaller than b, we maintain a pointer to the ancestor node of t that is of the same size as b. In this way, the pointers are always pointing to something that is at least as large as the current node, and hence there is only a constant number of pointers stored with each node. To traverse from a large leaf node to a small adjacent leaf neighbor, we follow the pointer to the neighbor's ancestor, and then go down 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, and hence the requirement of bounded-degree decomposition for inducing our cost predictor in theory is fulfilled (see Section 3.2). On the other hand, if the octree is unbalanced, then such traversal cost 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 requirement.

4.2 Test datasets

We evaluate our cost predictor using scenes drawn from the Standard Procedural Database (SPD) [14], as well as a sphere model, and five scenes provided by Steven Fortune from Bell Laboratories.

The SPD scenes include five `tetra`, twelve `teapot` and four `gear` scenes, all of various complexities. To these, we add five `sphere` scenes, each an approximation of a mathematical sphere with a convex polyhedron having a different number of vertices. The intent is that within a scene family, the surface geometry should be 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` scenes, the geometry actually changes with the number of triangles. Here we measure the effect of creating holes in the scene.

In addition to these scenes, we study our cost predictor on scenes of an architectural nature: the models of lower and mid Manhattan (both of modest size, less than 10,000

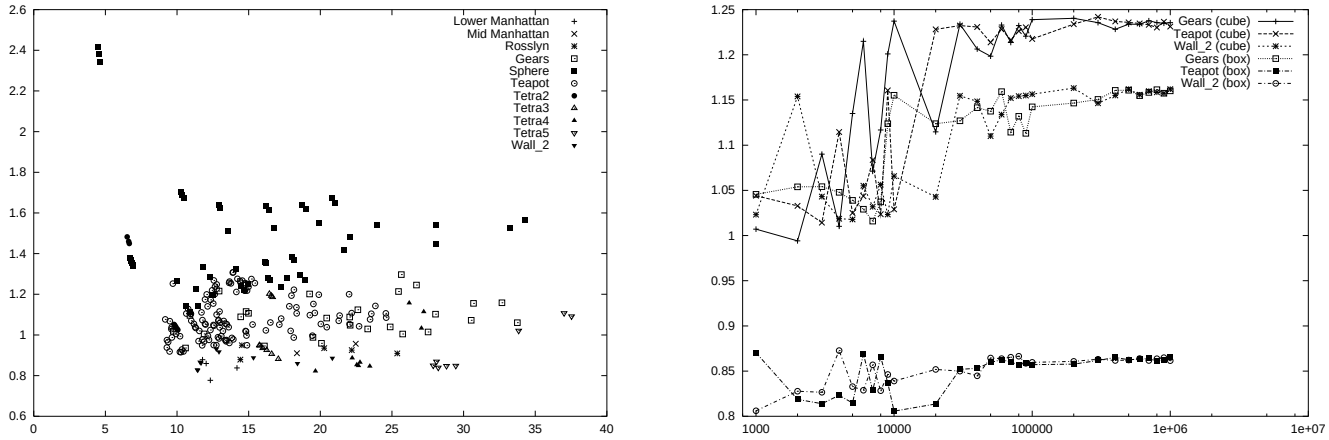

Figure 1: Results for random rays: (a) Ratio of the predicted to the actual cost (y-axis). The x-axis is the actual cost. (b) Evolution of the ratio as a function of the number of rays generated.

polygons to keep computational costs reasonable), as well as of Rosslyn, and two wall models of a Bell Laboratories building, communicated to us by Steven Fortune.

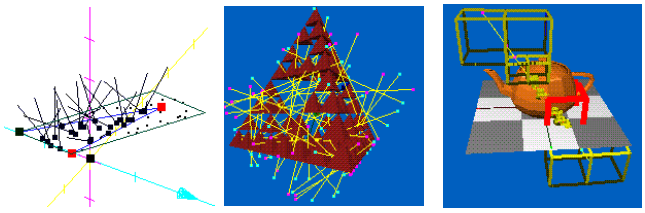

Figure 2: (a) 20 random rays emanating from a triangle. (b) Tetra scene with 50 random rays. (c) Simulation of octree traversal on a teapot scene.

