Cost-Driven Octree Construction Schemes: An Experimental Study*

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

Many algorithmic problems are interesting to both theoreticians and practitioners, but in a different manner. While the theoreticians have traditionally focused on worst-case scenarios which is often not very useful in practice, the practitioners are sometimes stuck in the hacking culture and arrive at solutions that only work well in a few specific cases. An example of such an algorithmic problem is ray shooting.

Imposing some data structure to support ray-shooting queries usually helps to improve the efficiency of the algorithm. We focus on one such data structure—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. It is difficult to fix a set of parameter values that is good for all possible scenes. One approach to resolve this problem is to construct a data structure which "tunes itself" to the input without using arbitrary preset parameters, so that a single algorithm is suitable for all situations. Surprisingly, only a few investigations have focused on this approach compared to the huge amount of research papers on ray shooting from both the theoreticians and the practitioners. We take some steps in this direction by evaluating several octree construction schemes for use in ray shooting, some widely used in the computer graphics literature (such as bounding the number of objects in a leaf and the maximum depth) and some developed in companion papers as part of this research (cost-driven k-greedy termination criteria). Our experimental results show that the octrees constructed using our schemes are better than those built with *a priori* fixed parameters.

Our octree construction algorithm is driven by a simple cost predictor and has been proven elsewhere to approximate the optimal tree to within a constant factor. We fine-tune the predictor and observe the behavior of our algorithm on octrees built to support a simple ray-tracing engine and compare its performance with those of commonly used alternatives. It appears to work well in practice.

*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'03, June 8–10, 2003, San Diego, California, USA.

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—*ray tracing*

General Terms

Algorithms, experimentation

Keywords

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

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. 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, many heuristic methods introduced by practitioners often work very well in many 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 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 [5]) 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.

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 observe experimentally that the constant factor approximation of [5] is actually very close to 1, but that the 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 the cost be "substantially improved" (by a minimum of t%) which 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 depend on the implementation of the program and on 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

$$C_{\text{total}} = C_{\text{struct}} + C_{\text{inters}} \tag{1}$$

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. [22,31], 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 [22,31]. As the data structures become more complicated, the cost in equation (1) shifts from C_{inters} towards C_{struct}. Kilmaszewski et al. [15] 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 [32]. As the time spent on traversing the data structure becomes significant, optimizing the structure cost becomes important. Once a choice of the data structure is made, the total ray-traversal time varies depending on the parameters used for constructing it. For a hierarchical data structure, the parameters 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. 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 [13] makes an extensive experimental study and concludes that no single data structure is uniformly better than others. Two surveys of data structures used for ray-tracing in practice were presented in [8] and [13].

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 is used widely in traditional theoretical analysis. The size of objects also receives attention in ray-tracing literature in terms of cost estimation. Research shows that the size of objects has more impact on ray-tracing time than the object count [9,21,24]. The "size" of an object can be evaluated by either its volume or its surface area.

Notice that the general cost function (1) does not directly depend on the number of objects in the scene. Instead, the number of objects being tested against is the main factor. Goldsmith and Salmon [11] 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:

$$C_{\text{struct}} = 1 + \sum_i c_i \cdot \Pr(i|\mathit{root}) = 1 + \sum_i c_i \cdot \frac{A(i)}{A(\mathit{root})} \tag{2}$$

where $\Pr(i|\mathit{root})$ 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, $A(\mathit{root})$ is the surface area of *root* node, and c_i is the number of children of node i. Naylor [20] employs a similar idea to estimate the cost of a BSP-tree, but uses volume to calculate the conditional probabilities. Another alternative is introduced by Cleary and Wyvill [9] who use *augmented volume* to estimate the cost of ray shooting in a uniform grid. The augmented volume of an object is the sum of the volumes of grid cells where the object resides.

