Dynamic Spatial Partitioning for Real-Time Visibility Determination

Joshua Shagam
New Mexico State University
Department of Computer Science

Contents

List of Figures

Chapter 1

Introduction

It is nearly axiomatic that users of interactive 3D systems, be they virtual environments, data visualizations, or user interfaces, have rising expectations with regards to the speed, malleability, and interactivity of the system. However, without infinite processing power, any significantly-complex interactive system must have some form of visibility determination which can discard large amounts of objects which have no chance at being visible. The existing algorithms for visibility determination within an interactive environment impose a number of constraints; for the vast majority of visibility algorithms, the data must either remain largely static or must be constrained to a precomputed set of possible locations [3]. Although this is acceptable for a large class of interactive systems, many applications, such as massively multi-user virtual spaces and high-performance data visualization, require the ability to modify the environment's data in any way at any moment, as there is simply no way to predict every possible visibility change.

Many commonly-used offline visibility algorithms precompute per-region potentially-visible sets from a static spatial partitioning [12, 15, 4]. A few algorithms take a more relaxed approach, performing the visibility determination on the fly but requiring a static spatial partitioning[10]. Other algorithms perform well without the need for spatial partitioning and provide visibility processing on the fly, but require significant amounts of dedicated processing power, predetermined sets of simplified potential occluders, and advance knowledge of motion constraints [14, 9]. Finally, a handful of algorithms come very close to the fully-dynamic ideal, but still require some offline processing and prescient knowledge of which objects will be present and how they may move [11].

In this paper, we take a simple spatial partitioning mechanism commonly used for collision detection and extend it to facilitate dynamic updates and to allow different heuristics for its partitioning strategy, describe four heuristics each with two variants, and compare their relative performance and efficiency. We also adapt a simple occlusion-culling algorithm from a static octree to this dynamic spatial partitioning, providing a dynamic visibility algorithm, which happens to perform adequately even without any imposed constraints, as a reference point for occlusion-culling research. We also show how to use the dynamic tree with a hierarchically-modeled data set and briefly explore collision detection.

Of course, like any object-level visibility algorithm, it can be used for any problem where an arbitrary region must be queried for object inclusion, such as object selection, motion planning, and various other volume-related tasks in computational geometry.

In the remainder of this paper, we describe a dynamic spatial partitioning data structure, show how to use it for a variety of tasks including visibility determination and occlusion-culling, describe several optimization heuristics, and then compare the heuristics' performance on a number of synthetic datasets.

Chapter 2

The dynamic AABB tree

Traditionally, an AABB tree is a simple tree structure where each node in the tree consists of an axis-aligned bounding box (AABB), and either two child nodes or a list of objects (usually individual polygons) [13]. Each internal node has the property that its AABB encompasses all of its objects and/or child nodes' AABBs, and the tree is built by successively splitting the top-level AABB in half along its longest axis and putting the objects into each child based on which side of the split it is on.

To facilitate dynamic operations and overall performance gains, it is useful to remove some of the constraints assumed in the traditional implementation of an AABB tree. For the dynamic AABB tree, we don't limit the number of children that a node can have, and we allow objects to be stored in internal nodes. Furthermore, rather than build the entire tree based on static data, we only build subtrees which are actively being queried. Finally, the tree as a whole has a nesting heuristic, which dictates how nodes are split and how objects are stored in the children. Four nesting heuristics will be described in detail in section 3.

2.1 Managing the tree

2.1.1 Splitting nodes

If a leaf node is queried and it contains more objects than a threshold value, the node is split. Splits do not happen at the time of object insertion, but at the time of query; this maintains better tree balance and improves overall efficiency, as node splits become "batched up" for later.

First, an algorithm chooses a splitting point. An ideal algorithm would be such that an equal number of objects will be on opposite sides of the point, such as by finding the median object coordinate in all dimensions. However, in our current implementation, we simply take the mean of every object's center.

Next, every object is given to the nesting heuristic, which selects the appropriate child node to put the object into based on the splitting point and the current node's previous bounding box. Some heuristics may decide to keep the object within the current node; as this does not violate the relaxed constraints of a dynamic AABB tree, this is acceptable.

Finally, if every object goes into the same child node (which is possible in certain pathological situations for some heuristics, *e.g.* all objects are the same size and at the same position, or are all within a certain distance of a boundary case), it means that there were enough of them to cause a split, and the child node has the exact same characteristics that the current node did; as a result, the nesting heuristic would end up putting every object into the same child node again, causing an infinite loop. In this case, we simply transfer the objects back to the current node, but keep the current node marked as split to prevent it from being split again.

This process is demonstrated with four different heuristics in figure 2.1, with the heuristics themselves explained in section 3.

Complexity analysis The cost of splitting a single leaf node is difficult to calculate directly, but it can be amortized by assuming the tree is fully split once and never changes significantly.

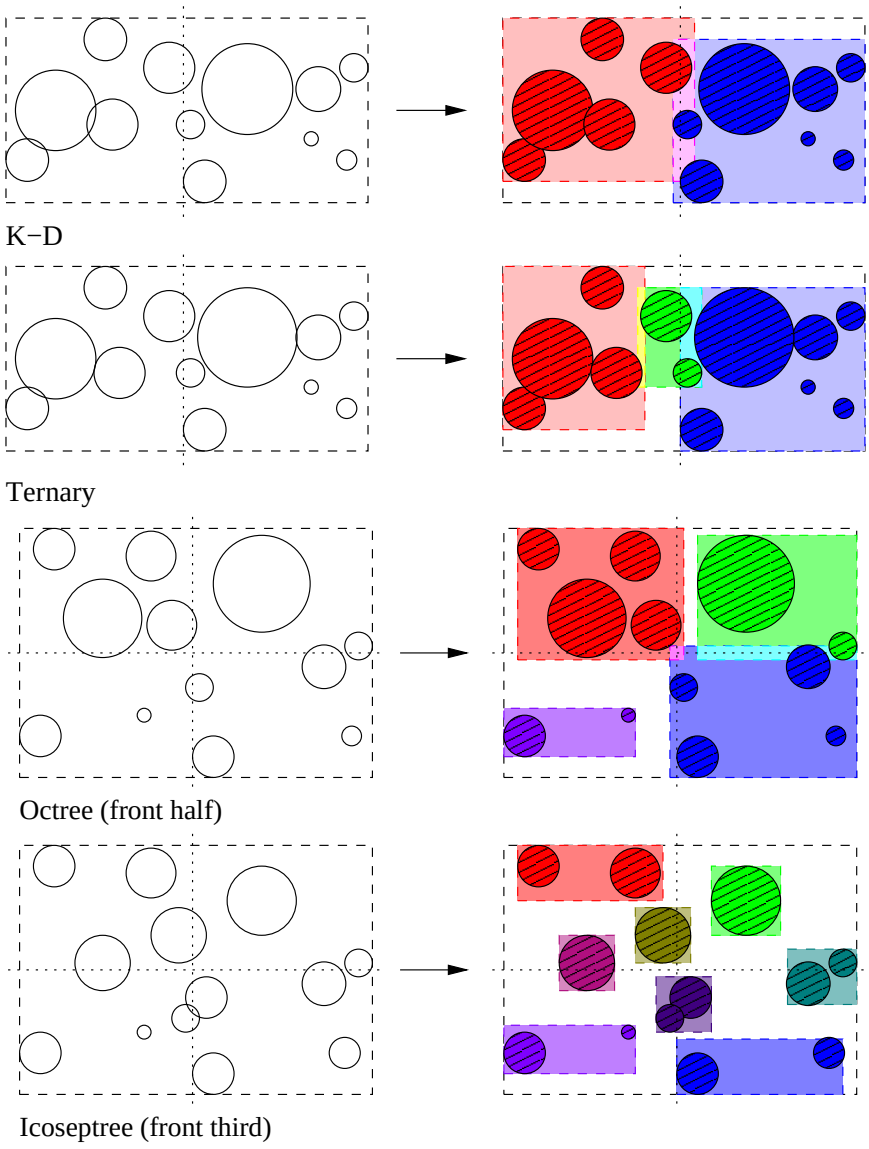

Figure 2.1: Four nesting heuristics

If there are n objects in the tree, then the time cost of splitting the entire tree is K cost of processing each individual object, plus the cost of splitting the child nodes. Assuming the nesting heuristic evenly distributes the objects across C children, then the total cost is $T(n) = Kn + CT\left(\frac{n}{C}\right) = O(n \log n)$; however, in practice, the total processing time to split an entire subtree scales essentially linearly with respect to the number of objects, and the cost of each individual object is very nearly constant. This will be demonstrated in section 4.

2.1.2 Inserting objects

Inserting an object into the tree is simple. Start at the root node, and using the nesting heuristic, select the appropriate child for the object until either we have reached a leaf node or the nesting heuristic determines that the object should stay at an internal node. Associate the final tree node with the object, then expand the AABBs of all nodes up to and including the root as necessary.

Complexity analysis This process takes average-case $O(\log n)$ time for the traversal and AABB expansion (worst-case $O(n)$ for an extremely unbalanced tree), and $O(1)$ time for the insertion into the tree node itself (assuming that the node's object list is maintained using a dynamic hash of pointers, such as a C++ `<hash_set>`, though in practice, an $O(\log n)$ binary tree, such as a C++ `<set>`, performs better).

2.1.3 Deleting objects

Deleting an object is simple; first, remove the object from the object list of the node which contains it. Next, starting with the current node, check to see if the object's bounding volume was flush against the edge of the AABB (meaning the AABB could potentially shrink as a result of this deletion); if so, mark the AABB for recalculation, traverse to the parent, and repeat this process until the object is no longer flush against the AABB. Finally, perform a simple garbage collection; if the node now contains no objects of its own and has no children, delete the node, and notify the parent of the child's deletion. If a parent node now has no children, then it should be unsplit, and if it has neither children nor objects of its own, it should be deleted. This process should continue all the way to the root of the tree.

Complexity analysis This process takes $O(1)$ time for the deletion and, in the worst case, $O(n)$ time for the garbage collection and AABB invalidation (for an extremely unbalanced tree). However, in general, only one or two nodes will ever be garbage-collected or have their AABB invalidated at any given time, making the average-case complexity essentially $O(1)$.

