Volume xxx, (1999) number yyy pp. 000–000

Partitioning and Handling Massive Models for Interactive Collision Detection

A. Wilson and E. Larsen and D. Manocha and M. C. Lin

Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175
http://www.cs.unc.edu/~awilson/mmc/

Abstract

We describe an approach for interactive collision detection and proximity computations on massive models composed of millions of geometric primitives. We address issues related to interactive data access and processing in a large geometric database, which may not fit into main memory of typical desktop workstations or computers. We present a new algorithm using overlap graphs for localizing the "regions of interest" within a massive model, thereby reducing runtime memory requirements. The overlap graph is computed off-line, pre-processed using graph partitioning algorithms, and modified on the fly as needed. At run time, we traverse localized sub-graphs to check the corresponding geometry for proximity and pre-fetch geometry and auxiliary data structures. To perform interactive proximity queries, we use bounding-volume hierarchies and take advantage of spatial and temporal coherence. Based on the proposed algorithms, we have developed a system called IMMPACT and used it for interaction with a CAD model of a power plant consisting of over 15 *million triangles. We are able to perform a number of proximity queries in real-time on such a model. In terms of model complexity and application to large models, we have improved the performance of interactive collision detection and proximity computation algorithms by* an order of magnitude.

1. Introduction

The current technology for virtual and immersive environments offers us great potential for use in industrial concept design and evaluation. It can provide a design space consisting of three-dimensional computer generated images and the users can interact with them using intuitive interfaces in real-time. Such technology is increasingly being used for simulation-based design and multi-disciplinary reviews of large CAD models composed of millions of primitives (e.g. submarines, airplanes, power plants, etc.). A *key* component of such environments is the ability to directly perceive and manipulate virtual objects *interactively*, rather than simply viewing a passive environment. This may involve interactions like grabbing an object and moving it around the virtual environment using natural and physical motion. For many CAD applications, the designers would also like to test for accessibility of parts and feasibility of the entire design. Furthermore, it should be possible to reach, manipulate and extract nearly any given part of the model for inspection and repair. Any such system for design and evaluation of massive models needs the capability to perform interactive collision detection and proximity computations between real and virtual objects. These include checking two or more objects for overlap.

A number of algorithms have been proposed for performing proximity computations on geometric models. The commonly used algorithms utilize bounding volume hierarchies to accelerate these queries. However, no good algorithms are known for partitioning massive models automatically, computing balanced hierarchies, and ordering the queries. Furthermore, these hierarchies require considerable storage. For example, some of the recently proposed bounding volumes for fast collision detection (e.g. OBBs [3, 12], k-DOPs [17], spherical shells [18]) require many hundreds of bytes per triangle on average. For a model composed of 15 million triangles, such hierarchies will need many gigabytes of memory, much more than the available main

© The Eurographics Association 1999. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 238 Main Street, Cambridge, MA 02142, USA.

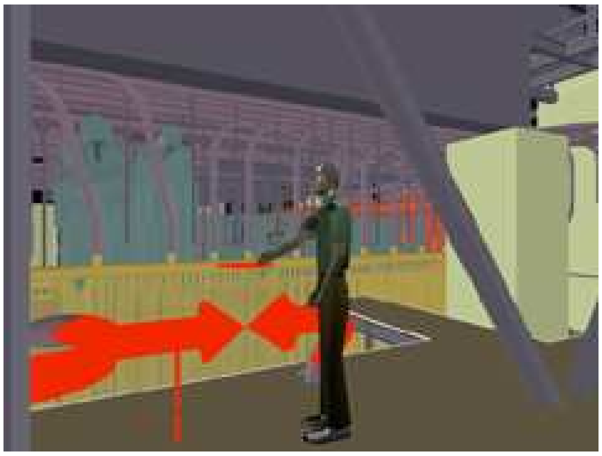

Figure 1: *Interactive collision detection & tolerance verification between a user and the pipes in the powerplant*

memory on even high-end graphics systems. As a result, earlier algorithms and the resulting systems can only handle relatively small models composed of hundreds of thousands of polygons at interactive rates. They are insufficient to handle massive models composed of several millions of polygons. This makes it difficult to achieve real-time interaction with massive models for virtual prototyping applications.

Main contributions: We present a new algorithm for performing interactive collision detection and proximity computations on massive models with a relatively small and bounded memory footprint. We introduce the concept of an *overlap graph* and use it to exploit locality of computation. For a large model, the algorithm automatically computes the proximity information between objects and represents it using an overlap graph. The overlap graph is computed off-line and pre-processed using graph partitioning, object decomposition and refinement algorithms. At run time we traverse localized sub-graphs, order the computations to check the corresponding geometry for proximity tests, and pre-fetch geometry and associated hierarchical data structures. To perform interactive proximity computations in dynamic environments, we use the bounding-volume-hierarchies, modify the localized sub-graph(s) on the fly, and take advantage of spatial and temporal coherence. The resulting algorithms have been implemented as part of a system, **IMMPACT** (Interactive Massive Model Proximity and Collision Tester), used for interactive proximity computations on a CAD model of a coal-fired power plant, composed of over 15 million triangles. The model itself takes about 1.3 GB of disk space. In practice, we are able to perform proximity computations in a few milliseconds on a SGI Infinite Reality with 195MHz R10000 processors and a memory cache size of 160 MB. In Fig. 1, we show a snapshot from our system where it interactively detects collisions between the user and the pipes in the power plant. In terms of application to massive models, we improve the performance of interactive collision detection algorithms by **an order** of magnitude.

