Simplification Culling of Static and Dynamic Scene Graphs

Carl Erikson, Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

We present a new approach for simplifying large polygonal environments composed of hundreds or thousands of objects. Our algorithm represents the environment using a scene graph and automatically computes levels of detail (LOD) for each node in the graph. For drastic simplification, the algorithm uses *hierarchical levels of detail* (HLOD) to represent the simplified geometry of whole portions of the scene graph. When HLOD are rendered, the algorithm can ignore these portions, thereby performing *simplification culling*. For dynamic environments, HLOD are incrementally computed on the fly. The algorithm is applicable to all models and involves no user intervention. It generates high quality and drastic simplifications and has been applied to CAD models composed of hundreds of thousands of polygons. In practice, it achieves *significant speedups* in rendering large static and dynamic environments with little loss in image quality.

Keywords: simplification, levels-of-detail, dynamic, environments, walkthroughs, interactive display

1 Introduction

The real-time display and walkthrough of large geometric environments is an important problem in computer graphics. Models composed of millions of polygons are common in CAD and scientific visualization applications. Current graphics systems are unable to render such models at interactive rates. A number of techniques that reduce the number of rendered polygons have been proposed in the literature. One of the major approaches involves computing different LOD (levels of detail) of a given object or portions of an environment. Typically, different LOD are used based on their distance from the viewpoint.

One of the goals of research in this area is to compute LOD for interactive walkthroughs of large environments composed of thousands of objects. Furthermore, the objects in the scene may move. Such scenarios are common in design evaluation of large assemblies, where a designer moves, adds, and deletes parts. Other applications of dynamic environments include simulation-based design, driving simulators, battlefield visualization, urban environments, and entertainment software. All of them require *interactive manipulation* of objects in the scene. As opposed to re-computing the LOD of the entire environment, which can take several minutes, we would like to continue displaying the dynamic scene at interactive rates.

1.1 Main Contribution

In this paper, we present algorithms to *automatically* generate simplifications of large static and dynamic polygonal environments. We represent these environments using scene graphs. Each node in the scene graph corresponds to an object in the environment and its LOD are stored in display lists. In addition to LOD, we add HLOD (hierarchical LOD) to enable drastic simplification of the scene. The HLOD of a node are computed by simplifying the lowest LOD of the node combined with the transformed lowest HLOD of each of the node's children. The algorithm performs topological simplification and uses a surface area preservation metric to preserve the basic shape and important features of objects. For dynamic environments, the algorithm *incrementally* updates HLOD. They are computed in a lazy manner, taking into account the degree of motion in the scene and the user's viewpoint. HLOD represent the simplified geometry of whole sections of the scene graph. When a node's HLOD is rendered, the algorithm need not traverse any of its descendants. This process of ignoring entire portions of the scene graph is called *simplification culling*. Our topological simplification algorithm is a *critical* component to facilitate simplification culling. It can efficiently obtain significant reductions in polygon count by merging disjoint objects and produce high quality approximations, capabilities *fundamental* for the creation of HLOD.

Some of the main features of our approach are:

- **Generality:** The algorithm is applicable to all polygonal environments. The input model may consist of meshes, non-manifold surfaces, and models with cracks and T-joints.
- **Automatic:** Our algorithm for object simplification and topological merging requires no input parameters.
- **Drastic Simplifications:** For each node of the scene graph, the algorithm is able to produce a very low polygon count approximation.
- **Fidelity:** The algorithm is able to generate high quality approximations for each node, retaining the basic shape and main features of the underlying geometry.
- **Scene Graph Simplification:** HLOD can be substituted for entire portions of the scene graph to perform simplification culling. As a result, the renderer traverses fewer nodes and performs fewer transformations.
- **Efficiency:** Our algorithm simplifies large static models relatively quickly. Simplifying a one million polygon model takes less than an hour. Furthermore, for dynamic environments the algorithm is able to incrementally compute HLOD in a few seconds.

We have applied the algorithm to a few static and dynamic environments consisting of hundreds of thousands of polygons and hundreds of objects. We achieve up to a 10 times speedup, depending on viewer position, with little loss in image quality.

1.2 Organization

The rest of the paper is organized in the following manner. We survey related work in model simplification and interactive display of large models in Section 2. Section 3 presents an overview of our approach. Our algorithm for simplifying polygonal objects is described in Section 4. We use it for simplifying static environments in Section 5 and extend it to dynamic environments in Section 6. Section 7 discusses our implementation and its performance on various models. We compare our approach with other simplification algorithms in Section 8.

2 Related Work

There is considerable work on model simplification, interactive display of large models and dynamic environments in computer graphics, GIS, and computational geometry.

2.1 Simplifying Polygonal Models

Most of the earlier work in polygonal simplification focuses on generating LOD of a given object. Different algorithms have been proposed, and they vary in terms of assumptions on the input model, underlying geometric operations used for computing the simplified model, error metrics, and the quality of the approximations generated. The common set of geometric operations correspond to *vertex removal* and re-triangulating holes [4, 22, 34, 36, 40], *edge collapses* [5, 15, 19, 20, 29], and triangle removal [16]. Most of these algorithms assume that the input model is a manifold and represented as a valid mesh. Other algorithms include those based on multi-resolution wavelet analysis [9], simplification envelopes [6] and progressive meshes [19]. All of these algorithms have been designed for manifold surfaces.

For general polygonal models, algorithms based *on vertex clustering* have been proposed by [24, 25, 30, 31]. These algorithms allow topological changes as the model is simplified. Schroeder [33] present a topology modifying decimation algorithm that can guarantee a requested reduction level. He et al. [17] and El-Sana and Varshney [10] demonstrate algorithms for controlled simplification of genus of polyhedral objects.

2.2 Simplification Using Quadric Error Metrics

Recently, Garland and Heckbert [11] presented a surface simplification algorithm using quadric error metrics. It uses *vertex collapse* as its basic simplification operation. As an initialization step, the algorithm determines candidates for vertex collapse. Candidates are any two vertices that are within a user specified distance threshold *t* of one another or are connected by an edge. Each vertex has an associated error quadric that is used to efficiently approximate the sum of distances between the vertex and several infinite planes. Each candidate vertex collapse is inserted into a minimum cost heap, weighted by the error induced by the operation according to sum of the error quadrics of the vertices. The location of the collapsed vertex is found by minimizing the error according to the metric. The algorithm iterates until a vertex collapse introduces too much error, or the model has been reduced to a target number of triangles. In practice, this algorithm produces very impressive results. It is simple, general, efficient, and produces high quality polygonal approximations of surfaces. For our applications, we extend this algorithm in three ways. Our algorithm handles attributes, it automatically and adaptively determines a distance threshold to use for determining collapse candidates, and it promotes merging unconnected regions of the model through surface area preservation. We address these issues in Section 4.

2.3 Large Static Environments

When dealing with large static environments, the object simplification and topology modification algorithms highlighted above can be combined with a suitable hierarchy. Based on the viewpoint, the portion of the hierarchy outside the view frustum is culled away and the rendering algorithm uses a suitable LOD for each visible node. The vertex clustering algorithm proposed by Rossignac and Borrel [30] is used in BRUSH [32] and applied to a number of CAD models. Similarly, Cohen et al. [6] use LOD generated by simplification envelopes along with a Performer scene graph representation on CAD models.