2.1.4 Updating objects

An object must be updated within the tree whenever its bounding volume changes with respect to the node it is stored in, such as when the object moves, rotates, or changes its geometry. The update process is simple. First, traverse up the tree until we find a node whose AABB will completely contain the object's bounding volume (stopping at the root of the tree). Next, use the nesting heuristic to determine the child node which should now contain the object. Finally, if this destination node is different than the object's current node, remove the object from the current node and insert it into the destination node, and if not, simply adjust the current node's AABB accordingly.

This technique exploits both spatial and temporal coherence; if an object does not move between visibility queries, then no processing is required, and if an object does move, only the space which it has moved within must be searched for its new location.

Complexity analysis As the search for the smallest subtree takes at most $O(\log n)$ time, the search for the new location place takes $O(\log n)$ time, and the individual deletion, insertion, and AABB update steps take $O(1)$ time, the entire process takes worst-case $O(\log n)$. However, given a good nesting heuristic and updates over short distances, the searches take only $O(1)$ time on average.

2.1.5 AABB determination

Determining the AABB of a node is simple. For each object stored within the node, grow the AABB to accommodate the object's bounding volume. Next, get each child node's AABB, and grow the current node's AABB to accommodate it. Finally, mark the AABB as current, and signal an update to the parent node, which may then require an update. This process initially takes an average of $O(1)$ time per node (amortized; $O(n)$ total time for $O(n)$ nodes), and subsequently only must be performed when a contained object was deleted if the object's bounding volume was flush against the AABB, which will take at most $O(\log n)$, assuming a change in bounding volume must be propagated all the way to the root of the tree.

2.1.6 Fragmentation avoidance

In many situations, it is quite possible for a number of objects to be added in such a way that it will "fragment" the tree, causing them to be inserted at much deeper levels of the tree than they would if the tree were built containing them to begin with, and also causes the tree as a whole to become imbalanced. In order to prevent this, the property that nodes within an AABB tree may overlap becomes very useful.

Whenever the extents of the AABB are recomputed after an invalidation, the split point is recomputed by taking the weighted mean of all of the objects in the current node and the split points of the child nodes. The objects are given a weight of 1, and the child split points are given a weight of the split threshold; with the assumption that the subtrees have remained defragmented, this new split point will be reasonably close to the split point which would be computed if the tree were generated anew. Because the leaf nodes of the tree are, by definition, balanced, by the inductive hypothesis, the tree as a whole will stay more or less balanced, assuming that any imbalance-causing objects stay in motion. Because imbalances are typically caused by moving objects, this assumption seems reasonable.

As a result, if an object is within a subtree which would now be bigger because of the new split point and it moves, it will remain within that subtree, growing that subtree's AABB towards the split point. Conversely, if an object is within a subtree which would now be smaller because of the new split point and it moves, either the move will cause the AABB to shrink anyway, or it will move outside of the AABB and be transferred to the subtree which is growing. The end result of this is that as objects move, the tree will eventually shift towards a balanced state. The rebalancing occurs quickest in the parts of the tree which are the most active. The three balance-shifting scenarios and an example of it in operation are shown in figure 2.2.

2.2 Using the tree

Except for the view volume clipping algorithms, the following algorithms are stated as examples on how to take advantage of the dynamic AABB tree to easily solve certain problems in computational geometry, and are not intended as the sole implementation for doing such. The dynamic AABB tree is a mechanism and a platform for future work, not an implicit solution to every problem in computational geometry.

2.2.1 View volume clipping

The process of clipping an entire data set against a convex volume of space, such as a potentially-visible volume of space, is simple; for any given node in the tree, test its AABB against the volume, for which there are many fast algorithms such as [1]; in our implementation, we first test a sphere circumscribed around the AABB against the visibility query, and if that succeeds, test the AABB itself using an algorithm similar to the inclusion test of Cohen-Sutherland [5]. If the AABB is totally visible, then insert all of the node's objects and all of the objects of all of its subtrees into the visible set. Otherwise, if the AABB is partially-visible, then compare its objects against the visibility query and recursively clip its subtrees. Finally, if an AABB isn't visible at all, simply discard it. Examples of a visibility query are shown in figures 2.3 and 2.4.

Complexity analysis The cost of clipping the tree is a function of the number of nodes which are visited and the number of objects stored in those nodes. Traversing an input set of n nodes produces two sets of objects, a candidate set of size w which must be further culled to the clipping region, and a visible set v which is guaranteed to be within the clipping region; the candidate set and visible set together form the working

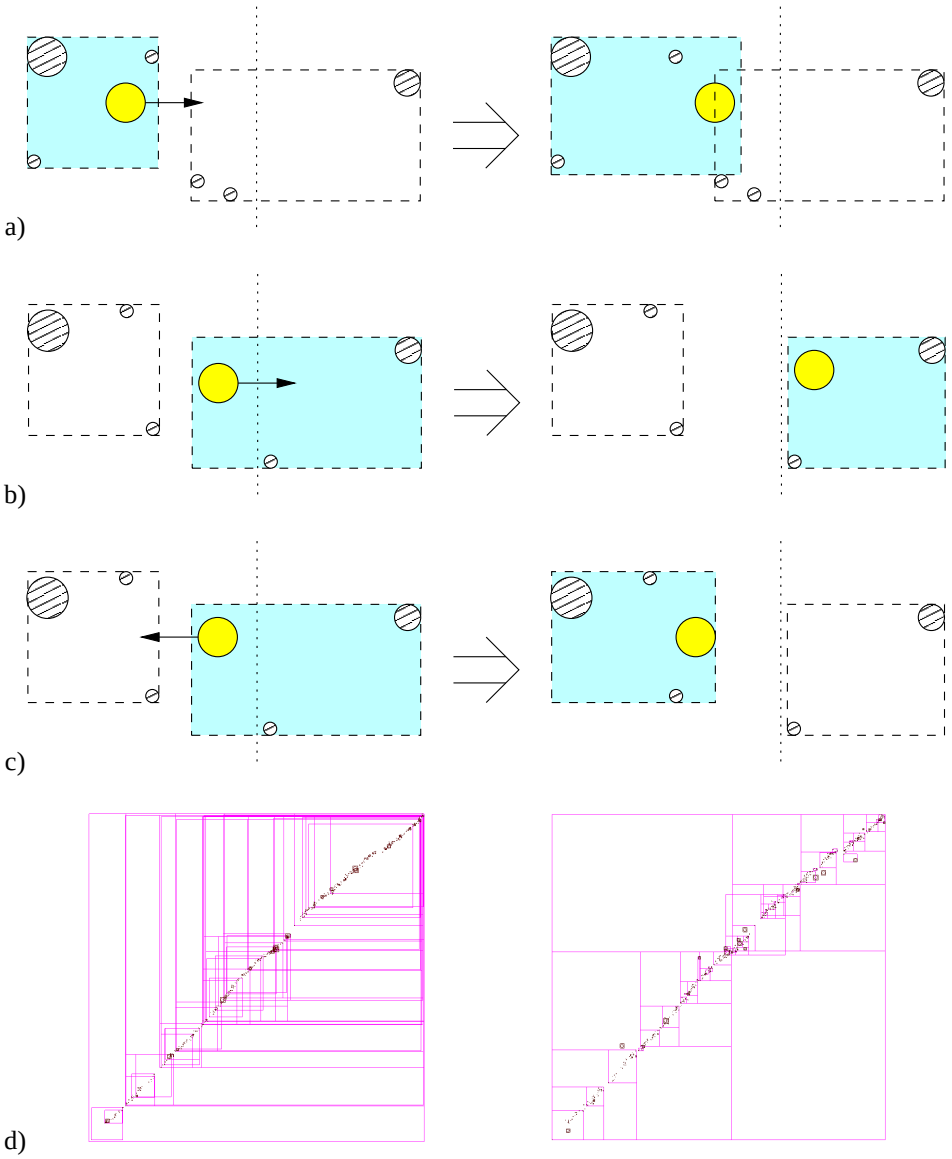

Figure 2.2: Moved split point facilitates rebalancing; a) moving object grows smaller subtree, b) moving object shrinks larger subtree, c) moving object moves into smaller subtree, d) a massively-imbalanced tree before and after 100 random movement cycles

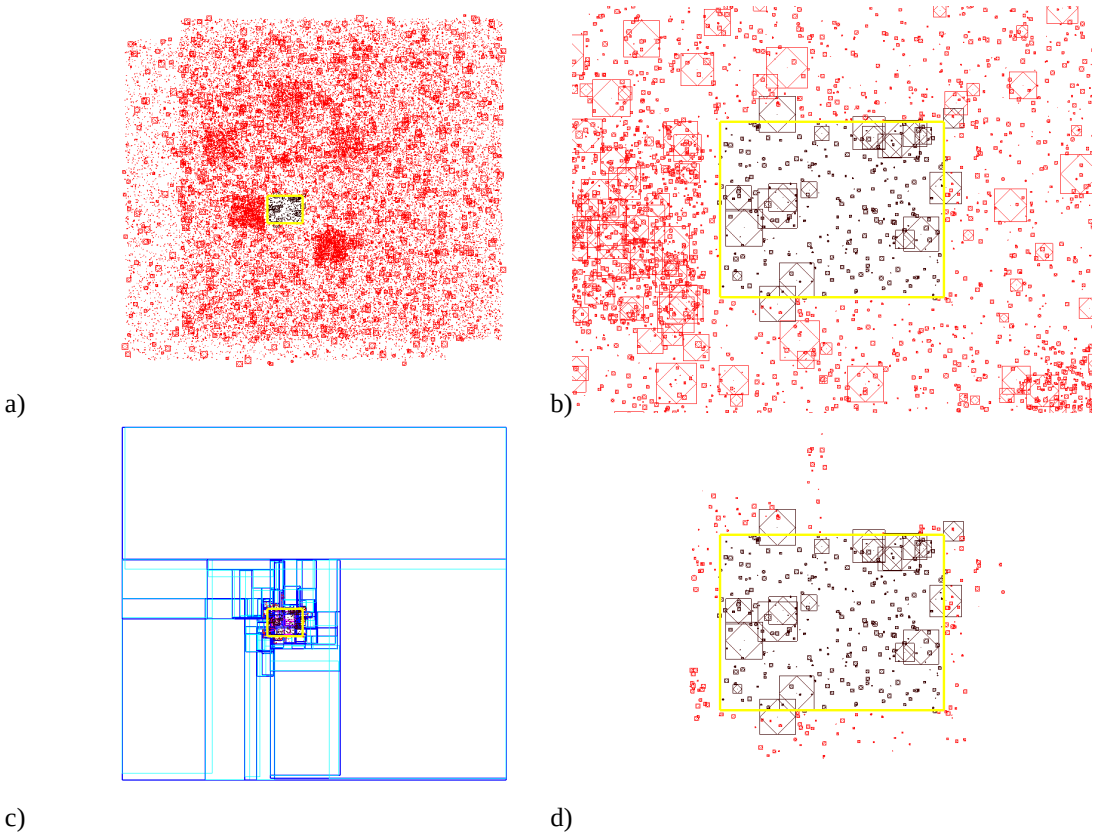

Figure 2.3: Visibility culling a two-dimensional dynamic AABB tree with the K-D heuristic; a) full input set (c. 10^6 objects), b) input set near query, c) queried node boundaries, d) working set and final query result

