Dynamic Irregular Octrees

Joshua Shagam Joseph Pfeiffer, Jr.
New Mexico State University*

Abstract

As consumer graphics hardware becomes faster and more capable, it is becoming desirable for realtime interactive content to become more dynamic. However, the static spatial partitioning mechanisms used in current interactive systems, such as games and other virtual reality environments, are not suitable for a fully-dynamic interactive environment. In this paper, we take a new look at octrees, and create a dynamic octree-like data structure which allows for real-time modification of a scene while retaining many of the benefits of a static spatial partitioning. Additionally, we explore a mechanism for providing fully-dynamic occlusion culling using this dynamic spatial partitioning.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Virtual reality—Visible line/surface algorithms

Keywords: real-time, visibility, spatial partitioning, occlusion, dynamic

1 Introduction

As the performance of commodity hardware increases, users of interactive 3D systems have rising expectations as to the degree of interactivity provided at high frame rates. However, present systems only have a limited amount of interactivity within the environment itself; due to the static spatial partitioning and visibility mechanisms in common use (such as BSPs), the underlying geometry of an interactive environment must remain largely static, or must be constrained to a precomputed set of possible locations [Cohen-Or et al. 2000].

Most visibility algorithms provide an optimal solution, but require offline preprocessing time. Many commonly-used offline visibility algorithms are only slightly conservative (in the sense that not many non-visible objects are included), but assume that the CPU is the bottleneck during rendering and precompute per-region potentially-visible sets from a static spatial partitioning [Teller and Séquin 1991; Zhang et al. 1997; Durand et al. 2000]. Another class of algorithm removes these constraints by using a regular grid for the spatial partitioning [Batagelo 2001], but such a partitioning suffers from performance and/or scalability problems when used for extremely large environments or where there is a high variability in object density. Other algorithms perform well without the need for spatial partitioning and provide visibility processing on-the-fly, but require significant amounts of processing power, predetermined sets of simplified potential occluders, and advance knowledge of motion constraints [Wonka et al. 2001]. Finally, a handful of algorithms come very close to the fully-dynamic ideal, but still require some static partitioning and prescient knowledge of how objects may move [Sudarsky and Gotsman 1999].

Thanks to exponential increases in processing power, an optimal spatial partitioning is no longer necessary, as a fast algorithm which reduces the working set to something manageable for a secondary brute-force test is sufficient. Additionally, because the major performance bottleneck has shifted from the host CPU to the communications pipeline with the graphics processor, the CPU can spend more time on per-frame visibility determination without degrading the overall performance. As such, an algorithm which provides an extremely-conservative working set is acceptable as long as there isn't a significant penalty to maintaining the partitioning or rejecting the extraneous objects.

In this paper, we describe a fully-dynamic adaptation of one of the simplest and fastest spatial partitioning algorithms – the octree – and show that even though it is extremely conservative, it still boosts performance by an order of magnitude over a brute-force test. We then adapt a simple but effective occlusion-culling algorithm from a static octree to a dynamic structure, providing a fast, output-sensitive, and totally-dynamic visibility algorithm with no need for constraints of any sort. Finally, we show how to use the dynamic octree with a hierarchally-modeled scene, and examine the use of the dynamic octree for collision detection.

2 The Dynamic Octree

Traditionally, octrees have been used for ray tracing [Glassner 1984; Fujimoto et al. 1986], background object collision detection [Bandi and Thalmann 1995], motion planning [Hamada and Hori 1996], and other operations which are performed on static objects. Typically, the octree is used to form a regular partitioning of a bounded region of space, and due to the way in which the octree is queried, objects which reside in multiple leaf nodes of the octree are simply inserted into all of the pertinent leaf nodes. Although this works well for algorithms where the calculation terminates once it finds an object, it adds a level of complexity when performing operations on entire regions of space where the operation should only be performed once on an object. Traditional scanline rendering is a good example of such an algorithm.

Our approach takes a different perspective; rather than require that objects be stored in a leaf node, they are simply stored at the deepest level of the octree at which they fit entirely in a single node. Additionally, by adding a bounding volume to each node and removing the requirement that space be divided uniformly, the octree can cover an arbitrary region of space while keeping the actual intersection footprints to a minimum.