Our algorithms and system implementation described in this paper have been specialized for interactive proximity queries on massive models for real-time interaction. However, the overall approach and algorithmic techniques are general enough to be applicable to other interactive operations that require processing and accessing of large geometric or spatial databases of complex 3D environments.

Organization: The rest of this paper is organized as follows. In Section 2 we briefly survey related work on proximity queries and management of large geometric datasets. In Section 3 we give an overview of our approach and describe algorithms to build proximity data structures in Section 4. In Section 5 we present algorithms for interactive proximity queries given the localized sub-graphs for both static and dynamic queries. Finally, in Section 6, we describe our system, *IMMPACT*, and highlight its performance.

2. Related Work

In this section, we briefly survey related work on collision detection and proximity computations as well as techniques for managing and partitioning large geometric datasets.

2.1. Collision Detection and Proximity Computations

The problems of collision detection, contact determination and distance computation have been extensively studied in computational geometry, robotics and simulated environments. For a recent survey, see [20]. Many efficient algorithms have been proposed for collision detection and distance computation between convex polytopes [21, 10, 25, 6]. Some of the recent algorithms for fast collision detection between nonconvex polyhedra and general polygonal models utilize bounding volume hierarchies. These include OBBTree [12], k-dops [17], Boxtree [3], Axis-aligned bounding boxes [7], ShellTree [18], Spheretrees [14], S-bounds, [5] etc. Bounding volume hierarchies have also been used for global distance computation [23, 15, 11]. The resulting algorithms and system provide real-time performance for relatively small models composed of hundreds of thousands of polygons. Furthermore, all the hierarchi-

© The Eurographics Association 1999

cal approaches are memory intensive, requiring many hundreds of bytes per triangle on average.

2.2. Managing Large Datasets

There is considerable work on managing large datasets corresponding to architectural models, CAD models, terrain models as well visualization datasets.

For architectural models, Teller et al. proposed techniques to compute a spatial subdivision of cells using a variant of the k-D tree data structure [27]. After subdivision, cells and portals are identified and used for visibility computation. Based on this spatial representation, Funkhouser et al. [9] construct an adjacency graph over the leaf cells of the spatial subdivision. As part of a runtime system, they only keep a portion of the model in main memory that is visible from the current observer viewpoint or that might become visible in future frames and use a pre-fetching scheme. Teller et al. [26] also proposed an algorithm for partitioning and ordering large polygonal environments for radiosity computations. They use visibility information to partition the environment into subsets and use the ordering information to minimize the number of reads and writes. Bukowski and Séquin [4] also used visibility preprocessing, spatial decomposition and database management techniques to integrate architectural walkthrough systems with simulators (e.g. fire simulators).

For large CAD models, Aliaga et al. [1] partition the model into virtual cells. At run time they ensure that the geometry and texture information associate with the current cell(s) is in the main memory and use prefetching algorithms to fetch neighboring cells. Avila and Schroeder [2] use a dynamic loading strategy to load objects and their LODs from a database. Cox and Ellsworth [8] have presented application-controlled demand paging algorithms for visualizing large CFD datasets.

3. Overview

In this section, we give a brief overview of our approach. We assume that the input model is given to us as a collection of objects. For proximity computations, we treat each object as a primitive. In many CAD environments, an "object" may correspond to a collection of disjoint and non-overlapping parts with similar functionality (e.g. all the steam pipes or all the walkways in a power plant).

Designers of these CAD environments may impose constraints on the proximity relationships of such objects. For instance, it may be necessary that the boundary of two objects should not overlap. In many cases certain objects need to be separated by a sufficient distance.

In verifying the design constraints of an entire model, one may need to perform collision detection and proximity computations on all pairs of objects. In addition, an interactive system may be needed to correct or test the placement of objects, with real time feedback regarding the proximity relationships between moving and static objects. Besides database management, our overall design goals include:

- *Efficiency*: Testing the entire model for overlaps should not significantly delay the design process. Moreover, user control of individual objects and collision feedback should run at interactive rates.
- *Automaticity*: The system shouldn't need human intervention or manual tweaking. It should be able to automatically handle a massive model.
- *Unstructured Datasets*: The system should be able to handle models with no hierarchy, structure or topological information.
- *Dynamic Environments*: Besides moving different objects, the user may insert or delete objects from the model.
- *Fixed Memory Cache Size*: The system is given a fixed memory cache size $\mathcal{M}$. In practice, $\mathcal{M}$ may be smaller than the size of the model (in terms of megabytes).

Our overall approach is based upon the local and pairwise nature of collision detection problem and we use bounding volume hierarchies to accelerate their computation. In most earlier systems, all the objects along with their bounding volumes are loaded into the main memory. However, each individual proximity test considers only one pair of objects at a time. Strictly speaking, only those two objects must actually be resident in main memory while that particular proximity test is under way. This principle can be carried even further, to the point where only a single pair of bounding volumes or polygons must be in memory at a time, but such extremes may not be useful in practice. We encode all of the proximity computations for a simulation environment in an *overlap graph*, then traverse it to determine precisely what data will soon be needed and should be resident in main memory at any time.

