HIERARCHICAL LEVELS OF DETAIL TO ACCELERATE THE RENDERING OF LARGE STATIC AND DYNAMIC POLYGONAL ENVIRONMENTS

by

Carl M. Erikson

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2000

Approved by

Advisor: Professor Dinesh Manocha

Reader: Professor Anselmo Lastra

Reader: Professor Ming Lin

ABSTRACT

Carl M. Erikson. Hierarchical Levels of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments

(Under the direction of Dinesh Manocha)

Interactive visualization of large three-dimensional polygonal datasets is an important topic in the field of computer graphics and scientific visualization. These datasets can be static, such as many walkthrough applications, or dynamic, such as CAD scenarios where a designer moves, adds, or deletes parts. In many cases, the number of primitives in these models overwhelms the rendering performance of current graphics systems. In order to view these environments in real-time, approximation techniques must augment the capabilities of the hardware.

One method for accelerating the rendering of these environments is polygonal simplification. This dissertation describes the creation and application of *levels of detail*, or *LODs*, and *hierarchical levels of detail*, or *HLODs*, to accelerate the rendering of large static and dynamic polygonal environments. The principal idea of this work is that simplification methods should not always treat objects, or collections of polygons, in a scene independently. Instead, they may be able to produce better and drastic approximations by simplifying disjoint regions of a scene together. This idea is applicable for creating LODs for individual objects that consist of unconnected polygons, and for constructing HLODs, or approximations representing groups of static or dynamic objects.

We demonstrate a polygonal simplification algorithm called *General and Automatic Polygonal Simplification*, or *GAPS*, that excels at merging disjoint polygons through the use of an *adaptive distance threshold* and *surface area preservation*. We use GAPS to create LODs and HLODs for nodes in the scene graphs of large polygonal environments. These approximations are used to enable two rendering modes, one that allows the user to specify pixels of error and another that targets a frame rate. When objects in the scene move, our

algorithm updates HLODs that have become inaccurate or invalid using asynchronous simplification processes. Our algorithm currently handles scenes with limited dynamic movement.

ACKNOWLEDGMENTS

Without pre-existing three-dimensional datasets to test our algorithms on, this thesis would have never come into existence. Model creation and management is a painstaking process, so we wish to thank all of the people and organizations who provided us with such excellent polygonal geometry. The Rotor object is courtesy of the Alpha_1 Project at the University of Utah. The Head model is courtesy of Hans Weber, my officemate, and was created using InSpeck Inc. software. The Chamber model was created by Mike Goslin and David Luebke, both UNC-CH alumni. The Econ and ShDivWest objects and the massive Power Plant environment that they came from are courtesy of James Close and Combustion Engineering, Inc. The scanned Bunny object is courtesy of the Stanford 3D Scanning Repository (http://graphics.stanford.edu/data/3Dscanrep/) created by the Stanford University Computer Graphics Laboratory. The Sierra terrain model is courtesy of Herman Towles and Sun Microsystems. Viewpoint and Division, Inc. allowed us to use the Ford Bronco dataset. The Cassini spacecraft model is courtesy of Stephen Wall, Gary Clough, Don Jacob, and JPL. The Torpedo Room model is courtesy of Greg Angelini, Jim Boudreaux, Ken Fast, and Electric Boat, a subsidiary of General Dynamics.

We would like to thank the groups that have provided either software, hardware, or free exchange of ideas necessary to conduct research. The UNC-CH computer science department offers vast resources with excellent support staff to every student eager to take advantage of it. The UNC Walkthrough group challenges us with massive environments to render at interactive rates as well as providing a melting pot of research ideas. The now-defunct UNC Simplification group promoted very useful discussion on avenues for potential research in the field of polygonal simplification. Finally, the graphics groups at UNC provide an incredible research environment, complete with an extraordinary graphics lab.

We have received financial support from a variety of sources for which we are grateful. Our work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, Honda, Intel Corp., NIH, National Center for Research Resources Award 2P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy, NSF grant CCR-9319957 and Career Award, an ONR Young Investigator Award, the NSF/ARPA Center for Computer Graphics and Scientific Visualization, and NCAA Graduate, NSF Graduate, and Intel Fellowships.

I would like to thank my dissertation committee – Fred Brooks, Anselmo Lastra, Ming Lin, and Mary Whitton – for guiding me on this journey and my advisor, Dinesh Manocha, for motivating me to pursue excellence relentlessly in this work.

I would especially like to thank my friends, housemates, officemates, teammates, and fellow students who have made this experience enjoyable. Gentaro Hirota helped me in my earlier years as a graduate student, especially in COMP205. Kenny Hoff inspired and entertained me with his energetic ideas. Jon McAllister showed me what North Carolina barbeque is all about. Thanks to Marco Jacobs for offering me a place to stay in the Netherlands. Hans Weber and Mark Mine were always eager to help me with any problems I encountered. Catherine Moga showed me what hard work really means by performing eight jobs simultaneously. Dave Luebke and Bill Mark convinced me to live with them, changing my life in unexpected and fortunate ways. Dave Luebke also introduced me to the sport of Ultimate and graciously allowed me to be part of his SIGGRAPH ’97 paper. The members of UNC Darkside gave me a sense of accomplishment rivaling this dissertation. My love and thanks to Sarah Danninger who amazes me with her ability to sacrifice and persevere in other countries in the world. Finally, I would like to thank my family for all of their support and generosity throughout this period of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Visualization of Large Polygonal Models

Interactive visualization of three-dimensional datasets is an important application and consequently an important research topic in the field of computer graphics and scientific visualization. Many applications, such as computer-aided design and walkthroughs, benefit from interactive navigation, i.e., being able to move around a model at greater than 10 frames per second. When users are able to move around the model interactively and view it from all directions, they are able to quickly gain a good understanding of the three-dimensional data.

In some applications, objects in the environment move. Such scenarios are common in design evaluation of large assemblies where a designer moves, adds, or deletes parts. Other cases of dynamic environments include simulation-based design, driving simulators, battlefield visualization, urban environment visualization, and entertainment software. All of these applications require that objects be capable of movement, either by programmed behavior or interactive manipulation. For example, Figure 1.1 shows a user interacting with a Ford Bronco model.

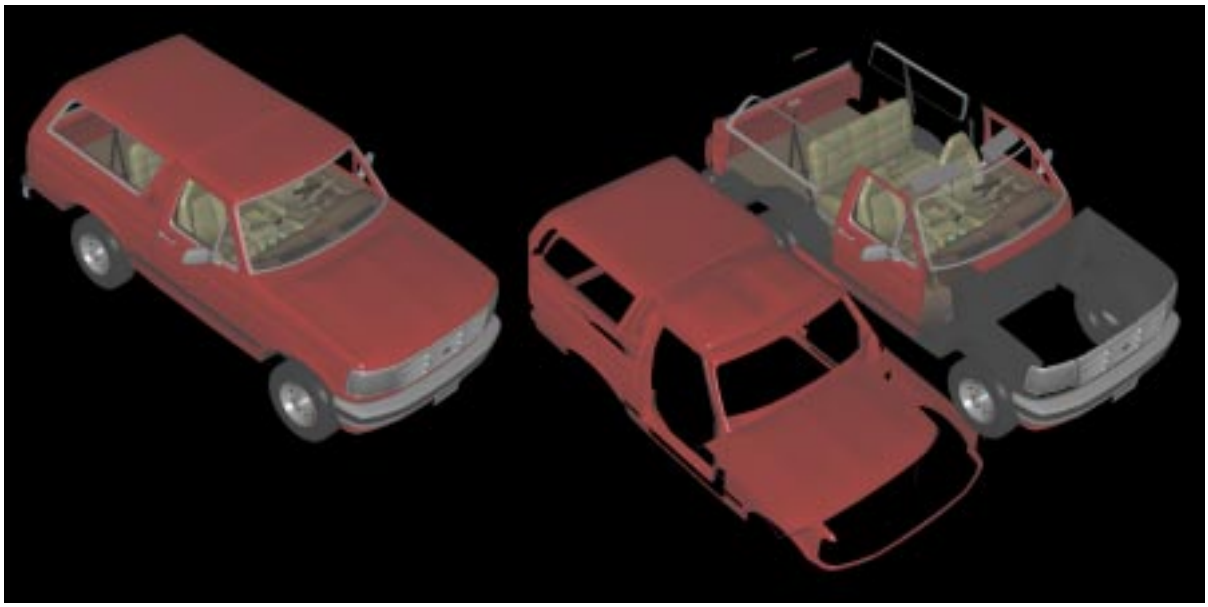


Figure 1.1: A user interacts with a Ford Bronco model consisting of 74,308 polygons. On the left is the original model. On the right, the user has removed the body of the Bronco to view its interior.

A common problem for visualization systems is that large three-dimensional datasets require a great deal of rendering power. Specialized graphics systems are commonly used to accelerate the process. However, models exist that cannot be rendered at interactive frame rates even with current high-end graphics machines. For example, Figure 1.2 shows a view of a complex power plant model that overwhelms an SGI Infinite Reality system [Montrym et al. 97]. For such models it is necessary to use algorithmic techniques to accelerate the rendering process. These methods attempt to render at interactive frame rates by either substituting simpler approximations for portions of the dataset or ignoring parts of it that are not visible. The goal is to improve interactivity without significant degradation in image fidelity.

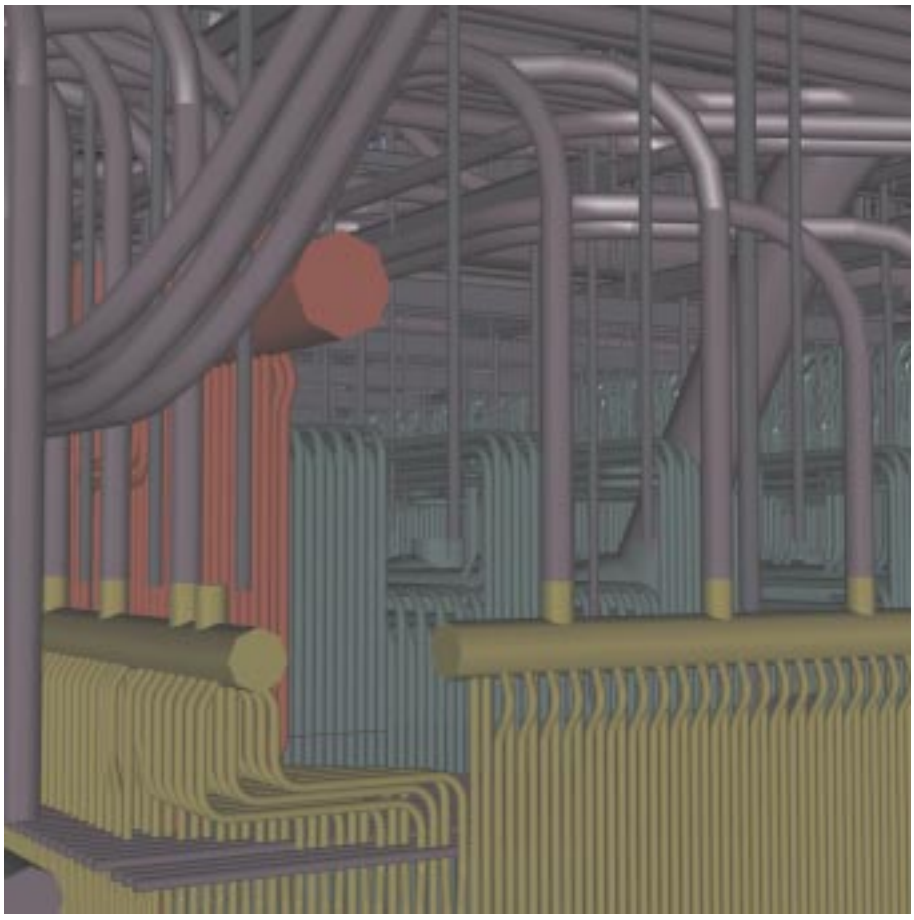


Figure 1.2: A view of the Power Plant model. This area is one of many that consists of a mass of pipes. The whole model consists of 12,731,154 polygons and cannot be rendered at interactive frame rates using an SGI Reality Monster with a 300 MHz R12000 processor, 16 GB of main memory, and an SGI Infinite Reality 2E graphics subsystem containing 4 GEs and 2 RM9s.

Three-dimensional datasets are usually represented using polygons because of their simplicity and generality. Polygons are mathematically simple and most graphics systems have been specifically designed to render them quickly. Other formats such as curved surfaces and volumetric data sets can be easily converted into polygonal representations to any desired accuracy. Polygons are the lowest common denominator of three-dimensional formats, as evidenced by their predominance in visualization applications. Therefore, most techniques for accelerating the rendering of three-dimensional datasets deal with polygonal models.

Visibility and simplification techniques are two commonly used methods for accelerating the rendering of large polygonal environments. Visibility techniques are designed to quickly cull away portions of a scene that are not visible to the viewer. Methods based on efficient

cell-to-cell visibility for architectural environments are widely used [Airey et al. 90, Teller and Séquin 91, Luebke and Georges 95]. More complicated algorithms have been developed to handle general polygonal environments [Greene et al. 93, Coorg and Teller 96, Hudson et al. 97, Zhang et al. 97]. Simplification methods are used to approximate regions of the model when rendering, usually sacrificing image quality for speed. Some simplification algorithms replace portions of the environment with images [Maciel and Shirley 95, Schaufler and Stuerzlinger 96, Shade et al. 96, Aliaga and Lastra 99] while others, using *polygonal simplification*, replace complex objects with lower polygon count approximations. This thesis introduces new techniques for the field of polygonal simplification.

The goal of polygonal simplification techniques is to generate *multi-resolution* representations [Clark 76, Heckbert and Garland 94] for polygonal objects. Based on the distance between an object and the viewer, a visualization application renders an appropriate representation for the object. If an object is far off in the distance and covers only a few pixels on the screen, a coarse representation will suffice; if the viewer is close to the object, a highly detailed version is probably necessary. Polygonal simplification is the problem of reducing the number of polygons needed to represent a three-dimensional polygonal object while retaining a good approximation of its original shape and appearance. Traditionally, simplification algorithms produce a series of representations, or *levels of detail*, for an object. Given the distance between the viewer and an object, the rendering system uses *switching distances* to determine which LOD to draw. Figure 1.3 shows LODs of a polygonal bunny model and Figure 1.4 demonstrates the distance at which some of the bunny LODs would be rendered.

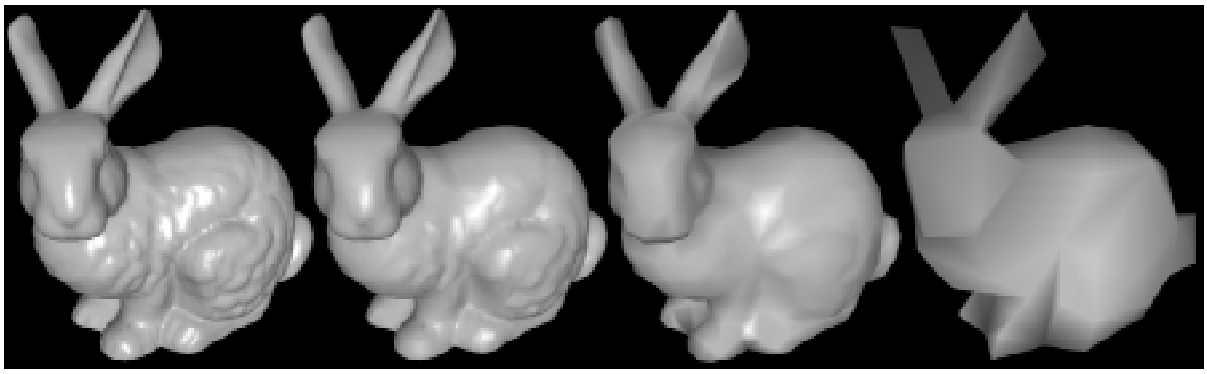


Figure 1.3: LODs of a bunny model. From left to right, these LODs consist of 69,451 polygons, 8,680 polygons, 1,085 polygons, and 135 polygons, respectively.

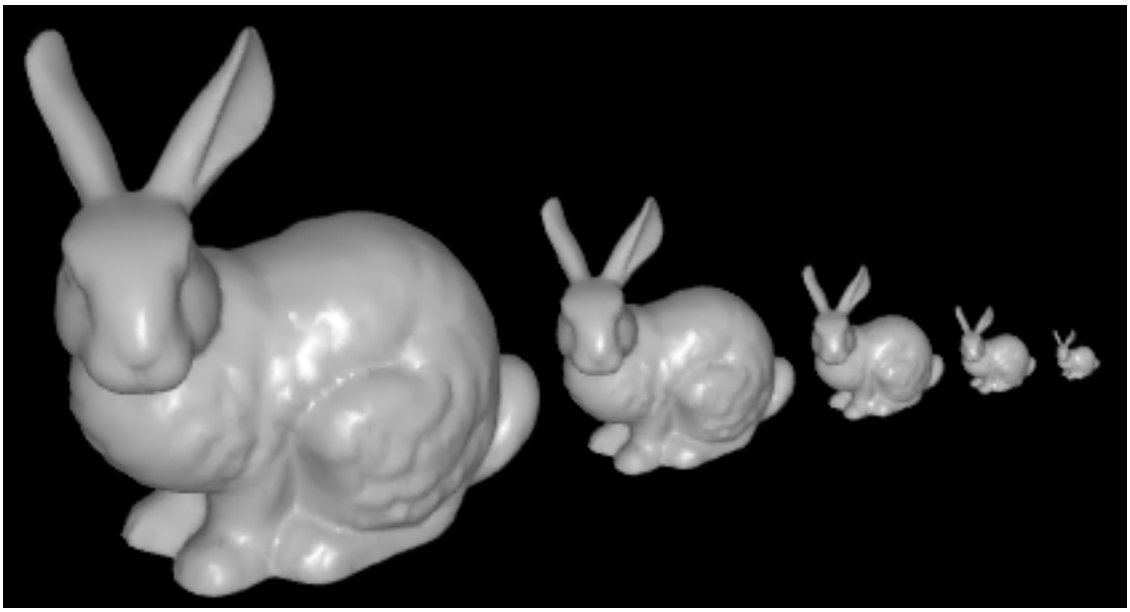


Figure 1.4: Switching distances for LODs of the Bunny object. From left to right these LODs consist of 17,361 faces, 8,680 faces, 4,340 faces, 2,169 faces, and 1,085 faces. The original object consists of 69,451 faces.

1.2 Definitions

Throughout this dissertation, we refer to *high quality* and *drastic* approximations. In this section, we define these terms.

1.2.1 High Quality

There is no standard definition of quality of approximation in the field of polygonal simplification. Usually, a metric that approximates error between the original and simplified object is used to guide a simplification algorithm. Simplification operations that introduce the least amount of error according to this metric are performed first. However, in the context of rendering, the output of a simplification algorithm is rarely judged by the error reported by a metric. Instead, the output of these algorithms is judged visually. If an approximation does not look good, then it is not good, regardless of the metric error. Therefore, the goal of polygonal simplification algorithms used for rendering is not to produce approximations that minimize an error metric, but rather to produce good-looking approximations.

Therefore, it is common that an error metric of a simplification algorithm is tweaked to enhance its visual output. For example, trying to preserve the silhouette or regions of high curvature of an object during simplification is a common technique used by most algorithms. These areas are deemed perceptually important and are preserved even at coarse approximations. In other words, simplification algorithms use heuristics, either along with or integrated with error metrics, in order to produce good-looking results. These heuristics are derived from empirical evidence.

Suppose that an error metric exists that could quantify the exact perceptual error between the original and simplified object. If this perfect error metric was known, it would define high quality. It would be used by simplification algorithms to produce the best possible approximations. Using imperfect error metrics and heuristics, simplification algorithms attempt to mimic this ideal metric. Since a perfect metric has not yet been discovered, there is currently no quantitative way to measure how close a simplified object is to the ideal approximation. However, we can view the simplified object with our own eyes and judge how well it approximates the original object. We can also perceive when it is appropriate to render this approximation in place of the original object. Since the ideal metric has not yet been realized, we define it by using our own eyes. Our perception defines high quality. Therefore, when we say that an algorithm produces high quality approximations, we imply that it produces good-looking approximations according to our own eyes. Figure 1.5 further explains this definition of high quality.

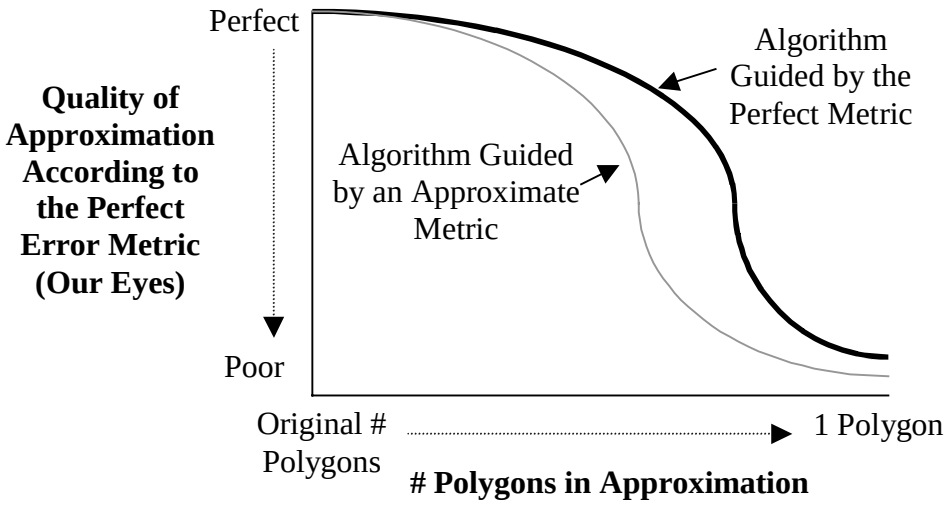


Figure 1.5: Our definition of high quality. The vertical axis measures the quality of an approximation using the perfect error metric. The horizontal axis shows the number of polygons in an approximation, ranging from the original object down to an approximation that consists of one polygon. The solid black curve denotes the quality of approximations generated by an algorithm that is guided by the perfect error metric. In other words, for every number of polygons, this curve shows the quality of the best possible approximation. The lighter gray curve denotes the quality of simplifications produced using an approximate error metric. The goal of a simplification algorithm is to minimize the area between its curve and the ideal curve, thus producing high quality approximations. Since a perfect error metric has not been discovered, the dark curve is defined by our visual judgement.

1.2.2 Drastic

A drastic approximation is a representation containing a small fraction of the polygons as compared to the original object or objects. This fraction depends on how many polygons make up the original objects being simplified, but it is always less than 10^{-2}. Thus, original objects consist of at least two orders of magnitude more polygons as compared to a drastic approximation of them. For objects or environments consisting of more than 1,000,000 polygons, this fraction can be as low as 10^{-4}. In these cases, a drastic approximation might consist of 100, 1,000, or 10,000 polygons.

1.3 Motivation

An abundance of research has been done in the field of polygonal simplification: some papers deal with simplifying single objects while others deal with visualizing static environments using the output of simplification algorithms. To the best of our knowledge, no previous work directly focuses on the problems associated with simplifying large dynamic environments. The motivation for this thesis can be understood by reviewing the previous work.

1.3.1 Simplification of Static Polygonal Objects

A great deal of research over several years has been done in the field of simplification of static polygonal objects. Some algorithms deal exclusively with simplifying polygonal geometry and ignore surface attributes such as colors, normals, and textures. [Schroeder et al. 92] iteratively decimates manifold meshes. [Turk 92] uses repulsion forces to determine the positions of vertices of the simplified object. [Hoppe et al. 93] attempts to minimize an energy function during simplification. [Rossignac and Borrel 93] overlays models with a uniform grid to simplify topology. [Eck et al. 95] uses wavelets to create a multi-resolution representations for objects. [He et al. 95] uses volumetric simplification techniques. [El-Sana and Varshney 97] detects and removes holes from objects. [Low and Tan 97] extends upon [Rossignac and Borrel 93] by using a more advanced spatial subdivision called *floating-cell clustering*. [Schroeder 97] extends upon [Schroeder et al. 92] by performing decimation operations in order of increasing error. Since none of these algorithms can handle surface attributes, they can produce poor approximations for objects containing attributes as illustrated by the radiositized Chamber model shown in Figure 1.6.

Figure 1.6: The benefits of handling surface attributes. On the left is an overhead view of the original radiositized Chamber model consisting of 10,423 polygons. In the middle is the result of simplifying the model while ignoring surface attributes. On the right, we use surface attributes to guide the simplification process, retaining more of the color information of the model. Each simplified model consists of 5,120 polygons.

Other algorithms use more advanced techniques to simplify and guarantee that the topology of an object will not change. [Cohen et al. 96] uses *simplification envelopes* to guarantee error bounds. [Hoppe 96] decimates manifold meshes using the *edge collapse* operation to create *progressive meshes*. [Ronfard and Rossignac 96] collapses edges to vertices that minimize error according to a set of local planes. [Cohen et al. 97] bounds geometric and texture-coordinate error using *successive mappings*. [Cohen et al. 98] preserves the surface appearance of an object by storing color and normal information in texture and normal maps, respectively. [Lindstrom and Turk 98] demonstrates that high quality simplifications can be produced even though information from previous decimation operations is ignored. Since none of these algorithms change the topology of objects, they are limited in the amount of simplification that they can perform in certain cases. For example, to simplify the Rotor model, shown in Figure 1.7, to low polygon count approximations, an algorithm must be able to close the holes of the object.

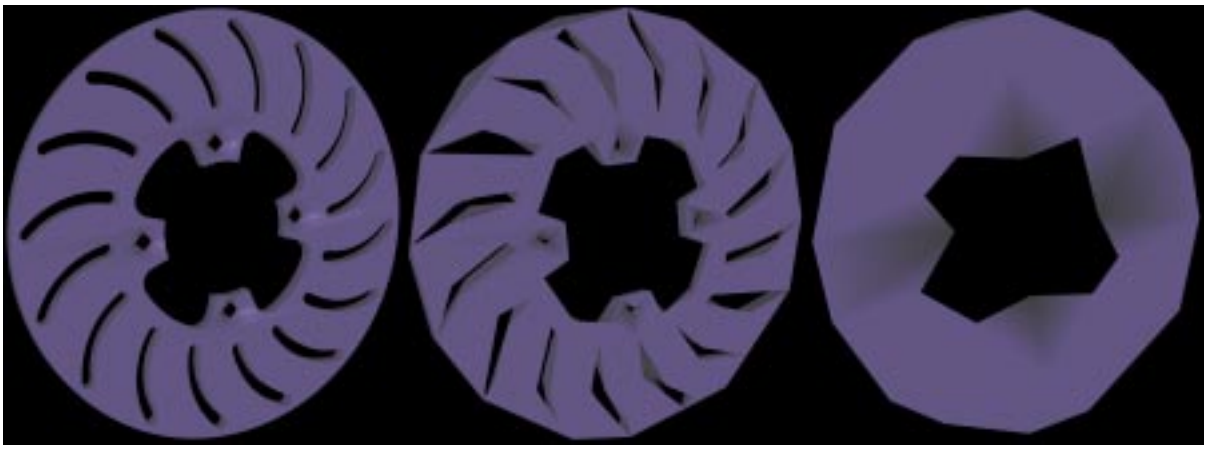


Figure 1.7: The benefits of topological simplification. The original Rotor model, consisting of 4,735 polygons, is on the left. If we do not allow topological simplification, then we cannot eliminate the tiny holes in this object. The coarsest approximation we can produce without topological simplification is in the middle and consists of 480 polygons. On the right, we simplify the Rotor model's topology to produce a 68 polygon approximation.

A few algorithms are capable of topological simplification. [Garland and Heckbert 97] uses *error quadrics* to measure geometric error at vertices. It simplifies topology, guided by a user-specified tolerance parameter. Specifying a good tolerance parameter is not trivial and in general, no one tolerance will work well on all objects. [Garland and Heckbert 98] extends upon [Garland and Heckbert 97] to handle surface attributes. [Popovic and Hoppe 97] presents a general technique that sacrifices speed in order to handle all types of polygonal input.

We desire a simplification algorithm that exhibits the following properties:

- Handles all types of polygonal input, including polygons that contain surface attributes as well as non-manifold and degenerate polygonal geometry;
- Simplifies objects automatically, without requiring the user to tweak parameters on a per-object basis (changing a global parameter for an entire scene of objects is acceptable);
- Produces useful error bounds for both polygonal geometry and surface attributes;
- Creates high quality approximations of an object, preserving its basic shape, main features, and surface appearance;

- Produces drastically low polygon count approximations;
- Executes quickly and uses as little memory as possible.

To the best of our knowledge, no one simplification algorithm *simultaneously* exhibits all of these properties.

1.3.2 Simplification of Static Polygonal Environments

Previous visualization algorithms have used polygonal simplification methods to render large environments at accelerated frame rates. Early methods used a traditional scene graph representation combined with object LODs in order to visualize large models in real-time. However, these techniques cannot simplify across different objects in the scene. Merging polygons between objects helps produce better-looking coarse approximations for models with numerous parts in close proximity (see Figure 1.8). [Rohlf and Helman 94] describes IRIS Performer, a scene graph based toolkit for real-time visualization of models. [Schneider et al. 94] presents a similar system where LODs are created using the algorithm described in [Rossignac and Borrel 93]. [Cohen et al. 96] integrates LODs produced by *simplification envelopes* into IRIS Performer to visualize large models.

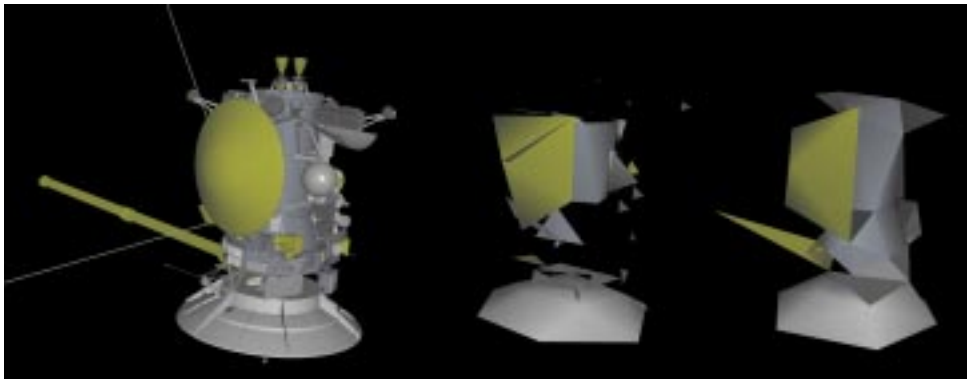


Figure 1.8: On the left is the Cassini spacecraft model consisting of 127 objects and 349,281 polygons. The middle image shows a drastic simplification of the Cassini model using 226 polygons where we do not merge polygons of different objects. On the right, we merge the polygons of different objects to produce a better drastic simplification consisting of 217 polygons.

Besides accelerating the frame rate, some research also focuses on rendering at constant frame rates. In other words, the user specifies a target frame rate and the algorithm attempts to render the best possible representation of the scene within this time constraint.

There are two types of target frame-rate methods based on polygonal simplification techniques, *predictive* and *reactive*. Predictive algorithms attempt to predict how long the next frame will take to render in order to select an appropriate set of LODs for the scene. In order to do this prediction, assumptions must be made about system performance that may not be accurate during run-time. [Funkhouser and Séquin 93] presents an algorithm that assigns a cost-benefit value for each LOD of each object in the scene. Picking which LODs to render is reduced to a Knapsack problem. A greedy method is used to pick an approximation solution to this problem. [Maciel and Shirley 95] extends [Funkhouser and Séquin 93] by hierarchically grouping objects into *meta-objects*.

Reactive algorithms use the performance of previous frames to guide the selection of LODs for the next frame. [Rohlf and Helman 94] uses a *feedback loop* to coarsen or refine objects in the scene depending on the performance of previous frames. [Mueller 95] describes how flight simulators also use feedback loops to adjust which LODs to render from frame to frame. These techniques must attempt to prevent oscillation between slow and fast frame rates.

Recently, several researchers have proposed *view-dependent simplification*, a method that enables adaptive simplification across objects or regions of an environment. These algorithms are very elegant and work well on spatially large objects. They impose significant CPU and memory overhead, involve traversing a vertex tree for every object in the scene, and are inherently immediate-mode techniques unable to take advantage of display lists. [Xia et al. 97] uses edge collapses in order to visualize single objects close to the viewer. [Hoppe 97] proposes a view-dependent technique based on the *progressive mesh* work of [Hoppe 96]. [Luebke and Erikson 97] demonstrates a view-dependent algorithm capable of merging different objects in the scene.

We desire a visualization algorithm for rendering large static polygonal environments that exhibits the following properties:

- Automatically merges polygons of different objects in the scene to produce drastic simplifications;
- Supports two rendering modes: one that allows the user to specify a desired image quality and another that allows the user to specify a target frame rate;
- Uses predictive techniques to target a frame rate but reacts to run-time peculiarities such as display list cache misses;
- Takes advantage of computer graphics hardware to render models as efficiently as possible.

To the best of our knowledge, no one visualization algorithm exhibits all of these properties *simultaneously*.

1.3.3 Simplification of Dynamic Polygonal Environments

Little research has been performed on the problem of simplification of large dynamic polygonal environments. Algorithms that support scene graphs, such as the one described in [Rohlf and Helman 94], allow movement of objects by modifying transformations between parent and children nodes. These techniques do not support merging polygons from different objects in the scene and cannot create high quality drastic approximations of groups of objects.

Other papers discuss how to update bounding-volume hierarchies and spatial partitionings after the movement of objects. [Torres 90] discusses dynamically updating BSP trees using a hierarchy of six levels of splitting planes. [Chrysanthou and Slater 92] updates BSP trees incrementally after movement. [Sudarsky and Gotsman 96] incrementally updates an octree spatial partitioning and also uses *temporal bounding volumes* to bound the extents of an object through a range of time.

We desire a visualization algorithm for dynamic polygonal environments that exhibits the following properties:

- It should dynamically update bounding volume hierarchies and spatial partitionings incrementally and efficiently;

- Since the algorithm merges polygons of different objects in the scene, the relative movement of these objects might change previously created approximations. The algorithm should be able to identify and update these approximations. Figure 1.9 shows this process.

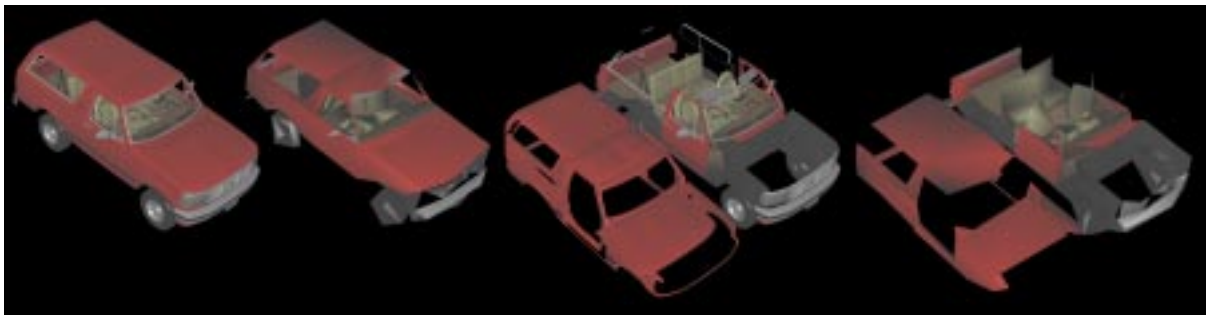


Figure 1.9: On the left is the original Ford Bronco model consisting of 466 objects and 74,308 polygons. Next to it is a coarse approximation consisting of 580 polygons where polygons of different objects have merged together. The next image, consisting of 74,308 polygons, shows a user removing the Bronco's top to view its interior. On the right, the coarse approximation of the model has been updated and consists of 552 polygons.

1.4 Assumptions

Like many other workers, [Clark 76, Rohlf and Helman 94, Cohen et al. 96], we assume that the polygonal environment being rendered is represented by a *scene graph*. A scene graph is a directed acyclic graph consisting of *nodes* connected by *arcs*. A node contains polygonal geometry as well as a bounding volume that encloses the node's polygons plus all of the bounding volumes of the node's children. A directed arc connects a parent and child node. The arc also contains a child-to-parent space transformation matrix. An *instanced* node is one that is a child of several parent nodes. Instancing allows objects in the scene to be replicated easily and efficiently.

The only visibility techniques we use in our approach are standard *back-face culling* and *view-frustum culling* [Clark 76, Rohlf and Helman 94]. Back-face culling assumes that objects in the scene are closed and ignores polygons that are facing away from the viewer. A *view-frustum* is a volume enclosing all space that a user could potentially see from a particular

vantage point. If the bounding volume associated with a node in the scene graph does not intersect this frustum, then the node can be ignored during rendering.

We assume that the polygonal geometry within each node of the scene graph is static and not deformable. Therefore, our work deals with *rigid body* environments where objects in the scene move due to modifications of the scene graph. Dynamic environments require several scene graph operations such as adding nodes and arcs, deleting nodes and arcs, and changing transformations at arcs. This last operation is the most common for scenes composed of moving objects.

1.5 Thesis Statement

Polygonal simplification techniques based on merging unconnected regions of polygons and groups of objects can be the foundation of a powerful algorithm for interactive visualization of large static and dynamic polygonal environments.

1.6 New Results

We approach the visualization of large static and dynamic polygonal scenes by splitting the problem into three steps that build upon each other. First, we tackle the problem of generating multi-resolution representations for an arbitrary set of polygons. We call such a collection of polygons an *object*. Since we deal with rigid bodies, this problem is the *simplification of static polygonal objects*. Using simplification techniques to render static polygonal environments represented by scene graphs is the second task, which we call *simplification of static polygonal environments*. Finally, we concentrate on rendering dynamic polygonal environments, or the problem of *simplification of dynamic polygonal environments*. Below we list our new results in each of these areas.

1.6.1 Simplification of Static Polygonal Objects

We introduce a new static polygonal object simplification algorithm called *General and Automatic Polygonal Simplification*, or *GAPS*. The main features of GAPS are:

- **General:** GAPS is able to simplify polygonal objects with cracks, non-manifold vertices and edges, coincident polygons, and T-joints.
- **Automatic:** No user input is required to guide the simplification process. Often it is necessary to simplify a large polygonal scene, containing numerous objects. Requiring the user to input parameters on a per-object basis is tedious and time-consuming.
- **Surface Attribute Handling:** In order to enhance visual realism, objects usually include surface attributes such as textures, colors, and normals. GAPS uses these surface attributes to guide the simplification process and approximates them in the result.
- **Unified Error Metric:** GAPS produces a unified error bound that is sensitive to both the polygonal geometry and the surface attributes of the simplified object. Our visualization algorithm uses these error bounds to automatically determine switching distances between LODs.
- **High Quality:** GAPS constructs high quality approximations of an object, preserving its basic shape, main features, and surface appearance.
- **Drastic Simplification:** GAPS is capable of drastic simplification of a polygonal object such that a user can target a number of polygons and it produces an appropriate simplification.
- **Topological Simplification:** GAPS joins unconnected regions of the object or changes its genus by closing holes in order to achieve high quality and drastic simplifications of an arbitrary group of polygons.
- **Efficient:** GAPS simplifies quickly.

GAPS has been tested on a wide variety of polygonal objects including terrain meshes, radiositized models, scanned meshes, and large CAD models. It produces high quality simplifications quickly. Figure 1.6, shown previously on page 9, demonstrates the effectiveness of GAPS on handling surface attributes during simplification. Figure 1.10, below, shows the result of using GAPS on a complicated pipe structure. On a 195 MHz

R10000 SGI Onyx 2, GAPS simplified these pipes in 26 seconds and the Chamber model in 7 seconds. On average, GAPS performs 500 vertex merge operations per second on this machine. If surface attributes are ignored and topological simplification is not used, GAPS runs approximately two times faster. In other words, handling surface attributes and performing automatic topological simplification comes at a cost of half the performance.

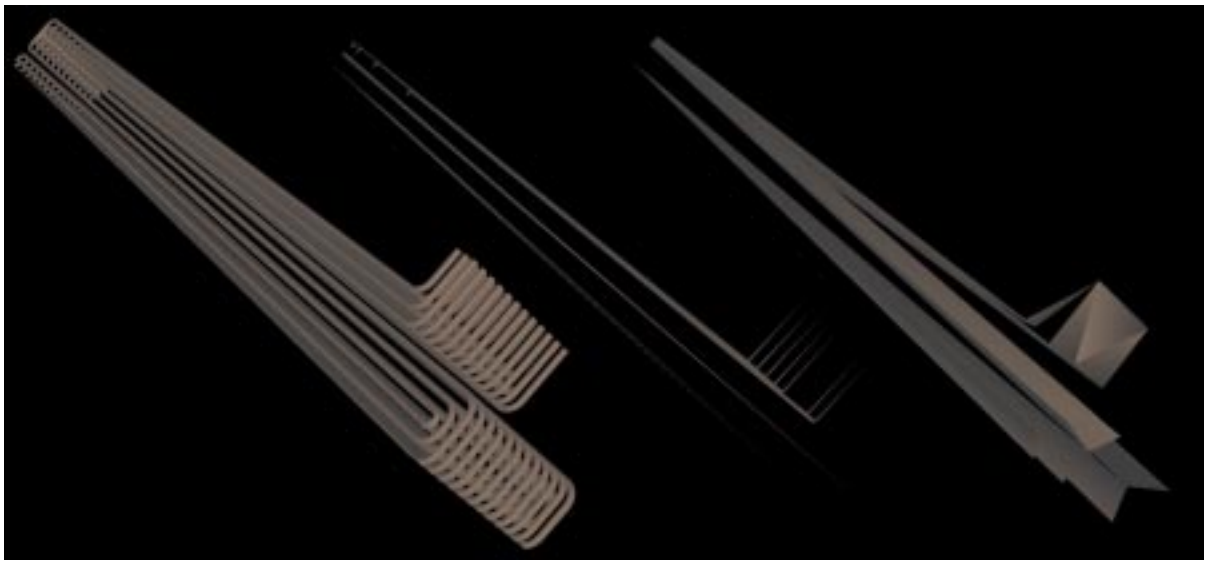


Figure 1.10: On the left is the original series of pipes consisting of 23,556 polygons that come from a power plant model. In the middle is the simplified output of an algorithm that does not merge unconnected regions of the model. On the right, GAPS merges unconnected regions of the pipes to produce a higher quality drastic simplification. Both approximations consist of roughly 90 polygons.

1.6.2 Simplification of Static Polygonal Environments

We present a new algorithm based on our simplification techniques for displaying large static polygonal environments. It uses GAPS to preprocess the model. The main features of our approach are described below:

- **Hierarchical Levels of Detail:** Not only does the method use GAPS to create LODs for polygonal geometry at each node of the scene graph, but it also uses GAPS to create *hierarchical levels of detail*, or *HLODs*. HLODs represent portions of the scene graph, or groups of objects. Our algorithm uses the higher-

order HLODs to cull away portions of the scene graph when necessary to achieve a target frame rate or for displaying a distant group of objects.

- **Association:** The creation of HLODs depends on *associating*, or grouping, nearby objects together. This grouping process is done hierarchically using an octree spatial subdivision.
- **Partitioning:** Spatially large objects are problematic for traditional LOD algorithms. By *partitioning* these objects and then grouping the partitions hierarchically, our technique gains limited *view-dependent* rendering capabilities.
- **Two Rendering Modes:** The algorithm can render using two different modes. A desired image quality can be achieved by specifying an allowable *pixel error*. The second mode is a *target frame-rate* mode where the algorithm attempts to render the best possible image given a frame-rate constraint.
- **Efficient:** To take advantage of graphics hardware, we use *display lists* to render LODs and HLODs.

We tested our technique on several different large CAD environments including a terrain mesh, an automobile, a spacecraft, a submarine, and a power plant. Our method allows us to interactively navigate and explore these environments. Figure 1.8, shown previously on page 11, demonstrates the visual benefit of HLODs compared to using solely LODs. Using an SGI Reality Monster with an SGI Infinite Reality 2E graphics subsystem containing 4 Graphics Engines (GEs) and 2 Raster Managers (RM9s), a 300 MHz R12000 processor, and 16GB of main memory, we preprocessed the Power Plant model, consisting of 12,731,154 polygons and 1,179 objects, in less than 4 hours and 15 minutes. Using a combination of LODs and HLODs, our algorithm rendered this model on average nearly 306 times faster than using no LODs at all on a selected viewing path with little or no loss in image quality. Due to the complexity of this environment, this speedup is not enough to achieve interactive frame rates from all viewpoints. Figure 1.11 shows the loss in image quality that we must accept to achieve 10 frames per second for a particular view of the model. Our algorithm was able to render the other CAD environments at interactive frame rates with little or no loss in image quality.

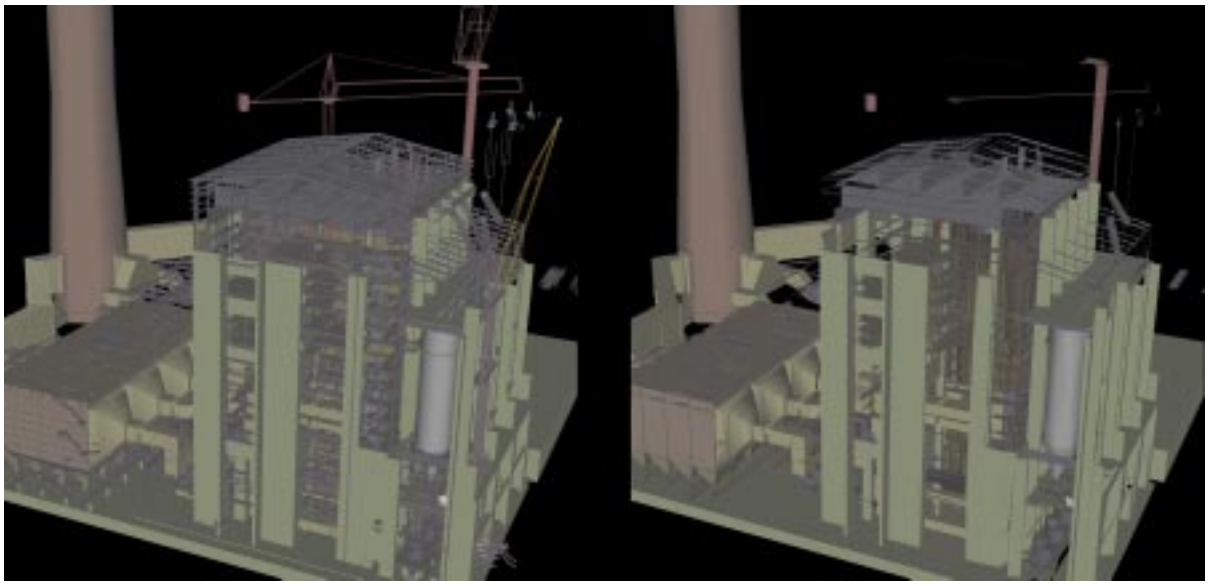


Figure 1.11: Two views of the Power Plant model rendered in a 1000 by 1000 pixel window on an SGI Reality Monster with an SGI Infinite Reality 2E graphics subsystem containing 4 GEs and 2 RM9s, a 300 MHz R12000 processor, and 16GB of main memory. On the left we render the original model at 0.05 frames per second from this viewpoint. On the right, we allow 45 pixels of error in order to achieve approximately 10 frames per second from the same viewpoint.

1.6.3 Simplification of Dynamic Polygonal Environments

In rigid-body dynamic environments, the bounding volume hierarchy of the scene graph, structure of the scene graph, and transformations at arcs can change. Therefore, the fundamental problem with dynamic scenes using our method is that HLODs change as objects move. To be more precise, only HLODs that represent objects that have moved must be recalculated. The main features of our algorithm's dynamic capabilities are described below:

- **Updating the Scene Graph:** Movement of objects in the scene changes error bounds of HLODs, the bounding volume hierarchy, and the structure of the scene graph. Previously created node associations, which group objects according to their proximity, may need to be updated. Therefore, the algorithm *re-associates* objects to update the scene graph structure. This re-association process involves associating objects that have moved close together and removing associations between objects that have moved far apart.

- **Asynchronous Simplification:** Polygonal simplification is currently a slow process compared to rendering. When objects move, and HLODs need to be recalculated, the algorithm creates the HLODs *asynchronously* from the rendering process. This technique allows our algorithm to render the scene without waiting for HLOD recalculations to complete.

We tested the dynamic portion of our visualization algorithm on the same large CAD environments we tested in our static approach. Our current implementation is able to recalculate HLODs in a few seconds only in scenes with limited dynamic movement. Thus, it is most useful for design and review scenarios, where the user infrequently manipulates a few objects in the environment at a time. Figure 1.9, shown previously on page 14, demonstrates the ability of the method to update the HLODs of an environment after objects in the scene have moved. The recalculation of these HLODs for this figure took 3 seconds using 4 processors on an SGI Reality Monster with 300 MHz R12000 processors and 16GB of main memory. Since these HLODs were recalculated asynchronously from the rendering process, the user was able to interactively navigate around the model during the entire viewing session.

1.7 Dissertation Overview

The rest of the dissertation is organized as follows. Chapter 2 describes related work in the areas of simplification of static polygonal objects, static polygonal environments, and dynamic polygonal environments. Since there exists an abundance of work in the field of polygonal simplification, this chapter does not attempt to cover all of it. Instead, we highlight specific algorithms that are most relevant to our dissertation.

Chapter 3 presents *General and Automatic Polygonal Simplification*, or *GAPS*, for short. This algorithm merges unconnected regions of polygons by using an *adaptive distance threshold* and *surface area preservation*. During execution, it handles surface attributes to guide the simplification process. It uses a *unified error metric*, that combines both geometric and surface attribute error, to produce error bounds useful for automatically calculating switching distances for LODs. We have used GAPS on a wide variety of models ranging from

radiositized meshes to complex CAD parts. GAPS simplifies quickly and produces high quality and drastic approximations.

In Chapter 4, we describe our scene-graph-based visualization algorithm used to render large static polygonal environments. By using *hierarchical levels of details*, or HLODs, we are able to merge polygons of different objects in the scene graph to produce better-looking drastic approximations for groups of objects. We use GAPS to produce these HLODs. For efficient view-frustum culling and HLOD creation, our technique is able to *associate*, or group, nodes in the scene graph based on spatial proximity. By *partitioning* spatially large objects and then grouping these partitions together hierarchically, we are able to approximate a discrete version of view-dependent rendering. HLODs also allow the algorithm to render at a target frame rate. We have used our algorithm to interactively navigate several large and complex CAD environments.

Chapter 5 describes how our visualization algorithm is extended to handle dynamic movement of objects in the scene. To handle dynamic movement, our algorithm updates error bounds of affected HLODs, *re-associates* nodes in the scene graph based on the new positions of objects, and updates the bounding volume hierarchy of the scene. It detects HLODs that have been affected by this movement and recomputes them *asynchronously* by using GAPS on separate simplification processes. We have demonstrated our algorithm on several large CAD environments with limited dynamic movement.

Finally, Chapter 6 presents conclusions and discusses possible avenues for future work.

2 PREVIOUS WORK

In this chapter, we describe previous work in the three areas addressed in this dissertation. The first area is simplification of static polygonal objects, or polygonal simplification. Next, we review work in simplification and visualization of static polygonal environments. The final area is the simplification and visualization of environments with moving objects.

2.1 Simplification of Static Polygonal Objects

Numerous methods have been proposed that produce simplified versions of polygonal objects. They vary in terms of quality of approximation, efficiency, simplification operations, and assumptions on the input model. In this section, we cover only some of these algorithms, namely ones that have been frequently referenced in the polygonal simplification literature. For a more thorough survey of simplification algorithms, consult [Heckbert and Garland 97]. Even though each algorithm is unique, we classify them by the general method they use to produce simplifications. The three classifications are *refinement*, *sampling*, and *decimation*. Similar classification schemes appear in [Erikson 96, Luebke 97, and Heckbert and Garland 97].

2.1.1 Refinement Algorithms

Refinement algorithms start with a very simple base representation of the original object and recursively refine it, adding more and more detail to local areas of the model at each step. Once the refined object approximates the original to some user-specified error tolerance or some polygon limit has been reached, the algorithm terminates.

2.1.1.1 Multiresolution Analysis of Arbitrary Meshes [Eck et al. 95]

This algorithm uses wavelets to create a multi-resolution representation for a polygonal object. Smoothly interpolating between these different representations simply requires adding or subtracting wavelet coefficients. This method can handle any three-dimensional manifold that is a valid triangulated mesh. The algorithm cannot exactly reconstruct models with sharp edges or creases, and thus its approximations of CAD objects tend to be of lower quality than algorithms that preserve these sharp edges.

