Fast Agglomerative Clustering for Rendering

Bruce Walter — Cornell University*
Kavita Bala — Cornell University
Milind Kulkarni — University of Texas, Austin†
Keshav Pingali — University of Texas, Austin

Abstract

Hierarchical representations of large data sets, such as binary cluster trees, are a crucial component in many scalable algorithms used in various fields. Two major approaches for building these trees are *agglomerative*, or bottom-up, clustering and *divisive*, or top-down, clustering. The agglomerative approach offers some real advantages such as more flexible clustering and often produces higher quality trees, but has been little used in graphics because it is frequently assumed to be prohibitively expensive ($O(N^2)$ or worse).

In this paper we show that agglomerative clustering can be done efficiently even for very large data sets. We introduce a novel locally-ordered algorithm that is faster than traditional heap-based agglomerative clustering and show that the complexity of the tree build time is much closer to linear than quadratic. We also evaluate the quality of the agglomerative clustering trees compared to the best known divisive clustering strategies in two sample applications: bounding volume hierarchies for ray tracing and light trees in the Lightcuts rendering algorithm. Tree quality is highly application, data set, and dissimilarity function specific. In our experiments the agglomerative-built tree quality is consistently higher by margins ranging from slight to significant, with up to 35% reduction in tree query times.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;
Keywords: Agglomerative clustering, Bounding volume hierarchy, Lightcuts rendering, Dendogram, Bottom-up tree construction

1 Introduction

Many algorithms in graphics involve repeatedly computing results based on queries of very large data sets. In this paper we use two examples: intersecting rays against the scene geometry and estimating the illumination from many lights. Each case involves many queries per image over data sets that are too large for brute-force evaluations to be practical. Constructing and using hierarchical or multi-resolution representations, such as a binary cluster tree, of these data sets is essential in achieving scalable algorithms with acceptable performance.

A binary cluster tree is a hierarchical partitioning of the data set where the leaves of the tree are the individual data points and each interior node represents a cluster containing all the points in its subtree. The root is thus the cluster containing all the elements in the data set. Each node or cluster typically stores some summary information about the points it contains such as their bounding volume, total strength, etc. During a query, the tree is traversed starting from the root and at each node the cluster summary information is used to decide if further traversal of that sub-tree is necessary for that query. Pruning whole subtrees from the traversal, when possible, greatly accelerates the query performance and enables sub-linear query cost. Different cluster trees built from the same data can vary greatly in quality as measured by average query cost.

*bjw@graphics.cornell.edu, kb@cs.cornell.edu
†{milind,pingali}@cs.utexas.edu

There are three basic approaches to building cluster trees: agglomerative, divisive, and incremental. *Agglomerative*, or bottom-up, clustering starts with the individual data points and progressively groups these together into larger and larger clusters until the root cluster containing all the data points is formed. This is typically done using a greedy approach that groups the two most similar clusters together at each step based on a user-supplied cluster dissimilarity function. *Divisive*, or top-down, clustering starts with the root cluster and progressively splits clusters to form smaller clusters until individual data points are reached (the leaves are clusters of size 1). The splits may be chosen by a simple heuristic such as uniform or median splitting, or may use a cost function to select from multiple candidate splittings at each step. *Incremental* clustering starts with an initial simple tree and progressively adds points to it to form new trees until all the data points have been added, while trying to maximize some measure of tree quality at each step.

Divisive clustering is typically the fastest tree build strategy and the easiest to parallelize. Agglomerative clustering allows more flexibility because it allows the user to supply an arbitrary function defining what constitutes a good pair to cluster together. This is especially convenient for data combining different types of properties and in higher dimensions. Incremental clustering is useful when not all the data points are initially known, but typically builds lower quality trees and will not be considered further here.

Agglomerative clustering is much less commonly used in graphics than divisive clustering. One major reason is that it is often assumed to be $O(N^2)$ or worse and thus prohibitively expensive for large data sets. In this paper we present efficient algorithms for agglomerative clustering including a novel locally-ordered one that outperforms more traditional heap-based versions. With these algorithms, agglomerative clustering becomes feasible even for very large data sets. We also show that trees built using agglomerative clustering are often of higher quality than those built by divisive methods. Cluster tree quality is very application-specific, so we analyze the costs and benefits of agglomerative clustering in two applications: bounding volume hierarchies and Lightcuts [23, 21].

In the next section, we briefly discuss some related work. Then we discuss general algorithms for agglomerative clustering in Section 3 and present our locally-ordered algorithm that is faster than traditional heap-based ones and easier to parallelize. Section 4 applies agglomerative clustering to the bounding volume hierarchies for ray casting and analyzes its performance. Section 5 presents our second example application: light cluster trees in the Lightcuts framework. Finally we conclude in Section 6.

