

Cost-Driven Octree Construction Schemes: An Experimental Study

Boris Aronov, Hervé Brönnimann, Allen Y. Chang, and
Yi-Jen Chiang

*Department of Computer and Information Science, Polytechnic University,
Brooklyn, NY 11201 USA; {aronov@,hbr@,achang@cis.,yjc@}poly.edu.*

Abstract

Given a scene consisting of objects, ray shooting queries answer with the first object encountered by a given ray, and are used in ray tracing and radiosity for rendering photo-realistic images in graphics, radio propagation simulation, and many other problems. We focus on one popular data structure for answering ray shooting queries—the octree. It is flexible and adaptive and has many applications. However, its degree of adaptiveness usually depends on manually selected parameters controlling its termination criteria. While practitioners usually rely on experience and heuristics, it is difficult to fix a set of parameter values that is good for all possible scenes.

Recently, we introduced a simple cost predictor that reflects the average cost of ray shooting with a given octree (Cost prediction for ray shooting, in *Proc. 18th Annu. ACM Sympos. Comput. Geom.* (ACM, New York, 2002, pp. 293–302), and showed a termination criterion (cost-driven k -greedy) that guarantees a cost within a constant factor of optimal (Cost-optimal trees for ray shooting, in *Proc. LATIN'04, LNCS 2976*, Springer, 2004, pp. 349–358). In this study, we compare this criterion with several octree construction schemes widely used in the computer graphics literature (such as bounding the number of objects in a leaf and the maximum depth). Our experimental results show that the octrees constructed using our schemes are generally comparable to or better than those built with *a priori* fixed parameters. We then fine-tune the predictor and observe the behavior of our algorithm on octrees built to support a simple ray-tracing engine. It appears to work well in practice.

Key words: Ray shooting, cost model, cost prediction, average performance, octree, space decomposition

1 Introduction

The ray-shooting problem was introduced in computer graphics as the most basic geometric primitive for computing global illumination [1]. Given a set \mathcal{S} of n objects, we would like to find the first object of \mathcal{S} hit by a given ray. This type of query is known as a *ray-shooting query*. The heavy computational cost required to answer a large number of such queries has for a long time remained a bottleneck for rendering and radio-propagation simulation applications [5, 6, 19] How fast ray-shooting queries can be answered is an important algorithmic problem that has received a lot of attention both in theory and in practice.

Unfortunately, solutions with provable guarantees are often not very efficient in practice due to large hidden constants. In addition, they are mostly geared towards worst-case scenarios. On the other hand, the heuristic methods introduced by practitioners often work very well in typical cases. However rarely, nevertheless, worst-case scenarios do happen, and cause heuristics to perform extremely poorly. The problem with most heuristics is that there is no way to know if poor performance results from the lack of theoretical guarantees (hence the hope remains that another method might be faster) or from the intrinsic hardness of the problem instance.

Our admittedly idealistic goal is to fill in the gap between the theory and the practice to arrive at a solution that is both backed up with a theoretical guarantee and behaves well in practice. In [2], we took the first step in this direction by introducing a very simple cost predictor in order to estimate the performance of a specific ray-shooting data structure on a given data set. The predictor is based on the ray distribution induced by the rigid-motion invariant distribution for lines. Our predictor has been proven to work on bounded-degree decompositions in theory [2, 3]. It is simple and easy to compute. In [2], we experimentally confirm that the predictor can accurately estimate the cost of a ray shot in practice, without actually running the ray-tracing rendering program. In this paper, we use the cost predictor to drive the construction of a data structure with insignificant time overhead.

Despite a huge amount of research on ray shooting in the ray-tracing literature, only a few references try to model the cost of ray shooting. We survey

¹ 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.

² A preliminary version of this paper appeared as B. Aronov, H. Brönnimann, A.Y. Chang, and Y.-J. Chiang, “Cost-Driven Octree Construction Schemes: An Experimental Study,” *Proc. 19th Annual ACM Symposium on Computational Geometry*, 2003, pp. 227–236.

the existing work in Section 2. In contrast to the usual lack of theoretical guarantees, our approach is driven by the presence of a provably accurate estimate of the average cost of ray shooting: using our cost predictor, we can show (in a companion paper [7]) that the following method will produce an octree whose cost is always within a constant factor (reasonably small in theory) of the cost of *any* octree. The criterion tries to analyze the scene in a more global way, extending standard greedy criteria by introducing a degree of lookahead: if subdividing a leaf up to a fixed additional depth k does not lead to an improved cost, then we stop the subdivision at the node. Otherwise, we subdivide the leaves of the best subtree of depth at most k using the same criterion recursively. Section 3.2 gives further details on how exactly the data structure is constructed.

Our experiments in Section 4.2 confirm that the octree constructed driven by the predictor is really better than octrees constructed without the predictor in terms of the cost, speed, and space. Moreover, we seem to experimentally observe that the constant factor approximation of [7] is actually very close to one, but that this theoretically provable method, although performing better, does not always perform best. Rather surprisingly, we find that the 1-greedy criterion seems to always lead to very good (if not outright to the best) octrees for practical instances, even though on some fabricated scenes it can be arbitrarily far from optimal.

In addition to the k -greedy method, we may impose a requirement that a node be subdivided only if this “substantially (i.e., by a minimum of $t\%$) improves” the cost. Intuitively, this prevents arbitrarily small cost improvements causing large tree expansion. Experiments described in Section 4.3 show this further improves the efficiency of the data structure. In Section 4.4, we try to fine-tune the coefficients of the cost predictor. The coefficients are not only machine-dependent but are also affected by the implementation of the program and by the termination criterion chosen. Finally, conclusions are presented in Section 5.

2 Previous Work

In general, the cost for shooting a ray can be represented as

$$C_{\text{total}} = C_{\text{struct}} + C_{\text{inters}}, \quad (1)$$

where C_{total} is the total ray-shooting cost, C_{struct} is the cost for traversing the data structure, and C_{inters} is the cost for ray-object intersection tests. A naive ray tracer does not use any data structure—all of the time is spent on ray-object intersection tests, i.e., $C_{\text{total}} = C_{\text{inters}}$. A simple data structure,

such as a bounding volume for each object constructed by Whitted et al. [27, 36], can reduce the time for ray-object intersection tests by lowering the number of the tests, at the expense of raising the number of ray-bounding-volume intersection tests which are usually simpler and faster than testing the primitive objects. Still, C_{inters} can take up to 95% of the ray-shooting time for complex scenes [27, 36]. As the data structures become more complicated, the cost in Equation (1) shifts from C_{inters} towards C_{struct} . Kilmaszewski et al. [20] point out that up to 60% of the rendering time can be spent on traversing modern data structures. The time for ray-object intersection tests is reduced to 20% or even 10% of the total rendering time [37]. As the time spent on traversing the data structure becomes significant, optimizing the structural cost becomes important. Once a choice of the data structure is made, the total ray-traversal time varies depending on the choice of parameters used for constructing it. For a hierarchical data structure, some of the parameters determining the structure include the number of primitive objects in the scene, the number of nodes in a hierarchy, the distribution of objects, the distribution of rays, the depth of the hierarchy, etc. Of those, one may tune the number of nodes and depth of the hierarchy. The prevalent method is to set these parameters manually and actually run the ray tracer to see which settings result in the best performance among all of the tests. Havran [17] makes an extensive experimental study and concludes that no single data structure is uniformly better than others. Two recent surveys of data structures used for ray-tracing in practice were presented in [10] and [17].

Some relevant parameters can be extracted directly from the scene data. The most obvious such parameter is the number of objects in the scene. This measure of scene complexity is used widely in traditional theoretical analysis. Yet research shows that the size of objects has more impact on ray-tracing time than the object count [11, 26, 29]. The “size” of an object can be evaluated by either its volume or its surface area.

Also, the statistics of the object distribution reveal certain properties of a scene that can help estimate the cost of ray shooting. Scherson and Caspary [29] discuss several properties of the object distribution and use those properties to analyze ray-shooting costs. Cazals and Sbert [9] enumerate several integral geometry tools useful for analysis of global statistical properties of the scene. The idea is to shoot a set of random rays. The rays probe the scene and reveal the object distribution in several ways. For example, the average number of intersection points for a transversal line can convey information about the sparseness of the scene or the percentage of screen coverage [29]. The length of the rays in free space tells us where the objects are or where most of the objects are located. This is the same as the *depth complexity* defined by Sutherland et al. [32].

Several structural cost functions C_{struct} have been proposed, which have to do

with the kind of data structure used, what is considered relevant, and how the size of objects is measured.