As mentioned above, the statistics of object distribution reveals certain properties of a scene that can help estimate the cost of ray shooting. Scherson and Caspary [24] discuss several properties of the object distribution and use those properties to analyze ray-shooting cost. Cazals and Sbert [7] 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 [24]. The length of the rays in free space tells us where the objects are or where most of the objects are located. It is the same as the *depth complexity* defined by Sutherland et al. [27].

Constructing a hierarchical search structure for ray tracing can usually improve the overall performance. However, if the hierarchical structure is too deep, one may spend a lot of 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 to the height of a search structure. However, 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. [29] investigate the trade-offs in (1) between the competing factors 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 [11] and by Subramanian and Fussell [26].

Subramanian and Fussell [25] propose to compute C_{struct} as follows: $C_{\text{struct}} = n_{\text{region}} \cdot C_{\text{per\_region}}$, 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 because we may have to traverse the hierarchy from the root to the leaf to find the next region hit by the ray. Reinhard et al. [21] use the surface area of octree cells to compute the weighted average tree depth $\bar{D}$. $C_{\text{per\_region}}$ is then roughly estimated by $\bar{D}$ when moving to the next cell involves descending the tree from the root.

Subramanian and Fussell [26] compute C_{struct} and C_{inters} during the construction of a hierarchy. To estimate the cost more accurately, Reinhard et al. [21] propose an alternative approach, which runs 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. The scene is further divided if the cost keeps decreasing.

MacDonald and Booth [16] use two heuristic functions to estimate the traversal cost of a hierarchy. The first is the *surface area heuristic*. The idea is similar to regional probability. First, C_{struct} is divided into *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, (2) can be rewritten as

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

where n_{int} and n_{ext} are the numbers of internal nodes and external nodes, respectively. Secondly, to calculate C_{inters}, the surface area of an object is approximated by considering the union of leaf nodes where the objects resides. Goldsmith and Salmon [11] use bounding volume to calculate the surface area, and Reinhard et al. [21] use the axis-aligned bounding box of an object to approximate its surface area.

The second heuristic function of [16] is 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 [14]. MacDonald and Booth [16] and Whang et al. [30] 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, the partitioning process stops when certain termination criteria are met. For example, the octree of [30] stops when the number of objects within a node is less than a threshold value. The bintree of [16] stops when equation (3) reaches the minimum.

Other approaches to optimizing the ray-shooting data structure, e.g., using evolutionary methods, have been explored as well [6, 17]; a common problem with such approach is its (lack of) speed.

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, let $A(\cdot)$ denote surface area, and let $\mathcal{B}_i$ range over the cells of the subdivision (in our case, induced by the leaves of the octree). The cost measure 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 $W_{\alpha,\beta}(\mathcal{T}) = \sum_{\mathcal{B}_i}(\alpha + \beta|\mathcal{S}_i|)A(\mathcal{B}_i)$, where the sum again ranges over all the cells $\mathcal{B}_i$ of the subdivision. Thus

$$E_{\alpha,\beta}(\mathcal{T}) = \frac{\sum_{\mathcal{B}_i}(\alpha + \beta|\mathcal{S}_i|)A(\mathcal{B}_i)}{A(\mathcal{B}) + \sum_{s\in\mathcal{S}} A(s)} \qquad (4)$$

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 (4) also accurately predicts the average amount of work required for reporting the first intersection of an average *ray* with the scene. 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 constant factor and thus all the theory remains valid within $O(1)$ factors. In practice, in order to obtain accurate costs, 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 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 the triangle, 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, even for a given platform. Experimentally, we find that the ratio of α to β is roughly between two 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 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. 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* [19], i.e., so 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.

The termination criteria include: a choice of a balanced or unbalanced[1] 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).

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 [5]: the greedy without lookahead (also known as 1-*greedy*) simply recommends to subdivide if the cost measure (as given in (4)) is reduced when a node is subdivided into eight subnodes. With a lookahead (what 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.

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 refer to these variants as the *substantial greedy* criteria. Even more involved 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.