Given the geometry database of a massive model that is larger than $\mathcal{M}$, we first decompose each object into pieces of bounded size with k-D tree subdivision. The resulting objects are organized into a scene graph and used for static model verification as well as for interactive design review. Nodes in the scene graph are used to construct an overlap graph, a data structure used for representing proximity information. We used a number of graph partitioning algorithms to compute localized subgraphs of the overlap graph. These subgraphs correspond to sets of objects simultaneously loaded from the disk for proximity testing. The system also employs pre-fetching techniques and

© The Eurographics Association 1999

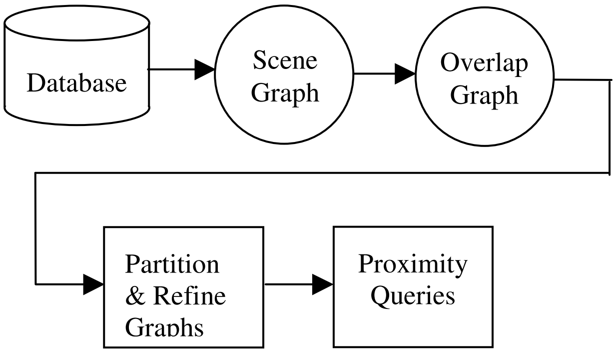

Figure 2: *Processing Large Geometric Databases for Massive Models*

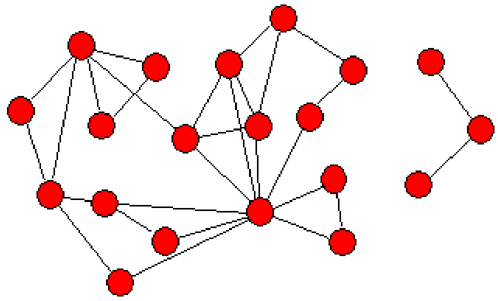

Figure 4: *The overlap graph constructed from the environment in Fig. 3. The edges in this graph correspond exactly to overlaps between bounding boxes in the original environment.*

performs proximity computations on multiple processors.

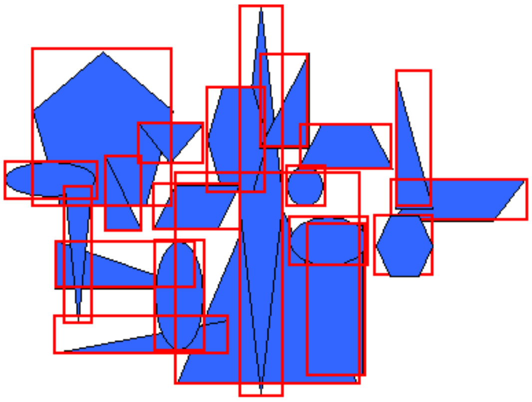

Figure 3: *An environment consisting of polygons annotated with axis-aligned bounding boxes. Figure 4 shows the overlap graph constructed from this environment.*

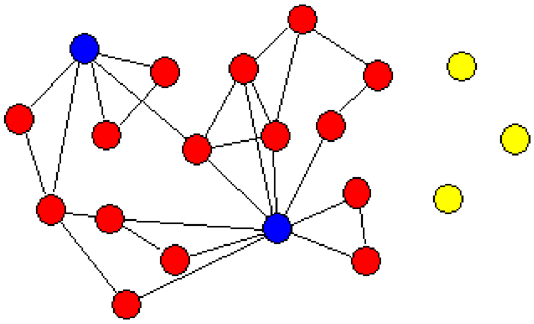

Figure 5: *Two high-valence nodes, shown in blue, are chosen and processed individually to reduce the connectivity of the overlap graph. The disconnected nodes at right have already been processed.*

4. Proximity Data Structures

In this section, we describe our pre-processing algorithm that automatically processes the model and builds an overlap graph. We make use of several graph algorithms including graph partitioning for computing localized sub-graphs of the overlap graph, and use these for ordering the proximity computations, thereby attempting to minimize the number of disk accesses.

4.1. Overlap Graph

We cast the sequence of proximity computations as the processing of an overlap graph. Each node in the overlap graph corresponds to one of the objects in the model, and edges of the graph connect the objects which may possibly overlap, or come within the required tolerance. Therefore, each edge indicates a pair of objects that must be compared by a proximity query.

In the worst case, where all objects in the model overlap all other objects, the overlap graph would contain $O(V^2)$ edges, where V is the number of nodes. Fortunately, such configurations are rare in large environments composed of tens of thousands of objects. We reduce the number of proximity tests that must be performed at runtime by not including in the overlap graph any edges between objects known trivially

© The Eurographics Association 1999

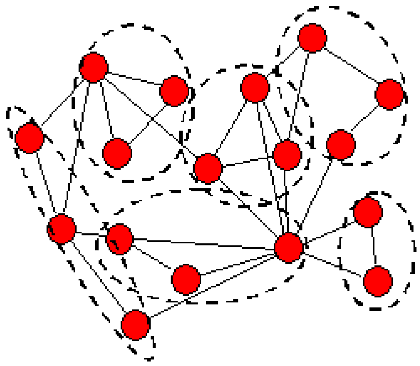

Figure 6: *Multi-level graph partitioning separates the overlap graph into groups of nodes which fit entirely into the memory cache.*