In order to estimate the traversal costs in the data structure, Goldsmith and Salmon [15] define C_{struct} to be the average number of nodes visited by a ray until it hits an object in a hierarchical data structure, as follows:

$$C_{\text{struct}} = 1 + \sum_i c_i \cdot \Pr(i \mid \text{root}) = 1 + \sum_i c_i \cdot \frac{A(i)}{A(\text{root})}, \quad (2)$$

where $\Pr(i \mid \text{root})$ is the conditional probability of the ray hitting node i given that it hits the root node, $A(i)$ is the surface area of internal node i , $A(\text{root})$ is the surface area of root node, and c_i is the number of children of node i . Naylor [25] and Cleary and Wyvill [11] employ a similar idea to estimate the traversal cost but with a different estimation of conditional probabilities. Naylor uses volume instead of surface area to calculate the conditional probabilities in a BSP-tree traversal, while Cleary and Wyvill [11] introduce the *augmented volume* to estimate the cost of traversing a uniform grid. The augmented volume of an object is the sum of the volumes of grid cells where the object resides.

MacDonald and Booth [21] use an idea similar to regional probability to estimate the traversal cost of a hierarchy. The cost C_{struct} is a combination of *internal cost* C_{int} and *external cost* C_{ext} , which are the average cost of traversing a non-leaf and a leaf node, respectively. Based on these definitions, Equation (2) can be rewritten as (note that in the data structure used in [21] each internal node has degree two)

$$C_{\text{struct}} = C_{\text{int}} \cdot \sum_{i=1}^{n_{\text{int}}} \frac{A(i)}{A(\text{root})} + C_{\text{ext}} \cdot \sum_{j=1}^{n_{\text{ext}}} \frac{A(j)}{A(\text{root})}, \quad (3)$$

where n_{int} and n_{ext} are the numbers of internal nodes and external nodes, respectively.

Subramanian and Fussell [30] propose to compute C_{struct} as follows:

$$C_{\text{struct}} = n_{\text{region}} \cdot C_{\text{per_region}}, \quad (4)$$

where $C_{\text{per_region}}$ is the average cost of traversing the ray from current region to the next. They argue that for a hierarchical structure, $C_{\text{per_region}}$ should be approximated by weighted average tree depth \bar{D} because we may have to traverse the hierarchy from the root to the leaf to find the next region hit by the ray. Again, the weight is related to the “size” of a region. This formulation is reused by Reinhard et al. [26], who specifically use the surface area of octree cells to compute \bar{D} .

The above discussion accounts for the structural costs C_{struct} , the first term of Equation (1). As for the intersection costs C_{inters} , the second term of Equation (1), in general they are the sum of all the object counts at the leaves traversed. Observing that sometimes the computed intersection falls outside the current leaf and will be recomputed in a subsequent leaf, MacDonald and Booth [21] find that many ray-tracing engines use a “mailbox” mechanism, in which the computed intersection is cached and reused in the subsequent leaf without recomputation. They revise their computation of C_{inters} to account for that fact, by determining the probability that a ray intersects the volume defined by the union of leaf nodes where an object resides. If β denotes the cost of an elementary ray-object intersection, they compute

$$C_{\text{inters}} = \beta \sum_{i=1}^n \frac{\tilde{A}(O_i)}{A(\text{root})}, \quad (5)$$

where $\tilde{A}(O_i)$ denotes the approximate area of object O_i . For $\tilde{A}(O_i)$, they use the sum of the areas of the projection of the union of leaf nodes where the objects reside onto the six faces of the root bounding volume.

Constructing a hierarchical search structure for ray tracing can usually improve the overall performance. If the hierarchical structure is too deep, however, one may spend too much time moving from one level to another. To prevent C_{struct} in Equation (1) from overwhelming the total cost, we can manually set a limit on the height of a search structure. Of course this may increase C_{inters} . How to control the depth of a hierarchy to automatically attain the best efficiency becomes an interesting issue.

Weghorst et al. [34] investigate the trade-offs in Equation (1) between the cost of traversing a hierarchy (“tree cost”) and the cost of ray-object intersection tests (“object cost”). The trade-off is further discussed by Arvo and Kirk [4]. Additional improvements were presented by Goldsmith and Salmon [15] and by Subramanian and Fussell [31]—although it takes them longer to construct the hierarchy, they construct one that is better than that produced by the approach of Weghorst et al. [34].

The approach of Subramanian and Fussell [31] uses the cost measure of Equation (4). The costs C_{struct} and C_{inters} are computed during the construction of the hierarchy. In the beginning, the decrease in C_{inters} overwhelms the increase in C_{struct} . The total cost drops quickly while the depth of the hierarchy increases. At some point, as the depth increases further, the total cost reaches a minimum value, then becomes stable or increases slowly. They stop subdividing the space at this point.

To estimate the cost more accurately, an alternative approach is to run a low resolution ray-tracing phase before the full functional ray tracing starts. The cost function is used to monitor the low resolution ray-tracing phase, as

proposed by Reinhard et al. [26]. As mentioned, they use the surface area of octree cells to compute the weighted average tree depth \bar{D} , which is then used to roughly estimate $C_{\text{per_region}}$ in an implementation in which moving to the next cell involves descending the tree from the root. Again, the scene is further divided if the cost keeps decreasing. There are several problems with using this approach. The first is the overhead of the low-resolution ray-tracing computation. The second problem is the choice of appropriate resolution that is low enough to have only insignificant impact on the actual ray-tracing speed, yet high enough to provide a representative sample of the entire scene. The user can only pick a good resolution based on her experience.

Some degree of freedom in constructing the hierarchy is exploited by MacDonald and Booth [21], based on the observation that the optimal splitting plane lies between the space median and the object median of a node in a “bintree,” which is essentially a three-level octree similar to Kaplan’s BSP-tree [18]. MacDonald and Booth [21] and Whang et al. [35] estimate the cost of splitting the node at a particular position by the surface area heuristic and pick the “optimal” by sampling ten positions between the spatial and object medians of the node; the quality of the split depends on the number of objects meeting the portion of the node on either side of the split and the surface area of the two portions.

In all of the hierarchical schemes in the literature, the partitioning process stops when certain termination criteria are met; the criteria may be global or local. For example, the octree of Whang et al. [35] stops further splitting when the number of objects within a node is less than a threshold value, a popular criterion that was already used by Glassner [12]. The bintree of MacDonald and Booth [21] stops further splitting when Equation (3) reaches the minimum. Subramanian and Fussell [31] and Reinhard et al. [26] follow a similar approach.

One should be aware that the minimum value found here is only a local minimum. The cost obtained using this method may not be a global minimum in some cases. Indeed, for the particular cost measure used in this paper, Brönnimann and Glisse [7] were able to prove that this “greedy” method (1-greedy in their terminology) may not lead to the optimal tree. They also prove, however, that a variant of greedy (3-greedy) produces octrees whose cost is within a constant factor of optimal (see Section 3.2). It is not known if similar results can be obtained for the cost measures discussed in this section.

Other approaches to optimizing the ray-shooting data structure, e.g., using evolutionary methods, have been explored as well [8, 22]; two common problems with such an approach are its lack of speed and of any theoretical guarantees.

3 Octree construction schemes

In this section we recapitulate the cost measure that we developed in [2] for predicting the runtime behavior of ray shooting that uses any bounded-degree space decomposition. We then recall several octree construction schemes, as well as some new ones based on the cost function.

3.1 Cost measure

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, i.e., cells sharing (a portion of) a face with it. 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 cost measure introduced in [2] reflects the expected cost of traversing the decomposition for an average ray, with rays drawn from the distribution induced by the rigid-motion invariant distribution μ_ℓ on lines. Let \mathcal{B} be the bounding box of the scene \mathcal{S} , let $A(\cdot)$ denote surface area, let \mathcal{B}_i range over the cells of the subdivision (in our case, induced by the leaves of the octree), and let \mathcal{S}_i be the set of scene objects meeting cell \mathcal{B}_i . The *cost* of a decomposition \mathcal{T} is then defined as the ratio of $W(\mathcal{T}) = \sum_{\mathcal{B}_i} (1 + |\mathcal{S}_i|)A(\mathcal{B}_i)$, which measures the average work performed during a line traversal (traversing through both cells and objects), to the “useful” portion of the work, $A(\mathcal{B}) + \sum_{s \in \mathcal{S}} A(s)$, which measures the number of line-object intersections reported for an average line. So far we have assumed that testing a ray for intersection with a scene object is equally costly to testing it for intersection with an octree box; both have been treated as unit-cost operations. If we refine the analysis further by introducing the cost α of a ray-box intersection, and the cost β for a ray-triangle intersection, we can express the numerator as (Eq. 2 from [2])

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

where the sum again ranges over all the cells \mathcal{B}_i of the subdivision. Thus

$$E_{\alpha,\beta}(\mathcal{T}) = \frac{\sum_{\mathcal{B}_i} (\alpha + \beta|\mathcal{S}_i|)A(\mathcal{B}_i)}{A(\mathcal{B}) + \sum_{s \in \mathcal{S}} A(s)} \quad (6)$$

measures the expected amount of work required, per intersection, for reporting all the line-object intersections with an average *line* in μ_ℓ .