2 Related Work

Agglomerative clustering is used in a wide variety of applications and fields (e.g., [9, 4, 5, 8, 3]) including data mining, compression, and bioinformatics. In some fields the cluster tree is called a *dendogram*. Its popularity is largely due to its ability to use arbitrary clustering dissimilarity or distance functions and the conventional wisdom that it produces higher quality trees than divisive or incremental approaches. Many papers have been devoted to fine-tuning the clustering function for specific applications or data types.

The simplest agglomerative clustering algorithm is $O(N^3)$ and clearly inefficient. More efficient algorithms are known, depending on the nature of the data and clustering metric, and $O(N^2)$ is typically reported as the complexity [16] of agglomerative clustering.

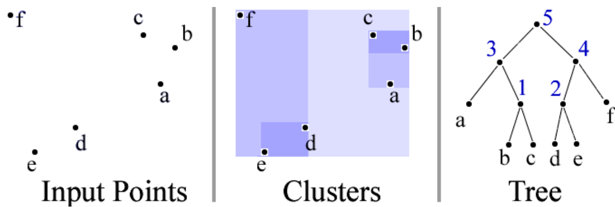


Figure 1: Agglomerative clustering starts with a set of inputs (left) and progressively forms them into larger clusters (center) based on similarity (as defined by a user-supplied dissimilarity function) to form a hierarchical clustering tree (right). In this simple 2D example, the numbers correspond to the order in which the clusters are created by standard greedy agglomerative clustering.

This is too slow for large data sets and hence approximations such as random data sub-sampling are often recommended [8]. For our example applications, we demonstrate that our optimized agglomerative clustering runs in sub-quadratic, and closer to linear, time and thus, can be directly applied to large data sets.

Our first example application is ray casting using hierarchical bounding volumes. Ray casting computes the geometry intersected by a ray and is the fundamental, and typcially most expensive, operation in ray tracing algorithms. A wide variety of acceleration structures have been proposed for accelerating ray casting including octrees, uniform grids, kd-trees, and hybrids[1, 20, 14]. In this paper we will use bounding volume hierarchies (BVH) which have received much interest lately [13, 19]. The first automated BVH construction algorithm combined a surface area heuristic for measuring cluster quality and an incremental clustering approach [7], but current methods use a divisive clustering approach instead, because it builds higher quality trees [18].

Our second example application is Lightcuts [23, 21], a scalable high-quality rendering techniques that accurately approximates the illumination from many point lights using a light cluster tree. The original Lightcuts work used agglomerative clustering to build the light tree but provided few details on how to do this efficiently [22, 15]. In this paper we provide more details on efficient light tree building and evaluate the quality of the agglomerative clustering trees as compared to some alternative divisive clustering strategies.

3 Fast Agglomerative Clustering

Agglomerative clustering is a greedy algorithm that takes a set of points, which may combine geometric and non-geometric properties, along with a cluster dissimilarity function and builds a binary clustering tree. The data points are initially considered clusters of size 1. At each step, it selects the best (e.g., most similar or closest) pair of clusters that are not yet part of a larger cluster and combines them into a single larger cluster. The process repeats until a single cluster containing all the data points is created (i.e. the root node of the cluster tree). A simple 2D example is shown in Figure 1.

The cluster *dissimilarity* function $d(A,B)$, also sometimes called the *distance*[1] function, measures how dissimilar two clusters are. It is assumed to be symmetric but can otherwise be arbitrarily defined. Some simple examples are the maximum distance between any two points in the clusters A and B, the volume of the convex hull of the union of A and B, or the distance between the centroids of A and B. Conceptually at each step we consider dissimilarity metric over all pairs of active clusters and select the pair with the smallest value of $d(A,B)$ and group them together into a single larger cluster. Pseudocode for a simple, naive implementation is shown in Figure 2. This naive version runs in $O(N^3)$ time where N is the number of

```
1: Set active = InputPoints
2: while( active.size() > 1) do {
3:   double bestD = infinity;
4:   Cluster left = null, right = null;
5:   foreach A in active do {
6:     foreach B in active do {
7:       if ((A != B) and (d(A, B) < bestD)) {
8:         bestD = d(A, B);
9:         left = A;
10:        right = B;
11:      }
12:    }
13:  }
14:  active.remove(left);
15:  active.remove(right);
16:  active.add(new Cluster(left,right));
17:}
```

Figure 2: Pseudocode for naive $O(N^3)$ agglomerative clustering.

input points and is clearly inefficient as it discards all the computed dissimilarity information between executions of the outer loop.