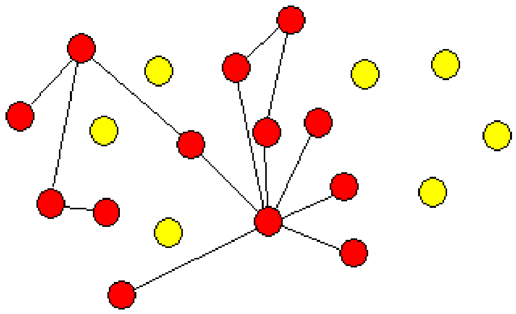

Figure 7: *Resolving edges within each partition leaves a set of edges which form the cut graph, which is processed with the same algorithms applied to the rest of the overlap graph.*

not to come into contact. To accomplish this, we annotate each node with the axis-aligned bounding box (AABB) of its corresponding object. Finally, we add edges between those nodes whose bounding boxes overlap. These bounding boxes can be computed with a single pass through the database, during which each object is loaded exactly once and then immediately discarded, and the intersection of an AABB with all others can be found efficiently using a sorted list of intervals as in [7]. As a result, a given pair of objects in the model may interfere with each other if and only if there is an edge between the corresponding nodes in the overlap graph. Figures 3 and 4 show a simple environment and its corresponding overlap graph.

For tolerance verification, a pair of objects are linked by an edge in the proximity graph when their AABBs are within the tolerance. To make use of the interval sorting, we simply add an offset to each AABB, so that if the "offset AABBs" do not overlap, the original AABBs will not come within the tolerance.

Furthermore, with each node of the graph we associate a weight that corresponds to the memory required by a bounding volume hierarchy for that object. This weight varies as a linear function of the number of polygons in the object, and the constant of proportionality varies based on the particular bounding volume (e.g., sphere, AABB or OBB). The weight associated with any subgraph is computed by adding the weights of all the nodes of that subgraph.

Our algorithm orders the proximity computations mandated by edges of the overlap graph by computing subgraphs that each have a weight less than the size of the memory cache, $\mathcal{M}$. Objects in the subgraph are loaded into memory, and the pairs of objects linked by an edge are checked for collisions. Once the result for an edge is recorded, this edge is removed from the overlap graph. We call these subgraphs *localized subgraphs*. They are computed to exploit memory locality.

4.2. Static Environments

As previously stated, the edges of the overlap graph specify object pairs that may violate some of the proximity constraints. However, since only a subset of the entire model's objects may fit in the memory cache at once, performing all the queries may require repeated loading of objects from disk. We present an approach that uses the overlap graph to order the proximity computations so that the objects are loaded from a disk as few times as possible.

4.2.1. Computing Connected Components

Separate connected components of the overlap graph can be considered in isolation, since objects whose nodes reside in different components need never be loaded into the memory cache simultaneously. Furthermore, if a component of the graph has a total weight smaller than the size of the object cache, all the objects represented by nodes of this component may be loaded into the memory cache at once, and all queries within the connected component performed without any further loading of objects.

Connected components which are too large for the memory cache are handled by three sub-algorithms:

1. **Decomposing Objects:** We decompose objects with high polygon count into two or more sub-objects whose sizes are each a fixed fraction of the size of the memory cache. We subdivide the geometry of an object using a k-D tree constructed at the centers of

© The Eurographics Association 1999

its component polygons. This k-D tree subdivision is computed from building the scene graph. For each sub-object, we create a separate node in the overlap graph. Each has a pointer to their parent object to avoid redundant comparison between the same pair of objects.

2. **Separately Handling High Valence Nodes:** We choose a set of nodes with high valence so that their total weight, plus the weight of any other node in the component, is less than the size of the memory cache $\mathcal{M}$. Fig. 5 shows two such high-valence nodes.

By swapping the neighbors of these nodes into the memory cache one at a time, all the proximity computations represented by edges incident to high valence nodes are evaluated. These edges can then be removed from the overlap graph. Note that this step can be performed by loading each object of a component only once. We decompose the remaining graph using multi-level partitioning algorithms.

3. **Multi-Level Graph Partitioning:** This involves three phases: coarsening, partitioning, and ordering or uncoarsening [13, 16]. To coarsen the graph, we use the weights of the vertices and ensure that the size of the partition of the coarse graph is within a small factor of $\mathcal{M}$. After coarsening, a bisection of this much smaller graph is computed, and then this partitioning is progressively projected back onto the original graph (the finer graph). At each step of the graph uncoarsening, the partition is further refined. The overall process involves:

- **Coarsening Phase:** The graph G_0 is transformed into a sequence of smaller graphs $G_1, G_2, \ldots, G_m$ such that $|V_0| > |V_1| > \ldots > |V_m|$.
- **Partitioning Phase:** A 2-way partition P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_0.
- **Uncoarsening Phase:** The partition P_m of G_m is projected back to G_0 by going through intermediate partitions $P_{m-1}, P_{m-2}, \ldots, P_1, P_0$. At each of these steps, the partition is further refined as highlighted in [13]. Since the finer graph has more degrees of freedom, such refinements usually decrease the number of edges crossing between partitions.