In [2], we argue that $E_{\alpha,\beta}(\mathcal{T})$ as given in (6) in practice also accurately predicts the average amount of work required for reporting the first intersection of an average *ray* with the scene even though the precise expression for the ray cost

a little more involved, see [2] for details. This assertion has been substantiated by experiments where we somewhat arbitrarily chose $\alpha = \beta = 1$.

In theory, changing α and β only changes the cost by a factor that depends only on α and β and thus all the theory remains valid within such “constant” factors. In practice, in order to obtain accurate cost prediction, accurate values of α and β need to be used. It turns out that α is reasonably stable, but that β actually varies quite a lot depending on the ray-object configuration. Indeed, the time it takes to compute a ray-box or a ray-object intersection is not a constant, due to short-circuit evaluation of boolean operators: geometric filters may cause an early exit, skipping remaining computation. For instance, if the ray points away from the plane of a triangular object, no ray-plane intersection need be computed at all (and thus we avoid having to locate the intersection inside or outside the triangle, which is a costly operation). Further reasons are discussed in Section 4.4. It is therefore difficult to evaluate β analytically, even for a given platform. Experimentally, we find that the ratio of α to β is roughly between two to one and ten to one. Further details and measurements are given in Section 4.4.

3.2 Octree construction schemes

The above framework can be applied to any well-behaved decomposition of the space containing the scene. In this paper we confine our attention to decompositions that are induced by the leaves of octrees, for various termination criteria.

An *octree* is a hierarchical spatial subdivision 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. In the experiments reported in this paper, the scene surface is modeled as a collection of triangles (polygons are triangulated).

Our octree construction implementation is based on the scheme described in [2], with the flexibility of producing different variants of octrees by adjusting its construction criteria. As in [3], the octree may be constructed starting with a *cube* as the root (as opposed to, say, a minimal axis-parallel bounding box). Then the nodes are recursively subdivided according to a subdivision termination condition evaluated at each leaf. The tree may further be refined to ensure *balance* [24], i.e., so that no two adjacent leaf boxes are at leaves whose tree depths differ by more than one, where two tree-node boxes are *adjacent to* or *neighboring* each other if a face of one overlaps a face of the other.

The termination criteria we implemented include: a choice of a balanced or unbalanced³ octree and a choice of subdivision termination conditions. For subdivision termination conditions, we consider two classes: the *separation* termination conditions (maximum number of objects that are permitted to meet any leaf node, and maximum octree depth allowed) as studied in [2], and a new class, namely the cost-driven *greedy* criteria (with or without lookahead). The notation for our criteria is summarized in the table next page for the benefit of the reader.

For the separation termination conditions, we require that the number of objects stored in a leaf is below a certain threshold, unless maximum permitted depth has been reached. We annotate such an octree with the letter ‘j’ followed by the object number threshold. We will consider j2, j10, j20 and j30, and even j50 and j100 for larger scenes. Octrees with larger threshold values sometimes exhibit similar behavior, so it is convenient to refer to them collectively, for example, as j10+, j20+, etc.

The greedy termination conditions are evaluated as recommended in [7]: the greedy without lookahead (which we call 1-*greedy*) simply recommends to subdivide if the cost measure (as given in Equation (6)) is reduced when a node is subdivided into eight subnodes. With a lookahead (which we call *k-greedy*, for $k > 1$), the criterion evaluates the smallest cost of a subtree of depth at most k rooted at a given node, using bottom-up dynamic programming. If the cost of that subtree is smaller than the cost of keeping that node as a leaf, then the node is subdivided up to the depth achieving the best cost. The same criterion is then applied to each leaf of the resulting subtree, recursively.

Octrees constructed with the separation criterion offer no guarantee with respect to cost. Indeed, it is possible to create artificial scenes for which the cost of these octrees is arbitrarily larger than the optimal cost for that scene, no matter how the parameters are chosen. In practice, for the scenes commonly considered, they perform quite well and this is supported by our experiments. By contrast, the greedy criteria offer some guarantees [7]: 3-greedy for triangles in 3-space, or 2-greedy for points (or very small triangles treated as points), the cost of the so-constructed octree can be proven within a constant factor of optimal, no matter what the scene. 1-Greedy has been considered in the literature for various cost measures (see Section 2), but can be proved to offer no cost guarantee because the cost function is not monotonic, although it does perform well also in practice.

One variant of the greedy criterion (with or without lookahead) is the substantial-improvement greedy, which only recommends to subdivide if the cost measure locally improves by a factor of at least $t\%$ for some fixed parameter t . We

³ We call an octree *unbalanced* if we do not perform an additional step to balance it, but it may happen to be already balanced.

- **k -greedy**: if there exists a subtree of depth at most k of a node, whose cost is less than the cost of the unsubdivided node, replace the node by the subtree and recurse on its leaves.
- **k -greedy $t\%$** : k -greedy with $t\%$ improvement (the cost measure must locally improve by a factor of at least $t\%$ in order to subdivide recursively).
- **jx** : separation termination criterion (the number of objects stored in a leaf is at most x).
- **$jx+$** : the family of criteria jy for all $y \geq x$.

Notation for the termination criterion in the figures.

refer to these variants as the *substantial greedy* criteria. A more sophisticated version of this approach (which we have not implemented) would be to vary the percentage improvement as a function of the depth, or as a function of the ratio of the cost of the node to the overall cost of the octree at the time the criterion is evaluated. The intent is of course to avoid subdividing (consuming space) when the improvement in cost does not warrant it. The cost guarantees of these variants are not known, and it could well be that by stopping subdivision early we miss a substantial improvement in cost within the subtree, since the cost function is not monotonic. Yet they seem to perform well in practice.

Finally, ray traversal is performed in both balanced and unbalanced octrees in a uniform way, as is done in [2]: 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 based on Samet’s table look-up for neighbor links [28, pp. 57–110], but using only the six facet-neighbor links instead of Samet’s 26.

4 Experimental Evaluation

For the preprocessing phase, we implemented an octree-construction algorithm based on the one described in Section 3.2. 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 trees built by greedy criteria, this computation is already part of the tree construction process. We call that the *estimated cost*. We also compute the sum of the total number of nodes (both internal and external)⁴ and the total size of the object lists $\sum_{\mathcal{B}_i} |\mathcal{S}_i|$ over the leaves. We call this the *octree size*.

⁴ Since an octree is an 8-regular arborescence, the number of leaves n_ℓ is always $7n_i + 1$, where n_i is the number of internal nodes.

For the run-time phase, we perform ray-shooting queries in the ray-tracing process, gathering the statistics such as the numbers of ray-box and ray-triangle intersection tests performed, as well as the CPU time spent on those. Using the numbers of ray-box and ray-triangle intersection tests, we obtain the *actual cost* measure, defined as the total number of these operations performed divided by the total number of ray-shooting queries, as is done in [2]; using the CPU time, we obtain the *runtime* cost. Note that the actual cost only involves the number of operations performed, rather than runtime.

We would like to answer the following questions:

- (1) Is the estimated cost correlated to the actual cost? This is the main question addressed in [2], and we should verify that it still holds for the additional types of octrees we consider here.
- (2) Are the estimated and/or actual costs correlated to the runtime?
- (3) Comparing the various octree schemes with respect to size, estimated and actual costs, and runtime, is a cost-aware criterion such as greedy an improvement over a cost-oblivious criterion such as j2?
- (4) Is substantial greedy an improvement over greedy?
- (5) What amount of lookahead is useful in practice?
- (6) How should we choose α and β to tighten the correlation between estimated, actual, and runtime costs?

4.1 Test datasets

We evaluate our cost-driven octrees using a wide variety of scenes drawn from the Standard Procedural Databases (SPD) [16], only a few of which are used in the figures (**gear3**, 13 556 triangles; **tetra7**, 16 384 triangles; **teapot**, ranging between 50 and 15 000 triangles, with **teapot13** used in Figure 3(a) and 3(b) the largest among all), a **sphere** model (**sphere6**, 16 384 triangles), and two scenes commonly used as test cases in computer graphics community (**happy buddha**, 1 087 716 triangles, and **dragon**, 871 414 triangles, from Stanford Computer Graphics Laboratory [33]; several others were used as well, but do not appear in any of the graphs we present in this paper). In addition to these scenes, we have used five data sets of an architectural nature : the models of **lower_manhattan** (6 826 triangles), **mid_manhattan** (7 312 triangles), **rosslyn** (2 467 triangles), **middletown** (2 722 triangles), and **learning_center** (7 460 triangles), communicated to us by Steven Fortune of Bell Laboratories. All of the architectural models have modest size to keep computational costs reasonable.