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 lookup for neighbor links [23, pp. 57–110], but using only six 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)[2] and the total size of the object lists $\sum_{\mathcal{B}_i} |\mathcal{S}_i|$ over the leaves. We call this the *octree size*.

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.

The questions to which we seek answers can be posed as: (1) Is the estimated cost correlated to the actual cost? This

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

[2]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.

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 scenes drawn from the Standard Procedural Databases (SPD) [12], as well as a `sphere` model, and two scenes used in computer graphics (`happy buddha` and `dragon` from Stanford Computer Graphics Laboratory [28]). In addition to these scenes, we have used five data sets of an architectural nature communicated to us by Steven Fortune. All of the architectural models have modest size (fewer than $10,000$ polygons) to keep computational costs reasonable. The scenes, their characteristics and the reasons we picked them are described in [2].

4.2 Evaluation of octree construction schemes

We ran 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 of substantial improvement, and (e) several values of the ratio γ of α and β in Equation (4). We have performed over $1,600$ test runs in our experiments, of which 738 contributed to the figures. We remark that, 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. 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.

For each of the dataset-octree combinations, we computed as in our previous paper the ratio of actual to estimated costs (actual being computed using the actual numbers of ray-box and ray-triangle intersections 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 (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 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 j10+ produce much more jagged, random-looking curves, in the interval $[1, 1.3]$.

In (b), we can see the effect of greedy criteria on the actual cost: j10+ are inferior with respect to the cost function, whereas j2 (completely cost-oblivious) and the 1- and 3-greedy criteria appear to find octrees with similar costs.

In (c), we plot the total number of nodes and objects ("octree size"), which should reflect the memory usage of the octree. 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 is in logarithmic scale.

In (d) and (e), we plot the actual time taken to build the octree, and the time taken to traverse the data structure 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 traversal time, as expected.

More puzzling is the relation between the time taken to traverse the octree in a ray-tracing process, as a function of the actual cost: 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 running times 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 running times 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 running time 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 traversal 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 [13], but it seems that a lot more would 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 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.