Another technique for handling large static environments is based on *view-dependent simplification* [18, 25, 41]. More details about this technique are given in Section 3.1.

2.4 Interactive Display of Large Models

Many techniques besides LOD have been proposed for interactive display of large environments. These include visibility culling [1, 7, 14, 38, 42], dynamic tessellation of spline models [23, 28] and image-based representations [2, 26, 35]. Many of these algorithms can be combined with LOD to render large polygonal environments.

2.5 Dynamic Environments

There is considerable literature in computational geometry and GIS related to dynamic environments. Most of the work in computational geometry focuses on the design of dynamic data structures and their applications to proximity and point location problems [3, 12, 13]. Sudarsky & Gotsman [37] present an output sensitive visibility algorithm that minimizes the time required to update a hierarchical data structure. Torres [39] use binary space partition trees for visualization of dynamic scenes. More recently, Drettakis and Sillion [8] present an algorithm which provides interactive update rates of global illumination for scenes with moving objects. Jepson et al. [21] describe an environment for real-time urban simulation that allows dynamic objects to be included in the scene.

3 Overview

In this section, we give an overview of our approach and highlight many issues of its design. One of our *main* goals is to render large environments of polygonal objects at interactive frame rates. We assume that the application used to view these objects is either automatically trying to achieve a constant frame-rate or allows the user to change a screen space error tolerance on the fly. Therefore, the application may be forced to substitute a coarse LOD for an object even when it is near the viewer. As a result, we want our simplification algorithm to be simple, general, automatic, and efficient. Most of all, it must efficiently produce high quality approximations when performing drastic reductions of models. This property is crucial for creating and updating HLOD. All design choices in this paper are motivated by these criteria.

3.1 View-Dependent vs. View-Independent

Many researchers [18, 25, 41] have proposed using *view-dependent simplification* for large models. These algorithms adaptively simplify across the surface of an object. They store simplifications in a hierarchical tree of vertices produced by collapse operations and traverse this tree when rendering. Different types of selective refinement criteria based on surface orientation and screen-space projected error are used. Through the use of *geomorphs* for progressive meshes [19] and adaptive simplification at the vertex level, rather than the object level, view-dependent algorithms provide an elegant solution to the *popping problem* associated with static LOD. Also, since view-dependent systems render spatially large objects using adaptive simplification, they eliminate the inefficiency of using static LOD to represent them.

Although view-dependent algorithms are elegant and provide nice capabilities, there are many *drawbacks* to using these algorithms for our applications. The implementation and optimization of view-dependent algorithms is relatively complex. Furthermore, they impose a good deal of overhead in terms of memory and CPU during the execution of the viewer. Instead of choosing an LOD per visible object, view-dependent algorithms typically refine every active vertex of every visible object. These algorithms can handle instancing, but do so inefficiently since each instance must contain its own list of active vertices. Even though view-dependent algorithms handle spatially large objects well, they may spend considerable CPU time simplifying regions of these objects that lie outside the view frustum, instead of ignoring them all together. Furthermore, *display lists* often run two to three times faster than immediate mode on current highend graphics systems and typically view-dependent algorithms use immediate mode techniques. As a result, they must simplify the environment two to three times more to achieve a frame rate comparable to using display lists for static LOD. Finally, current algorithms for computation of a hierarchical tree of vertices can be *slow* for dynamic environments. As a result, our approach *uses static LOD*.

3.2 High Quality and Drastic Simplifications

To achieve significant improvement in frame rate, a simplification algorithm must generate drastic simplifications of objects. This involves merging disjoint regions and modifying topology. At the same time, we are interested in preserving important features of the models.

To motivate the need for high quality drastic simplifications, let us consider simplifying an object that consists of an $n \times n$ grid of triangulated squares of length s with a distance d between horizontal and vertical neighbors. If d is much greater than s, then the approach taken by previous algorithms is to simplify each square individually, resulting in either all the squares disappearing or each square becoming a triangle. None of these alternatives is attractive. To achieve a *target frame rate* in our driving applications, it is possible for a coarse LOD to be rendered close to the viewer. If there are no triangles in the lowest LOD, the entire field of squares will vanish. This *disappearance* is visually unacceptable. If each square can be reduced to only a triangle, then we limit the possible amount of simplification to half the original number of triangles. This limit is contrary to our goal of drastic simplification since the lowest LOD of the object will contain at least n^2 triangles. If n is large, this restriction on simplification becomes unacceptable. One of the goals of our approach is to produce high quality and drastic simplifications for such environments.

3.3 Scene Representation

We represent polygonal environments using a Given a *scene graph*. The scene graph we use consists of *nodes* and *links*. A node contains a pointer to a model and its LOD, a bounding volume, and pointers to a list of links to other nodes. The bounding volume of a node bounds its own geometry plus all of the bounding volumes of the node's children. A link contains a pointer to a node and a transformation matrix. An example scene graph for a simple environment is shown in Figure 1. As part of pre-processing, the algorithm computes LOD for each node and stores them as display lists. Furthermore, it partitions spatially large objects and generates LOD for each partition. More details are given in Section 5.3.

3.4 HLOD and Simplification Culling

A key component of our approach is computing HLOD for each node and using them for simplification culling. The HLOD are computed in *addition* to the LOD for a node. We illustrate the representation, computation and application of HLOD for a simple environment in Figure 1. The numbered circles are nodes in the scene graph. Each node's bounding sphere, in blue, and geometry are connected by a blue line. The bounding sphere bounds the geometry of the node and the bounding spheres of the node's children, shown in green. The LOD for the geometry of nodes are grouped with braces. A node's HLOD are grouped with braces, bounded by a red rectangle, and connected to the node by a red line. The arrows connecting the nodes are links. The larger unfilled arrow symbols near the links depict the link's transformation. For example, the head in node 2 is translated upwards to rest on the torso in node 1.

We illustrate the *difference* between earlier rendering algorithms that use view frustum culling and LOD with our

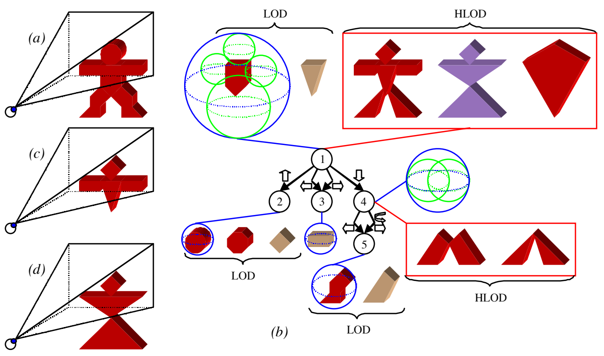


Figure 1: *A scene graph of a simple model. (a) Current view of the original model. (b) Model's scene graph containing LOD and HLOD. (c) Geometry rendered using view-frustum culling and LOD. (d) Geometry rendered using view-frustum culling, LOD, and simplification culling via HLOD.*

algorithm that, in addition, uses simplification culling. In Figure 1(c), the rendered image is generated using view frustum culling and LOD. In this case the rendering algorithm traverses the scene graph in the following order: 1-2-1-3-1-3-1-4-1. Tan geometry indicates the LOD rendered. Both legs are culled since the entire bounding sphere of node 4 is outside the view-frustum. In Figure 1(d) we highlight our rendering algorithm, which uses view frustum culling, LOD, and simplification culling via HLOD. We start the traversal of the scene graph at node 1. We render the purple HLOD in node 1 because it is an acceptable approximation from this viewpoint. We stop our traversal since this HLOD is a crude approximation to the whole scene graph. This *process* of ignoring portions of the scene graph due to HLOD rendering is called *simplification culling*.