The scenes, their characteristics and the reasons why we picked them are described in some detail in [2]. Basically, the intent is to cover various kinds

of scene topologies and geometries, and also that, within a single family, e.g. **teapot**, the geometry of the scene remains constant and only the object subdivision changes.

4.2 Evaluation of octree construction schemes

We performed our experiments on the test datasets described in Section 4.1, with number of triangles ranging from 4 to 1 087 716, 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: (a) maximum number of objects allowed to reside in a leaf node being 2, 10, 20, 30, and sometimes 50, 100 for large scenes, (b) the amount of lookahead, (c) the root box of the octree being a cube vs. not being a cube, (d) several settings of the percentage required for substantial improvement, and (e) several values of the ratio γ of α and β in Equation (6). In addition, in all the experiments, we used the termination condition of maximum cut-off tree depth, set at the binary logarithm of the number of objects in the scene. We have performed over 7 000 test runs in our experiments, of which 738 contributed to the figures. Also, our previous study [2] indicates that balancing an octree only increases the data structure size without improving the cost performance in practice. There is no reason to think that this would be any different for the trees produced with greedy criteria, especially since our previous study included a variety of octree types, even random ones. Thus here we ran all our experiments on octrees without performing the balancing step. We remark that we re-ran the experiments reported in this paper with tree balancing turned on and discovered, confirming our finding in [2], that the overhead of vertical motions in the octree, even in unbalanced ones, was insignificant for our data. This justifies our use of the predictor on unbalanced octrees.

For each of the dataset-octree combinations, we computed as in our previous paper [2] the ratio of actual to estimated costs (actual being computed using the actual numbers of ray-box and ray-triangle intersections per ray shooting query in a ray-tracing process). Again, we observe that the ratio is close to one for all the scenes and all the octrees encountered—the ones not tested in [2] are trees generated by the k -greedy algorithm and they also confirm our predictions. This justifies using the word “cost” without explicit reference to actual or estimated.

We include a profile of our experiments for the **teapot** family in Figure 1 as a representative one; other dataset families we tested, such as **tetra**, exhibit similar trends. In Figure 1(a), we plot the ratio of the actual to the estimated cost as a function of the scene size. We indeed observe that, as in [2], this

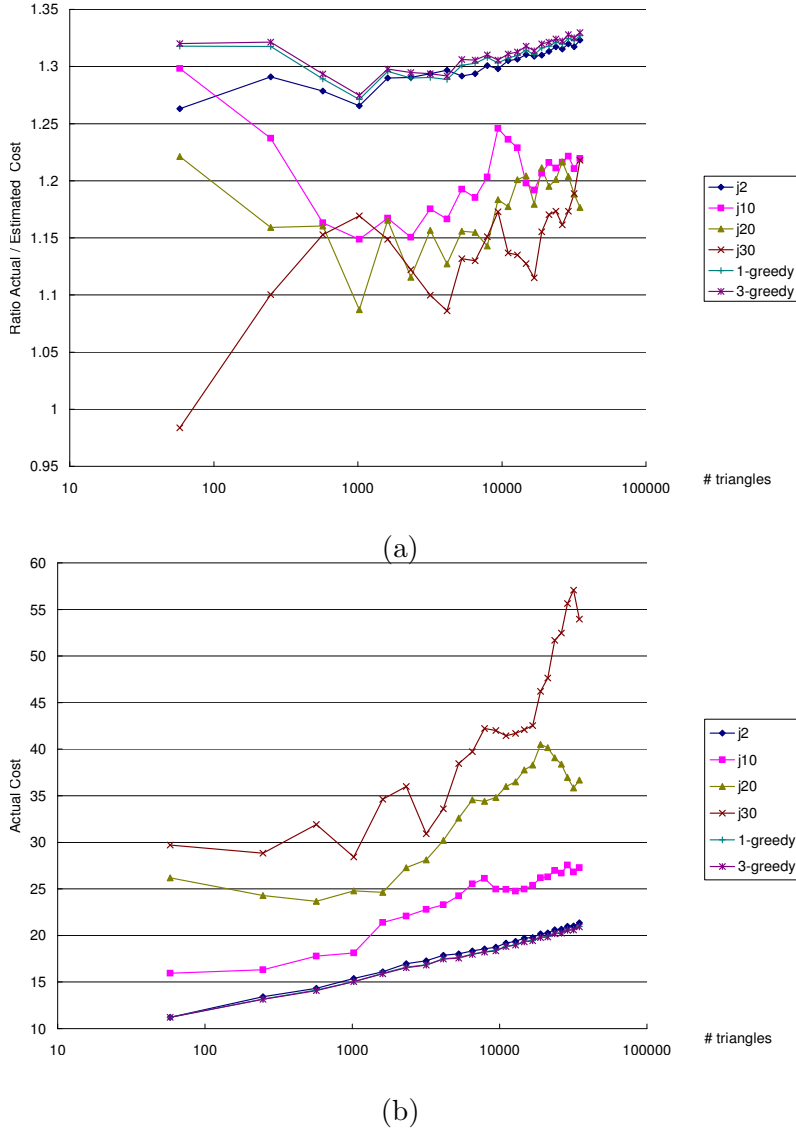
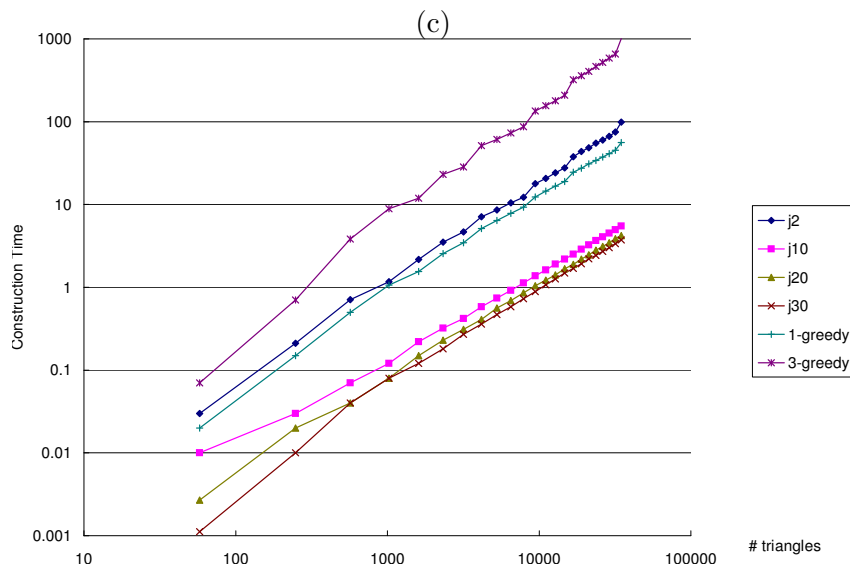
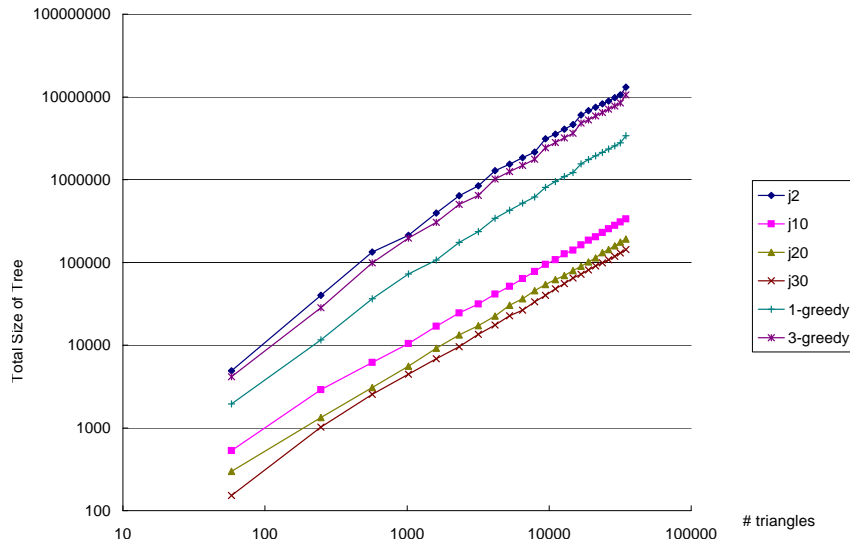


Fig. 1. A comprehensive analysis of the `teapot` family, as a representative dataset family. The number of triangles ranges from 58 to 105 280. For each plot, we display a curve for each termination criterion (`j2`, `j10`, `j20`, `j30`, `1-greedy`, `3-greedy`). (a) Ratio of actual to estimated cost, as a function of the scene size (should be close to one). (b) The effect of the choice of termination criterion on the actual cost.

ratio remains close to one for any dataset-octree combination. In passing, also observe that trees built using `j2` and the k -greedy criteria have a much more consistent correlation (a factor of 1.3) between estimated and actual costs, while larger thresholds cause a rougher estimation of costs and a weaker correlation. Indeed, `j10+` produce much more jagged, random-looking curves, with an actual-to-estimated cost ratio in the interval $[1, 1.3]$.