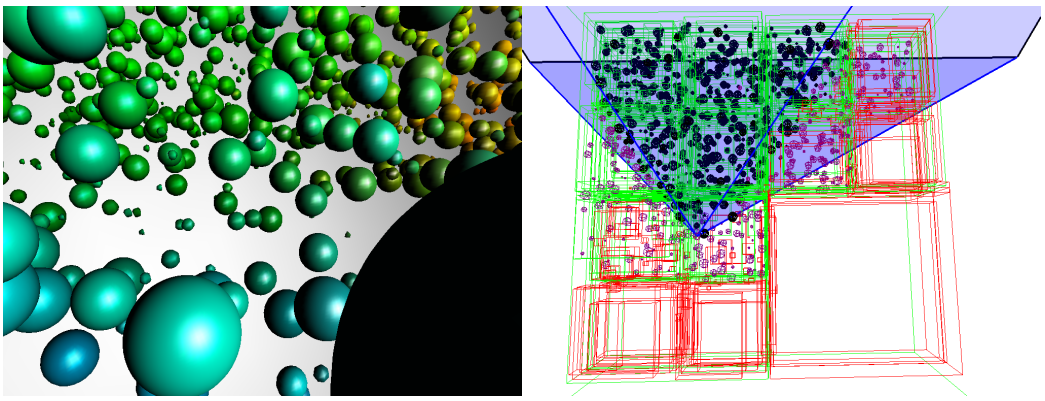

Figure 2.4: Visibility culling in 3D with the icoseptree heuristic; 539 objects visible of an 851-object working set, and a 1331-object input set

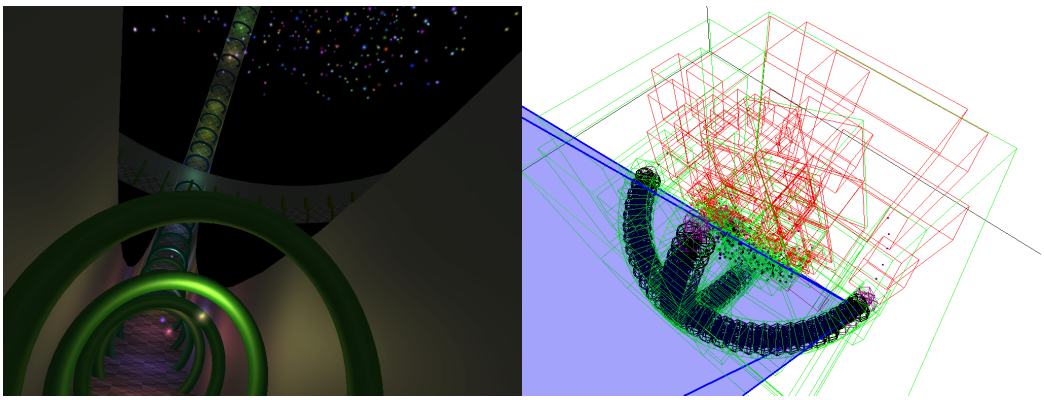

Figure 2.5: A hierarchically-modeled scene; 423 objects total are visible from a 680-object working set clipped from multiple nested AABB trees (icoseptree heuristic)

set. The final visible set is of size $\Omega(v)$ and $O(v+w)$, and takes $O(v+w)$ time to compute, as inserting a known-visible object takes $O(1)$ time, as does testing and inserting a candidate object. Although w is $O(n)$ (for the worst-case scenario of a heuristic which simply keeps all objects in a single node), it depends on the heuristic, the input set distribution, and how the query happens to align with the node boundaries.

2.2.2 Hierarchical geometry clipping

The data structure can still work with traditional hierarchically-modeled data. The simplest way to do this is to use the bounding geometry of the entire hierarchical object group when inserting it into the tree. In the case of bounding spheres, naïvely calculating the radius of the overall bounding sphere $R(N)$ is fairly trivial, by recursively finding

$$\begin{aligned} R(N) &= \max\{N_r, R_C(C_1), R_C(C_2), \ldots, R_C(C_n)\} \\ R_C(C) &= \|C_p\| + C_r \end{aligned}$$

where N is an object with bounding radius N_r and children $C_1 \ldots C_n$; each child's contributed bounding radius $R_c(C)$ is calculated from its radius C_r and its position C_p within its parent's coordinate space. These values can be stored with the individual objects once calculated, and if a child object moves, the change in overall bounding radius can simply be propagated upwards through the scene hierarchy in a manner similar to the AABB updates described in section 2.1.5.

Additionally, the hierarchical scene structure can itself be maintained as a set of nested AABB trees; our current implementation stores a visibility tree per object alongside a traditional hierarchical scene model, using the traditional hierarchy for object-management queries and the visibility tree for visibility queries. When the nested trees are queried for visibility, the AABBs are simply translated by the parent object's local coordinate space like any other object. Because a leaf tree node will only be split if there are a sufficient number of objects, simple hierarchical geometry does not significantly increase the overhead over simply storing the hierarchy normally, but for complex scenes, the nested visibility trees are clipped to the viewing volume, providing a very fine-grained level of hierarchical visibility determination without incurring any significant overhead.

Figure 2.5 shows an example of a hierarchical visibility query. There are five major groups of objects, each with its own AABB tree. Each of the three tracks is made of 72 segments, each of which is an object which further contains four parts (two walls, a floor, and a ring). The cloud of shiny points in the center is another hierarchical group of 1500 randomly-moving billboard-style sprites, and all of the moving light sources are in a fifth group. These top-level objects are all descended directly from the root scene object.

As each track segment is modeled hierarchically, they each have their own AABB tree; though they are unsplit, the root AABBs, which are rotated along with the segment grouping object, are still visible in the

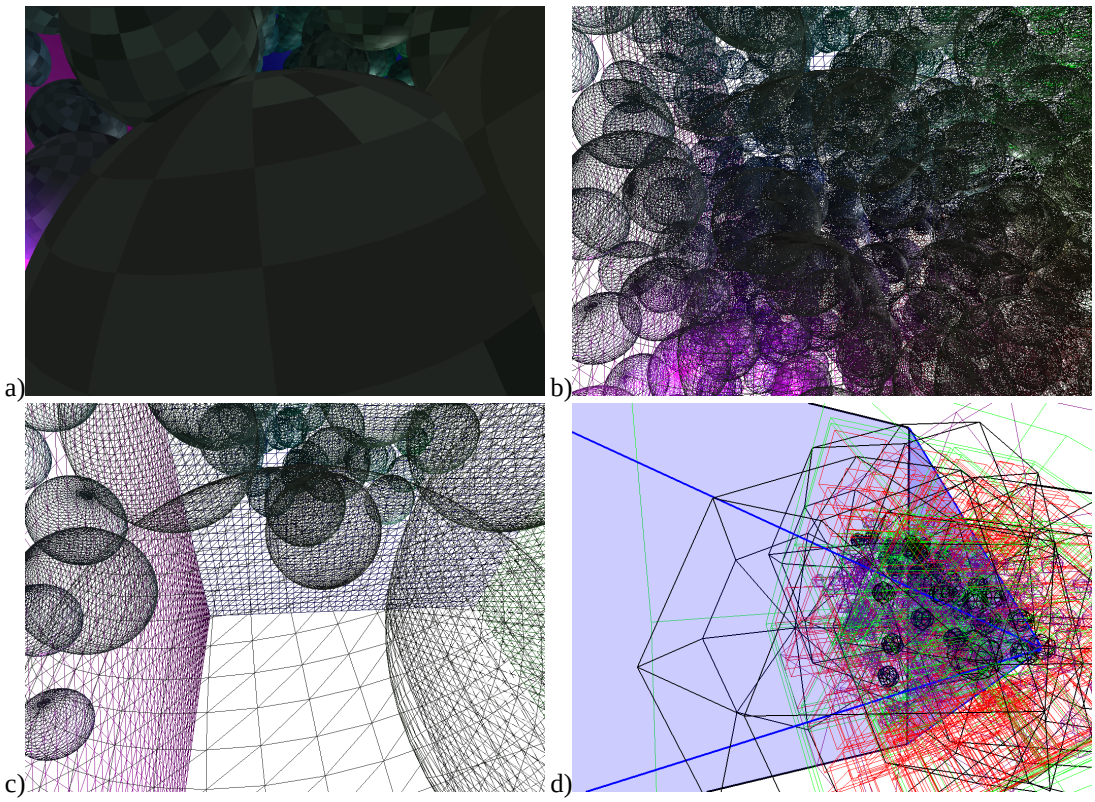

Figure 2.6: Simple occlusion culling; a) the rendered view; b) wireframe view without culling; c) with culling; d) visibility query (icoseptree heuristic)

query. Additionally, the entire group of shiny points has been rotated a bit to emphasize that its AABB tree, too, is in a different coordinate system than those of the tracks.

2.2.3 Occlusion culling

Adapting occlusion-culling algorithms which can use bounding volumes for group rejection is straightforward. In our current implementation we use a technique similar to [6], where we first draw objects which were visible in the previous frame at a reduced level of detail into a low-resolution software depth buffer while rendering them (or use the hardware depth buffer after rendering those objects normally), reduce the depth buffer in the style of a Z-pyramid [7], and then test the AABBs of the tree nodes against the depth buffer while obtaining the working set. The bounding volumes of the objects are also tested against the occlusion buffer before they're rendered, culling out even more geometry.