The algorithm initially creates a base mesh that is the coarsest representation of the object. Polygons are grouped together to approximate a Voronoi diagram based on geodesic distance over the surface of the object. The algorithm performs a Delaunay triangulation of this diagram resulting in a base mesh consisting of base faces. Next, this method creates a globally continuous parameterization of the base mesh by ensuring continuity between the boundaries of base faces. Using this parameterization, it can refine the base faces by splitting each triangle into four triangles. This refinement process continues until a user-specified error bound is met.

2.1.2 Sampling Algorithms

Sampling algorithms initially sample the polygonal geometry of the original object by either sampling random points from its surface, or by voxelizing it. These algorithms reconstruct a lower polygon count object to fit this sampled data. The user typically has control over how much sampling is done, but not much control over the resulting number of geometric primitives in an approximation.

2.1.2.1 Re-Tiling Polygonal Surfaces [Turk 92]

This algorithm allows the user to specify a specific number of vertices in the final simplified object, useful for creating a series of approximations. It can handle three-dimensional manifolds with concave or convex polygons, even ones with holes. The method performs best on polygonal objects that approximate curved surfaces since it has difficulty preserving sharp edges.

Initially, the algorithm randomly places the user-specified number of points over the surface of the object. Next, it simulates repulsion forces between the points to distribute them evenly across the surface. It incorporates these points as new vertices in the original object by re-triangulating each face to include them. The algorithm removes the original vertices one by one, filling in the resulting holes by re-triangulation (see Figure 2.1). The remaining vertices of the simplified object are the sampled points.

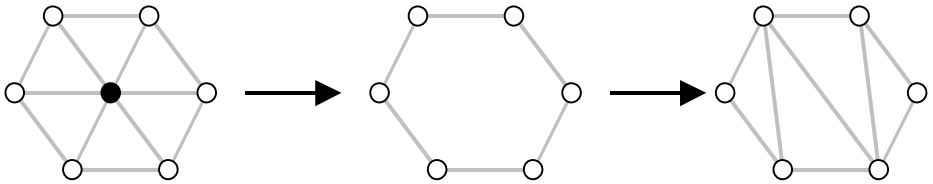


Figure 2.1: Example of vertex removal of the black vertex. When the vertex is removed, a hole forms. This hole must be re-triangulated.

2.1.2.2 Mesh Optimization [Hoppe et al. 93]

This algorithm was originally designed to reduce the polygonal geometry of meshes reconstructed from sampled points. By first point sampling a polygonal object, the algorithm can be applied to any three-dimensional manifold. It produces excellent simplifications slowly and allows the user to specify a constant that determines the number of polygons in the resulting output.

The algorithm distributes numerous point samples across the surface of the input object. Using these samples, plus the original vertices of the object, it reconstructs an initial mesh consisting of triangles. The method tries to minimize an energy function consisting of a distance term, a vertex term, and a spring term. The distance term is proportional to the sum of squared distances between the sample points and the simplified mesh. The vertex term is proportional to the number of vertices in the mesh. The spring term simulates the effect of placing springs on the edges of the mesh, effectively penalizing long edges. An iteration of the algorithm consists of an outer and inner optimization loop. The outer loop minimizes the energy function by randomly selecting an edge and performing one of three different operations (edge collapse, edge swap, or edge split). The inner loop minimizes the distance

and spring terms by optimizing the position of vertices. The algorithm gradually decreases the spring term to guide this optimization process to a local minimum. It stops after a fixed number of iterations.

2.1.2.3 Multi-resolution 3D Approximations for Rendering Complex Scenes [Rossignac and Borrel 93]

This algorithm accepts any input object, no matter if it contains non-manifold or degenerate geometry. This method runs very quickly and ignores the topology of the original object, sometimes producing low quality approximations.

The initial step of the algorithm involves automatically weighting vertices according to their perceptual importance, estimated using local curvature and incident edge lengths. Each polygonal face is triangulated and a three-dimensional grid is placed on top of the object. Vertices that lie in a single grid cell are grouped together. These vertices are collapsed to the most important vertex in the cell, determined by the initial weighting process. Degeneracies created by the collapsing of vertices are removed and normals for the final triangles are recalculated. The user can specify the size of the grid cells in order to control the degree of simplification.

2.1.2.4 Model Simplification Using Vertex-Clustering [Low and Tan 97]

This algorithm extends on [Rossignac and Borrel 93] in order to improve the quality of approximations, while still executing quickly. It can handle all three-dimensional inputs, including non-manifold and degenerate geometry.

The details of the algorithm are equivalent to [Rossignac and Borrel 93] except for two aspects. It uses an improved system for initially weighting vertices and abandons the uniform subdivision of [Rossignac and Borrel 93] by creating cells centered on the most important vertices first. This *floating-cell clustering* is less sensitive to changes in the size of the grid cells. The algorithm uses thick lines to draw elongated parts of an object that have collapsed to a single edge. As in [Rossignac and Borrel 93], the user can specify the size of the grid cells in order to control the degree of simplification.

2.1.2.5 Voxel-Based Object Simplification [He et al. 95]

This algorithm uses a signal processing approach to simplify polygonal objects. It eliminates high frequency detail of an object and thus smoothes sharp edges or creases. It assumes the input object is a closed volume, meaning that the object has a clearly defined inside and outside.

First, the algorithm voxelizes the object by overlaying a three-dimensional grid on top of it. For each voxel, it approximates the percentage of the voxel that is inside and outside the polygonal object. Using a low pass filter, the final density of each voxel is determined. The algorithm uses a variation on the Marching Cubes algorithm [Lorensen and Cline 87] to reconstruct the simplified object from this voxel representation. The user can vary the voxel size in order to produce a series of approximations for an object.

2.1.3 Decimation Algorithms

The majority of polygonal simplification algorithms are decimation algorithms. These algorithms start with the original object and repeatedly remove vertices, edges, or faces. This decimation process continues until either the object can no longer be simplified and still meet a user-specified error bound, or a polygon target has been reached. It is assumed in all of the decimation algorithms described below that the input object has been triangulated.

Common operations that decimation algorithms use to reduce the amount of polygonal geometry in an object are *vertex removal*, *edge collapse*, and *vertex merge* (see Figure 2.2). The vertex removal operation removes a single vertex and faces incident to that vertex. A hole is created because of this removal and must be filled in by re-triangulation. Performing an edge collapse involves sliding the two vertices that make up the edge onto a common point. The vertex merge operation forces two vertices to move to a common point. If the two vertices share an edge, then this operation is equivalent to an edge collapse. If they do not share an edge, then we say they share a *virtual edge*.

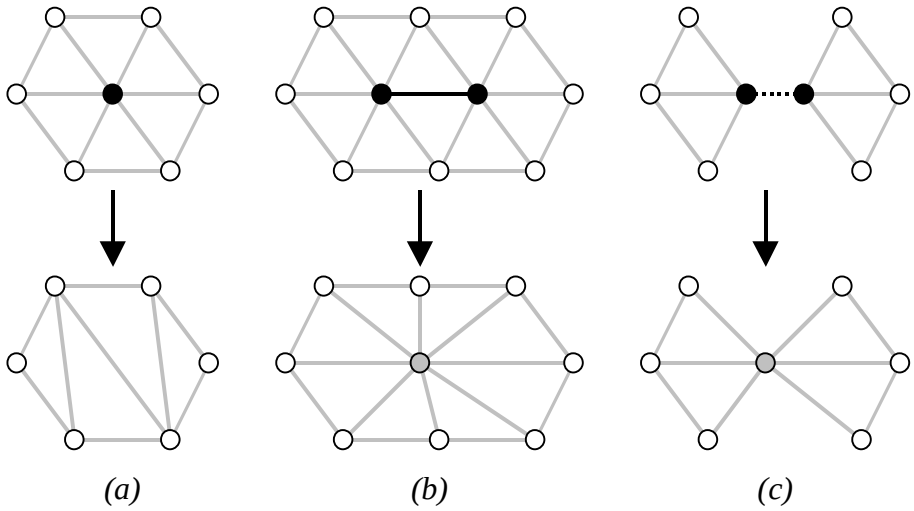


Figure 2.2: Examples of operations used in decimation algorithms. *(a)* Vertex removal of the black vertex. When the operation takes place, a hole forms that is subsequently re-triangulated. *(b)* Edge collapse of the black edge incident to the two black vertices. The edge is collapsed to a common point at the gray vertex. *(c)* Vertex merge of the two black vertices. The dotted black line denotes a virtual edge. The two vertices are merged to a common point at the gray vertex.

2.1.3.1 Decimation of Triangle Meshes [Schroeder et al. 92]

The Marching Cubes algorithm [Lorensen and Cline 87] tends to create polygonal objects with large numbers of polygons even in planar regions. [Schroeder et al. 92] describes an algorithm to simplify these types of polygonal objects. This method, based on vertex removal, handles three-dimensional manifolds. The vertices of the simplified object are guaranteed to be a subset of the original. Therefore, attribute data stored at vertices, such as color and texture coordinates, need not change or be recomputed during simplification.

The algorithm chooses a vertex and calculates an average plane of its local polygonal geometry. The normal of this plane is the area-weighted average of normals of incident faces to the vertex. One point on the plane is the area-weighted average of the centers of incident faces to the vertex. This normal and point define the average plane of the local polygonal geometry. If the distance between the vertex and its average plane is less than a user-specified error bound, the vertex is removed from the object. This process of choosing a vertex and removing it if it meets the error criterion repeats until no more vertices can be removed.

2.1.3.2 A Topology Modifying Progressive Decimation Algorithm [Schroeder 97]

This algorithm uses the same error metric introduced in [Schroeder et al. 92]. However, it has been modified to simplify in order of increasing error. It handles non-manifold and degenerate geometry, simplifies topology, and is extremely fast. The algorithm is not capable of merging unconnected polygons and thus can produce low quality approximations for objects with closely grouped, but disjoint polygons.

As in [Schroeder et al. 92], the algorithm calculates the distance between each vertex and the average plane of its local polygonal geometry. Vertices are placed in a heap according to this distance and are extracted in order of increasing error. A vertex is removed by an edge collapse operation involving its shortest incident edge. Edge collapses that cause faces to flip orientation and fold over one another are deemed invalid. When no valid edge collapse operations exist for a vertex, it is split into two vertices. These two vertices can then be simplified independently of each other, potentially causing the local polygonal geometry to split apart. This splitting process modifies the topology of the object. The algorithm continues to simplify until it reaches a specified error bound or polygon count.

2.1.3.3 Progressive Meshes [Hoppe 96]

This paper presents the *progressive mesh*, a new representation for simplified objects that allows for continuous levels of detail. A progressive mesh is simply an ordered list of decimation operations. An object can be simplified or refined by performing or undoing these operations.

The paper also describes a simplification algorithm that produces progressive meshes by minimizing an energy function much like that of [Hoppe et al. 93]. However, it starts with the original object, not a sampled version, and uses only the edge collapse operation to decimate. During simplification, it handles both *scalar attributes*, such as vertex colors, normals, and texture coordinates, and *discrete attributes*, such as face materials and textures. It also handles instances of attribute discontinuities, such as normals surrounding sharp edges. It

runs slowly and allows the user to specify the relative importance between geometric and surface attribute error to achieve high-quality simplifications.

The algorithm minimizes an energy function consisting of a distance term, a spring term, a scalar term, and a discontinuity term. The distance and spring terms are identical to the corresponding terms in [Hoppe et al. 93]. The scalar term measures the distance in attribute space between the original and the simplified mesh. The discontinuity term measures the geometric distance between the original and the simplified mesh along attribute discontinuity edges. An iteration of the algorithm again consists of an outer and inner optimization loop. The outer loop minimizes the energy function by selecting an edge collapse that is on top of a heap, ordered by increasing error. The inner loop minimizes the distance, spring, scalar, and discontinuity terms by optimizing the position of vertices. The algorithm terminates when it simplifies to a user-specified triangle limit.

2.1.3.4 Progressive Simplicial Complexes [Popovic and Hoppe 97]

This algorithm extends on [Hoppe 96] in order to simplify any three-dimensional object, even if it is non-manifold or degenerate. Vertex merge is the decimation operation used. Unlike other algorithms that filter degenerate triangles, it uses spheres and cylinders to render triangles collapsed to points and lines, respectively. The algorithm executes extremely slowly.

A set of candidate vertex pairs is initially determined by finding vertices that share an edge or are in close proximity according to a Delaunay triangulation. Vertices that do not share an edge, but are a candidate pair, share a virtual edge. By using vertex merges on these pairs in order of increasing error, the algorithm is able to change the topology of the simplified object. As in [Hoppe 96], the algorithm minimizes an energy function representing error. This function consists of distance, discontinuity, area, and fold terms. The distance term is equivalent to that of [Hoppe et al. 93] and the discontinuity term preserves sharp edges in the object. The area and fold terms penalize surface stretching and folding respectively.

2.1.3.5 Simplification Envelopes [Cohen et al. 96]

The simplification envelopes algorithm guarantees that each point on the approximation is within a user-specified distance tolerance ε of the original object. This feature allows the user to switch automatically between representations of an object, depending on its distance from the viewer. The algorithm performs many involved geometric calculations to insure this distance tolerance guarantee, causing it to run slowly. This method accepts three-dimensional manifolds as input.

The algorithm creates inner and outer offset surfaces, or envelopes, a distance ε from the object such that they contain no self-intersections. Having no self-intersections implies that the topology of the object will not change during simplification, which sometimes limits the degree of simplification possible. The method then selects vertices to be removed from the object. A vertex removal operation is disallowed if any of the re-triangulated faces intersect either the inner or outer envelope. The algorithm repeatedly removes vertices until no more vertex removal operations are allowed.

2.1.3.6 Simplifying Polygonal Models Using Successive Mappings [Cohen et al. 97]

This algorithm uses the edge collapse operation to simplify polygonal objects with manifold geometry. It keeps track of both polygonal geometry and texture-coordinate error bounds during simplification and executes slowly.

This method uses a heap of edges, sorted by increasing error. An edge is weighted by first finding a projection plane of its local neighborhood, performing the collapse in this plane, computing a mapping between the original and collapsed neighborhoods, and then optimizing the position of the collapsed vertex. This computed mapping is also used to assign new texture coordinates to the collapsed vertex. Using axis-aligned boxes at each vertex, the algorithm keeps a tight bound on both geometric and texture-coordinate error during simplification.

2.1.3.7 Appearance-Preserving Simplification [Cohen et al. 98]

This algorithm, similar to [Cohen et al. 97], simplifies not only the polygonal geometry of an object but color and normal information in texture and normal maps respectively. It introduces an error metric for texture maps which guarantees that a texture does not shift more than a user-specified number of pixels on the screen. This method does not handle non-manifold or degenerate geometry and runs fairly slowly.

The algorithm converts an input object into a decoupled representation consisting of its polygonal geometry, texture maps representing its surface colors, and normal maps representing the curvature of its surface. Edges are collapsed in order of increasing total error, a combination of both geometric and texture deviation error. Axis-aligned boxes are stored at each vertex to help bound both geometric and texture deviation error efficiently. The simplified objects are of high quality as compared to algorithms that simplify colors and normals directly, rather than converting them to texture and normal maps. Currently only a few specialized machines, such as PixelFlow [Molnar et al. 92, Eyles et al. 97], developed jointly by Hewlett Packard and the University of North Carolina at Chapel Hill, can accelerate the rendering of models with normal maps.

2.1.3.8 Full-range Approximation of Triangulated Polyhedra [Ronfard and Rossignac 96]

This algorithm uses the edge collapse operation to decimate objects. It uses a set of planes at each vertex to bound the error of the simplified object. The algorithm is able to change topology during simplification. It executes at a respectable speed and produces high quality approximations. It does not merge unconnected regions of the model and uses a lot of memory. The vertices of the simplified object are a subset of the original vertices.

The algorithm initially calculates the error associated with each edge collapse in the object and inserts them into a heap, sorted by increasing error. To determine the error of an edge collapse, this method uses a local geometric error and a local tessellation error that prevents meshes from folding back on themselves. The geometric error at a vertex is defined to be the maximum distance between the vertex and its set of planes. This set of planes is

initially the planes of faces incident to the vertex. A new merged vertex has a set of planes equal to the union of the set of planes of the merged vertices. The position of a merged vertex is one of the two vertices being merged.

2.1.3.9 Surface Simplification Using Quadric Error Metrics [Garland and Heckbert 97]

This algorithm, similar to [Ronfard and Rossignac 96], tracks an error bound by associating each vertex with a set of planes. However, it approximates the set of planes using a symmetric matrix called an *error quadric*. It uses vertex merging as its decimation operation, so it is able to join unconnected regions of an object. It is fast and produces high quality results. It is fairly memory intensive since each vertex has an associated error quadric that requires 10 floating-point values of storage. Figure 2.3 shows a geometric interpretation of error quadrics.

If $\mathbf{p} = [a\ b\ c\ d]^T$ represents the plane defined by the equation $ax + by + cz + d = 0$ where $a^2 + b^2 + c^2 = 1$, then this algorithm associates with $\mathbf{p}$ a *fundamental error quadric* $\mathbf{K_p}$ where

$$\mathbf{K_p} = \mathbf{p}\mathbf{p}^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

To find the squared distance from a plane $\mathbf{p}$ to a point $\mathbf{v} = [v_x\ v_y\ v_z\ 1]$, the algorithm uses the quadric form $\mathbf{v}^T\mathbf{K_p}\mathbf{v}$. Error quadrics can represent a union of two quadrics, or a set of planes, simply by adding their corresponding matrices together. Each vertex $\mathbf{v}$ has an associated error quadric $\mathbf{Q}$ that may consist of several fundamental error quadrics summed together. The error at a vertex $\mathbf{v}$ with quadric $\mathbf{Q}$ is defined to be $\Delta(\mathbf{v}) = \mathbf{v}^T\mathbf{Q}\mathbf{v}$. This value represents the sum of squared distances between the vertex $\mathbf{v}$ and all of the planes associated with the quadric $\mathbf{Q}$. Given a quadric $\mathbf{Q}$, an optimal vertex position $\mathbf{v}$ can usually be calculated that minimizes $\Delta(\mathbf{v})$. This calculation involves inverting a matrix that is defined by $\mathbf{Q}$. If $\mathbf{Q}$ is not invertible, then other approximations are used to determine the new vertex position $\mathbf{v}$.

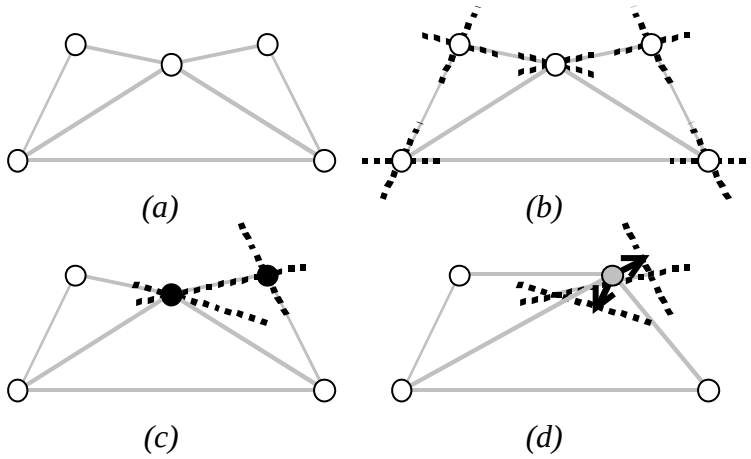


Figure 2.3: A geometric interpretation of error quadrics. *(a)* A simple object consisting of 5 vertices and 3 planar faces. *(b)* An error quadric represents a set of planes and each vertex has an associated error quadric. Initially, the error quadric for a vertex consists of the planes of the vertex's adjacent faces plus planes to preserve sharp edges and boundary edges. A vertex's adjacent faces are any faces that are incident to that vertex. In this example, there are only planes to preserve boundary edges, shown as dotted black lines. These planes define the error quadric at each vertex. *(c)* The two black vertices are next to be merged. Only the planes of error quadrics involved in this merge are shown. *(d)* The error quadric of the merged vertex is constructed by taking the union of the planes of the error quadrics involved in the merge. Therefore, all of the planes shown are included in the error quadric of the new vertex. The position of the new vertex is determined by attempting to minimize the sum of squared distances between it and all of the planes in its error quadric. The component distances used in this calculation are shown as black arrows and the new vertex is colored gray.

Initially, the error quadric of each vertex consists of planes of the vertex's adjacent faces plus planes to preserve sharp edges and boundary edges. A vertex's adjacent faces are any faces that are incident to that vertex. Candidates for vertex merging include vertices that share an edge or two vertices that are within an optional user-specified distance threshold τ. Vertices that do not share an edge, but are a candidate pair, share a virtual edge. For each candidate merge, the optimal vertex position and total error are calculated. The candidates are inserted into a heap, sorted by increasing error. The vertex merge on top of the heap is performed, and the error quadric of the new vertex becomes the sum of the quadrics of the merged vertices. The error of candidate pairs in the local neighborhood of the merge must be

updated. The algorithm continues to merge vertices until a target number of polygons is achieved.

2.1.3.10 Simplifying Surfaces with Color and Texture Using Quadric Error Metrics [Garland and Heckbert 98]

This algorithm extends [Garland and Heckbert 97] in order to simplify objects with surface attributes such as normals, colors, and texture coordinates. However, this algorithm uses edge collapse, rather than vertex merge, as its decimation operation. Therefore, the distance threshold τ presented in [Garland and Heckbert 97] does not exist in this version of the algorithm. This method produces high quality approximations and runs slower than [Garland and Heckbert 97] due to the extra overhead of surface attributes.

The algorithm extends the error quadric of [Garland and Heckbert 97] to incorporate colors, normals, and texture coordinates. For example, a vertex with color information is considered to be a 6-dimensional entity $\mathbf{v} = [v_x\ v_y\ v_z\ v_r\ v_g\ v_b]$ with an associated generalized error quadric consisting of a 6x6 matrix and a 6-dimensional vector. When an edge is collapsed, the optimal merged 6-dimensional vertex position is calculated by inverting a 6x6 matrix. Therefore, not only does this process determine the position of the new vertex, but the new color at the vertex as well. Normals and texture coordinates are handled in a similar fashion. In the extreme, a vertex containing a color, normal, and texture coordinate is represented by an 11-dimensional vector. Since the generalized error quadrics are of higher dimension than that of the error quadrics in [Garland and Heckbert 97], calculating merged vertices takes longer and memory requirements are greater.

2.1.3.11 Fast and Memory Efficient Polygonal Simplification [Lindstrom and Turk 98]

This algorithm demonstrates that it is not necessary for a simplification method to retain information about previous decimation operations in order to simplify well. Since it does not store any information during simplification, the memory it requires is equivalent to the data needed to represent the original object. This method uses the edge collapse operation,

simplifies fairly quickly, and requires little memory compared to other techniques. It is capable of simplifying degenerate and non-manifold geometry and is shown to produce high quality approximations by comparing its output with other simplification algorithms.

The algorithm places candidate edges in a heap, sorted by increasing error. The placement of the new merged vertex is guided by volume preservation, volume optimization, and triangle shape optimization. Volume preservation tries to place the new vertex such that the volume of the local neighborhood is equivalent before and after the collapse operation. Volume optimization rewards edge collapses that make only small changes to the local volumes of individual neighboring triangles. Triangle shape optimization penalizes long and skinny triangles as compared to equilateral triangles. Other techniques are used in the special case of boundary edges, where local volume is undefined. The goal of the algorithm is to find three non-parallel constraint planes using these techniques. These three planes uniquely define a point, the position of the new merged vertex.

2.1.3.12 Controlled Simplification of Genus for Polygonal Models [El-Sana and Varshney 97]

This algorithm simplifies the topology of polygonal objects. In particular, it detects and tries to eliminate holes in the object. It accepts degenerate non-manifold geometry and executes fairly quickly. The algorithm assumes that after it eliminates holes that a topology preserving simplification algorithm is executed on the resulting object.

The algorithm first determines boundary edges for holes by detecting where there are sharp edges in the object. Next, these edges are grouped together to form chains of edges that represent the boundary of a single hole. Alpha prisms of a user-specified size are created around these edge chains. An alpha prism of an edge is the Minkowski sum, or convolution, of the edge and a sphere of the user-specified size. The polygonal geometry inside these alpha prisms is then re-triangulated by trying to minimize edge lengths. This process is usually all that is needed to eliminate holes. Some triangles, initially on the surface of the object, become interior faces of the simplified object. These triangles are detected and eliminated.

2.1.4 Summary of Algorithms

To the best of our knowledge, no one technique *simultaneously* exhibits all of the desired properties for a simplification algorithm as outlined in Section 1.3. Table 2.1 below categorizes the capabilities of algorithms described above. The *Non-manifold* column shows whether the algorithm accepts non-manifold meshes as input while *Attributes* indicates if the technique handles surface attributes during simplification. *Automatic* refers to whether the method is completely automated. *Geometric Error* and *Attribute Error* indicate if the algorithm keeps track of geometric error and surface attribute error respectively. *Drastic* refers to whether the technique is capable of drastic simplification such that a user can specify a target number of polygons. Whether the algorithm can change the topology of an object by joining unconnected regions together is noted in *Join Unconnected*. *Execution Speed* and *Memory Usage* refer to the efficiency of the technique in both of these categories.

Some columns of Table 2.1, such as *Non-manifold* and *Join Unconnected* use boolean values. A blank entry implies that the algorithm does not have that capability while an entry with a "✓" means that it does. Other columns, such as *Execution Speed*, can take on four values. A "–" equals low marks, an "="means average marks, a "+" indicates high marks, and a blank entry implies the category is not applicable to the method. Due to lack of information from the description of an algorithm, some entries are unknown, designated by a "?".

Note that [Garland and Heckbert 97] appears twice in Table 2.1; one for the basic algorithm and the other for when the distance threshold τ is specified for virtual edge selection. None of the algorithms shown in the table exhibit all of the desired properties of Section 1.3.

	Non-manifold	Attributes	Automatic	Geometric Error	Attribute Error	Drastic	Join Unconnected	Execution Speed	Memory Usage
[Cohen et al. 96]			✓	✓				=	?
[Cohen et al. 97]		✓	✓	✓	✓			=	=
[Cohen et al. 98]		✓	✓	✓	✓			=	=
[Eck et al. 95]			✓	✓				=	?
[El-Sana and Varshney 97]	✓		✓	✓		✓	✓	+	?
[Garland and Heckbert 97] without τ	✓		✓	✓		✓		+	=
[Garland and Heckbert 97] with τ	✓			✓		✓	✓	+	=
[Garland and Heckbert 98]	✓	✓	✓	✓	✓	✓		+	=
[He et al. 95]	✓		✓	✓		✓	✓	?	?
[Hoppe et al. 93]			✓	✓				=	?
[Hoppe 96]		✓		✓	✓			=	?
[Lindstrom and Turk 98]	✓		✓			✓		+	+
[Low and Tan 97]	✓		✓	✓		✓	✓	+	?
[Popovic and Hoppe 97]	✓	✓	✓	✓		✓	✓	–	?
[Ronfard and Rossignac 96]			✓	✓				=	?
[Rossignac and Borrel 93]	✓		✓	✓		✓	✓	+	?
[Schroeder et al. 92]			✓	✓				+	+
[Schroeder 97]	✓		✓	✓		✓		+	+
[Turk 92]			✓					?	?

Table 2.1: Properties of previous simplification algorithms. A "–" equals low marks, an "="means average marks, and a "+" indicates high marks. Due to lack of information from the description of an algorithm, some entries are unknown, designated by a "?".

2.2 Simplification of Static Polygonal Environments

This section highlights previous work dealing with the acceleration of rendering large static polygonal environments using polygonal simplification techniques. Some algorithms deal with traditional levels of detail within a scene graph representation. Others calculate a continuous level of detail representation that is able to be adaptively refined. Finally, a few methods deal with the problem of targeting a frame rate during the rendering of these large environments.

2.2.1 IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics [Rohlf and Helman 94]

This paper describes IRIS Performer, a toolkit for real-time visualization applications on Silicon Graphics machines. Polygonal environments are represented by scene graphs in Performer. The scene graph structure allows the user to store polygonal geometry at nodes as well as to specify LOD nodes that are used to select which children to render based on switching distances. Performer supports view-frustum culling through the use of a bounding volume hierarchy created for the scene graph. The application uses a multiprocessing scheme to work on different stages of rendering concurrently. Finally, IRIS Performer provides a target-frame-rate mode by using a *feedback loop*. Using the time it took to render the last couple of frames as feedback, the application coarsens or refines the objects in the scene for the upcoming frame. IRIS Performer is very general, and allows the user to provide custom designed callbacks to meet needs that are outside the scope of the toolkit. The basic toolkit does not automatically generate LODs and switching distances for objects in the environment. Output from a polygonal simplification algorithm must be inserted into the Performer scene graph, as was done in [Cohen et al. 96].

2.2.2 Adaptive Real-Time Level-of-Detail-Based-Rendering for Polygonal Models [Xia et al. 97]

This paper expands upon the progressive mesh representation presented in [Hoppe 96] by enabling selective refinement of the mesh. Instead of storing a list of decimation operations, the algorithm stores a tree of decimation operations. By traversing this tree selectively, it can adaptively simplify across the surface of an object. It uses image-space feedback, such as local illumination, screen-space projections, visibility culling, and silhouette boundaries to selectively refine objects being visualized. This method aids the visualization of complex individual objects that are close to the viewer and is less effective on large polygonal environments.

The algorithm creates a *merge tree* for each object being visualized. The merge tree is a tree of edge collapse operations. The edge collapses associated with the children of a parent

node cannot be performed until the parent node has collapsed its associated edges. The algorithm attempts to build well-balanced trees while collapsing edges in order of increasing length. This focus on well-balanced trees sometimes means sacrificing quality of approximation. At vertices, it stores distance errors and bounding cones for normal vectors in order to aid the selective refinement process. During each frame, the algorithm determines which vertices to display by traversing the merge tree according to the image-space criteria. It takes advantage of coherence to update these display vertices from frame to frame.

2.2.3 View-Dependent Simplification of Arbitrary Polygonal Environments [Luebke and Erikson 97]

This paper presents *hierarchical dynamic simplification*, or *HDS*, that uses *view-dependent simplification* to render large polygonal environments. View-dependent simplification enables adaptive simplification across the surface of objects. Using the vertex merge operation, HDS creates a hierarchical tree of vertices, called the *vertex tree*, which provides a continuous level of detail representation for the whole environment. HDS traverses this tree and renders any polygons in the scene that are *active*. Active polygons are determined by a variety of criteria such as a screen-space error threshold, silhouette preservation, and a triangle budget.

HDS can use any polygonal simplification algorithm to create the vertex tree for an environment. Its default algorithm partitions space hierarchically using an octree and then merges vertices within each octree node to the most important vertex in that space. Importance is calculated in a fashion similar to [Rossignac and Borrel 93]. Using this method, HDS is able to merge unconnected regions of the environment together to produce drastic approximations for groups of objects. This preprocessing method is very fast and tends to produce low quality approximations.

2.2.4 View-Dependent Refinement of Progressive Meshes [Hoppe 97]

This paper presents an elegant view-dependent simplification algorithm based on the author's previous work on progressive meshes [Hoppe 96]. Similar to [Luebke and Erikson

97], this paper traverses a hierarchy of vertices during visualization to produce a view-dependent representation of an object. The algorithm creates this vertex tree using the simplification algorithm presented in [Hoppe 96]. The paper uses surface orientation and screen-space geometric error for its view-dependent refinement criteria. It also simplifies portions of an object that are outside the view frustum. *Geomorphing* is used to morph between different states of the vertex tree, thereby providing a smooth transition between different approximations of the same object. The paper concentrates on simplifying individual objects. It does not merge polygons across different objects in the scene.

2.2.5 Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments [Funkhouser and Séquin 93]

This algorithm attempts to visualize complicated polygonal scenes at user-specified target frame rates. It treats the problem as an optimization problem by attempting to choose a level of detail and rendering method for each potentially visible object in the scene in order to display the best possible image within the time constraint. This method is able to produce a more uniform frame rate during visualization as compared to techniques using no LODs, LODs with a uniform error tolerance, and feedback loops.

Associated with each LOD of each object in a scene is a cost and benefit value. The cost of an LOD of an object is an approximation to the amount of time required to render the LOD from a particular viewpoint. This cost metric depends on a number of factors including the number of faces and vertices in the LOD, the approximate number of pixels it will cover in the final image, and the average performance of the graphics machine performing the visualization. The benefit of an object is an approximation of how much an LOD of an object will perceptually contribute to a scene from a particular viewpoint. This value depends mostly on the size of the LOD in the final image, but other factors are involved such as accuracy of representation, user-specified importance, viewer focus, motion blurring, and hysteresis.

The problem of choosing appropriate representations for each object in the scene to produce the best image given a target frame rate is equivalent to a version of the Knapsack

problem. Since this problem is NP-complete, the authors use a simple and greedy method to choose representations for objects. Each LOD of each object has an associated value, which is defined to be the object's benefit divided by its cost. LODs are added to the scene in descending order of value until the cost of the scene is equivalent to the frame-rate time constraint. By using coherence between subsequent frames, the algorithm minimizes the overhead of selecting which LODs to render.

2.2.6 Visual Navigation of Large Environments Using Textured Clusters [Maciel and Shirley 95]

This algorithm, building upon [Funkhouser and Séquin 93], presents another target frame-rate technique. Unlike [Funkhouser and Séquin 93], the algorithm is able to cluster multiple objects of a scene into one rendering primitive. Therefore, this algorithm uses a hierarchy of levels of detail for objects [Clark 76], instead of just levels of detail for each object. It represents clusters of distant objects using view-dependent textured faces. This method produces a more uniform frame rate during visualization as compared to brute force rendering.

Each LOD of each object, or each view-dependent textured cluster of objects has an associated cost and benefit value. The definitions of cost and benefit and the method used to select representations for objects are similar to the ones in [Funkhouser and Séquin 93], except that textured clusters can be chosen to represent numerous objects at once. In theory, the hierarchical levels of detail used by this algorithm should enable it to avoid looking at every object in the scene every frame. In practice, the benefit of clusters of objects changes in a view-dependent fashion, requiring the algorithm to check the benefit of a cluster's individual objects every frame. Therefore, just as in [Funkhouser and Séquin 93], every object is checked during the rendering traversal.

2.3 Simplification of Dynamic Polygonal Environments

Not much research has been done on simplification of dynamic environments consisting of rigid bodies. Systems such as IRIS Performer [Rohlf and Helman 94] that use traditional

LOD techniques can modify transformations in the scene graph to make objects move. These systems do not hierarchically group objects after movement in order to produce drastic approximations. Most research on dynamic environments has dealt with updating bounding volume hierarchies and spatial partitionings when objects move. Since our algorithm deals with this same problem, this section covers the previous work in this area.

2.3.1 Optimization of the Binary Space Partitioning Algorithm (BSP) for the Visualization of Dynamic Scenes [Torres 90]

Torres introduces a six-level structure called a dynamic BSP tree. The first level of the BSP tree is made of *divisor planes* that are defined by the user. The second set of planes, or *first range separating planes*, is used to separate objects completely from other objects. In some cases, complete separation is not possible so Torres uses a third set of planes to *sacrifice*, or split up objects. A fourth set of planes, called *wrapping planes*, is used to surround and prevent the splitting of objects from planes due to sacrificed objects. The algorithm tries to build balanced intra-object BSP trees using *halving planes*. These planes attempt to split polygons of an object into two equally dense halves. Finally, the sixth level of the structure includes polygons of sacrificed objects. When objects move, only portions of the BSP tree need to be updated. By choosing the top-level structure of the dynamic BSP tree carefully, a user can achieve efficient updates of the tree.

2.3.2 Computing Dynamic Changes to BSP Trees [Chrysanthou and Slater 92]

This paper presents efficient methods for updating BSP trees due to the movement, insertion, or deletion of objects in a scene. Movement of objects is handled by deleting them from the BSP tree and then reinserting them. The algorithm inserts objects into the tree using a filtering process. Insertion starts at the root of the BSP tree. If the polygon being inserted is in front of the polygon at the BSP tree node, then the algorithm recursively traverses the front space of the node. If it is behind the polygon at the node, the algorithm recursively descends down the back space of the node. If this process reaches a leaf node, then a new child node is created and the polygon inserted. If the polygon shares the same plane as the

polygon at a node, it is added to the node. Deletion of a polygon is split up into several cases. If the polygon exists in a leaf node without any other polygons, then the whole node is deleted. If the polygon shares a node with other polygons, then the polygon is deleted. If the polygon's node has only one child, then this child node can replace the node. Finally, if a polygon's node has two children, then the larger child replaces the node while the smaller child is filtered into this new node. These techniques allow incremental updates of a BSP tree that are faster than recomputing the tree from scratch.

2.3.3 Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality [Sudarsky and Gotsman 96]

This paper presents an efficient way to update an octree spatial partitioning after movement of objects in a scene. The central idea of the paper is that instead of deleting a moving object from the octree and then reinserting it, it attempts to efficiently update the octree locally. Therefore, if an object does not move a great distance, this algorithm will change the octree structure slightly or not at all. This type of local update is more efficient than traversing the tree once for deletion and once for insertion.

The paper also uses *temporal bounding volumes*, or *TBVs*, to bound not only the object's extents, but its possible locations in the future due to movement. Each TBV is associated with an object plus a certain span of time. This volume can be used to aid visibility queries. For example, if the TBV is completely outside the view frustum and the current time is within the volume's span of time, then the object can be ignored during rendering. Once the span of time for a TBV is over, it must be recalculated. TBVs can be easily constructed for objects that have a maximum velocity or have constrained movement. Choosing the optimal span of time for a TBV's existence is difficult, and adaptive methods are necessary.

3 SIMPLIFICATION OF STATIC POLYGONAL OBJECTS

This chapter presents our approach for simplifying static polygonal objects. Our method is general in that it works on models that contain both non-manifold geometry and surface attributes. It is automatic since it requires no user input to execute and it returns approximate error bounds used to calculate switching distances between levels of detail. Our algorithm, called *General and Automatic Polygonal Simplification*, or GAPS for short, uses an adaptive distance threshold, surface area preservation, and a quadric error metric to join unconnected regions of an object. Its name comes from this ability to "fill in the gaps" of an object. Our algorithm uses a new object space error metric that combines approximations of geometric and surface attribute error. GAPS efficiently produces high quality and drastic simplifications of a wide variety of objects.

The rest of this chapter is organized in the following manner. We provide an overview of our algorithm in Section 3.1. In Section 3.2, we describe symbology used in the chapter. Section 3.3 discusses the technical details of GAPS. Implementation details of GAPS are presented in Section 3.4 and performance results are shown in Section 3.5. Analysis of the running time of GAPS is provided in Section 3.6. We compare GAPS to other polygonal simplification algorithms in Section 3.7 and we conclude the chapter in Section 3.8.

3.1 Overview

As described in Section 2.1.4, no previous algorithm simultaneously exhibits all of the desired properties for a simplification algorithm as outlined in Section 1.3. We introduce a new simplification algorithm, called GAPS, that is general, automatic, runs quickly, and produces high quality and drastic approximations.

3.2 Symbology

Symbols used in Section 3 and their meanings are briefly described in Table 3.1. Note that subscripts *n*, *c*, and *t* refer to normal, color, and texture-coordinate point clouds respectively. For example, c_{n0} refers to the surface area associated with normal point cloud $\mathbf{c_n}$ and $A_t(\mathbf{p})$ is the average error of point **p** with respect to the texture-coordinate point cloud $\mathbf{c_t}$. Point clouds are described in detail in Section 3.3.3.

τ	Distance threshold used for selecting virtual edges
α, α^-, α^+	An amount of surface area
$\Delta(\mathbf{v})$	Error of an area-weighted error quadric of vertex **v** (as described in Section 2.1.3.9)
$S(\mathbf{v})$	Surface area associated with a vertex's quadric
$\Gamma(\mathbf{v})$	GAPS geometric error at vertex **v**
$a_0, a_1, \ldots$	Weights corresponding to surface areas
$\Pi(\mathbf{p})$	Squared error of point **p** in point cloud **c**
c_0	Surface area associated with point cloud **c**
$A(\mathbf{p})$	Average error of point **p** in point cloud **c**
$N(\mathbf{v})$	GAPS normal error at vertex **v**
$C(\mathbf{v})$	GAPS color error at vertex **v**
$T(\mathbf{v})$	GAPS texture-coordinate error at vertex **v**
$E(\mathbf{v})$	GAPS unified object space error at vertex **v**

Table 3.1: Brief descriptions of symbols used in this chapter.

3.3 General and Automatic Polygonal Simplification

In GAPS, we use the vertex merge operation along with the quadric error metric, as defined in Section 2.1.3.9, and extend upon this base in three ways to meet our simplification goals.

- We automatically and adaptively select the distance threshold τ in order to simplify topology.
- We do not allow vertex merges that change the local surface area greatly in relation to τ through the use of a technique called *surface area preservation*.

- Surface attributes are updated during the simplification process. We define a *unified error metric* that combines both geometric and surface attribute error to approximate an object space distance error for each vertex.

Like other algorithms, GAPS triangulates input objects as a preprocess step.

3.3.1 Automatic and Adaptive Selection of Distance Threshold

As stated in Section 2.1.3.9, [Garland and Heckbert 97] requires the user to specify a distance threshold τ that determines all virtual edge pairs for the rest of its execution. If two vertices are within this distance τ, then the virtual edge between them is inserted into the heap of candidate edges. We desire an algorithm that runs without user intervention. Furthermore, for some polygonal objects, specifying a single τ will result in too little or too many virtual edge candidates (see Figure 3.1). If there are not enough virtual edge candidates, the quality of the simplified object may suffer. If there are too many candidates, then the algorithm will execute slowly. Therefore, the process of manually picking a good τ is a difficult one.

GAPS does not use a single distance threshold. Instead, τ starts at a very small value and grows during the simplification process. Determination of the initial value τ and its subsequent growth happen automatically, without user intervention. τ can be thought of as the current scale at which GAPS is operating. The basic idea of the method is illustrated in Figure 3.2, Figure 3.3, and Figure 3.4.

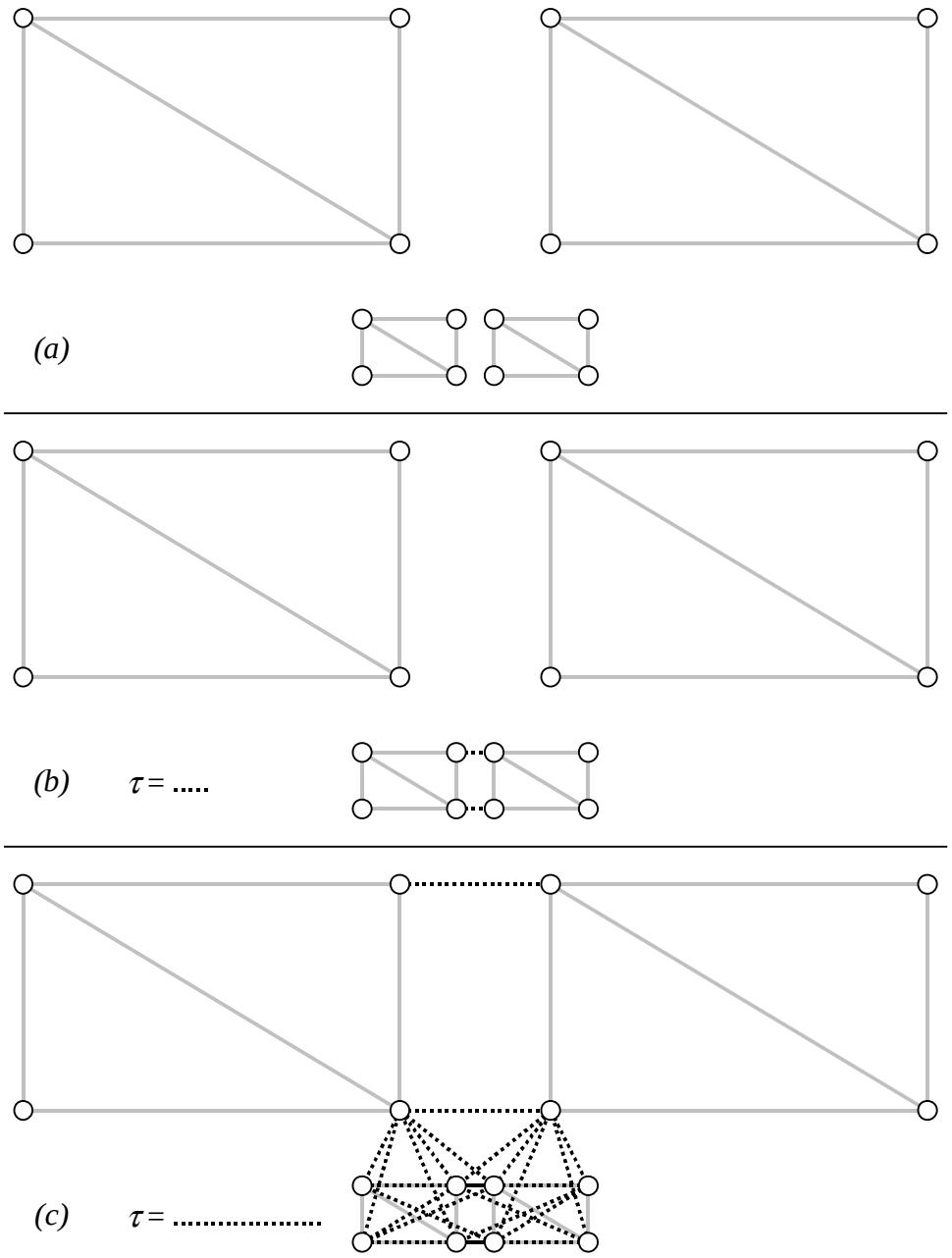


Figure 3.1: The problem with specifying a single distance threshold τ. *(a)* The top pair of rectangles is a scaled copy of the bottom pair. What is a good τ for this model? Ideally, τ should be independent of scale and simplify both pairs of rectangles identically. *(b)* Grey edges are real edges and black dotted edges are virtual edges. There are not enough virtual edges when τ is the shortest distance between the bottom rectangles. *(c)* There are too many virtual edges when τ is the shortest distance between the top rectangles.

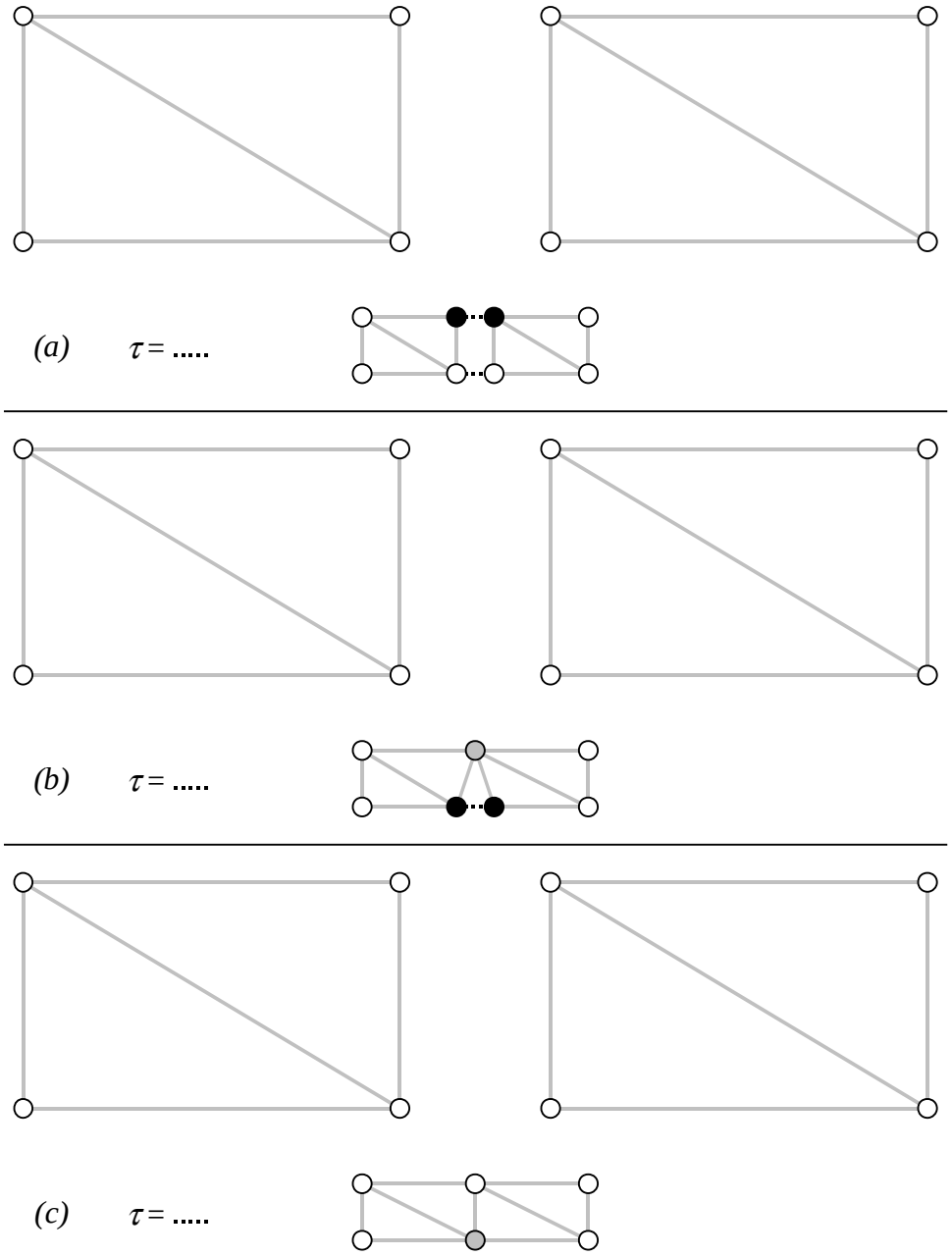


Figure 3.2: Simplification using an adaptive distance threshold. The polygonal geometry is the same as in Figure 3.1. Gray edges are real edges and dotted black edges are virtual edges. *(a)* The initial value of τ is the shortest distance between the bottom pair of rectangles. The black vertices joined by a virtual edge are the best pair to merge. *(b)* The gray vertex is the position of the newly merged vertex. Again, then next pair to be merged is joined by a virtual edge. *(c)* Because there are no more edges or virtual edges with length less than or equal to τ, GAPS must double τ.

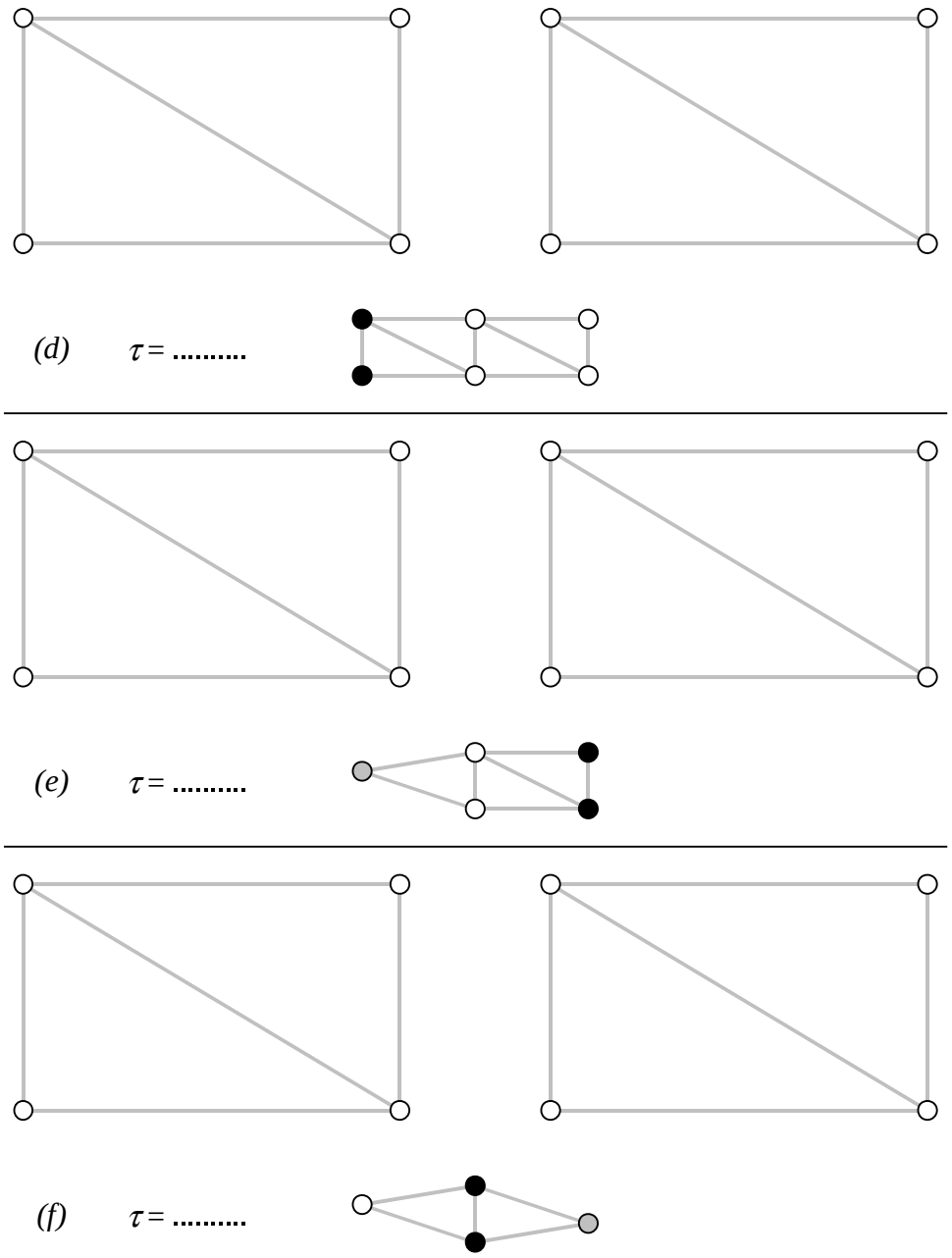


Figure 3.3: Continued from Figure 3.2. *(d)* τ has doubled. A normal edge is about to be collapsed. *(e)* GAPS selects another edge to collapse. *(f)* The bottom pair of rectangles will disappear due to the next vertex merge.