3.1 Heap-based implementation

We can greatly improve the efficiency of the agglomerative clustering by adding two acceleration structures to the algorithm: a kd-tree to accelerate the search for the best match for any given cluster, and a min-heap to preserve and reuse dissimilarity information across outer loop iterations. Pseudo-code for this efficient heap-based implementation is shown in Figure 3.

The kd-tree is an extension of the *active* set from the naive algorithm (so it supports add and remove operations) optimized to answer findBestMatch(A) queries quickly. We define findBestMatch(A) to return the cluster B that minimizes $\mathrm{d}(A,B)$ over all clusters in the kd-tree with the restriction that B is not the same as A. This is equivalent to the innermost loop in the naive algorithm. To answer these queries efficiently, the kd-tree is itself a hierarchical clustering tree, where the leaves contain the current active clusters and the interior nodes contain enough summary information to compute a lower bound on the dissimilarity metric to any element within their subtree. The query then performs a top-down traversal, always visiting the nearer child first, and at each interior node computes a lower bound on the dissimilarity metric and only recurses into its sub-tree if it could contain a better match than the best found so far.

It may seem counter-intuitive that we use one hierarchical clustering tree of the data to build another. The idea is to use a simple-to-build, but lower quality, clustering tree to bootstrap the construction of a higher quality tree. Our kd-tree is built using a fast divisive, or top-down, construction that simply splits the longest axis of the node's bounding box in the middle. Points in the kd-trees used in this paper are 3D points with some extra application-specific properties and each kd-tree node stores a tight axis-aligned bounding box of the points in its sub-tree plus enough summary information about the application-specific properties to be able to compute bounds on the dissimilarity function $\mathrm{d}(A,B)$.

The minimum heap stores the best match for each active cluster along with the corresponding dissimilarity value. This allows us to retrieve the globally best matching pair (removeMinPair() operation) in $O(\log N)$ time. Note that once we create a cluster (lines 14-19), we may invalidate some pairs in the heap that involved either of the two clusters being removed. We could immediately scan through the heap to find and update such invalidated pairs (e.g., [8]), but we have found it to be faster to update the heap lazily. Instead each time we remove an element from the heap, we check to see if either of the clusters in the pair has already been incorporated into a larger cluster and update the pair if needed. This is safe because

[1] The dissimilarity or distance function need not correspond to Euclidean distance and frequently does not obey the triangle inequality required of mathematical metrics.

```
1: KDTree kd = new KDTree(InputPoints);
2: MinHeap heap = new MinHeap();
3: foreach A in InputPoints do {
4:   Cluster B = kd.findBestMatch(A);
5:   heap.add(d(A,B), new Pair(A,B));
6: }
7: while( kd.size() > 1 ) {
8:   Pair <A,B> = heap.removeMinPair();
9:   if (! kd.contains(A) ) {
10:    //A was already clustered with somebody
10:  } else if (! kd.contains(B) ) {
11:    //B is invalid, find new best match for A
11:    B = kd.findBestMatch(A);
12:    heap.add(d(A,B), new Pair(A,B));
13:  } else {
14:    kd.remove(A);
15:    kd.remove(B);
16:    Cluster C = new Cluster(A,B);
17:    kd.add(C);
18:    Cluster D = kd.findBestMatch(C);
19:    heap.add(d(C,D), new Pair(C,D));
20:  }
21:}
```

Figure 3: Pseudocode for efficient heap-based agglomerative clustering. Empirical performance is better than $O(N^2)$ and closer to linear scaling with input size (e.g., see Figure 9).

each globally best pair will generally appear in the heap twice (once as $<A,B>$ and once as $<B,A>$) and even if one has a stale dissimilarity value, the one with the most-recently-formed cluster first will always have an up-to-date dissimilarity value.

In our experiments, this heap-based agglomerative clustering has sub-quadratic and almost-linear performance and can be used even on large data sets (e.g., see Figure 9). However it is noticeably slower than the new algorithm we present next.

3.2 Locally-ordered or heap-less implementation

The greedy aspect of agglomerative clustering imposes a global ordering on the creation of the clusters, but it is possible to build exactly the same tree while creating the individual nodes in a different order. For example, in Figure 1 we could create node 2 before node 1 while still creating exactly the same tree. This is the inspiration for our novel locally-ordered agglomerative clustering algorithm.

In principle, we can immediately group two clusters together whenever we can prove that the greedy algorithm would also group those same two clusters eventually. This is easy to prove if the dissimilarity functions obeys a *non-decreasing property* defined as:

$$\mathrm{d}(A,B) \leq \mathrm{d}(A \cup C, B) \quad (1)$$

for all sets/clusters A,B,C where $\cup$ is the set-union or clustering operator and d is the dissimilarity function. Many common dissimilarity functions[2] obey this property including the ones used in the examples in this paper. With this property, if two clusters, A and B, agree that they are each other's best match among all the current clusters, then it is impossible for any future grouping of the other clusters to create a better match for either. Thus the greedy algorithm must eventually cluster A and B together and it is safe for us to cluster them immediately. Pseudocode for this locally-ordered agglomerative clustering is shown in Figure 4.