Figure 2.6 shows this algorithm in operation, with a scene of 1331 randomly-moving objects. Without the culling algorithm, there are 622 objects rendered of a working set of 804, whereas with the culling algorithm, there are only 45 objects rendered out of a working set of 285; the remaining 519 objects are culled out based on using the previous frame's visible objects as occluders.

Unfortunately, there are many circumstances which cause this algorithm to perform sub-optimally. For example, if an object occludes a significant portion of the tree, then a sudden change of visibility with respect to that object will cause many more objects to be rendered than are actually visible, and it will take a few frames for the correct visible set to emerge once again. In extreme cases, this scenario may cause the entire set of objects to be rendered even if only one is actually visible.

However, even in light of this simple occlusion-culling algorithm's weaknesses, it shows quite a bit of promise for future occlusion-culling algorithms which could, for example, determine and prioritize additional potential occluders by traversing the tree from the camera's position, only using the temporally-coherent

visibility information in order to provide an initial set of occluders.

2.2.4 Collision detection

Using the dynamic AABB tree to accelerate collision-detection queries is simple and intuitive. The objects are stored in a collision AABB tree (which should be kept separate from the visibility AABB tree), with their bounding volumes augmented to contain their motion over the next time increment. To perform collision queries against an object, a region containing the augmented bounding volume is queried; the resulting visible set contains only objects which require a detailed collision test as required by the application.

Although this isn't fundamentally different from any other octree- or AABB-tree-based collision-detection algorithm such as [2] or [13], this has the advantages that the query region doesn't have to be grown by a known global maximum velocity, and the internal book-keeping is significantly simpler because an entire object will only be stored in a single tree node. Its major disadvantage is that it requires quite a bit of tree updating and querying, with one update and query for each moving object per test cycle, though if it is used as a predictive collision detection, one test cycle doesn't necessarily correspond to one frame. Depending on the problem domain and necessary level of accuracy, this simple collision-detection algorithm may be sufficient.

Chapter 3

Nesting heuristics

In order to maximize performance, it is useful to compare a number of different nesting heuristics. We have implemented four heuristics, each with two variants, one where, like a traditional AABB tree, objects are stored only within leaf nodes if possible ("leafy"), and one where an object will be stored in an internal node if its bounding volume is at least 1/8 the volume of the internal node's AABB, so as to avoid creating a lot of extra child nodes simply to accommodate a particularly large object. Each heuristic is named after the algorithm it is patterned after; for example, the K-D tree heuristic is patterned after a K-D tree, but is not, strictly-speaking, a K-D tree, due to the relaxed properties of the dynamic AABB tree and also because the split dimension doesn't follow a strict depth-based pattern.

In order to test the nesting heuristics, a benchmark program creates a set of objects with n spheres of radius randomly varying from 1 to 10 inclusive, placed with a uniform random distribution within a cube, $3n^{1/3}$units on a side. It queries the entire tree (forcing it to fully split), and records the elapsed time. It then repeats the following process 1000 times, timing the actual CPU usage of each step using the UNIX `times()` system call: query a random $100 \times 100 \times 100$ cubical region of the tree, then randomly select 1000 objects to move in a random direction, with a distance proportional to its size. All times were taken on a 1133MHz Athlon with 512MB of PC133 SDRAM. The results of running the benchmark on a few fixed values of n while varying the splitting threshold are shown in figure 3.1.

Representative images of each distribution will be shown alongside the timing results in section 4.

3.1 K-D

The K-D heuristic, which is the splitting strategy employed by traditional AABB trees, is conceptually analogous to a K-D tree. The AABB is split across its longest axis into two regions through the calculated split point, each corresponding to one subtree. The center of the object determines which subtree it goes into.

By varying the splitting threshold with a few fixed numbers of objects, we find that the optimum splitting threshold is around 35 for the leafy variant, and 30 for the non-leafy variant.

3.2 Ternary

The ternary heuristic is similar to the K-D heuristic in that it splits the AABB across the longest axis into two regions, but there are three subtrees; objects which are entirely within one of the two regions goes into the corresponding subtree, and objects whose bounding volume extends into both regions – that is, objects which straddle across the two regions – go into the third.

By varying the splitting threshold, we find that the optimum splitting threshold is around 25 for the leafy variant, and 20 for the non-leafy variant.

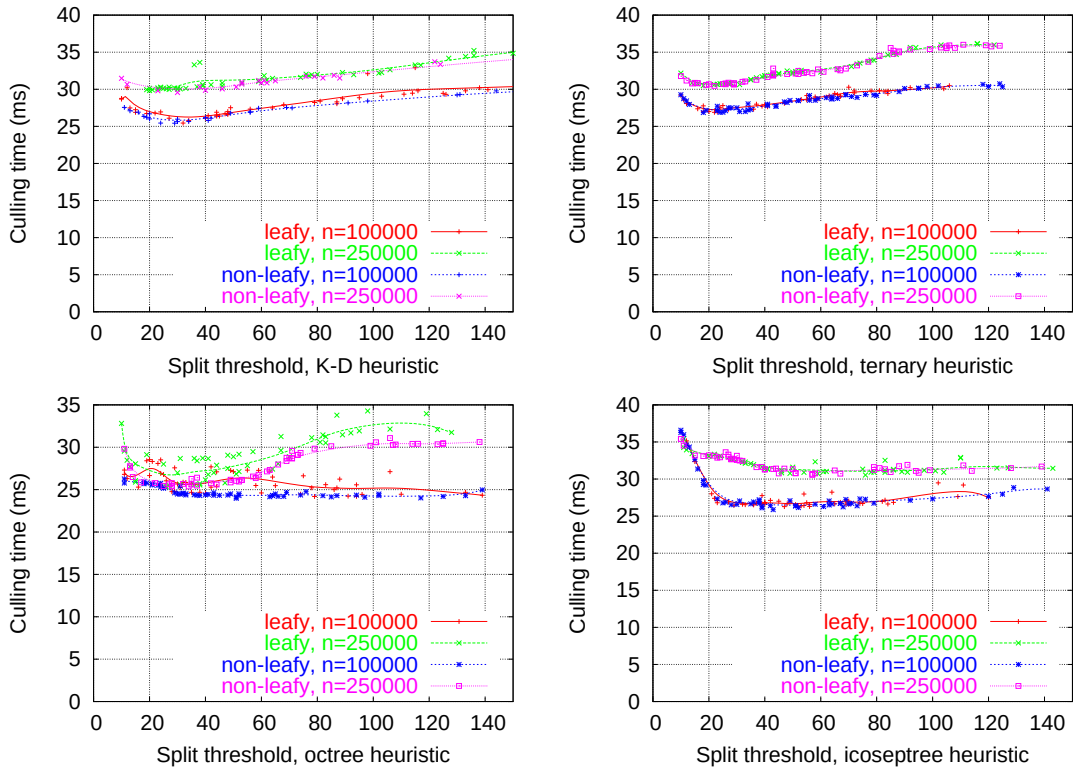

Figure 3.1: Finding the optimum split values for the heuristics

3.3 Octree

The octree heuristic is similar to an irregular octree; three axis-aligned planes through the split point partition the node into 8 regions, each with a corresponding subtree. An object is inserted into the subtree corresponding with the region its center is in.

By varying the splitting threshold, we find that the optimum splitting threshold is around 40 for the leafy variant, and 30 for the non-leafy variant.

3.4 Icoseptree

The icoseptree heuristic is the octree equivalent of the ternary tree, above. Each axis is divided into three regions, two of which are for objects entirely on one side of the split, and the third for objects whose bounding volumes intersect the split for that axis. The three regions on the three dimensions combine to provide 27 regions, each one representing a child node for objects to be inserted into.

This particular nesting heuristic merits exploration because of an interesting property; if the objects all lie close to a single axial plane or even along a straight line, an octree will not provide a very good partitioning, as pairs or quartets of subtrees will have bounding regions which mostly overlap. However, with the icoseptree, these objects will go into the intermediate subtrees, and the corner subtrees will be constrained to objects which are further away from the splitting planes, and in many cases this will cause the corner subtrees' AABBs to be significantly smaller than they would with the octree heuristic. In essence, it theoretically provides a mixture of octree, quadtree, and K-D tree behavior as necessary.

The optimum split threshold for the icoseptree heuristic is much more nebulous than with the other heuristics, as there is a large plateau of performance. As such, we select values which provide plenty of wiggle room, around 70 for the leafy variant, and 50 for the non-leafy variant.

Chapter 4

Timing comparisons

In order to do a full comparison of the various nesting heuristics, it is not sufficient to simply use a uniform distribution. Although a uniform distribution does a good job of finding the worst-case performance in order to find the best overall split threshold for each heuristic, most applications which require the use of online visibility determination have decidedly non-uniform distributions of objects. Therefore, in order to properly compare the nesting heuristics, it is important to compare them based on a number of different distributions which represent common scenarios in visualization and interactive graphics.

However, a comparison between all eight heuristics is unnecessary, as both the leafy and non-leafy variants show similar results. By varying the number of objects and comparing each heuristic types' leafy and non-leafy variants with each other, as in figure 4.1, we see that the non-leafy variant consistently edges out the leafy variant when both are at their optimum split threshold.

Additionally, the benchmark as used in section 3 is good for finding the optimum splitting threshold, but is inadequate for a decent real-world test, for a very simple reason: as given, the input data fills a region which only grows to the size of the query region when it reaches 10^5 objects. As a result, most queries for $n < 10^5$ end up simply inserting the entire tree, which tests nothing but the tree traversal up to that point. So, we reduce the queried region to $10 \times 10 \times 10$, and make the filled volume a cube $n^{1/3}$ on a side, so that when $n > 1000$, the query covers only part of the space.

For all distributions, we show and analyze graphs showing the query time, the time to randomly move 1000 random objects, the time taken to split the entire tree, the size of the working sets of each of the heuristics with the theoretical minimum size (i.e. the visible set), and the maximum tree depth. Although each distribution demonstrates different strengths and weaknesses for the heuristics, one universal result is that for all heuristics and distributions, the tree split time doesn't quite scale linearly, but is close enough for all practical purposes.

4.1 Uniform distribution

The uniform distribution (figure 4.2), as before, randomly fills the volume with objects in a uniform distribution. In terms of sheer performance, the icoseptree and octree heuristics clearly outperform the K-D and ternary heuristics, with the icoseptree heuristic overtaking the octree heuristic at around $n = 4.5 \times 10^5$. All the heuristics' query times appear to scale linearly with the size of the input set, but the visible set itself grows with the size of the input set in a fashion which indicates that if the output set were of a fixed size, the query time would scale more or less logarithmically. In any case, the output size is clearly the dominating factor.