Figure 6 shows the results of such partitioning on a simple graph.

Finally, the edges that link objects in different partitions, and along with the incident nodes, form a new graph that we call the *cut graph*. (See Fig. 7.) We compute its connected components and recursively apply the three sub-algorithms.

This processing does not need to be interleaved with proximity queries. We apply these algorithms to precompute the localized subgraphs, where each subgraph corresponds to a set of objects that will be loaded into memory together.

We repeat them until we can decompose the overlap graph into localized subgraphs, $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k$, such that the weight of each subgraph is less than $\mathcal{M}$.

4.3. Runtime Ordering & Traversal

Given the localized subgraphs $\mathcal{L}_i$, we traverse them to check their component objects for proximity. The traversal is rooted at the node with the greatest number of edges and proceeds in a breadth-first fashion, with neighboring nodes visited in descending order of their valences.

During traversal, object geometry and bounding-volume hierarchies are cached in main memory. By looking ahead to the next few proximity tests to be performed (based on the graph representation), we are able to prefetch geometry and compute bounding volume hierarchies in advance. More details about the proximity computations are given in Section 5. After the traversal of each subgraph terminates, memory used by its component objects is released to be reused by subsequent traversals.

4.4. Overlap Graph in Dynamic Environments

In many scenarios, objects in a model may be moved by the user or new objects may be added to or deleted from the environment. These objects are treated as floating nodes in the overlap graph. For each floating node, we maintain a list of potential overlaps with objects in the rest of the world. These lists are updated and evaluated each time the node moves. The potential-overlap lists are maintained using AABBs and a sweep and prune algorithm similar to the one in [7] to exploit coherence between time steps. Bounding volume hierarchies for pairs of objects on the potential-overlap lists are constructed lazily and used to evaluate the proximity queries corresponding to edges in the lists.

4.5. Prefetching Geometry

The algorithm uses temporal and spatial coherence to pre-fetch geometry on one processor, while it is performing queries on the other processors. For static environments, it makes use of the ordering specified by the edges of the localized subgraphs to pre-fetch object geometry. For dynamic objects, the algorithm estimates the velocities of moving objects. Based on these velocities and the time interval used for prediction, it expands the AABBs of moving objects by an appropriate offset. The algorithm pre-fetches the geometry corresponding to all the nodes overlapping with the "expanded" AABBs.

© The Eurographics Association 1999

5. Proximity Computations

In this section, we briefly describe the algorithms based on bounding volume hierarchies used for performing collision, distance, and tolerance tests. A number of algorithms have been proposed in the literature for these computations based on hierarchies of bounding volumes. They vary based on the choice of bounding volume, whether the trees are constructed in a top-down or bottom-up manner and the order of traversal (depth-first or breadth-first). As a result, the performance of different algorithms varies in terms of speed, storage requirements, and robustness on different models. In our system, we have provided support for three different bounding volumes (spheres, AABBs and OBBs) as part of a generic framework where one can easily introduce a new bounding volume.

Given a large model, the tree of tight-fitting bounding volumes is constructed top-down by recursively subdividing a group of primitives (polygons, triangles, etc.) using statistics of vertex distribution, linear algebra and geometric techniques [12]. After tree-building, any proximity test, whether collision detection, distance computation or tolerance verification, proceeds by recursively checking bounding volumes for the desired queries. If the parent bounding volumes (BVs) fail the query, then the children of these BVs are tested pairwise. If the children satisfy the query condition, then that recursion branch terminates. Otherwise, the recursive test continues in a similar fashion.

If the query is collision detection, then the query condition is to check BVs for overlap. If the query is distance computation, then the test checks whether the separation between the current BVs is greater than the upper bound distance attained so far. Initially the upper bound is set to the distance between any two points on the model. At each node, the algorithm computes the distance between all four cross-pairs of children nodes and recursively traverses the closest pair of nodes after comparing it with the global minimum distance. Finally, if the query is tolerance verification, then the recursion terminates when the BVs are separated by more than the user-specified threshold amount.

5.1. Choice of Bounding Volumes

Our system for interactive proximity queries also allows the user to select from a palette of desirable options using compile-time switches. These switches control conditional compilation of the source code using the #if C++ compiler directive to effectively specialize the code to suit the needs of applications. The basic system prompts the user to make application-dependent choices regarding the bounding volume type, coordinate system for updates (nested or flat) and tree traversal scheme (breadth-first, depth-first or priority-directed). Defaults are used when none is specified.

The type of bounding volumes available include spheres, axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs). This choice affects memory usage, tree pruning and bounding volume overlap tests. The selection of bounding volumes is likely to change depending on the geometry of the models used in the applications, the nature of interactions with the virtual environments, contact frequencies and configurations, and the type(s) of most-frequently performed proximity computations.

5.2. Lazy Hierarchy Construction

The system also allows the user to have the trees of the bounding volume hierarchies built on an as-needed basis. Children of a node are constructed just prior to being visited. As a result, only those portions of the trees which get visited are actually built. For a short sequence of queries this can yield significant time and storage savings. Interaction with a massive model is often localized to only a small region of the model. For the hierarchy construction, we use a top-down approach based on the vertex distribution [12] to compute tight-fitting bounding volumes.