In Figure 1(b), we can see the effect of greedy criteria on the actual cost: `j10+`



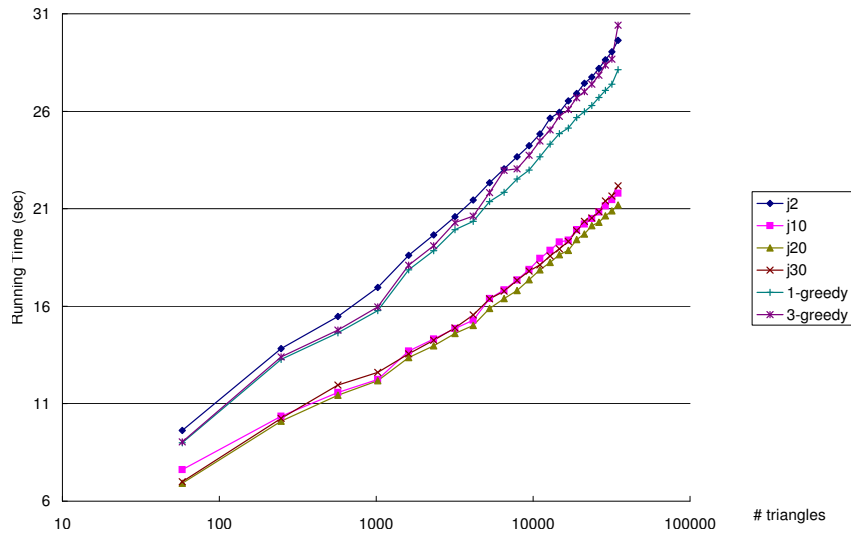
(d)

Fig. 1 — continued.

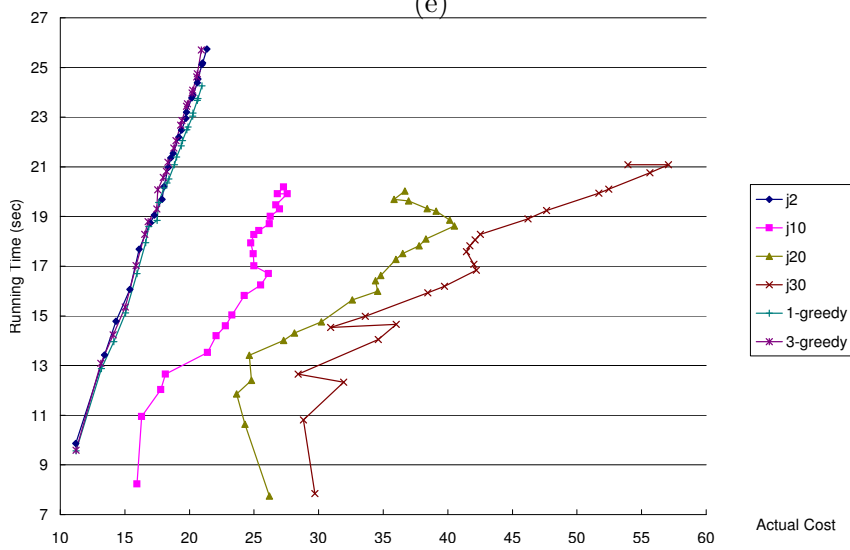
(c) The total memory usage of the octree (number of objects, nodes, and total size of the cells' object lists). (d) Time taken to build the octree.

are inferior with respect to the cost function, whereas j2 (which is completely cost-oblivious) and the 1- and 3-greedy criteria appear to find octrees with similar costs.

In Figure 1(c), we plot the total memory usage of the octree (number $|\mathcal{S}|$ of objects, of nodes, and total size $\sum_i |\mathcal{S}_i|$ of the object lists \mathcal{S}_i for each cell \mathcal{B}_i). We observe that, while the costs of j2 and the greedy criteria are similar, the size of the octrees generated by 3-greedy is about 25 to 30% less than j2, and 1-greedy leads to dramatically smaller trees, by as much as 70%; the figure



(e)



(f)

Fig. 1 — continued.

(e) Time taken to query the octree data structure for all rays in a ray-tracing process. (f) Time taken to query the octree in a ray-tracing process, as a function of the actual cost (should be a common linear relationship).

is in logarithmic scale. The criteria j10+ lead to even smaller trees, but for a cost much worse as shown in Figure 1(b). Compared with Figure 1(b), observe that the order of the curves is reversed, showing clearly that smaller costs are paid by larger storage size.

In Figures 1(d) and (e), we plot the actual time taken to build the octree, and the time taken to query the data structure (traversal and ray-object intersections) for all the rays in a ray-tracing process. We can see an almost

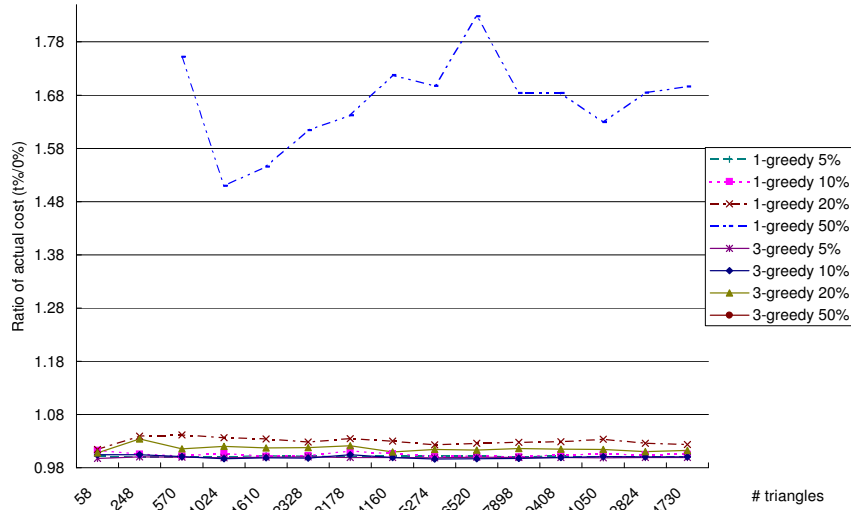
linear dependency of the construction time on the size of the scene, and a logarithmic dependency for the ray-tracing runtime, as expected.

More puzzling is the relation between the time taken to query the octree in a ray-tracing process, as a function of the actual cost (refer to Figure 1(f)): if our theory is correct, we should observe that all octrees lead to the same dependency, which should be linear. Indeed, this is the case for $j2$ and the greedy criteria; but it is clearly not so for $j10+$, which not only exhibit lower runtimes than they should (according to the costs) but also do not correlate linearly with the costs at all! At this point, our best understanding is that the discrepancy between runtimes and costs is due to memory cache performance: these trees being smaller, and having a larger contiguous block of triangles at every leaf, the ray-triangle intersection algorithm better utilizes the memory hierarchy. This would explain the consistent 20% improvement in runtime for the large threshold separation criteria. Note that those improvements must have been offset by the algorithmic inefficacy of those octrees: the cost is higher, and so the runtime must have been longer, yet the memory access pattern is better enough so that overall we still notice an improvement in performance. We could try changing the memory allocator and tuning the memory layout, as was done for kd-trees in [17], but it seems that a lot more would have to be involved to derive cache-efficient versions of our octrees.

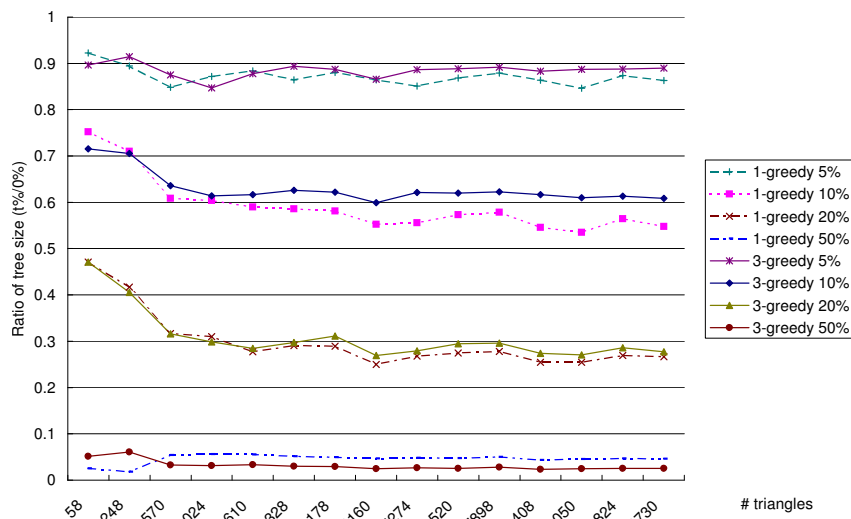
Moreover, memory cache effects cannot explain the jumps and apparent irregularity of the costs and the runtimes, nor their apparent lack of correlation. Since taking a larger threshold introduces rougher granularity in the cost measure, it is possible that those irregularities come from our computation of costs. Taking different values for α and β might actually restore some consistency for those. In Section 4.4, we revisit those experiments while setting the ratio of α to β to a value different from one.