4.3 Experimental results

We ran our experiments on test datasets described in Section 4.2, with number of triangles ranging from 16 to 71752, on various Sun Blade 1000 workstations with 750MHz UltraSPARC III CPU and up to 4GB of main memory. For each dataset, we built an octree for every possible combination of the following options: (1) balanced vs. unbalanced (recall from Section 4.1 that "unbalanced" means that we do not perform the balancing step), (2) maximum number of objects allowed to reside in a leaf node being 2, 5, or 10, and (3) the root box of the octree being a cube vs. not being a cube. The total number of combinations of datasets and octree variants we have tested is more than 270.

For each of the dataset-tree combinations, we estimated the average ray-shooting cost per ray using our predictor $E(\mathcal{T})$ given in Equation (3). We refer to this value as the *predicted cost*. We also performed ray-shooting operations on each dataset-tree combination, by simulating ray traversal for random rays and by running our ray tracer on the octree, respectively, and computed the average ray-shooting cost per ray, defined as the total number of octree nodes visited plus the total number of ray-object intersection tests, divided by the total number of rays. We refer to this value as the *actual cost*. We would like to see how close the predicted cost and the actual cost are, in both settings of random rays and ray tracing.

4.3.1 *Random rays*

We produce a set of random rays corresponding to the rigid motion invariant measure on the lines. We enclose the scene in an axis-aligned bounding box. To generate a random ray, we first randomly pick a polygon from the set of polygons on the surface of the objects and the bounding box, where each polygon is weighted by its surface area. We then pick a random point uniformly on the surface. Finally, we pick a random direction with probability proportional to the cosine of its angle with the normal of the polygon surface. Figure 2(a) shows an example of generating 20 rays on a triangle surface, and Figure 2(b) shows 50 generated rays in a tetra scene. For each of the rays, we simulate the actual ray traversal and count the number of visited octree boxes as well as the number of the ray-object intersection tests. The ray starts from the top level root box, then zooms into the leaf box that contains the ray origin. From there we test for intersection against each object stored in the leaf. If there is no hit, we go to the neighbor box. This process repeats until the ray meets an object or goes out of scope. A simulation of this process on a teapot scene is shown in Figure 2(c). Each cube in the figure represents an octree leaf box.

For each scene, we compute the ratio of the predicted cost (given by our predictor) to the actual cost (for our randomly generated rays). If this ratio is close to 1, then our simplified predictor $E(\mathcal{T})$ given in Equation (3) is close to the more complicated theoretical predictor $E^*(\mathcal{T})$ given in Equation (2). We observe that the ratio lies between 0.8 and 1.7 for all but the smallest scenes. Figure 1(a), we plot this ratio as a function of the number of triangles in the scene. Although not apparent in the figure, for all scenes, the balanced octree leads to a better ratio (as expected).

In Figure 1(b), we plot this ratio as a function of the number of random rays for scenes `gears`, `teapot`, and `wall_2`. For these three scenes, we observe a rather slow but steady convergence of the process in terms of the number of random rays cast in our experiments.