3.5 Dynamic Environments

Our algorithm also handles dynamic environments. In general, scene graphs that support static LOD assume objects are *rigid* bodies. We make this assumption for our scene graph augmented with HLOD as well. Objects move only as a result of transformations changing in the links of the scene graph. To support dynamic environments we allow several operations on the scene graph. A user can modify the scene graph by adding nodes and links, deleting nodes and links, and by changing transformations of links. This last operation is the most common for environments composed of moving objects.

Since our environments consist of rigid-bodies, we expect that LOD for inserted nodes are pre-computed. If not, we compute the LOD on the fly, but not in real-time. HLOD for static environments can be pre-computed as well, since the scene graph does not change. In dynamic environments, the structure of the scene graph and the transformations at links constantly change. Therefore, the *fundamental* problem with dynamic environments using our scene graph representation is that the HLOD change as the scene graph changes. We need to efficiently and incrementally re-compute HLOD when objects move.

4 Simplifying Polygonal Objects

In this section, we present an algorithm that uses vertex collapses and the quadric error metric proposed by Garland and Heckbert [11]. In addition, our algorithm handles surface attributes and promotes merging of unconnected regions of the model through the use of *surface area preservation*. It is automatic and requires no specification of any parameters. The main factor for our algorithm's design is that HLOD generation,

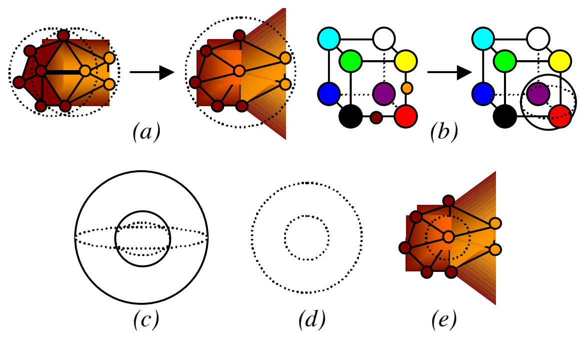


Figure 2: *Attribute handling. (a) Vertex collapse involving color interpolation. The bounding circles are adjacency bounding volumes. (b) Growth of color bounding volumes due to collapse. (c) Ratio of new color bounding volume to maximum color bounding volume. (d) Ratio applied to new adjacency bounding volume. (e) Color distance error due to collapse.*

highlighted in Section 5, requires high quality simplification of coarse models.

4.1 Surface Attributes

In general, handling surface attributes during simplification is an important, but very difficult problem that involves many unsolved perceptual issues. We present a simple, efficient, and intuitive method that provides a reasonable error metric. It works reasonably well in practice but we make no claims of perceptual correctness.

We de-couple the handling of attributes from the simplification of geometry. An exception to this rule is when attribute discontinuities exist. If either a face or vertex attribute discontinuity occurs at an edge between two faces, additional constraint planes are added to the error quadrics of the vertices that lie on the edge. To handle vertex attributes such as colors and texture coordinates, we add an *adjacency bounding volume* and *attribute bounding volumes* to each vertex. An adjacency bounding volume bounds the position of the vertex plus all of its adjacent vertices (see Figure 2(a)). Attribute bounding volumes are specific to the type of attribute being handled. For example, (r, g, b) colors are bounded by bounding spheres in (r, g, b) space (see Figure 2(b)). Alpha values are bounded by a one-dimensional interval. Texture coordinates are bounded by circles in (u, v) space and are assumed to be in the range [0, 1) for purposes of simplicity and finiteness. Texture coordinates are forced into this range by performing a modulo 1 operation. Initially, the attribute bounding volumes for each vertex represent a single point in the corresponding attribute space. A weighted average of the attributes surrounding each vertex determines these initial points, or average attributes.

When two vertices collapse, not only do their error quadrics combine and attributes get interpolated, but their adjacency bounding volumes and attribute bounding volumes merge as well. For two bounding volumes, a combine operation implies finding the tightest fitting bounding volume that encloses the pair. In addition, the new vertex position is combined with the two adjacency bounding volumes since the new vertex is not guaranteed to be placed on the line between the two vertices being collapsed. To be useful for our applications, the error at a vertex due to an attribute is transformed into an equivalent distance error in model space. Then comparisons between different types of errors such as geometric, color, and texture coordinates are possible. The distance error due to an attribute is the ratio of the size of the attribute bounding volume to the maximum possible size of the attribute volume, multiplied by the radius of the adjacency bounding volume (see Figure 2). Assuming that attributes at vertices are interpolated, the intuition behind this scheme is that changing the attribute at a vertex not only changes the appearance of the vertex, but also the appearance of its adjacent faces as well. The adjacency bounding volume bounds the locations of all adjacent faces that might be affected by the collapse, while the attribute bounding volumes determine the degree of attribute error across the region. The error of a vertex is the sum of its geometric, color, and texture coordinates errors. The algorithm approximates the geometric error of a vertex by taking the square root of the error computed by the quadric error bound [11]. Normals are considered to be a consequence of the geometry and are handled adequately by the quadric error metric. Bump-mapping coordinates could be handled in the same fashion as texture coordinates.

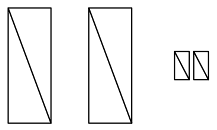


Figure 3: *What is the correct distance threshold for this model? If we use the distance between the large rectangles, then we will consider too many collapse candidates. If we use the distance between the small rectangles, then we will never consider virtual edges between the large rectangles.*

4.2 Automatic Selection of Distance Threshold

The algorithm proposed by Garland and Heckbert [11] uses a user-specified distance threshold, *t*. Determining a good *t* value for a particular model is not a trivial problem and for some models, no single *t* value is appropriate (see Figure 3). Instead of fixing *t*, we initially set it to a suitably small distance and then adaptively grow *t* during simplification. To assign the initial *t*, we assume a uniform distribution of the vertices within the model's bounding volume. Assuming the bounding volume is a cube of side length *s* and the number of vertices in the model is *v*, then $t = s / (v^{1/3})$. We validate *t* by checking if it produces too many virtual edge candidates from any one vertex. If it does, *t* is halved repeatedly until it becomes valid.

We change the quadric error metric algorithm [11] slightly to incorporate the dynamic *t*. Instead of one heap of vertex collapse candidates, we use two. The *local heap* contains any vertex collapse candidate such that the distance between the two vertices is less than *t*. The *edge heap* contains any vertex collapse candidate such that there exists an edge between the two vertices and the distance between the pair is greater than or equal to *t*. For each iteration, the algorithm selects the minimum cost vertex collapse by comparing the collapses on top of both heaps, ties favoring the local heap. The collapse is removed from the heap and it is adjusted accordingly. Whenever the local heap becomes empty, we double *t* repeatedly until the local heap becomes non-empty. The algorithm stops when both heaps become empty, a vertex collapse introduces too much error, or a target number of faces has been achieved.