4.3 *Substantial-improvement greedy*

In order to see whether anything could be gained by demanding that the cost be substantially improved for subdividing a node in the greedy approach to constructing an octree, we test the $t\%$ -substantial improvement greedy criteria, for t being 5, 10, 20, and 50, for both 1-greedy and 3-greedy termination criterion. Recall that this involves requiring that subdividing a node produces a reduction of at least $t\%$ in the cost. In Figure 2(a), we plot the ratio of the costs of the trees as a function of the number of triangles in the scene. The closer the ratio is to one, the better. We can see that the cost of 50%-improvement greedy for 1-greedy is much higher, while all other ratios are very close to one. This lends support to our intuition that higher values of t correspond to a cruder heuristic and thus increase the cost, while the data



(a)



(b)

Fig. 2. A comparison of the $t\%$ -improvement variants versus original greedy criteria for the `teapot` family, with t being 5, 10, 20, 50, in both (a) cost and (b) size.

structure size is reduced because it recommends subdivision less often. The latter is confirmed by Figure 2(b), in which we look at the ratio between data structure sizes. The fact that the cost of 50%-improvement greedy for 1-greedy is so much worse than the others is not too surprising, because it is the crudest heuristic and produces the smallest data structure size, showing a trade-off between performance and space.

It is important to observe that using 5%-improvement greedy for both 1-

and 3-greedy, we obtain size gains up to 10% while the cost is essentially unchanged. (See Figures 2(a) and (b).) Therefore, we have decided to use 5%-improvement greedy for *all* the remaining greedy criteria experiments reported in this paper. (It appears that going up to as high as 20%-improvement greedy would still allow us to keep nearly the same costs while gaining even higher space savings.)

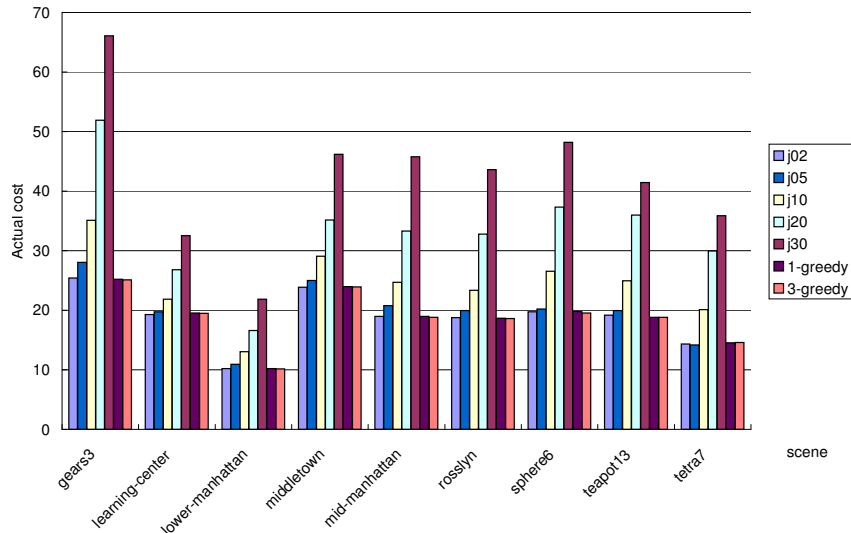
We compare the various greedy criteria (as mentioned, with the 5%-improvement greedy) to the j2+ criterion, for a variety of scenes. (See Figure 3.) We observe the same trends as with the `teapot` scenes, uniformly across the board. In particular, the greedy criteria lead to substantially smaller octrees, sometimes by a factor of ten (note that the y -scale is logarithmic), with no observable loss in the cost.

4.4 *Fine-tuning the cost functions*

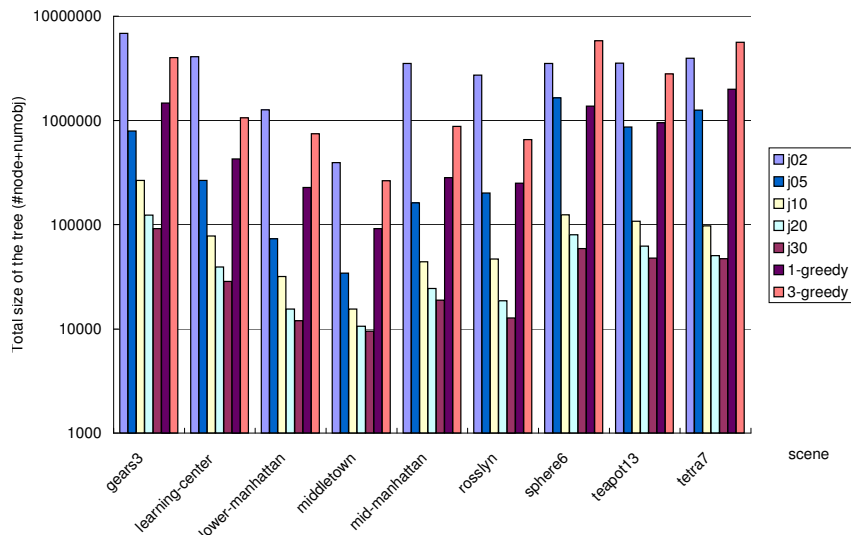
As mentioned above, in the cost measure we have so far considered $\alpha = \beta$, thereby postulating equal cost for ray-box and ray-triangle intersection detection. For a more accurate cost estimation, and for smoothing irregularities such as those encountered with the j10+ criteria (see Section 4.2), estimates of α and β for a given platform need to be actually computed and used in the cost function. There are two natural ways to compute these estimates: analytically and experimentally.

Analytically, we observe that ray-box intersection, as implemented in our program, involves approximately 32 floating-point comparisons, 12 additions, 6 multiplications, and 3 divisions, plus a host of assignment operations. (The numbers may be lower due to short-circuit boolean evaluation, but we observe α to be rather stable in practice.) Ray-object intersections are much more complicated, and use the algorithm of Moller and Trumbore [23]. It turns out that while α is reasonably stable, β actually varies quite a lot depending on the ray-object configuration. For instance, the conditional probability that a ray intersects a triangle, given that it intersects the octree cell, is twice the ratio of the area of the intersection of the triangle with the cell to the boundary area of the cell; the smaller the triangle relative to a cell of the decomposition, the smaller that conditional probability. Computing these actual values at the leaves does not satisfy our simplicity requirement (which more or less dictate that β be some fixed value), and trying to compute the average value of β under some distribution model seems both arbitrary and out of hand. Moreover, such a value would be tied to the ray-object intersection algorithm. For these reasons, we chose not to pursue the analytical approach.

Instead, we will take advantage of the degree of freedom in β to design better



(a)



(b)

Fig. 3. A comparison of performance of two 5%-improvement greedy criteria with cost-oblivious ones on several scenes, in both (a) cost and (b) size.