Based on the maximum tree depth, we see that each of the heuristics' trees depths scale logarithmically, as expected.

4.2 Clustered distribution

In the clustered distribution (figure 4.3), five "seed" points are placed through the area at the same position every time, and are used for generating clusters of objects with a pseudo-Gaussian distribution.

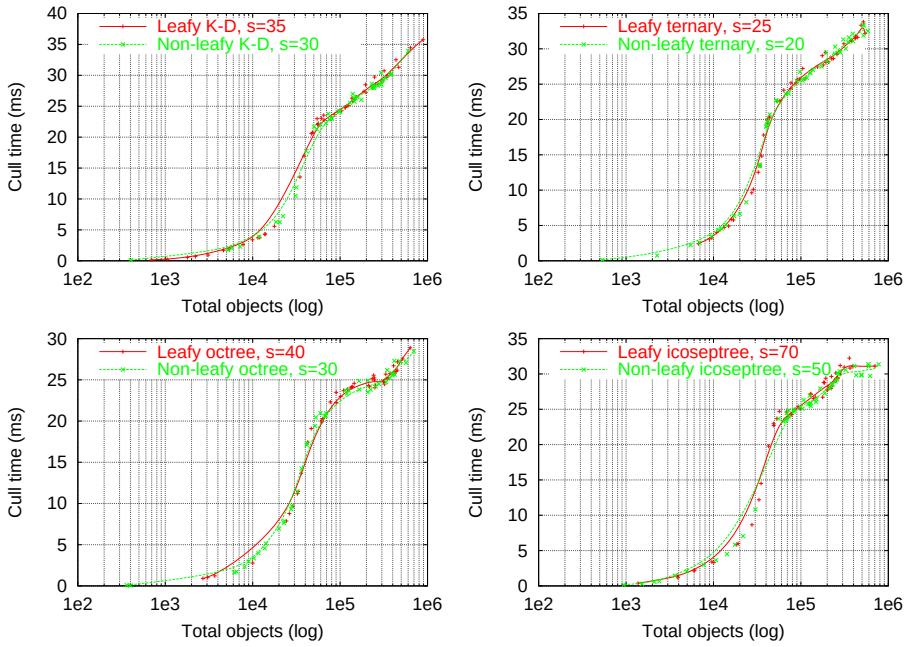

Figure 4.1: Comparing the leafy vs. non-leafy variants of the four heuristic types

This time around, the icoseptree clearly outperforms the other heuristics, and scales logarithmically with the input size, whereas a linear term begins to dominate on the other heuristics at around $n = 10^5$. By looking at the maximum tree depth, we see that the icoseptree and ternary heuristics scale logarithmically, depth-wise, while the K-D and octree heuristics scale more inconsistently. This clearly demonstrates that the icoseptree heuristic can adapt to inconsistent object distributions far better than an octree can.

It is also worth noting that as the number of objects increases, both the icoseptree and K-D heuristics' working sets approach the absolute minimum of the final visible set size.

4.3 Lissajous distribution

The Lissajous distribution (figure 4.4) places the objects in a Lissajous loop described by the equation $P(t) = \langle sin(c_1 t + k_1), sin(c_2 t + k_2), sin(c_3 t + k_3) \rangle$ with fixed values for $c_{1...3}$ and $k_{1...3}$, and t varies from 0 to 5000. This distribution is interesting because it appears like a straight line at a local level (which works well for a K-D tree but horribly for an octree), but like a cube at a global level (which works well for an octree but horribly for a K-D tree). Furthermore, most of the space within the region is empty of all objects.

From the timing graph, it is obvious that the icoseptree heuristic is the clear winner; the K-D heuristic likely fails because of the global cube-like distribution of objects, though it is interesting that the ternary heuristic performs about as well as the octree heuristic, even though the global distribution of objects should have caused the same problems as for the K-D heuristic. The octree fails to perform as well as the icoseptree due to the local distribution of objects, whereas the icoseptree simply falls back to K-D-like behavior in that case. It is also interesting to note that most of the heuristics' performance is dominated by a linear term, but for the icoseptree it is logarithmic overall, after accounting for the working set size which peaks around $n = 10^4$.

The maximum tree depth on each heuristic scales similarly to the clustered distribution, with a consistent logarithmic growth on the ternary and icoseptree heuristics and less consistency on K-D and octree. Additionally, the icoseptree and K-D heuristics' working sets again approach the minimum possible size.