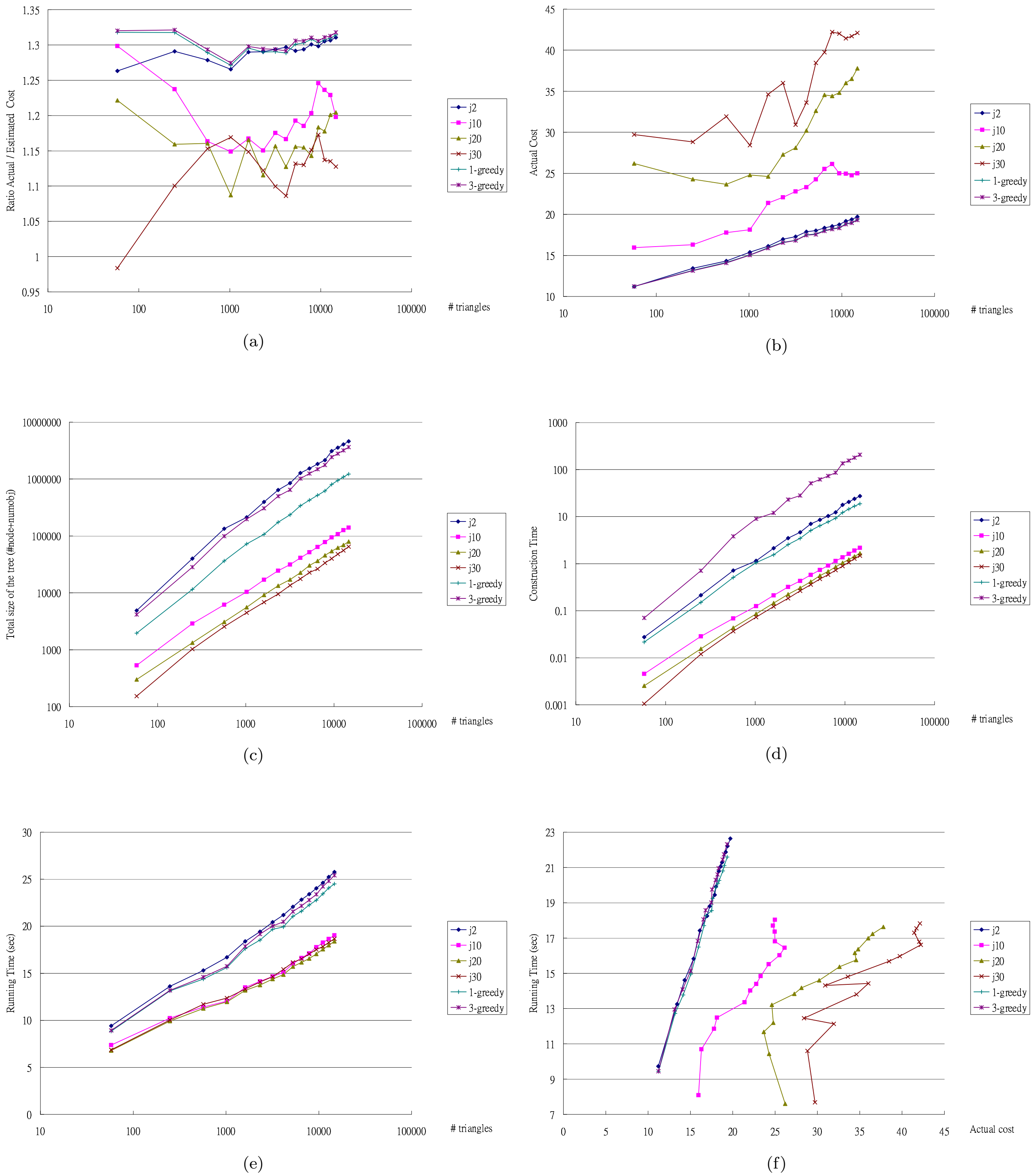
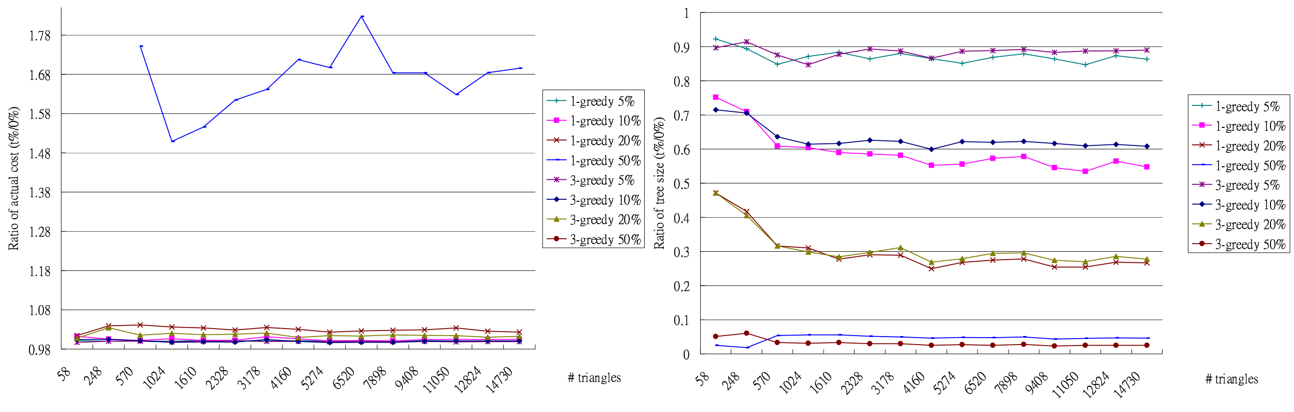
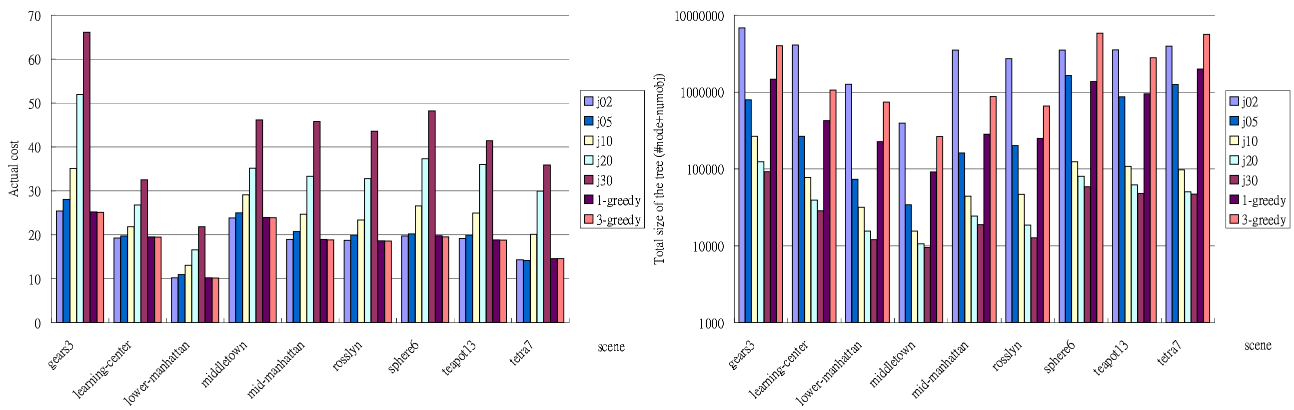