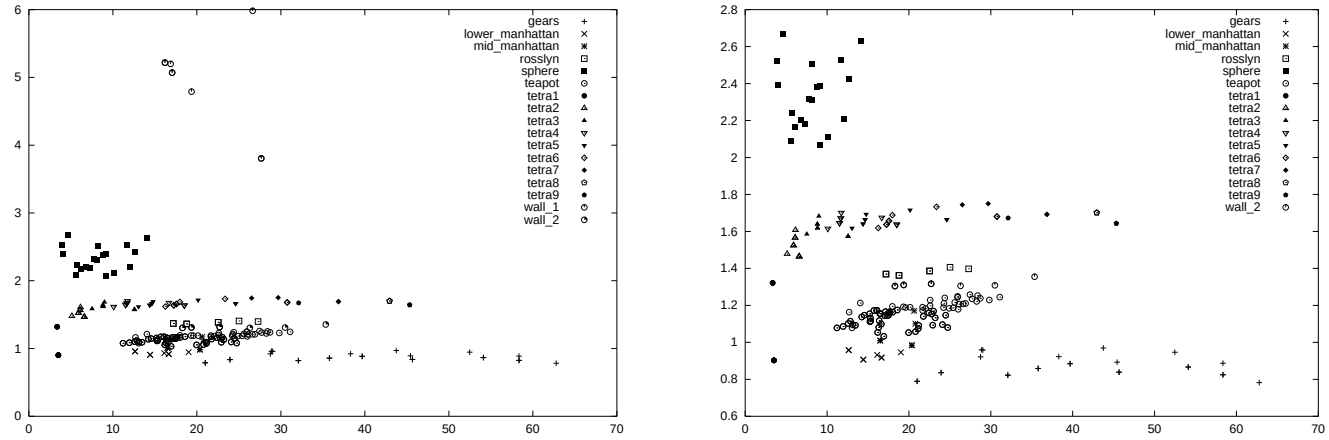

Figure 3: Results for ray tracing: Ratio of the predicted to the actual cost (y-axis). The x-axis is the actual cost. (a) All data sets. (b) The same plot without `wall_1`.

4.3.2 *Ray tracing*

For each of the dataset-octree combinations that we consider, we compute the predicted cost and run our ray tracer on the octree to render the scene and compute the actual cost. We want to evaluate the accuracy of our predictor by examining how close the predicted cost and the actual cost are.

In Figure 3(a), we plot the ratio of predicted cost to the actual cost as a function of the actual cost, for each family of the scenes. Notice that we plot each of `tetra1`, ..., `tetra9` separately since each has different geometry, while for `gears` and `spheres` the underlying geometry of the object surface stays the same for different input sizes and hence we plot them together. The y-values of the plot measure how good our predictor is, the ratio of 1 corresponding to a perfect prediction. Except for `wall_1`, for which the ratio lies between 3.8 and 6.0, we get very good predictions for all data sets. To take a closer look at these values, we repeat the same plot but exclude `wall_1`, in Figure 3(b). All the ratios are between 0.8 and 2.7, indicating that our predictor works quite well for all but one of our test datasets. We remark that in all these plots, we do not distinguish among different variations of the underlying octrees of a given input, and the results show that the corresponding y-values do not vary much. In fact, for a single scene (e.g., `rosslyn`, `wall_2`, any single `tetra` scene—recall that the geometric surface is different for each `tetra` scene), the cost predictor is remarkably independent of the underlying octree and of the scene size, and seems to only depend on the scene geometry. For instance, the ratio for `wall_2` is $1.2^{\pm 0.05}$, for `gears` it is $0.85^{\pm 0.1}$, and for `teapot` $1.1^{\pm 0.1}$. The data sets `sphere` and `wall_1` seem to be less well-behaved. This illustrates that the behavior of our predictor is very robust with respect to different octree structures.

Since, for a given scene, the ratios in Figures 3(a) and (b) 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. We observed that the number of tree leaves increases by a factor of 2 to 5 after balancing, showing that the tree structures are indeed quite different.

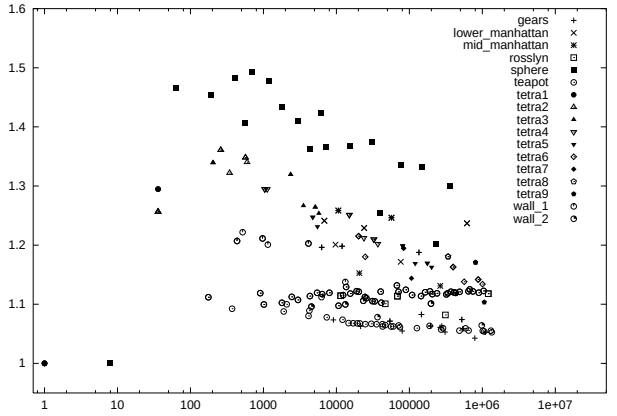

Figure 4: Overhead of vertical motions. The y-axis is the ratio of the total number of nodes traversed to the total number of leaf nodes traversed in running our ray tracer. The x-axis is the number of octree nodes in logarithmic scale.