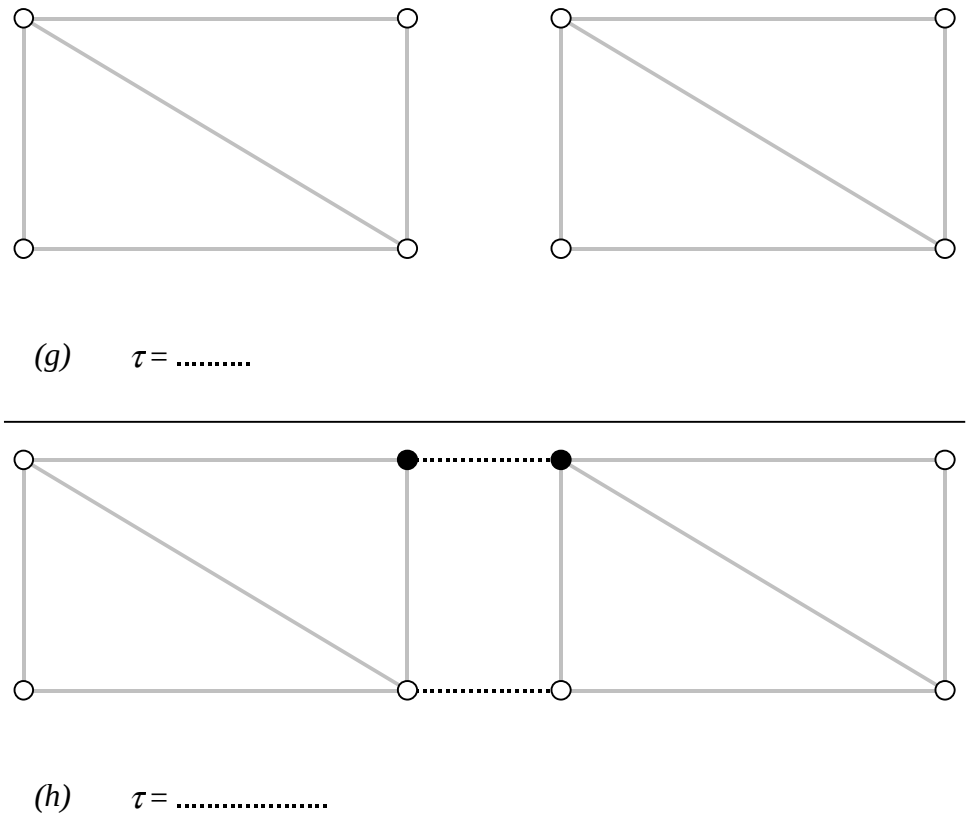


Figure 3.4: Continued from Figure 3.3. *(g)* Again, there are no more edges or virtual edges with length less that or equal to τ, so GAPS will double τ. The bottom pair of rectangles has disappeared because the rectangles were collapsed to a line. Lines are filtered from the object. *(h)* Note how the top pair of rectangles is being simplified in the same fashion as the scaled bottom pair. Growing τ while simplifying allows GAPS to achieve scale independence.

In order to find virtual edges within the distance threshold efficiently, GAPS partitions space to avoid $O(n^2)$ growth. Given the bounding box of the object, GAPS partitions it uniformly into cubes of side τ. To determine vertices within τ from a specific vertex, GAPS checks only vertices that lie in the same cube or corner-adjacent cubes of the vertex (see Figure 3.5). However, when τ is small, the number of cubes is too large to fit into memory. Therefore, our method represents this uniform grid by a hash table of size *h*, a prime number greater than the number of vertices in the object. Hashing collisions are resolved by storing the vertices in the same bin, a technique known as *chaining*. According to [Cormen et al. 94], Donald Knuth, in [Knuth 73], credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. This hashing scheme has been widely used to

solve the problem of sharing vertices of an object given a distance tolerance [Turk 94]. Suppose a vertex has coordinates [*x y z*], the object has a bounding box with minimum coordinates [$x^- y^- z^-$], and *a* and *b* are two prime numbers (we use 17 and 101 as in [Turk 94]). Then the hash function *f* for a vertex is

$$f(x,y,z) = \left(\left\lfloor (x - x^-)/\tau \right\rfloor \cdot a + \left\lfloor (y - y^-)/\tau \right\rfloor \cdot b + \left\lfloor (z - z^-)/\tau \right\rfloor\right) \bmod h$$

The initial guess for τ assumes that the vertices are distributed uniformly throughout the bounding box of the object. Suppose there are *v* vertices and the object has a bounding box with maximum coordinates [$x^+ y^+ z^+$]. Then τ is initialized such that

$$\frac{(x^+ - x^-)}{\tau} \cdot \frac{(y^+ - y^-)}{\tau} \cdot \frac{(z^+ - z^-)}{\tau} = v \Rightarrow \tau = \sqrt[3]{\frac{(x^+ - x^-)(y^+ - y^-)(z^+ - z^-)}{v}}$$

Using this starting value, GAPS partitions space into cubes of length τ using the hashing method described above. If an insertion of a vertex into the hash table causes a bin to contain more than a constant number of vertices, then GAPS deems τ to be invalid. This constant could be any reasonable value but we used 10 for all results in this thesis. If τ is not valid, then GAPS halves τ and rechecks its validity. GAPS repeats this process until τ becomes valid or falls below ε ($1e^{-10}$ in our implementation). Because GAPS uses a hash table, there is a chance that a particular object's vertices will cause numerous hashing collisions, resulting in τ never being valid. However, for the wide variety of objects we have tested, GAPS produces a reasonable initial value of τ. Objects that cause the starting value of τ to be less than ε would most likely benefit from a vertex sharing preprocess. Vertex sharing replaces groups of vertices that are within a specified distance threshold of each other with a single vertex.

Vertex pairs that are within the distance threshold τ are flagged as *local* pairs, whether they represent real or virtual edges. Vertices that are connected by real edges and separated by more than τ are deemed *global* pairs. If the error (described in Section 3.3.3) associated with the next pending vertex merge is greater than τ or there are no local pairs remaining, then τ doubles. When τ grows, a new hash table is created and any remaining vertices are re-inserted.

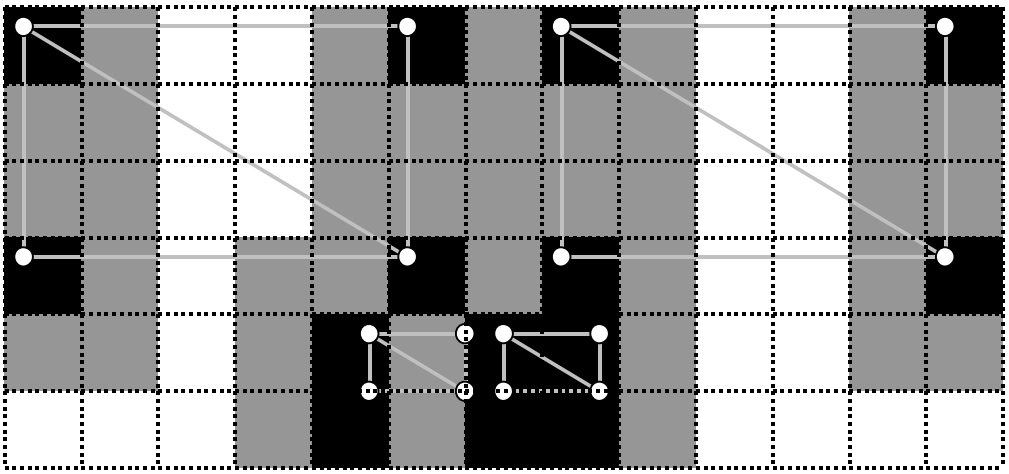


Figure 3.5: A two-dimensional example of uniform spatial partitioning to determine pairs of vertices within the distance threshold τ. The polygonal geometry is the same as in Figure 3.1. Darkly shaded squares contain at least one vertex. Lightly shaded squares hold no vertices but are corner-adjacent to darkly shaded squares.

3.3.2 Surface Area Preservation

In [Garland and Heckbert 97], vertex merges are performed in order of increasing error based on the error quadrics at each vertex (see Section 2.1.3.9). Although we feel these error quadrics provide a good approximation of geometric error at vertices, there are some cases where merging in order of increasing error leads to poor simplification choices. Because error quadrics measure distance error at vertices, pairs that are joined by short edges will most likely be merged first. However, if the merging of two vertices changes the local surface area drastically, the visual results are unappealing. For example, an isolated long and skinny triangle will disappear if any of its edges collapse. However, this disappearance is very noticeable even though the quadric error metric returns a small error for the operation (see Figure 3.22 and Figure 3.27). Through the use of surface area preservation, vertex merges that increase or decrease the local surface area greatly in relation to τ are not allowed. In many cases, this technique allows GAPS to join unconnected regions of objects, thereby producing higher quality simplifications. A demonstration of the difference that surface area

preservation makes during the simplification of a simple model is shown in Figure 3.6 and Figure 3.7.

To determine whether a vertex merge changes the local surface area drastically relative to τ, we first sum the surface areas of all faces adjacent to at least one of the vertices involved in the merge. This sum, α^-, is the local surface area before the merge operation executes. To calculate α^+, the local surface area after the merge, we sum the surface areas of all adjacent faces of the new merged vertex. Besides being the distance threshold for virtual edge selection, τ also determines the allowable change in surface area during a vertex merge. The allowable change in surface area is defined to be $\alpha = \pi\tau^2$, i.e., the area of a circle of radius τ. We chose this definition of α for three reasons. We wanted to produce a surface area related to τ using a simple geometric primitive. Since polygons are planar, and we total the surface areas of polygons to calculate α^- and α^+, we wanted this simple geometric primitive to be planar. Finally, for the objects we tested in Section 3.5, using the area of a circle of radius τ worked well.

If the change in surface area $|\alpha^+ - \alpha^-|$ is greater than α, then the merge is not allowed. If τ grows (see Section 3.3.1), previously disallowed merge operations may become legal. Intuitively, surface area preservation temporarily blocks merge operations that cause isolated collections of polygons to disappear. It thereby promotes the merging of unconnected regions of an object in order to produce higher quality simplifications. Figure 3.8 shows an example of determining whether vertex merges are allowed according to surface area preservation.

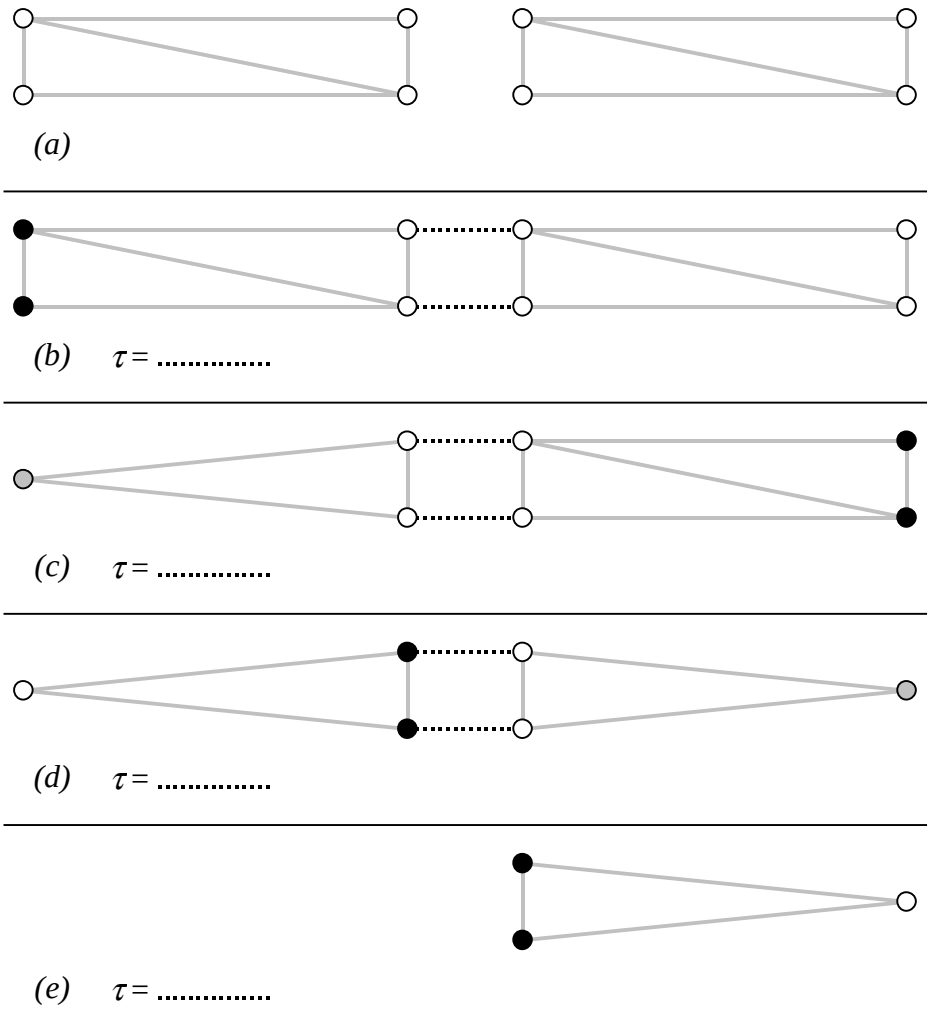


Figure 3.6: Simplification without surface area preservation. ***(a)*** **The original model.** ***(b)*** **Since the two black vertices are in close proximity, they are next to be merged.** ***(c)*** **The gray vertex shows the newly merged vertex.** ***(d)*** **Note that each vertex merge deletes a significant amount of surface area from the model.** ***(e)*** **The two rectangles disappear independently.**

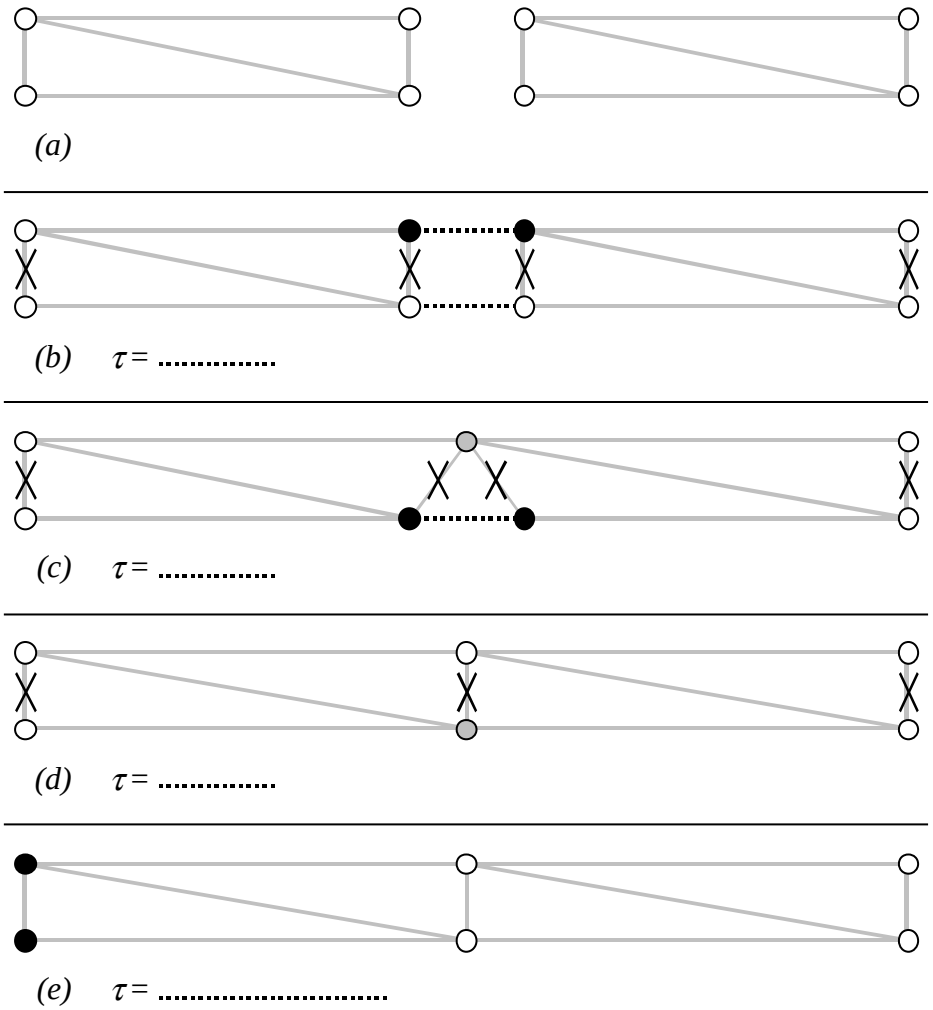


Figure 3.7: Simplification using surface area preservation. *(a)* The original model. *(b)* All of the edges marked with an "X" are not allowed to collapse because the operation would delete too much surface area from the model in relation to τ. The best remaining pair spans a virtual edge. *(c)* Again, the best vertices to merge collapse across a virtual edge. *(d)* There are no more edges or virtual edges with length less than or equal to τ that are allowed to collapse. Therefore, GAPS doubles τ. *(e)* The amount of surface area that can be deleted or inserted in a single vertex merge operation depends on τ. Since τ has grown, previously disallowed merges are now allowed. The two rectangles have joined and produced a higher quality simplification.

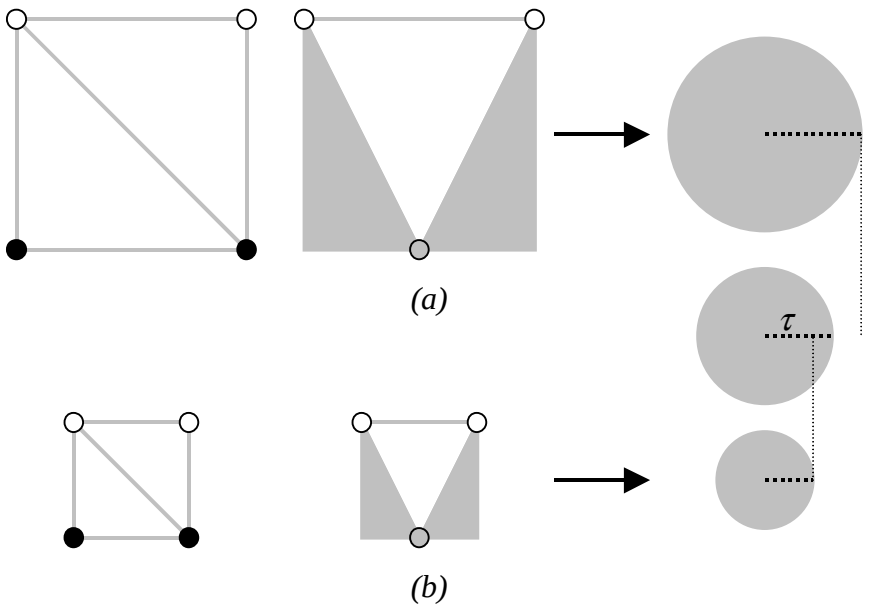


Figure 3.8: Determining if a vertex merge is allowable according to surface area preservation. On the left, the pairs of black vertices are potential merge candidates. In the middle, the shaded area represents the surface area change due to the merge. On the right, the shaded area has formed a circle with equivalent surface area. *(a)* This operation is not allowed since the area it changes shown in the top circle is greater than $\alpha = \pi\tau^2$, the middle circle. However, if τ doubles, then this operation becomes legal. *(b)* The merge is legal because the area in the bottom circle is less than the middle circle.

3.3.3 Attribute Handling and a Unified Error Metric

Many objects contain surface attributes such as normals, colors, and texture coordinates as well as polygonal geometry. Handling these attributes during simplification is an important, but difficult problem that involves many perceptual issues that are not well understood. We present a simple, efficient, and intuitive method that provides a reasonable object space error metric. Each type of error, whether it is due to polygonal geometry, normals, colors, or texture coordinates, is independently tracked during simplification.

3.3.3.1 Interpolating Attributes

We decouple geometric and attribute error and use quadric error metrics for determining merged vertex positions. Besides locating new vertices, we update any vertex

attributes that are adjacent to the edge being collapsed. If the pair of vertices forms a virtual edge, then only the polygonal geometry of the object and not its attributes are affected by the merge (see Figure 3.9). To merge two vertex attributes, we first find all faces that both disappear during the merge and contain these attributes at their corners. From these faces, we find the face nearest to the new merged vertex. We calculate the point on this nearest face that is closest to the merged vertex.

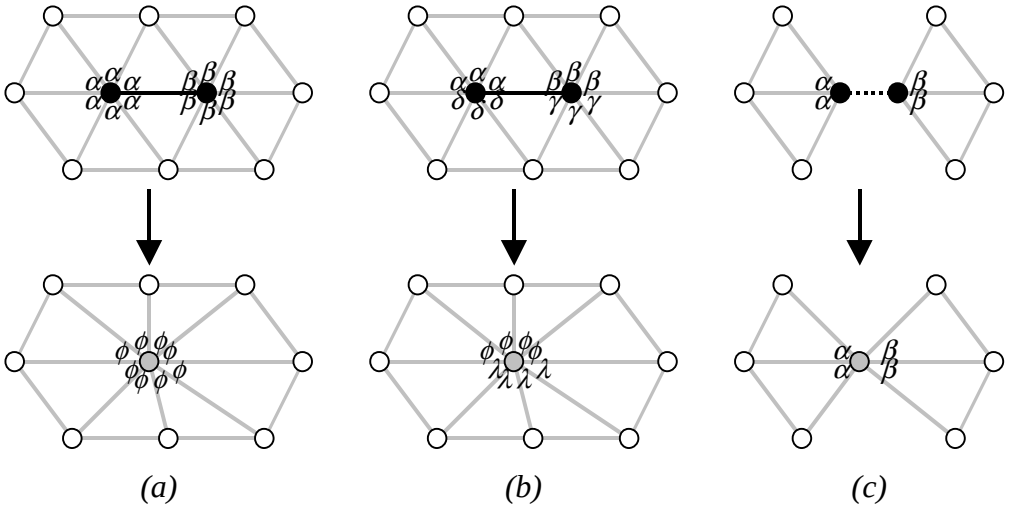


Figure 3.9: Some sample cases of attribute merging. The black vertices are next to be merged. The symbols at the corners of these vertices represent the attribute of the face at that corner. *(a)* A case involving continuous attributes. When the two vertices merge, the middle faces disappear and attributes α and β combine into ϕ. For example, if α is the color red and β is blue, then ϕ would be purple. *(b)* A case involving an attribute discontinuity. When the vertices merge, the middle faces disappear and the pairs of attributes α and β, and δ and γ combine into ϕ and λ, respectively. For example, if α is red, β is blue, δ is white, and γ is black, then ϕ would be purple and λ would be gray. *(c)* A case involving a virtual edge. After the pair merges, the attributes are unaffected.

Using this nearest point, we calculate the barycentric coordinates of the point on the nearest face. By weighting the attributes at the corners of the nearest face with these barycentric coordinates, we produce a new interpolated attribute (see Figure 3.10). Therefore, the positions of new vertices produced by error quadrics also directly determine the appearance of new attributes. This method is simple and efficient, but makes polygonal geometry inherently more important than surface attributes during simplification.

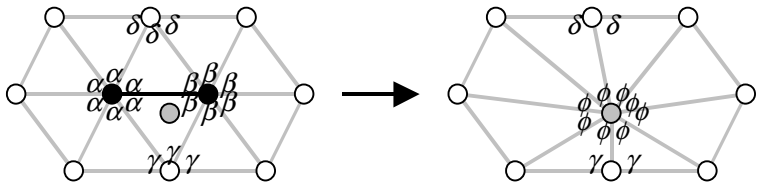


Figure 3.10: Interpolation of a new attribute. The black vertices are the next to be merged and the gray vertex is the best merge point according to error quadrics. We find the barycentric coordinates of the nearest point on the nearest face to the gray vertex to produce an interpolated attribute ϕ. In this case, $\phi \approx .3\alpha + .3\gamma + .4\beta$.

3.3.3.2 Geometric Error

We handle geometric error by associating an error quadric with each vertex (see Section 2.1.3.9). Initially, we insert the planes of a vertex's adjacent faces, weighted by the surface area of each face, into the vertex's error quadric. In addition, we keep track of the amount of surface area involved in each quadric. Whenever we merge vertices, the error quadric of the new vertex is simply the sum of the error quadrics of the merged vertices. The surface area involved in the new quadric is the sum of the surface areas involved in the merged quadrics. Assume $\mathbf{v} = [x\ y\ z\ 1]$ is the position of a vertex, n is the number of planes in the vertex's error quadric, $\mathbf{K}_0$, $\mathbf{K}_1$, $\mathbf{K}_2$, …, $\mathbf{K}_{n-1}$ are the fundamental error quadrics for each plane, and a_0, a_1, a_2, …, a_{n-1} are their associated surface area weightings. According to [Garland and Heckbert 97], the geometric error at this vertex is

$$\Delta(\mathbf{v}) = \mathbf{v}^T \left(\sum_{i=0}^{n-1} a_i \mathbf{K}_i \right) \mathbf{v}$$

This error is the area-weighted sum of squared distances between the vertex and its set of planes. The total surface area involved in an error quadric at a vertex is stored at that vertex and is defined to be

$$S(\mathbf{v}) = \sum_{i=0}^{n-1} a_i$$

We convert $\Delta(\mathbf{v})$ into a distance error in object space to obtain the final geometric error

$$\Gamma(\mathbf{v}) = \sqrt{\frac{\Delta(\mathbf{v})}{S(\mathbf{v})}}$$

$\Gamma(\mathbf{v})$ conservatively approximates the weighted average deviation error of the vertex from its set of planes. For a proof of this result, see the appendix. Thus, while $\Delta(\mathbf{v})$ measures a sum of squared distances (see Section 2.1.3.9), $\Gamma(\mathbf{v})$ estimates the average distance between a vertex and its set of planes.

3.3.3.3 Attribute Error Via Point Clouds

We opted to use a simple, efficient, but approximate method for computing error in attribute space. We make the assumption that values in attribute space are bounded. We independently track different types of attribute error due to normals, colors, and texture coordinates, by using *point clouds*. A point cloud is a collection of points enabling the efficient calculation of the sum of squared distances between a specified point and all the points in the cloud. Assume we are working in three-dimensional space, and there are n points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_{2, \ldots,} \mathbf{p}_{n-1}$, with corresponding weights a_0, a_1, a_2, …, a_{n-1}, in the cloud. Each point has coordinates $\mathbf{p}_i = [x_i\, y_i\, z_i]$, and $\mathbf{p} = [x\, y\, z]$ is the point of interest. Then the weighted sum of squared distances from the specified point to the cloud of points is

$$\Pi(\mathbf{p}) = \sum_{i=0}^{n-1} a_i \left\| \mathbf{p} - \mathbf{p}_i \right\|^2 = \sum_{i=0}^{n-1} a_i \left[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 \right] =$$
$$\sum_{i=0}^{n-1} a_i \left[\left(x^2 + y^2 + z^2\right) - 2\left(x \cdot x_i + y \cdot y_i + z \cdot z_i\right) + \left(x_i^{\,2} + y_i^{\,2} + z_i^{\,2}\right) \right] =$$
$$\left(\sum_{i=0}^{n-1} a_i \right) \left(x^2 + y^2 + z^2\right) - 2\left[\left(x \sum_{i=0}^{n-1} a_i x_i \right) + \left(y \sum_{i=0}^{n-1} a_i y_i \right) + \left(z \sum_{i=0}^{n-1} a_i z_i \right) \right] +$$
$$\sum_{i=0}^{n-1} a_i \left(x_i^{\,2} + y_i^{\,2} + z_i^{\,2}\right)$$

For each point cloud, we store a vector $\mathbf{c} = [c_0\, c_1\, c_2\, c_3\, c_4]$ where

$$c_0 = \sum_{i=0}^{n-1} a_i$$
$$c_1 = \sum_{i=0}^{n-1} a_i x_i$$
$$c_2 = \sum_{i=0}^{n-1} a_i y_i$$
$$c_3 = \sum_{i=0}^{n-1} a_i z_i$$
$$c_4 = \sum_{i=0}^{n-1} a_i \left({x_i}^2 + {y_i}^2 + {z_i}^2 \right)$$

Using the vector **c**, we can quickly calculate $\Pi(\mathbf{p})$ by

$$\Pi(\mathbf{p}) = c_0\left(x^2 + y^2 + z^2\right) - 2\left(c_1 \cdot x + c_2 \cdot y + c_3 \cdot z\right) + c_4$$

We define the error of a point **p** in respect to a point cloud **c** as

$$\mathrm{A}(\mathbf{p}) = \sqrt{\frac{\Pi(\mathbf{p})}{c_0}}$$

Intuitively, $\mathrm{A}(\mathbf{p})$ approximates the average error at point **p** with respect to cloud **c**.

This representation for a cloud of points makes combining clouds very efficient. Suppose we have two clouds $\mathbf{d} = [d_0\ d_1\ d_2\ d_3\ d_4]$ and $\mathbf{e} = [e_0\ e_1\ e_2\ e_3\ e_4]$, and we create a new cloud of points **c** which contains all points from both clouds. Then $\mathbf{c} = [(d_0 + e_0)\ (d_1 + e_1)\ (d_2 + e_2)\ (d_3 + e_3)\ (d_4 + e_4)]$, or $\mathbf{c} = \mathbf{d} + \mathbf{e}$.

For a given point cloud **c**, the minimum of $\Pi(\mathbf{p})$ occurs when

$$\frac{\partial}{\partial x}\Pi(\mathbf{p}) = 2c_0 x - 2c_1 = 0 \Rightarrow x = \frac{c_1}{c_0}$$
$$\frac{\partial}{\partial y}\Pi(\mathbf{p}) = 2c_0 y - 2c_2 = 0 \Rightarrow y = \frac{c_2}{c_0}$$
$$\frac{\partial}{\partial z}\Pi(\mathbf{p}) = 2c_0 z - 2c_3 = 0 \Rightarrow z = \frac{c_3}{c_0}$$

This minimum occurs at the weighted average of all the points included in the cloud.

Every vertex, besides containing an error quadric to measure geometric deviation, also keeps track of a point cloud per type of attribute used. Thus, if an object used normals, colors, and texture coordinates, the vertex would contain an error quadric, a normal point cloud, a color point cloud, and a texture-coordinate point cloud. To simplify the handling of attribute discontinuities, we initialize each point cloud with exactly one point. We calculate this point by performing a weighted average of attributes contained by faces adjacent to the vertex, where each attribute is weighted by the surface area of its containing face. Therefore, attributes in the local neighborhood affect the point cloud, not just attributes adjacent to the vertex. When we insert this point into the cloud, we weight it by the sum of surface areas of all the faces adjacent to the vertex. When two vertices merge, we combine their error quadrics plus their attribute point clouds. Since we initially store an average attribute in a point cloud, we do lose attribute discontinuity information at vertices. However, we accept this approximation for reasons of efficiency and the fact that attribute discontinuities are handled in part by the quadric error metric (see Section 3.4.2).

3.3.3.3.1 Normal Error

Normals for polygonal objects are usually initially calculated from the polygonal geometry itself. Therefore, since we are already handling error due to geometric deviation, it is unclear whether there is much benefit to tracking error due to normals. We believe that approaches such as [Cohen et al. 98] that bound the error of normal maps are superior to calculating the error of normals directly. Since normal maps are not widely available in hardware, GAPS includes the ability to calculate error due to normals. If normal maps are required for a polygonal object, the procedure for tracking the normal map coordinate error is exactly the same as for tracking texture coordinates (see Section 3.3.3.3.3).

Normal point clouds consist of three-dimensional points, as in the example point cloud description above. To find the error in normal space due to two vertices merging, we first combine the normal point clouds. Ideally, the interpolated normal resulting from this merge would be used to determine the error in normal space using the new normal point cloud. However, the operation of interpolating the normals (see Section 3.3.3.1) is too expensive to

be performed every time we rate the quality of a merge candidate. Therefore, we use an efficient, but approximate method in its place. For a combined cloud, we assume that the interpolated normal will be the one that creates the least amount of normal space error.

In order to approximate how much error in object space is due to normals, we use the degree of error in normal space plus the amount of surface area this affects. Since the Euclidean distance between two normals on the unit sphere is at most 2, we halve the average error $A_n(\mathbf{p})$ of the normal point cloud $\mathbf{c_n}$. This normalized value is within the range 0 to 1, indicating the degree of error in normal space.

The more surface area this normal error affects, the more visual error there will be due to simplification. Based on this idea, we transform normal error into object space by using a simple and approximate technique. Associated with the point cloud $\mathbf{c_n}$ is c_{n0}, the sum of all weights involved in the point cloud. However, these weights equal the total surface area represented by this merged vertex. Therefore, we use c_{n0} as the total surface area in which the error occurs, and the normalized error as the measure of error across this area. Suppose $\mathbf{p_n}$ is the minimal error point for point cloud $\mathbf{c_n}$. Then, the transformation from error in normal space to error in object space for a vertex $\mathbf{v}$ with normal point cloud $\mathbf{c_n}$ is

$$N(\mathbf{v}) = \sqrt{\frac{\frac{A_n(\mathbf{p_n})}{2} \cdot c_{n0}}{\pi}} = \sqrt{\frac{A_n(\mathbf{p_n}) \cdot c_{n0}}{2\pi}}$$

Intuitively, this equation starts with surface area c_{n0} and multiplies it by the measure of normal error. Next, it finds the radius of the circle whose surface area is equivalent to this multiplied area. This radius is defined to be the object space distance error due to normals (see Figure 3.11). We chose this radius as the distance error because of three reasons. We wanted to convert from a surface area to a simple geometric primitive. Since the surface area c_{n0} is a sum of surface areas of polygons, and polygons are planar, we wanted this simple geometric primitive to be planar. Finally, for the objects we tested in Section 3.5, using the radius of this circle as the distance error worked well.

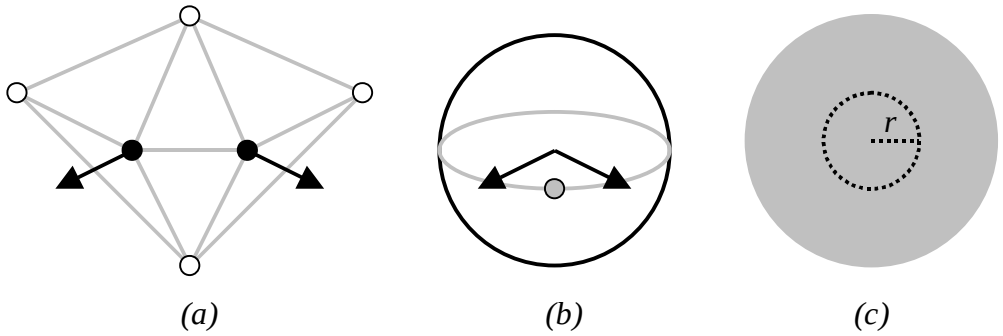


Figure 3.11: Conversion of normal space error to object space error. *(a)* The two black vertices are next to be merged. The average normals at the vertices are shown, depicting their initial normal point clouds. *(b)* The combined normal point cloud in normal space when the vertex pair is merged. The gray point represents the point of minimum error according to this point cloud. The average distance between this point and the normals in the point cloud is approximately 0.4. Dividing by 2 results in the normalized error of 0.2. *(c)* The surface area of the adjacent faces of the two vertices being merged form the shaded circle above. This area is multiplied by 0.2 to obtain the final affected surface area, represented by the smaller circle with radius *r*. *r* is defined to be the object space distance error due to normals. In other words, *r* is a distance error in $\Re^3$.

We have noticed when working with normals on several objects that they change rapidly during simplification as compared to other surface attributes. The normalized error is a value between 0 and 1. Thus, if we square the normalized error, then it *decreases* and remains bounded between 0 and 1. If we perform a square root on the normalized error, then it *increases* and remains bounded between 0 and 1. Therefore, in order to decrease the sensitivity of error due to normals, we square the normalized error in normal space and modify $\mathrm{N}(\mathbf{v})$ to be

$$\mathrm{N}(\mathbf{v}) = \sqrt{\frac{\left(\frac{\mathrm{A_n}(\mathbf{p_n})}{2}\right)^2 \cdot c_{n0}}{\pi}} = \sqrt{\frac{(\mathrm{A_n}(\mathbf{p_n}))^2 \cdot c_{n0}}{4\pi}}$$

3.3.3.3.2 Color Error

We assume colors have four components: red, green, blue, and alpha and that these components are within the range 0 to 1. Therefore, we use four-dimensional point clouds to

keep track of color. We represent a four-dimensional point cloud in almost the same way as a three-dimensional one, except we keep track of six values, rather than five.

Since the Euclidean distance between two colors in color space can be at most

$$\sqrt{(1-0)^2+(1-0)^2+(1-0)^2+(1-0)^2}=2,$$

we normalize color error by this value. Suppose $\mathbf{p_c}$ is the minimal error point for color point cloud $\mathbf{c_c}$. Then, the transformation from error in color space to error in object space for a vertex $\mathbf{v}$ with normal point cloud $\mathbf{c_c}$ is

$$\mathrm{C}(\mathbf{v})=\sqrt{\frac{\frac{\mathrm{A_c}(\mathbf{p_c})}{2}\cdot c_{c0}}{\pi}}=\sqrt{\frac{\mathrm{A_c}(\mathbf{p_c})\cdot c_{c0}}{2\pi}}$$

It has been our experience that the visual effect of slight color changes is noticeable. Therefore, we enlarge the error due to color by taking the square root of the normalized error in color space and changing C($\mathbf{v}$) to

$$\mathrm{C}(\mathbf{v})=\sqrt{\frac{\sqrt{\frac{\mathrm{A_c}(\mathbf{p_c})}{2}}\cdot c_{c0}}{\pi}}=\sqrt{\frac{\sqrt{2\cdot\mathrm{A_c}(\mathbf{p_c})}\cdot c_{c0}}{2\pi}}$$

3.3.3.3.3 Texture-Coordinate Error

We assume two-dimensional texture coordinates for polygonal objects, so we use two-dimensional point clouds to track texture-coordinate error. We represent a two-dimensional point cloud in almost the same way as a three-dimensional one, except we keep track of four values, rather than five. Since we rely on attribute values being bounded, we assume that each dimension of a texture coordinate is within the range 0 to 1. Therefore, clamped or repeated textures that use texture coordinates outside the range of 0 to 1 are not handled directly using this method. We can transform an object with clamped or repeated textures to an object that meets our texturing requirements. We also assume that the entire texture is applied over the model. If only a subset of the texture is used, then our approximation will return erroneous results. We can fix this problem by clipping the texture to its used subset.

Since the Euclidean distance between two texture coordinates in texture-coordinate space is at most

$$\sqrt{(1-0)^2+(1-0)^2}=\sqrt{2},$$

we normalize texture-coordinate error by this value. Suppose $\mathbf{p_t}$ is the minimal error point for texture-coordinate point cloud $\mathbf{c_t}$. Then, the transformation from error in attribute space to error in object space for a vertex $\mathbf{v}$ with texture-coordinate point cloud $\mathbf{c_t}$ is

$$\mathrm{T}(\mathbf{v})=\sqrt{\frac{\frac{\mathrm{A_t}(\mathbf{p_t})}{\sqrt{2}}\cdot c_{t0}}{\pi}}=\sqrt{\frac{\sqrt{2}\mathrm{A_t}(\mathbf{p_t})\cdot c_{t0}}{2\pi}}$$

3.3.3.4 Unified Error Metric

We define the average error at a vertex $\mathbf{v}$ with normal point cloud $\mathbf{c_n}$, color point cloud $\mathbf{c_c}$, and texture point cloud $\mathbf{c_t}$, to be a weighted average of all of its component errors. In other words,

$$\mathrm{E}(\mathbf{v})=\frac{\mathrm{S}(\mathbf{v})\cdot\Gamma(\mathbf{v})+c_{n0}\cdot\mathrm{N}(\mathbf{v})+c_{c0}\cdot\mathrm{C}(\mathbf{v})+c_{t0}\cdot\mathrm{T}(\mathbf{v})}{\mathrm{S}(\mathbf{v})+c_{n0}+c_{c0}+c_{t0}}$$

If a particular vertex does not have a specific attribute associated with it, then that attribute is not included in the average. For example, if a vertex in the object does not have color information, then the final error would just be the average of the polygonal geometry, normal, and texture-coordinate error. We use E($\mathbf{v}$) to determine the best candidate vertex pair to merge. Since E($\mathbf{v}$) is an approximation to the average error, and not the maximum error, we cannot directly use it to calculate switching distances for LODs. However, we have found that by scaling the average E($\mathbf{v}$) of all vertices by a constant (we use 10), we can automatically generate reasonable switching distances. We empirically chose this scaling constant because it almost always overestimated the maximum error for the models we tested. Details on the results of this error approximation are shown in Section 3.5.4.

3.4 Implementation

We have implemented GAPS using C++, GLUT, and OpenGL. The code is portable across PC and SGI platforms. GAPS has produced visually pleasing results on a variety of datasets including scanned, terrain, radiositized, and CAD objects.

3.4.1 Generality

GAPS handles all polygonal objects, whether they are manifold meshes or unorganized lists of polygons. As a preprocess, objects are triangulated and then represented by sharing vertices and calculating normals. Although these operations are not necessary, GAPS can take advantage of the increase in the resulting topology information to produce better-looking simplifications more efficiently.

Objects consist of vertices, triangular faces, vertex attributes, and face attributes. Vertex attributes are colors, normals, and texture coordinates. Face attributes are colors, materials, and textures. This categorization is equivalent to the one described in [Hoppe 96] and [Hoppe98], except that Hoppe names them *scalar* and *discrete* attributes respectively. We use a simple internal representation for objects where faces consist of three corners. Each corner has a pointer to a vertex and a vertex attribute. Vertices consist of a three-dimensional point plus 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 used GAPS on large CAD models composed of non-manifold and degenerate geometry.

3.4.2 Discontinuities

As in [Garland and Heckbert 97], additional perpendicular constraint planes are added to the error quadrics of vertices that lie on boundary edges or sharp edges. Unlike [Garland and Heckbert 97] that assigns a large penalty to these planes, we weight them using the surface area of the face that contains the boundary edge or sharp edge. Similarly, if there is a vertex attribute or face attribute discontinuity occurring at an edge, constraint planes are inserted as if the edge was a boundary edge. It has been our experience that these extra

constraint planes work very well at preserving both geometric and attribute discontinuities during simplification.

3.4.3 Preventing Mesh Inversion

Vertex merges can cause faces to flip orientation and fold over one another. As in [Garland and Heckbert 97], we prevent these mesh inversions from occurring by comparing the normals of faces before and after a vertex merge operation. If any of the normals flip, the merge is not allowed.

3.4.4 Candidate Merge Pairs

We use a heap containing candidate merge pairs to merge vertices in order of increasing error. Candidate pairs consist of pointers to two vertices, a bit flag, a merged vertex position if the candidate is chosen, the error involved in the merge operation, and the index of the pair in the pair heap.

The bit flag specifies whether the pair is active, legal, local, virtual, or dirty. A pair is inactive if a merge operation eliminates the pair from consideration. A pair is legal if it does not cause any mesh inversions and meets the requirements of surface area preservation (see Sections 3.3.2 and 3.4.3). Illegal pairs are never inserted into the pair heap. A pair is local if the distance between the vertices is less than or equal to the current distance threshold τ as defined in Section 3.3.1. A pair is virtual if the edge connecting the two vertices of the pair is virtual. A pair is dirty if it is in need of recalculation (dirty pairs are introduced in [Cohen et al. 97]). After merging a pair, we recalculate the error and legality for each pair that contains the new merged vertex. However, because the merge operation changes the polygonal geometry of faces surrounding the new merged vertex, we must also check for mesh inversion and surface area preservation in pairs that have any vertex adjacent to the new vertex. Most of the time, the error and legality of these pairs will not change appreciably due to a remote vertex merge. Therefore, instead of slowing down the algorithm by recalculating the error and legality of these pairs, we mark them as dirty. When a dirty pair is on top of the heap, we

recalculate its error and then reinsert it if it is legal. Marking pairs as dirty is a technique to increase the efficiency of GAPS while not reducing its apparent quality.

3.4.5 Main Loop

After initialization, the main loop of GAPS executes as follows:

- Check if τ needs to be doubled. Doubling occurs when either there are no more legal and local pairs in the heap or when the pair on top of the heap has error greater than τ. When GAPS doubles τ, we recalculate the error and legality of all active pairs.
- If the pair on top of the heap is dirty, recalculate its error and reinsert it if legal. Repeat this step of checking the top pair.
- Extract the top pair from the heap. Delete the two vertices in the pair from the hash table with grid cells of size τ. Insert the new vertex into the hash table to find virtual edges incident to it. Add quadrics and point clouds of the two vertices and store this information in the new vertex.
- Determine which pairs need to be updated or deleted due to the merge. Mark remote pairs as dirty.
- Repeat until a specified number of vertices or faces remain, or an error threshold has been reached.

3.5 Results

In this section, we describe the performance of GAPS on various polygonal models. We review the speed at which GAPS executes and its memory efficiency, as well as a discussion of its approximation of geometric error.

3.5.1 Execution Speed

All of our timing results were gathered on a 195 MHz R10000 SGI Infinite Reality2 with 2 gigabytes of main memory. Table 3.2 shows timing comparisons between GAPS and QSlim, Michael Garland's publicly available simplification code [Garland 97]. These timings include both setup time, such as initializing the error quadrics and determining virtual edge pairs, plus simplification time. For QSlim, we specified options *–m* (preserve mesh quality), *–a* (area weight error quadrics), and *–B 1000* (penalize boundary edges with weight 1000). Assuming these settings, QSlim runs about twice as fast as GAPS on average.

The Bunny object is the traditional scanned mesh from Stanford University, Head is a texture-mapped scanned mesh, and Sierra is a terrain model (see Figure 3.25, Figure 3.16, and Figure 3.30, respectively). If the user knows that an object consists of one connected mesh with no holes and genus zero, then there is little point in using a distance threshold for the selection of virtual edges. Therefore, there is an option in GAPS to turn off virtual edge selection. The timings for GAPS using this option are shown in Table 3.2 as well. Even though GAPS is still handling attributes, the results are comparable to QSlim.

Only objects that contain disconnected regions of polygons in close proximity benefit from using τ. Rotor, Econ, and ShDivWest, and Chamber all fall under this type of object (see Figure 3.14, Figure 3.22, Figure 3.27, and Figure 3.19, respectively). Rotor, Econ, and ShDivWest are CAD models while Chamber is a radiositized object. Note that if we specify a reasonable τ for QSlim, then GAPS, with its adaptive distance threshold, outperforms QSlim on Econ and ShDivWest.

3.5.2 Memory Usage

GAPS needs to store extra information at vertices and faces during simplification. For each vertex, we store an error quadric, a 4x4 symmetric matrix, plus the total area weight of the quadric, a total of 11 floating-point values. In addition, we store point clouds for normals, colors, and texture coordinates, shown in Section 3.3.3.3. These point clouds total 5, 6, and 4 floating-point values respectively. Finally, we store a list of pair heap indices associated with the vertex. This list of indices allows efficient updates of specific pairs in the pair heap.

Therefore, assuming GAPS is tracking normals, colors, and texture coordinates, each vertex consists of at least 26 extra floating-point values, plus an integer per pair heap index. For each face, we keep track of its associated plane, its surface area, and whether it is still active. A face is active if it has not yet disappeared during the simplification process. The plane and surface area information are used to make GAPS more efficient when checking for mesh inversion and surface area preservation. Therefore, each face entry consists of 5 extra floating-point values plus a boolean value. GAPS also stores a hash table that contains pointers to vertices and a heap of candidate pairs. Therefore, this algorithm uses a lot of memory.

Object	**Vertices**	**Triangles**	**QSlim Time (secs.)**	**Qslim Time using τ (secs.)**	**GAPS not using τ (secs.)**	**GAPS (secs.)**
Rotor	2328	4736	1.45	1.59	1.56	2.19
Head	4925	9580	2.89	3.63	3.51	4.63
Chamber	5685	10423	9.48	17.09	3.50	6.98
Econ	10032	23556	7.61	68.60	7.72	25.22
Bunny	34834	69451	27.23	30.15	27.91	38.72
ShDivWest	65245	141180	42.02	170.55	47.35	147.49
Sierra	81920	162690	67.18	182.54	60.41	193.09

Table 3.2: Simplification timings for various models running on an SGI Infinite Reality2 with a 195 MHz R10000 processor and 2 gigabytes of main memory. The default settings used for QSlim were to preserve mesh quality, area weight quadrics, and penalize boundary edges. To choose τ for QSlim, we used 1% of the maximum bounding box dimension of the object being simplified. The "GAPS not using τ" column signifies that attribute error was handled, but that no virtual edges were considered and no surface area preservation was performed.

3.5.3 Geometric Error Versus QSlim

There is no standard method of comparing the output of two simplification algorithms, especially if the simplification involves attributes. However, a natural way of calculating the geometric error between the original object and its simplified version is to first point sample both objects. For each point, find the distance to the closest point on the other object. If we want to find the average geometric error, we average these distances. If we wish to calculate the maximum geometric error, we take their maximum. This last calculation approximates the Hausdorff error metric as described in [Rossignac 97].

Our results, shown in Figure 3.12, show that the average and maximum geometric error of GAPS and QSlim are roughly the same. These results are not surprising since GAPS uses almost the same geometric error bound as QSlim. Note that for these results, GAPS used an adaptive τ, surface area preservation, and attribute handling. However, only the geometric error is shown in Figure 3.12 as there is no standard error metric for surface attributes in the simplification literature. The graph of the Chamber model shows that QSlim does a better job of simplifying polygonal geometry initially, but that GAPS catches up quickly. However, the simplified model produced by GAPS is visually superior because it handles attributes (see Figure 3.20). [Lindstrom and Turk 98] presents geometric error comparisons between QSlim and various other simplification algorithms. If we were to compare GAPS versus these algorithms in terms of geometric error, the results would almost be equivalent to substituting QSlim in its place.

Our goal was to show that adaptively changing τ and preserving surface area decrease the geometric error of some simplified objects as compared to QSlim. Econ and ShDivWest are objects where an adaptive τ and surface area preservation seem to make a large visual difference in the simplified output (see Figure 3.23 and Figure 3.28). The ShDivWest graph shows that GAPS mainly outperforms QSlim when the simplified object is less than a thousand faces. This behavior matches visual results where GAPS seems to produce better-looking drastic simplifications than QSlim for ShDivWest. However, GAPS appears to produce better looking drastic simplifications for Econ as well, but this difference is not reflected in the graph in Figure 3.12. A possible explanation for this discrepancy comes from

[Rossignac 97] that gives examples where the error metric we use does not do a good job of measuring shape similarity.