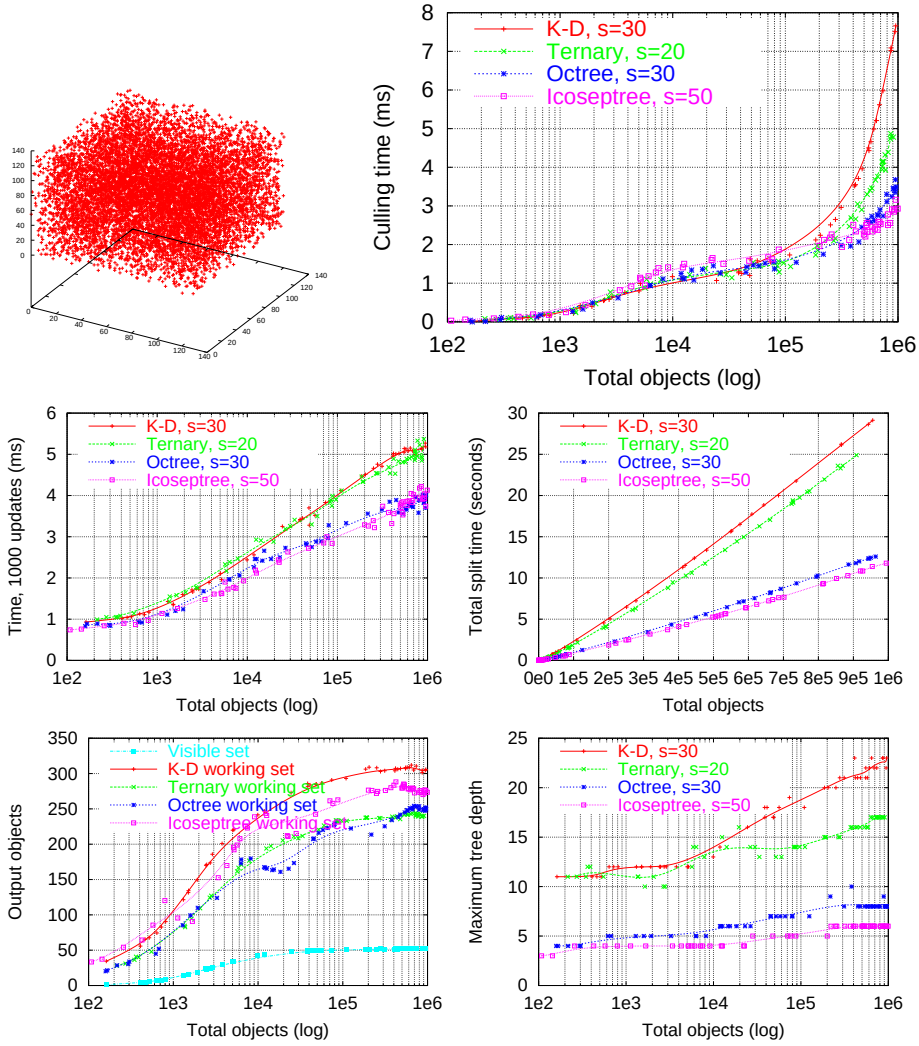

Figure 4.2: Performance on a uniform distribution

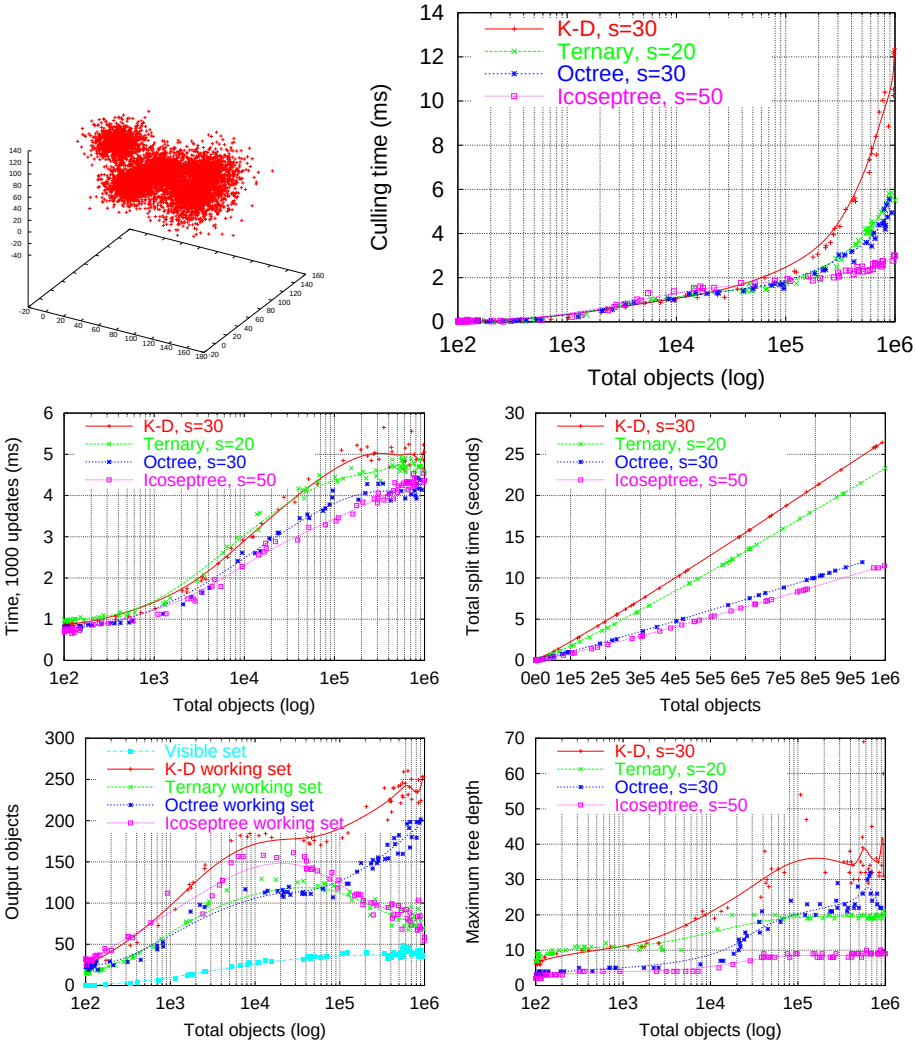

Figure 4.3: Performance on a clustered distribution

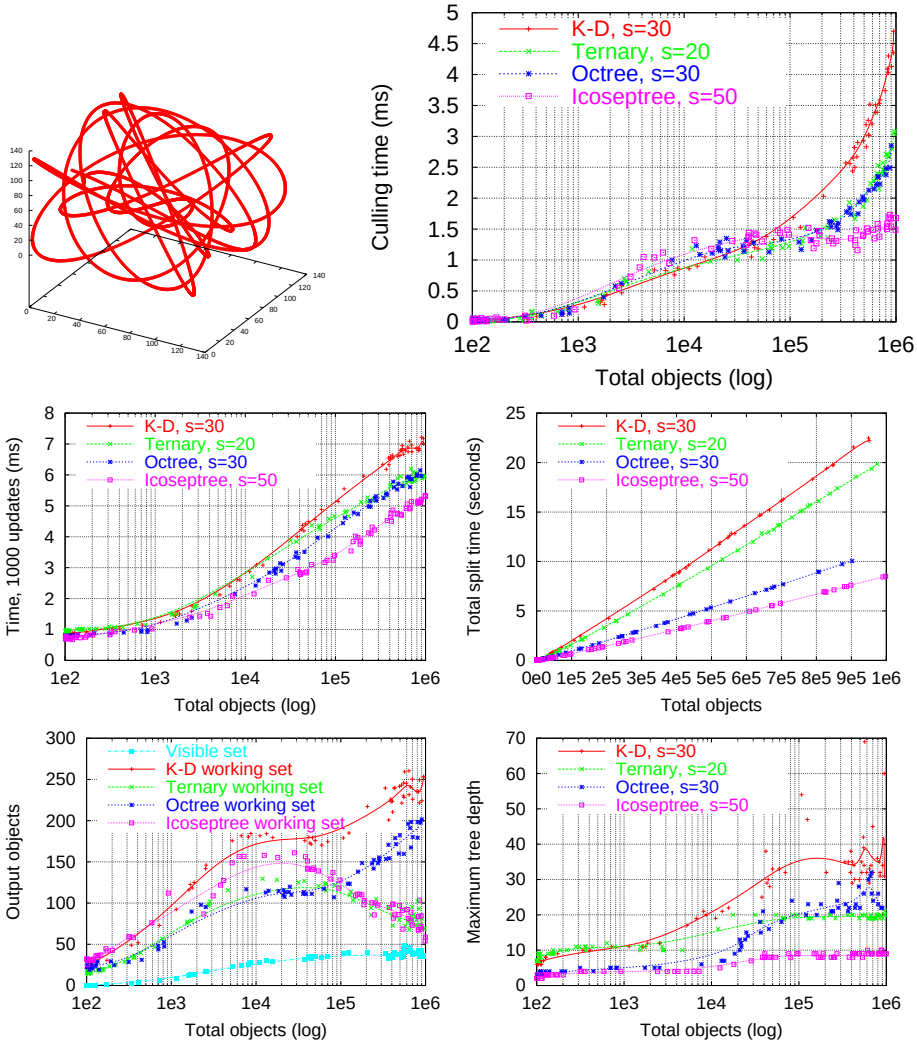

Figure 4.4: Performance on a Lissajous loop distribution

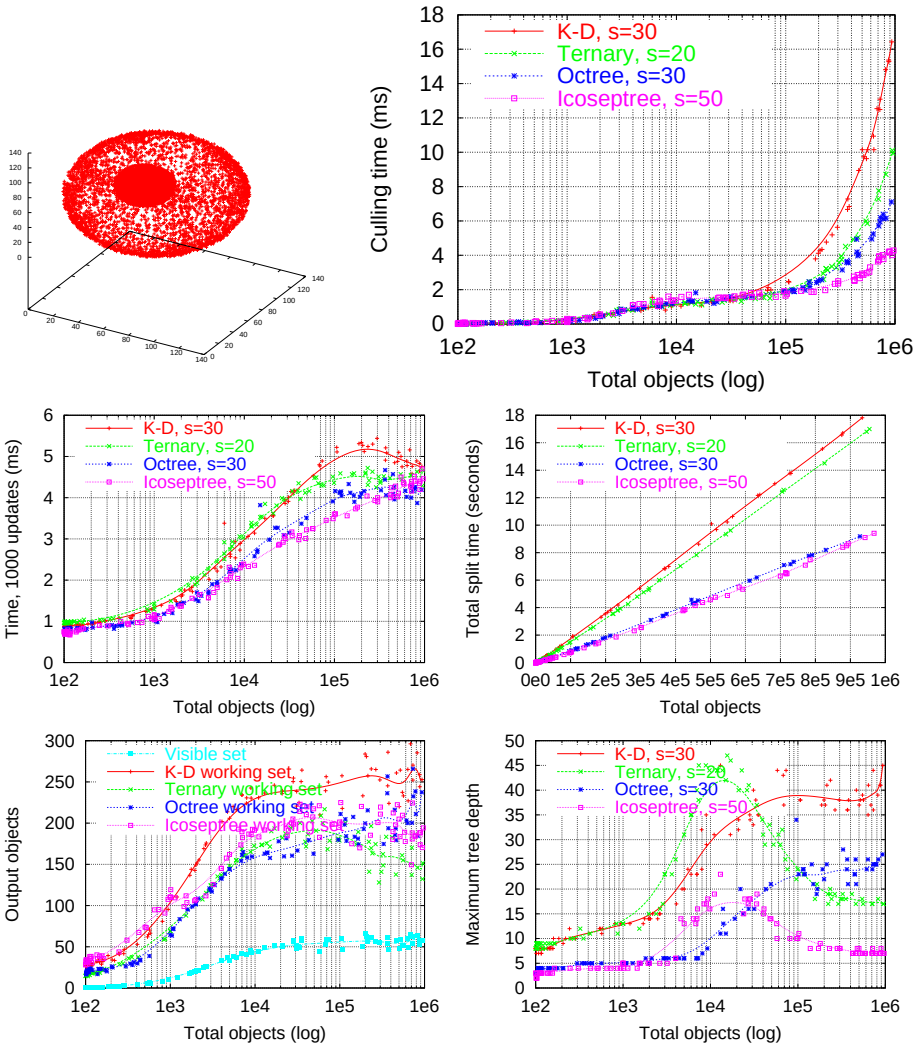

Figure 4.5: Performance on a multiple sphere distribution

4.4 Sphere

The spherical distribution (figure 4.5) places half of the objects near the surface of a hollow sphere, and the other half of the objects in a solid sphere of 1/3 the radius centered within the hollow sphere at a fixed location. The distribution throughout the solid sphere is uniform.

Once again, the best performance is had by the icoseptree heuristic. In this case, all of the heuristics have about the same performance which scales logarithmically until about $n = 10^5$, at which point a linear term begins to dominate for all heuristics except icoseptree, which doesn't see a linear term dominate until about $n = 8 \times 10^5$. However, at $n = 10^6$ the octree and icoseptree heuristics perform about the same, and the linear term for the icoseptree appears to be much more dominant than for the octree. For higher values of n, it is likely that the octree will perform better.

The maximum depth of the tree shows something quite interesting; as the number of objects increases after $n = 10^4$, the tree depth begins to decrease for the ternary and icoseptree heuristics. It is likely that at that point, the space-filling volume is large enough that fewer objects are in contact with the splitting plane, causing the split-plane branches to go unused. However, this does not explain why both of those heuristics have much shallower trees than their non-overlapping counterparts. A reasonable theory is that at the lower

levels of the tree, the split-plane branches are still beneficial. Regardless, aside from the hump, the depths again scale logarithmically for the ternary and icoseptree heuristics, while the K-D and octree heuristics experience much less consistent scaling after 3×10^3 and 10^4 objects, respectively.

4.5 Overall assessment

Based on the preceding information, it is fairly straightforward to determine that the best performance overall is with the icoseptree heuristic, and that the icoseptree heuristic is also likely to continue to scale exceptionally well for even greater orders of magnitude.

Chapter 5

Conclusions and future work

The modifications to the AABB tree presented here make it quite effective at efficiently determining visibility based on view-volume culling without imposing restraints on movement of objects. Furthermore, the presented occlusion-culling mechanism can be extremely effective at rejecting non-visible geometry from a complex 3D visualization, and is efficient enough for generalized real-time use. However, it remains far from optimal; a group-culling algorithm which takes advantage of the AABB tree to identify new occluders as well as existing ones should be considered.

Additionally, the most effective static nesting heuristic of those tested is the icoseptree, which exhibits a very good adaptive blend of K-D tree and octree behavior. However, it could still be improved as its working set is still quite a bit larger than the visible set, and in future work, the development of even more nesting heuristics, including the creation of a meta-heuristic which selects a nesting heuristic on a per-node basis, would be extremely beneficial.

Chapter 6

Acknowledgments

I would like to thank Dr. Joseph Pfeiffer, Jr. for being my advisor, Hue McCoy, Janet and Richard Shagam, and Bob DeKinder for their endless encouragement, and an anonymous SIGGRAPH review panel which offered the valuable insights and suggestions that ultimately led to this particular research. This work was made possible by National Science Foundation MII grants EIA-9810732 and EIA-0220590 and funded by the United States Department of Education GAANN program.

Appendix A

AABB tree implementation

The following code implements the dynamic AABB tree and the test code used to generate the two-dimensional figures and timing graphs for this paper. As presented it is not compilable, as it relies on the majority of the other object classes which comprise Solace[1], a complete high-performance graphics research platform, which represents many years of independent work and approximately 16,000 lines of code.