This locally-ordered version consistently outperforms the heap-based version because it eliminates the overhead of maintaining the min-heap and increases the spatial and temporal locality in the kd-tree searches, despite performing more of them. The heap-based

[2]This property holds for "size"-type dissimilarity functions such as maximum point-wise distance or volume of the combined convex hull, but fails for some other common ones such as distance between cluster centroids.

```
1: KDTree kd = new KDTree(InputPoints);
2: Cluster A = kd.getAnyElement();
3: Cluster B = kd.findBestMatch(A);
4: while( kd.size() > 1 ) {
5:   Cluster C = kd.findBestMatch(B);
6:   if (A == C) {
7:     kd.remove(A);
8:     kd.remove(B);
9:     A = new Cluster(A,B);
10:    kd.add(A);
11:    B = kd.findBestMatch(A);
12:  } else {
13:    A = B;
14:    B = C;
15:  }
16:}
```

Figure 4: Pseudocode for locally-ordered agglomerative clustering. This algorithm assumes the dissimilarity function d is non-decreasing but is empirically faster than the heap-based algorithm.

version forms clusters in a global ordering regardless of their locality, while the locally-ordered one proceeds exclusively from neighbor to nearest neighbor. This increases the memory locality in subsequent searches which is especially important in large data sets.

In the exceptional case that three clusters are exactly equally good matches (i.e. $\mathrm{d}(A,B) = \mathrm{d}(B,C) = \mathrm{d}(C,A)$), the pseudocode could possibly run into an infinite loop, but this is easily fixed. One can change the condition on line 6 to allow A and B to be clustered as long as C is no better of a match (i.e. $\mathrm{d}(A,B) \leq \mathrm{d}(B,C)$). Note that in this specific case, the results of the agglomerative clustering is under-determined and implementation dependent so the different agglomerative algorithms may not produce exactly the same tree.

3.3 Parallelizing agglomerative clustering for multicore

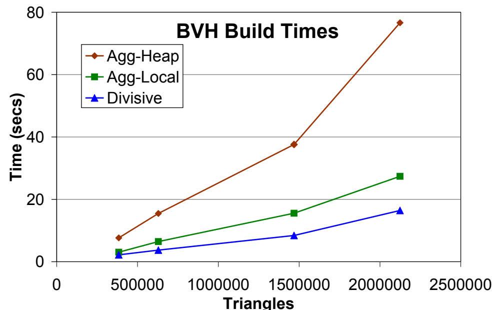
As a greedy algorithm, agglomerative clustering may seem inherently serial in nature. The naive $O(N^3)$ algorithm is trivial to parallelize but uninteresting. The efficient kd-tree based algorithms are challenging since each iteration potentially has data dependencies with the prior ones. In previous work [12, 10, 11], we showed that there is exploitable parallelism in the heap-based and locally-ordered algorithms using the Galois optimistic approach.

Divisive clustering with its divide-and-conquer approach is easy to parallelize; once a node is split into two sub-clusters, they do not interact and can constructed in parallel. In agglomerative clustering, subsequent clusters are often, but not always, independent and can be constructed in parallel. This lends itself to optimistic parallelism, where we attempt to form many clusters in parallel, while checking for conflicting operations and undoing operations when necessary. Because each iteration updates shared data structures (i.e. the heap and kd-tree), memory-location-based conflict checking is likely to be expensive and report many false conflicts. Instead we check conflicts at a higher semantic level on each access to the shared data structures and conflicts are rare (around 0.1%).

We originally created the locally-ordered algorithm when searching for more easily parallelized versions of agglomerative clustering, and we were pleasantly surprised to discover that it is also faster than the heap-based version even in purely serial execution. Parallelizing the locally-ordered version is briefly discussed in [11] where we achieved a speedup of 2.3x on 4 cores. With further improvements we can now get around a 3x speedup on 4 cores, but all the results in this paper are for single threaded clustering.

4 Bounding Volume Hierarchies for Ray Casting

Our first example application for agglomerative clustering is building bounding volume hierarchies for ray casting. Ray casting is the



Model	Triangles	Agglomerative		Divisive
		Heap-based	Locally-ordered	
Kitchen	384034	7.7s	3.1s	2.2s
Tableau	628945	15.5s	6.5s	3.7s
GCT	1468407	37.6s	15.6s	8.4s
Temple	2123971	76.6s	27.4s	16.4s

Figure 5: Bounding volume hierarchy (BVH) tree build times for four different models using the heap-based and locally-ordered agglomerative clustering algorithms and the divisive algorithm.