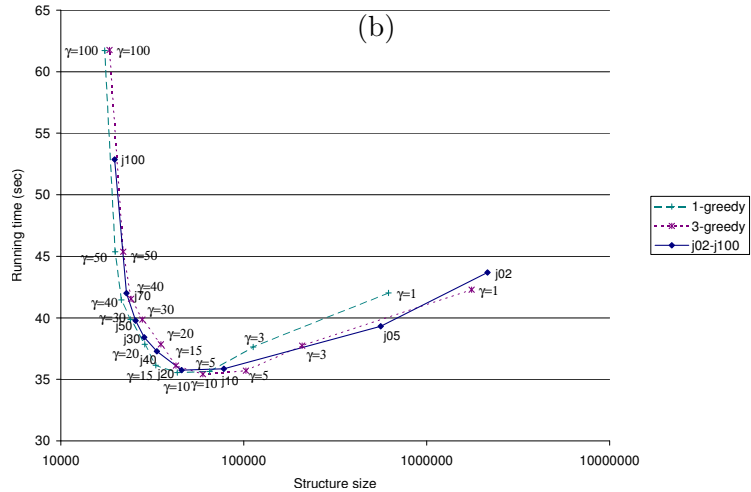
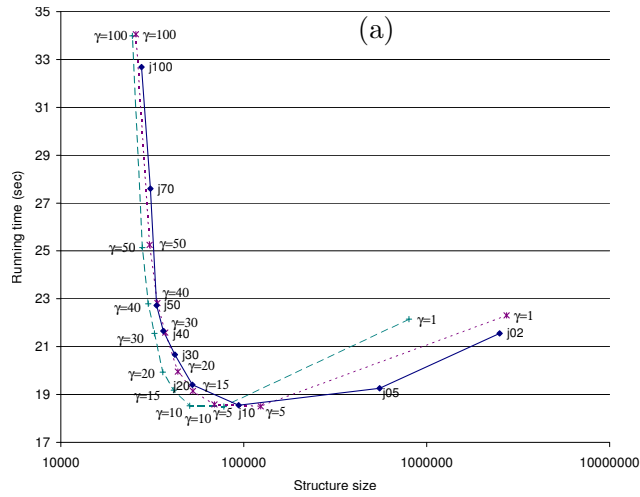
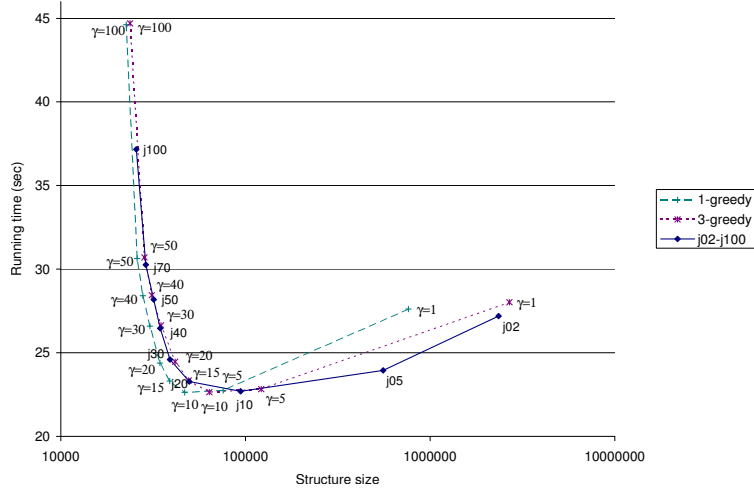
octrees. Experimentally, we have already observed that j10+ lead to smaller trees than both j2 and the greedy criteria, but that the costs substantially increase. Increasing $\gamma = \alpha/\beta$ in the greedy criterion also leads to smaller trees: indeed, this means that for larger γ , we downplay the ray-object intersection cost in the predictor (see Equation (6)), to allow more objects in a box, and hence prefer earlier stopping of subdivision and a smaller tree size. What is interesting here is that while the tree size decreases, we observe that the cost remains approximately the same. The question now becomes: how large should

we take γ to have the smallest tree possible while preserving the near-optimal cost?

Taking the large `dragon` and `happy buddha` scenes, and a moderate `teapot` scene with 4168 triangles, we plot the total size of the octree versus the ray-tracing time (traversal and intersection testing), for various values of γ in Figure 4. First, we observe that among `j10`, `j50` and `j100`, as the `j` value increases, the tree size reduces while the ray-tracing runtime increases, showing a trade-off between space and runtime as expected. It is interesting to see that going from `j10` to `j50`, the tree size reduces dramatically with only a moderate increase in runtime, while going from `j50` to `j100`, the tree size only reduces slightly but the runtime increases at a larger rate. Ideally, we would like to have an octree with “reasonable” tree size, i.e., with tree size much smaller than that produced by `j10`, while having runtime comparable to `j10`. Surprisingly, this is essentially achieved by 1- and 3-greedy, as noticed in our second observation: we see that 1- and 3-greedy all exhibit the advantage of having much smaller trees than `j10`, with ray-tracing runtimes about the same as `j10` for γ between 5 and 20.

To further confirm this surprising result, we take the `teapot` scenes again, revisit Section 4.2, and proceed to set the ratio $\gamma = \alpha/\beta$ to different values (5, 10, 15, 20) and compute the octrees either using the `j` criterion alone or with 1- or 3-greedy (with the 5%-improvement greedy). In the conference version, we did plot the total tree size and the ray-tracing runtime (the construction time is similar to the size with minor variations) as a function of the number of triangles. The plot is not readable as all the curves overlap; instead, we describe here the results in text. Both size and runtime show a linear dependency on the number of triangles of the teapot. For the size, we do observe a cluster of very similar (within less than an order of magnitude), parallel curves, ordered by reverse γ (the smaller γ , the larger the size). For the runtimes, the curves are completely overlapping. This is true for all criteria except `j2`, the latter being both larger and slower than the others by more than an order of magnitude. Note that `j2` has much larger sizes and thus much worse runtimes due to a poor memory cache performance, but `j10+` overlap with the `k`-greedy. Also, the runtime of 3-greedy for $\gamma = 20$ is sometimes a bit slower than the 1-greedy or `j` criteria (or 3-greedy for smaller γ), although the corresponding octree size is unaffected by these variations. Overall, thus, we conclude that the runtimes are essentially unchanged for these different γ values, and yet the size decreases uniformly for increasing γ . (The same can be observed for the `jx` criterion with increasing x .)

There is some reason to believe that there is an ‘optimal’ value of γ which is independent of the scene. In Figures 4(a), (b) and (c), the leftmost lower-left dominating data point, which corresponds to the best runtime with reasonably good tree size, is always 1-greedy with $\gamma = 10$, even though the plots are



(c)

Fig. 4. A comparison of the relative cost of construction and tracing time for different families of trees. Different curves correspond to different termination criteria, while points on the same curve correspond to different choices of the ratio $\gamma = \alpha/\beta$ on the (a) dragon, (b) happy buddha, and (c) teapot scenes, with the 5%-improvement variant of the greedy criteria.

for different scenes (**dragon**, **happy buddha**, and **teapot**). We also tried the following fitting method to decide the best γ . We took different runs of ray tracing on various sizes of the **teapot** and the **tetra** families. For each run, we recorded the total number x of ray-box intersection tests, the total number y of ray-triangle intersection tests, and the total runtime z of the ray-tracing process. Now each run is represented as a data point (x, y, z) in three dimensions; we fit the best plane of the form $z = \alpha x + \beta y$ to the data points. With a total of 328 data points (runs), 238 from **teapot** and 90 from **tetra**, each point weighted equally, we got the best fitting α and β values whose ratio $\gamma = \alpha/\beta$ is about 10. Observe that this value of 10 came from the **teapot** and **tetra** scenes, and yet the value is consistent with the γ value of the left-most lower-left dominating points in Figures 4(a) and (b) mentioned above.

To summarize, with our combination of γ and greedy criterion, when we pick a value of γ larger than 1, the tree size is dramatically reduced (and there is no need for the j criterion—we can just set j to 0). So it appears that the γ value controls the tree size more effectively than the j value. Also, since the greedy criterion performs some optimization according to the *adjusted* cost function (i.e., the new γ value has been incorporated into the cost function), the runtime is much less sensitive to the γ value change. Moreover, perhaps because the performance is insensitive, the ‘optimal’ γ value (for the best runtime and good tree size) is more or less independent of the scenes, unlike j —another advantage of the γ -plus-greedy combination. Of course, the ratio γ also reflects the speed ratio of the two different operations on a given machine, so we cannot choose it to be too far away from the ‘fact.’ (This is where we found the fitting method quite useful.) But again, as long as γ is chosen to be in a reasonable range, the actual runtime does not seem to change much as we vary γ .

5 Conclusion

Although most of the graphs we have presented here are for the **teapot**, **dragon** and **happy buddha** family of scenes (intended to model the effect of a single subdivided manifold), we have carried out analogous experiments for the other scenes mentioned in Section 4.1 and observed similar behavior. Although the scenes we consider are rather small by computer graphics standards due to the limitations of our implementation, our experiments seem to scale with size without changing in conclusions.

One purpose of our experiments was to determine whether the subdivision criterion sometimes used in computer graphics (at most 2 objects per leaf, with a cut-off depth), called j_2 in this paper, was indeed the best possible, with regards to cost and runtime of the traversal and intersection tests, and

size of the data structure. A second purpose was to evaluate the impact of the amount of lookahead on the cost-driven greedy strategy.

All reasonable criteria seem to reach near optimal costs, or at least similar costs. This seems to imply that the approximation ratio of 3-greedy, which is a somewhat large constant in theory [7], is close to one in practice. (Of course, it could be that all these methods approximate the best octree with a common factor greater than one but we find it difficult to believe.) In any case, the real benefit of our methods lies in obtaining much more compact data structures while still keeping the minimal cost.