3.5.4 Geometric Error Approximation

It would be useful to know how close the approximate average geometric error (see Section 3.3.3.2) is to the more precise average geometric error described in Section 3.5.3. Our results in Figure 3.13 show cases where the approximate average error underestimates and overestimates the more precise average error. In order to approximate the maximum error, we scale the average error of all vertices by a constant (we use 10 in our current implementation). Note that these approximate maximum values, also shown in Figure 3.13, also underestimate and overestimate the more precise maximum error. These more precise error bounds can be calculated if an application requires them, but at great computational cost. Note that this discussion avoids the issue of our attribute error handling since there is no standard error metric for colors, normals, and texture coordinates. In practice, this approximation of maximum error produces reasonable switching distances for LODs. In Chapter 4, we discuss how to calculate a switching distance based on this approximate error.

In Figure 3.13, *Avg. App.* refers to the approximation of the average geometric error. *Max. App.* shows the approximation of the maximum error of the simplified object. *Avg. Prec.* is the more precise average geometric error. *Max. Prec.* refers to the more precise maximum geometric error.

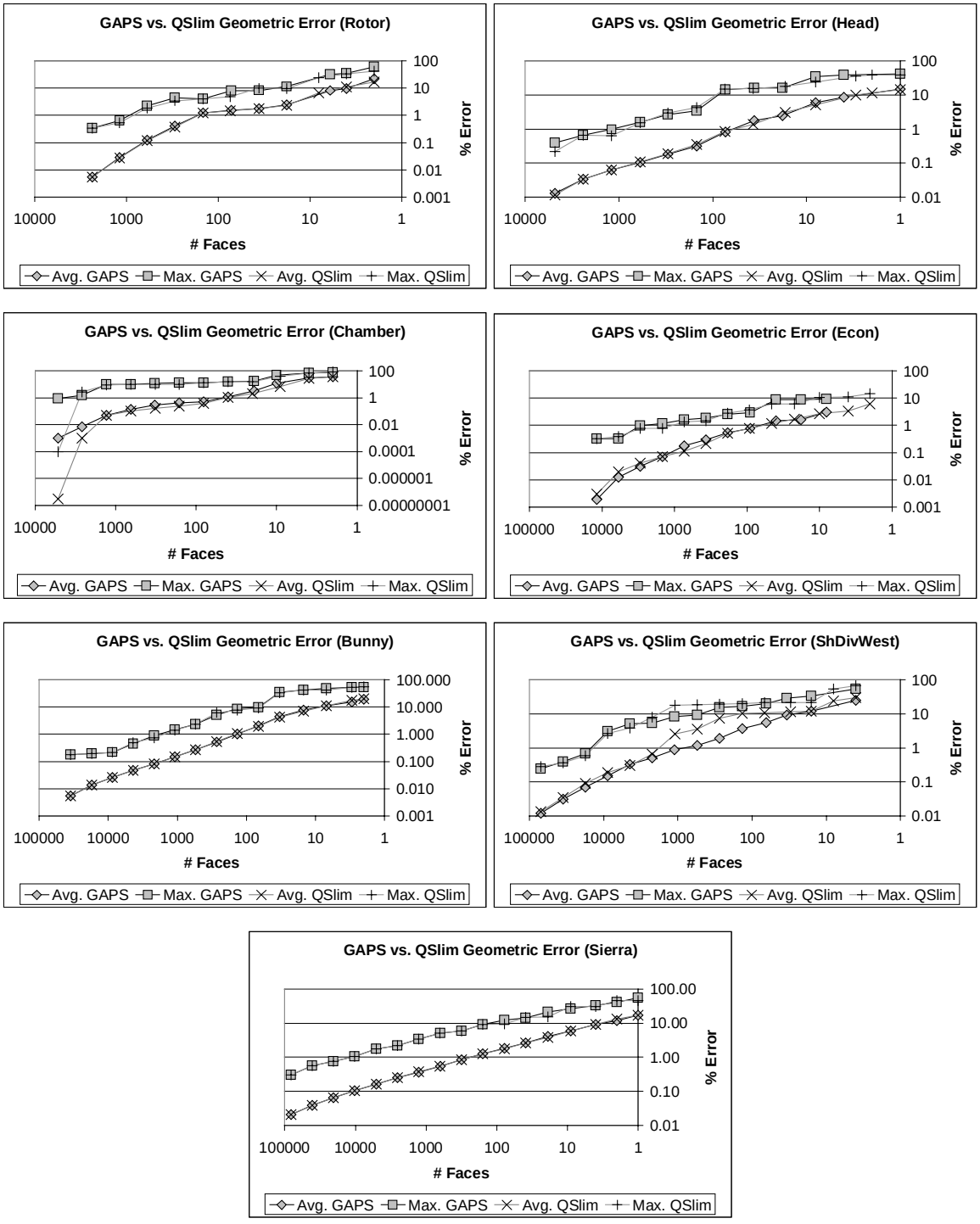


Figure 3.12: Geometric error comparison, as detailed in Section 3.5.3, between GAPS and QSlim on various objects. The percentage error is in terms of the maximum dimension of the object's bounding box.

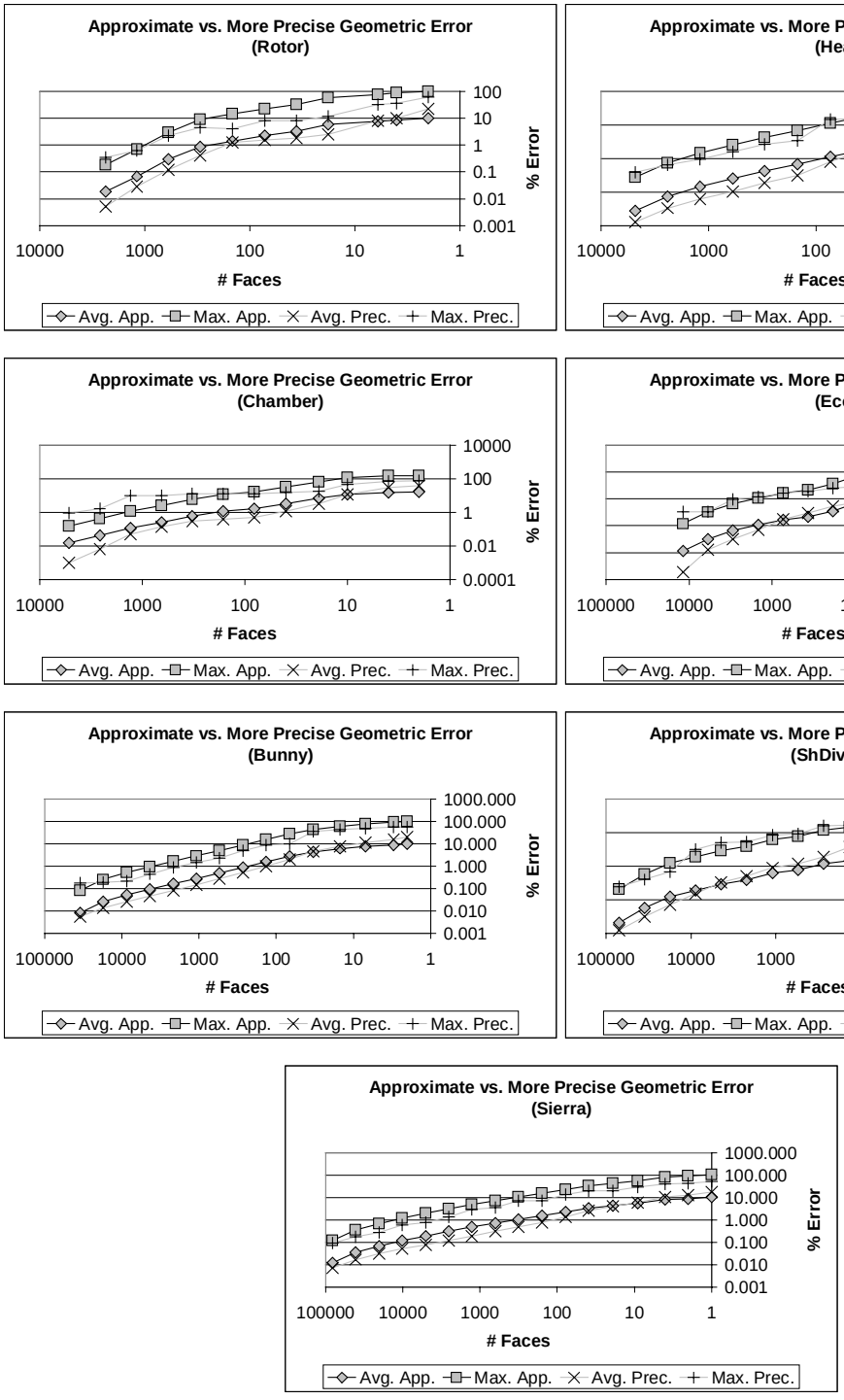


Figure 3.13: Comparison between the approximate geometric error used by GAPS and a more precise geometric error calculated during the simplification of various objects. The percentage error is in terms of the maximum dimension of the object's bounding box.

3.5.5 Visual Comparison

Although the results in Section 3.5.3 do not clearly show a distinction between the output of GAPS and using the quadric error metric alone, a visual comparison of the two techniques on more intricate objects, such as Econ (Figure 3.22 and Figure 3.23) and ShDivWest (Figure 3.27 and Figure 3.28), does show a difference.

Figure 3.14: The original Rotor object and its LODs created by GAPS. From left to right these LODs consist of 4,736 faces, 1,184 faces, 296 faces, and 72 faces.

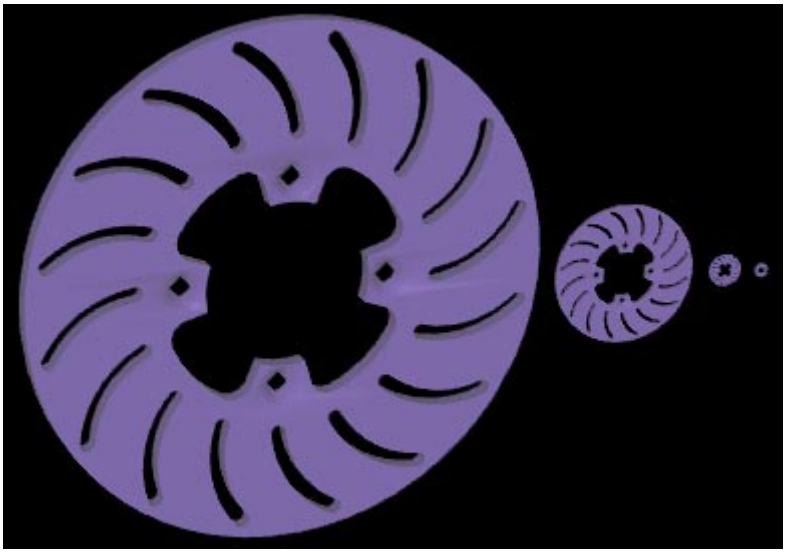


Figure 3.15: Switching distances for LODs of the Rotor object if we allow 1 pixel of error according to our approximate error metric described in Section 3.3.3. From left to right these LODs consist of 23,681 faces, 1,184 faces, 592 faces, and 296 faces. The original object consists of 4,736 faces.

Figure 3.16: The original texture-mapped Head object and its LODs created by GAPS. From left to right these objects consist of 9,580 faces, 2,395 faces, 597 faces, and 148 faces.

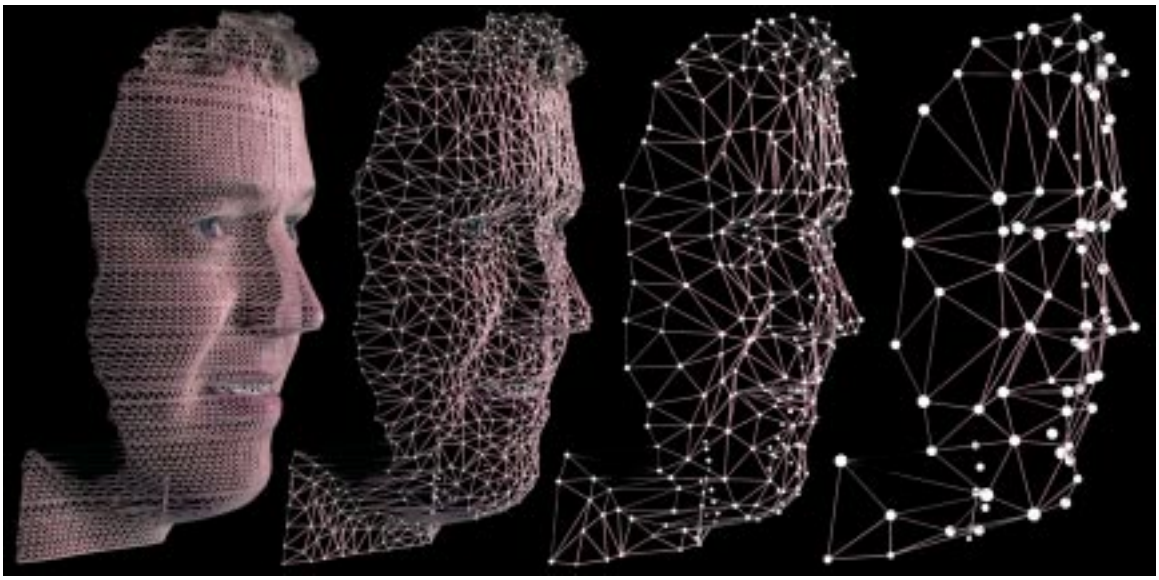


Figure 3.17: The approximate error bounds reported by GAPS for the same LODs as in Figure 3.16. The radii of the spheres are the approximate errors at the vertices they enclose as defined by the unified error metric in Section 3.3.3.4. These error bounds are used to automatically determine switching distances for LODs. From left to right these objects consist of 9,580 faces, 2,395 faces, 597 faces, and 148 faces.

Figure 3.18: Switching distances for LODs of the Head object if we allow 1 pixel of error. From left to right these LODs consist of 4,789 faces, 2,395 faces, 1,196 faces, and 597 faces. The original object consists of 9,580 faces.

Figure 3.19: LODs for the Chamber object from a top-down view created by GAPS when error due to attributes is ignored. Note how the shadow and lighting information of this radiositized model rapidly disappears as the simplification proceeds. From left to right, these LODs consist of 10,423 faces (the original object), 9,119 faces, 7,817 faces, and 5,210 faces.

Figure 3.20: LODs for the Chamber object created by GAPS using the unified error metric described in Section 3.3.3.4. The results are superior to the LODs in Figure 3.19 in terms of color preservation. However, notice that the polygonal geometry of the lamp, i.e., the yellow ring, is preserved better in Figure 3.19. From left to right these LODs consist of 10,423 faces (the original object), 9,120 faces, 7,817 faces, and 5,211 faces.

Figure 3.21: Switching distances for LODs of the Chamber object if we allow 1 pixel of error. From left to right these LODs consist of 7,817 faces, 5,211 faces, 2,605 faces, and 1,302 faces. The original object consists of 10,423 faces.

Figure 3.22: LODs for the Econ object created by GAPS with no distance threshold and no surface area preservation. Note how the pipes thin during simplification. From left to right these LODs consist of 23,556 faces (the original object), 5,888 faces, 1,472 faces, 368 faces, and 92 faces.

Figure 3.23: LODs for the Econ object created by GAPS using an adaptive distance threshold and surface area preservation. The pipes join together at the latter stages of simplification, preserving more of the surface area of the object as compared to Figure 3.22. From left to right these LODs consist of 23,556 faces (the original object), 5,888 faces, 1,470 faces, 368 faces, and 90 faces.

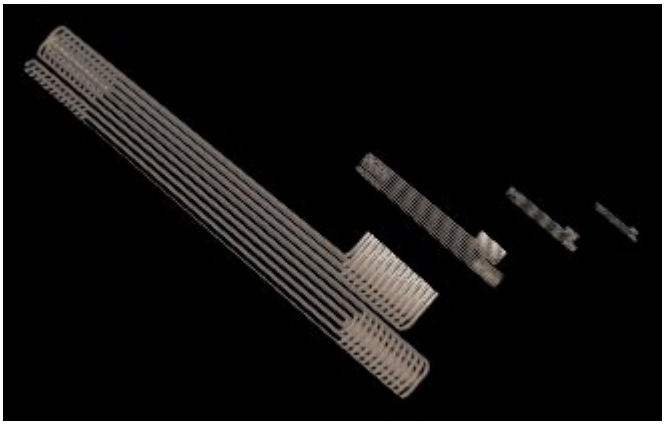


Figure 3.24: Switching distances for LODs of the Econ object if we allow 1 pixel of error. From left to right these LODs consist of 5,888 faces, 2,944 faces, 1,470 faces, and 736 faces. The original object has 23,556 faces.

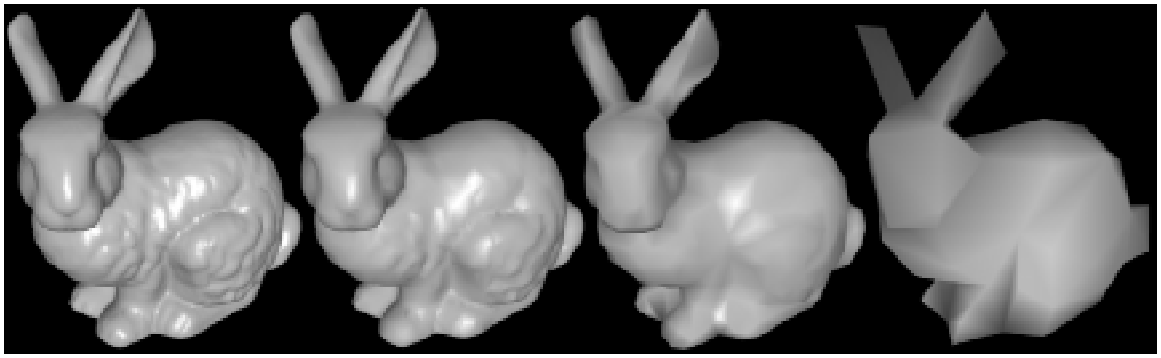


Figure 3.25: The original Bunny object and its LODs created by GAPS. From left to right these LODs consist of 69,451 faces, 8,680 faces, 1,085 faces, and 135 faces.

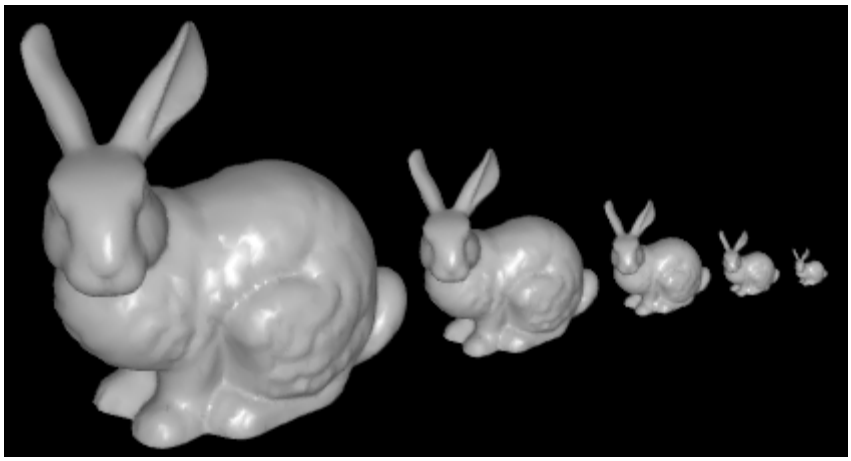


Figure 3.26: Switching distances for LODs of the Bunny object if we allow 1 pixel of error. From left to right these LODs consist of 17,361 faces, 8,680 faces, 4,340 faces, 2,169 faces, and 1,085 faces. The original object consists of 69,451 faces.

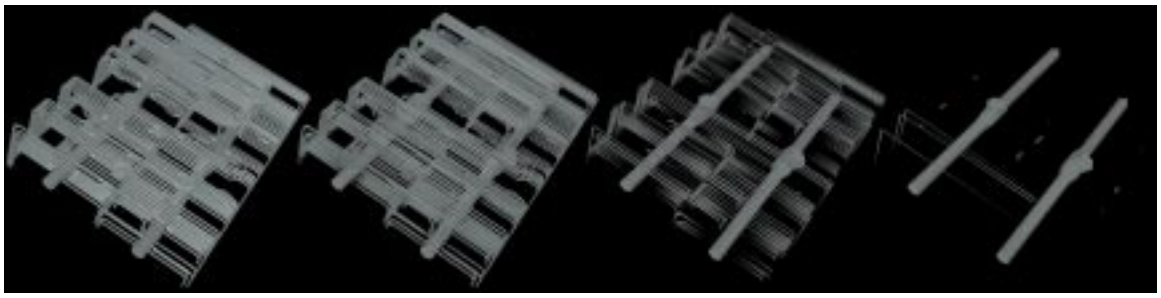


Figure 3.27: LODs for the ShDivWest object created by GAPS with no distance threshold and no surface area preservation. From left to right these LODs consist of 141,180 faces (the original object), 17,646 faces, 2,204 faces, and 272 faces.

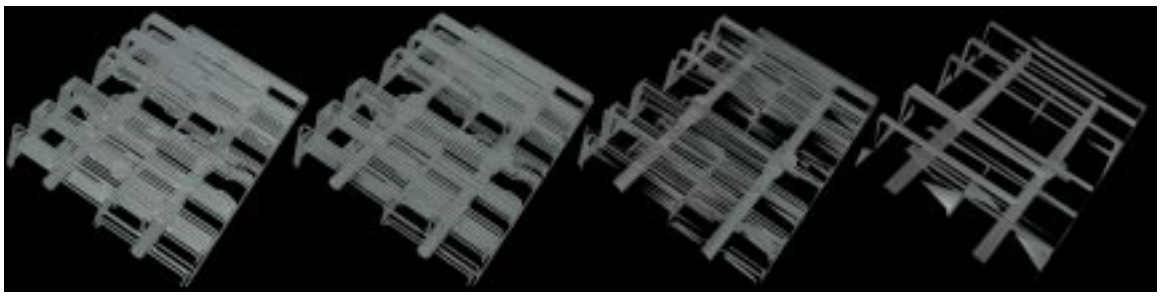


Figure 3.28: LODs for the ShDivWest object created by GAPS using an adaptive distance threshold and surface area preservation. In the latter stages of simplification, these LODs retain more of the overall structure of the pipes than the ones in Figure 3.27. From left to right these LODs consist of 141,180 faces (the original object), 17,644 faces, 2,202 faces, and 272 faces.

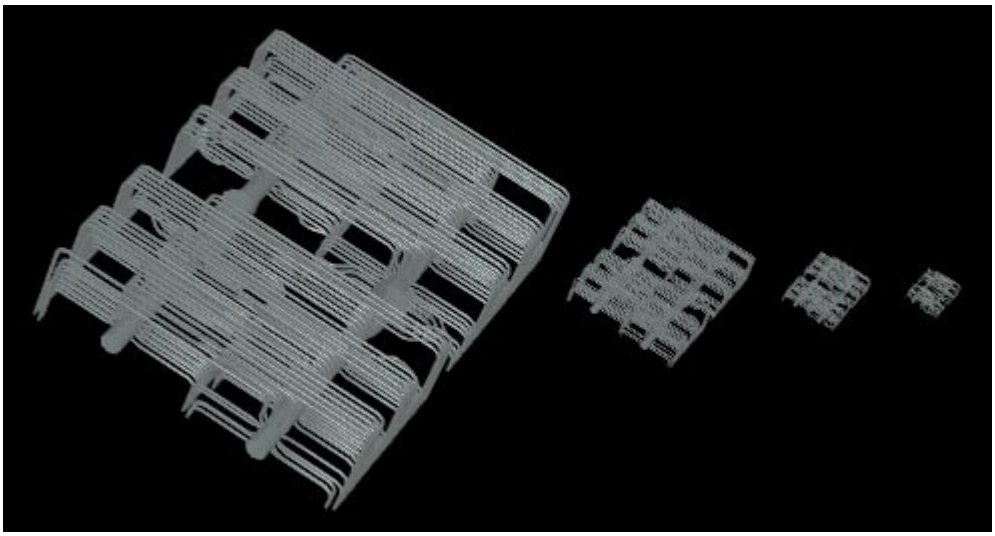


Figure 3.29: Switching distances for LODs of the ShDivWest object if we allow 1 pixel of error. From left to right these LODs consist of 70,588 faces, 35,292 faces, 17,644 faces, and 8,822 faces. The original object consists of 141,180 faces.

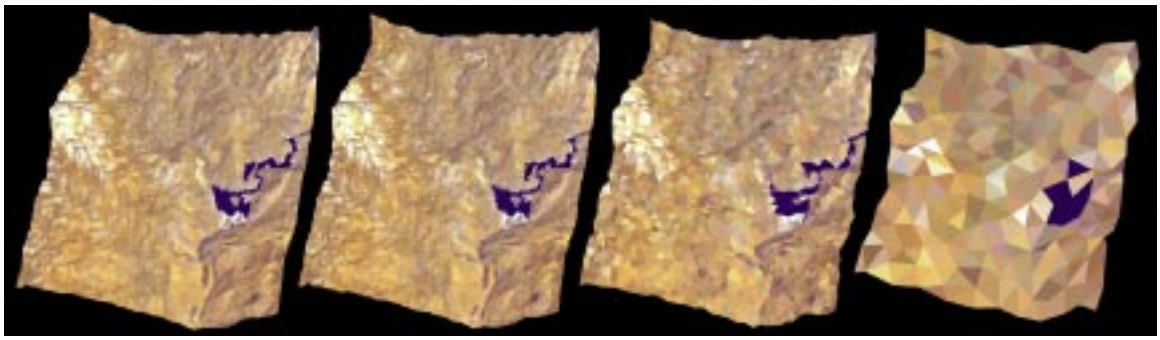


Figure 3.30: The original Sierra object and its LODs created by GAPS. From left to right these LODs consist of 162,690 faces, 20,335 faces, 2,541 faces, and 317 faces.

Figure 3.31: Switching distances for LODs of the Sierra object if we allow 1 pixel of error. From left to right these LODs consist of 81,345 faces, 40,671 faces, 20,335 faces, and 10,168 faces. The original object consists of 162,690 faces.

3.6 Analysis

Because we use a pair heap, GAPS has a bound on running time of $O(p \lg p)$ where p is the number of pairs in the heap. However, the number of pairs changes depending on the legality of pairs and the doubling of the distance threshold τ. To simplify our analysis of GAPS, we assume that the number of vertices that are within the distance threshold τ, the number of incident edges, and the number of adjacent faces of a specific vertex are at most a constant c, even after τ doubles. This assumption relies on τ doubling only when the vertex and edge density of the simplified object becomes lower than some constant density during the simplification process. For all the polygonal objects we have tested, GAPS exhibits this behavior, although one could create counter-examples. Another assumption we make is that hashing is a constant-time operation. Again, an object could be crafted to make this assumption invalid, but it holds true for our test results.

To find an initial distance threshold τ, we first make an initial guess in constant time. If it is not valid, then we halve τ and check its validity again, otherwise we stop. Note that τ has a lower bound of ε ($1e^{-10}$ in our implementation). Each validity check of τ involves hashing v vertices, where v is the number of vertices in the original object. We assume that each vertex is hashed in constant time. Let n be the maximum number of times we could halve the initial guess of τ. To calculate n,

$$\left(\frac{1}{2}\right)^n \tau = \varepsilon \Rightarrow \left(\frac{1}{2}\right)^n = \frac{\varepsilon}{\tau} \Rightarrow n \cdot \lg\left(\frac{1}{2}\right) = \lg\left(\frac{\varepsilon}{\tau}\right) \Rightarrow n = \frac{\lg\left(\frac{\varepsilon}{\tau}\right)}{\lg\left(\frac{1}{2}\right)} = \log_{\frac{1}{2}}\left(\frac{\varepsilon}{\tau}\right)$$

Since each validity check of τ involves hashing v vertices, and hashing is assumed to be a constant-time operation, calculating the initial distance threshold τ costs

$$v \cdot O(1) \cdot n = v \cdot O(1) \cdot \log_{\frac{1}{2}}\left(\frac{\varepsilon}{\tau}\right) = O\left(v \cdot \lg\left(\frac{\varepsilon}{\tau}\right)\right)$$

For the objects tested in Section 3.5, the initial value of τ was almost always found by the third iteration of halving its value.

Initially, we insert a maximum of $cv/2$ edges and $cv/2$ virtual edges as pairs into the heap, given our assumption that there are no more than c incident edges and c vertices within the distance threshold τ of a particular vertex. Therefore, there could be a maximum of cv pairs in the heap initially. In our analysis, we ignore the case where τ doubles because the maximum error on top of the pair heap is greater than τ. The only other way τ doubles is when all local edges are collapsed, resulting in new virtual edges being added to the heap. Assuming there were v vertices originally, then the number of vertices remaining when τ doubles is approximately v/c by our assumption of maximum constant density. In other words, in order for all of the virtual edges of a vertex to have been collapsed, it implies that each of the approximately c neighboring vertices have merged with it. Therefore, at this stage there will be a maximum of $c(v/c) = v$ pairs in the heap. In general, after the nth doubling of τ, there will be at most $c(v/c^n)$ candidate pairs. Once $v/c^n = 1$, the simplification process stops. Each time a pair is extracted from a heap of size p, the operation costs $O(\lg p)$. Therefore, a bound on the running time due to the heap can be determined using the Master Theorem, described in [Cormen et al. 94], as

$$O(cv \cdot \lg(cv)) + O\left(c\left(v/c\right) \cdot \lg\left(c\left(v/c\right)\right)\right) + O\left(c\left(v/c^2\right) \cdot \lg\left(c\left(v/c^2\right)\right)\right) + \cdots + O\left(c\left(v/c^n\right) \cdot \lg\left(c\left(v/c^n\right)\right)\right) = O(cv \cdot \lg(cv))$$

Finding virtual edges for one vertex, assuming constant time hashing, takes $O(c)$, or constant time. Therefore, finding virtual edges for v vertices takes $O(cv)$ time. The time taken to find virtual edges after τ doubles during the entire simplification process can be derived from the Master Theorem, described in [Cormen et al. 94], as

$$O(cv) + O\left(c\left(v/c\right)\right) + O\left(c\left(v/c^2\right)\right) + \cdots + O\left(c\left(v/c^n\right)\right) = O(cv)$$

Other time critical operations, such as checking for surface area preservation, checking for mesh inversion, calculating optimal vertex positions, joining quadrics, and combining point clouds can be considered constant time operations. Therefore, using our simplified analysis, the expected execution time of GAPS is

$$O\left(v \lg\left(\varepsilon/\tau\right)\right) + O(cv \cdot \lg(cv)) + O(cv) = O(v) + O(v \lg v) + O(v) = O(v \lg v)$$

As empirical proof of this analysis, the timings of GAPS versus QSlim are similar (see Section 3.5.1). With a few assumptions, QSlim can be shown to run in $O(p \lg p)$ time where p is the number of initial candidate pairs in its heap. In the worst case, $p = O(v^2)$. Thus the worst case running time for QSlim and GAPS is $O(v^2 \lg v^2)$. This worst case scenario would involve simplifying very contrived or degenerate polygonal geometry. In practice, none of the test objects in Section 3.5 produced this worst case behavior.

3.7 Comparison

In this section, we first compare simplification algorithms that lie in the domain of topological simplification followed by ones that handle surface attributes. [Popovic and Hoppe 97] presents an elegant topological simplification algorithm to produce a progressive simplicial complex. It is slow and uses primitives other than polygons. [Rossignac and Borrel 93] and [Low and Tan 97] use vertex clustering to topologically simplify models quickly and robustly, sometimes producing low quality approximations. [He et al. 95] uses a voxel-based approach that makes it difficult to control the degree of simplification. [El-Sana and Varshney 97] controls the genus of an object and [Schroeder 97] simplifies to any target number of faces quickly, and both are not able to join unconnected regions of an object.

Increasingly more algorithms are addressing surface attributes during the simplification process. [Hoppe 96] includes *scalar* and *discrete* attributes in the energy function the algorithm tries to minimize. This method is slow in practice. [Cohen et al. 98] uses texture and normal maps for appearance preserving simplification. It requires a surface parameterization that is difficult to compute for arbitrary polygonal objects. Machines that do not support normal mapping will render the simplified output slowly. Garland and Heckbert extend their original algorithm in [Garland and Heckbert 98] to handle surface attributes. They ignore topological simplification and use a higher dimensional error quadric to track geometric and surface attribute error together. We efficiently decouple surface attributes from polygonal geometry and use a unified error metric that aids automatic selection of switching distances for LODs of an object. [Cignoni98] proposes an efficient algorithm that

creates surface attributes for a simplified mesh by sampling the original object. This method requires the storage of potentially large texture maps for each level of detail desired.

3.8 Summary

We present a new topological simplification algorithm that is general, automatic, executes efficiently, handles surface attributes, produces high quality approximations, and merges unconnected regions of objects using an adaptive distance threshold and surface area preservation. It uses an object space error metric, incorporating both geometric and surface attribute error, to aid the calculation of switching distances for LODs. The algorithm appears to work well on a wide range of models, ranging from smooth surfaces to disjoint pipes.

4 SIMPLIFICATION OF STATIC POLYGONAL ENVIRONMENTS

In the previous chapter, we presented techniques to achieve static intra-object polygonal simplification. It was shown that merging unconnected regions of polygons in a single object is an effective technique for producing high quality and drastic approximations. However, large polygonal environments usually consist of numerous objects. If objects are in close proximity in the scene, a simplification algorithm might be able to produce visually better approximations by merging regions of different objects. In other words, the algorithm would perform inter-object in addition to intra-object simplification. In this chapter, we present an algorithm to accelerate the rendering of large static polygonal environments. We represent these polygonal scenes using a modified scene graph that includes both traditional levels of detail, or LODs, and *hierarchical levels of detail*, or *HLODs*, that approximate the polygonal geometry of portions of the scene graph. HLODs are the product of inter-object simplification. We discuss how they are created, how they are integrated into the traditional scene graph traversal, and how they can be used to gain limited view-dependent rendering capabilities.

The rest of this chapter is organized in the following manner. We provide an overview of our algorithm in Section 4.1 and discuss the technical details in Section 4.2. Implementation details are presented in Section 4.3 and performance results are shown in Section 4.4. Analysis of the running time of the system is shown in Section 4.5. We compare our algorithm based on HLODs to other rendering methods in Section 4.6 and we conclude the chapter in Section 4.7.

4.1 Overview

We assume that the polygonal environment is represented using a scene graph [Clark 76, Rohlf and Helman 94] and that our algorithm uses standard view-frustum culling. A scene

graph is a directed acyclic graph consisting of nodes connected by arcs. Polygonal geometry and bounding volumes are stored at nodes while transformations are stored at arcs. We can efficiently replicate objects in the scene by using instancing. An example scene graph is shown in Figure 4.1.

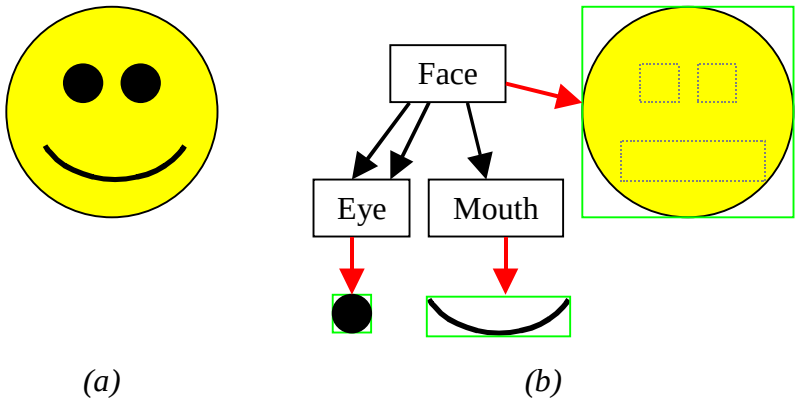


Figure 4.1: A simple scene graph. *(a)* A model of a face. *(b)* The model's scene graph.

In Figure 4.1, boxes enclosing text represent nodes while black arrows represent arcs. Red arrows show the polygonal geometry that is contained in each node. Bounding boxes for each node are shown in green. Even though the face node is the root node and not a leaf node, it contains polygonal geometry. The dotted gray boxes in the face node's polygonal geometry denote the bounding boxes of its children. Note the bounding box of the face node encloses its polygonal geometry and the bounding volumes of its children. There is only one definition of the polygonal geometry of an eye, but there are two in the model due to the eye node's two incoming arcs. Each arc uses a different transformation to instance the eye at different positions on the head.

A traditional scene graph rendering system such as Performer [Rohlf and Helman 94] uses LODs to accelerate the rendering of polygonal environments. We demonstrated in Chapter 3 that high quality and drastic LODs can be created for certain objects by merging unconnected regions of polygonal geometry during simplification. Polygons located in different nodes of the scene graph will never be merged together in a traditional LOD-based system. For example, Figure 3.23 shows the Econ model. By creating LODs for this model, we assume its polygons are located in one node in a scene graph. In Chapter 3, we use GAPS

to merge the disjoint pipes of the Econ model together, thereby performing intra-object simplification. Suppose the polygonal geometry of each pipe of the Econ model was located in its own node in a scene graph. If we use a purely LOD-based system, we would simplify the polygons of each pipe independently, regardless of the simplification technique used. Given the results shown in Figure 3.23, we would prefer to merge the pipes.

The idea of hierarchical levels of detail, or HLODs, is not new. HLODs are equivalent to what [Clark 76] calls *objects* and what [Maciel and Shirley 95] calls *meta-objects*. [Clark 76] introduces the idea of a hierarchy of levels of detail. [Maciel and Shirley 95] presents a method of generation based on creating view-dependent images. Unlike this previous work, we use a polygonal simplification method, namely GAPS presented in Chapter 3, to generate LODs and HLODs. Since it is able to merge disjoint polygons, GAPS is well suited for the creation of HLODs. In order for this merging to occur, we pool the polygons of different nodes in the scene graph and perform inter-object simplification on this combined polygonal geometry. By doing so, we create hierarchical levels of detail that represent multiple nodes in the scene graph.

During the visualization of polygonal environments, HLODs provide several advantages. By rendering an HLOD, we can ignore portions of the scene graph that the HLOD represents, making the scene graph traversal more efficient. Since HLODs are created by simplifying and merging the combined polygonal geometry of multiple nodes in the scene graph, they are, in general, better-looking representations for this group of objects as compared to the individual LODs of these objects. Figure 1.8 demonstrates the visual benefit of HLODs compared to using solely LODs. Other visual comparisons between using LODs and HLODs to render various scenes are shown in Section 4.4.4. By partitioning spatially large objects, hierarchically grouping these partitions, and creating HLODs for this hierarchy, we gain limited view-dependent simplification capabilities. This partitioning scheme is described in Section 4.2.3. Finally, when we traverse the scene graph, HLODs guarantee that we will always have a complete representation of the scene. As shown in Section 4.2.4, this guarantee helps our method to target a frame rate when rendering.

4.1.1 Levels of Detail Versus View-Dependent Techniques

Traditionally, a simplification algorithm produces a series of levels of detail for a polygonal object. For example, Figure 1.3 shows LODs generated for a bunny model. A rendering algorithm chooses between these LODs when drawing the bunny. As the viewer moves around in the environment, different representations of the bunny are rendered. If the system renders different LODs for the bunny from one frame to the next, the viewer can sometimes see a "pop," or subtle shift in the image. This problem associated with static LODs is called the *popping problem*. Early techniques used alpha-blending to transition smoothly between two different LODs. This method requires rendering two sets of the model during the transition period.

Spatially large objects pose another problem for traditional LOD-based algorithms. Suppose there are LODs that represent a long, skinny, and highly tessellated polygonal cylinder, and the viewer is near one end. If we render a coarse LOD, then parts of the cylinder that are close to the viewer will be of low quality. If we render a highly tessellated LOD, then we will be unnecessarily rendering polygons at the other end of the cylinder in high detail, even though they are a great distance away. The problem with this object is that it spans a great distance and thus the viewer can simultaneously be both near to and far from different parts of it.

Recently, many researchers [Hoppe 97, Luebke and Erikson 97, Xia et al. 97] have proposed using *view-dependent simplification* for polygonal environments. These algorithms adaptively simplify across the surface of objects. 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 [Hoppe 96] 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 LODs. Also, since view-dependent systems render spatially large objects using adaptive simplification, they eliminate the inefficiency of using LODs to represent them. In the case of the cylinder example, view-dependent simplification algorithms would reduce the polygon count

significantly by rendering areas of the cylinder close to the viewer in high detail and regions further away in lower detail.

There are reasons why we chose not to use view-dependent simplification for our applications.

- They impose significant overhead in terms of memory and CPU usage 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 handle instancing inefficiently since each instance must contain its own list of active vertices. In many CAD applications, instancing is heavily used.
- *Display list* rendering is often significantly faster than *immediate mode* rendering on current high-end graphics hardware [Aliaga 97] such as SGI Infinite Reality systems. Existing view-dependent algorithms are inherently immediate mode techniques. Therefore, polygonal data must be transferred to the graphics machine each frame. As a result, these techniques must simplify the environment more to achieve a frame rate comparable to using display lists for LODs.
- View-dependent methods must constantly insert and delete faces from a linked list of active faces. To render an object, the view-dependent system sequentially traverses this list and draws each face. There is no guarantee that these faces are contiguous in memory, which can lead to inefficient memory access.
- Finally, in Chapter 5, we extend our system to handle dynamic environments. For a view-dependent system, it is unclear how one could efficiently update the hierarchical tree of vertices to account for moving objects.

For these reasons, our approach to rendering static scenes uses traditional LODs augmented with HLODs. HLODs approximate the polygonal geometry of portions of the scene graph. When an HLOD is rendered, the system can ignore those portions during its scene graph traversal. HLODs are created by first *associating*, or grouping together, nearby

nodes in the scene graph based on an octree spatial partitioning, and then performing simplification on this pooled polygonal geometry. Traditional LOD-based algorithms can be ineffective at representing spatially large objects. However, by *partitioning* these objects, building a scene graph structure for these partitions complete with HLODs, and guaranteeing that we produce no cracks between partitions, we gain limited view-dependent capabilities. Since each node of the scene graph contains an HLOD representing the scene graph rooted at that node, our algorithm can render using a target frame-rate mode, as well as a pixel-error mode. Finally, the system uses *display lists* to efficiently render all LODs and HLODs. By choosing LODs and HLODs, we offer no new solution for the popping problem. Our algorithm has been used on several complicated CAD environments. When viewing these scenes with our techniques, we achieve significant speedups with little loss in image quality as compared to using no LODs at all.

4.2 Hierarchical Levels of Detail to Accelerate the Rendering of Static Environments

We extend the traditional scene graph representation for polygonal models to include inter-object simplification via HLODs. The main techniques that make this algorithm practical are described below.

- The system precomputes and includes HLODs at each node in the scene graph and its scene graph traversal algorithm is modified accordingly.
- Objects are hierarchically grouped together for inter-object simplification using a process called *association*.
- The algorithm *partitions* spatially large objects in order to gain limited view-dependent rendering capabilities.
- Besides using a *pixel-error* rendering mode, our algorithm is capable of targeting a frame rate as a result of using HLODs.
- All LODs and HLODs are efficiently rendered using *display lists*.

4.2.1 Hierarchical Levels of Detail

Hierarchical levels of detail, or HLODs, are a set of LODs for a group of objects in an environment. We use a polygonal simplification method, namely GAPS, to generate LODs and HLODs. GAPS is well suited for the creation of HLODs since it is able to merge unconnected regions of polygons from separate objects.

LODs represent the polygonal geometry of single nodes in the environment's scene graph. HLODs represent portions of the scene graph, or the polygons of multiple nodes. Using only LODs, we would render an appropriate level of detail for each object in the scene (see Figure 4.2). Using HLODs, we might render a hierarchical level of detail in place of a portion of the scene graph (see Figure 4.3).

A more precise summary of how HLODs are constructed is as follows. LODs represent the polygons of a single node. If a node in the scene graph is a leaf node, then its HLODs are equivalent to its LODs. Otherwise, HLODs of a node are approximations that represent the polygons contained in the LODs of the node plus the HLODs of any children nodes. The starting geometry for HLODs of a node is the combination of the coarsest LOD of the node combined with the coarsest HLODs of the node's children. HLODs are created from these base polygons using GAPS. For example, in Figure 4.3, the HLODs in Face represent the polygonal geometry of the LODs of Face and the HLODs of Eye (left), Eye (right), and Mouth. Since Eye and Mouth do not have children nodes, their HLODs are equivalent to their LODs. Thus, when an HLOD is rendered, the system can ignore the node and all of its children while traversing the scene graph.

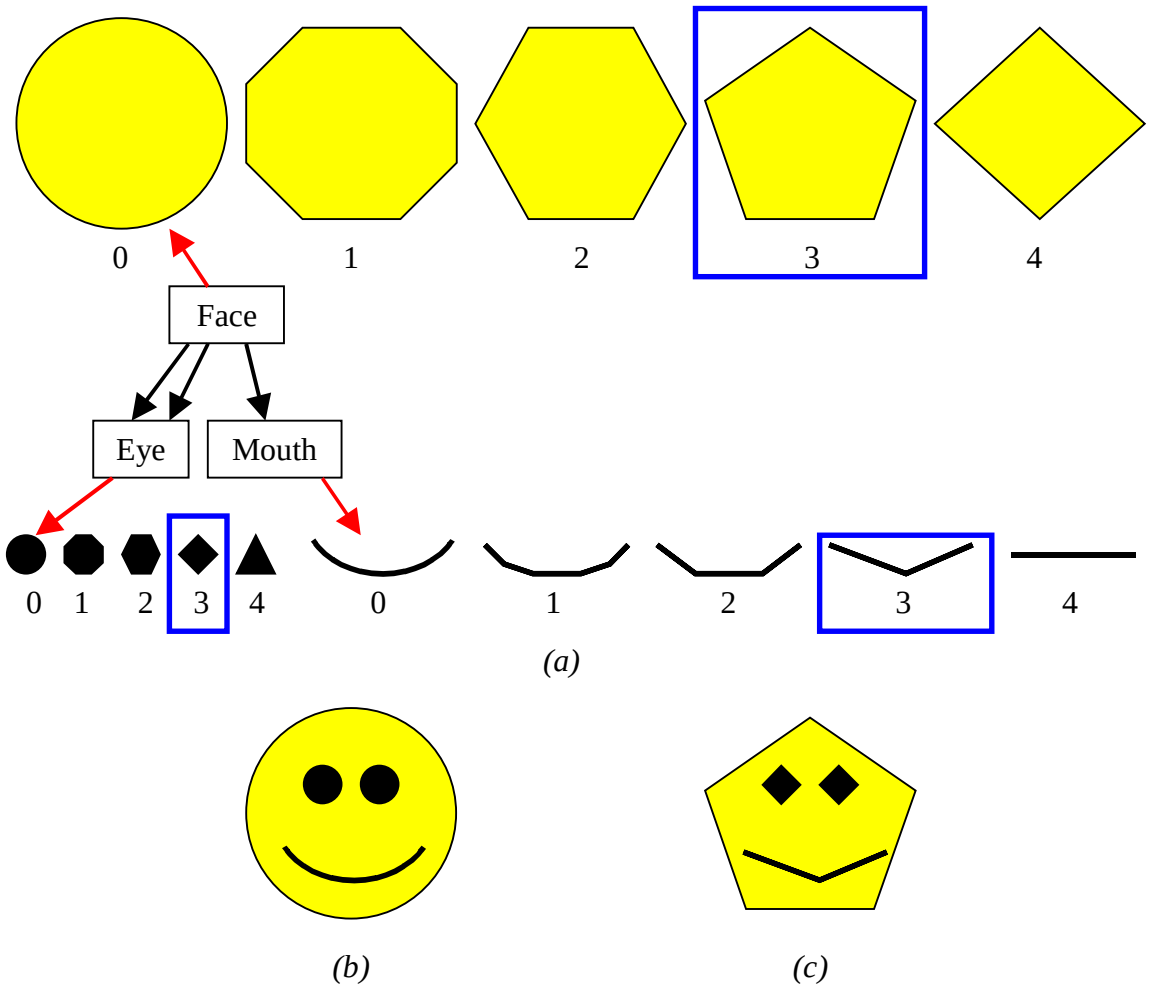


Figure 4.2: Rendering of a face model using LODs. *(a)* Scene graph of the face model. Red arrows show LODs representing polygonal geometry contained in each node. *(b)* The original face model. *(c)* Since the viewer is far away, this simplified face is an acceptable approximation. The LODs enclosed in blue boxes in *(a)* are the ones rendered. The rendering algorithm traverses the scene graph starting at Face. It renders an appropriate representation of the face using LOD 3, and then traverses the node's children. Next, it visits Eye and renders the left eye with LOD 3. It then visits Eye again and renders the right eye with LOD 3. Finally, it enters Mouth and renders LOD 3.

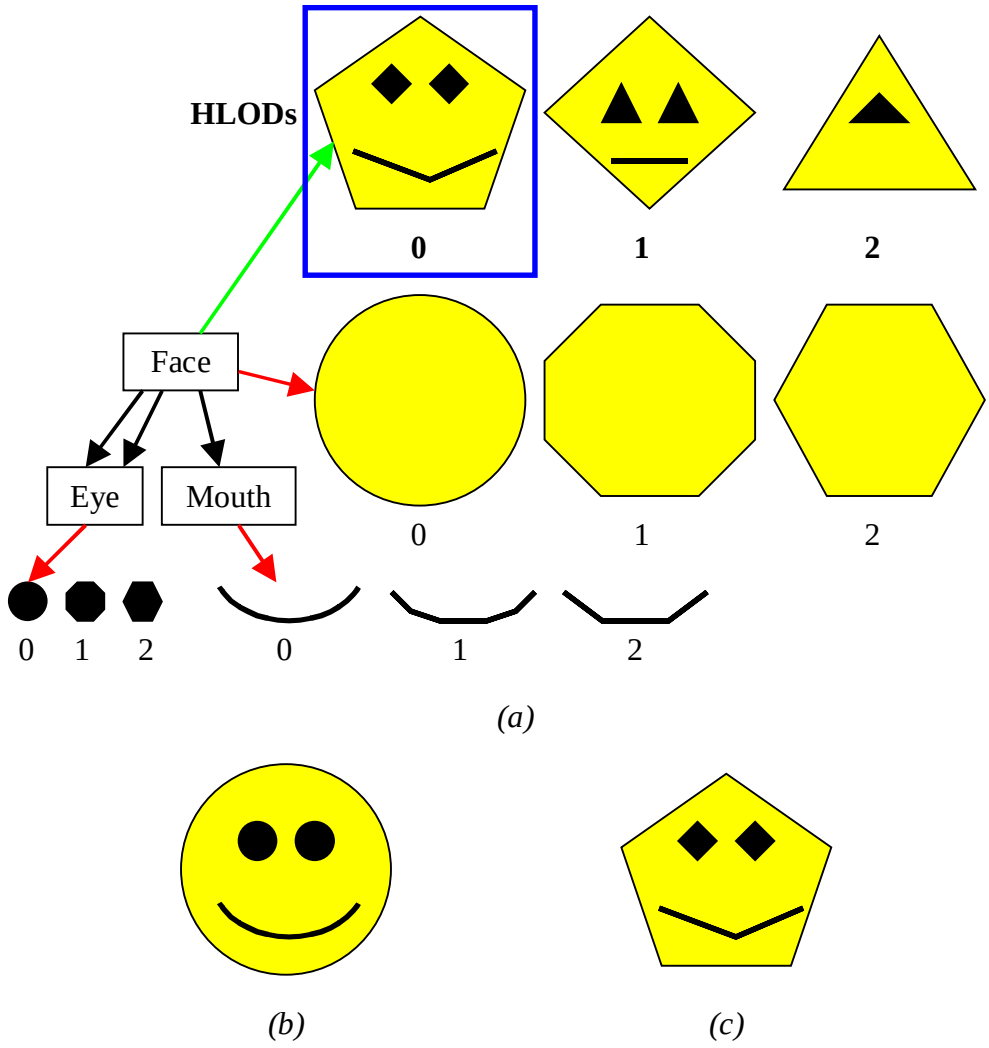


Figure 4.3: Rendering of a face model using LODs and HLODs. *(a)* Scene graph of the face model. Red arrows show LODs representing polygonal geometry contained in each node. The green arrow shows HLODs representing portions of the scene graph. In this case, the HLODs represent the entire model. *(b)* The original face model. *(c)* Since the viewer is far away, this simplified face is an acceptable approximation. The HLOD enclosed in the blue box in *(a)* is the one rendered. Our algorithm traverses the scene graph starting at Face. It renders an appropriate representation of the face using HLOD 0. Since this HLOD represents the entire scene graph, the system ignores the node's children and is finished rendering. Note that HLOD 2 demonstrates the merging of the two eyes, something not possible in a traditional object-based LOD algorithm.