Clustering Method	Random Line Cost Statistics			Image Times	
	E(boxes)	E(tris)	E(cost)	Eye	Shadows
Kitchen: 384834 triangles					
Agglomerative	42.3	15.0	36.1	2.0s	32.3s
Divisive	61.8	15.2	46.1	2.5s	49.2s
Tableau: 628945 triangles					
Agglomerative	21.4	4.8	15.5	1.5s	15.8s
Divisive	25.7	4.9	17.7	1.5s	16.4s
Grand Central Terminal: 1468407 triangles					
Agglomerative	79.0	14.5	54.0	2.0s	29.0s
Divisive	111.6	14.7	70.5	2.5s	35.1s
Temple: 2123971 triangles					
Agglomerative	31.1	7.0	22.6	2.0s	32.2s
Divisive	43.7	7.5	29.4	2.6s	41.2s

Figure 6: BVH tree quality statistics for agglomerative and divisive clustering for four models. Note that both agglomerative methods build the same tree and thus have the same per-ray performance. Using the surface area heuristic we compute the expected number of box and triangle intersection for a random line and its expected cost as a measure of BVH tree quality. We also give the time to generate two 1280x960 images as additional measures of BVH quality; the *eye* image traces only one eye ray per pixel while the *shadows* image uses 16 eye rays and 16 shadow rays per pixel.

problem of finding the intersections between a ray and all the geometric primitives, such as triangles, in the scene. Depending on the usage, one may want the first, any, or all intersections, but herein we will assume we want the first intersection. This can be naively implemented by testing every triangle for ray intersections but that is prohibitively expensive. A variety of acceleration structures have been proposed and we will consider bounding volume hierarchies (BVH) with a binary BVH tree using axis-aligned bounding boxes.

Ray casting performs a top-down traversal of the BVH tree where at each node it tests the ray against the node's axis-aligned bounding box and only recurses to its children if the ray intersects its box. The leaf nodes are the geometric primitives and these are tested for intersection only if the ray intersected all its parent boxes. Typically only a tiny fraction of the triangles will be tested against a ray resulting in much faster queries. Tree quality can be measured by how many intersection tests are needed for an average ray.

The probability that a random line will intersect a convex volume is proportional to its surface area [7]. This is the basis of the surface area heuristic (SAH) which is used to construct a simple cost-model used as a heuristic to construct good BVH trees. The current standard method for fast construction of high quality BVH trees is a greedy divisive approach [19]. We have implemented the binned BVH construction described in [18] which is reported to build high quality BVH trees quickly. At each step it splits the node along its longest axis and chooses the lowest cost split from 16 candidates.

Following the surface area heuristic[3], we define the dissimilarity function for BVH agglomerative clustering as: $d(A,B)$ equals the surface area of the bounding box of $A \cup B$. Its easy to show that this obeys the non-decreasing requirement of Equation 1.

For the kd-tree used in construction, we consider each cluster to be a 3D point defined by the center of its axis-aligned bounding box plus some extra properties (i.e. the size of its bounding box in each axis). Each kd-tree node keeps a bounding box for its cluster's centers plus their minimum box sizes in each axis. Given a query findBestMatch(A), we can quickly compute a lower bound on the bounding box of $A \cup B$ for any cluster B in this node's subtree. Its x axis box size must be at least the minimum x separation between A's center and the bounding box of cluster centers plus $0.5*(A.sizeX + \min(B.sizeX))$, and similarly for minimum box sizes in the y and z axes. The surface area of this minimum box is then a lower bound on $d(A,B)$ and if it is larger than the best match found so far, we stop traversing this subtree. Otherwise we recurse to its children starting with the nearer one (i.e. one on the same side of the kd-node's splitting plane as A).

[3]This assumes the cost of a BVH box intersection is roughly constant. If using BVH boxes with variable number of children or primitives of strongly variable cost, one might want a more complicated dissimilarity function.