2.1 Data Structure

A single node in the dynamic octree contains a set of objects contained within the node, up to 3 axis-oriented splitting planes, up to 8 child octree nodes, and an axis-aligned bounding box (AABB) with a flag stating whether it is current.

Each object stored in the octree has a reference to the octree node it is contained in, and a bounding volume; in our implementation[1], we use a simple bounding sphere centered on the origin of the object's coordinate system for octree insertion (described in sections 2.2.1 and 2.2.3), and the intersection of the bounding sphere and an oriented bounding box (OBB) for occlusion culling (described in section 3.2).

*email:{joshagam,pfeiffer}@cs.nmsu.edu

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

2.2 Managing the Tree

2.2.1 Child Selection

To determine into which child node an object should go, the object's bounding volume is compared with the splitting plane(s). If the object straddles any of the planes, it is kept in the current node. Otherwise, build a bitmask, setting the nth bit based on whether the object is on the positive side of the nth plane, numbered arbitrarily starting from 0. This bitmask is the identifier of the appropriate child node, which is created as it is needed.

2.2.2 Splitting Nodes

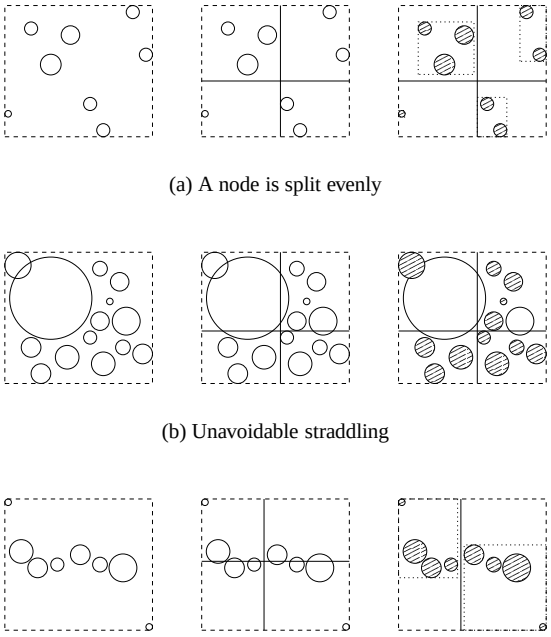
If a leaf node is queried and it contains more objects than a threshold value, the node is split. Our implementation currently sets this threshold at 16, with the assumption that the split will be perfect and the rationale that there should never be fewer than 2 objects in a node. Splits do not happen at the time of object insertion, but at the time of query; this maintains better tree balance and improves 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. However, in our current implementation, we simply take the mean of every object's center, weighted by the inverse of its radius; this heuristic is selected in order to bias the split towards clusters of small objects. Three axis-oriented splitting planes are created through this splitting point.

Next, all of the objects are tested against the splitting planes. If the number of objects straddling a splitting plane exceeds a threshold[2], that plane is disabled. If all of the splitting planes are disabled, the ones with the least number of straddling objects are enabled. This step avoids a very important problem; if all objects happen to be lying within a single axis-oriented plane or line (for example, resting on a table top, or people walking on a crowded city street), removing the planes with excessive straddling causes the octree to emulate a quad-tree or KD-tree for better performance.

Finally, after splitting the node, all objects which don't straddle a plane are inserted into the appropriate child node, as per the child selection algorithm.

This process is demonstrated in figure 1.



(a) A node is split evenly

(b) Unavoidable straddling

(c) Excessive straddling; a split plane is discarded

Figure 1: Splitting leaf nodes; objects in the current node are clear, objects in child nodes are shaded

Complexity Analysis The cost of splitting a single leaf node is difficult to calculate directly, but it can be amortized across the entire octree. If there are n objects in the octree, then the time cost of splitting the entire octree into unsplit leaf nodes is K cost of processing each individual object, plus the cost of splitting the child nodes. Assuming a perfectly-balanced split with no straddling objects, this is $T(n) = Kn + 8T\left(\frac{n}{8}\right) = O(n)$. As there are at most $\frac{8n-1}{7}$ total nodes in the octree[3], the amortized cost of splitting a single node is $O(1)$.

2.2.3 Inserting Objects

Inserting an object into the dynamic octree is nearly identical to a standard octree. Start at the root node, and select the appropriate child for the object until either we have reached a leaf node or the object straddles a splitting plane. Associate the final octree 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 octree node itself (assuming that the octree 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).