Each LOD and HLOD has an associated distance error that is produced by GAPS (see Section 3.3.3.4). This error is used to determine if a particular LOD or HLOD is an

acceptable representation given a maximum allowable pixel-error threshold and the position of the viewer relative to the object. Figure 4.4 shows two methods of LOD selection.

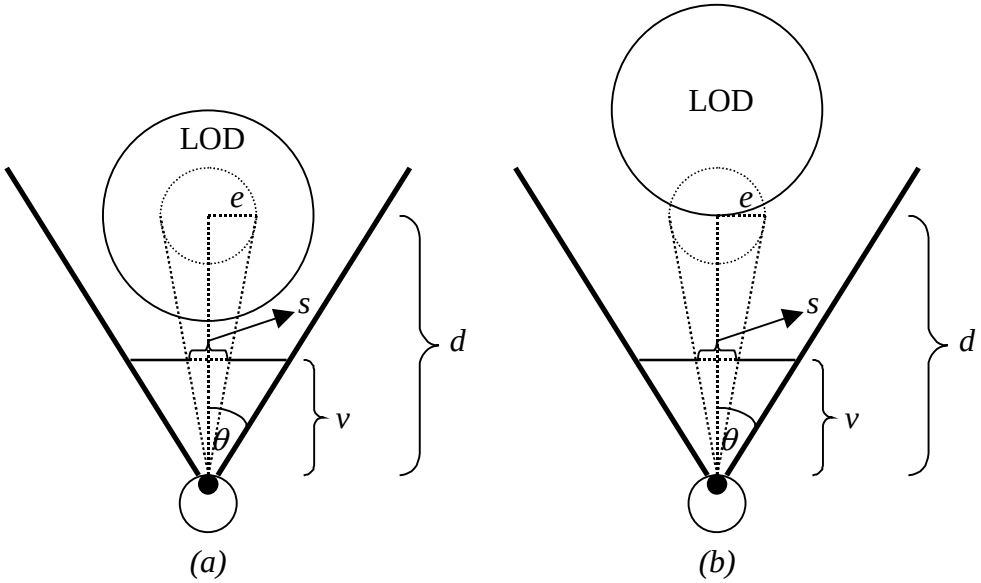


Figure 4.4: Methods of LOD selection. In *(a)*, *d* represents the distance between the eye of the viewer and the center of the LOD's bounding circle. In *(b)*, *d* represents the shortest distance between the eye and the LOD's bounding circle.

In Figure 4.4, the distance from the eye to the view plane is v and the field of view angle for half the view frustum is θ. Assume the user specifies a maximum of P pixels of error in the final image and the viewing screen consists of N by N pixels. To transform P pixels of error into a distance error p along the view plane, we multiply the ratio of pixels of error to view screen pixels by the length of the view plane. The length of the view plane in eye space is $2v \cdot \tan\theta$. Thus,

$$p = P/N \cdot (2v \cdot \tan\theta)$$

The LOD's bounding circle is shown as well as a dashed smaller circle representing the distance error e associated with the LOD. The projected screen-space distance error is s. By using similar triangles,

$$\frac{1/2\, s}{v} = \frac{e}{d} \Rightarrow s = \frac{2ve}{d}$$

The LOD is an acceptable approximation given P as a constraint if s is less than or equal to the allowable pixel distance error p. In other words, the LOD can be rendered if

$$s \le p \Rightarrow \frac{2ve}{d} \le P/N \cdot (2v \cdot \tan\theta) \Rightarrow e \le d \cdot P/N \cdot \tan\theta$$

We render the coarsest LOD possible that meets the pixel error criteria. Note that in Figure 4.4, *(b)* is a more conservative method for determining which LOD to render than *(a)* since d is always smaller in *(b)* than *(a)* for the same object. Our algorithm is capable of using either method, but uses method *(a)* by default. For efficiency and rotational invariance, we assume the viewer is looking directly at an object when we calculate which LOD to render. In other words, the distance d between the viewer and an object remains constant as the viewer rotates in a stationary position**.**

4.2.2 Node Association

We assume that for each polygonal environment, a scene graph already exists. If the model has no hierarchy, then our algorithm is capable of creating one using *partitioning* (see Section 4.2.3). Because we use the scene graph associated with a polygonal environment as is, the creator of the scene graph has control over which objects are grouped together into HLODs and in which order they are combined. Therefore, the scene graph can be structured in a way that groups objects intelligently and efficiently. For example, suppose we are working with a polygonal model of a sparse office consisting of walls, a garbage can and filing cabinet in one corner, and a chair and desk in another. For the purpose of creating HLODs, an efficient grouping of these objects to utilize spatial coherence would be to group the garbage can and cabinet together as well as the chair and desk. Then the walls and these two groups could be joined as the root of the scene graph. Figure 4.5 demonstrates this grouping. Note that by grouping nearby objects first, we make both HLOD creation and view-frustum culling more efficient.

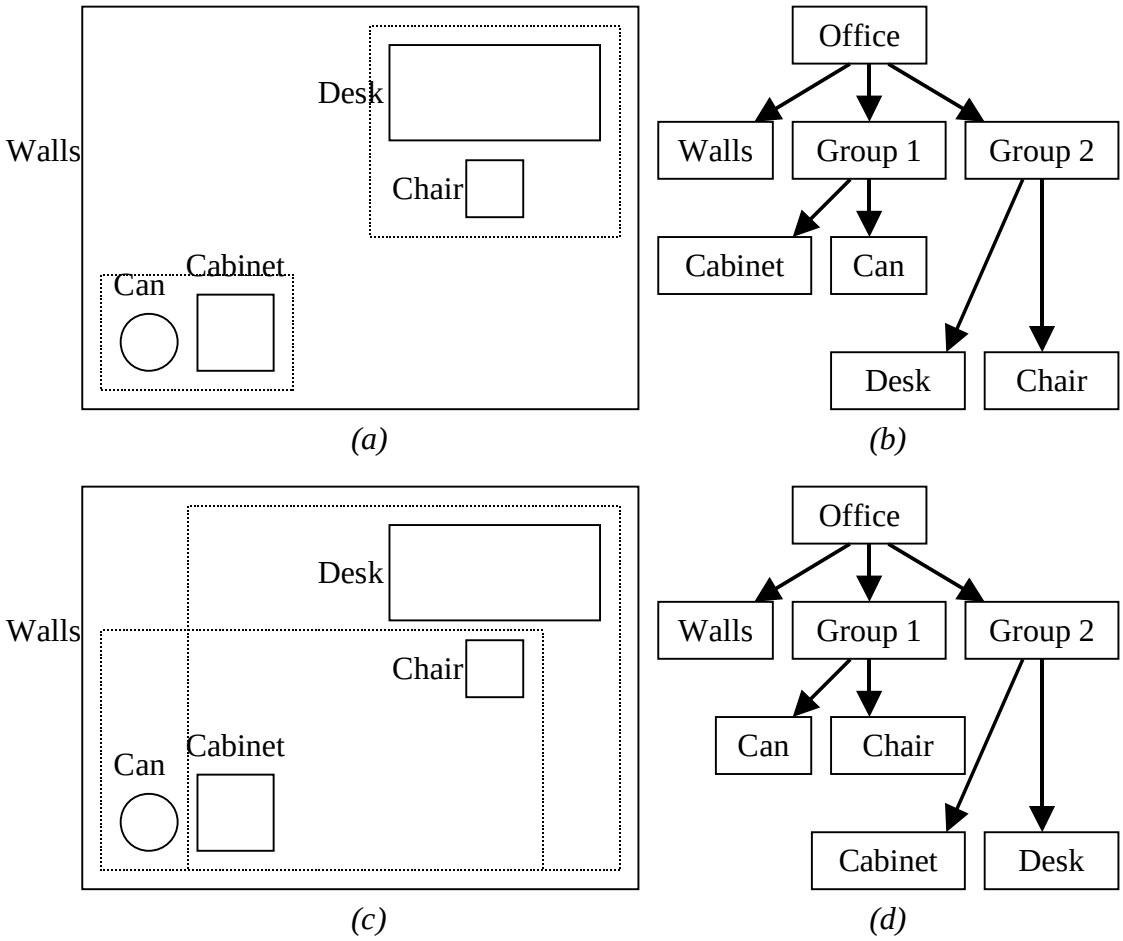


Figure 4.5: Since the structure of the scene graph controls the creation of HLODs, the creator of the scene can dictate the order used for grouping objects. *(a)* A sparse office model. The creator of the office scene graph has intelligently grouped the can and cabinet together and the desk and chair together because of their proximity. *(b)* The resulting scene graph from this grouping. Since nearby objects are grouped together, view-frustum culling efficiency and HLOD quality are maximized. *(c)* A poor choice of grouping objects. *(d)* This scene graph leads to inefficient HLOD creation and view-frustum culling.

Unfortunately, not all scene graphs are built for rendering performance. It is common for CAD models to group things by functionality, ignoring the relative locations of objects. In these cases, a user can tell our algorithm to ignore the incoming scene graph and instead use a flattened graph. A flattened scene graph consists of a root node with as many children nodes as there are objects in the scene. For example, Figure 4.6 shows the office model using a flattened scene graph.

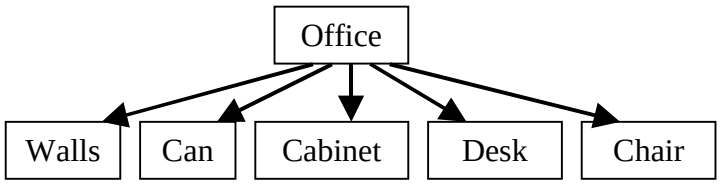


Figure 4.6: A flattened scene graph of the office model. No objects are hierarchically grouped together.

Flat scene graphs are not efficient for either view-frustum culling or HLOD creation. Therefore, we augment scene graph nodes that contain multiple children through a process called *node association*. If a node has more than two children, then we use an octree spatial partitioning to *associate* nearby nodes together. We combine nodes that are in close proximity to attempt to minimize the bounding volume size of each new node in this *association graph*. This grouping of nodes is performed hierarchically by combining objects in close proximity first, making view-frustum culling more efficient. Grouping nearby objects also allows for the creation of higher quality HLODs. This approach makes HLOD computation feasible for nodes with large numbers of children, since it bounds the number of children of any node in the association graph by eight. Figure 4.7, Figure 4.8, and Figure 4.9 demonstrate node association on a small scene.

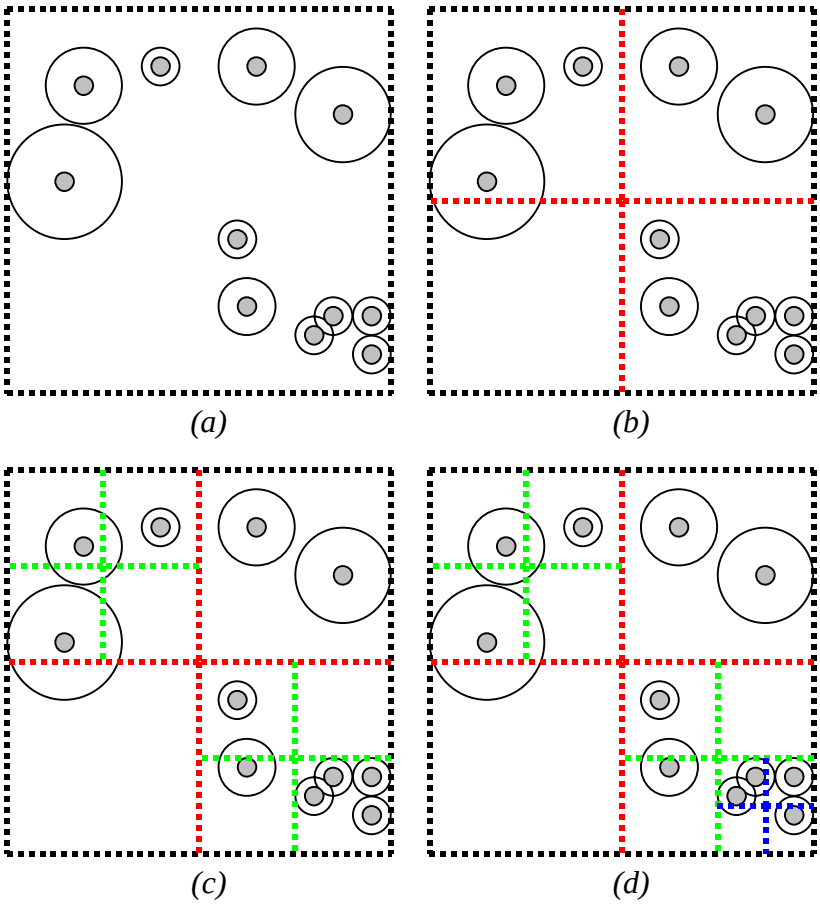


Figure 4.7: Associations for a small two-dimensional scene. Objects are depicted using their bounding circles along with a gray point representing the circle's center. The center of an object's bounding circle determines which partition the object lies in. Association is a top-down process. *(a)* The entire scene and its bounding rectangle. There are more than two objects in the rectangle so the space is subdivided using a quadtree. *(b)* There are more than two objects in the upper left and lower right quadrant. These quadrants must be subdivided. *(c)* Only the lower right quadrant needs to be subdivided. *(d)* The final quadtree subdivision.

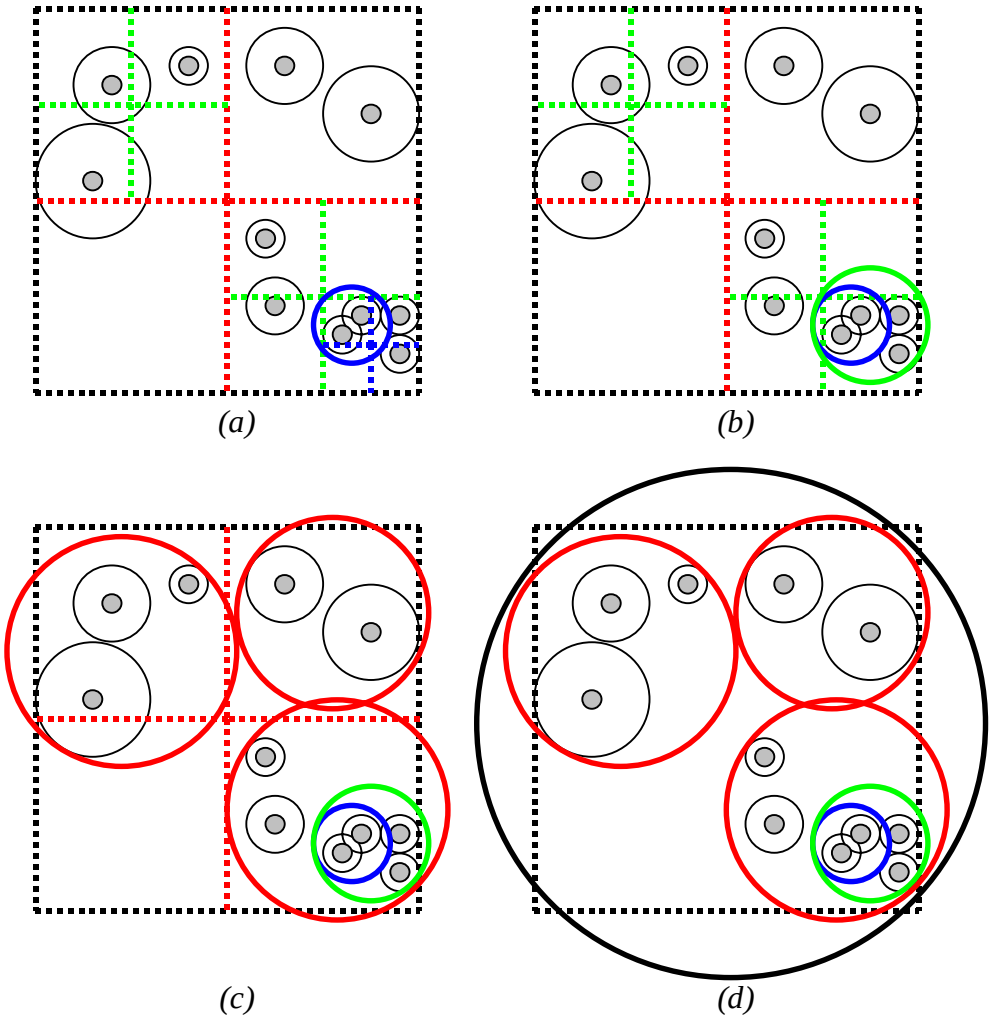


Figure 4.8: Creation of the bounding volume hierarchy from the quadtree subdivision in Figure 4.7. ***(a)*** **The hierarchy is built bottom-up. Each bounding circle encloses any objects or bounding circles that lie within a particular partition. The blue bounding circle bounds the lowest level objects in the quadtree subdivision.** ***(b)*** **The green bounding circle encloses both nodes and the blue bounding circle.** ***(c)*** **Red bounding circles enclose objects and bounding circles created during the first quadtree subdivision.** ***(d)*** **The root node bounding circle encloses everything.**

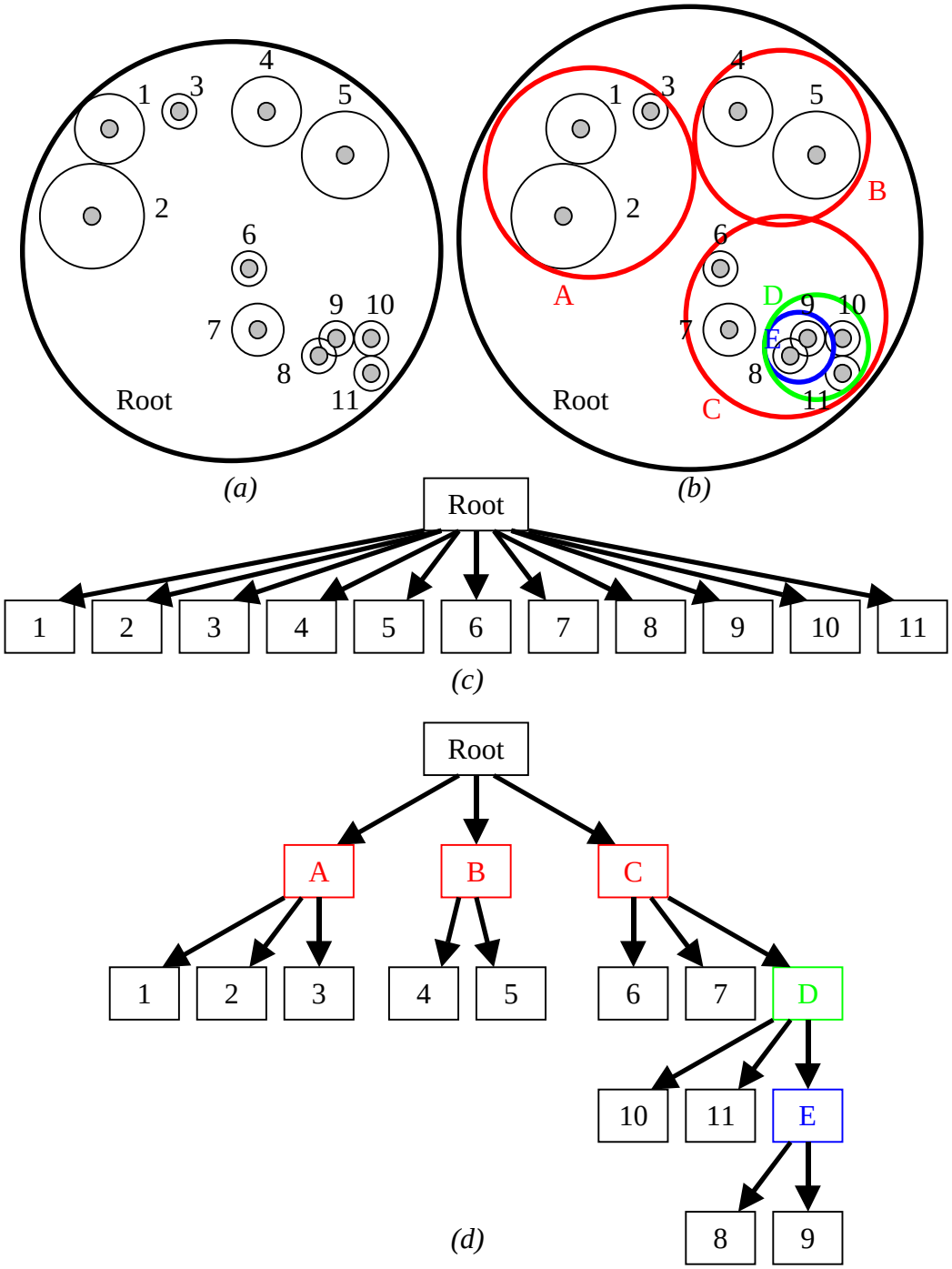


Figure 4.9: The resulting association graph from Figure 4.7 and Figure 4.8. *(a)* The original scene with one bounding volume. The objects are numbered. *(b)* The bounding volume hierarchy created in Figure 4.8. *(c)* The scene graph for the original scene. *(d)* The scene graph for the associated scene. This scene graph allows for more effective view frustum culling and HLOD creation.

4.2.3 Partitioning Spatially Large Objects

Since we use static LODs, spatially large objects pose a problem. When the viewer is close to any region of a spatially large object, the entire object must be rendered in high detail, even though portions of it might be very far from the viewer. To alleviate this problem, we *partition* the model to gain limited view-dependent rendering capabilities. We simplify each partition while guaranteeing that we do not produce cracks between partitions by imposing simplification restrictions at vertices. Finally, we use HLODs to combine the polygonal geometry of these partitions hierarchically.

We partition large objects using a three-dimensional uniform spatial subdivision. A user-specified distance p determines the size of the cubes into which we partition the object. This distance is usually based on a user-specified percentage of the size of the bounding volume of the entire scene graph. Objects that are completely contained within this distance are not partitioned. Therefore, we use this technique only on objects that are large relative to the whole environment. We lay a uniform three-dimensional grid over the object and determine into which cube each polygon's centroid falls. Each partition is assigned a collection of polygons, namely the ones whose centroids fall within the partition. To insure that we do not create cracks during simplification of partitions, each vertex in the original object has an associated number of *restrictions*. If a polygon's centroid lies within a particular partition, but the polygon's vertices do not all lie in that same partition, then the number of restrictions for each vertex of the polygon is incremented by one. It is possible for a vertex to have more than one restriction if it is part of polygons that are included in different partitions. If a vertex has one or more restrictions, then it is a restricted vertex. An example of this partitioning scheme is shown in Figure 4.10.

Next, the system simplifies the geometry within each partition independently. GAPS cannot use a restricted vertex in a vertex merge operation. This restriction guarantees that GAPS will not create cracks between partitions. GAPS simplifies the unrestricted geometry until no more vertices can be merged or a user-specified distance error threshold has been exceeded. This distance error is defined as a certain percentage of the size of the bounding volume of the partition. This simplification process is demonstrated in Figure 4.11.

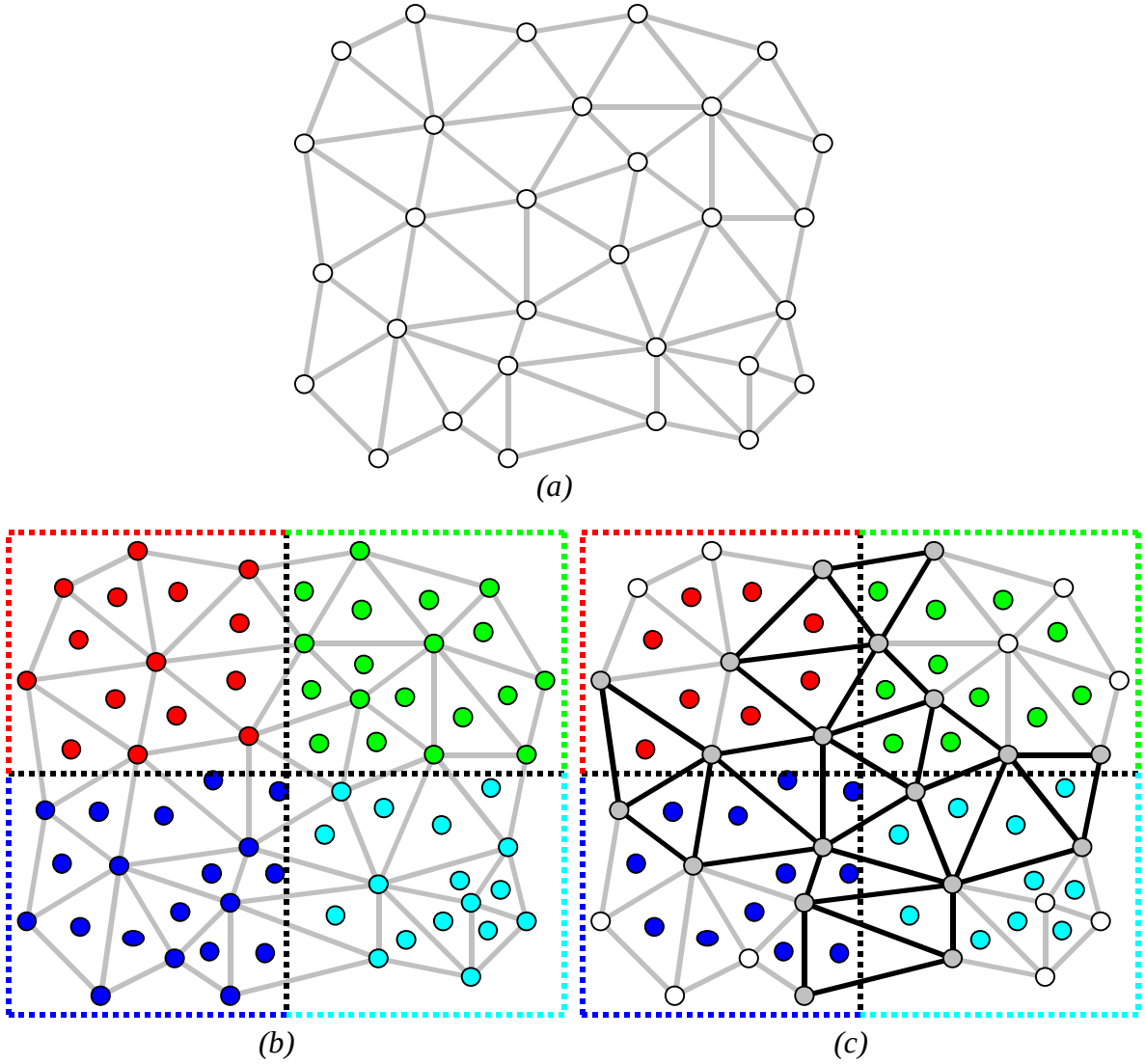


Figure 4.10: Partitioning a small two-dimensional object. *(a)* The original object. *(b)* The object has been spatially partitioned into four uniform sized quadrants. The upper left quadrant is colored red. Any vertex or any centroid of a face that falls within this partition is colored red. Similarly, the other quadrants are colored green, blue, and cyan. Faces are included in the partition that contains their centroid. *(c)* Any face whose vertices are contained in more than one partition are restricted. A vertex included in a restricted face is itself restricted. Restricted faces consist of black edges. Gray vertices in the diagram denote restricted vertices while white ones are unrestricted.

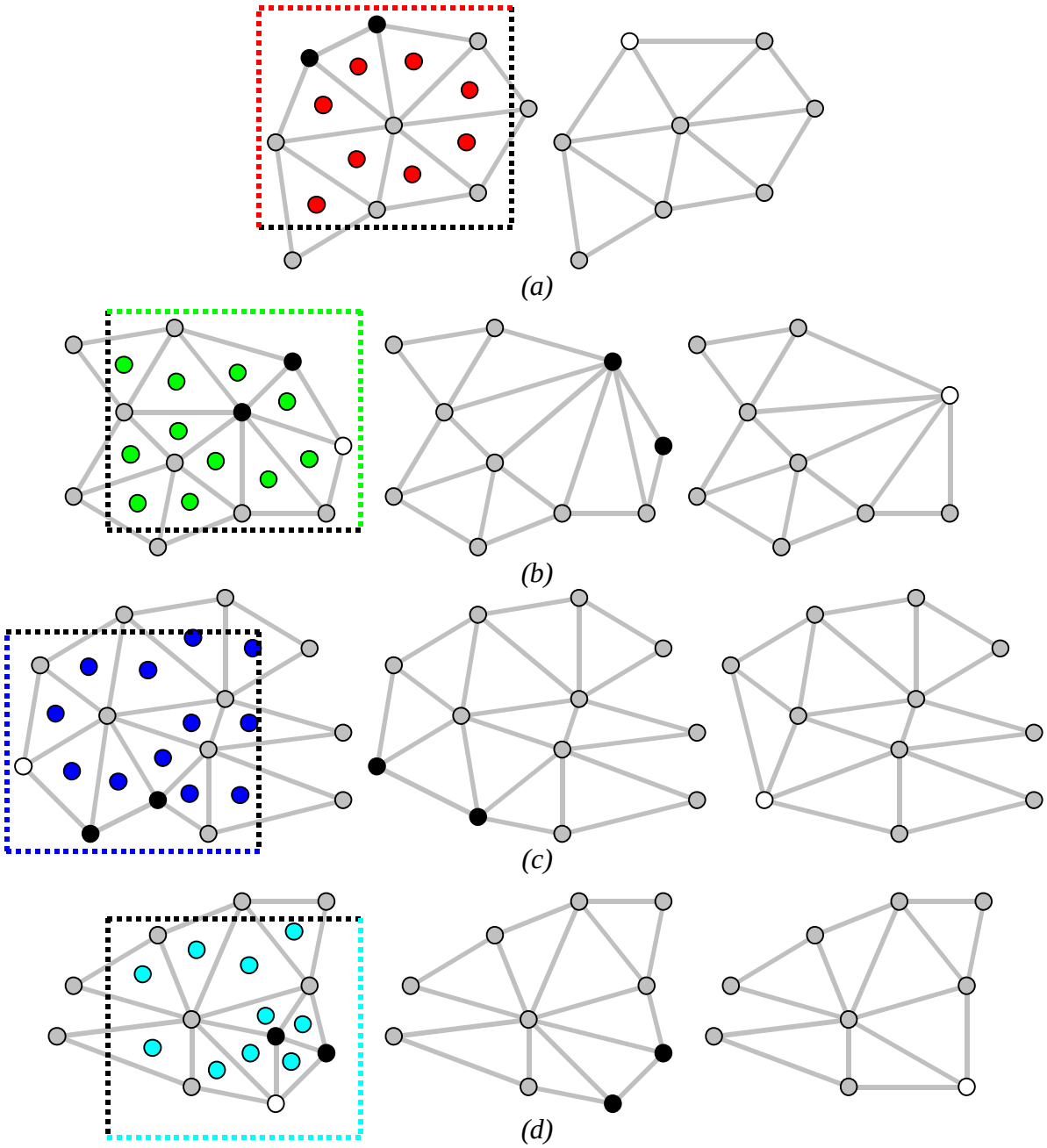


Figure 4.11: Simplification of each partition from Figure 4.10. *(a)* Simplification of the red partition. None of the gray restricted vertices are allowed to be merged. The next pair to be merged is colored black. Simplification stops when there are no more pairs to merge. *(b)* The green partition. *(c)* The blue partition. *(d)* The cyan partition.

When all partitions have been simplified independently, our algorithm associates them hierarchically, similar to the approach described in Section 4.2.2. However, since the partitions were created using a uniform subdivision, we take advantage of this information to

group them efficiently. We double *p* and group previously created partitions that fall within the same partition cube based on this new distance. When partitions are merged, the number of restrictions for each vertex is updated. In other words, if by doubling *p* a face that was once restricted becomes unrestricted, then the number of restrictions on each of the face's incident vertices is decremented by one. Therefore, some of these vertices that were once restricted may become unrestricted. This freeing of vertices enables GAPS to perform more vertex merge operations in order to create HLODs for this new hierarchical grouping of partitions. This process of grouping and simplifying is repeatedly applied until *p* is greater or equal to the size of the object's bounding volume. At this point, there is only one partition for the object and so there are no more vertex restrictions. Thus, GAPS can drastically simplify the root node of this partitioned scene graph to any target number of polygons. Figure 4.12 and Figure 4.13 demonstrate these concepts.

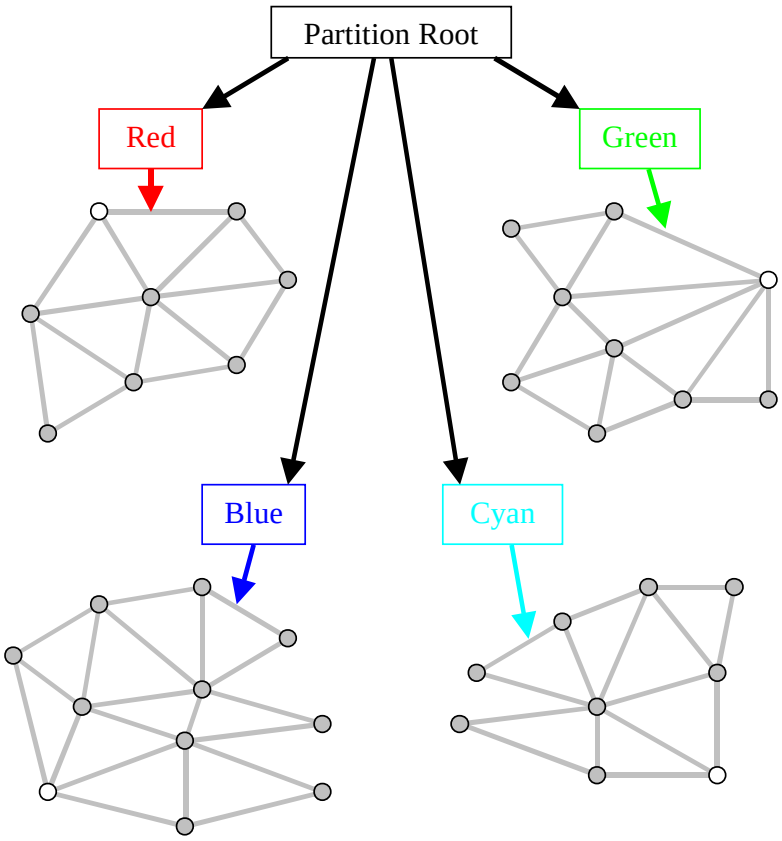
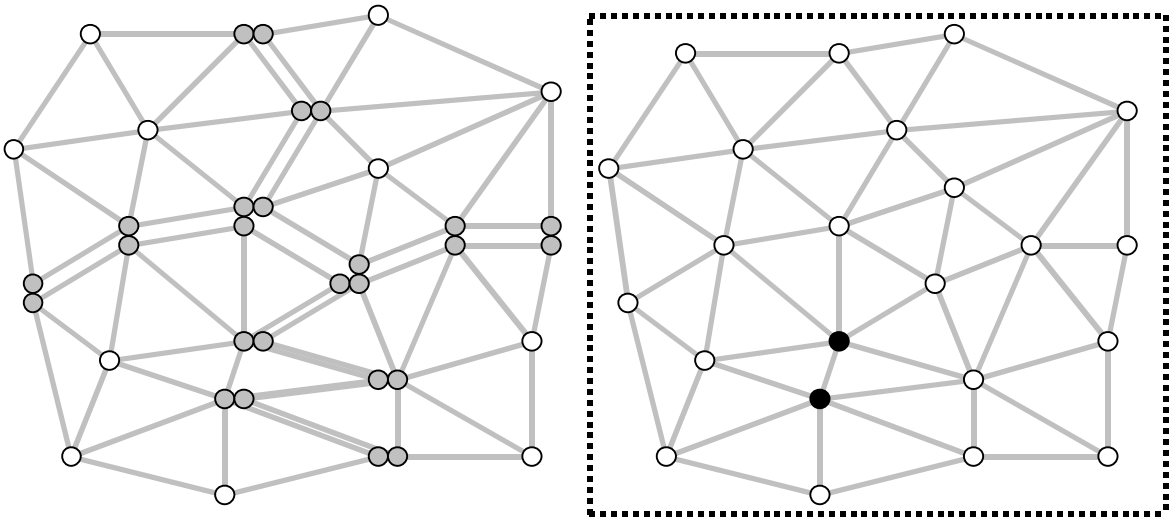


Figure 4.12: Creation of the partition scene graph for the scene in Figure 4.10 and Figure 4.11. The four quadrants are children of the partition root node. Shown below each quadrant node is its coarsest simplified geometry.



(a) *(b)*

Figure 4.13: Forming the polygonal geometry of the partition root node from Figure 4.12. *(a)* The polygons of each of the children nodes are combined. *(b)* The partition size shown as a dotted black rectangle doubles to include all of the polygonal geometry. Therefore, all vertices that were restricted are now free to be merged since they all lie in the same partition. Duplicate vertices are shared and HLODs for the partition root node are created. The black pair of vertices shows the next pair to be merged during this simplification process.

One can view this partitioning process as a discrete approximation to view-dependent simplification techniques. Since each leaf node is simplified independently of other partitions, partitions far away from the viewer will be rendered in lower detail while partitions near the viewer will be rendered in higher detail. When several partitions in close proximity are very far away from the viewer, they are rendered together using HLODs. In addition, partitioning allows us to view-frustum cull parts of the object which is not possible with traditional LODs. Finally, partitioning does not limit the amount of simplification possible since there are no vertex restrictions at the root node of the partition scene graph.

4.2.4 Targeting a Frame Rate with HLODs

Target frame-rate systems have the goal of rendering the best image possible within a user-specified frame-rate constraint. [Funkhouser and Séquin 93, Maciel and Shirley 95] show that targeting a frame rate using prediction techniques is a variant of the Knapsack

problem. Both of these algorithms rely on precomputed calculations of system performance that may not be accurate during run-time. [Rohlf and Helman 94, Mueller 95] describe reactive *feedback loops* to target a frame rate, meaning that the time it took to draw previous frames would be used to calculate the image quality for the next. Feedback loops suffer from the potential problems of *oscillation* and *hysteresis* and many steps must be taken to prevent them from occurring. Our algorithm, using a combination of predictive and reactive techniques, is capable of achieving a targeted frame rate due to its use of HLODs.

In our algorithm, we keep track of a target number of faces. This number is the best guess of how many polygons the system can render given the user-specified frame-rate constraint. It is this number that is updated reactively. Thus, if we could not render the number of faces within the frame-rate constraint, the target number of faces is decreased for the next frame. Therefore, before each frame is rendered, we have a strict limit on the number of faces that can be drawn. Suppose a user is viewing a model and comes upon a region of the model with a large number of textures. The user might encounter a slowdown in frame rate due to texture paging. Solely predictive systems, such as [Funkhouser and Séquin 93, Maciel and Shirley 95] will have problems with such a scenario since they measure system performance before the application runs in order to predict how much time a textured polygon takes to draw. Also, these algorithms will have problems with objects that are partially clipped by view-frustum culling. They will assume that the model takes a specific time to draw, but because of view-frustum culling and clipping, this time is not representative of the actual time needed to render the polygons. The method we use adapts the number of target polygons based on the performance of previous frames. Therefore, if the viewer were to enter a highly textured area, our algorithm would compensate for the slowdown.

We search the scene graph to determine which faces to render. We use a greedy method, one that at each step refines the node with the most projected pixel-error. We repeat this procedure until any refinement would cause the total number of faces to be above the target number of faces. The algorithm starts with the coarsest HLOD of the root node of the entire scene graph. It attempts to refine the node with the most screen-space error by replacing it with its children nodes. If replacing a node would cause our algorithm to render more polygons than the target number of faces, then this action is not allowed. We refine

nodes until no more nodes can be replaced. For example, the sequence of Figure 4.14, Figure 4.15, Figure 4.16, Figure 4.17, Figure 4.18, and Figure 4.19 shows our algorithm targeting a frame rate when rendering a simple scene.

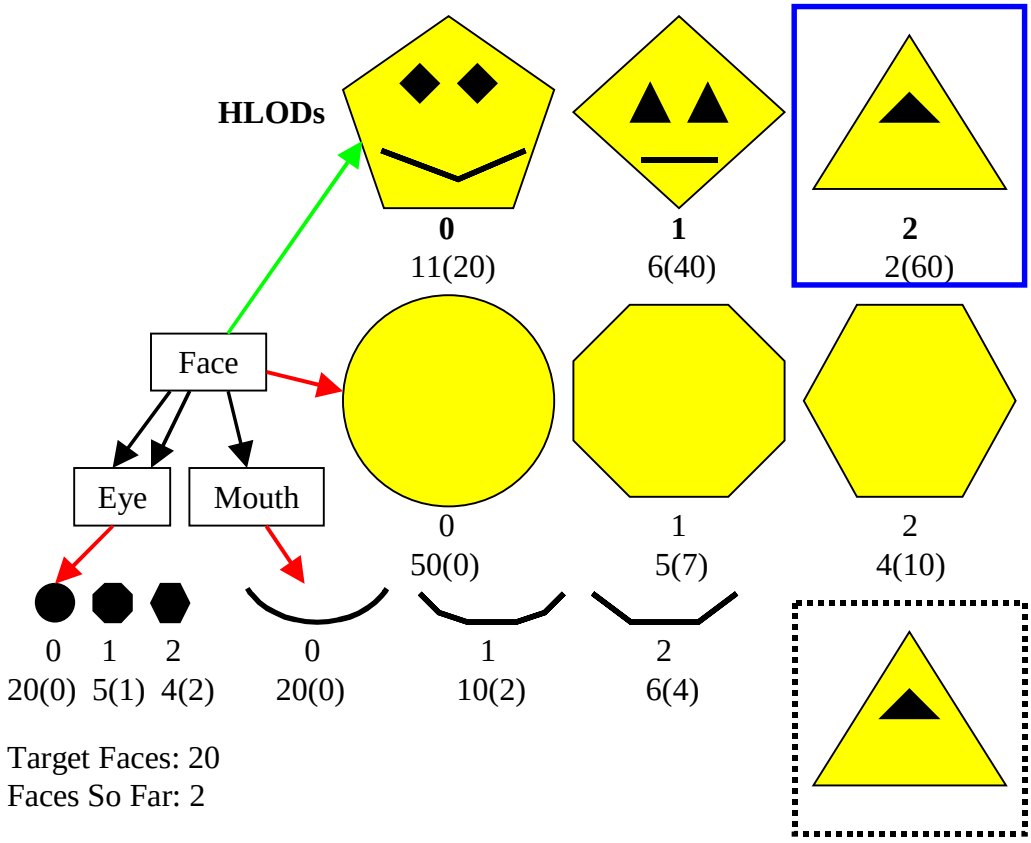


Figure 4.14: Example of targeting a frame rate. We start with the coarsest representation possible, namely the coarsest HLOD of the root node. Note that this HLOD represents the entire scene. The portions of the scene graph that are currently active are highlighted in blue. The number of faces that can be drawn within this example frame-rate constraint is 20. The current representation of the scene is 2 polygons. The polygonal geometry in the dotted black box is the current representation we would draw. The HLODs and LODs are numbered. Also, the number of polygons that make up an HLOD or LOD and the error associated with them is shown. For example, the coarsest HLOD of the root node consists of 2 polygons and has a projected pixel-error of 60, shown in parentheses.

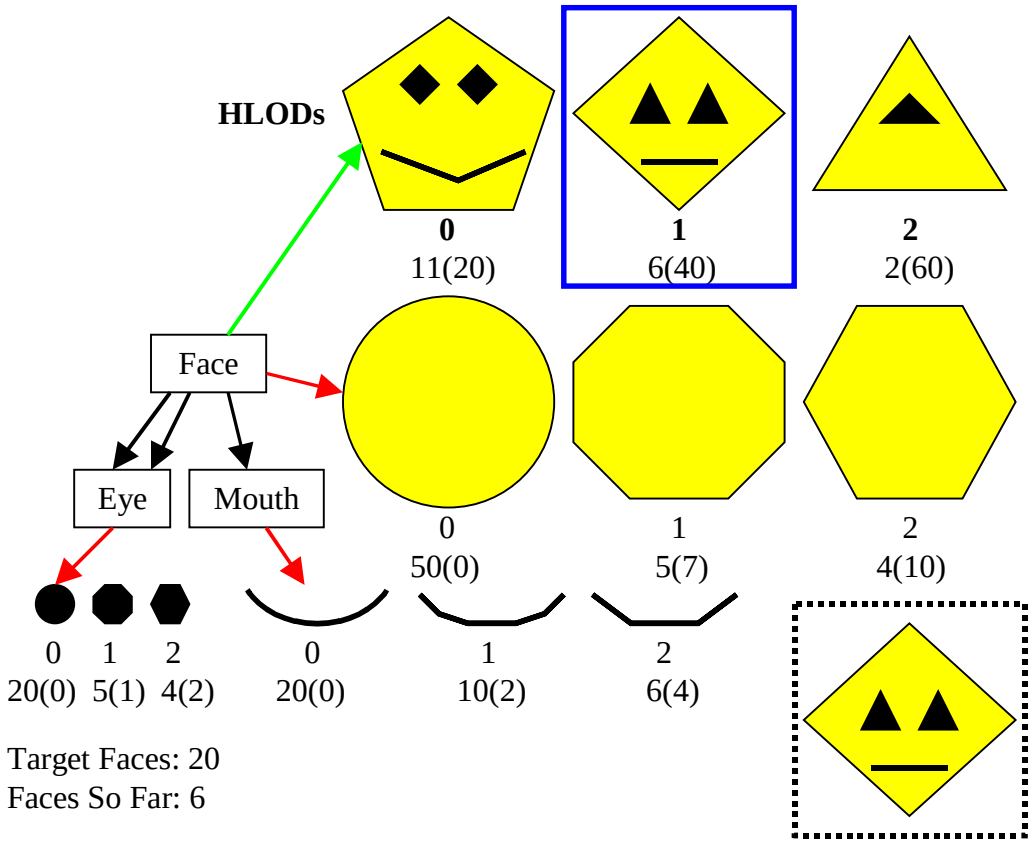


Figure 4.15: We refine the scene graph from Figure 4.14 since we can draw 18 more polygons. We substitute a finer HLOD for the coarsest HLOD.

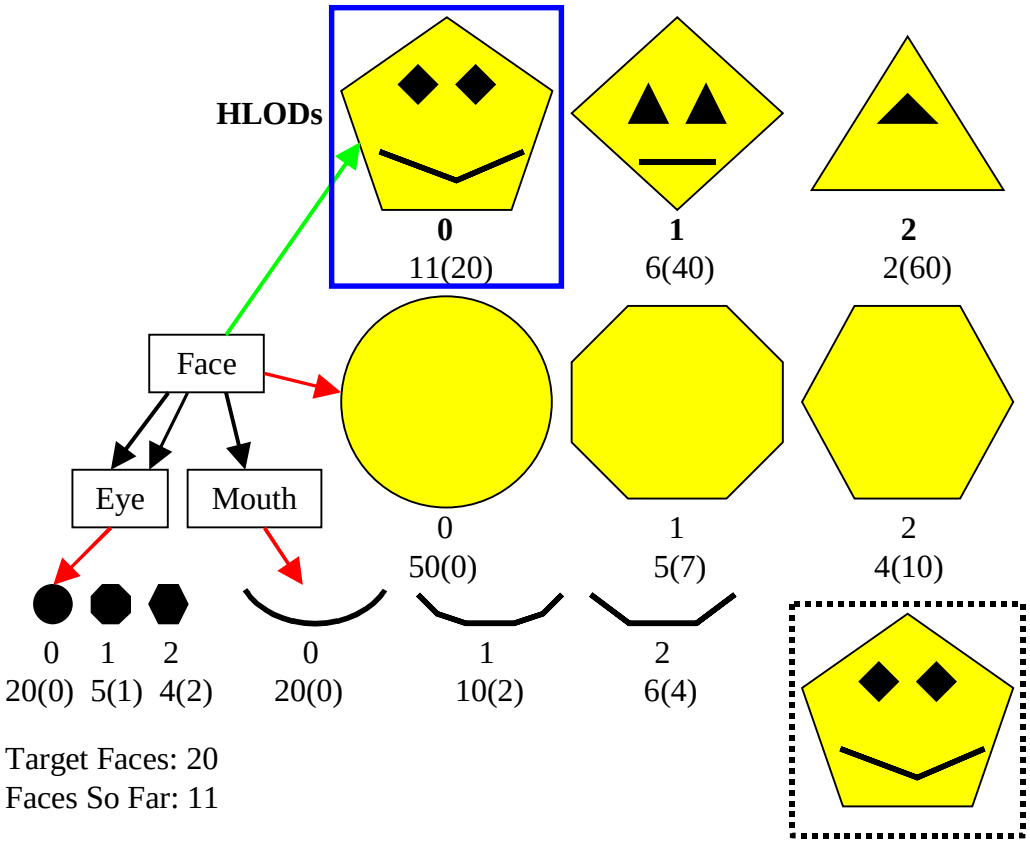


Figure 4.16: We again substitute a more detailed HLOD for the previous representation in Figure 4.15.

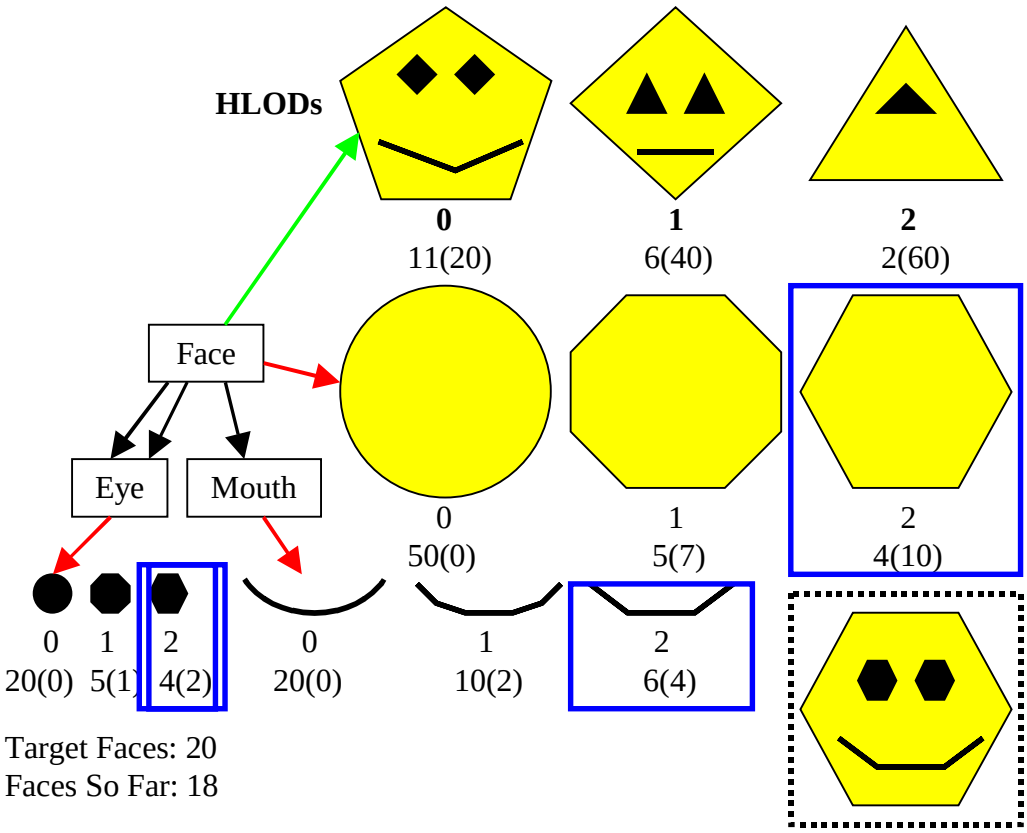


Figure 4.17: We refine further because the representation in Figure 4.16 is only 11 polygons. There are no more HLODs in the root node. To refine the previous HLOD, we descend into the scene graph and choose the coarsest LODs for each of the children nodes. Note that there are two blue boxes around Eye LODs, showing that there are currently two instances of eyes in the model.