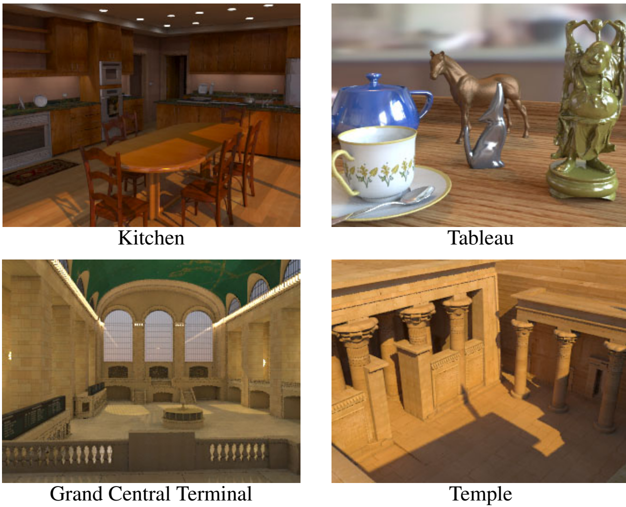
4.1 BVH results

All results are computed on a 3GHz Intel Core2 workstation with four cores. The reported tree building times are single threaded while the ray tracing and image generation components run on 4 threads. All code was written in Java and run using the Sun 1.6 Server JVM. We have not applied low-level optimizations such as packet ray tracing, cache alignment, and SIMD optimizations, so our absolute performance is not as good as that reported for the most highly optimized systems (e.g., [18]), but should be sufficient for comparing the relative merits of different BVH tree building strategies. We tested on four different scenes shown in Figure 7 that were used in the original Lightcuts paper [23].

Figure 5 shows the times to build the BVH trees for four complex models using agglomerative clustering with the heap-based and locally-ordered algorithms and our implementation of the divisive clustering from [18]. Our locally-ordered version is consistently much faster than the heap-based agglomerative clustering and within a factor of two of the divisive tree builder.

Some statistics on BVH tree quality and its effect on the per-ray costs are shown in Figure 6. Using the surface area heuristic, we compute the expected number of box and triangle intersections for a random line based on the surface areas of the clusters in the BVH tree. We combine this into an expected ray cost figure by assuming that ray-box and ray-triangle intersection costs are 0.5 and 1.0 respectively. This expected cost is a good measure of BVH tree quality. We also recorded the time to trace two 1280x960 images to see how well predicted BVH quality correspond to actual ray costs. The *eye* image traces one eye ray per pixel while the *shadows* image traces 16 eye rays and 16 shadow rays per pixel. These timings do not include the build time.

We can see that the agglomerative clustering produces higher quality trees but the results are highly dependent on the model. In one scene (Tableau) the quality is nearly the same while in the other scenes the agglomerative per-ray costs are significantly lower. Whether the improvement in tree quality is worth the extra build time is highly dependent on application. If we are tracing many



Kitchen | Tableau | Grand Central Terminal | Temple

Figure 7: Lightcuts renderings of our four test scenes.

Clustering Method	Preprocess Build	Preprocess Total	Per Image Stats Avg Cutsize	Per Image Stats Time	Summed Time
Kitchen					
Agg-Heap	5.1s	7.7s	644.4	129.4s	137.1s
Agg-Local	3.2s	5.7s	643.6	132.7s	138.4s
Divisive-6D	0.5s	3.0s	740.6	157.5s	160.5s
Divisive-3D	0.4s	2.9s	806.8	162.5s	165.4s
Tableau					
Agg-Heap	5.0s	7.3s	313.1	67.3s	74.3s
Agg-Local	3.0s	5.2s	314.2	66.7s	71.9s
Divisive-6D	0.4s	2.6s	351.5	73.8s	76.4s
Divisive-3D	0.4s	2.6s	358.6	75.4s	78.0s
Grand Central Terminal					
Agg-Heap	7.4s	11.1s	817.0	152.8s	163.9s
Agg-Local	4.9s	7.8s	820.3	152.7s	160.5s
Divisive-6D	0.4s	2.8s	913.7	175.7s	178.5s
Divisive-3D	0.4s	2.7s	1121.8	193.7s	196.4s
Temple					
Agg-Heap	5.3s	7.2s	438.2	53.9s	61.1s
Agg-Local	3.1s	5.0s	435.8	53.8s	58.8s
Divisive-6D	0.4s	2.3s	544.1	66.4s	68.7s
Divisive-3D	0.4s	2.3s	494.3	59.0s	61.3s

Figure 8: Lightcuts data for the four test scenes. We report the time to build the 200 000 point indirect tree as well as the total preprocess time which includes this build. We also report the average cutsize which is the average number of lights evaluated per pixel and the time to compute the pixels of a 640x480 image with 16x anti-aliasing and the summed time for the image including the preprocess.

rays per pixel or can reuse the tree across many frames then the higher quality tree is likely to produce net computational savings. If we cannot amortize the extra build cost across many rays, then a faster build time outweighs any improvements in quality and agglomerative clustering is not warranted.

5 Light Cluster Hierarchies for Lightcuts

Our second example application is building light cluster hierarchies for the Lightcuts [23] rendering algorithm. Lightcuts provides a scalable method to accurately approximate the illumination from many points lights that can be used to unify and solve a variety of difficult illumination problems including area lights, HDR environment lighting, sun/sky models, and indirect illumination. It constructs a hierarchical tree of light clusters and for each point to be illuminated it performs a top-down traversal of this light tree. For each cluster visited it computes a cheap estimate of the illumination from that cluster along with an error bound on this approximation and will only recurse to the node's children if the error bound exceeds a perceptually-based threshold. Thus it can accurately approximate the illumination from many lights while only actually evaluating a small fraction of them; the number of lights actually evaluated is called the cutsize.