It is not clear that 3-greedy is an improvement on 1-greedy in practice; in fact, the latter produced comparable costs but smaller trees on the scenes we tried. This is to be contrasted with the worst-case analysis results available which assert that 1-greedy strategy does not have constant approximation ratio whereas 3-greedy does [7]. It seems that 3-greedy is quite competitive if one desires guaranteed results, but that 1-greedy is preferable for its better practical behavior.

What surprised us most is that 1-greedy with a large value of γ is the best criterion in practice, and, while its cost is comparable to j2, the trees produced are much smaller (by a factor of four or five). Thus substantial-improvement 1-greedy is a practical approach to construct an efficient octree, either using the 1-greedy method by itself or combined with other termination criteria. In the worst case, the cost will not increase, but the total size of the tree can be significantly reduced. Moreover the gains in tree size (without losing on the cost either) can be greatly amplified by taking a larger value of $\gamma = \alpha/\beta$. In turn, this leads to being able to process larger scenes, and to better locality of reference.

Thus, it appears that the best criterion is uniformly (for all scenes) the 1-greedy strategy with substantial improvement and large α/β ratio, and that it performs at least comparably to its competitors and sometimes substantially better in *all* of: octree total size (number of nodes and total size of object lists at leaves), construction time, *and* ray-tracing time.

We conclude by mentioning a few directions for further research. One main open question is to further understand the nature of the correlation between the actual cost and the actual runtime. The discrepancies seem to arise from interaction with the memory hierarchy. A cost measure that can cope with external-memory and eventually cache-oblivious models of computation would be a longer term goal.

Secondly, despite our best effort, the “self-tuning octree” we have sought is not entirely free of somewhat arbitrarily fixed parameters. Doing away with them altogether would be an interesting challenge.

Acknowledgments

We wish to thank Steven Fortune of Bell Laboratories for providing some of the test scenes, as well as himself and Marc Glisse for several fruitful discussions and for commenting on and contributing to the third author's implementation.

References

- [1] A. Appel, Some techniques for shading machine renderings for solids, in: *AFIPS Joint Computer Conf. Proc.*, Vol. 32 (AFIPS, Spring 1968) 37–45.
- [2] B. Aronov, H. Brönnimann, A.Y. Chang, and Y.-J. Chiang, Cost prediction for ray shooting, in: *Proc. 18th Annu. ACM Sympos. Comput. Geom.* (ACM, New York, 2002) 293–302.
- [3] B. Aronov and S. Fortune, Approximating minimum weight triangulations in three dimensions, *Discrete Comput. Geom.*, Vol. 21(4) (Springer, New York, 1999) 527–549.
- [4] J. Arvo and D. Kirk, A survey of ray tracing acceleration techniques, in: A.S. Glassner, ed., *An Introduction to Ray Tracing* (Morgan Kaufmann Publishers, Inc., 1989) 201–262.
- [5] H.L. Bertoni, *Radio Propagation for Modern Wireless Systems*, Prentice-Hall, Upper Saddle River, NJ, 2000.
- [6] H. L. Bertoni and S.A. Torrico, Propagation Prediction for Urban Systems, in: L. Godara, ed., *Handbook on Antennas in Wireless Communications* (CRC Press, 2002).
- [7] H. Brönnimann and M. Glisse, Cost-optimal trees for ray shooting, in: *Proc. 6th Latin American Symp. Theoretical Informatics* (Lecture Notes in Computer Science 2976, Springer, 2004) 349–358.
- [8] T. Cassen, K.R. Subramanian, and Z. Michalewicz, Near-optimal construction of partitioning trees using evolutionary techniques, in: *Proc. of Graphics Interface '95* (Canad. Inf. Proc. Soc., Toronto, 1995) 16–19.
- [9] F. Cazals and M. Sbert, Some integral geometry tools to estimate the complexity of 3d scenes, Research report no. 3204 (INRIA, July 1997).
- [10] A.Y. Chang, A survey of geometric data structures for ray tracing, Technical report TR-CIS-2001-06 (CIS Department, Polytechnic University, 2001).
- [11] J.G. Cleary and G. Wyvill, Analysis of an algorithm for fast ray tracing using uniform space subdivision, *The Visual Computer*, Vol. 4 (Springer, New York, 1988) 65–83.
- [12] A. S. Glassner, Space subdivision for fast ray tracing, *IEEE Computer Graphics and Applications*, pages 15–22, October 1984.
- [13] S. Fortune, Algorithms for the prediction of indoor radio propagation, Manuscript (1998). Available at <http://cm.bell-labs.com/cm/cs/who/sjf/pubs.html>

- [14] A. Glassner, Space subdivision for fast ray tracing, *IEEE Comput. Graphics Appl.* (IEEE, 1984) 15–22.
- [15] J. Goldsmith and J. Salmon, Automatic creation of object hierarchies for ray tracing, *IEEE Comput. Graphics Appl.* (IEEE, 1984) 14–20.
- [16] E. Haines, *The standard procedural database (SPD)*, Version 3.13 (3D/Eye, 1992). Home page at <http://www.acm.org/tog/resources/SPD/overview.html>,
- [17] V. Havran, *Heuristic Ray Shooting Algorithms*, Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. Available at <http://www.cgg.cvut.cz/~havran/phdthesis.html>
- [18] M.R. Kaplan, The use of spatial coherence in ray tracing, *Techniques for Computer Graphics* (Springer-Verlag, 1987), 173–193.
- [19] S.C. Kim, B. Guarino, T. Willis, V. Erceg, S. Fortune, R. Valenzuela, L. Thomas, J. Ling, and J. Moore. Radio propagation measurements and prediction using three-dimensional ray tracing in urban environments at 908 Mhz and 1.9 Ghz. *IEEE Transactions on Vehicular Technology*, 48:931–946, 1999.
- [20] K. Klimaszewski, A. Woo, F. Cazals, and E. Haines, Additional notes on nested grids, *Ray Tracing News*, Vol. 10(3) (ACM, New York, 1997). Available at <http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art8>
- [21] J.D. MacDonald and K.S. Booth, Heuristics for ray tracing using space subdivision, *The Visual Computer*, Vol. 6 (Springer, New York, 1990) 153–166.
- [22] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd edition (Springer-Verlag, 1996).
- [23] T. Möller and B. Trumbore, Fast, minimum storage ray-triangle intersection, *J. Graphics Tools*, Vol. 2(1) (ACM, New York, 1987) 21–28. Code available at <http://www.acm.org/jgt/papers/MollerTrumbore97/code.html>,
- [24] D.W. Moore, *Simplicial Mesh Generation with Applications*, Ph.D Dissertation, Cornell University, 1992.
- [25] B. Naylor, Constructing good partitioning trees, in: *Proc. of Graphics Interface '93* (Canad. Inf. Proc. Soc., Toronto, 1993) 181–191.
- [26] E. Reinhard, A.J.F. Kok, and F.W. Jansen, Cost prediction in ray tracing, in: X. Pueyo, P. Schröder, eds., *Proc. Eurographics Workshop, Rendering Techniques '96*, Porto, Portugal (Springer, Berlin, 1996) 41–50.
- [27] S.M. Rubin and T. Whitted, A 3-dimensional representation for fast rendering of complex scenes, *Proc. Computer Graphics (SIGGRAPH'80)*, Vol. 14(3) (ACM, New York, 1980) 110–116.
- [28] H. Samet, *Applications of Spatial Data Structures* (Addison-Wesley, 1990).
- [29] I. Scherson and E. Caspary, Data structures and the time complexity of ray tracing, *The Visual Computer*, Vol. 3(4) (Springer, New York, 1987),

- 201–213.
- [30] K.R. Subramanian and D.S. Fussell, Factors affecting performance of ray tracing hierarchies, TR-90-21, University of Texas at Austin, August 1990.
 - [31] K.R. Subramanian and D.S. Fussell, Automatic termination criteria for ray tracing hierarchies, in: *Proc. of Graphics Interface '91* (Canad. Inf. Proc. Soc., Toronto, 1991) 93–100.
 - [32] I.E. Sutherland, R.F. Sproul, and R.A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys*, Vol. 6(5) (ACM, New York, 1974) 1–55.
 - [33] Stanford University, *The Stanford 3D Scanning Repository*. Home page at <http://www-graphics.stanford.edu/data/3Dscanrep>
 - [34] H. Weghorst, G. Hooper, and D.P. Greenberg, Improved computational methods for ray tracing, *ACM Trans. Graphics*, Vol. 3(1) (ACM, New York, 1984) 52–69.
 - [35] 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*, Vol. 1 (IEEE, 1995), 343–349,
 - [36] T. Whitted, An improved illumination model for shading display, *Comm. ACM*, Vol. 23(6) (ACM, New York, 1980) 343–349.
 - [37] R. Yagel, D. Cohen, and A. Kaufman, Discrete ray tracing, *IEEE Comput. Graphics Appl.*, Vol. 12(5) (IEEE, 1992) 19–28.