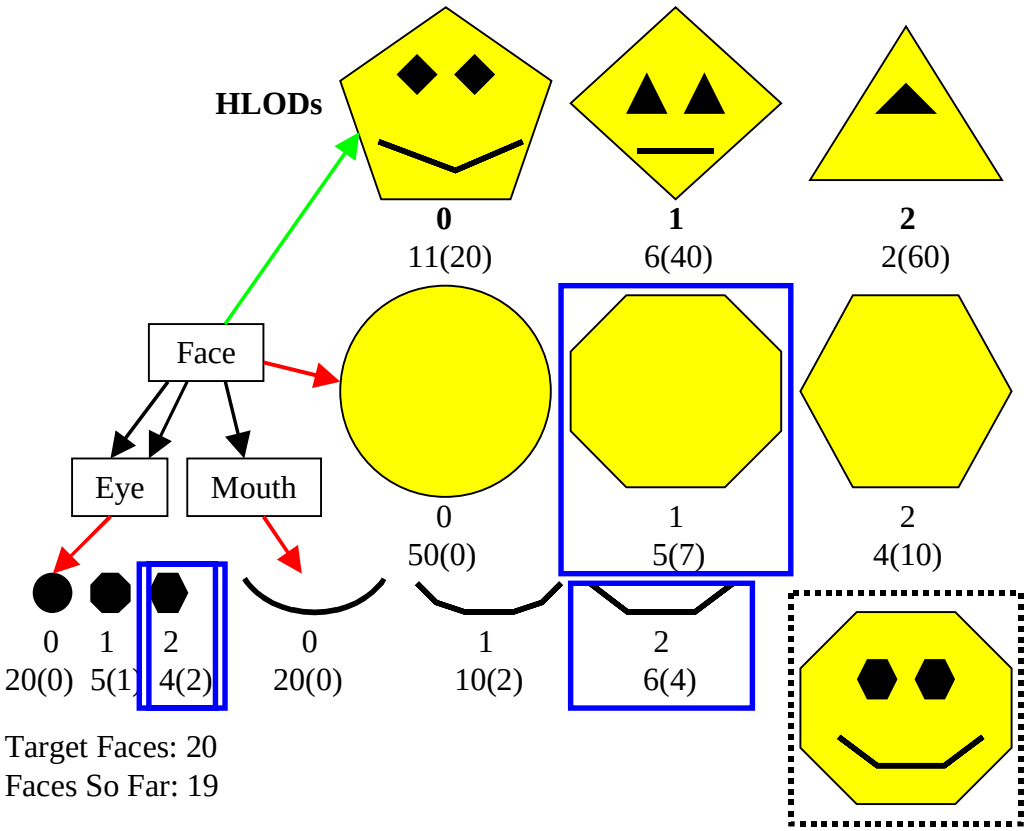


Figure 4.18: We refine the LOD or HLOD that exhibits the most error. In the previous representation in Figure 4.17, the Face polygonal geometry has a projected pixel-error of 10, which is greater than the error at the Mouth and Eyes. Therefore, we refine it first by choosing the next finer LOD of the Head.

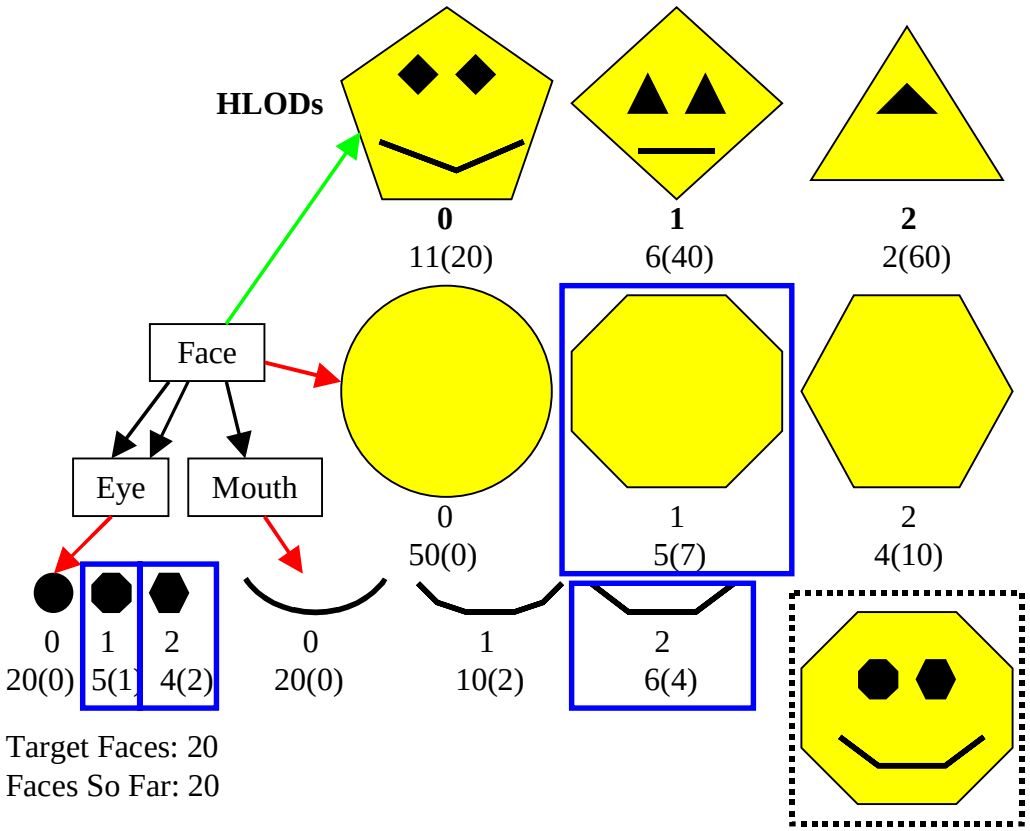


Figure 4.19: We still have one more polygon in our budget so we attempt to refine the Head LOD again since it exhibits the most error. However, we cannot refine it since it would add 45 polygons to the scene. Therefore, we attempt to refine other parts of the scene graph. The only LOD that can be refined and still be within the polygon budget is one of the Eyes. Once the Eye is refined, we have a representation of the scene that cannot be refined further without exceeding the polygon budget. The final image rendered is shown in the dotted black box. Note that the Eyes are rendered with different LODs.

4.2.5 Display Lists

Each object or group of objects is represented by a set of LODs or HLODs. Even spatially large objects are represented using a combination of partitioned LODs and HLODs. Since each LOD or HLOD is precomputed, we are able to store them in display lists. Display list rendering is 2 to 3 times faster than immediate mode rendering on high-end graphics systems [Aliaga 97]. Therefore, we can take advantage of the efficiency of display lists.

4.3 Implementation

We have implemented our visualization algorithm based on LODs and HLODs using C++, GLUT, and OpenGL. The code is portable across PC and SGI platforms. This application has allowed us to visualize extremely large models, including a nearly 13 million polygon Power Plant model.

4.3.1 Generality

Our algorithm, as a preprocess step, uses GAPS to produce all LODs and HLODs for the scene graph. As discussed in Chapter 3, GAPS is very general. In practice, we have used our algorithm on complex and degenerate CAD models.

4.3.2 LOD and HLOD Generation

The scene graph used by our algorithm has LODs and HLODs at every node. A node's LODs are created using GAPS by repeatedly halving the number of faces of the polygonal geometry. By doing so, we limit the amount of memory it takes to store the LODs of a node to at most double the memory of the original polygons. GAPS simplifies until the distance error associated with the node's simplified polygons is greater than a user-specified percentage of the size of the node's bounding sphere. At this point, GAPS halts simplification since any vertex merge operation on the remaining polygons would produce too much error. These remaining polygons are included as the base polygons for the node's first HLOD, along with the polygons of the coarsest HLODs from the node's children. Note that if an arc from the parent node to a child node contains a transformation, the children's polygons must be transformed into the parent's coordinate space. Again, HLODs are constructed using GAPS so that each successive HLOD has half the number of polygons as the previous one. By doing so, we limit the amount of memory it takes to store the entire model to at most double the memory of the original polygonal geometry of the whole scene graph.

Our current implementation could be improved in terms of the error metric GAPS uses when producing LODs and HLODs (see Section 3.3.3.4). Grouping the polygons of multiple nodes to create HLODs often produces better drastic approximations of objects rather than

using the individual LODs of each object. However, even though the HLOD looks better than a set of LODs, it is usually the case that GAPS reports a higher distance error for the HLOD than for any of the individual LODs.

The reason for this behavior is that GAPS ignores the quadric error metric when using surface area preservation (see Chapter 3). By using surface area preservation, GAPS temporarily prohibits certain vertex merges that an algorithm using the quadric error metric alone would perform. However, the geometric error term of our unified error metric uses the distance error associated with the error quadric at each vertex. Therefore, when we do not allow vertex merges, we are actually forcing another pair of vertices to merge. This new merge operation results in larger error according to our unified error metric. Therefore, GAPS reports that the individual LODs can be substituted closer to the viewer than their HLOD representation. In other words, the LODs have a smaller distance error associated with them as compared to the HLOD. However, usually the HLOD is visually a better approximation of the objects. We discuss this issue further in the future work section of Chapter 6.

4.3.3 Targeting a Frame Rate

Most of the time, the performance of the previous frame is a good predictor of the next frame. However, there are often cases where the performance from one frame to the next changes significantly. It is difficult to predict such abrupt changes as well as determine the reason for the change in performance. It could be due to a cache miss or that numerous polygons have suddenly come into view that were previously invisible. If we just use the previous frame time to predict the performance of the next frame, then the frame rate will sometimes oscillate from fast to slow when one of these events occurs. If a slowdown in frame rate is a one-frame occurrence, then the rendering system can probably ignore this "hiccup." However, if the frame rate slows for several frames, the rendering algorithm must adapt and render fewer polygons.

The first problem to overcome when implementing a target frame-rate algorithm is to estimate what the frame rate is. We estimate the frame rate by using a hierarchy of simple

moving averages. When the frame rate is changing rapidly, we use a two-frame moving average. If the frame rate is changing less rapidly, we use a four-frame moving average. If the frame rate is hardly oscillating, we use a ten-frame moving average. In other words, we estimate the frame rate by using the moving average representing the largest number of frames possible out of these three options. The user can specify a delta percentage that determines when a moving average is an acceptable representation of the true frame rate. If the two-frame moving average is within this percentage of the four-frame moving average, then the four-frame average is used. Similarly, if the four-frame moving average is within this delta of the ten-frame moving average, we use the ten-frame average. The user can modify this percentage during run-time and we have found that 10% to 20% gives good results. This percentage is also used to define a range of acceptable frame rates. In other words, suppose the user targets a frame rate of 20 frames per second but specifies a delta percentage of plus or minus 10%. It tells the system that 20 frames per second is the target, but 18 to 22 frames per second is acceptable. Allowing the system to target a range such as this and the use of a hierarchy of moving averages greatly cuts down on oscillation problems.

Suppose the target number of polygons for the previous frame was p_{old}, the target frame rate is f, our hierarchy of moving averages estimates the frame rate as f_{est}, and the user-specified delta percentage is δ. If $f_{est} \leq f - \delta f$ or $f_{est} \geq f + \delta f$ then the target number of polygons for the next frame is updated to p_{new} where

$$p_{new} = p_{old} \cdot f_{est} / f$$

4.3.4 Main Loop

The main loop of our algorithm is shown below.

- Handles input to change viewer position.
- If the algorithm is in pixel-error mode, it performs a depth-first search of the scene graph, locally terminating when an HLOD or LOD is reached that has an associated screen-space error less than the allowable pixel-error.

- If the algorithm is in target frame-rate mode, it traverses the scene graph to determine which polygons to render, terminating when the polygon budget is reached.

4.4 Results

We have tested our algorithm on three large CAD environments. The first model is a polygonal version of the Cassini spacecraft. The second environment is a Torpedo Room model. The final scene is of a nearly 13 million polygon Power Plant environment. We also use a small model of a Ford Bronco to demonstrate the effectiveness of HLODs as well as a Sierra Terrain model to show the benefits of partitioning. This terrain model is a single mesh that has no scene graph hierarchy.

4.4.1 Preprocessing Time

We use GAPS to create LODs and HLODs for polygonal scenes as a preprocessing step. Table 4.1 shows the amount of time it took to preprocess our test models. The number of objects, or leaf nodes in the scene graph, is shown in the table as well as the number of triangles that make up the model. Note that there is a certain amount of overhead to grouping together and simplifying polygons to create HLODs. The amount of overhead depends on the complexity of the scene graph and how much polygonal geometry is pooled to form each HLOD. There are two timing columns in this table. The first column shows the time it takes to create only LODs for each node in the scene graph while the second shows the time it takes to create both LODs and HLODs for the scene graph. Note that HLOD creation takes less time on the Sierra Terrain model than LOD creation. The reason for this difference in performance is that by partitioning the terrain model, GAPS is initially able to work on local portions of the terrain model independently. Simplification of subsets of polygons is faster than simplifying the entire polygonal geometry at once due to the performance behavior of GAPS. All of these tests were performed on an SGI Reality Monster with a 300 MHz R12000 processor and 16GB of main memory.

Scene	Objects	Triangles	LOD Time (secs.)	HLOD Time (secs.)
Bronco	466	74,308	31	43
Sierra Terrain	1	162,690	141	117
Cassini	127	349,281	172	195
Torpedo Room	356	883,537	450	495
Power Plant	1,179	12,731,154	14358	15136

Table 4.1: Preprocessing times for several polygonal environments.

4.4.2 Rendering Speed

For each test environment, we recorded a walkthrough path through the scene. The walkthrough starts with the entire object fully in view. The viewer first zooms out of the scene so that the entire environment is only a small portion of the screen. Then the viewer zooms into the model and explores its individual parts briefly. Finally, the viewer zooms out to approximately the original viewing distance. For each path, we tested our current implementation in five ways. First, we tested our algorithm on the model using no LODs and no display lists. Second, we used display lists, but no LODs. Third, we rendered the scene using LODs and display lists. Fourth, we used display lists, LODs, and HLODs. Finally, we tested the target frame-rate mode of our algorithm using a target of 20 frames per second. We ran all of these tests on an SGI Reality Monster with a 300 MHz R12000 processor and 16GB of main memory. We specified a transition between LODs and HLODs if the distance error during simplification was greater than 1% of the radius of the bounding sphere of an object. In other words, if we reached this error threshold when creating an LOD of a node, its remaining polygons would be used as base polygons for the HLOD of the node's parent. For the Sierra Terrain model we specified that the initial partition size be 10% of the radius of its bounding sphere. Partitions were simplified until the distance error due to simplification was greater than 1% of the size of the partition.

4.4.2.1 Immediate Mode Versus Display Lists

We tested each polygonal environment to validate our use of display lists over immediate mode rendering. Figure 4.20 shows the results. On most models, display lists accelerate the rendering of the model by around 40 to 50 percent. These timings include the overhead of scene graph traversal, including view-frustum culling. The Sierra Terrain model does not have a scene graph hierarchy and the performance increase is notably higher. Assuming there is no scene graph traversal overhead, then high-end SGI Infinite Reality machines render 2 to 3 times faster using display lists as compared to immediate mode rendering [Aliaga 97].

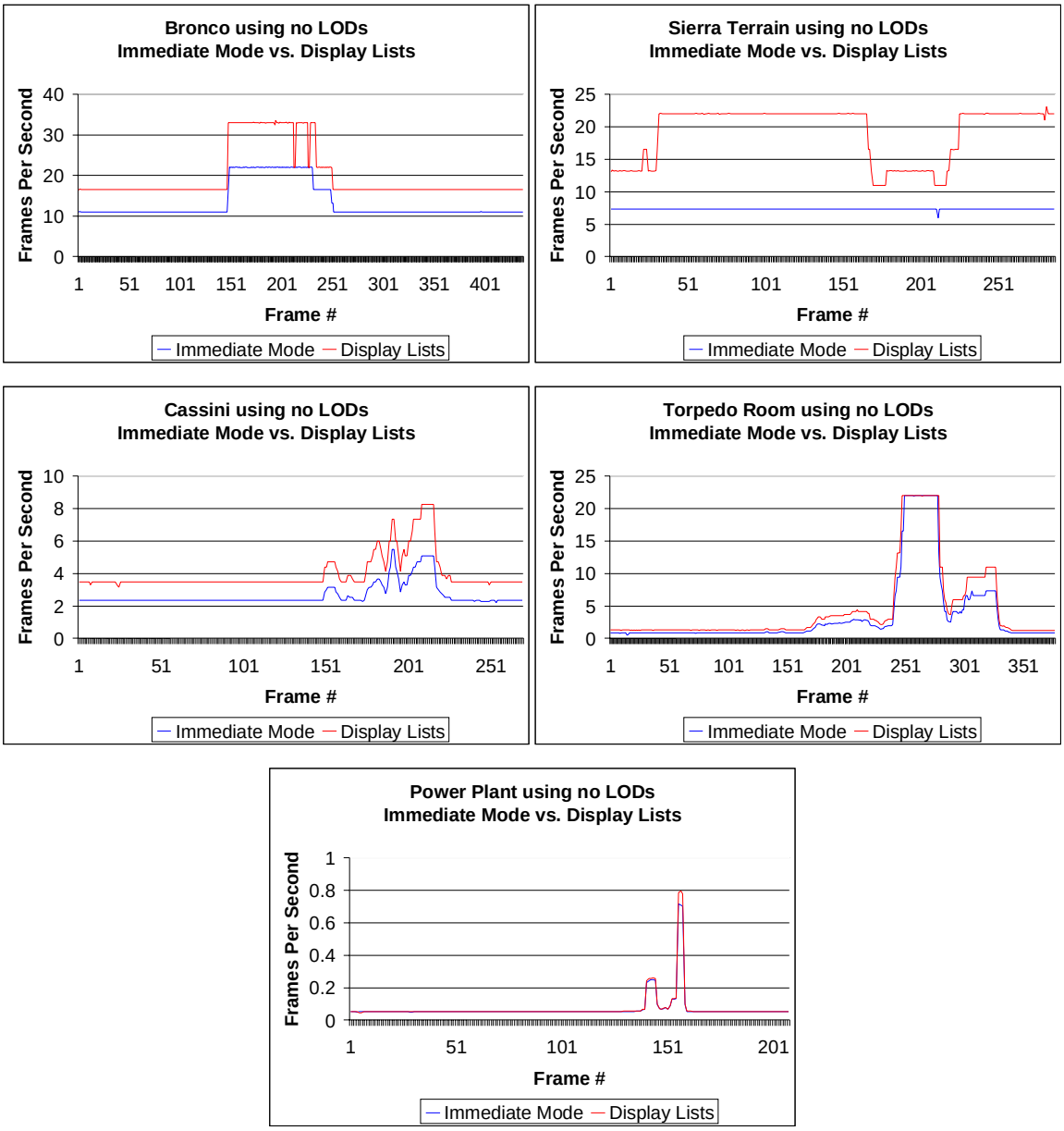


Figure 4.20: Performance difference between display lists and immediate mode on a SGI Reality Monster with a 300 MHz R12000 processor and 16GB of main memory.

For the Bronco, display lists outperform immediate mode by 49.4% on average. Display lists outperform immediate mode by 162.6% on average on the Sierra Terrain model. Since the Sierra Terrain model has no scene graph hierarchy, there is no traversal overhead which explains this higher gap in performance. We conjecture that the dips in the display list performance for this model are due to the cost of clipping numerous polygons when we zoom in on the terrain. For the Cassini, display lists outperform by 48.7% on average. On the

Torpedo Room model, they accelerate rendering by 44.5% on average. For the Power Plant model, they accelerate rendering by 3.0% on average. Note that the Power Plant graph shows almost equivalent performance between immediate mode and display lists. We conjecture that since only a small percentage of this massive Power Plant model can fit into the display list cache of the machine, that display lists have little effect when rendering all 13 million polygons. The use of LODs and HLODs would alleviate this problem.

4.4.2.2 No LODs Versus LODs

To demonstrate the effectiveness of LODs, we measured the performance of rendering using LODs versus without. Display lists were used during these timing runs and we selected a pixel error tolerance of 5 pixels on a 1000 by 1000 pixel window. There was little noticeable loss in image quality using this error tolerance. Figure 4.21 shows the results on our five test models.

For brief periods of time, rendering with no LODs is faster than rendering with LODs, probably due to display list cache misses when switching between LODs. The performance is equivalent during stretches when the viewer is very close to objects in the scene since we render the scene's original geometry in both cases. For the Bronco model, we achieve a 3.18 times average speedup by using LODs. For the Sierra Terrain model, we achieve a 1.72 times average speedup by using LODs. We render the Cassini model on average 9.97 times faster using LODs. We achieve a 9.07 times average speedup on the Torpedo Room model. For the Power Plant, our system renders on average 305.88 times faster using LODs. Clearly, this average speedup depends on the viewing path we selected. Note that in the Power Plant graph, the frame rate when using no LODs is so low that it is barely visible.

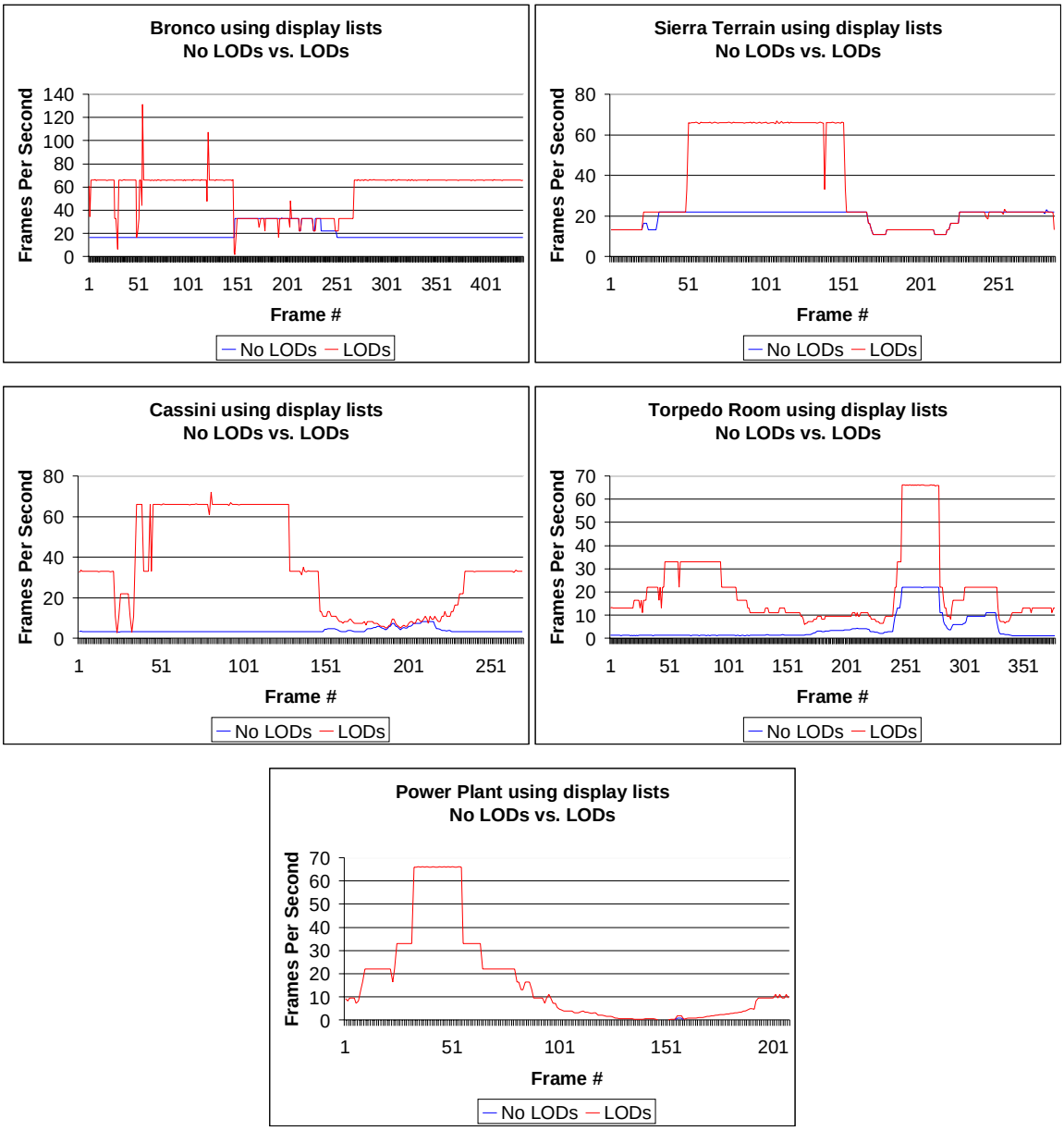


Figure 4.21: Performance difference between using LODs and not using LODs.

Another statistic we gathered was the performance difference if the model is totally within the viewing window, but takes up most of the viewing screen. We render the Bronco model at 66 frames per second using LODs and 16.5 frames per second using no LODs, a 4 times speedup. The Sierra Terrain model is rendered at 13 frames per second using LODs and 13 frames per second using no LODs. There is no performance difference on this model because in order to meet the 5 pixel-error bound, we must render the highest LOD, i.e,. the original polygonal geometry. We render the Cassini model at 33 frames per second using

LODs and 3.5 frames per second using no LODs, a 9.4 times speedup. The Torpedo Room model is rendered at 13 frames per second using LODs and 1.3 frames per second using no LODs, a 10 times speedup. We render the Power Plant model at 10 frames per second using LODs and 0.05 frames per second using no LODs, a 200 times speedup.

4.4.2.3 LODs Versus HLODs

Our next series of performance tests compare using only object LODs to using both object LODs and HLODs for nodes in the scene graph. To be fair, we allowed 5 pixels of error in a 1000 by 1000 pixel viewing window for timing runs in both configurations. This amount of pixel error results in little or no loss in image quality. However, as described in Section 4.3.2, sometimes our unified error metric is not a reliable indicator of relative visual quality of approximations. Therefore, by specifying the same pixel error for only LODs and for LODs and HLODs, the current implementation of the system will usually render more polygons when using HLODs. Figure 4.22 shows the results of our timing runs. Given the 5-pixel error, there was no apparent difference in image quality between using only LODs and using LODs and HLODs together. Both configurations produced images that had little loss in image quality compared to rendering the original model.

For these test runs, using HLODs caused a 5.4% average slowdown on the Bronco, a 75.3% average speedup on the Sierra Terrain, a 0.1% average slowdown on the Cassini, a 5.4% average slowdown on the Torpedo Room, and a 2.9% average slowdown on the Power Plant model. Most of these results show that HLODs slow down the current implementation of our system. However, as will be shown in Section 4.4.4, HLODs are usually higher quality approximations for closely spaced groups of objects as compared to the individual LODs of these objects. The reason that HLODs greatly accelerated the rendering of the Sierra Terrain model is mostly due to partitioning and view-frustum culling. If we use LODs for the terrain mesh then it is either completely in or completely out of the view-frustum. When we partition the model, some partitions may fall outside the view-frustum and can be culled.

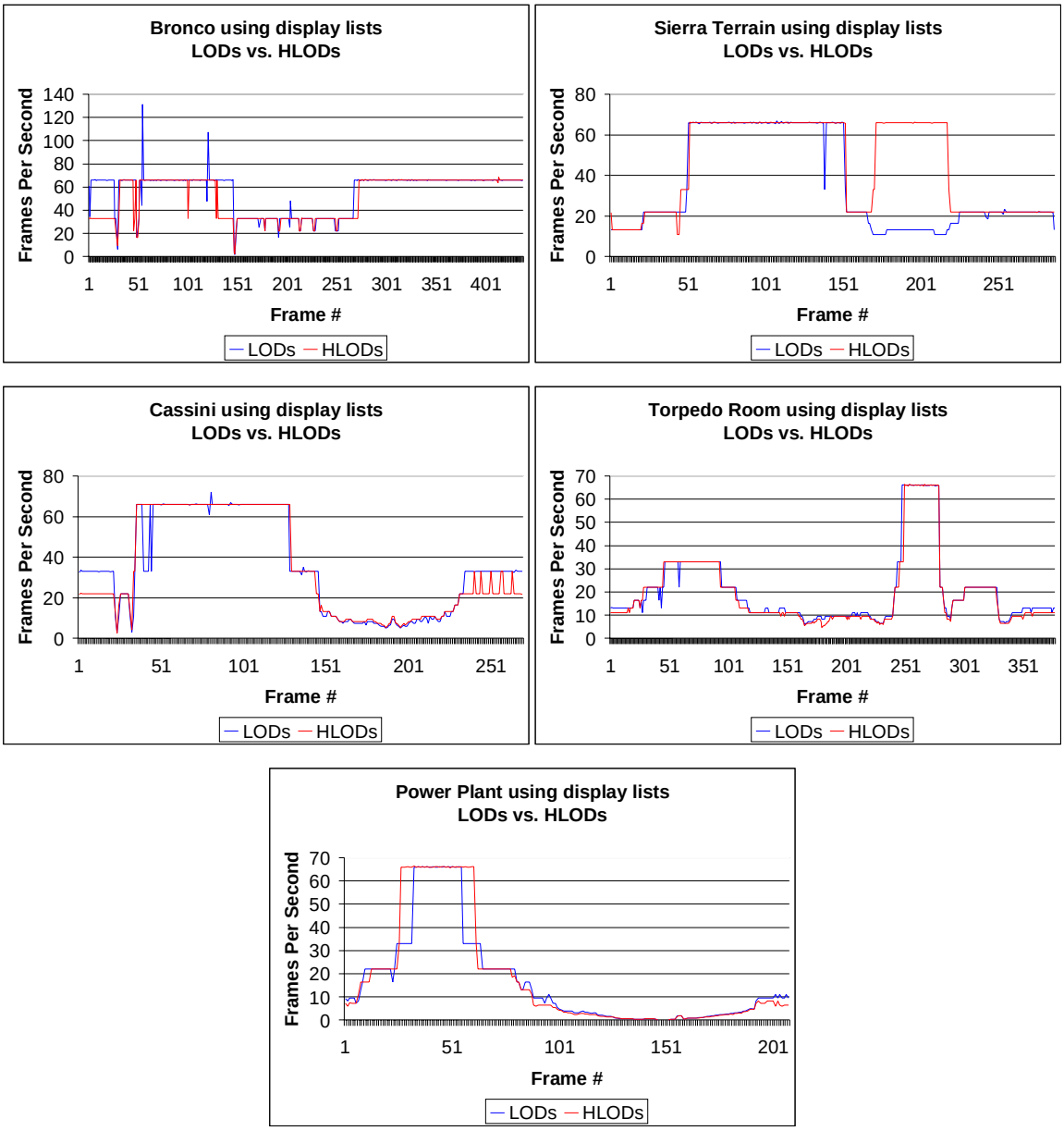


Figure 4.22: Performance difference between using LODs versus using LODs and HLODs.

4.4.2.4 Targeting a Frame Rate

We measured the performance of our target frame-rate mode to see how well the algorithm would react to paths through several different models. Figure 4.23 shows the target frame rate results on the four test environments. Note that our algorithm knew nothing about the specifications of the SGI Reality Monster on which it was running. In fact, our algorithm

can run on any graphics system without knowing its performance specifications. It does so by combining both reactive and predictive techniques as described in 4.2.4.

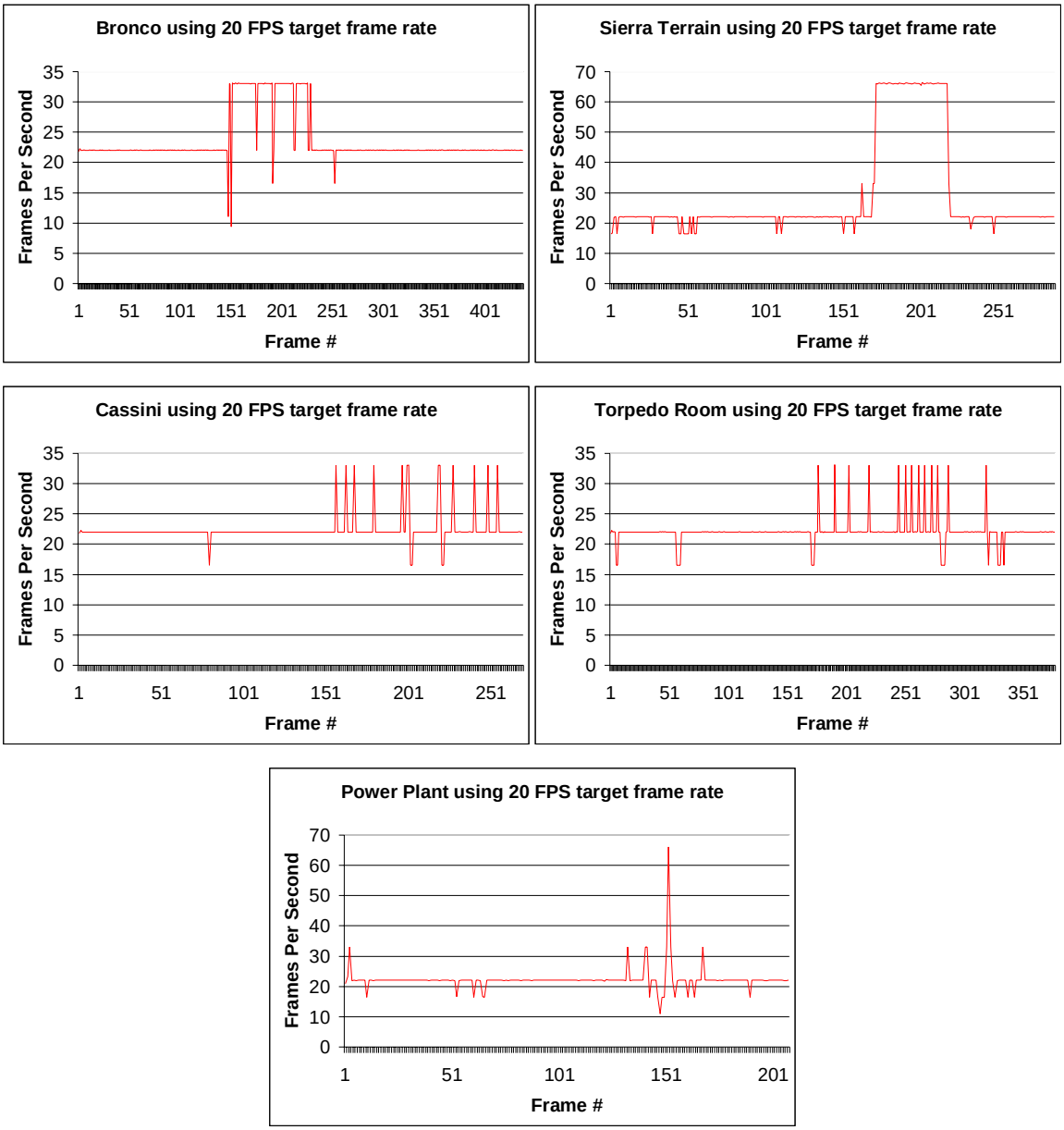


Figure 4.23: Performance of our target frame-rate mode.

For each scene, we used 20 frames per second as our target and our delta percentage (see Section 4.3.3) was 15%. Therefore, an acceptable frame rate for the algorithm was in the range 17 to 23 frames per second. For a majority of the time, our method rendered each model within the acceptable range. However, for each viewing path of each model there are sharp transitions in polygonal geometry, causing low or high spikes in these performance

graphs. Our algorithm is able to quickly react to these changes and bring the frame rate within acceptable bounds. Sometimes we render much faster than the target frame rate. In these cases, we are rendering the original polygons of the model, but there is not enough of it in the view frustum to keep a constant frame rate. This behavior is not really a problem since the frame rate could easily be clamped if so desired.

4.4.3 Memory Usage

As described in Section 4.3.2, we create a series of levels of detail or hierarchical levels of detail such that LODs consist of half the number of polygons of the previous LOD. By doing so, we limit the memory usage due to the geometry of the scene to double that of its original geometry.

4.4.4 Visual Comparison

So far, there has been no performance evidence that HLODs actually help the visualization of large polygonal models. We will show many visual results of the five models examined in this section. The effectiveness of HLODs depends on the model and whether its objects are closely spaced together. For example, HLODs would not help much for a model of the universe, since objects such as stars are generally not close to other stars. Therefore, not much merging would occur between different objects so using only LODs would be just as effective. However, some models consist of tightly packed, but independent objects. HLODs are most effective on these scenes, where an abundance of polygonal geometry from different objects is close together.

We will show a specific view of four of the models using three different methods. The first method uses the quadric error metric alone to create LODs for the scene. The second technique uses GAPS to create LODs for the model. The last method uses GAPS to create LODs and HLODs for the environment. These images show how GAPS promotes the merging of unconnected regions of geometry as compared to the quadric error metric alone. However, it also shows that by creating only LODs, even GAPS cannot create acceptable drastic approximations of these models. By creating HLODs, GAPS is able to combine

geometry from different objects in the scene to produce higher quality drastic approximations. One thing to notice is that by promoting the merging of unconnected regions of objects, GAPS must sometimes choose merging over the preservation of well-defined geometry. Therefore, HLODs tend to look "fuzzy" as compared to the individual LODs. However, the overall shape tends to better represent the whole set of polygonal geometry. The next series of figures show a visual comparison of the three methods on our four test scenes.

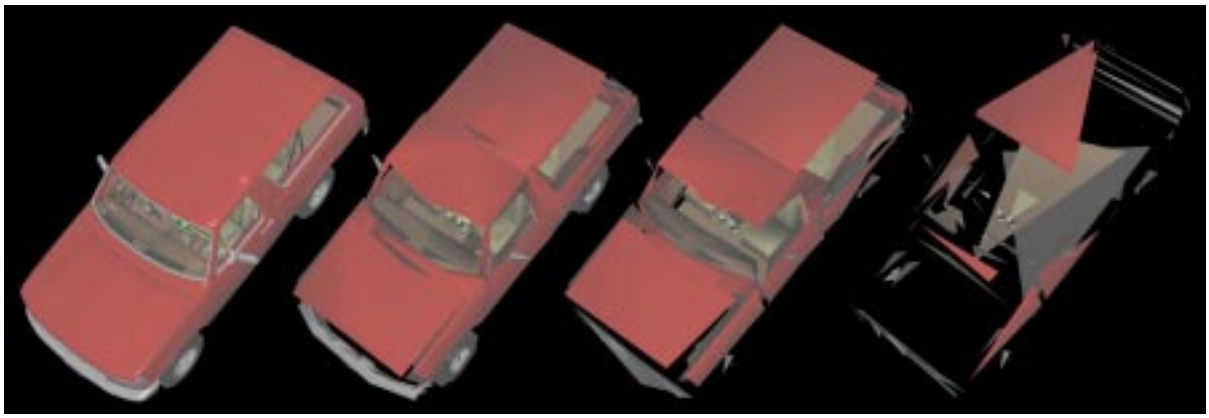


Figure 4.24: LODs created for the Bronco model using the error quadric metric alone. They consist of 74,308 faces (the original model), 1,366 faces, 343 faces, and 107 faces.

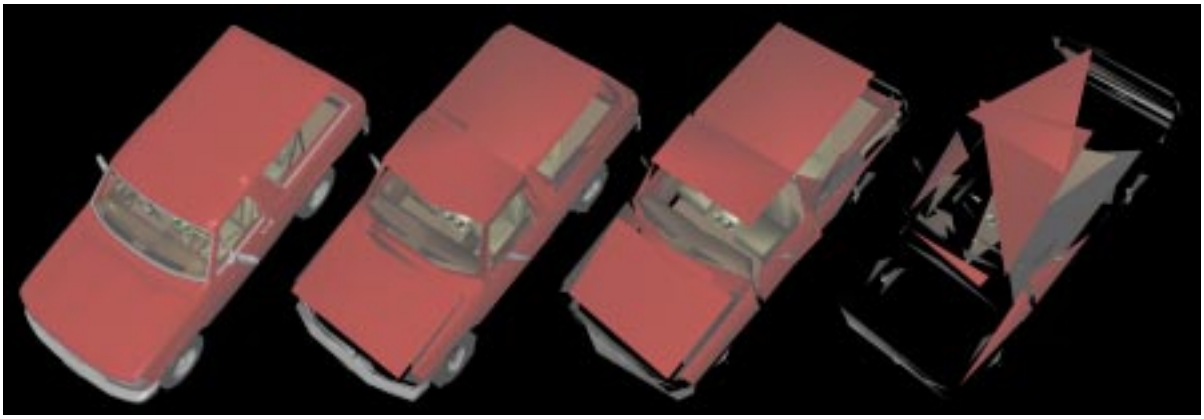


Figure 4.25: LODs created for the Bronco model using GAPS. They consist of 74,308 faces, 1,357 faces, 341 faces, and 108 faces.

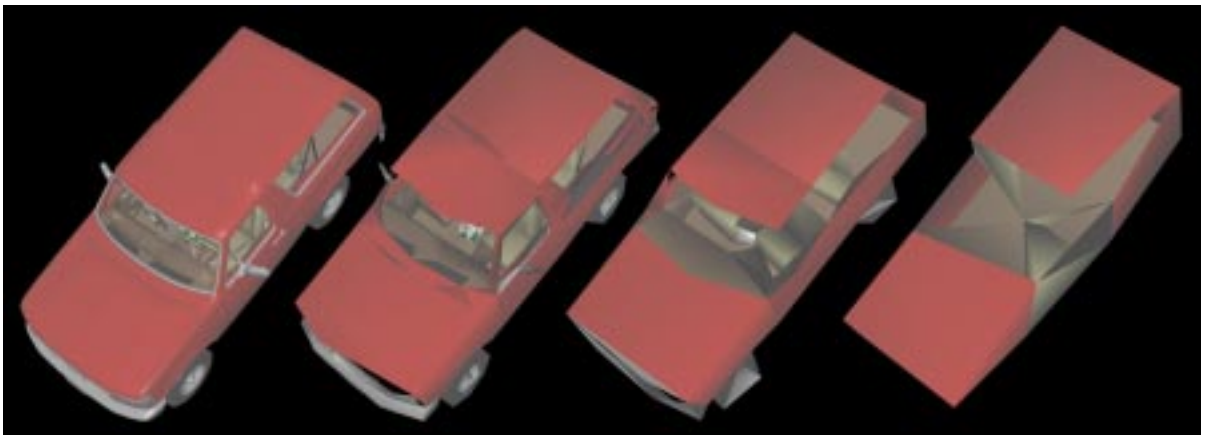


Figure 4.26: LODs and HLODs created for the Bronco model using GAPS. They consist of 74,308 faces, 1,357 faces, 338 faces, and 80 faces.

Notice that if we use LODs alone, the Bronco splits apart at the seams at low polygon count approximations. Using HLODs, we are able to preserve the general shape of the Bronco further into the simplification process. Note that in the 300 polygon approximations, the HLOD version has a fuzzy interior while the GAPS LOD version has sharper geometry in its interior. This example shows how HLODs sacrifice individual areas of polygons to better approximate the whole set.

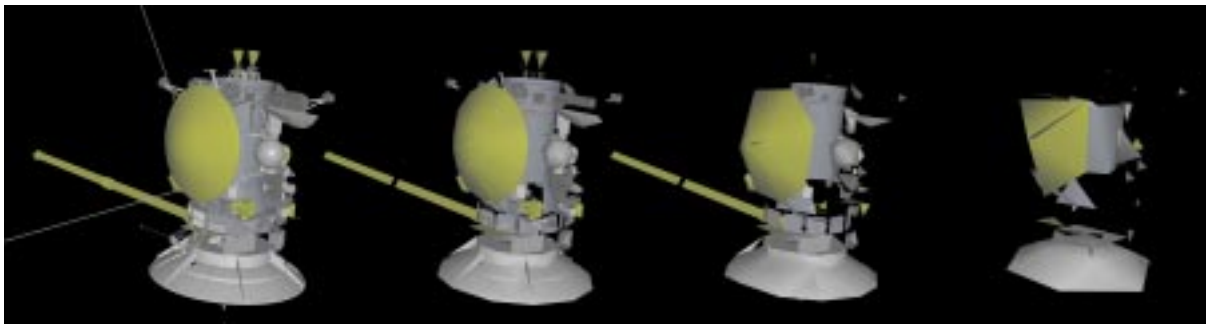


Figure 4.27: LODs created for the Cassini model using the error quadric metric alone. They consist of 349,281 faces (the original model), 3,629 faces, 939 faces, and 226 faces.

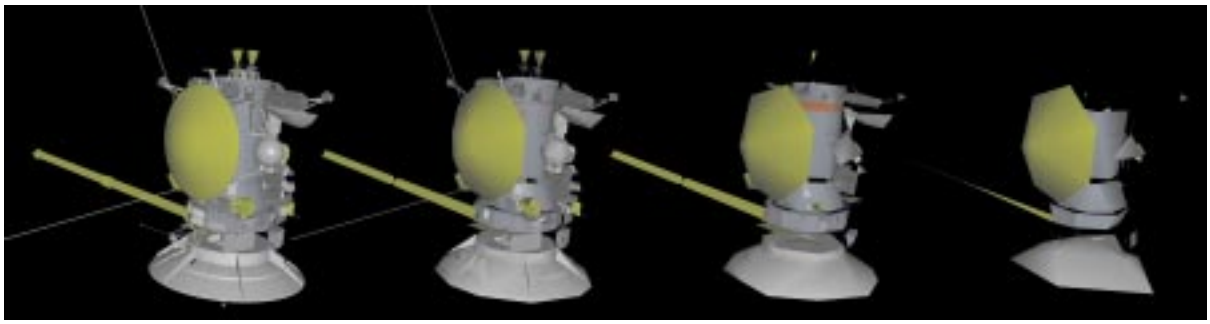


Figure 4.28: LODs created for the Cassini model using GAPS. They consist of 349,281 faces, 3,601 faces, 906 faces, and 228 faces.

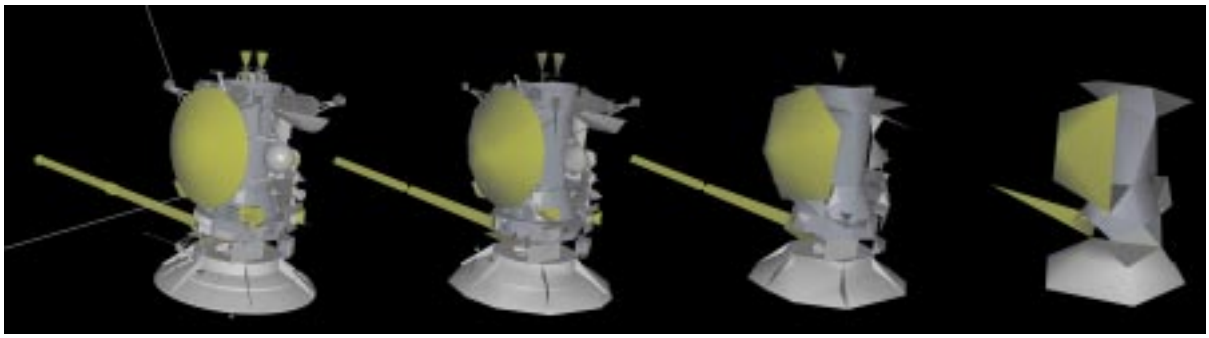


Figure 4.29: LODs and HLODs created for the Cassini model using GAPS. They consist of 349,281 faces, 3,587 faces, 892 faces, and 217 faces.

Notice that the LODs created using the quadric error metric alone break apart at the seams at low polygon count approximations. GAPS does a better job with its LODs, but holes between the individual objects of the Cassini are still visible at coarse approximations. Using HLODs, we are able to preserve the general shape of the Cassini further into the simplification process.

Figure 4.30: LODs created for the Torpedo Room model using the error quadric metric alone. They consist of 883,537 faces (the original model), 6,386 faces, 827 faces, and 100 faces.

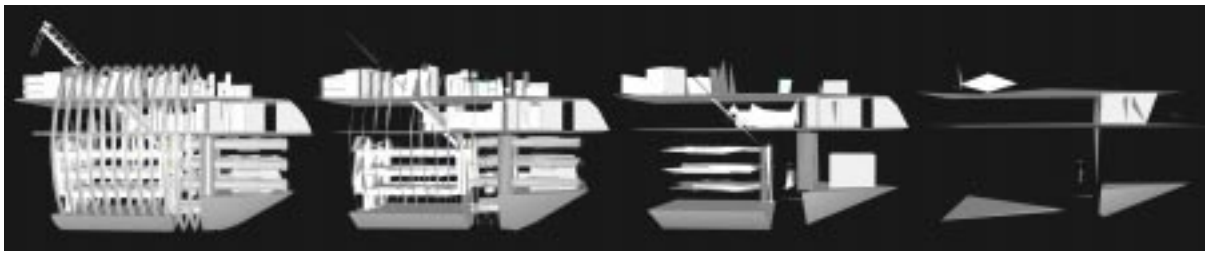


Figure 4.31: LODs created for the Torpedo Room model using GAPS. They consist of 883,537 faces, 6,160 faces, 822 faces, and 95 faces.

Figure 4.32: LODs and HLODs created for the Torpedo Room model using GAPS. They consist of 883,537 faces, 6,160 faces, 822 faces, and 95 faces.

Notice that the majority of the geometry disappears at low polygon count approximations if we do not use HLODs. Using HLODs, we are able to merge unconnected objects and produce a more solid shape at drastic approximations. The last HLOD is coarse, but still provides a better solid approximation of the whole Torpedo Room than the equivalent set of LODs.

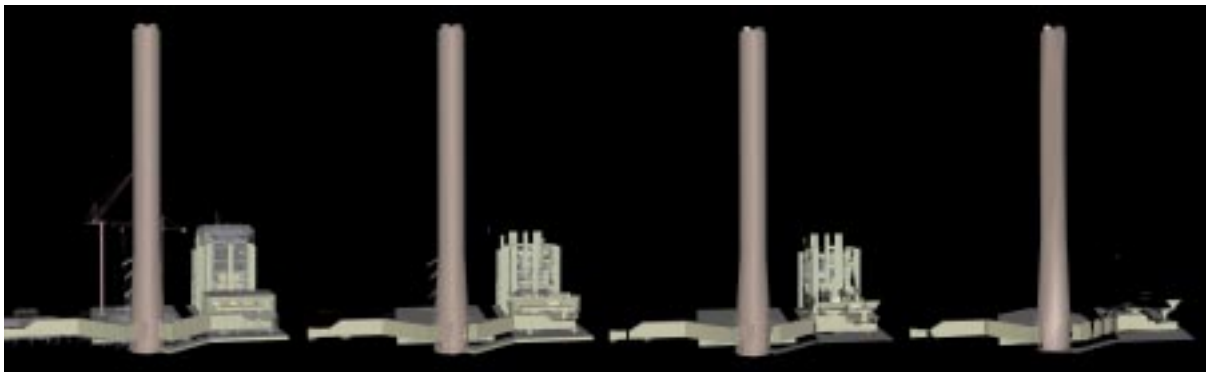


Figure 4.33: LODs created for the Power Plant model using the error quadric metric alone. They consist of 12,731,154 faces (the original model), 9,627 faces, 2,494 faces, and 607 faces.

Figure 4.34: LODs created for the Power Plant model using GAPS. They consist of 12,731,154 faces, 9,558 faces, 2,405 faces, and 612 faces.

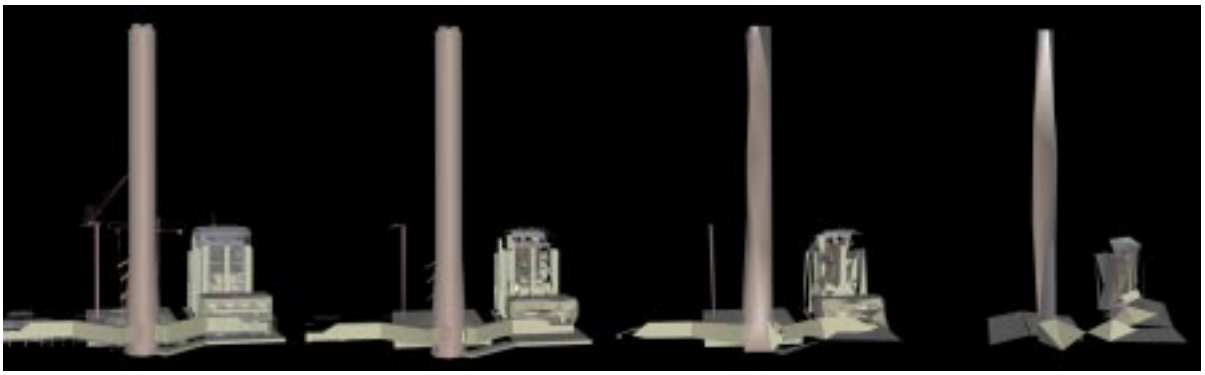


Figure 4.35: LODs and HLODs created for the Power Plant model using GAPS. They consist of 12,731,154 faces, 9,503 faces, 2,375 faces, and 590 faces.

The Power Plant consists of a large smokestack, a central building housing numerous complex pipe structures on the right of these images, and other structures. Using the quadric error metric alone makes the central building completely disappear at drastic approximations. LODs created by GAPS are able to preserve the central building further into the simplification process, but most of the interior of the building vanishes. Using HLODs, we are able to merge a majority of the central building into a solid block at low polygon counts. Unfortunately, in the process we have made the left edge of the model vanish. Note how the HLODs merge the smokestack with its surrounding geometry.