To efficiently find pairs of vertices within the distance threshold *t*, we must partition space to avoid $O(n^2)$ behavior. Given the bounding volume of the model, we partition it uniformly into $t \times t \times t$ cells. When *t* is small compared to the bounding volume, the number of cells is too large to keep in memory. Thus, we use a hash table that is of size greater than or equal to the number of vertices in the model. To hash a vertex, we find its cell coordinates and hash these integers using an appropriate hashing function. When *t* changes size, we create a new hash table and re-insert the uncollapsed vertices. When two vertices collapse, they are removed from the hash table and the new collapsed vertex is inserted. To find all vertices within distance *t* of a particular vertex, we check the hash table entries for the vertex's cell, plus all neighboring cells, to see if any of the vertex entries meet the distance requirement.

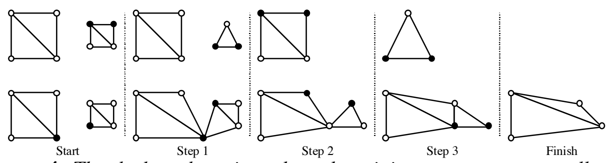


Figure 4: *The darkened vertices show the minimum error vertex collapse. On the top, simplification without surface area preservation. Both squares eventually collapse to nothing. On the bottom, simplification using surface area preservation with p = 75%. The squares merge together resulting in a better-looking drastic simplification.*

4.3 Surface Area Preservation

In Section 3, we motivate the need to merge unconnected regions of a model for our applications. To promote merging, we introduce the notion of *surface area preservation*. Vertex collapse operations that decrease the local surface area by more than a user specified percentage *p* are not considered as candidates. For all the examples shown in this paper, *p* is 75%. Surface area preservation prevents sections of the model from completely disappearing before merging can take place. This simple restriction results in drastic simplifications of polygonal objects that retain important features of the original model.

The computation of surface area preservation is fairly simple. We sum the areas of each face adjacent to at least one of the two vertices considered for collapse. This sum is the *pre-merge surface area*, labeled a^{-}. We collapse the vertices and find the location of the new vertex as well as its remaining adjacent faces. We sum the areas of each of these faces to calculate the *post-merge surface area*, labeled a^{+}. If $a^{+} / a^{-} < p$, then the collapse is not allowed. Surface area preservation bounds the reduction of surface area, but does not bound its growth. As described in Section 3.2, we cannot allow objects to vanish during simplification. Figure 4 shows an example of surface area preservation. The first step of our algorithm joins the two unconnected squares since any collapse within the squares would eliminate at least 50% of the local surface area.

5 Simplifying Static Environments

In the previous section, we presented an algorithm for topological simplification of polygonal objects. In this section, we use that algorithm to simplify large static polygonal environments represented by scene graphs. We use *HLOD* for drastic simplification, *partition* spatially large objects, and use *associations* for nodes containing more than two children.

5.1 Hierarchical Levels of Details (HLOD)

In addition to the LOD representing a node's model, we add HLOD to each node. HLOD represent the geometry of the entire scene graph rooted at the node. To create HLOD at a node, we merge geometry from the node and the node's children. We combine the lowest LOD of the node and the lowest HLOD of the children. If a child node is a leaf node, its HLOD is equivalent to its lowest LOD. Note that HLOD of the children must already exist for this merging operation to be valid. If they do not, we recursively compute the HLOD for the children. Furthermore, we transform the lowest children HLOD into the coordinate system of the node using link transformations. Since we combine the lowest LOD and lowest children HLOD, the highest HLOD for the node is normally a fairly coarse model. Our algorithm generates a series of approximations of this model to produce HLOD for the node.

5.2 Using HLOD for Simplification Culling

With a traditional scene graph, rendering a node involves determining which LOD to use for the geometry contained in the node. Each LOD has an associated maximum distance error. This error is projected onto the view plane to determine the screen-space error of the LOD. If this screen-space error is less than a user specified error tolerance, the LOD can represent the geometry at the node. The lowest LOD that passes the screen-space error criteria is rendered. The traversal of the scene graph continues recursively, determining the LOD to represent the node's children. Adding HLOD to each node changes the traversal of the scene graph. When the traversal reaches a node, it first determines whether an HLOD can be rendered. Like LOD, each HLOD has an associated maximum distance error. If an HLOD's screen-space error is less than the current error tolerance, then it can be rendered. The algorithm renders the lowest HLOD that meets the screen-space error criteria and does not traverse any of the node's child links. As a result, we perform *simplification culling* since the scene graph rooted at the node is culled away by substituting a simpler representation for it. If no HLOD meets the error tolerance, we select an LOD to represent the node and then recursively traverse each of its child links.

5.3 Partitioning Spatially Large Objects

Since we use static LOD, spatially large objects pose a problem. When the viewer is close to any region of a spatially large object, we must render the entire object in high detail, even though portions of it might be very far from the viewer. To alleviate this problem, we partition the model. We simplify each partition while guaranteeing that we do not produce cracks between partitions. Finally, we use HLOD with *partition restrictions* to combine the partitions hierarchically.

We partition large objects such that each partition is less than some percentage of the bounding volume of the entire scene graph. (see Figure 5(a)-(b)). We lay a uniform three-dimensional grid over the object and determine which grid each vertex and each face centroid falls in. Any face whose centroid lies in a grid

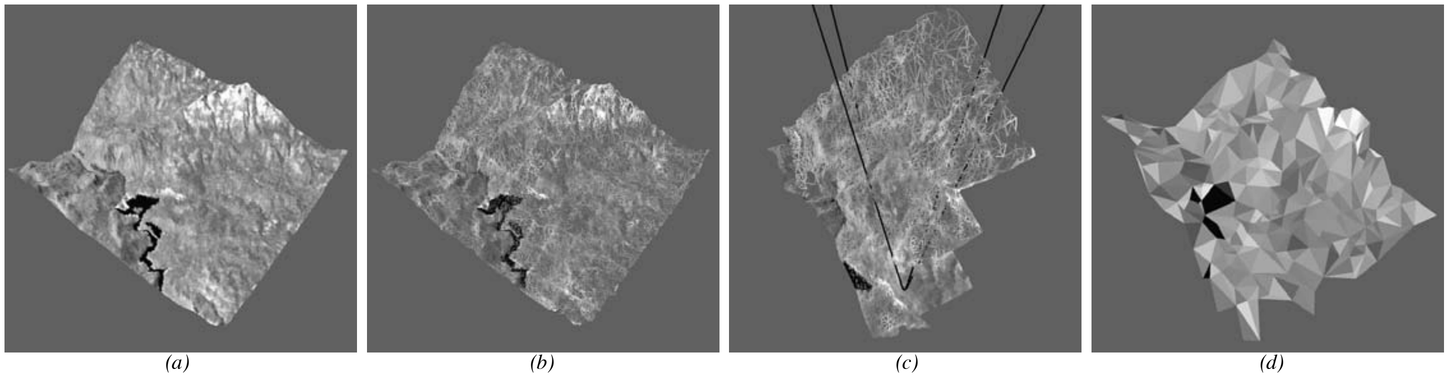


Figure 5: *Partitioning of a spatially large terrain model. (a) Original model (162,290 triangles). (b) Wireframe showing partitions. The boundaries of the partitions cannot be simplified while the interiors can. (32,332 triangles). (c) View-frustum culling and adaptive simplification due to partitioning. The partitions farthest away from the viewpoint merge together to relax their restriction boundaries. (13,668 triangles). (d) Drastic simplification on a partitioned model. No restrictions remain. (440 triangles).*

becomes a part of that grid's model, as well as the vertices making up that face. Any vertex that lies in a grid becomes a part of that grid's model as well, assuming it is not already a member. After this classification, we eliminate any grid that has no vertices and faces. The grids left are the partitions of the model, and they are assigned unique identifiers. For each vertex, we compare its partition to the partitions of its adjacent vertices. If an adjacent vertex is in a different partition, we union this partition's identifier with a set of *partition restrictions* for the vertex. Initially, the set of partition restrictions for each vertex is empty.