Figure 1: A comprehensive analysis of the `teapot` family, as a representative dataset family. The number of triangles ranges from 58 to 105, 280. For each plot, we display a curve for each termination criterion (j2, j10, j20, j30, 1-greedy, 3-greedy). (a) Ratio of actual to estimated cost, as a function of the scene size (should be close to 1). (b) The effect of the choice of termination criterion on the actual cost. (c) The total number of nodes and objects plus the total size of all object lists ("tree size"). (d) Time taken to build the octree. (e) Time taken to traverse the octree data structure for all rays in a ray-tracing process. (f) Time taken to traverse the octree in a ray-tracing process, as a function of the actual cost (should be a common linear relationship).



(a) (b)

Figure 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.



(a) (b)

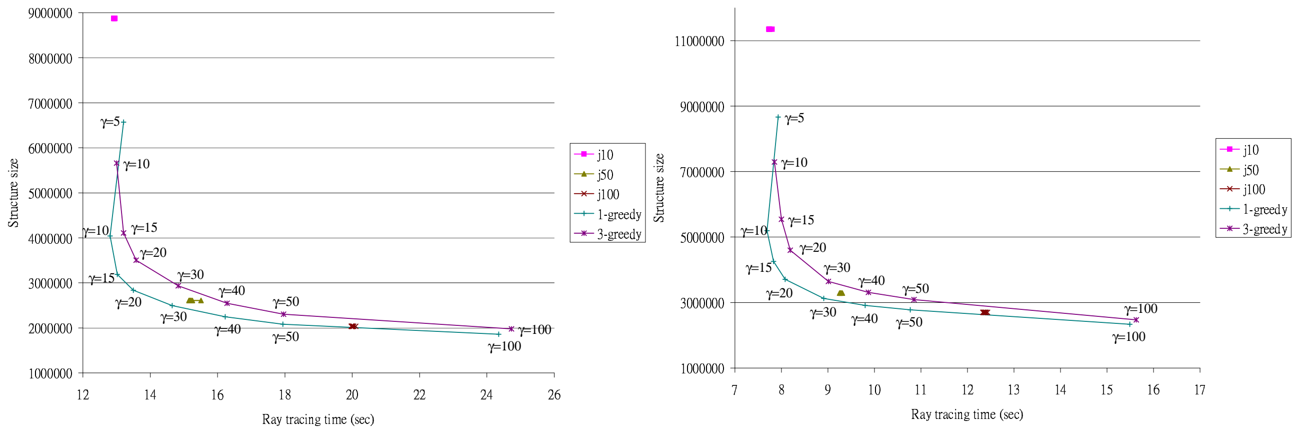
Figure 3: A comparison of performance of two 5%-improvement greedy criteria with cost-oblivious ones on several scenes, in both (a) cost and (b) size.