6. System Implementation and Performance

In this section, we describe the implementation of our system. This includes a system overview, graph partitioning algorithms, and the runtime system for dynamic environments. We also highlight its performance on a CAD model of a coal-fired power plant composed of 15 million triangles (as shown in Fig. 12). The model came to us as a collection of more than 1800 objects or function groups with no topology, structure or hierarchy information. It occupies more than 1.3 GB of disk space.

6.1. Scene Graph

Our scene graph closely resembles that of IRIS Performer [24]. Objects are contained in the leaf nodes of the scene graph, and each internal node is annotated with the bounding box of all of its children. Each of the roughly 1800 functional groups from the original model becomes a subtree whose root is a grandchild of the root of the entire graph. The direct children of the root are simply containers for these functional groups. To quickly render objects being checked for interference (as part of a runtime system), we generate multiple geometric levels of detail (LODs) for most

© The Eurographics Association 1999

objects. The LODs are stored in the scene graph as siblings of the original geometry. They are only used for rendering and not proximity queries.

6.2. Bounding Volume Hierarchies

Our test model, the coal-fired power plant, consists of many complex piping structures that are axis-aligned. Spheres are not a good approximation for this type of geometry. Since the user can only interact with a small portion of the massive model at a time (due to size differential), most part of the massive model can be assumed to be stationary. Furthermore, OBBs require more storage than AABBs in general and one of our goals is to minimize the frequency of disk access. Therefore, we have used AABBs as the bounding volumes in performing queries on the power plant. To reduce the memory overhead, the hierarchies are not fully traversed during interference tests, and we used lazy construction. Only the root of the tree is created during initialization, and construction of further levels is deferred until some interference test accesses them.

6.3. Graph Partitioning and Refinement

We applied the partitioning algorithm (composed of three sub-algorithms) presented in Section 4 to perform proximity queries between objects in the power plant. The estimated memory usage per triangle was 200 bytes (since we are using double precision arithmetic), including space to store the triangle itself and the overhead of AABB hierarchy construction. This allowed a conservative choice of object cache size, given a particular memory limitation. For instance, our target memory cache size, $\mathcal{M}$, was 160 Megabytes, which corresponds to about $800,000$ triangles.

Object decomposition, or node splitting, during the graph processing was based on k-d tree decompositions of objects in the scene graph. Each object was decomposed into some set of descendants in its k-d tree, such that each descendant was no larger than one-tenth of the size of the cache, (i.e. $80,000$ triangles). We used a public domain implementation of a multi-level partitioning algorithm, METIS [16], available from the University of Minnesota. High valence node removal and partitioning were applied in alternation. When one sub-algorithm was used to decompose a component, any resulting components still larger than the cache size were decomposed by the other sub-algorithm. We found that using these sub-algorithms together resulted in better cache utilization than one method alone.

6.3.1. Impact of Cache Size

Our graph partitioning and refinement algorithms try to minimize the number of disk accesses. We applied

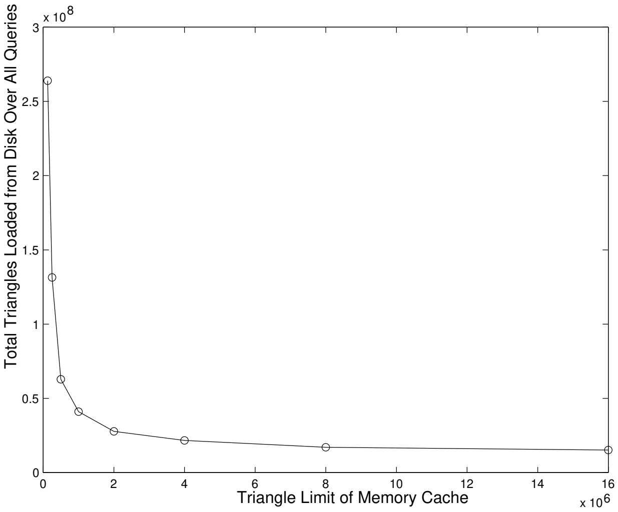

Figure 8: *This graph highlights the data fetched from the disk during graph partitioning and refinement algorithm as a function of cache size. While a small cache size ($< 250K$ polygons) results in a very high number of disk accesses, the algorithm is able to efficiently partition the model and perform proximity queries with a cache size of $800K$ polygons. The model is composed of more than 15 million triangles.*

the partitioning sub-algorithms to the power plant model with several different cache sizes. In Fig. 8, we show the number of triangles loaded from the disk as a function of the cache size. For a small cache for 150K triangles (i.e. 30MB), we need to load each triangle 60 times from the disk on average. However, with a cache $800K$ triangles (i.e. 160 MB) we load each triangle about 4.2 times on average. Notice that we would otherwise need more than 3.2GB to load the entire model and its bounding volume hierarchy.

6.4. System Pipeline

We have divided our system into three separate phases: collide/proximity query, render/draw, and prefetch, as shown in Fig. 9. The collide phase is responsible for traversing the overlap graph, determining which proximity computations must be performed, and evaluating those tests. The render phase displays the objects currently being examined. Finally, the prefetch phase is responsible for looking ahead to tests soon to be performed and retrieving from disk any objects that are not already available in main memory.