5.3.1 Crack Prevention

When we simplify an individual partition, we never move a vertex that has a non-empty set of partition restrictions. The vertex can be involved in a collapse operation, but the new vertex must be positioned exactly at the location of the restricted vertex. This way, we guarantee that we do not create cracks between partitions due to simplification.

5.3.2 Scene Graph Generation

In order to produce drastic simplifications for the entire object, we merge partitions together, thereby relaxing the partition restrictions at vertices. We produce a scene graph for the object that has the partitions at its leaf nodes. Each node in this scene graph has a list of partition identifiers that it represents. Initially, each leaf node's list consists of its own partition identifier. To maximize view-frustum culling of this scene graph, we try to minimize the size of the bounding volumes of each node of the graph. Using an adaptive spatial partitioning similar to the one described in Section 4.2, we determine which two nodes combine to form a minimally sized bounding volume.

By grouping nodes together, we relax the partition restrictions on the vertices. The partition identifier list for the new node is the union of its children's identifiers. Before HLOD creation, we discard partition restrictions of vertices that are members of this list. We create HLOD at the new node until the error of simplification becomes greater than some percentage of the node's bounding volume. We repeatedly group nodes until the scene graph has one root node. At this point, no vertex has a partition restriction, so the object can be drastically simplified.

One can view this partitioning process as a *discrete approximation* to view-dependent simplification. Since each leaf node is simplified independently of other partitions, partitions far away from the viewer will be rendered in low detail while partitions near the viewer will be rendered in high detail. When several partitions in close proximity are very far away from the viewer, they are rendered together using HLOD. Partitioning also allows us to view-frustum cull parts of the object (see Figure 5(c)). Finally, partitioning does not limit the amount of simplification possible since there are no partition restrictions at the root node of the scene graph (see Figure 5(d)).

5.4 Node Association

If a node has more than two children, then we *associate* them together hierarchically until the node has exactly two children links (see Figure 6). This association process is equivalent to the method shown in Section 5.3 for creating the scene graph of a partitioned object. We combine nodes while trying to minimize the bounding volume size for each new node in this *association graph*. When two nodes remain, they are linked to the parent node. In this way, we aid view-frustum culling by grouping together objects in close proximity first. We also aid simplification culling since grouping nearby objects produces nodes that contain HLOD with less approximation error. This approach makes HLOD computation feasible for nodes with

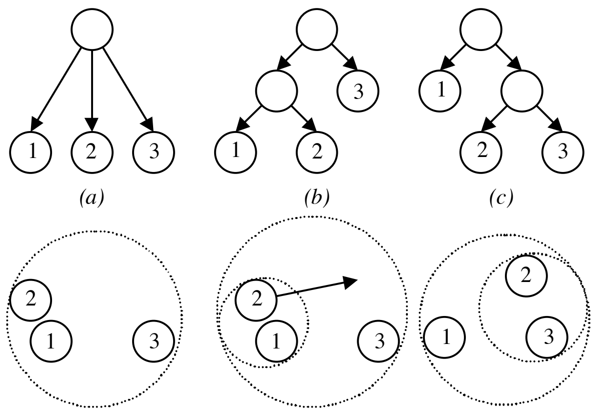


Figure 6: *Node re-association. Bounding volumes for unnumbered internal nodes are dashed circles. (a) A simple scene graph with no association. (b) The same scene graph, with an initial association. (c) Node 2 moves causing node re-association.*

large numbers of children, since it bounds the number of children of any node in the association graph by two.

6 Simplifying Dynamic Environments

In this section, we present an algorithm for simplifying dynamic environments by re-computing HLOD and re-associating nodes in the scene graph. It uses the object simplification algorithm presented in Section 4 and the HLOD computation algorithm in Section 5.

6.1 Incremental Computation of HLOD

Suppose a node in the scene moves because a link's transformation changes. First, we traverse up the scene graph, updating the bounding volumes of each node according to the movement. If the traversal does not change a bounding volume at a node, it stops. Once we update the bounding volumes, view-frustum culling will work properly. Next, we consider the HLOD at the node's parent. We conservatively calculate the maximum distance that any vertex of the parent node's HLOD could have moved due to the new transformation. We add this distance to the maximum error distance associated with each HLOD in the parent node and mark the HLOD as modified. We traverse up the tree, and add the transformed error to each HLOD, marking them as modified. Therefore, as a node moves, the error associated with any HLOD representing that node will increase according to the degree of motion. After the scene graph changes, HLOD created for a particular scene graph can still be rendered without modification, but their error distances will be larger and they will be rendered at greater distances from the viewer. New, more efficient HLOD are computed only when the scene graph traversal reaches a modified HLOD node and does not choose an HLOD to render. In this way, we *lazily* update the HLOD.

Insertion and deletion of links and nodes are more drastic operations. In these cases, we update bounding volumes and recompute all HLOD in the affected portion of the scene graph.

6.2 Node Re-association

When a node moves, the association graph of its scene graph parent will change. We must *re-associate* the nodes in this graph (see Figure 6). To update associations efficiently, we first remove the node from its association graph and do an upward traversal to update the bounding volumes of the graph as in Section 6.1. We then perform a downward search, starting from its root node of the association graph. If the moved object is not close enough to the node's bounding volume to change any associations in the descendants of the node, then we stop at that node. Otherwise, we traverse down the link to the child that when merged with, would increase the bounding volume of the child the least. If the traversal encounters a leaf node, we stop. Next, we start an upward traversal. If we merge the removed node's bounding volume with the current node's bounding volume and it is smaller than the parent's bounding volume, then we splice the removed node into the association graph. If the bounding volume of the node's scene graph parent changes, we update the bounding volumes of the scene graph as in Section 6.1. As these bounding volumes change, they can trigger re-associations further up the scene graph.

7 Implementation and Performance

We have implemented the algorithm using C++, GLUT, and OpenGL. The code is portable across PC and SGI platforms. We observe significant speedups on a large static and dynamic model due to our drastic simplification algorithm and the use of simplification culling.

7.1 Implementation

Currently, all three-dimensional bounding volumes are bounding spheres for purposes of simplicity. The use of better bounding volumes might increase the performance of the algorithm due to view-frustum culling and better attribute simplification. For example, several partitions that seemingly lie outside the view-frustum are not culled in Figure 5(c).