2.2.4 Deleting Objects

Deleting an object is simple; first, remove the object from the object list of the node which contains it. Next, if the object's bounding volume was flush against the edge of the AABB (i.e. the AABB could potentially shrink as a result of this deletion), mark the AABB for recalculation, as well as the AABB of any parent nodes where the object was similarly flush. 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] Our implementation currently sets this at $1/4$ of the number of objects in the node.

[3] Assuming n leaf nodes and $\lceil \log n - 1 \rceil$ levels of empty internal nodes in a perfectly-balanced tree; $n + \sum_{k=0}^{\lceil \log_8 n-1 \rceil} 8^k = n + \frac{8^{\lceil \log_8 n-1 \rceil}-1}{8-1} \leq n + \frac{n-1}{7}$

2.2.5 Updating Objects

An object must be updated within the octree whenever its bounding volume changes with respect to the node it is stored in, for example when the object moves, rotates, or changes its geometry. The update process is simple – remove the object from the octree, update it, then reinsert it, for an $O(\log n)$ average-case running time.

We do not attempt to avoid garbage collection by reinserting the object before removing it from its old node. If a node is removed, it is because the node only had one object in it. The lasting performance impact of a nearly-empty octree node is much greater than the one-time penalty of deleting (and possibly recreating) a node. Additionally, if the deleted leaf is the only child of its parent, the deletion of the leaf allows for the parent to be re-split with better balance anyway.

2.2.6 AABB determination

Determining the AABB of an octree 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 (recursively recalculating it if necessary), and grow the current node's AABB to accommodate it. Finally, mark the AABB as current. This process takes an amortized $O(1)$ time, and only must be performed when a node is queried for the first time or after a contained object was deleted if the object's bounding volume was flush against the AABB.

3 Using the Tree

3.1 View Volume Clipping

The process of clipping an entire scene against a viewing frustum or other visibility primitive is simple; for any given node in the octree, test its AABB, for which there are many fast algorithms [Assarsson and Möller 2000]. If the AABB is visible, add its contents to the working set, and then recursively apply this test to its child nodes. Example outputs of this test are shown in figures 2 and 3.

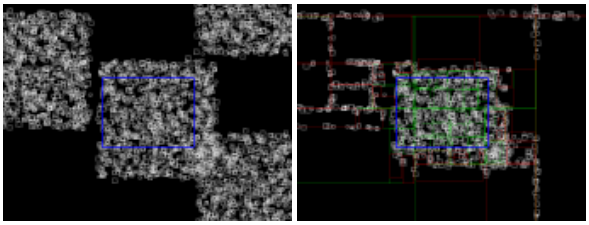
The cost of traversing the octree is a function of the number of octree nodes which are visited and the number of objects stored in those nodes. The number of octree nodes for a working set of w objects is $O(w)$, making the total query time $O(w)$. Ideally, w would be equal to v, the size of the visible set, but due to object straddling, w itself is $O(n)$, making the worst-case time cost for determining the working set $O(n)$, even if v is relatively small.

In practice, w/v scales sublinearly with n, as is demonstrated in section 5.

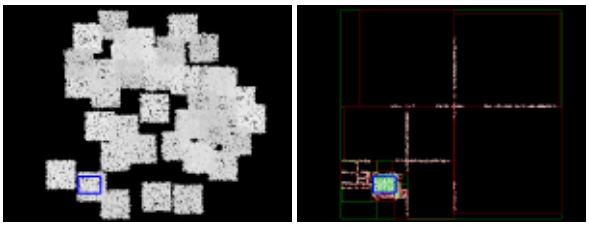
3.2 Occlusion Culling

Adapting algorithms which use octrees for occlusion culling is fairly straightforward. In our implementation we use a technique similar to [Greene 2002], 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's depth buffer after rendering those objects normally), reduce the depth buffer in the style of a Z-pyramid [Greene et al. 1993], and then test the AABBs of the octree 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.