© The Eurographics Association 1999

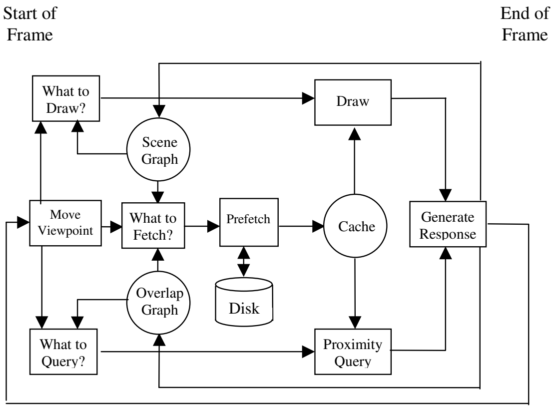

Figure 9: *Overview of the Interactive Proximity Query System*

6.4.1. Collide Phase

The proximity queries are performed during the collide phase. We implement this phase as two or more processes: one to traverse the overlap graph, and one or more to perform the proximity computations indicated by the traversal process. This allows us to take greater advantage of multiprocessor configurations. Each individual collide process requests object data from the prefetch phase on demand.

6.4.2. Prefetch Phase

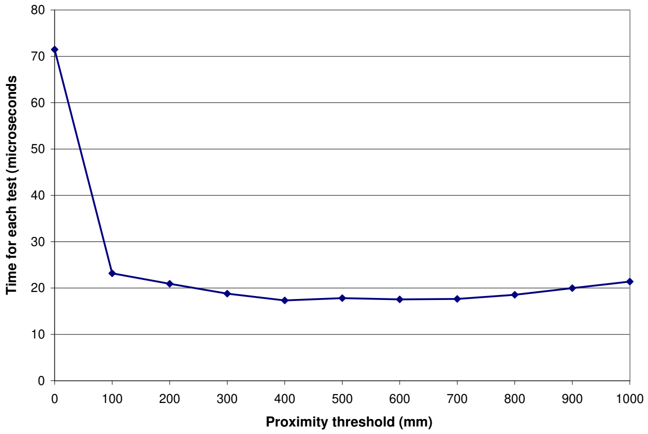

Figure 10: *This graph shows the average time for a proximity query along a sample path. The proximity threshold corresponds to the value used for tolerance verification. A zero value indicates collision.*

The prefetch phase is responsible for ensuring that objects and renderables are available in main memory at the moment when they are needed for rendering or proximity testing. For static environments, this is accomplished by traversing the overlap graph in exactly the same manner as the collide tasks, but staying a few steps ahead of the collide tests and loading the two objects in each test instead of actually testing them. These objects are maintained in a memory cache whose size $\mathcal{M}$ is given as a parameter to the graph partitioning and refinement procedures. To take advantage of the localized nature of our method, this cache is maintained with a *least-recently-used* eviction policy. We implement the prefetch phase as a single, free-running process that accepts requests for objects from the render and collide phases and provides access to the contents of the model cache. If a particular model is accessed before it has been loaded from disk, the request is blocked until the data is available.

6.4.3. Render Phase

The render phase displays on-screen the two objects currently being checked for collision or proximity. Particularly in a massive model, it is possible for objects to be large enough that rendering them may be significantly slower than performing a proximity test between them. For this reason, we may disable the render phase when dealing with a strictly static environment. One possibility is to run the collision and rendering tasks asynchronously; however, that can result in race conditions. In a dynamic environment, the render phase drives the rest of the computation. During each frame, the render phase quickly traverses the scene graph to find objects which might overlap or might soon overlap dynamic objects under the user's control. We do not even attempt to render distant objects in this system. Any necessary proximity queries are dispatched to collide tasks, and any necessary data are requested from the prefetch task. As soon as the results of the proximity queries are available, the objects for the current frame are drawn to indicate whether or not they participate in a overlap. We implement the render phase as a single task in order to avoid costly OpenGL context switches during rendering.

6.5. Performance

The system has been used to perform a number of static and dynamic proximity queries on the power plant models. These include finding all interferences or objects within a tolerance threshold. We controlled the motion of a avatar, modeled with $4,000$ triangles, and were able to interactively perform collision detection and tolerance verification queries in a few milliseconds on a SGI Infinite Reality with four 195 MHz processors and using a memory cache of 160 MB. The performance of the algorithm along some sample paths is shown in Graph 10 and 11. Color plates II-IV show some snapshots from our system. Movies showing live footage of our system in operation as well as

© The Eurographics Association 1999

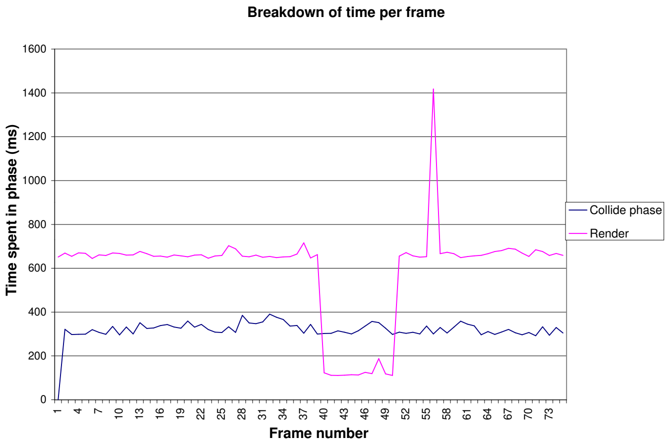

Figure 11: *Time spent between the collision phase and rendering phase for a sample path*