Our current implementation for dynamic environments is sequential. Since simplifying geometry is typically slow compared to rendering, we amortize the work over several frames by queueing nodes that require HLOD recomputation. Every few frames, we dequeue a node and recompute its HLOD. Since the highest HLOD should be coarse to begin with, this process normally takes a fraction of a second. This delay is noticeable and lowers the frame-rate considerably. We are currently working on a parallel implementation for multi-processor machines that spawns three processes; a rendering process, a view-frustum culling process, and a simplification process that *asynchronously* updates HLOD. The rendering process will use HLOD if they are available, otherwise no simplification culling is performed. We expect this parallel version to eliminate frame rate degradation due to HLOD recomputation.

7.2 Generality and Robustness

Our system handles all polygonal models whether they are manifold meshes or unorganized list of polygons. As a pre-process, models are triangulated and then cleaned using vertex sharing and recalculation of normals if desired by the user. Although these operations are not necessary for the simplification algorithm, the more topology information it has, the better it can simplify.

Model	Triangles	Objects	Time
Airheaters	800	1	2 seconds
Chamber	10,423	1	20 seconds
Pipes	23,556	1	40 seconds
Terrain	162,690	1	10.5 minutes
Torpedo Room	883,537	356	33 minutes

Table 1: *Simplification timings for various models running on an SGI Infinite Reality with 250 MHz R4400 processors and 2 gigabytes of main memory.*

Faces consist of face attributes and pointers to vertices and vertex attributes at the corners of the face. Vertices contain a list of pointers to adjacent faces. We assume no ordering on these adjacent faces. Almost any polygonal model can be described with this representation. In practice, we have applied our system to large CAD models composed of non-manifold and degenerate geometry.

7.3 Results

All of our timing results were generated on an SGI Infinite Reality with 250 MHz R4400 processors and 2 gigabytes of main memory. A maximum of 512 megabytes was used to produce these results.

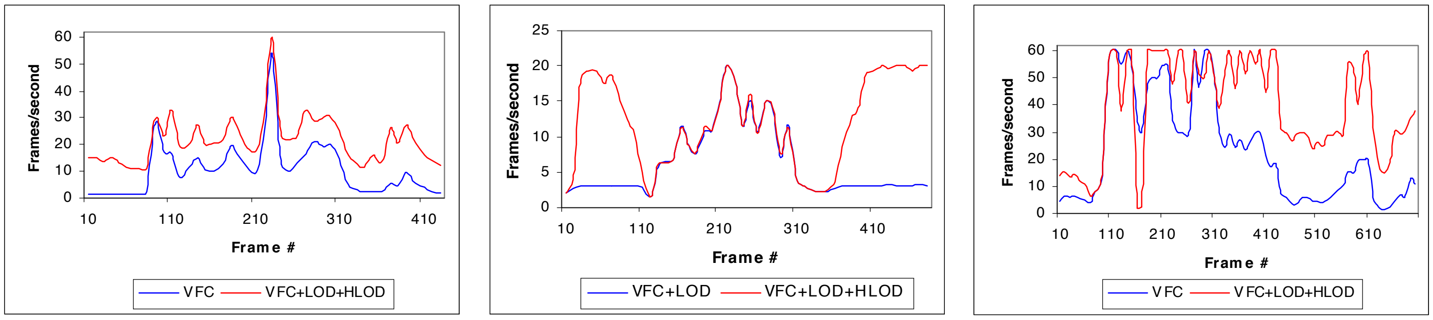
Table 1 shows timings for our simplification pre-process algorithm. The airheaters, pipes, and torpedo room model (see Plate 2) are CAD models. The chamber is a radiosity model used to test our attribute handling (see Plate 1). The terrain model was used to demonstrate partitioning (see Figure 5). Our current implementation has not been optimized and is about an order of magnitude slower than the algorithm presented by [11]. However, it handles attributes, is efficient, simple, general, automatic, and produces high quality and drastic approximations. If we eliminate attribute handling, our algorithm runs 2 to 3 times faster.

7.4 Attribute Simplification

Our attribute simplification is intuitively simple, easy to implement, and produces reasonable results. Shown in Plate 1 are the results on the chamber model. In (c), no attribute handling results in co-planar regions collapsing together without regard for color error. In (b), attribute handling does a better job of preserving the shadows and lighted areas in the scene.

7.5 Static Environments

We demonstrate our algorithm on the torpedo room model (see Figure 8), a large static environment consisting of 883,537 triangles and 356 objects (see Plate 2). Without LOD and HLOD, we cannot view it interactively from all viewpoints on



(a) (b) (c)

Figure 7: *Timing results for different static and dynamic scenes. (a) Walkthrough of a static torpedo room consisting of 883,537 triangles and 356 objects. The graph compares view-frustum culling with view-frustum culling, LOD, and simplification culling using HLOD. (b) Walkthrough of a static environment consisting of 27 torpedo rooms (obtained by instancing) consisting of 27 ×883,537 = 23,855,499 triangles and 27 ×356 = 9,612 objects. The graph compares view-frustum culling and LOD with view-frustum culling, LOD, and simplification culling using HLOD. (c) Walkthrough of the torpedo room while it is exploding. The graph compares view-frustum culling with view-frustum culling, LOD, and simplification culling using HLOD.*

our SGI Infinite Reality. Using LOD and simplification culling aided by HLOD, we are able to view this model at over 10 frames a second with little loss in image quality. In Figure 7(a), we show our results as compared to just using view-frustum culling. On average, we achieve approximately a 1.8 times speedup. Note that when the torpedo room is far away, our system runs at over 10 frames a second while just using view-frustum culling runs at 1 frame a second. This verifies that LOD and simplification culling are most useful when objects are far away from the viewer.

In Figure 7(b), we show our results on a static environment composed of 27 instanced torpedo rooms, consisting of 27 □ 883,537 = 23,855,499 triangles and 27 x 356 = 9,612 objects. Each LOD for each part of the torpedo room ranges from its original model down to a model with approximately 1% of the original geometry. To determine the impact of HLOD, we compare our algorithm using simplification culling to one using regular LOD. On average, we achieve approximately a 1.9 times speedup due to simplification culling. When the scene of torpedo rooms is far off in the distance, HLOD and simplification culling increase the frame rate significantly. As we get close to the scene of torpedo rooms, there is little benefit to using HLOD. This verifies that HLOD and simplification culling are most useful when objects are very far away from the viewer or when there is a target frame-rate. Using simplification culling and HLOD, we can achieve drastic simplification for any static environment.

7.6 Partitioning Algorithm

In Figure 5, we show our results of partitioning a spatially large terrain model. Partitioning the terrain took 2 minutes of pre-processing time. The difference in rendering performance between the terrain without partitioning and the terrain with partitioning is not noticeable until the viewer is close to a portion of the model. Only then do the benefits of view-frustum culling and adaptive simplification become apparent. When the viewer is close to the terrain model, we routinely achieve a 10 times speedup, mostly due to view-frustum culling.

7.7 Dynamic Environments

We demonstrate the dynamic component of our algorithm on the torpedo room as well. Currently, we consider two dynamic scenarios on the model. The first scenario is a design scenario where a user can move parts of the torpedo room around. The second scenario shows an extremely dynamic environment, namely the torpedo room exploding (see Plate 3). In both scenarios, objects move and then come to a rest. In Figure 7(c), we show the results of our algorithm on the exploding torpedo room as compared to just using view-frustum culling. On average, we achieve approximately a 1.9 times speedup. Again, the most speedup occurs when the viewer is outside the perimeter of the torpedo room.