4.3 Substantial-improvement greedy

In order to see whether anything could be gained by demanding that the cost be substantially improved for subdividing, 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. In Figure 2(a), we plot the ratio of the cost of the t%-improvement greedy to the cost of the 0%-improvement greedy (i.e., the original greedy), as a function of the number of triangles in the scene. The closer the ratio is to 1, the better. We can see that the cost of 50%-improvement greedy for 1-greedy is much higher, while all others are very close to one. This lends support to our intuition that higher values of t corresponds to a cruder heuristic and thus increase the cost, while the data structure size is smaller 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 use 5%-improvement greedy for *all* the remaining greedy criteria. (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+ criteria, 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.



(a) (b)

Figure 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` and (b) `happy buddha` scenes, with the 5%-improvement variant of the greedy criteria.

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 intersection 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. (It could be fewer because of 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 [18]. At this point, 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. We could try and compute this conditional probability as in [10], but then we would only be left with computing the expectation of the area ratio at the leaves, which depends both on the scene configuration and on the octree; such a cost estimate does not satisfy our simplicity requirement. It thus appears infeasible to carry the analytical approach further.

Experimentally, we have already observed that j10+ lead to smaller trees than both j2 and the greedy criteria, but that the costs substantially increase. We can achieve similar effect for the greedy criteria by increasing $\gamma = \alpha/\beta$: indeed, this means that for larger γ, we downplay the ray-object intersection cost in the predictor (see Equation (4)), 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 `dragon` and the `happy buddha` scenes, we plot the total size of the octree versus the ray-tracing time, for various values of γ in Figure 4. First, we observe that among j10, j50 and j100, as j increases, the tree size reduces while the ray-tracing running time increases, with j10 having a substantially larger tree size than others. Moreover, we see that 1- and 3-greedy all exhibit the advantage of having much smaller trees than j10, with ray-tracing running times 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); the plots are labeled '_pγ'. We plot the total tree size and the traversal time (the construction time is similar to the size with minor variations) in Figure 5. Again, the traversal times are essentially unchanged for these different γ values, and yet the size decreases uniformly for increasing γ. Note that j2 has much larger sizes and thus much worse traversal times due to a poor memory cache performance.

There is some reason to believe that there is an 'optimal' value of γ which is independent from the scene. In Figures 4(a) and (b), the leftmost lower-left dominating data point, which corresponds to the best running time with reasonably good tree size, is always 1-greedy with $\gamma = 10$, even though the two plots are for different scenes (`dragon` and `happy buddha`). 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 the ray-box intersection tests, the total number y of the 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)