a pre-rendered zoom on the power plant model can be downloaded from the following URL:

http://www.cs.unc.edu/˜awilson/mmc/

7. Conclusion and Future Work

In this paper we have presented an algorithm and a system to perform proximity computations at interactive rates on massive models. As part of pre-processing, our algorithm automatically computes proximity data structures in terms of overlap graphs and localized sub-graphs and tries to minimize the number of disk accesses. We use bounding volume hierarchies to accelerate proximity queries and present algorithms that load a small and local subset of the model in the main memory. We have implemented our algorithm as a system called IMMPACT and used it to perform simple interactions with the model of a coal-fired powerplant composed of 15 million triangles with a memory cache size of 160 MB. We believe that our algorithm and system scale well with the model size. In terms of application domain, it can perform proximity queries on models composed of tens of millions of polygons at interactive rates. Earlier systems for collision detection and tolerance verification could only handle models composed of hundreds of thousands of polygons at interactive rates.

There are many avenues for future work. We would like to perform more complex interaction tasks using our system. Many designers are interested in automatic placement of parts, given some tolerance constraints. We would like to use robot motion planning algorithms for computing collision free configurations and paths. Finally, we would combine this system with an interactive massive model rendering system [1] and use them for simulation-based design applications.

8. Acknowledgement

We are grateful to Stefan Gottschalk for providing us with a framework to implement different bounding volume hierarchies [11]. The IMMPACT system is build on top of that framework. We are also also grateful to James Close and Combustion Engineering Inc. for providing us with the model of a power plant.

References

1. Aliaga et al. A framework for real-time walkthroughs of massive models. Technical Report TR98-013, Department of Computer Science, University of North Carolina, 1998.

2. Lisa Sobierajski Avila and William Schroeder. Interactive visualization of aircraft and power generation engines. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization 9́7*, pages 483–486. IEEE, November 1997.

3. G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. Boxtree: A hierarchical representation of surfaces in 3d. In *Proc. of Eurographics'96*, 1996.

4. Richard Bukowski and Carlo H. Séquin. Interactive simulation of fire in virtual building environments. In *SIGGRAPH 97 Conference Proceedings*, pages 35–44, 1997.

5. S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

6. Stephen Cameron. A comparison of two fast algorithms for computing the distance between convex polyhedra. *IEEE Transactions on Robotics and Automation*, 13(6):915–920, December 1996.

7. J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995.

8. Michael B. Cox and David Ellsworth. Application-controlled demand paging for Out-of-Core visualization. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization 9́7*, pages 235–244. IEEE, November 1997.

9. T. Funkhouser, C. Sequin, and S. Teller. Management of large amounts of data in interactive building walkthroughs. In *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 11–20, 1992.

© The Eurographics Association 1999

10. E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:193–203, 1988.

11. S. Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, University of North Carolina. Department of Computer Science, 1998.

12. S. Gottschalk, M. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

13. B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratory, 1993.

14. P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

15. D. Johnson and E. Cohen. A framework for efficient minimum distance computation. *IEEE Conference on Robotics and Automation*, pages 3678–3683, 1998.

16. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, pages 269–278, 1996.

17. J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. In *Siggraph'96 Visual Proceedings*, page 151, 1996.

18. S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. In *Proc. of Eurographics'98*, 1998.

19. E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Swept sphere volumes for fast proximity queries. Technical report, Department of Computer Science, University of North Carolina, 1998.

20. M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.

21. M.C. Lin and John F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

22. Brian Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

23. S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

24. J. Rohlf and J. Helman. Iris performer: A high performance multiprocessor toolkit for realtime 3d graphics. In *Proc. of ACM Siggraph*, pages 381–394, 1994.

25. R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

26. S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and ordering large radiosity computations. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, pages 443–450. ACM SIGGRAPH, 1994.

27. S. J. Teller. *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.

© The Eurographics Association 1999

Figure 12: *CAD model of a coal-fired powerplant with more than 15 million triangles. The model consists of more than 1800 objects and takes more than 1.3GB on disk.*

Figure 13: *Proximity queries between an avatar and the powerplant model. IMMPACT takes a few milliseconds to perform these queries.*

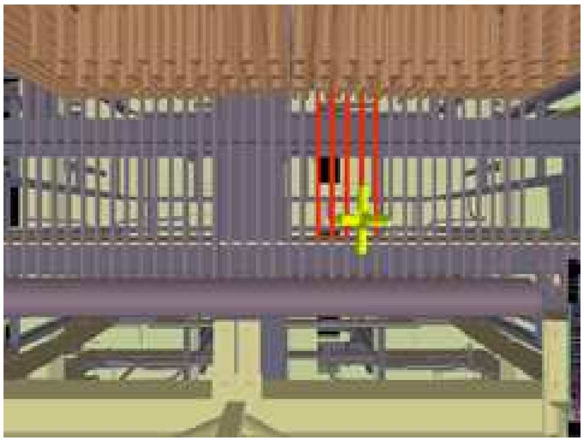

Figure 14: *Interactive collision detection between a moving objects and pipes in the powerplant.*

© The Eurographics Association 1999