After the initial explosion, the parts of the torpedo room come to a rest. At this point, our algorithm determines that HLOD recomputation is necessary. In the graph, note the dip in frame-rate between frames 110 and 210. This slowdown shows when our algorithm recomputes HLOD for the nodes representing the 356 exploded objects of the torpedo room. This drop in frame-rate is significant and noticeable and is due to our current serial implementation. The time to traverse up and down the scene graph to recompute the bounding volume hierarchy and re-associate objects is insignificant compared to the time required for HLOD recomputation.

8 Analysis and Comparison

As we use an edge heap, our simplification algorithm has a bound on running time of O(e lg e) where e is the number of edges. Assume hashing takes O(1) time, the number of vertices within distance t of each vertex is a constant c as vertices collapse and t doubles, and the bounding cube of the model has side length s. Because of the use of the local heap, the algorithm has a bound on running time of $O(\lg_t s\ (cv \lg cv)) = O(v \lg v)$ where v is the number of vertices. Thus, the expected running time of the algorithm is O(e lg e) + O(v lg v).

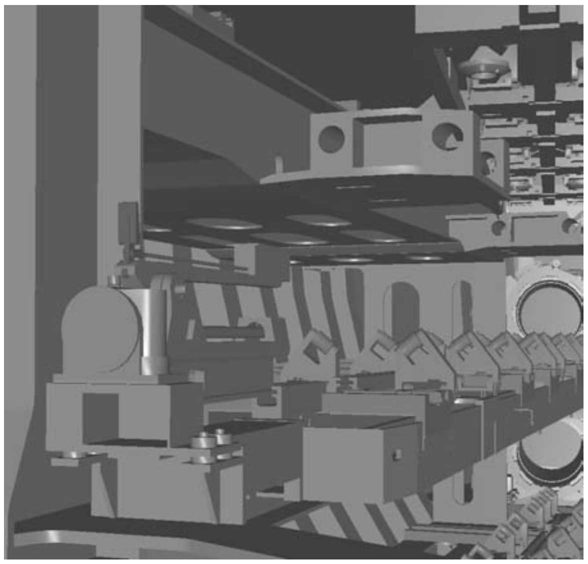


Figure 8: *A close up view of the torpedo room used for our static and dynamic scenarios. It consists of 883,537 triangles and 356 objects.*

For static environments, the performance increase due to HLOD and simplification culling depends on the degree to which parts of the model are simplified. If the lowest LOD of each object is a single polygon, then there is very little benefit to creating HLOD unless there are millions of objects in the scene. Also, having extremely low polygon count LOD limits the quality of simplification that HLOD can provide since there is not much geometry to work with. On the other hand, we do not want the coarsest LOD to have too many polygons since HLOD creation would require the duplication of too much geometry and would use up too much memory. A balance must be found between these two options.

For dynamic environments, every time an object moves some work must be performed. Updating bounding volumes requires a traversal up the height of the entire scene graph in the worst case. Re-associating nodes involves two traversals, both the height of the association graph at each node in the worst case. Therefore, performance depends a great deal on the depth of the scene graph and the number of objects that are moving. Scenes where a majority of the objects are constantly moving will not benefit much from simplification culling. There is a possibility that groups of objects far away from the viewer could be grouped into HLOD, but the movement of these objects would eventually make the HLOD invalid and in need of recomputation. Fortunately, most objects in real-world scenarios do not move incessantly.

8.1 Comparison with other Topological Simplification Algorithms

There are many simplification algorithms, and in this section we compare ones that lie in the domain of topological simplification for interactive rendering. Garland and Heckbert propose a general and robust algorithm based on error quadrics [11]. It produces very impressive approximations of high quality. To meet the needs of our driving application, we extended this algorithm to handle attributes and to automatically promote merging between unconnected regions of the model. Schroeder [33] presents another general and robust algorithm capable of producing guaranteed levels of reduction. However, it does not merge objects to the degree we require for HLOD computation. Popovic and Hoppe [27] present an algorithm to produce a progressive simplicial complex. This algorithm is elegant, but is too slow for HLOD recomputation and uses primitives other than polygons. Rossignac and Borrel [30] and Low and Tan [24] use vertex clustering to topologically simplify models very efficiently and robustly. Although these algorithms are very quick, the quality of their approximations is too low for our purposes. El-Sana and Varshney [10] control the genus of an object while simplifying, but does not merge disjoint objects.

Not many simplification algorithms have focussed on environments of objects except Luebke and Erikson's [25] hierarchical dynamic simplification. The algorithm builds a hierarchical tree of vertices and uses a view-dependent system during rendering. It is general, robust, and can drastically simplify every object in a static environment to a single polygon. It is capable of simplifying any static environment to a single polygon if the scene graph's hierarchy is flattened. In order to handle dynamic environments efficiently, we cannot flatten the scene graph. Also, for reasons explained in Section 3.1 and 5.3, we use view-independent rendering and approximate view-dependent techniques.

9 Summary and Future Work

We present an approach for simplifying large static and dynamic polygonal environments. It includes a new topological simplification algorithm that is general, efficient, produces high quality approximations, and merges unconnected regions of objects using an adaptive distance threshold and surface area preservation. This algorithm is very good at creating HLOD for large environments to aid simplification culling. Furthermore, we describe how HLOD can be recomputed for dynamic environments.

There are many avenues for future work. Our attribute handling could be improved and extended. Our simplification algorithm needs to be optimized and used for a system targeting a constant frame-rate. We would like to continue investigating the tradeoffs between different partition sizes and different polygon counts for the lowest LOD for objects. Most importantly, we wish to eliminate the frame rate degradation due to HLOD recomputation for dynamic environments by parallelizing our algorithm. Our ultimate goal is to demonstrate the algorithm on a truly massive dynamic environment, one consisting of millions of polygons.

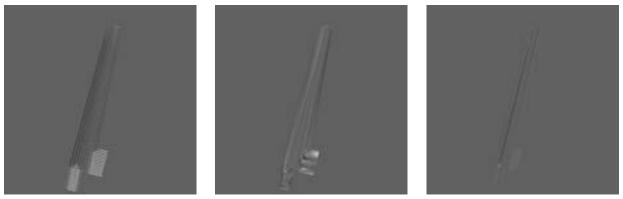


Figure 9: *Comparison of our topological simplification algorithm with QSlim, Michael Garland's publicly available simplification code. (a) Complicated pipes model consisting of 23,556 triangles. (b) Using surface area preservation and an adaptive distance threshold, our algorithm retains the basic features of the pipes after drastic simplification (683 triangles). (c) For QSlim, we set the distance threshold to a tenth of the bounding sphere radius of the pipes but its error metric prevents it from merging many pipes. As a result, the pipes appear to thin (683 triangles).*