For an example of this in action, see figure 4, which depicts a scene with 1331 randomly-moving 7000-polygon sphere objects, many of which are hidden behind a partition. With the occlusion buffer disabled, about 1000 of the spheres are rendered after the visibility test, but with the occlusion buffer enabled, only about 500

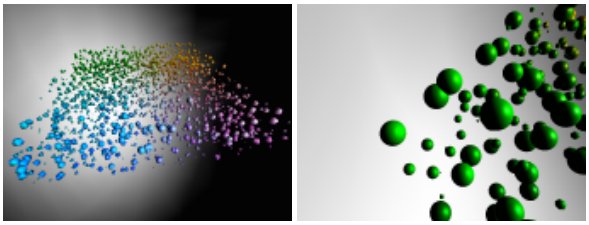


(a) Unculled (b) Culled to 1598 candidates

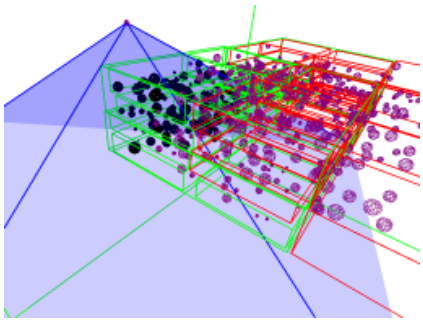


(c) Zoomed out, unculled (d) Zoomed out, culled

Figure 2: 41000 simple objects moving around in an octree; red = discarded node, green = included node, blue = query region, white = candidate object



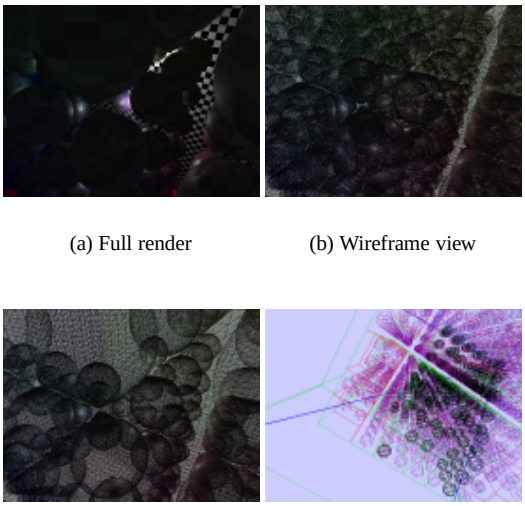
(a) Full scene (b) Close-up view



(c) Octree traversal performed to render (b); objects eventually culled from the working set are shown in magenta

Figure 3: A cloud of 1765 random spheres

of the spheres are even included in the working set, the rest being culled out at the octree level, and only the spheres which are visible are actually rendered. The octree query (including occlusion culling) only accounts for approximately 0.01% of the total render time, and even with the large number of culled objects, visibility and occlusion queries only account for 1.5% of the total CPU time. Almost all of the remaining CPU time is spent on rendering the objects which are actually visible, at approximately 10Hz at 640x480 (on a 1.1GHz Athlon equipped with a Radeon 7000; the primary bottleneck is the lack of vertex processing power). On a video card with hardware vertex processing and stored display lists, the framerate is significantly higher.



(a) Full render (b) Wireframe view

(c) Wireframe with hidden geometry culled (d) Octree query for (c)

Figure 4: Densely-occluded scene with over 1300 7000-polygon objects

3.3 Hierarchal Geometry

The dynamic octree can still work with traditional hierarchally-modeled scenes. The simplest way to do this is to use the bounding geometry of the entire hierarchal 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 $C_1 \ldots C_n$ (its contribution to the bounding radius $R_c(C)$ being 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 octree's AABB updates.

Additionally, the hierarchal scene structure can itself be maintained as a set of nested octrees; our current implementation stores an octree per object alongside a traditional hierarchal scene model, using the traditional hierarchy for object-management queries and the octree for visibility queries. When the nested octrees are queried for visibility, the AABBs are simply translated by the parent object's local coordinate space like any other object. Because a leaf octree node will only be split if there is a sufficient number of objects, simple hierarchal geometry does not significantly increase the overhead over simply storing the hierarchy normally, but for complex scenes, the hierarchal octrees are clipped to the viewing volume, providing a very fine-grained level of hierarchal visibility determination while also providing implicit spatial-coherence hinting to the top-level partitioning.

Figure 5 shows an example of a hierarchal octree query. In this scene, there is a large circular track in which each segment is modeled hierarchally. Additionally, the particles (rendered as billboarded sprites) are contained within a toplevel object, which has been rotated in order to emphasize the separation of its octree from the top-level octree.