Recall that in our ray-shooting queries, moving from one leaf to its neighbor along the ray, may require visiting some internal nodes. We refer to such internal-node traversal as the *overhead of vertical motions*. In Figure 4, we plot the ratio of total number of nodes traversed, including internal and leaf nodes, to the total number of leaf nodes traversed, from running our ray tracer, against the number of octree nodes in logarithmic scale. We see that the ratio ranges between 1 and 1.5. Intuitively, the ratio should be upper-bounded by 1.5 for *balanced* octrees. Consider two adjacent leaf boxes. If they are at the same tree depth, then the ratio is 1. If they are at different depths—differing by 1, going from the smaller to the larger box has unit cost, and going in the opposite direction has cost 2 since we need to go through one additional internal node. If two rays of the 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. Our experiments presented in Figure 4 include *both* balanced

and unbalanced octrees, but the ratio is still between 1 and 1.5, showing that balanced and unbalanced octrees exhibit similar behaviors in terms of the average overhead of vertical motions, for a large set of sample rays from a typical ray-tracing computation in practice. This also experimentally justifies our choice of not including this overhead into our cost predictor for ease of computation.

To further verify that the quality of 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 levels (levels 1 and 2), we always subdivide the node; for levels 3 and beyond, we compute the value $v = rand \cdot level$, 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 using our predictor, and ran our ray tracer to obtain the actual cost, for each such random octree built. Since each run on the same input resulted in a different octree structure, we conducted 5 runs for each of the input datasets, and plotted the predicted cost and the actual cost for each run separately.

In Figure 5, we plot the ratio of the predicted cost to the actual cost, against the actual cost, for the random octrees. We do not distinguish between different input datasets because their ratio values are very similar. Ratio values range between 0.91 and 1.55, except for one data point at 0.65. This shows that our predictor performs very well even for random octrees. Notice that the actual cost can be as bad as nearly 290, as opposed to less than 65 for "well-built" octrees shown in Figures 3(a) and (b). No matter how good or bad the actual cost is, we predict the actual cost quite accurately.

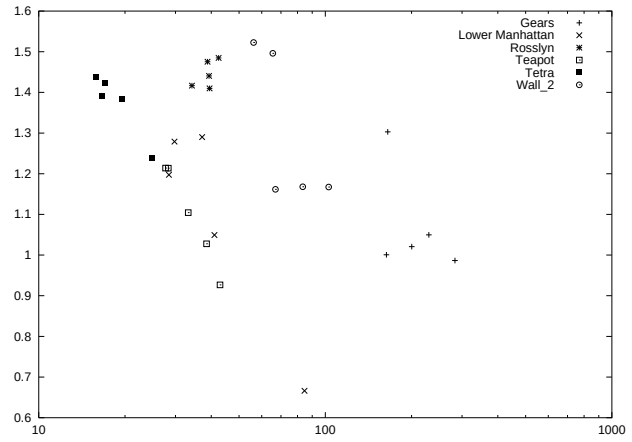

Figure 5: Results for the random octrees: the ratio of the predicted to the actual cost vs. the actual cost.

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 space 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 complex 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, intuitively should be close to the provable measure. In particular it is within a constant factor for octrees 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 particular type of space decompositions, namely 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 (meaning octrees to which the balancing step is not applied) are used commonly in ray-tracing applications. They do not necessarily have bounded degree. Yet we experimentally verify the accuracy of our predictor for those octrees as well. 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 relevance of our cost predictor.

One note is in order: we did not treat the cost optimization problem for ray tracing, but concentrated specifically on the ray shooting. Optimizing ray tracing must also take into account various illumination parameters (to name but a few: colors, reflectance and material properties, lights, and minimum threshold on the contribution of a ray to a pixel in order to cast its reflections). It is beyond the scope of this paper, indeed of our general approach, to address this problem.

We conclude by mentioning a few directions for further research. We begin with the problem of computing an octree 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 [3] (where the cost measure is simply the surface area), but the problem is still open for octrees. Their construction first proceeds to build an octree using the termination criterion mentioned in Section 3.3, but it is not known if this octree has a cost within a constant factor of the optimal.

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 3.2.)

Lastly, it is intriguing that our cost predictor is insensitive to whether or not the octree has been balanced. While there may be some scenes (constructed with the knowledge of the unbalanced octree) for which the balancing process will substantially change the cost, it is not clear by how much that change could be in the worst case.

Acknowledgments

We wish to thank Steven Fortune of Bell Laboratories for providing some of the test scenes, as well as for several fruitful discussions. Many thanks also go to an anonymous reviewer who greatly helped in improving this article.