The original Lightcuts work built the light tree using agglomerative clustering though it did not describe how to do this efficiently. In fact, it used the heap-based algorithm discussed earlier. The point lights are characterized by a 3D position, a direction (the local surface normal), and an intensity. Clusters are characterized by an axis-aligned bounding box over spatial positions, a bounding cone over directions, and the summed intensity of all their lights. Clusters also randomly select some of their contained lights as their representatives to use when approximating their illumination.

Axis-aligned boxes provide a very convenient way to bound spatial positions, but bounding cones are harder to efficiently compute incrementally. If the bounding cone of $A \cup B$ is only computed from the bounding cones of A and B then the computed bounding cones become increasingly loose higher up in the tree. Alternatively we could use $O(N)$ approximate [17] or exact bounding cone methods but this is expensive for nodes with many children. To get around this issue we use an $O(1)$ technique adapted from [2]. If we treat the directions as points on the 3D unit sphere and have a bounding sphere for these points, then we can easily find a bounding cone from the intersection of the bounding sphere and the unit sphere. The intersection of the two spheres is a circle and the cone (with its apex at the origin) through this circle is a bounding cone for the directions. Finding exact bounding spheres [6] is $O(N)$, so instead we keep track of an axis-aligned bounding box for the points on the unit sphere and use its circumscribed sphere as the bounding sphere. We have found this to be a cheap and convenient way to compute reasonably tight bounding cones in very large data sets[4].

Agglomerative clustering was originally chosen for Lightcuts because it outperformed divisive clustering in early tests, but we wanted to retest this. We implemented several divisive clustering strategies and the one that performed the best was also the simplest. Uniform divisive clustering that simply splits each node along the middle of its longest axis. It can be performed either in 3D based solely on spatial position or in 6D by including spatial positions and directions as points on unit sphere scaled by c, the diagonal length of the scenes bounding box divided by 16. The agglomerative clustering dissimilarity function uses this same scaling factor and is:

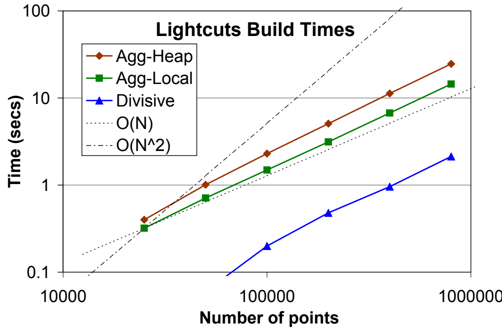
$$d(A,B) = I(A \cup B)\left(L(A \cup B)^2 + c^2 S(A \cup B)\right)^2 \tag{2}$$

where $I()$ is the summed intensity of a light cluster, $L()$ is the diagonal length of a cluster's spatial bounding box, $S()$ is sine of the half-angle of its directional bounding cone or one if the half-angle is greater than 90 degrees. This is a little different from the clustering function in [23] and results in somewhat higher quality trees.

5.1 Lightcuts results

Results for our four test scenes are shown in Figure 8 using the same machine as for the BVH results. All Lightcuts results used the same ray tracing acceleration structure, so the only difference is the method used to build the light tree. These scenes used the same parameters as in the original Lightcuts paper [23], except that we used 200000 indirect lights for all scenes and we used the simplified clamping, anti-aliasing (with 16 eye rays per pixel), and multiple representatives techniques from [21]. Although both agglomerative

[4] The special cases when the bounding cone spans $>$ 180 degrees or completely contains the unit sphere can be easily handled with a little care.



Num Points	250K	50K	100K	200K	400K	800K
Agg-Global	0.4s	1.0s	2.3s	5.1s	11.3s	24.7s
Agg-Local	0.3s	0.7s	1.5s	3.1s	6.7s	14.5s
Divisive	0.03s	0.06s	0.2s	0.5s	1.0s	2.1s

Figure 9: Log-log plot of clustering build times versus number of points for the Kitchen model with Lightcuts. Agglomerative (heap-based and locally-ordered) and simple uniform divisive (only 6D is included because 3D build times are so similar) clustering build times are plotted. The graph includes $O(N)$ and $O(N^2)$ trend lines to demonstrate that agglomerative clustering build complexity scaling is much closer to linear than quadratic.

methods build the same tree, the representative lights are chosen randomly causing small differences in cutsizes and image times.