In order to visualize complex environments, such as the Power Plant model, at a target frame rate of 20 frames per seconds, sometimes image quality has to be sacrificed by substituting coarse approximations for the original polygonal geometry (see Figure 1.11 in the introduction chapter). Depending on the graphics system and the complexity of the model,

these coarse approximations could replace objects that are very close to the viewer. In these cases, we cannot hope to render a high quality image. However, we can hope to preserve the general shape of the region in view such that the image conveys a general impression of the objects involved. As shown by the previous images in this section, HLODs are better visual representations for objects in close proximity as compared to the individual LODs of these objects. Therefore, HLODs make a large impact on the visual quality of our target frame-rate mode.

One might think an acceptable approach for a model like the Bronco would be to flatten the scene graph hierarchy and treat the model as one object. There are two reasons not to flatten the model. The first reason is that by flattening the Bronco's hierarchy, one eliminates the rendering algorithm's ability to choose an LOD for each object in the scene. This ability allows objects far from the viewer to be coarser than ones that are near. Second, the objects in the scene might move. Suppose a user wanted to edit the Bronco model and move a door out of the way so he or she could better see into its interior. By flattening the hierarchy, one eliminates any possibility of dynamic movement of these individual parts. We address dynamic environments in Chapter 5.

We do not show the same set of images for the Sierra Terrain model because it is a single mesh. Thus, the quadric error metric, GAPS, and GAPS using HLODs will all look the same. To look at LODs of the Sierra Terrain model, see Figure 3.30 and Figure 3.31. However, the images shown in Figure 4.36 and Figure 4.37 demonstrate the power of partitioning on this model. Most of the performance gain when rendering a partitioned version of this model comes from view-frustum culling, rather than the hierarchical structure of the partition scene graph (see Section 4.4.2.3).

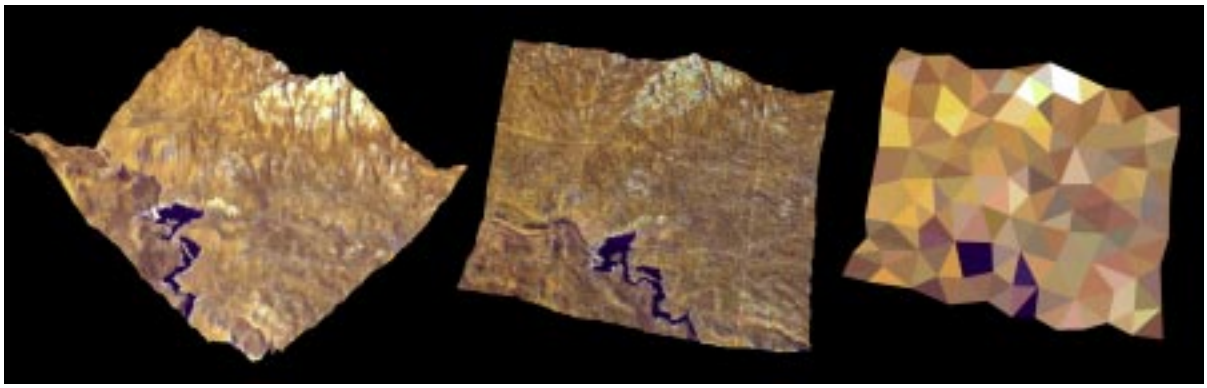


Figure 4.36: Partitioning the Sierra Terrain model.

In Figure 4.36, the left image shows the original Sierra Terrain model consisting of 1 object (the mesh itself) and 162,690 polygons. The middle image shows a rendering of the partitioned terrain model in wire frame using 83,559 polygons. Notice the faint boundaries within the model. These boundaries are noticeable because our algorithm preserves them during simplification to avoid cracks between partitions. Finally, the last image shows that even though we use partitioning, we are still able to simplify the model drastically. It shows the geometry of the root node of the partition graph consisting of 168 polygons. As described in Section 4.2.3, the root node does not have any remaining vertex restrictions and thus can be simplified to any target number of polygons.

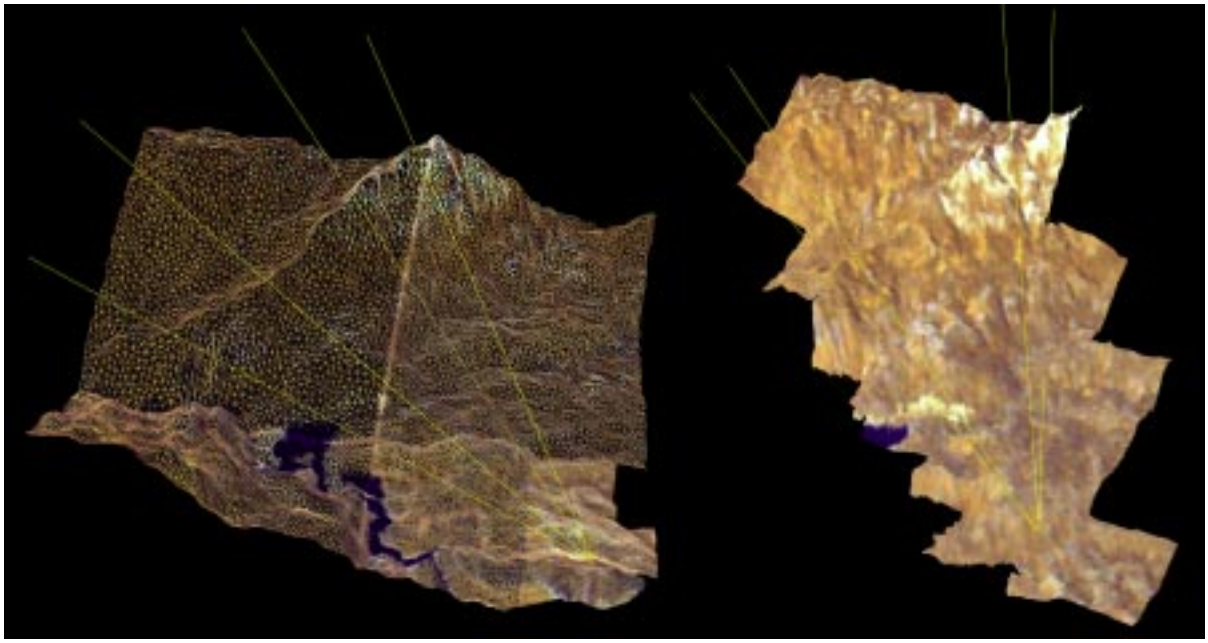


Figure 4.37: Adaptive simplification and view-frustum culling using partitioning.

In Figure 4.37, the left image in wire frame demonstrates the ability of our system to render using adaptive simplification. The view-frustum is colored yellow and there are 35,591 polygons rendered. Partitions near the viewer are drawn in high detail while partitions far away from the viewer are drawn in lower detail. The right image has 74,359 polygons and shows that by partitioning the model, we achieve more efficient view-frustum culling. If we only used LODs for this terrain, we would not see any view-frustum culling since the model is one mesh and part of it is in view.

4.4.5 Sweetening Mode

For very complex environments such as the Power Plant model, it is nice to be able to move around at 20 frames per second using the target frame-rate mode. However, once the viewer stops moving, usually they want to look at the scene in finer detail. Therefore, the user will normally switch to pixel-error mode and specify a low threshold of error in order to view a high quality image. We combine the two rendering modes into a *sweetening* mode. When the viewer is moving, we render using the target frame-rate mode. When the viewer stops, we progressively refine the image [Bergman et al. 86, Airey et al. 90]. If the viewer stays still for any length of time, we eventually render the original geometry of the scene. As soon as the viewer starts to move, we switch to our target frame-rate mode. We have found that this sweetening mode is very useful for quickly finding places of interest in the model and then viewing these locations in high detail without having to switch modes manually.

4.5 Analysis

There is not much analysis we can perform on the run-time portion of our visualization algorithm. However, there are parts of the preprocessing stage that we can examine. We use GAPS to create LODs and HLODs and we have already analyzed this algorithm in Section 3.6. The only other steps in the preprocess stage that could adversely affect performance are node association and partitioning.

We do add nodes to the scene graph whenever we perform node association or partitioning. If a node has n children, then in the worst case we could grow the height of the

scene graph by *n* during node association. However, this result assumes that the children nodes are spatially organized in a degenerate manner. For most situations where polygons are reasonably spatially balanced, we expect to increase the height of the scene graph by approximately *lg n* using node association. Partitioning could potentially create any number of leaf nodes in the partition scene graph depending on the user-specified partition distance. Suppose there are *p* leaf nodes in this partition scene graph. Since we group partitions by doubling the partition distance, the height of this partition scene graph will always be bounded by *lg p*.

4.6 Comparison

Our algorithm is different from traditional scene graph methods such as [Rohlf and Helman 94] because of its inclusion of HLODs in each node of the scene graph. HLODs allow our algorithm to group polygons of multiple nodes for purposes of merging them together during simplification. Often, this combination of polygonal geometry results in better drastic approximations as shown in Figure 4.26.

[Luebke and Erikson 97], [Hoppe 97], and [Xia et al. 97] present research on view-dependent simplification schemes. In Section 4.1.1, we go into detail on why we chose an algorithm based on LODs and HLODs. With our approach, we are able to render using display lists and approximate view-dependent rendering schemes using partitioning.

[Funkhouser and Séquin 93], [Maciel and Shirley 95], and [Aliaga and Lastra 99] present target frame-rate systems. [Funkhouser and Séquin 93] does not use any hierarchical representations of objects and thus cannot guarantee the frame rate if the time it takes to consider each object in the scene is more than the time allowed for a single frame. This algorithm uses strictly predictive methods to choose which LODs to render. Sometimes run-time performance cannot be predicted in advance. [Maciel and Shirley 95] extends the work presented in [Funkhouser and Séquin 93] by using view-dependent images as hierarchical representations of multiple nodes in the scene graph. The way in which the algorithm traverses the scene graph touches every object in the environment in order to determine the benefit of view-dependent images. Our algorithm uses LODs augmented with HLODs. It

traverses the scene graph to determine which polygons to render and can terminate this search at any time since it always has a complete representation of the environment. Our target frame-rate mode uses a combination of predictive and reactive methods that do not rely on performance specifications of a machine.

4.7 Summary

We have presented an algorithm for accelerating the rendering of large static polygonal environments. It augments the traditional level of detail scene graph with hierarchical levels of detail. HLODs allow polygons of multiple nodes in the scene graph to be grouped together for the purposes of simplification. For groups of objects that are close together, HLODs are usually higher quality drastic approximations than the union of the individual LODs of these objects. When an HLOD is rendered, we cull away portions of the scene graph. To make HLOD creation and view frustum culling more efficient, we use a process called node association which groups nodes in the scene graph by spatial proximity. Spatially large objects are partitioned and then these partitions are grouped hierarchically to approximate view-dependent rendering techniques and to maximize the benefit of view-frustum culling. Our algorithm is capable of rendering using a pixel-error mode or a target frame-rate mode. Both partitioning and this target frame-rate mode depend on the use of HLODs. Since LODs and HLODs are static, we efficiently render them using display lists. Finally, our system has accelerated the rendering of a wide variety of large polygonal environments with little or no loss in image quality.

5 SIMPLIFICATION OF DYNAMIC POLYGONAL ENVIRONMENTS

In the previous chapter, we presented an algorithm to accelerate the rendering of static polygonal environments represented by scene graphs by using a combination of LODs and HLODs. It was shown that by using HLODs to merge polygons from multiple nodes in the scene graph, we could produce higher fidelity and more drastic approximations for particular types of static environments. A user often needs to insert, delete, or move objects in the scene, such as in spatial design applications. Since the polygonal geometry in each node does not change due to this movement, the LODs of each node never change. However, when objects move, their relative positions change and thus any HLODs that represent these objects must be updated. In this chapter, we extend our algorithm to handle rigid-body dynamic environments. This extension consists of two operations. When objects move, we update the error of affected HLODs, regroup nodes based on spatial proximity, and update the bounding volume hierarchy of the scene. After nodes are regrouped, we recalculate their HLODs in parallel with the rendering process.

The rest of this chapter is organized in the following manner. We provide an overview of our approach in Section 5.1. Section 5.2 discusses the technical details of our algorithm. Implementation details are presented in Section 5.3 and our performance results are shown in Section 5.4. Analysis of the running time of the algorithm is shown in Section 5.5. We compare our dynamic environment algorithm based on HLODs to other rendering algorithms in Section 5.6 and we conclude the chapter in Section 5.7.

5.1 Overview

A dynamic environment is one where objects or even individual polygons change positions or orientations between frames. If individual polygons within an object change shape, then this movement is considered to be a *deformation* of the object. Deformation can

be a computationally expensive operation since it involves changing every polygon of an object that is affected by the movement. *Rigid-body* motion is where objects move, but the polygons in the objects do not change their relative positions. Scene graph structures for polygonal environments are well suited for rigid-body motion. Objects change position or orientation by updating transformations at arcs in the scene graph. Since we use scene graphs to represent polygonal environments, it is natural for us to support rigid-body motion. The current algorithm does not handle deformable objects.

Since we assume rigid-body motion, the polygonal geometry of a node in the scene graph never changes. The node's LODs remain the same throughout the visualization of the scene. Only the relative placement of objects in the scene changes due to movement. The creation of HLODs depends on the relative locations of objects in the environment. Therefore, as objects move, previously created HLODs may no longer be accurate.

Our approach updates HLODs in response to dynamic movement within the environment. It first updates error bounds of HLODs affected by the movement. It then performs an incremental process called *node re-association* where it groups together nodes in the scene graph according to their proximity. Finally, it updates the scene graph's bounding volume hierarchy. Once the scene graph has been modified, we insert nodes whose HLODs must be recalculated into a queue. Simplification processes running asynchronously and in parallel with the rendering process remove nodes from the queue to create HLODs using GAPS. If the motion of the objects is small, then it may be acceptable for the rendering process to use previously created HLODs while waiting for the simplification process to finish.

Rigid body dynamic environments can be classified in terms of their degree of motion. Dynamic environments are a crucial part of entertainment software where almost every entity in the game is capable of some sort of movement. For example, in the first-person shooter genre, players are expected to move quickly, evade creatures that are chasing them, and fire weapons to defeat them [Erikson 99]. Almost everything in this scenario is a moving object, except for perhaps the city or maze in which the action occurs. There are environments where there is continuous movement, but only in localized sections or in few objects of the scene.

An example of this type of environment is a model of a miniature train set. The terrain, tunnels, trees, and most of the track is stationary while the train itself is one of a few moving objects. Finally, there are scenes where the majority of the time no objects move at all. Occasionally, a single object, or group of objects, will move to a new location. This type of environment is characterized by bursts of activity. An example of such an environment is a design and review scenario. A user can move objects around by selecting them and placing them in new locations. When the movement is over, usually the user will take some time to make sure everything looks okay before continuing.

We will show that our current implementation for dynamic environments is most effective for this last scenario. If too many objects are moving in the scene, our polygonal simplification process will not be able to update HLODs quickly enough. In this chapter, we describe a simple model for defining the degree of motion in an environment. This model is used to estimate the computing power it would take to recalculate the HLODs of a scene in a specified amount of time. The results of this model verify that the current implementation of our technique works best on environments with limited dynamic movement.

5.2 Dynamically Updating HLODs

For our algorithm, the key problem for dynamic environments is updating HLODs that are affected by object movement. We use the following techniques to handle such scenes:

- When objects move, HLOD error bounds, the grouping of nodes, and the bounding volume hierarchy are all updated. We call the process of regrouping nodes according to their spatial proximity *node re-association*.
- Nodes in need of HLOD recalculation are inserted into a queue. Our algorithm spawns parallel and asynchronous simplification processes that extract nodes off this queue to recalculate their HLODs using GAPS.

5.2.1 Updating the Scene Graph Due to Object Movement

Objects in a scene move because of changing transformations at arcs in the scene graph. An object may be inserted or deleted from the scene graph, but we view these as different operations than pure movement. One can argue that by changing transformations at an arc, objects can be thought of as being deleted from their old position and inserted into their new position. However, in terms of efficiency, these operations are not equivalent and we handle them differently. We first discuss objects that move in the scene due to a change of transformations.

5.2.1.1 Modification of Transformations

Assuming a transformation at an arc has changed, our algorithm determines its effect on the accuracy of HLODs, re-associates nodes based on the new positions of objects, and updates the bounding volume hierarchy of the scene graph.

5.2.1.1.1 Updating Error Bounds of HLODs

Our algorithm compares the relative locations of the bounding spheres of the old object position and the new object position, and measures changes in orientation and scale. It produces a distance error in object space due to this movement and adds this error to the distance error associated with each HLOD of the node (see Section 4.2.1). By calculating the error using the bounding volumes of the moving objects, we conservatively estimate the distance error that this movement adds to the HLODs. This movement also affects HLODs of the parent nodes in the scene graph. Therefore, we propagate this error up the scene graph and add it to all of the affected HLODs.

Figure 5.1, Figure 5.2, and Figure 5.3 show an example of the effects of movement on the accuracy of a node's HLODs. In Figure 5.1, we show a simple scene graph. The House node has two children that are rooms and contains HLODs that represent the whole scene. The Game Room node has two children and it contains HLODs that represent this whole room. One of the Game Room's arcs points to a node with a moving Ball object. The other points to a stationary table. The bold arc contains a transformation that we use to change the

position, scale, and orientation of the Ball. When we change this transformation, it affects the accuracy of HLODs in the Game Room's node. Note that by changing this transformation, we change the location of the Ball in the Game Room's coordinate space.

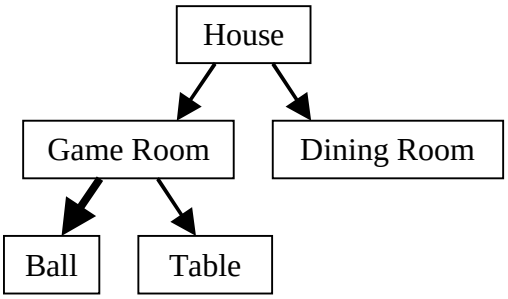


Figure 5.1: A simple scene graph.

In Figure 5.2, we show error changes in HLODs due to object movement. In *(a)*, circles denote bounding volumes of the Ball. The dotted circle represents the Ball's old position while the solid circle is its new position. The length of the arrows shows how much error an operation causes. The error caused by translation is the distance that the Ball moves. In *(b)*, the error caused by rotation is the distance a point moves along the surface of the circle. In *(c)*, the error caused by scaling is the distance a point on the surface of the circle moves. In *(d)*, if the Ball is scaled, rotated, and then translated, then the error is a sum of all three component errors. If the operations occur in a different order, the resulting error would change.

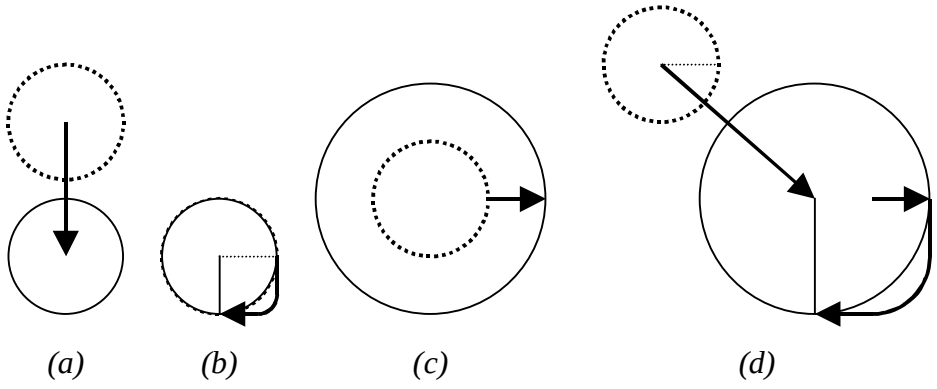


Figure 5.2: Error changes in the HLODs due to object movement. *(a)* Translation. *(b)* Rotation. *(c)* Scaling. *(d)* Scaling followed by rotation followed by translation.

In Figure 5.3, we show that the error due to movement not only affects the Game Room's HLODs, but it also affects any HLODs that represent the Ball's geometry. The dotted region denotes parts of the scene graph affected by the Ball's movement. Thus, after each of the HLODs in the Game Room is modified, the error is propagated up the scene graph. The dotted arrows show this error propagation. In this case, the movement also affects the House node's HLODs. Note that the distance error in the Game Room is transformed into the coordinate system of the House.

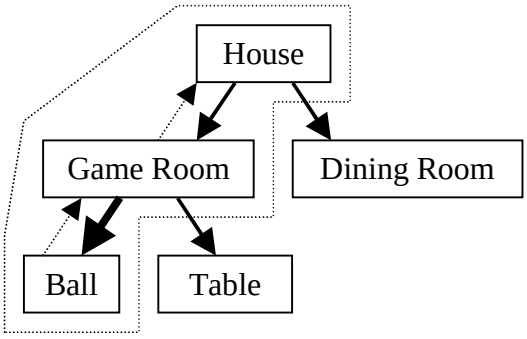


Figure 5.3: The error due to movement propagates up the scene graph.

Adding to the distance error of affected HLODs implies that they will be rendered further away from the viewer than in previous frames. These HLODs are still usable, but are of lower quality since some of the polygonal geometry that they represent has changed position.

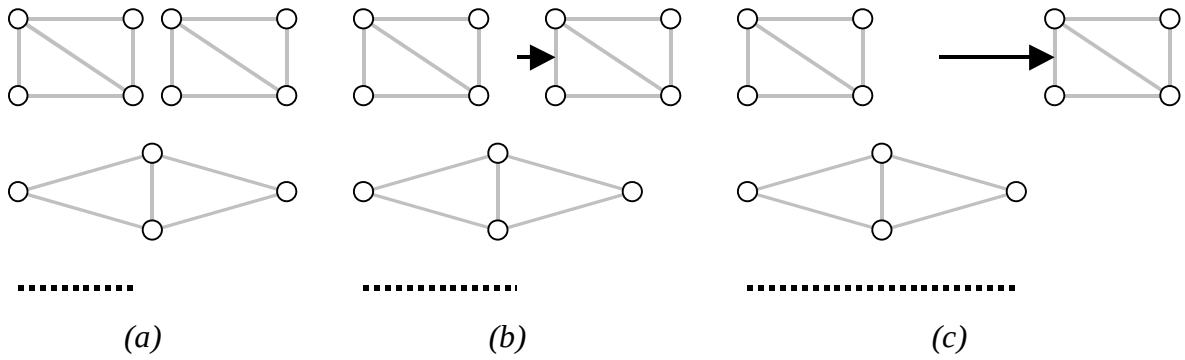


Figure 5.4: Even though an HLOD is inaccurate, it may be used to approximate groups of objects.

Figure 5.4 demonstrates how our approach reuses inaccurate HLODs. In *(a)*, each rectangle in the top row represents the polygonal geometry for an individual node in the scene

graph. The HLOD representing these top two rectangles is in the middle and consists of two merged triangles. The dotted line represents the distance error associated with this HLOD. In *(b)*, the right rectangle moves away from the left one. The distance error associated with the HLOD is increased by the distance the rectangle has moved. Note that the HLOD is still a fairly good approximation for these two rectangles. In *(c)*, the right rectangle moves even farther away from the left one. The HLOD's distance error is again increased by the distance the rectangle moved. Now the HLOD is a very poor approximation for the two rectangles. However, it is still possible to substitute this HLOD for the rectangles if the viewer is a great distance away.

5.2.1.1.2 Node Re-Association

As described in Section 4.2.2, we associate nodes when a parent node has multiple children nodes. By hierarchically grouping these children according to their proximity, we make HLOD creation and view-frustum culling more efficient. As objects move in the scene, the relative locations of nodes change. Thus, what was once an efficient association of nodes might no longer be as efficient. Our system updates the association scene graph by re-associating these nodes. The degree of movement affects how much work we have to perform on the scene graph. If a node moves a small distance, then it is sometimes possible to update the bounding volume hierarchy without changing the structure of the scene graph. If a node moves a larger distance, then the scene graph structure itself changes. Examples of this process are shown in Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, and Figure 5.9.

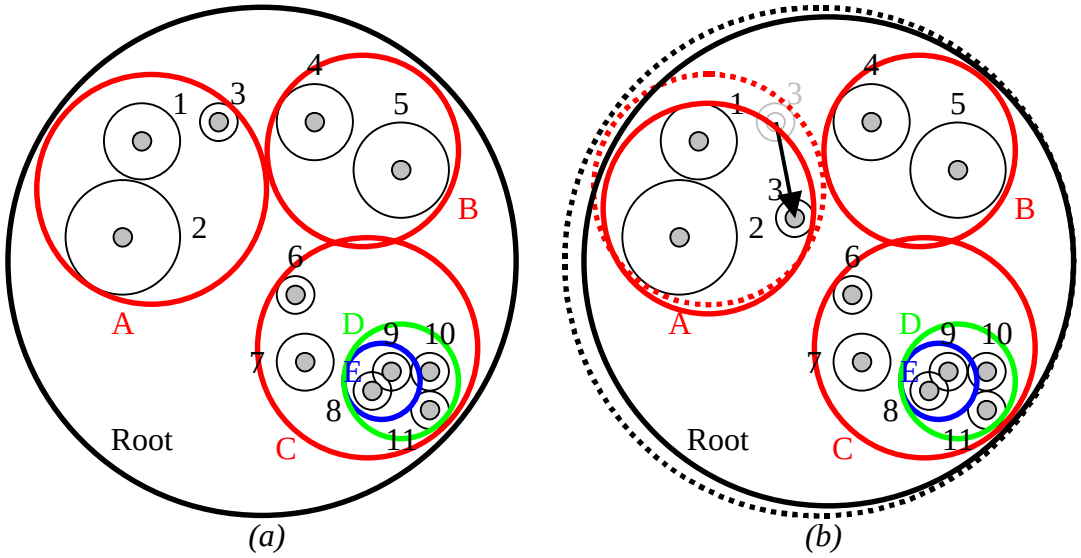


Figure 5.5: Example of movement that does not affect the association of nodes in a scene graph. *(a)* This scene is the same as in Figure 4.9. *(b)* Node 3 moves slightly. However, its entire movement is contained within the red bounding circle labeled A. The dotted red bounding circle is what A used to be. The solid red bounding circle is the new tighter fitting bounding circle for Nodes 1, 2, and 3. Similarly, we recursively recalculate bounding circles further up the scene graph. In this case, the Root node is a parent of A so we recalculate its bounding circle. The old circle is shown in dotted black and the new circle is shown in solid black.

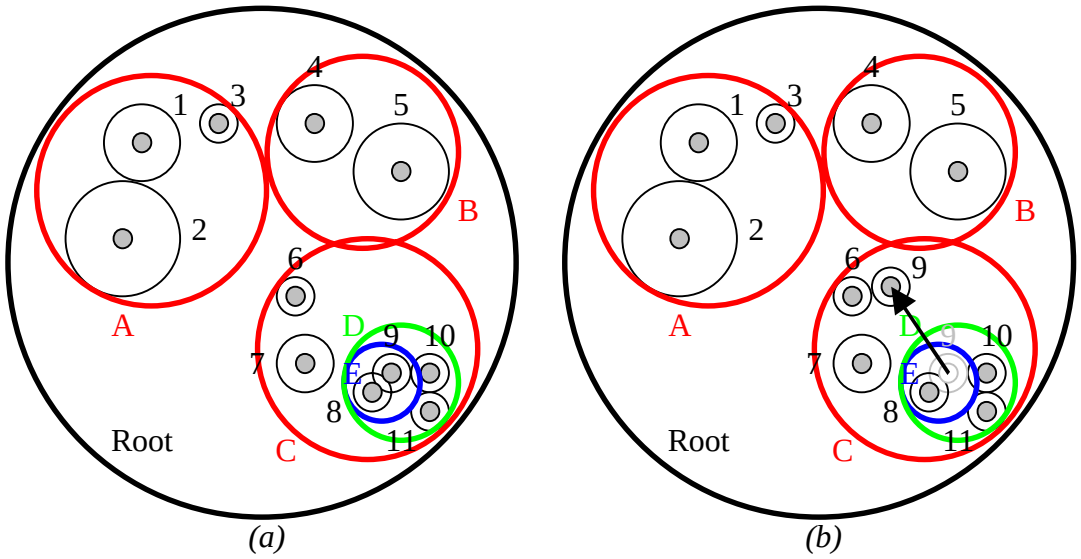


Figure 5.6: Example of movement that affects the association of nodes in a scene graph. *(a)* This scene is the same as in Figure 4.9. *(b)* Node 9 moves outside of its blue bounding circle.

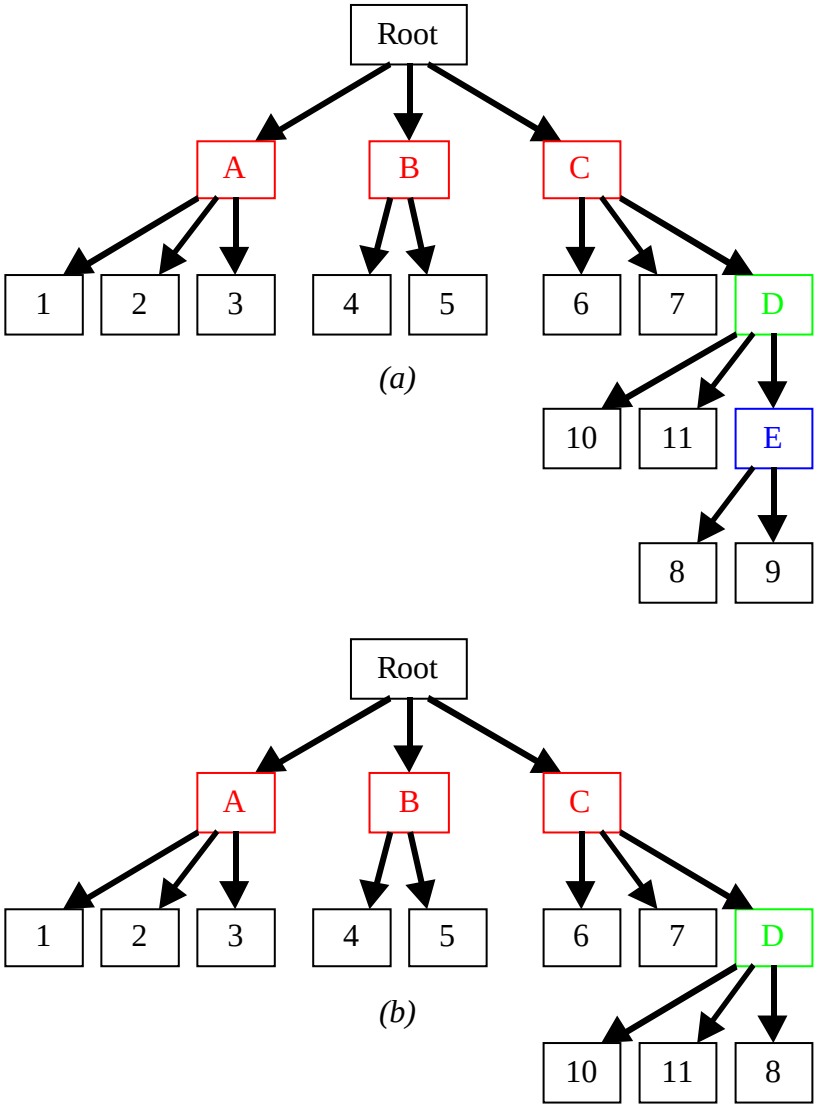


Figure 5.7: Continued from Figure 5.6. *(a)* The initial scene graph corresponding to the original positions of the objects. *(b)* Since node 9 has moved outside of its bounding circle, it is temporarily deleted from the scene graph. When we delete node 9, node E only has one child, namely node 8. Having only one child is inefficient in terms of HLOD creation and view-frustum culling. Node E was created by our association process and is not an original node in the scene graph. Therefore, we collapse node 8 upward to replace node E.

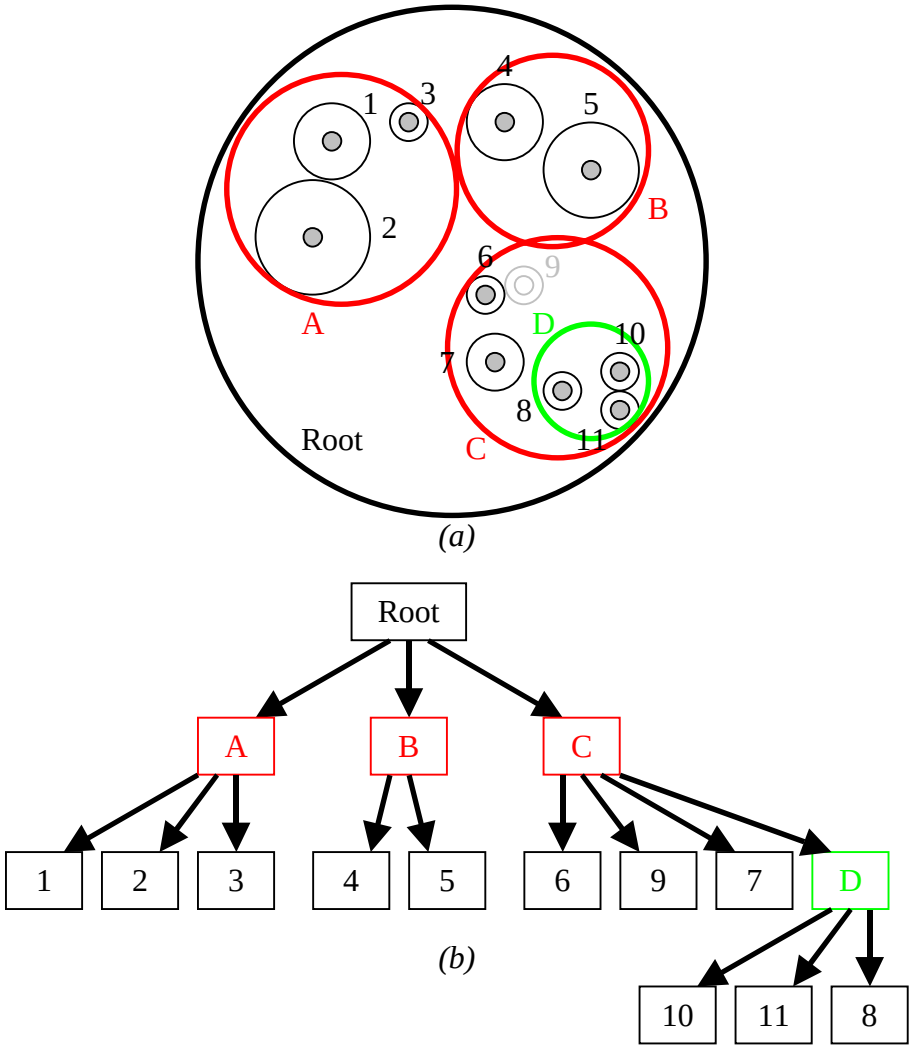


Figure 5.8: Continued from Figure 5.7. ***(a)*** **Here is the bounding volume hierarchy of the scene graph with node 9 deleted. A gray outline shows the real position of node 9. Next, we perform a search to determine where node 9 should be located in the scene graph. We first perform an upward search, starting from node 9's former parent. Node 9's parent used to be node E, but that was replaced by node 8. Therefore, we start searching upward at node 8. We continue going up the scene graph until we find a node whose bounding circle contains node 9's bounding circle. In this case, node 8's bounding circle does not contain node 9's. Node D's bounding circle also does not contain node 9's. Node C's bounding circle does contain node 9's. We next perform a downward search, starting from where we ended our upward search. We continue going down the scene graph until we find no children nodes whose bounding circles enclose node 9's bounding circle. In this case, no children of node C have bounding circles that enclose node 9's. At this point, we insert node 9 as a child node of the node where we ended our downward search. Therefore, we make node 9 a child of node C.** ***(b)*** **The new scene graph after this operation.**

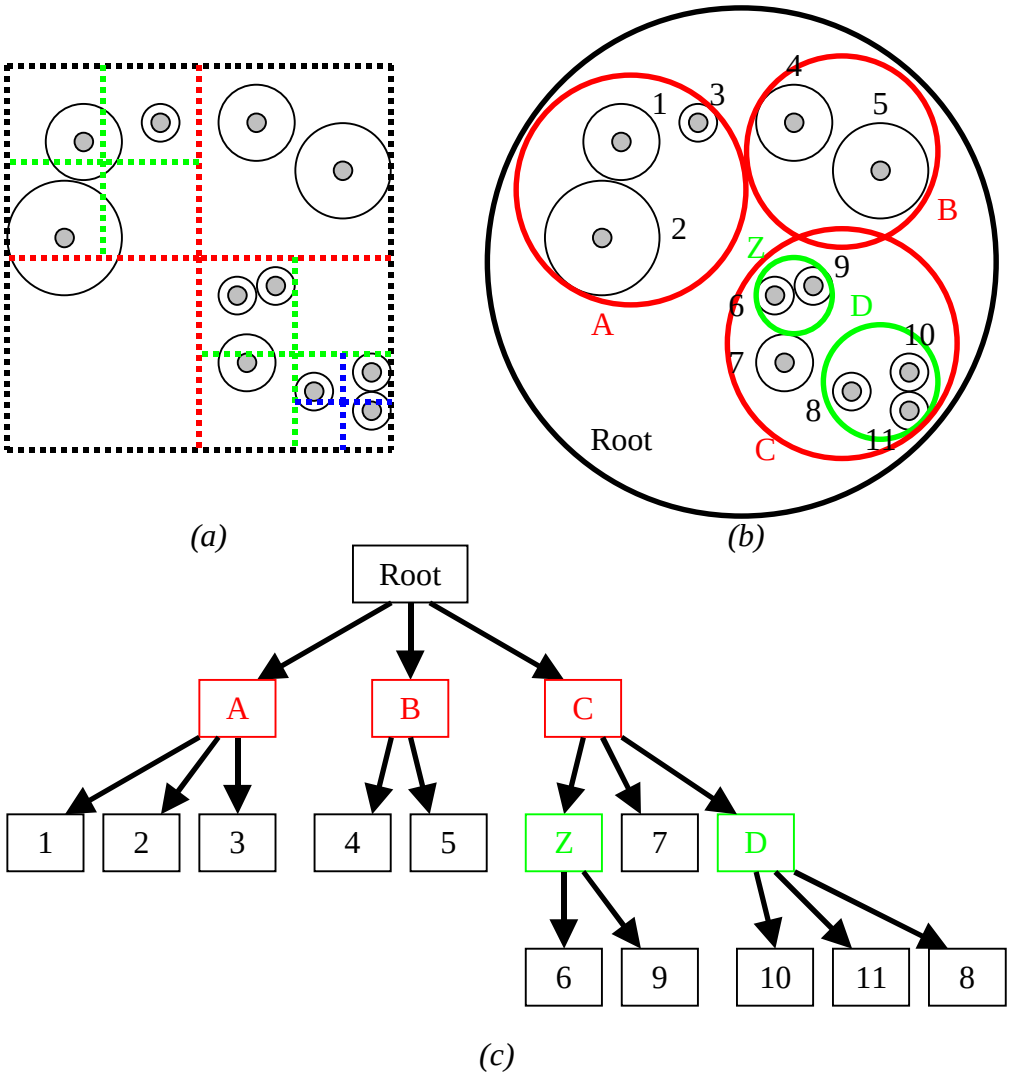


Figure 5.9: Continued from Figure 5.8. *(a)* We attempt to associate children nodes of the node where we terminated our downward search. In this case, we are able to associate nodes 6 and 9 using our quadtree subdivision. *(b)* The new bounding volume hierarchy for this scene, including a new node Z which encloses nodes 6 and 9. We calculate this new hierarchy in a bottom up fashion from the node C, the end of our downward search. *(c)* The new scene graph for this environment.

In Section 4.2.2, we describe how the creator of a scene graph can dictate how objects will be grouped for purposes of view-frustum culling and HLOD creation. Nodes that make up this scene graph are *original nodes* and should not be modified. During our upward and downward searches, we never traverse past an original node in the scene graph. Suppose that the scene shown in Figure 5.9 was a small portion of the entire environment and the Root node was an original node that had a parent. The upward traversal could not go farther than

the Root node because it is an original node in the scene graph. Not allowing these searches to penetrate original nodes in the model makes the algorithm stay true to the wishes of the designer of the scene graph. In other words, just like the original association of the scene graph, we isolate the re-association of the nodes to places that will not modify the original design of the environment. If there was no pre-existing scene graph for the environment, then our algorithm can freely modify the scene graph hierarchy.

Movement of objects causes the approximation quality of HLODs to decrease. However, when node re-association occurs, some HLODs become invalid altogether. For example, in the scene shown in Figure 5.7 and Figure 5.9, any HLODs of node D in the original scene graph represented all of its descendants, namely nodes 8, 9, E, 10, and 11. However, when node 9 moves, node D has only nodes 8, 10, and 11 as descendants. Therefore, its HLODs have no chance of representing the correct geometry anymore. In fact, the HLODs of every node visited by our upward or downward searches are invalidated except for the node where our upward search terminates. In Figure 5.8, nodes C and D were visited during our upward and downward search. However, node C is where we terminated our upward search. Therefore, the only set of HLODs that we invalidate is node D's HLODs. We add these invalidated HLODs into our simplification queue. These HLODs remain invalid until our parallel simplification process can recalculate them.

5.2.1.1.3 Updating the Bounding Volume Hierarchy

After re-associating each moving node, we recalculate the bounding volume hierarchy of the scene graph. We start from the node where we terminated our downward search during the re-association process. We recursively update the bounding volume hierarchy upward from this node. It is possible that movement by a node in the scene graph causes bounding volumes higher up in the hierarchy to shift. Modifying a bounding volume of an original node in the scene graph is considered to be a movement of the node. Thus, further re-association of nodes higher up in the scene graph might be necessary.

5.2.1.2 Insertion and Deletion

Insertion and deletion of nodes in the scene graph are abrupt operations and cannot be handled as elegantly as changing transformations at arcs. If a node is inserted into the original scene graph, then we invalidate all HLODs further up the scene graph. These HLODs were created to represent geometry that did not include this new object and thus are now almost completely wrong. Furthermore, the insertion of this new node might change associations that were created earlier. Therefore, we re-associate the scene graph starting at the parent of this new node. We skip the upward search in this case and perform the downward search to determine where the new node will be inserted. For deletions, we also invalidate HLODs further up the scene graph, starting from the parent of the deleted node. If the parent node was created by association and now only has one child, we compress the scene graph so that the child becomes the parent. An example of this type of collapse is shown in Figure 5.7 *(b)*, where node 8 replaces node E. Both insertion and deletion can cause the bounding volume hierarchy to change and thus could cause additional re-associations further up in the scene graph. Nodes with invalid HLODs are inserted into the simplification queue in order to recompute the HLODs.

5.2.2 Asynchronous Simplification

Simplifying polygonal geometry is much slower than rendering the same polygonal geometry on current platforms. For example, we render one frame of the original polygonal geometry of the Power Plant model of Section 4.4 in 20 seconds. However, simplification preprocessing of this model takes more than 4 hours as shown in Table 4.1. For interactive visualization, it is not feasible to have one process both rendering the scene and updating HLODs in the scene graph. If we used only one process, the updating of HLODs would dominate the running time of our algorithm. Thus, in addition to a rendering process, we use one or more simplification processes.

The movement of nodes in the scene causes many HLODs to be inaccurate or even invalid. These nodes are inserted into a simplification queue. We run multiple simplification processes on multiple CPUs to recalculate HLODs. The job of a simplification process is

simply to dequeue a node and create a set of HLODs for that node. We use GAPS at run-time to update these HLODs.

Each simplification process ideally resides on a different CPU in the machine. The process polls the simplification queue. If there are no nodes to simplify, it does nothing. If there is at least one node on the queue, it dequeues the node and starts simplifying the node's associated polygons. Once it finishes creating a set of HLODs for the node, it copies these HLODs into the scene graph for future use. There can be one or more simplification processes running independently of the rendering process. They perform their tasks asynchronously, while the rendering process continues to render the scene. The rendering process has access to an updated set of HLODs only when a simplification process has finished creating them. We describe how we handle several issues dealing with accessing and modifying the scene graph with multiple processors in Section 5.3.

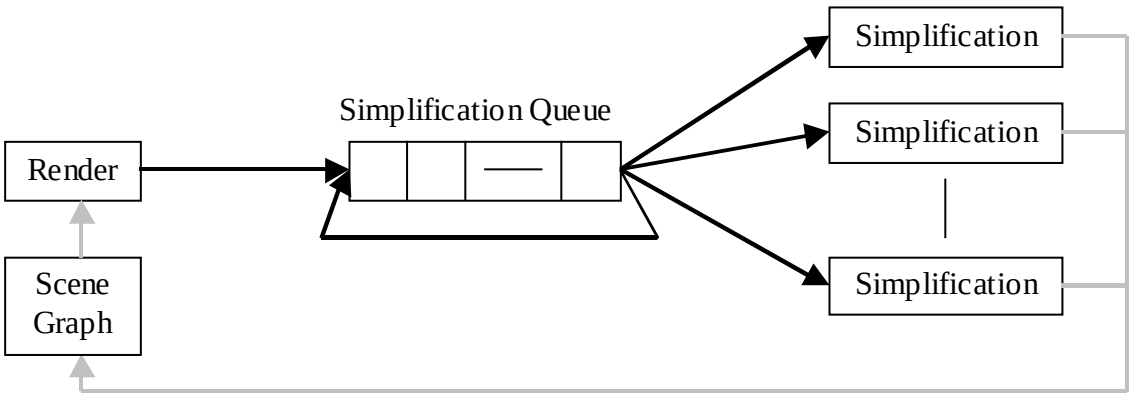


Figure 5.10: Diagram showing how the different processes in our algorithm interact.

The interaction between the rendering process and simplification processes is shown in Figure 5.10. The Render process determines which nodes in the scene graph need their HLODs updated. It inserts these nodes into the Simplification Queue. The multiple Simplification processes pull nodes off the Simplification Queue and create HLODs for the node's polygons. The black arrows show the data flow of these nodes in need of HLOD recomputation. The lowest black arrow emanating from the Simplification Queue shows that sometimes nodes are dequeued and then immediately re-inserted in the queue. This re-insertion happens when the node is a parent of a child node in need of HLOD recalculation.

Since the polygonal geometry of the child node's HLODs affects the HLODs of the parent node, the child node must be updated before the parent. Any newly created HLODs are re-inserted in the scene graph, ready to be used by the Render process. The gray arrows show this re-insertion process.

Since there is a speed discrepancy between rendering and simplification, we cannot expect to update the HLODs interactively if there is a great deal of movement in the scene. However, our system works well for design and review scenarios where a user occasionally interacts with objects. Since movement in the scene is infrequent, the simplification processes are usually able to update HLODs in a few seconds (see Section 5.4). Often, a user will only manipulate objects that are near their current viewing position. Since we only recalculate HLODs and not LODs, and HLODs are used for coarse approximations, the user must move some distance away from these nearby objects before the new HLODs will be needed for rendering. By the time the user moves to a new viewpoint and the HLODs are needed, hopefully they will have been updated. We perform a simple analysis of how much movement our algorithm can currently handle in Section 5.5.

5.3 Implementation

We have implemented our dynamic environment visualization algorithm using C++, GLUT, and OpenGL. The code is portable across PC and SGI platforms. Our algorithm has allowed us to interactively visualize complex models as we move some of their parts around.

5.3.1 Generality

The simplification processes use GAPS to recalculate HLODs for the scene graph. As demonstrated in Chapter 3, GAPS is very general. In practice, we have used this system on a variety of complex and degenerate CAD models.

5.3.2 Node Status

We store one integer, used as a bit flag, in each node in the scene graph to represent its status. A node can be dirty, in which case its HLODs are inaccurate or invalid and it is currently in the simplification queue. A node can be an original node, meaning that it was a node in the original scene graph. As described in Section 4.2.2, our algorithm has more control over nodes created by association, rather than the original nodes. A node can be marked as deleted, meaning that either deletion or re-association has removed the node from the scene graph (see Section 5.2.1). In some situations, we cannot immediately delete this node from memory. For example, suppose the node has been inserted into the simplification queue, but is later deleted from the scene graph. The simplification queue still refers to the node so we wait until this reference is dequeued before deleting the node from memory. Finally, a node's HLODs can be marked as deleted in the node's bit flag. This flag is useful when an HLOD of a node is determined to be invalid while a simplification process is already trying to recalculate its HLODs from a previous frame. When the simplification process is finished, we ignore its output and re-insert the node into the simplification queue because the node's HLOD is marked as deleted.

5.3.3 Asynchronous Simplification

Both the rendering process and simplification processes need to access and modify the scene graph during the execution of our algorithm. We prevent multiple processes from corrupting the scene graph data by using a combination of three semaphores. One semaphore protects the scene graph, another protects HLODs at nodes, and the final semaphore controls access to the simplification queue. To clarify the following explanations, we label these semaphores *SG*, *H*, and *SQ*, respectively.

We use *SG* to control access to the scene graph. Whenever the rendering process is about to render a frame, we lock *SG*, render the frame, and then unlock *SG*. Whenever our system re-associates nodes, it first locks *SG*, changes its structure, and then unlocks *SG*. When a simplification process finishes creating a set of HLODs, we lock *SG*, insert the newly created HLODs into the correct node, and then unlock *SG*. We also must lock and unlock *SG*

when checking to see if a node on the simplification queue has been marked as deleted or its HLODs have been marked as deleted. Basically, anytime we modify or query the structure of the scene graph, *SG* must be locked. Having *SG* locked prevents other processes from modifying the scene graph data simultaneously.

We use *H* to control access to HLODs of nodes. In order for simplification processes to create HLODs, they must be able to pool the polygons of the node being recalculated, plus the polygons of the children nodes' HLODs. Since *SG* deals with the structure of the scene graph and not the polygonal geometry within nodes, the simplification process can read, but not write, the polygons of LODs and HLODs when *SG* is locked. Therefore, this grouping of polygons to create HLODs can execute independently of actions such as rendering. When pooling polygonal geometry, a simplification process first locks *H*. After the polygons are grouped, *H* is unlocked. During the simplification of this polygonal geometry, no semaphores need to be locked. When the HLODs have been created, *H* and *SG* are both locked, the newly created HLODs are inserted into the scene graph, and then both *H* and *SG* are unlocked.

The final semaphore, *SQ*, is used to restrict access to the simplification queue. When the rendering process determines that certain nodes are in need of HLOD recomputation, it can lock *SQ*, insert a new node in the simplification queue, and then unlock *SQ*. All simplification processes attempt to lock *SQ* in order to remove nodes from the queue. If the process gains access to the queue, but it is empty, then the process unlocks *SQ* for further queries. If the queue is not empty, it dequeues the head node and then unlocks *SQ*. The process then has access to a node in need of HLOD recalculation.

All of the processes rely on being able to access the scene graph. However, we do not want to slow down the rendering process because *SG* or *H* is locked by a simplification process. Therefore, it is very important that simplification processes lock both *SG* and *H* for only brief periods of time. Once a simplification process has finished creating HLODs for a node, it must reinsert this new polygonal geometry into the scene graph. To do so, it must lock *SG* and *H*. If we perform a copy of the HLODs into the scene graph, it could take a significant amount of time. Instead, a node contains a pointer to its HLODs. We simply

assign a new pointer to change the HLODs of the node and then unlock both *SG* and *H*. Thus, our system only executes one pointer assignment between locking and unlocking *SG* and *H*. After unlocking *SG* and *H*, we free the memory of the HLODs pointed to by the old pointer.

5.3.4 Targeting a Frame Rate

When displaying dynamic environments, our algorithm can still target a frame rate as long as there are valid HLODs at each node in the scene graph. As objects move, some HLODs become inaccurate. Our system can still use these inaccurate HLODs to target a frame rate. However, large movement causes some HLODs to become invalid, meaning that they cannot possibly represent the true polygonal geometry of the scene (see Section 5.2.1). In these cases, we cannot guarantee that our system will be able to target a frame rate assuming that no polygons is not a valid representation of an LOD or HLOD. In other words, a traversal of the scene graph, as outlined in Section 4.2.4, will encounter invalid HLODs that cannot be used. Our algorithm must then search further down the scene graph until a complete set of LODs and valid HLODs for the scene is found. Since this traversal cannot stop at an invalid HLOD, we cannot bound the time it will take to finish, and thus we cannot target a frame rate. If no polygon representations are acceptable for LODs and HLODs, then we can still target a frame rate.