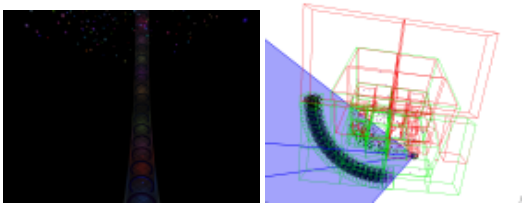


Figure 5: A scene modeled hierarchally, and its respective octree query showing the clipped object octrees

3.4 Collision Detection

Using the dynamic octree to accelerate collision-detection queries is simple and intuitive. The objects are stored in a collision octree (which should be kept separate from the visibility octree), 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 working set will contain all of the objects whose bounding volumes are going to collide with the object during the current time step. The working set is then culled to the augmented bounding volume, and the remaining objects are the only ones which require a detailed collision test as required by the application.

Although this isn't fundamentally different from any other octree-based collision-detection algorithm such as [Bandi and Thalmann 1995], 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 object will only be stored in a single octree node.

The disadvantage to this approach is that the time taken to find pairs of colliding objects scales linearly with the number of objects and the size of the working set; assuming an $O(\sqrt[4]{n})$ working set size (which will be explained in section 5), this implies an $O(n\sqrt[4]{n}) = O\left(n^{5/4}\right)$ complexity. However, in many interactive applications, it is acceptable to only perform detailed collision detections on visible objects, as objects outside of the viewer's field of view can often be performed with a larger time step and less accuracy.

4 Drawbacks

The most notable drawback is that although the tree is reasonably balanced to start with, certain access patterns can cause the octree

to become extremely unbalanced or to cause many objects to straddle a split plane. An extreme example is shown in figure 6. In this case, the octree was initially split with a large number of spatially-localized objects, after which many objects were added to straddle a split, preventing them from going further down in the octree structure. However, even with this extreme level of straddling, only about 1/4 of the objects are included as candidates in the octree query on average, still providing a reasonable speedup over naïvely testing every object. Efficiently rebalancing the tree would avoid this problem, though this is a topic for future research.

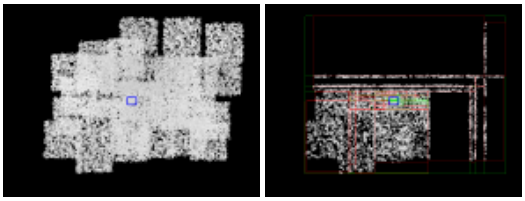


Figure 6: Excessively-imbalanced tree with many split-straddling objects

Also, the worst-case linear relationship between total objects and working set size tends to send a significant amount of geometry to the visibility culling stage; however, in practice, this does not provide a significant reason to discard the dynamic octree. Some may consider it wasteful that in figure 7, the octree query returns around 5,000 candidates when there are only around 500 within the queried region, but this is certainly an order of magnitude better than brute-force testing all 1 million objects; on a 1.1GHz Athlon, the octree query takes around 10ms.

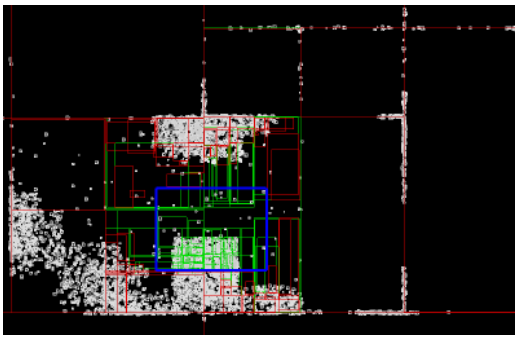


Figure 7: Octree containing 1 million objects (extreme closeup)

Additionally, splitting nodes only when they're traversed tends to cause extreme jumps in performance when the view enters a previously unviewed region. We can partially mitigate this by a number of means, such as splitting the children of traversed nodes which are rejected from the query, splitting all nodes within a reasonable region of anticipated motion, or by splitting the entire tree once we are sure that the boundaries aren't going to change significantly.

5 Results

In order to test the scalability of the dynamic octree, a benchmark program creates an octree with n spheres of radius 1, placed with a uniform random distribution within a cube, $3n^{1/3}$units on a side, averaging one sphere per 9 cubic units of volume. It queries the entire octree (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 $100 \times 100 \times 100$ region of the octree to produce a working set, cull the working set to the region, then randomly move 1000 random objects from anywhere in the octree.