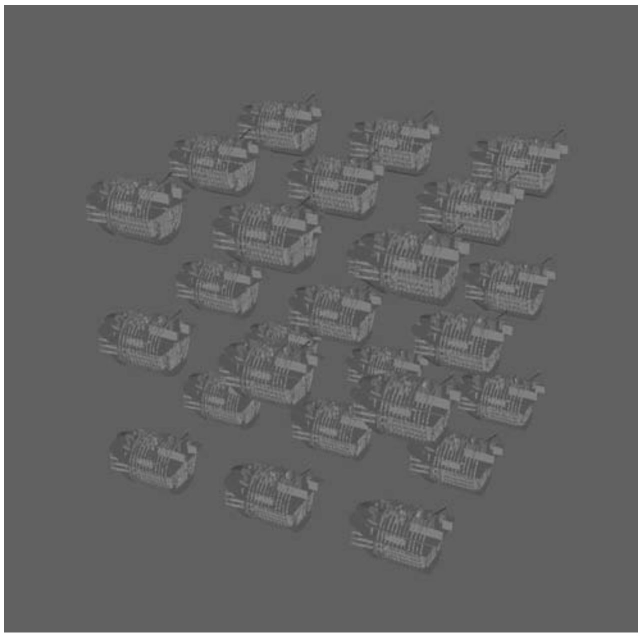


Figure 10: *Far away view of a static environment composed of 27 instanced torpedo rooms (27 □ 883,537 = 23,855,499 triangles, 27 □ 356 = 9,612 objects).*

10 References

[1] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Proc. of ACM Symposium on Interactive 3D Graphics*, pp. 41--50, 1990.

[2] D. G. Aliaga. Visualization of Complex Models using Dynamic Texture-based Simplification. *Proc. of IEEE Visualization'96*, pp. 101--106, 1996.

[3] M. J. Atallah. Dynamic computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, 1983, pp. 92--99.

[4] C. Bajaj and D. Schikore. Error-bounded reduction of triangle meshes with multivariate data. *SPIE*, vol. 2656, pp. 34--45, 1996.

[5] J. Cohen, D. Manocha, and M. Olano. Simplifying Polygonal Models Using Successive Mappings. *Proc. of IEEE Visualization'97*, pp. 395-402, 1997.

[6] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification Envelopes. In *SIGGRAPH'96 Conference Proceedings*, 1996, pp. 119--128.

[7] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. *Proc. of 1997 Symposium on Interactive 3D Graphics*, pp. 83-90, 1997.

[8] G. Drettakis and F. Sillion. Interactive Update of Global Illumination Using a Line-Space Hierarchy. *SIGGRAPH'97 Conference Proceedings*, pp. 57-64, 1997.

[9] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution Analysis of Arbitrary Meshes. In *SIGGRAPH'95*, 1995, pp. 173--182.

[10] J. El-Sana and A. Varshney. Controlled Simplification of Genus for Polygonal Models. *Proc. of IEEE Visualization'97*, pp. 403-410, 1997.

[11] M. Garland and P. Heckbert. Surface Simplification using Quadric Error Bounds. *SIGGRAPH'97 Conference Proceedings*, pp. 209-216, 1997.

[12] C. M. Gold. Dynamic spatial data structures -- the Voronoi approach. In *Proc. Canad. Conf. GIS*. Ottawa, 1992, pp. 245--255.

[13] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, 1991, pp. 523--533.s

[14] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM Siggraph*, 1993, pp. 231--238.

[15] A. Gueziec. Surface Simplification with Variable Tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, November 1995, pp. 132--139.

[16] B. Hamann. A Data Reduction Scheme for Triangulated Surfaces. *Comput. Aided Geom. Design*, vol. 11, pp. 197--214, 1994.

[17] T. He, L. Hong, A. Varshney, and S. Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, pp. 171-814, 1996.

[18] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *SIGGRAPH'97 Conference Proceedings*, pp. 189-198, 1997.

[19] Hugues Hoppe. Progressive Meshes. *SIGGRAPH 96 Conference Proceedings*, pp. 99--108, August 1996.

[20] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, vol. 27, J. T. Kajiya, Ed., August 1993, pp. 19--26.

[21] W. Jepson, R. Liggett, and S. Friedman. An Environment for Real-time Urban Simulation. *1995 Symposium on Interactive 3D Graphics*, pp. 165--166, 1995.

[22] A. D. Kalvin and R. H. Taylor. Superfaces: Polyhedral Approximation with Bounded Error. In *Proceedings of SPIE Medical Imaging 1994*, vol. 2164, Y. Kim, Ed.: SPIE, February 1994.

[23] S. Kumar, D. Manocha, H. Zhang, and K. Hoff. Accelerated Walkthrough of Large Spline Models. *Proc. of ACM Symposium on Interactive 3D Graphics*, pp. 91--102, 1997.

[24] K. Low and T. Tan. Model Simplification using Vertex Clustering. *Proc. of 1997 Symposium on Interactive 3D Graphics*, pp. 75-82, 1997.

[25] D. Luebke and C. Erikson. View-Dependent Simplification Of Arbitrary Polygonal Environments. *SIGGRAPH'97 Conference Proceedings*, pp. 199-208, 1997.

[26] Paulo W. C. Maciel and Peter Shirley. Visual Navigation of Large Environments Using Textured Clusters. *Proc. of 1995 Symposium on Interactive 3D Graphics*, pp. 95--102, 1995.

[27] J. Popovic and H. Hoppe. Progressive Simplicial Complexes. *SIGGRAPH'97 Conference Proceedings*, pp. 217-224, 1997.

[28] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of Trimmed Surfaces. In *Proceedings of ACM Siggraph*, 1989, pp. 107--17.

[29] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, vol. 15, pp. 67--76, 462, August 1996.

[30] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering. In *Modeling in Computer Graphics*: Springer-Verlag, 1993, pp. 455--465.

[31] G. Schaufler and W. Sturzlinger. Generating Multiple Levels of Detail from Polygonal Geometry Models. *Virtual Environments'95 (Eurographics Workshop)*, pp. 33-41, 1995.

[32] B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. Brush as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, July 1994, pp. 389--399.

[33] W. Schroeder. A Topology Modifying Progressive Decimation Algorithm. *Proc. of IEEE Visualization'97*, pp. 205-212, 1997.

[34] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *Proc. of ACM Siggraph*, 1992, pp. 65--70.

[35] J. Shade, D. Lischinski, T. DeRose, J. Snyder, and D. Salesin. Hierarchical Image Caching for Accelerate Walkthroughs of Complex Environments. In *Proc. of ACM Siggraph*, New Orleans, 1996, .

[36] M. Soucy and D. Laurendeau. Multi-Resolution Surface Modeling Based on Hierarchical Triangulation. *Computer Vision and Image Understanding*, vol. 63, pp. 1-14, 1996.

[37] O. Sudarsky and C. Gotsman. Output sensitive visibility algorithms for dynamic scenes with applications to Virtual Reality. *Computer Graphics Forum*, vol. 15, pp. 249--58, 1996.

[38] S. Teller and C. H. Sequin. Visibility Preprocessing for interactive walkthroughs. In *Proc. of ACM Siggraph*, 1991, pp. 61--69.

[39] Enric Torres. Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes. In *Eurographics '90*, C. E. V. a. D. A. Duce, Ed.: North-Holland, September 1990, pp. 507--518.

[40] G. Turk. Re-Tiling Polygonal Surfaces. In *Proc. of ACM Siggraph*, 1992, pp. 55--64.

[41] J. Xia, J. El-Sana, and A. Varshney. Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pp. 171--183, 1997.

[42] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. *Proc. of ACM SIGGRAPH*, pp. 77-88, 1997.