References

[1] P.K. Agarwal, B. Aronov, and S. Suri. Stabbing triangulations by lines in three dimensions. *Proc. 11th ACM Sympos. Comput. Geom.*, pages 267–276, 1995.

[2] P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. *Advances in Discrete and Comput. Geom.*, 1998.

[3] B. Aronov and S. Fortune. Approximating minimum weight triangulations in three dimensions. *Discrete Comput. Geom.*, 021(04):527–549, March 1999.

[4] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):55–64, 1987.

[5] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Morgan Kaufmann Publishers, Inc., 1989.

[6] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray tracing. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 11–20, 1997.

[7] F. Cazals and M. Sbert. Some integral geometry tools to estimate the complexity of 3d scenes. Research report n.3204, INRIA, July 1997. `http://www-sop.inria.fr/prisme/personnel/cazals/papers/INRIA-TR-3204.ps.gz`.

[8] A.Y. Chang. A survey of geometric data structures for ray tracing. Technical Report TR-CIS-2001-06, CIS Department, Polytechnic University, 2001.

[9] J.H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[10] J.G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4:65–83, 1988.

[11] M. de Berg, M. Katz, F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. *Proc. 13th ACM Symp. on Computational Geometry*, pages 294–303, 1997.

[12] A.S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann Publishers, Inc., 1989.

[13] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[14] E. Haines. *Standard procedural database*. 3D/Eye, 1992. version 3.13, `http://www.acm.org/tog/resources/SPD/overview.html`.

[15] V. Havran, J. Bittner, and J. Prikryl. The Best Efficiency Scheme project proposal - version 1.90jp, April 1st 2000. `http://www.cgg.cvut.cz/GOLEM/proposal2.html`.

[16] V. Havran, J. Prikryl, and W. Purgathofer. Statistical comparison of ray-shooting efficiency schemes. Technical report TR-186-2-00-14, Czech Technical University, Czech Republic, 2000. `http://www.cgg.cvut.cz/GOLEM/bes.html`.

[17] M.R. Kaplan. The use of spatial coherence in ray tracing. *Techniques for Computer Graphics*, pages 173–193, 1987.

[18] J.D. MacDonald and K.S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[19] J.S.B. Mitchell, D.M. Mount, and S. Suri. Query-sensitive ray shooting. *Int. J. Comput. Geom. & Appls.*, 7(3):317–347, August 1997.

[20] D.W. Moore. *Simplicial Mesh Generation with Applications*. Ph.D dissertation, Cornell University, 1992.

[21] B. Naylor. Constructing good partitioning trees. In *Proceedings of Graphics Interface '93*, pages 181–191, Toronto, Ontario, may 1993. Canadian Information Processing Society.

[22] M. Pellegrini. Ray shooting and lines in space. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 599–614. CRC Press LLC, NY, 1997.

[23] E. Reinhard, A.J.F. Kok, and A. Chalmers. Cost distribution prediction for parallel ray tracing. In *Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 77–90. Eurographics, September 1998.

[24] E. Reinhard, A.J.F. Kok, and F.W. Jansen. Cost prediction in ray tracing. In P. Hanrahan and W. Purgathofer et al., editors, *Rendering Techniques '97*, pages 42–51. Porto, Portugal, 1996.

[25] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[26] L. Santaló. Integral Probability and Geometric Probability. In *Encyclopedia of Mathematics and its Applications*, volume 1. Addison-Wesley, Reading, MA, 1979.

[27] I. Scherson and E. Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, December 1987.

[28] K.R. Subramanian and D.S. Fussell. Automatic termination criteria for ray tracing hierarchies. In *Proc. of Graphics Interface '91*, June 3-7 1991.

[29] I.E. Sutherland, R.F. Sproul, and R.A. Schumacker. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6(5):1–55, 1974.

[30] J. Vleugels. *On Fatness and Fitness — Realistic Input Models for Geometric Algorithms*. Ph.D. thesis, Dept. Comput. Sci., Univ. Utrecht, Utrecht, The Netherlands, 1997.

[31] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[32] K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Choand, C. M. Park, and I. Y. Song. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Trans. Visual and Comp. Graphics*, 1:343–349, 1995.