The object classes missing for the are, as follows:

- `Camera`: a structure describing a viewing origin (position, direction, field of view, etc.)
- `GeoImage`: a geometry image, the mesh representation used by the research platform (similar to [8])
- `HashMap`, `HashSet`: wrapper objects which (at compile-time) select a C++ `hash_map`/`hash_set` if available and desired, or an ordinary `map`/`set` if not
- `Matrix`: a 4x4 homogeneous transformation matrix
- `Node`: a node in the hierarchal scene geometry
- `Shader`: a component in a shader script (which programmatically describes how an object should be rendered)
- `vec3`: a 3-dimensional vector

A.1 Interface

```
/* BBTree.h
** Dynamic bounding-box trees
*/

#ifndef BBTREE_H
#define BBTREE_H

#include <vector>

#include "HashMap.h"
#include "vec3.h"

class Node;
```

[1] `http://www.cs.nmsu.edu/~joshagam/Solace/`

```
class BBTree {
public:
        BBTree(BBTree *parent = NULL);
        ~BBTree();

        // Get the ABB of the tree
        void GetExtents(vec3 &min, vec3 &max, bool split = true);

        // Add a node, returning the subtree it's in
        BBTree *AddNode(Node *);

        // Find the subtree a node is in
        BBTree *FindNode(Node *);
        BBTree *FindNode(vec3, float);

        // Remove a node from this tree (no search is performed)
        bool RemoveNode(Node *);
        bool RemoveNode(Node *, vec3 oldp, float oldr);

        // Update a node's location within the tree, taking advantage
        // of coherence
        bool UpdateNode(Node *, vec3 oldp, float oldr);

        // Add all of our nodes to a set
        typedef HashSet<Node *> Nodes;
        const Nodes &GetNodes() const {
                return m_nodes;
        }
        int GetNodes(std::vector<Node *> &) const;
        int NumNodes() const { return m_nodes.size(); }

        // Get the children (for external traversal)
        typedef std::vector<BBTree *> Children;
        const Children &GetChildren() const {
                return m_children;
        }
        bool IsLeaf() { return !m_issplit; }

        static void SetSplitVal(int);

private:
        BBTree *m_parent;

        bool m_issplit;
        enum {SplitX, SplitY, SplitZ} m_splitdir;
        vec3 m_split;
        vec3 m_min, m_max;
        bool m_extentsCorrect;

        int WhichChild(vec3, float) const;

        Nodes m_nodes;

        Children m_children;
```

```
        BBTree *GetChild(int c);

        void InsertNode(Node *);

        void TestSuicide();
        void RemoveChild(BBTree *c);

        void GrowExtents(const vec3& lo, const vec3& hi);
        void ShrinkExtents(const vec3& pos, float r);
        void UpdateExtents();
        bool Contains(const vec3& minp, const vec3& maxp);

        void DoSplit();
        void Unsplit();

        static unsigned int s_splitval;
};

#endif
```