Results of this benchmark (on a 1.1GHz Athlon with 512MB of RAM) are shown in figure 8. Object insertion (indicated by the shuffle time) performs with approximately $O(\log n)$ (as predicted in section 2.2). The working set size w clearly scales better than the worst-case $O(n)$; according to a least-squares regression, it . Both the query and cull times appear to be $O(w)$ (section 3.1), and by a simple least-squares estimation of an^b, w correlates well with $1.21n^{.236} = O(\sqrt[4]{n})$. We can improve w even further by reducing the number of split-straddling objects with a better node-splitting algorithm and a rebalancing mechanism.

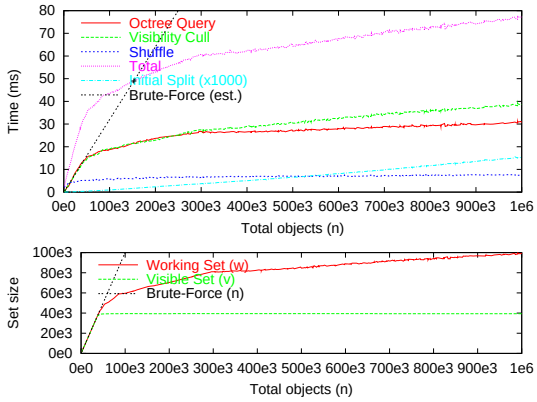


Figure 8: Algorithmic scalability; operation time (in milliseconds, except total tree split time shown in seconds), and output size

From this benchmark, the break-even point between a brute-force and a dynamic octree cull appears to be around 153,000 objects, which takes 49 milliseconds for both the brute-force and octree cull. However, due to the large query and relatively dense packing, and because the benchmark does not include the occlusion culling of section 3.2, this does not accurately reflect real-world performance; in normal operation, the tradeoff occurs closer to 2,000 objects for typical scenes.

The only distressing time is the initial splitting, which takes around 15.5 seconds with 1 million objects. However, because splits normally only happen on parts of the octree which are actually being queried (rather than all at once), and because those splits only happen once, those 15.5 seconds would normally be amortized across several frames, if they'd even happen at all. Additionally, we feel that 15.5 seconds isn't particularly troublesome, considering that bringing those 1 million objects into memory would take significantly more time.

On the subject of memory, when the benchmark was working with 1 million objects, the operating system reported a memory usage of around 330MB, implying that 330 bytes were used per object. This would be excessive if the benchmark were using a minimal object representation, but it used the same 3D object class from our full rendering system, which includes a per-object scene hierarchy with its own dynamic octree (as in 3.3), as well as other structures which have no bearing on this paper. In short, the objects themselves use far more memory than the octree.

6 Conclusions and Future Work

Although our algorithm does far from a thorough job of removing objects from the culling stage, we feel that it is still quite effective at reducing the number of objects which must be tested against the view volume. Furthermore, when coupled with an appropriate occlusion culling mechanism, it is extremely effective at rejecting non-visible geometry from a complex 3D scene to an extent previously only enjoyed by expensive off-line visibility preprocessing steps, and we feel that it is efficient enough for generalized real-time use.

There are many places where the algorithm could be greatly improved. For example, the current thresholds for node splitting and straddle avoidance are off-the-cuff guesses, and a thorough analysis should be made to find the optimum threshold values. Additionally, the current node split algorithm is extremely naïve, and alternates should be explored, as should an efficient means of rebalancing the tree.

Finally, it would be interesting to see if adapting other partitioning algorithms (such as binary space partitions) to the dynamic partitioning mechanism produces better results.

7 Acknowledgments

This work was funded by National Science Foundation MII grants EIA-9810732 and EIA-0220590 and the United States Department of Education GAANN program.

References

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

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

BATAGELO, H. C., 2001. 2D dynamic scene occlusion culling using a regular grid.

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

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

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. Arts: Accelerated ray tracing system. *IEEE Computer Graphics and Applications 6*, 4, 16–26.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10, 15–22.

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

GREENE, N. 2002. 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, H1–H37.

HAMADA, K., AND HORI, Y. 1996. Octree-based approach to real-time collision-free path planning for robot manipulator. *ACM96-MIE*, 705–710.

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

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

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

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