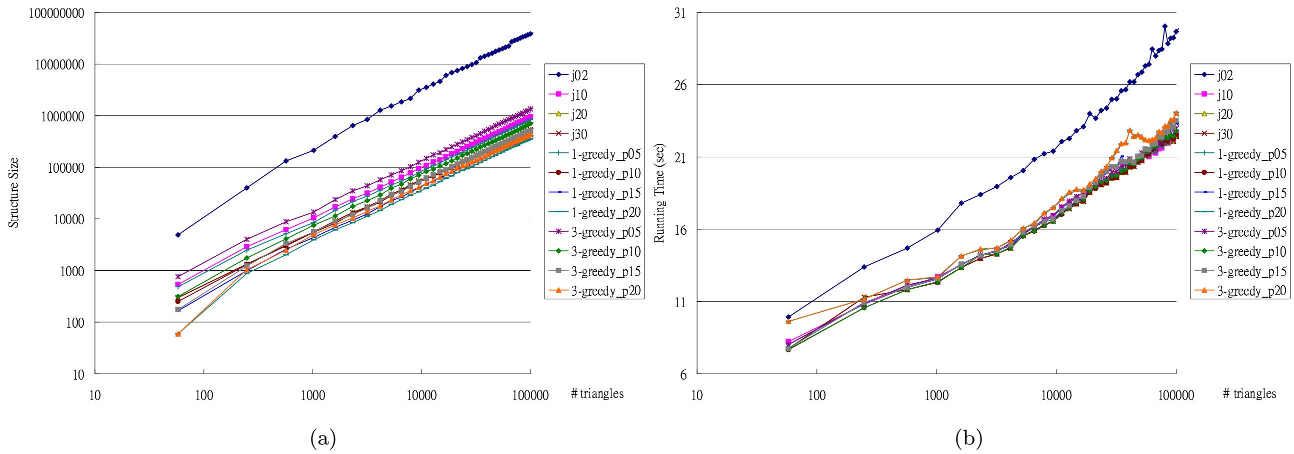


Figure 5: A comparison of several choices of the ratio $\gamma = \alpha/\beta$ on the `teapot` scenes, in both (a) size and (b) ray-tracing times, with the 5%-improvement variant of the greedy criteria. The values of γ are 5, 10, 15 and 20.

in 3D; 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 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 with respect to the γ value change.

5. Conclusion

Although most of the graphs we have analyzed 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 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 j2 in this paper, was indeed the best possible, with regards to cost and running time of the traversal, 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 optimal costs, or at least similar costs. This seems to imply that the approximation ratio of 3-greedy, which is O(1) in theory [5], 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 for the minimal cost.

It is not clear that 3-greedy is an improvement on 1-greedy; in fact, in practice the latter is just as good on the scenes we tried. This is to be contrasted with the theory which asserts that 1-greedy strategy does not have constant approximation ratio whereas 3-greedy does [5].

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 greedy without lookahead 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 won't increase, but the total size of the tree can be 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) 1-greedy strategy with substantial improvement and large α/β ratio, and that it performs at least comparably to its competition 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 and contributing to the third author's implementation.

References

[1] A. Appel. Some techniques for shading machine renderings for solids. In *AFIPS Joint Computer Conference Proceedings*, volume 32, pages 37–45, Spring 1968.

[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.*, pages 293–302, 2002.

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

[4] 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.

[5] H. Brönnimann and M. Glisse. Cost-optimal trees for ray shooting, 2002. Manuscript.

[6] T. Cassen, K.R. Subramanian, and Z. Michalewicz. Near-optimal construction of partitioning trees using evolutionary techniques. In *Proc. of Graphics Interface '95*, May 16–19, 1995.

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

[10] O. Devillers and F.P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.

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

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

[13] 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. http://www.cgg.cvut.cz/~havran/phdthesis.html.

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

[15] K. Klimaszewski, A. Woo, F. Cazals, and E. Haines. Additional notes on nested grids. *Ray Tracing News*, 10(3), 1997. http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art8.

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

[17] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 3rd edition, 1996.

[18] T. Möller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997. http://www.acm.org/jgt/papers/MollerTrumbore97/code.html.

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

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

[21] 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 '96*, pages 41–50. Porto, Portugal, 1996.

[22] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics (SIGGRAPH 80)*, 14(3):110–116, 1980.

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

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

[25] K.R. Subramanian and D.S. Fussell. Factors affecting performance of ray tracing hierarchies. Tr-90-21, University of Texas at Austin, August 1990.

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

[27] 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.

[28] Stanford University. Stanford 3D Scanning Repository. http://www-graphics.stanford.edu/data/3Dscanrep.

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

[30] 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.

[31] T. Whitted. An improved illumination model for shading display. *Communications of the ACM*, 23(6):343–349, 1980.

[32] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer graphics and applications*, 12(5):19–28, September 1992.