A.2 Implementation

```
/* BBTree.cc
** Implementation of dynamic AABB trees
*/

#include <algorithm>
#include <iostream>

#include "BBTree.h"
#include "Node.h"

using std::cerr;
using std::endl;
using std::min;
using std::max;

//#define DEBUG
//#define DEBUG2
//#define DEBUG3

#define TREETYPE 27
#define NONLEAFY
#define DEFRAG

unsigned int BBTree::s_splitval = 50;

void BBTree::SetSplitVal(int s)
{
        s_splitval = s;
```

```
}

inline std::ostream& operator<<(std::ostream& lhs, const vec3 &rhs)
{
        lhs << '(' << rhs.x << ',' << rhs.y << ',' << rhs.z << ')';
        return lhs;
}

BBTree::BBTree(BBTree *p)
{
        m_parent = p;
        Unsplit();
        m_extentsCorrect = false;
}

BBTree::~BBTree()
{
        for (Children::iterator iter = m_children.begin();
             iter != m_children.end(); ++iter)
                if (*iter)
                        delete *iter;
}

void BBTree::GetExtents(vec3 &minp, vec3 &maxp, bool split)
{
        if (!m_extentsCorrect)
                UpdateExtents();

        if (split && !m_issplit && m_nodes.size() >= s_splitval)
                DoSplit();

        minp = m_min;
        maxp = m_max;
}

BBTree *BBTree::AddNode(Node *n)
{
#ifdef DEBUG3
        cerr << "BBTree::AddNode("
             << this << ", " << n << ")" << endl;
#endif

        BBTree *oo = FindNode(n);
        assert(oo);

        oo->InsertNode(n);

        vec3 pos = n->GetPos();
        float r = n->Radius(true);
        vec3 pr(r, r, r);
        oo->GrowExtents(pos - pr, pos + pr);

#ifdef DEBUG3
        cerr << "return = " << oo << endl;
```

```
#endif
        return oo;
}

BBTree *BBTree::FindNode(Node *n)
{
        if (n)
                return FindNode(n->GetPos(), n->Radius(true));
        return NULL;
}

BBTree *BBTree::FindNode(vec3 pos, float r)
{
#ifdef DEBUG3
        cerr << "BBTree::FindNode(" << this << ", "
             << pos << ", " << r << ")" << endl;
#endif

        BBTree *oo = this;
        int c = WhichChild(pos, r);
        while (c >= 0)
        {
                oo = oo->GetChild(c);
                c = oo->WhichChild(pos, r);
        }

        return oo;
}

bool BBTree::RemoveNode(Node *n)
{
        return RemoveNode(n, n->GetPos(), n->Radius(true));
}

bool BBTree::RemoveNode(Node *n, vec3 oldp, float oldr)
{
        ShrinkExtents(oldp, oldr);

        if (m_nodes.find(n) == m_nodes.end())
        {
                cerr << "Warning: octree " << this
                     << " tried to remove node "
                     << n << " which it doesn't contain" << endl;
                if (m_parent)
                {
                        cerr << "Parent says it should be in "
                             << m_parent->FindNode(n) << endl;
                        cerr << "Trying parent" << endl;
                        return m_parent->RemoveNode(n);
                }
                return false;
        }

        m_nodes.erase(n);
```

```
        // THIS MUST COME LAST
        TestSuicide();
        return true;
}

void BBTree::TestSuicide()
{
        if (m_parent)
        {
                bool killme = m_parent && m_nodes.empty();
                for (Children::iterator iter = m_children.begin();
                     killme && iter != m_children.end(); ++iter)
                        killme = (*iter == NULL);
                if (killme)
                        m_parent->RemoveChild(this);
        }
}

BBTree *BBTree::GetChild(int c)
{
        if (c < 0)
                return this;

        if (!m_children[c])
        {
                m_children[c] = new BBTree(this);
#ifdef DEBUG
                cerr << "BBTree " << this << " added child " << c << " = "
                     << m_children[c] << endl;
#endif
        }

        return m_children[c];
}

void BBTree::RemoveChild(BBTree *foo)
{
#ifdef DEBUG
        cerr << "BBTree " << this << "'s child " << foo
             << " suicided" << endl;
#endif
        bool unsplit = true;
        for (Children::iterator iter = m_children.begin();
             iter != m_children.end(); ++iter)
                if (*iter == foo)
                {
                        delete *iter;
                        *iter = NULL;
                } else if (*iter)
                        unsplit = false;

        if (unsplit)
        {
```

```
#ifdef DEBUG
                cerr << "BBTree " << this << "unsplitting" << endl;
#endif
                Unsplit();
        }

        TestSuicide();
}

void BBTree::UpdateExtents()
{
        bool set = false;

#ifdef DEFRAG
        vec3 newsplit(0,0,0);
        int ct = 0;
#endif

        for (Nodes::iterator iter = m_nodes.begin();
             iter != m_nodes.end(); ++iter)
        {
                vec3 pos = (*iter)->GetPos();
                float r = (*iter)->Radius(true);
                vec3 lo(pos.x - r, pos.y - r, pos.z - r);
                vec3 hi(pos.x + r, pos.y + r, pos.z + r);

                if (!set)
                {
                        m_min = lo;
                        m_max = hi;
                        set = true;
                } else
                {
                        m_min = vec3(min(m_min.x, lo.x),
                                     min(m_min.y, lo.y),
                                     min(m_min.z, lo.z));
                        m_max = vec3(max(m_max.x, hi.x),
                                     max(m_max.y, hi.y),
                                     max(m_max.z, hi.z));
                }

#ifdef DEFRAG
                newsplit = newsplit + pos;
                ct++;
#endif
        }

        for (Children::iterator iter = m_children.begin();
             iter != m_children.end(); ++iter)
        {
                vec3 lo, hi;
                if (*iter)
                {
                        (*iter)->GetExtents(lo, hi, false);
```

```
                        if (!set)
                        {
                                m_min = lo;
                                m_max = hi;
                                set = true;
                        } else
                        {
                                m_min = vec3(min(m_min.x, lo.x),
                                             min(m_min.y, lo.y),
                                             min(m_min.z, lo.z));
                                m_max = vec3(max(m_max.x, hi.x),
                                             max(m_max.y, hi.y),
                                             max(m_max.z, hi.z));
                        }

#ifdef DEFRAG
                        newsplit = newsplit + (*iter)->m_split*s_splitval;
                        ct += s_splitval;
#endif
                }
        }

#ifdef DEFRAG
        m_split = newsplit/ct;
#endif

        m_extentsCorrect = true;
}

void BBTree::Unsplit()
{
        m_issplit = false;
        for (Children::iterator iter = m_children.begin();
             iter != m_children.end(); ++iter)
                *iter = NULL;

}

void BBTree::DoSplit()
{
#ifdef DEBUG
        cerr << "BBTree " << this << " splitting ("
             << m_nodes.size() << " nodes)" << endl;
#endif

        assert(!m_issplit);

        // Find the split point
        m_split = vec3(0,0,0);
        float ttl = 0;
        for (Nodes::iterator iter = m_nodes.begin();
             iter != m_nodes.end(); ++iter)
        {
                vec3 pos = (*iter)->GetPos();
```

```
                float r = (*iter)->Radius(true);
#if 0
                // weigh by 1/r
                float rw = 1.0/(r + 1e-12);
#else
                const float rw = 1;
#endif
                m_split = m_split + (pos + vec3(r, r, r))*rw;
                ttl += rw;
        }
        m_split = m_split/ttl;

        m_issplit = true;
        m_children.resize(TREETYPE, (BBTree *)NULL);

#if (TREETYPE == 2) || (TREETYPE == 3)
        // Determine which way to actually split
        float sx = m_max.x - m_min.x,
                sy = m_max.y - m_min.y,
                sz = m_max.z - m_min.z;
        if (sx > sy && sx > sz)
                m_splitdir = SplitX;
        else if (sy > sx && sy > sz)
                m_splitdir = SplitY;
        else
                m_splitdir = SplitZ;
#endif

        bool safe = false, first = true;
        int lc = 0;
        Nodes here;
        for (Nodes::iterator iter = m_nodes.begin();
             iter != m_nodes.end(); ++iter)
        {
                int c = WhichChild((*iter)->GetPos(), (*iter)->Radius(true));

                // Make sure at least one object doesn't go into the
                // same child as everyone else
                if (!safe)
                {
                        if (c < 0)
                                safe = true;
                        else
                        {
                                if (first)
                                {
                                        lc = c;
                                        first = false;
                                } else
                                        safe = (c != lc);
                        }
                }

                if (c < 0)
```

```
                {
#ifdef DEBUG
                        cerr << "Retaining node " << *iter
                             << " (pos=" << (*iter)->GetPos()
                             << " r=" << (*iter)->Radius(true)
                             << ")" << endl;
#endif
                        here.insert(*iter);
                } else
                        GetChild(c)->InsertNode(*iter);
        }
        m_nodes = here;


#ifdef DEBUG
        cerr << "min=" << m_min << " max=" << m_max << endl;
        cerr << "Split at " << m_split << endl;
        cerr << "Retained " << m_nodes.size() << endl;
        for (unsigned int i = 0; i < m_children.size(); ++i)
                if (m_children[i])
                        cerr << "  child " << i << " got "
                             << m_children[i]->m_nodes.size() << endl;
#endif

        if (!safe)
        {
#ifdef DEBUG
                cerr << "*** All nodes went to " << lc << "!  Reclaiming..."
                     << endl;
#endif
                // Oops, all objects went into the same child node!
                // Take them back.
                BBTree *cn = GetChild(lc);
                Nodes::iterator iter = cn->m_nodes.begin();
                while (iter != cn->m_nodes.end())
                {
                        Node *n = *iter;
                        ++iter;
                        cn->RemoveNode(n);
                        InsertNode(n);
                }

                // The last removal marked this node as unsplit, so
                // we'll just have to do this all over again next time
                // this AABB is queried, unless we simply keep this
                // node split
                m_issplit = true;
        }

        for (Children::iterator iter = m_children.begin();
             iter != m_children.end(); ++iter)
        {
                if (*iter)
                        (*iter)->UpdateExtents();
```

```
        }
}

int BBTree::WhichChild(vec3 pos, float radius) const
{
        if (!m_issplit)
                return -1;

        int c = 0;

#ifdef DEBUG2
        cerr << "testing "
             << pos << ':' << radius << " against " << m_split << endl;
#endif

#ifdef NONLEAFY
        //if (m_extentsCorrect)
        {
                // If this object's bounding volume is at least 1/8
                // the last known tree bounding volume, just keep it
                // here
                float sv = radius*4*M_PI/3;
                float bv = (m_max.x - m_min.x)
                        *(m_max.y - m_min.y)
                        *(m_max.z - m_min.z);
                if (sv > bv/8)
                        return -1;
        }
#endif

#if TREETYPE == 2
        if (m_splitdir == SplitX)
                c = (pos.x < m_split.x)?0:1;
        else if (m_splitdir == SplitY)
                c = (pos.y < m_split.y)?0:1;
        else // if (m_splitdir == SplitZ)
                c = (pos.z < m_split.z)?0:1;
#elif TREETYPE == 3
        float p, s;
        if (m_splitdir == SplitX)
        {
                p = pos.x;
                s = m_split.x;
        } else if (m_splitdir == SplitY)
        {
                p = pos.y;
                s = m_split.y;
        } else
        {
                p = pos.z;
                s = m_split.z;
        }

        if (p + radius < s)
```

```
                c = 1;
        else if (p - radius > s)
                c = 2;
#elif TREETYPE == 8
        if (pos.x < m_split.x)
                c |= 1;
        if (pos.y < m_split.y)
                c |= 2;
        if (pos.z < m_split.z)
                c |= 4;
#elif TREETYPE == 27
        if (pos.x + radius < m_split.x)
                c += 1;
        else if (pos.x - radius > m_split.x)
                c += 2;

        if (pos.y + radius < m_split.y)
                c += 3;
        else if (pos.y - radius > m_split.y)
                c += 6;

        if (pos.z + radius < m_split.z)
                c += 9;
        else if (pos.z - radius > m_split.z)
                c += 18;
#else
# error "Unknown tree type " TREETYPE
#endif

#ifdef DEBUG2
        cerr << "got child " << c << endl;
#endif

        if ((unsigned int)c >= m_children.size())
        {
                cerr << this << "s == " << m_issplit
                     << ":" << m_split.x << ","
                     << m_split.y << ","
                     << m_split.z
                     << " c == " << c
                     << " sz == " << m_children.size() << "/"
                     << TREETYPE << endl;
                abort();
        }

        return c;
}

int BBTree::GetNodes(std::vector<Node *> &foo) const
{
        int ret = 0;
        for (Nodes::const_iterator iter = m_nodes.begin();
             iter != m_nodes.end(); ++iter)
        {
```

```
                ret++;
                foo.push_back(*iter);
        }

        return ret;
}

void BBTree::GrowExtents(const vec3& lo, const vec3& hi)
{
        BBTree *pp = this;
        while (pp)
        {
                if (pp->m_extentsCorrect)
                {
                        pp->m_min = vec3(min(pp->m_min.x, lo.x),
                                         min(pp->m_min.y, lo.y),
                                         min(pp->m_min.z, lo.z));
                        pp->m_max = vec3(max(pp->m_max.x, hi.x),
                                         max(pp->m_max.y, hi.y),
                                         max(pp->m_max.z, hi.z));
                }

                pp = pp->m_parent;
        }
}

void BBTree::ShrinkExtents(const vec3& c, float r)
{
        BBTree *pp = this;
        while (pp)
        {
                if (c.x - r <= pp->m_min.x
                    || c.y - r <= pp->m_min.y
                    || c.z - r <= pp->m_min.z
                    || c.x + r >= pp->m_max.x
                    || c.y + r >= pp->m_max.y
                    || c.z + r >= pp->m_max.z)
                        pp->m_extentsCorrect = false;
                pp = pp->m_parent;
        }

}

void BBTree::InsertNode(Node *n)
{
        m_nodes.insert(n);
        n->SetBBTreePos(this);
}

bool BBTree::Contains(const vec3& minp, const vec3& maxp)
{
        return (minp.x >= m_min.x
                && minp.y >= m_min.y
                && minp.z >= m_min.z
```

```
                && maxp.x <= m_max.x
                && maxp.y <= m_max.y
                && maxp.z <= m_max.z);
}

bool BBTree::UpdateNode(Node *n, vec3 oldp, float oldr)
{
#ifdef DEBUG3
        cerr << "update of " << n << " from " << this << endl;
#endif

        float r = n->Radius(true);
        vec3 rp(r, r, r);
        vec3 minp = n->GetPos() - rp;
        vec3 maxp = n->GetPos() + rp;

        BBTree *ptr = this;
#ifdef DEBUG3
        cerr << "starting at " << ptr << endl;
#endif
        while (ptr->m_parent
               && !ptr->Contains(minp, maxp))
        {
                ptr = ptr->m_parent;
#ifdef DEBUG3
                cerr << "traverse to " << ptr << endl;
#endif
        }
#ifdef DEBUG3
        cerr << "adding at   " << ptr << endl;
#endif

        // Find its final reinsertion point
        ptr = ptr->FindNode(n);

        if (ptr != this)
        {
                // Moved to a new node
                ptr->AddNode(n);
                if (!RemoveNode(n, oldp, oldr))
                        return false;
        } else
        {
                // Just update the extents of this node
                GrowExtents(minp, maxp);
                ShrinkExtents(oldp, oldr);
        }

        return true;
}
```

Bibliography

[1] ASSARSSON, U., AND MÖLLER, T. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools: JGT 5*, 1 (2000), 9–22.

[2] BANDI, S., AND THALMANN, D. An adaptive spatial subdivision of the object space for fast collision detection of animating rigid bodies. *Eurographics'95* (August 1995), 259–270.

[3] COHEN-OR, D., CHRYSANTHOU, Y., AND SILVA, C. A survey of visibility for walkthrough applications. In *EUROGRAPHICS'00 course notes* (2000).

[4] DURAND, F., DRETTAKIS, G., THOLLOT, J., AND PUECH, C. Conservative visibility preprocessing using extended projections. In *Siggraph 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 239–248.

[5] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHS, J. F. *Computer Graphics: Principles and Practice*, second ed. Addison-Wesley, 1990, ch. 3.12.3, pp. 113–117.

[6] GREENE, N. Visibility culling using graphics hardware. In *ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations Using Graphics Hardware*, D. Manocha, Ed., ACM SIGGRAPH 2002 Course Notes. ACM SIGGRAPH, 2002, pp. H1–H37.

[7] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-buffer visibility. *Computer Graphics 27*, Annual Conference Series (1993), 231–238.

[8] GU, X., GORTLER, S. J., AND HOPPE, H. Geometry images. *ACM SIGGRAPH 2002* (July 2002), 355–361.

[9] KOLTUN, V., CHRYSANTHOU, Y., AND COHEN-OR, D. Hardware-Accelerated from-Region visibility using a dual ray space. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering* (2001), pp. 205–216.

[10] SCHAUFLER, G., DORSEY, J., DECORET, X., AND SILLION, F. X. Conservative volumetric visibility with occluder fusion. In *ACM SIGGRAPH 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 229–238.

[11] SUDARSKY, O., AND GOTSMAN, C. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics 5*, 1 (1999), 13–29.

[12] TELLER, S. J., AND SÉQUIN, C. H. Visibility preprocessing for interactive walkthroughs. *Computer Graphics 25*, 4 (1991), 61–68.

[13] VAN DEN BERGEN, G. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools: JGT 2(4)* (1997), 1–14.

[14] WONKA, P., WIMMER, M., AND SILLION, F. X. Instant visibility. In *EG 2001 Proceedings*, A. Chalmers and T.-M. Rhyne, Eds., vol. 20(3). Blackwell Publishing, 2001, pp. 411–421.

[15] ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. E. Visibility culling using hierarchical occlusion maps. *Computer Graphics 31*, Annual Conference Series (1997), 77–88.