The locally-ordered agglomerative cluster building is again faster than the heap-based version and only constitutes a small fraction of the overall image times. The agglomerative clustering consistently builds higher quality trees and is faster overall than the divisive clustering even including build times, though divisive clustering comes close in two scenes, Tableau and Temple. One surprising result is the good performance of the simple 3D divisive clustering, which implies that one can build reasonably good trees even while ignoring the lights' orientations.

In building light trees, there is an inherent tradeoff between wanting to create spatially compact clusters to tightly bound distance (intensity falls off with distance squared) and wanting to create directionally compact clusters to tightly bound the cosine fall off term in the lighting. The 3D uniform-divisive strategy is very good at creating spatially compact clusters and this partially compensates for its other shortcomings. Based on these results, when implementing Lightcuts, one might start with just the simpler divisive clustering and then add agglomerative clustering as a later optimization.

Figure 9 illustrates the scaling of agglomerative clustering with the number of input points. In Lightcuts, the indirect lights are generated stochastically, allowing us to easily vary their number. This example is typical of our experience where the agglomerative clustering complexity scaling is clearly sub-quadratic, and nearly linear. Thus, though not as fast as the simplest divisive clustering, agglomerative clustering can be applied to even very large data sets.

6 Conclusions

We have presented two fast algorithms for agglomerative clustering including a novel locally-ordered one that relaxes the order in which the tree is built and is easier to parallelize. We have then evaluated agglomerative clustering in two sample applications: ray casting and Lightcuts both in terms of tree build times and tree quality. While not as fast as the simplest divisive clustering algorithms, we have shown that agglomerative clustering often builds higher quality trees and is fast enough to be used even on very large data sets. The increase in quality is highly scene and application specific, but sometimes can be significant. Thus fast agglomerative clustering is a technique that graphics practitioners should be aware of and have in their toolbox of useful algorithms.

Acknowledgements: Thanks to Jeremiah Fairbanks (Kitchen), Moreno Piccolotto, Yasemin Kologlu, Anne Briggs, Dana Getman (Grand Central), Veronica Sundstedt, Patrick Ledda, and the Bristol Graphics Group (Temple) for the models. This work is supported in part by NSF grants 0719966, 0702353, 0615240, 0541193, 0509307, 0509324, 0426787 and 0406380, as well as grants from IBM, Microsoft, and Intel Corporation. Milind Kulkarni is supported by a DOE HPCS Fellowship. Kavita Bala is supported in part by NSF Career Grant 0644175.

References

[1] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In A. S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[2] G. Barequet and G. Elber. Optimal bounding cones of vectors in three dimensions. *Information Processing Letters*, 93:83–89, January 2005.

[3] D. Beeferman and A. Berger. Agglomerative clustering of a search engine query log. In *Knowledge Discovery and Data Mining*, pages 407–416, 2000.

[4] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[5] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster Analysis and Display of Genome-Wide Expression Patterns. *Proc. of the National Academy of Science*, 95:14863–14868, Dec. 1998.

[6] B. Gärtner. Fast and robust smallest enclosing balls. In *ESA '99: European Symposium on Algorithms*, pages 325–338, 1999.

[7] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comp. Graphics & Applications*, May 1987.

[8] S. Guha, R. Rastogi, and K. Shim. CURE: an efficient clustering algorithm for large databases. In *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 73–84, 1998.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[10] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII: Architectural support for programming languages and operating systems*, pages 233–243. ACM, 2008.

[11] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, P. Carribault, P. Chew, and K. Bala. Scheduling strategies for optimistic parallel execution of irregular programs. *ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[12] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *ACM Conf on Programming Language Design and Implementation (PLDI)*, 2007.

[13] T. Larsson and T. Akenine-Möller. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Technical Report, Mälardalen University, February 2003.

[14] J. P. M. Massó and P. G. López. Automatic hybrid hierarchy creation: a cost-model based approach. *Comp. Graphics Forum*, Mar. 2003.

[15] M. Miksik. Implementing lightcuts. In *Central European Seminar on Computer Graphics for students*, 2007.

[16] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, 1995.

[17] J. Ritter. An efficient bounding sphere. In *Graphics Gems I*, pages 301–303, 723–725. Academic Press, 1990.

[18] I. Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Symposium on Interactive Ray Tracing*, 2007.

[19] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[20] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.

[21] B. Walter, A. Arbree, K. Bala, and D. P. Greenberg. Multidimensional lightcuts. *ACM Trans. on Graphics*, 25(3):1081–1088, July 2006.

[22] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Implementing lightcuts. In *ACM SIGGRAPH 2005 Sketches*, page 55, New York, NY, USA, 2005. ACM Press.

[23] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics*, 24(3):1098–1107, Aug. 2005.