5.3.5 Main Loop

Similar to the static algorithm, the main loop of the rendering process for dynamic scenes is as follows:

- Handles input to change the viewer position.
- If the algorithm is in pixel-error mode, then it performs a depth-first search of the scene graph, locally terminating when an HLOD or LOD is reached that has an associated screen-space error less than the allowable pixel-error.

- If the algorithm is in target frame-rate mode, it traverses the scene graph to determine which polygons to render, terminating when the polygon budget is reached and all HLODs are valid.
- If an object moves, we update error bounds of HLODs, perform node re-association, and update the bounding volume hierarchy. Nodes in need of HLOD recalculation are inserted into the simplification queue.

The main loop of a simplification process is as follows:

- Extract a node off the simplification queue.
- Create new HLODs for this node.
- Insert the newly created HLODs back into the scene graph.

5.4 Results

This section consists of four parts. We first measure the performance of our asynchronous simplification technique on a simple scene as we vary the number of simplification processes. We also measure how long it takes to update the scene graph when objects move in this scene. We then discuss the amount of memory our algorithm uses. Finally, we show visual results of our implementation on several real world CAD environments. These environments are the same ones that were shown in Chapter 4.

5.4.1 Asynchronous Simplification

We tested the performance of multiple asynchronous simplification processes to determine if there are levels of movement in scenes for which our current implementation is not effective. To perform the tests, we created a simple scene consisting of a root node that has multiple instances of a cube object as its children. We tested multiple scenes using multiple simplification process configurations. The scenes we tested consisted of 2^3, 3^3, 4^3, 5^3, 6^3, 7^3, 8^3, 9^3, 10^3, and 11^3 cubes. These cubes were arranged in a grid and then randomly moved a distance less than the side of the whole grid from their original positions (see Figure 5.11 and Figure 5.12). After the movement was completed, the simplification processes

recomputed HLODs for the scene. The configurations used were 1, 2, 4, 8, 16, and 31 simplification processes. We ran these tests on an SGI Reality Monster with 32, 300 MHz R12000 processors and 16GB of main memory. Since we reserve one process for rendering, we can have 31 simplification processes each running on its own CPU on this machine.

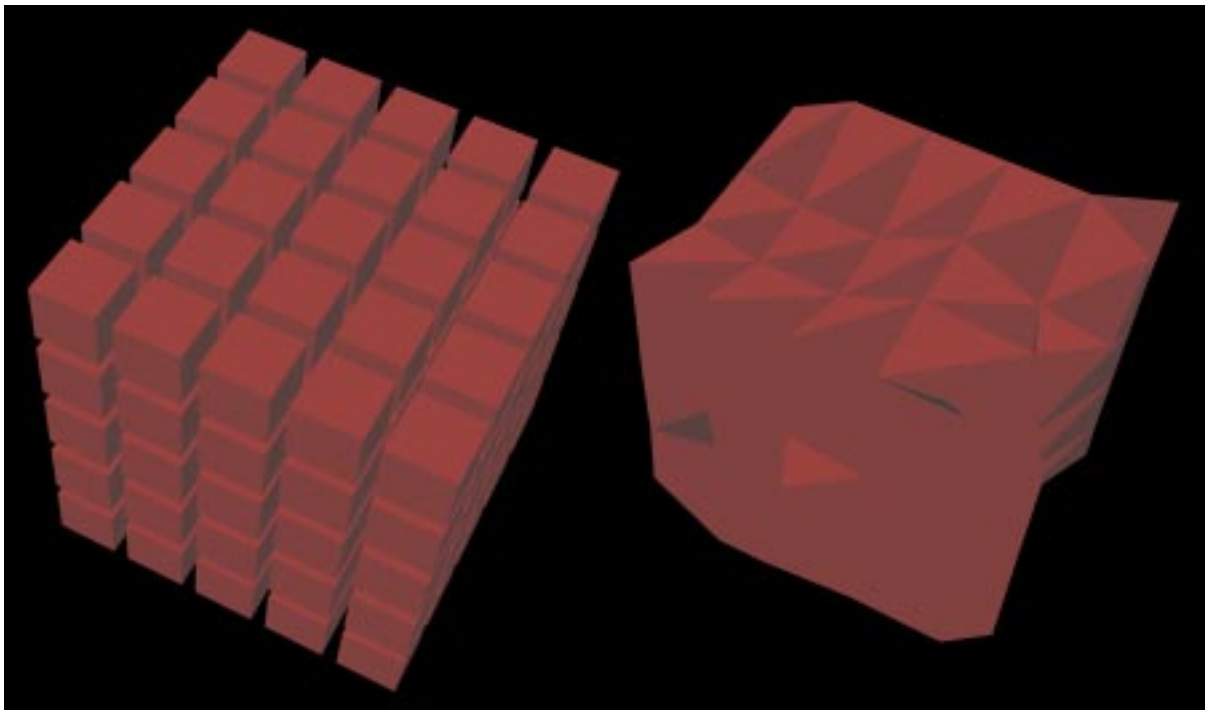


Figure 5.11: On the left we have a 5x5x5 grid of cubes consisting of 1,500 polygons. On the right is an HLOD of these cubes consisting of 750 polygons.

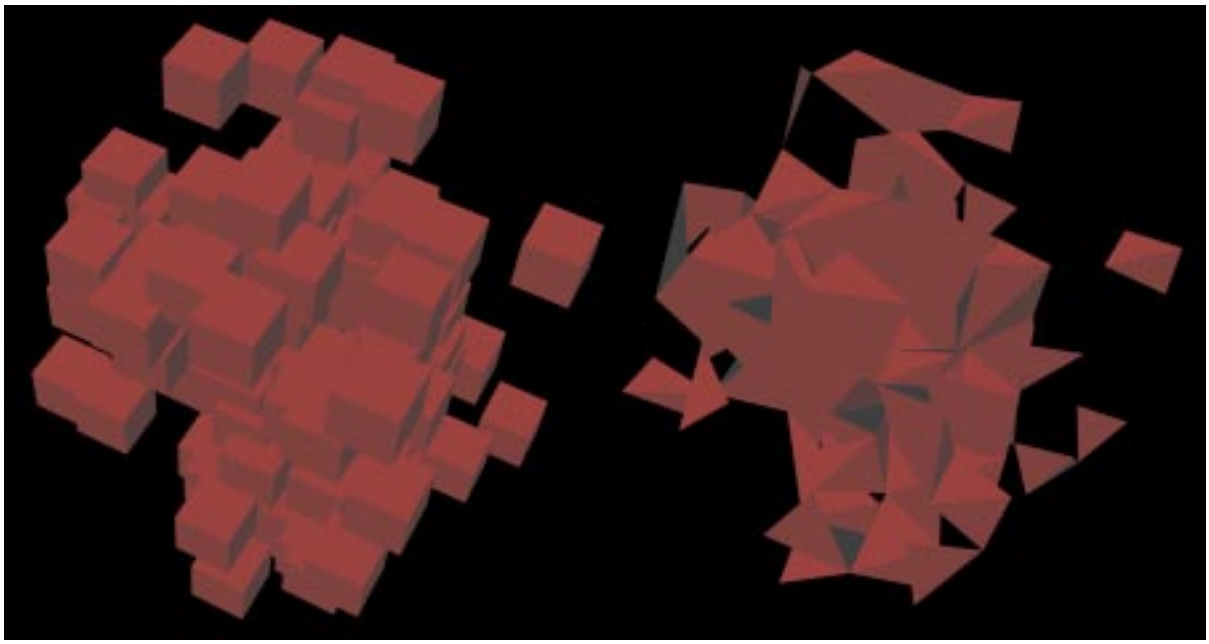


Figure 5.12: On the left, each of the cubes from Figure 5.11 has moved to a new location. As before, there are 125 cubes consisting of 1,500 polygons. On the right is an HLOD consisting of 692 polygons that was recomputed for the cubes after they moved.

The timing results of our tests are shown in Table 5.1. Graphs of this data are shown in Figure 5.13 and Figure 5.14. Figure 5.13 shows that adding simplification processes incurs some overhead cost and in cases where the scene is not dynamically complex, slows down the system. This overhead cost is mainly due to contention for data in the scene graph. In other words, adding processes causes more blocking to occur when locking and unlocking the three semaphores described in Section 5.3.3. We briefly discuss approaches to alleviate this contention in Chapter 6.

Figure 5.14 demonstrates that as the scenes consist of more moving objects, the benefit of using more simplification processes becomes apparent. Theoretically, we should see a linear speedup as we add more simplification processes. However, in the test cases we have observed, we achieve a sub-linear speedup. We conjecture that these results are due to overhead costs such as contention for data in the scene graph, plus the relatively small size of the scenes being tested. As the scene grows larger, we would expect the efficiency of multiple simplification processes to increase.

These results show that using multiple simplification processes is only beneficial when we have several hundred moving objects in the scene. However, even with 31 simplification processes, the recomputation of HLODs of these scenes does not occur in real-time. For a scene of moderate dynamic complexity, such as the environment with 512 cubes, it takes more than half a minute to update the affected HLODs. And we assume that the cubes move for only one frame. If the cubes moved constantly, clearly the simplification processes would never catch up. This performance suggests that our dynamic system is best used on environments with limited dynamic complexity such as design and review scenarios. Since a large number of simplification processes seem to slow the recalculation of HLODs in scenes with limited dynamic movement, it makes sense to use only a few of them for these environments.

# Cubes→	**2^3=**	**3^3=**	**4^3=**	**5^3=**	**6^3=**	**7^3=**	**8^3=**	**9^3=**	**10^3=**	**11^3=**
# Procs.↓	**8**	**27**	**64**	**125**	**216**	**343**	**512**	**729**	**1,000**	**1,331**
1	0.1	0.4	1.1	1.9	3.8	21.4	44.3	125.1	215.7	313.1
2	0.1	0.4	1.2	1.7	3.3	19.4	46.9	92.0	155.2	226.4
4	0.1	0.5	1.2	1.9	3.4	15.7	34.5	69.3	110.6	155.0
8	0.1	0.6	1.5	2.3	4.0	13.4	28.7	49.7	79.9	116.2
16	0.2	0.7	1.9	2.6	4.5	14.2	28.8	45.8	61.7	89.2
31	0.2	0.8	2.2	3.3	5.5	15.5	32.1	47.0	61.9	79.0

Table 5.1: Performance results in seconds of multiple simplification processes on scenes of varying dynamic complexity. Adding simplification processes causes the recalculation of HLODs to be slower on simple scenes. This behavior is the result of overhead incurred by adding more processes. As the scenes grow larger, using more simplification processes is justified. The only time 31 simplification processes accelerate the recomputation of HLODs, as compared to 16 processes, is when there are 1,331 cubes.

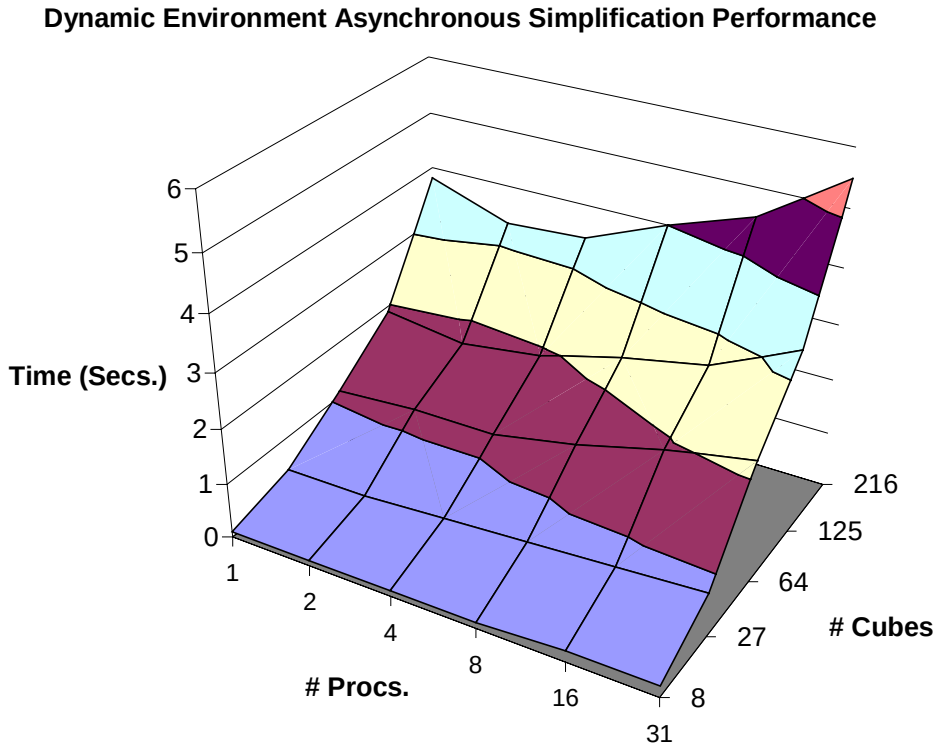


Figure 5.13: This graph shows the time it takes to recalculate HLODs of a scene consisting of a specific number of cubes utilizing a specific number of simplification processes. It shows simple scenes, with 216 cubes or less. Note that adding processes actually slows down the performance of the system due to contention overhead.

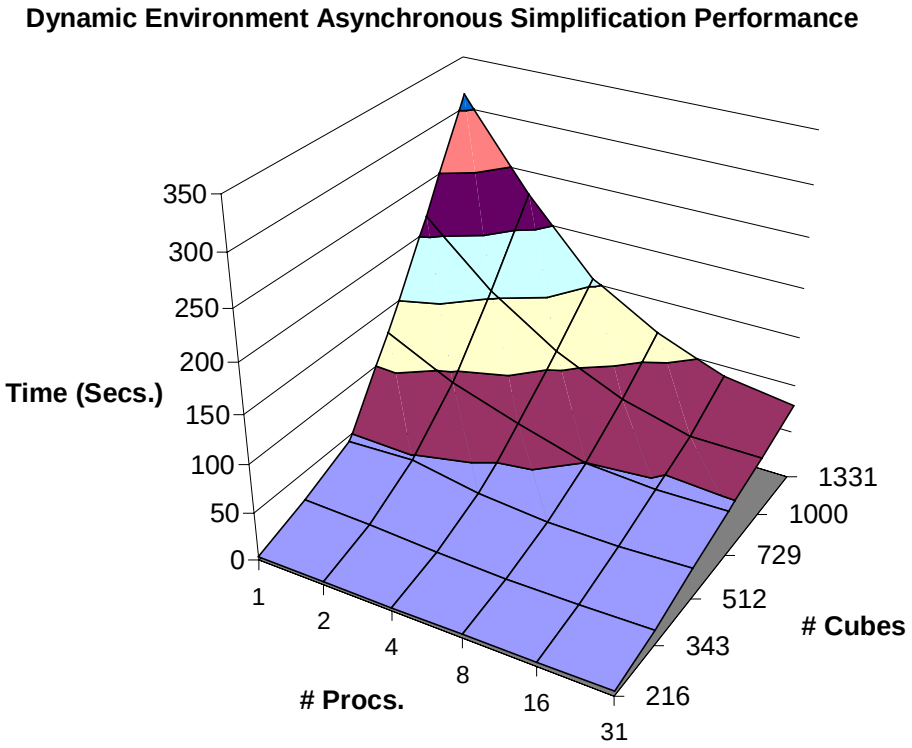


Figure 5.14: This graph shows the time it takes to recalculate HLODs of a scene consisting of a specific number of cubes utilizing a specific number of simplification processes. It shows complex scenes ranging from 216 to 1,331 cubes. Adding processes significantly speeds up the recalculation process as the scenes grow in complexity. For small scenes, adding processes may reduce the performance of the system (see Figure 5.13).

5.4.2 Updating the Scene Graph Due to Object Movement

Before recalculating HLODs when objects move, we update the scene graph as described in Section 5.2.1 using one CPU. During our test runs with the cube scenes in Section 5.4.1, we also recorded how much time updating the scene graph took after the cubes moved. The results are shown in Table 5.2. Even though we incrementally refine the scene graph, each refinement involves a fair amount of work. Error bounds on HLODs that represent the moving object are increased, the scene graph structure is modified using an octree spatial partitioning, and the bounding volume hierarchy is updated. Table 5.2 shows that for scenes with hundreds of moving objects, updating the scene graph cannot be performed in real-time. This limitation suggests that our algorithm is currently most useful for

limited dynamic environments such as design and review scenarios. We discuss possible techniques to accelerate updating the scene graph in Chapter 6.

2^3= 8	3^3= 27	4^3= 64	5^3= 125	6^3= 216	7^3= 343	8^3= 512	9^3= 729	10^3= 1,000	11^3= 1,331
0.002	0.005	0.053	0.114	0.235	2.967	6.045	8.291	11.998	15.248

Table 5.2: The amount of time that our current implementation takes to update the scene graph for scenes of varying dynamic complexity. The top row is the number of cubes in the scene and the bottom row is the time in seconds. Our system updates the scene graph at interactive frame rates only for scenes with less than a hundred moving objects.

5.4.3 Memory Usage

As described in Section 4.3.2, we create a series of levels of detail or hierarchical levels of detail such that LODs consist of half the number of polygons of the previous LOD. By doing so, we limit the memory usage due to the polygons of the scene to double that of the original polygonal geometry. However, with dynamic environments, we pool polygonal geometry from different nodes to create HLODs at run-time. The polygonal geometry we need to access is the coarsest HLOD of each node and it is not possible to access this data from an OpenGL display list. Therefore, if we use display lists to render LODs and HLODs, then we must store a separate copy of this polygonal geometry. We also store simplification information with the polygonal geometry, such as error quadrics at vertices. Note that we do not store extra copies of static LODs, since they never change. In practice, dynamic scenes require roughly six times the memory of the original polygonal geometry. Table 5.3 shows the memory increase going from static to dynamic environments for the CAD models shown in Chapter 4. Our system uses a large amount of memory for dynamic environments.

Scene	Objects	Triangles	Memory Increase
Bronco	466	74,308	3.25
Cassini	127	349,281	2.92
Torpedo Room	356	883,537	2.99
Power Plant	1,179	12,731,154	2.64

Table 5.3: Memory increase going from static to dynamic environments. On average, the memory increase is roughly three times. Since static environments take up double the memory of the original polygons, dynamic environments take up roughly six times the memory of the original polygonal geometry.

5.4.4 HLOD Recalculation Visual Results

We have used our dynamic scene algorithm to visualize the four CAD models shown in Chapter 4. Section 5.4.1 showed our current implementation to be most useful in scenes with limited dynamic movement. This type of movement occurs in design and review scenarios. To simulate a design and review scenario, we allow the user to select and move objects in view. For each scene, we changed the locations of a few of the objects. We visually compare the difference that this movement caused in the HLODs of the scene graph as well as how long it took for our algorithm to re-associate nodes and generate the new HLODs. For each test run, we used 4 simplification processes. Figure 5.15 through Figure 5.22 show HLODs being updated after object movement in these scenes. The execution speed of HLOD recomputation for these examples is shown in Table 5.4.

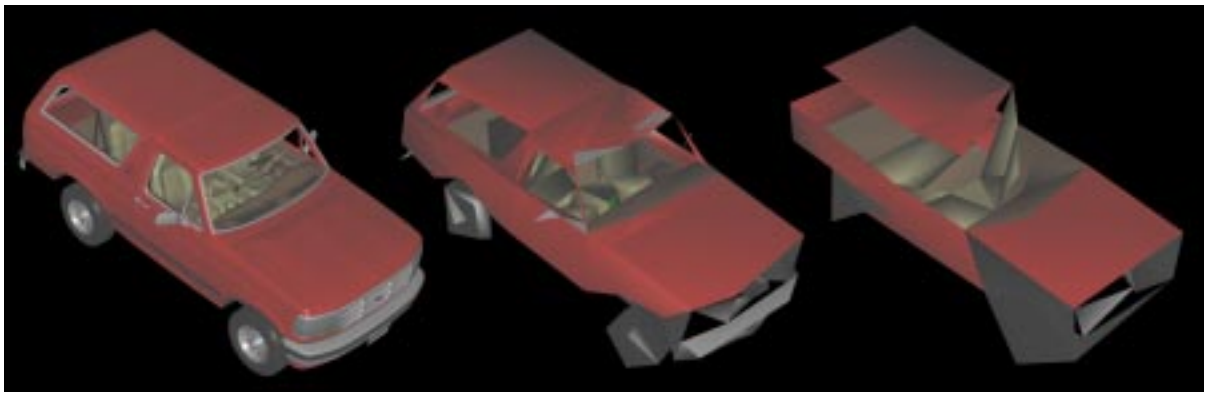


Figure 5.15: The original Bronco model consisting of 74,308 polygons and two HLODs consisting of 580 and 143 polygons respectively.

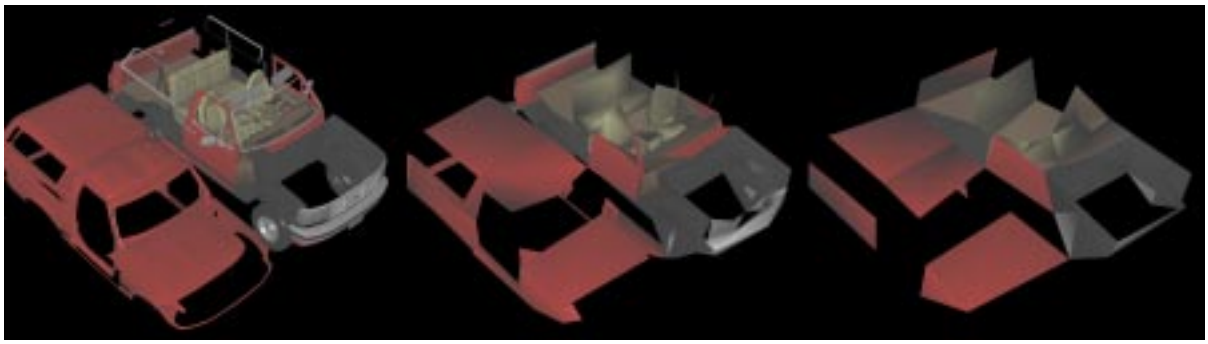


Figure 5.16: Dynamic modification of the Bronco model. We have moved the top of the Bronco in order to look into its interior. The two HLODs consist of 552 and 136 polygons respectively and took 3 seconds to recompute using 4 simplification processes on an SGI Reality Monster with 300 MHz R12000 processors and 16GB of main memory.

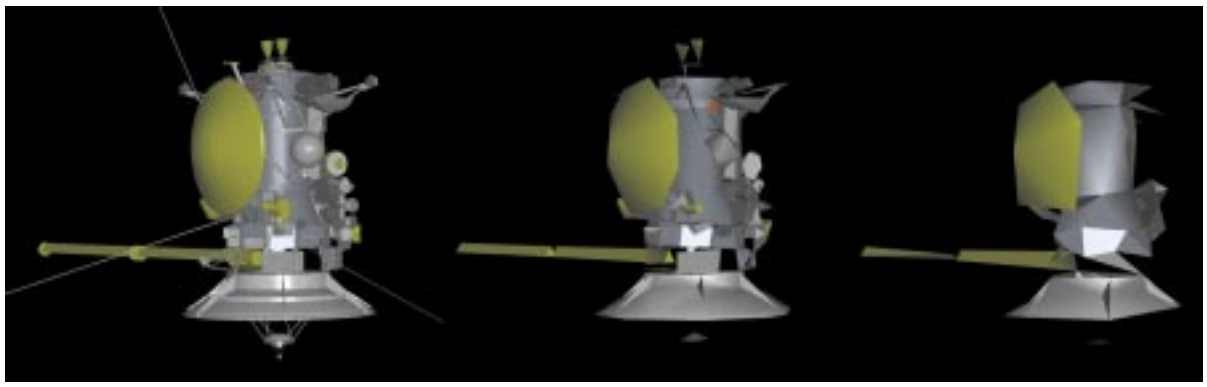


Figure 5.17: The original Cassini model consisting of 349,281 polygons and two HLODs consisting of 1,790 and 445 polygons respectively.

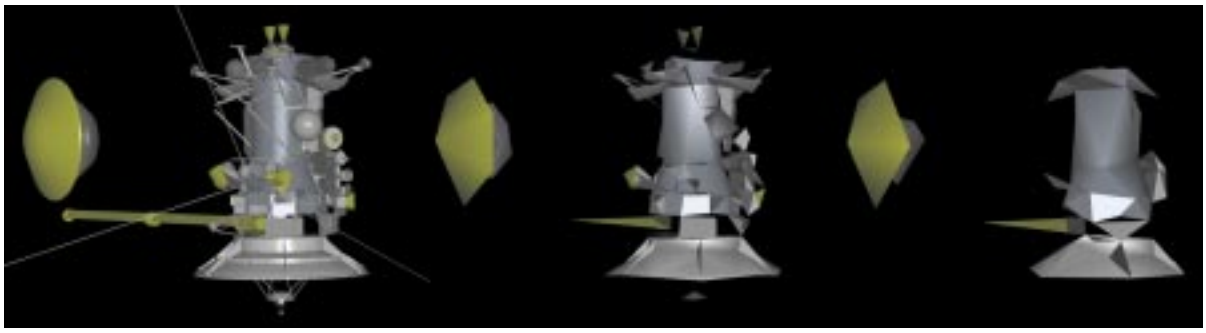


Figure 5.18: Dynamic modification of the Cassini model. We have moved the gold disc away from the Cassini. The two HLODs consist of 1,236 and 307 polygons respectively and took 6 seconds to recompute using 4 simplification processes on an SGI Reality Monster with 300 MHz R12000 processors and 16GB of main memory.

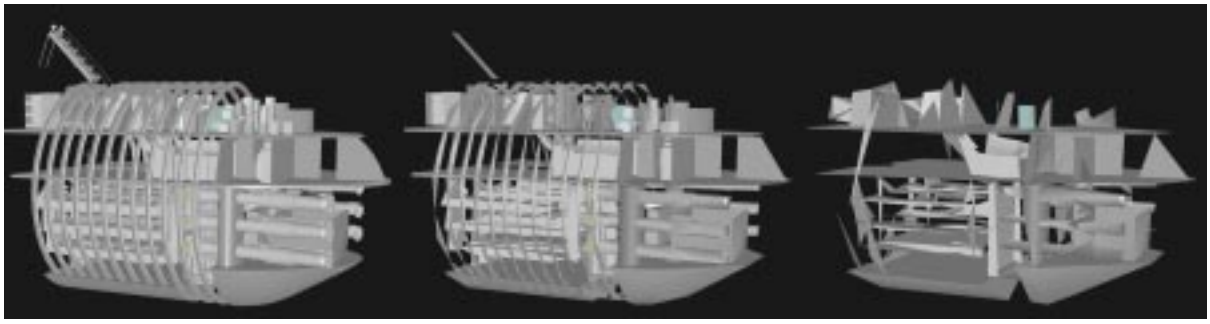


Figure 5.19: The original Torpedo Room model consisting of 883,537 polygons and two HLODs consisting of 5,572 and 1,393 polygons respectively.

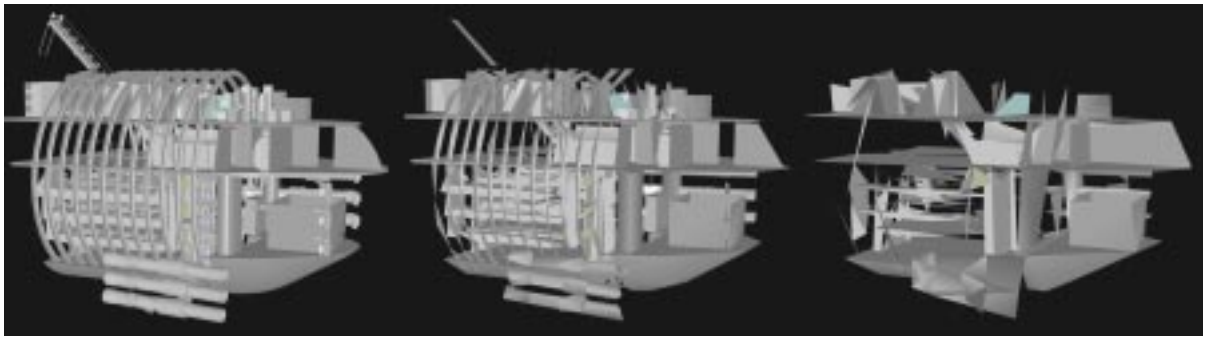


Figure 5.20: Dynamic modification of the Torpedo Room model. We have moved 3 of the torpedo tubes to the side of the main structure. These two HLODs consist of 5,191 and 1,296 polygons respectively and took 9 seconds to recompute using 4 simplification processes on an SGI Reality Monster with 300 MHz R12000 processors and 16GB of main memory.

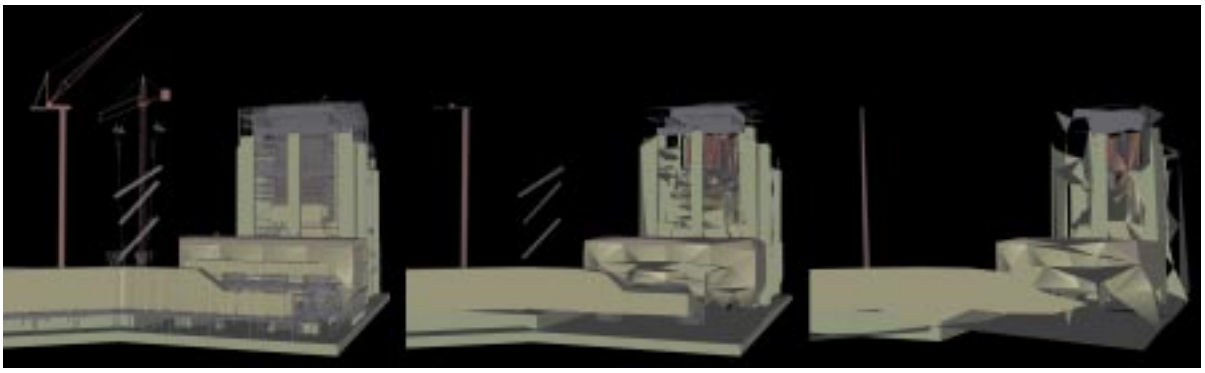


Figure 5.21: The original Power Plant model consisting of 12,731,154 polygons and two HLODs consisting of 9,384 and 2,379 polygons respectively.

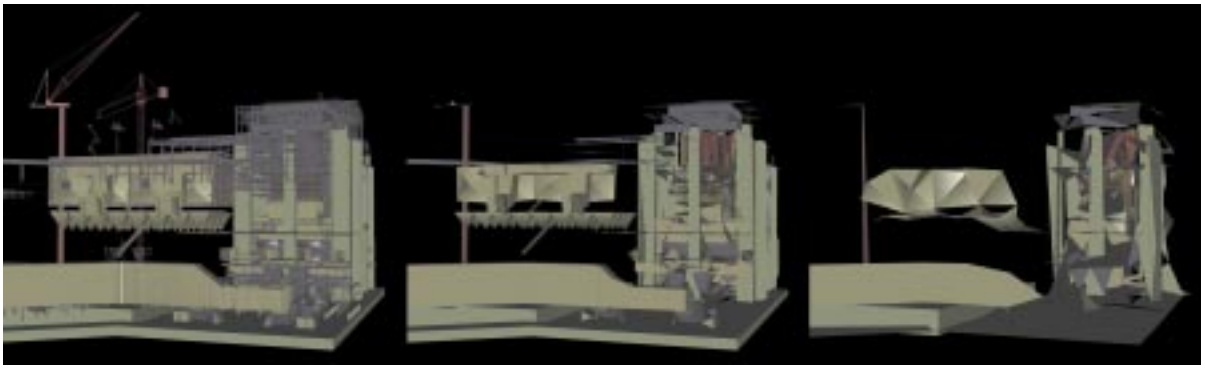


Figure 5.22: Dynamic modification of the Power Plant model. We have moved multiple parts in the scene around. These two HLODs consist of 9,441 and 2,395 polygons respectively and took 43 seconds to recompute using 4 simplification processes on an SGI Reality Monster with 300 MHz R12000 processors and 16GB of main memory

Scene	Objects	Triangles	HLOD Recalculation with 4 CPUs (secs.)
Bronco	466	74,308	3
Cassini	127	349,281	6
Torpedo Room	356	883,537	9
Power Plant	1,179	12,731,154	43

Table 5.4: HLOD recalculation execution speed for the simulations shown in the figures above.

5.4.5 Using LODs In Place of Invalid HLODs

HLODs can become invalid, meaning that they cannot possibly be acceptable representations of the scene graph viewed from any distance, due to the movement of objects (see Section 5.2.1). If the HLODs for a node are invalid, then our algorithm will not render these HLODs while they are in the process of being recomputed. By ignoring these HLODs, our algorithm must traverse further down the scene graph to find a valid representation consisting of LODs and HLODs. However, if we store a complete set of LODs for each node in the scene graph, then it is possible to use these LODs while HLODs are being recalculated. By rendering the individual LODs, it is possible to reduce the polygon count until valid HLODs are created. A complete set of LODs means that there are representations ranging from the original polygonal geometry down to one or a few polygons. Unfortunately, storing a complete set of LODs for each node in the scene graph requires more memory. Assuming each LOD consists of half the polygons of the previous one, then in the worst case, they could add another copy of the original model in terms of storage space.

5.5 Analysis

In this section, we derive a rough measure of how much motion our algorithm can handle. Since we use a number of symbols in the following discussion, they are summarized in Table 5.5.

b	Number of siblings of a node in the scene graph
c	Number of children for parent nodes
h	Height of the scene graph
M	Number of vertex merges needed to update HLODs after a single object moves
m	Number of vertex merges GAPS can perform in a second
n	Number of moving objects in the scene
r	How much the polygons of an HLOD are reduced by during simplification
s	Number of seconds before updated HLODs are needed for rendering
v	Number of vertices in an object in the scene graph

Table 5.5: Brief descriptions of symbols used in our analysis.

We first analyze the performance of updating the scene graph's bounding volume hierarchy and structure through node re-association. Suppose the scene graph is of height *h*,

and we move an object that has *b* siblings in the original scene graph. As described in Section 4.5, in the worst case our re-associated graph will be of height *b*. In non-degenerate cases, it will be of height *lg b*. Therefore, when we move a node, we expect the re-association of the scene graph to take *O(lg b)* time. However, we must also update the bounding volume hierarchy as well as HLODs all the way up the scene graph. Updating the bounding volume hierarchy takes *O(h)* time. Marking HLODs as invalid or inaccurate and then inserting them into a simplification queue takes *O(h)* time as well. Therefore, for each object that moves, the work involved in updating the scene graph is *O(h)*.

Next, we categorize the type of dynamic environment that our algorithm can currently handle. We use the following assumptions in our analysis.

- Since HLOD recalculation is the most time consuming operation to be performed when objects move, we ignore the cost of updating error bounds of HLODs, re-associating nodes, and updating the bounding volume hierarchy.
- GAPS performs *m* vertex merge operations per second.
- We use a single simplification process.
- The algorithm will not need to render the newly created HLODs before *s* seconds have passed.
- The height of the scene graph is *h* and polygons of objects are at leaf nodes. Therefore, for every object that moves, we must recalculate $h - 1$ sets of HLODs up the scene graph.
- The average number of vertices that make up the original polygonal geometry of an object is *v* and LODs or HLODs contain half the number of vertices and faces of the previous LOD or HLOD.
- GAPS simplifies a node's polygonal geometry containing *v* vertices until *v*/*r* vertices remain. These *v*/*r* vertices are combined into the parent of the node to form the base polygonal geometry for the HLODs of the parent.
- Each parent node in the scene graph has *c* children.

Assuming $c \neq r$, the number of vertex merges M that we must perform in such a scenario for each object that moves is

$$M =$$
$$\left[c\left(v/r\right) - c\left(v/r^2\right)\right] + \left[c^2\left(v/r^2\right) - c^2\left(v/r^3\right)\right] + \cdots + \left[c^{h-2}\left(v/r^{h-2}\right) - c^{h-2}\left(v/r^{h-1}\right)\right] + c^{h-1}\left(v/r^{h-1}\right) =$$
$$\left(cv/r\right)\left(1 - 1/r\right) + \left(c^2 v/r^2\right)\left(1 - 1/r\right) + \cdots + \left(c^{h-2} v/r^{h-2}\right)\left(1 - 1/r\right) + c^{h-1}\left(v/r^{h-1}\right) =$$
$$\left(cv/r\right)\left[\sum_{k=0}^{h-3}\left(c/r\right)^k\right]\left(1 - 1/r\right) + c^{h-1}\left(v/r^{h-1}\right) =$$
$$\left(cv/r\right)\left[\frac{\left(c/r\right)^{h-2} - 1}{\left(c/r\right) - 1}\right]\left(1 - 1/r\right) + c^{h-1}\left(v/r^{h-1}\right)$$

If $c = r$, then

$$M =$$
$$\left[c\left(v/r\right) - c\left(v/r^2\right)\right] + \left[c^2\left(v/r^2\right) - c^2\left(v/r^3\right)\right] + \cdots + \left[c^{h-2}\left(v/r^{h-2}\right) - c^{h-2}\left(v/r^{h-1}\right)\right] + c^{h-1}\left(v/r^{h-1}\right) =$$
$$\left(cv/r\right)\left(1 - 1/r\right) + \left(c^2 v/r^2\right)\left(1 - 1/r\right) + \cdots + \left(c^{h-2} v/r^{h-2}\right)\left(1 - 1/r\right) + c^{h-1}\left(v/r^{h-1}\right) =$$
$$\left(cv/r\right)\left[\sum_{k=0}^{h-3}\left(c/r\right)^k\right]\left(1 - 1/r\right) + c^{h-1}\left(v/r^{h-1}\right) =$$
$$v(h-2)\left(1 - 1/r\right) + v =$$
$$v\left[(h-2)\left(1 - 1/r\right) + 1\right]$$

If $M \leq ms$ then our algorithm is capable of recomputing the affected HLODs before they are needed for rendering.

The basic idea behind these formulas is that they are calculating the number of vertex merges required to update HLODs up the entire scene graph. For this analysis, we assume that objects are located at leaf nodes. HLODs of a leaf node are equivalent to its LODs so they cannot change. Thus, we first calculate the number of vertex merges required to recompute HLODs for the parent node of the leaf node. To perform this calculation, we need to know how many vertices the parent node will pool together from its children. By our assumptions, each child's coarsest HLOD contains v/r vertices and there are c children. Therefore, cv/r vertices are pooled. The coarsest HLOD of the parent node will consist of

cv/r^2 vertices since the cv/r vertices will be reduced by $1/r$. The number of vertex merge operations required to create this coarsest HLOD is thus $cv/r - cv/r^2$. This number of vertex merge operations is represented by the first term in the definition of M. Subsequent terms are created using recursion. For example, for the grandparent node, each child's coarsest HLOD contains cv/r^2 vertices and there are c children. Therefore, c^2v/r^2 vertices are pooled. This recursion terminates at the root node of the scene graph.

For a more concrete example, suppose we are attempting to recalculate HLODs for the Torpedo Room model while we render it. This model consists of 356 objects, 883,537 polygons, and 545,949 vertices. The scene graph has an average of 3.1 children per node, its HLOD reduction ratio is approximately 8.3, and its height is 7. Assume HLODs must be recalculated in 10 seconds and that GAPS performs, on average, 650 vertex merge operations per second on an SGI Reality Monster with a 300 MHz R12000 processor and 16GB of main memory. Then the above constants for the Torpedo Room model are approximately

$$
\begin{aligned}
m &\approx 650 \\
s &= 10 \\
c &\approx 3.1 \\
h &= 7 \\
v &\approx {}^{545949}\!/_{356} \approx 1534 \\
r &\approx 8.3
\end{aligned}
$$

These constants imply that

$$M \approx 803$$

Since $803 = M \leq ms = 6500,$ our algorithm should be able to handle this scenario within the given time constraint. Suppose we move n objects in the scene. As long as $nM \leq ms,$ our algorithm should still be able to recalculate the HLODs within the time constraint. Thus, our analysis suggests that our current implementation should be able to update the Torpedo Room HLODs within 10 seconds after 8 objects have moved ($8 \cdot 803 = 6424 \leq 6500$). In practice, our algorithm took approximately 13 seconds to update the HLODs of the scene using 1 simplification process after moving 8 parts of the Torpedo Room model. In this case, the results of our analysis are overoptimistic. However, it uses

approximate values for *m*, *c*, *v*, and *r* and ignores the execution cost of updating the scene graph and semaphore access. Thus, this model of performance should only be used as a rough guide to how much dynamic movement our algorithm can handle.

5.6 Comparison

Not much research has been performed on simplifying dynamic environments. In fact, not many rendering systems handle dynamic environments at all. Since traditional scene graph methods such as [Rohlf and Helman 94] use only LODs, dynamic environments are handled by simply transforming arcs in the scene graph. Since our system extends the traditional scene graph with HLODs, dynamic movement causes these HLODs to become inaccurate or even invalid. To retain the benefits of HLODs, namely the ability to merge polygons from distinct objects in the scene graph, we must update these HLODs as quickly as possible. For scenarios with limited dynamic movement, this recomputation is possible and provides higher quality coarse approximations of the scene. However, the memory requirements of our dynamic visualization algorithm are large and its implementation is more complex than traditional scene graph systems mainly due to its simultaneous use of processes that render and simplify.

5.7 Summary

We have demonstrated an algorithm that updates simplification approximations of groups of objects in response to dynamic movement in a polygonal environment. When objects move in the scene, our algorithm updates error bounds of affected HLODs, re-associates nodes, and recalculates the bounding volume hierarchy of the scene graph. Some HLODs may become inaccurate or even invalid due to this movement. Our approach uses asynchronous simplification processes to recalculate these HLODs while the rendering process continues to render. Once these HLODs have been recomputed, they can be used to accelerate the rendering of the polygonal environment or be used as coarse approximations when targeting a frame rate (see Chapter 4). Our analysis shows that our current implementation is best suited for environments with limited dynamic movement, such as

design and review scenarios. We presented a simple model to determine if our algorithm is able to recompute HLODs quickly enough to handle a given dynamic environment. In practice, our algorithm has been used on several large polygonal environments where we allowed the user to interactively manipulate objects in the scene.

6 CONCLUSION

6.1 New Results

This dissertation presents new techniques in the areas of simplification of static polygonal objects, simplification of static polygonal environments, and simplification of dynamic polygonal environments.

6.1.1 Simplification of Static Polygonal Objects

We presented GAPS, an algorithm that simultaneously achieves the following goals for the simplification of static polygonal objects:

- It handles objects with degenerate polygonal geometry.
- It does not require any user input.
- It handles surface attributes during simplification.
- It uses a unified error metric, combining both geometric and surface attribute error, that can be used to automatically calculate switching distances of LODs.
- It produces high quality and drastic approximations of objects.
- It can reduce the number of polygons of an object to any target number.
- It merges unconnected regions of polygons using automatic topological simplification.
- It executes quickly.

6.1.2 Simplification of Static Polygonal Environments

We presented an algorithm that accelerates the rendering of large static polygonal environments. It achieves the following goals for the simplification of static polygonal environments:

- It creates hierarchical levels of detail, representing groups of objects in the scene graph, using GAPS.
- It associates nodes in the scene graph based on their spatial proximity.
- It partitions spatially large objects in order to gain limited view-dependent rendering capabilities.
- It can render in a pixel-error or target frame-rate mode.
- It efficiently renders polygonal geometry using display lists.

6.1.3 Simplification of Dynamic Polygonal Environments

We presented an algorithm that updates simplification approximations when objects in the scene move. It achieves the following goals for the simplification of dynamic polygonal environments:

- It re-associates nodes in the scene graph based on their spatial proximity after objects move.
- It uses asynchronous simplification processes to update HLODs that are either inaccurate or invalid due to object movement. Therefore, the rendering process can continue to render while the simplification processes simplify.

6.2 Future Work

This section describes possible avenues for future work in the fields of simplification of static polygonal objects, simplification of static polygonal environments, and simplification of dynamic polygonal environments.

6.2.1 Simplification of Static Polygonal Objects

We would like to improve our approach for handling surface attributes, as it is currently a very approximate technique. Combining our topological simplification techniques with appearance preserving methods in [Cohen et al. 98] might produce higher quality simplifications. Now that more simplification algorithms are handling surface attributes, future work may include developing a comparison tool that measures the quality of each algorithm's output. This tool would define "quality" by using an error metric that involves both polygonal geometry and surface attributes.

We would like to incorporate the work of [Lindstrom and Turk 98] in order to lessen the memory consumption of GAPS. Not only would these techniques help simplification of static polygonal environments, but it would also aid the memory requirements of our dynamic algorithm as well.

We are interested in creating progressive mesh representations [Hoppe 96] from the topological simplifications produced by GAPS. Our topological simplification techniques might also be extended to models composed of spline primitives [Gopi and Manocha 98].

Furthermore, since our approach tends to produce solid shapes for close, but disjoint polygonal geometry, we are interested in using GAPS to create approximations for occluders that can be used by visibility algorithms [Greene et al. 93, Coorg and Teller 96, Hudson et al. 97, Zhang et al. 97].

GAPS uses spatial partitioning to determine which vertices are in close proximity. However, a vertex that is close to a large face, but not to another vertex, will never be moved to close the gap between the vertex and the face. This behavior is due to undersampling. The smaller the faces, the less likely this scenario will be a problem. Simplification techniques tend to be vertex-centric, but maybe future research will consider the proximity of faces as well as vertices to produce higher quality simplifications.

Finally, through the use of surface area preservation, GAPS prohibits certain vertex merges from occurring. By not merging these vertices, we are merging another pair that produces greater error according to the quadric error metric. One component of our unified

error metric, presented in Section 3.3.3.4, is the quadric error metric. Thus, according to our error metric, using surface area preservation creates lower quality approximations. However, these approximations tend to be higher quality representations for the object even though our error metric says otherwise. We would like to extend our error metric so that it measures error more in line with the subjective visual quality of the approximation.

6.2.2 Simplification of Static Polygonal Environments

Our system works well on complicated polygonal environments where most of the polygonal geometry is closely spaced. For wide-open environments, the power of HLODs is greatly diminished. Therefore, we could extend our algorithm to judge which regions of the environment would benefit from HLODs and which would not. For example, if we associate two nodes but their bounding spheres are not close to intersecting, then there is probably little point in creating HLODs for this pair of nodes.

Also, since HLODs tend to be coarse, but solid approximations of objects, we are interested in using them as occluders for visibility algorithms.

6.2.3 Simplification of Dynamic Polygonal Environments

We would like our dynamic algorithm to handle environments that exhibit a great deal of dynamic movement. In order to do so, we would have to improve our algorithm in several ways. First, by speeding up the execution of GAPS, we could simplify more objects in less time. Second, by updating error bounds of HLODs, re-associating nodes, and updating the bounding volume hierarchy more efficiently when objects move, we could handle more restructuring of the scene graph in less time. Currently, once each object moves, the scene graph is updated incrementally. If we updated the scene graph on multiple processes, we could accelerate the process for large scene graphs. Finally, instead of using three global semaphores to access data in the scene graph (see Section 5.3.3), we are interested in using semaphores that are localized to portions of the scene graph. These localized semaphores might decrease the contention between simplification processes and therefore accelerate HLOD recomputation.

Our current dynamic implementation requires a great deal of memory, about 6 times the memory of the original polygonal environment. We would like to investigate ways to reduce this memory consumption. Note that by making GAPS more memory efficient, we would cut down on our memory usage for dynamic environments.

Finally, we would like to extend our techniques to handle more general dynamic movement such as deforming polygonal objects. However, given that our current implementation can only handle limited rigid-body dynamic movement, this extension seems very difficult.

APPENDIX

Suppose there are *n* planes stored in an error quadric for a particular vertex. Suppose that d_I represents the distance between the vertex and the *i*th plane in its error quadric. The proof we demonstrate below shows that the square root of the average of squared distances is greater than or equal to the average distance from the vertex to its set of planes. It implies that our approximation to the average distance in Section 3.3.3.2 is conservative.

Lemma 1 If $a, b \in \Re$, then

$$a^2 + b^2 \geq 2ab$$

Proof.

$$(a-b)^2 \geq 0 \Rightarrow a^2 - 2ab + b^2 \geq 0 \Rightarrow a^2 + b^2 \geq 2ab$$

Lemma 2

$$n\sum_{i=0}^{n-1} d_i^{\,2} \geq \left(\sum_{i=0}^{n-1} d_i\right)^2$$

Proof. Using Lemma 1,

$$\begin{aligned}
&n\sum_{i=0}^{n-1} d_i^{\,2} = n\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) = \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + (n-1)\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) = \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + \left[\left(d_0^{\,2} + d_1^{\,2}\right) + \left(d_0^{\,2} + d_2^{\,2}\right) + \ldots + \left(d_0^{\,2} + d_{n-1}^{\,2}\right)\right] + \\
&(n-2)\left(d_1^{\,2} + d_2^{\,2} + \ldots + d_{n-1}^{\,2}\right) \geq \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + \left[2d_0d_1 + 2d_0d_2 + \ldots + 2d_0d_{n-1}\right] + \\
&(n-2)\left(d_1^{\,2} + d_2^{\,2} + \ldots + d_{n-1}^{\,2}\right) = \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + \left[2d_0d_1 + 2d_0d_2 + \ldots + 2d_0d_{n-1}\right] + \\
&\left[\left(d_1^{\,2} + d_2^{\,2}\right) + \left(d_1^{\,2} + d_3^{\,2}\right) + \ldots + \left(d_1^{\,2} + d_{n-1}^{\,2}\right)\right] + (n-3)\left(d_2^{\,2} + d_3^{\,2} + \ldots + d_{n-1}^{\,2}\right) \geq \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + \left[2d_0d_1 + 2d_0d_2 + \ldots + 2d_0d_{n-1}\right] + \\
&\left[2d_1d_2 + 2d_1d_3 + \ldots + 2d_1d_{n-1}\right] + (n-3)\left(d_2^{\,2} + d_3^{\,2} + \ldots + d_{n-1}^{\,2}\right) = \\
&\ldots \geq \\
&\left(d_0^{\,2} + d_1^{\,2} + \ldots + d_{n-1}^{\,2}\right) + \left[2d_0d_1 + 2d_0d_2 + \ldots + 2d_0d_{n-1}\right] + \\
&\left[2d_1d_2 + 2d_1d_3 + \ldots + 2d_1d_{n-1}\right] + \ldots + \left[2d_{n-2}d_{n-1}\right] = \\
&\left(\sum_{i=0}^{n-1} d_i\right)^2
\end{aligned}$$

Theorem 1

$$\sqrt{\frac{\sum_{i=0}^{n-1} d_i^{\,2}}{n}} \geq \frac{\sum_{i=0}^{n-1} d_i}{n}$$

Proof. Using Lemma 2,

$$n\sum_{i=0}^{n-1} d_i^{\,2} \geq \left(\sum_{i=0}^{n-1} d_i\right)^2 \Rightarrow \frac{\sum_{i=0}^{n-1} d_i^{\,2}}{n} \geq \frac{\left(\sum_{i=0}^{n-1} d_i\right)^2}{n^2} \Rightarrow \sqrt{\frac{\sum_{i=0}^{n-1} d_i^{\,2}}{n}} \geq \frac{\sum_{i=0}^{n-1} d_i}{n}$$

REFERENCES

[Airey et al. 90] Airey, J., Rohlf, J., and Brooks, F., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," *Symposium on Interactive 3D Graphics '90 Proceedings,* 41-50, 1990.

[Aliaga 97] Aliaga, D., "SGI Performance Tips," *http://www.cs.unc.edu/~aliaga/IR.html,* 1997.

[Aliaga and Lastra 99] Aliaga, D. and Lastra, A., "Automatic Image Placement to Provide a Guaranteed Frame Rate," *Computer Graphics (SIGGRAPH '99 Proceedings),* 307-316, 1999.

[Bergman et al. 86] Bergman, L., Fuchs, H., Grant, E., and Spach, S., "Image Rendering by Adaptive Refinement," *Computer Graphics (SIGGRAPH '86 Proceedings),* 29-37, 1986.

[Chrysanthou and Slater 92] Chrysanthou, Y., and Slater, M., "Computing Dynamic Changes to BSP Trees," *Computer Graphics Forum (Eurographics '92 Proceedings),* 321-332, 1992.

[Clark 76] Clark, J., "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM,* 547-554, 1976.

[Cohen et al. 96] Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F., and Wright, W., "Simplification Envelopes," *Computer Graphics (SIGGRAPH '96 Proceedings),* 119-128, 1996.

[Cohen et al. 97] Cohen, J., Manocha, D., and Olano, M., "Simplifying Polygonal Models Using Successive Mappings," *IEEE Visualization '97 Proceedings,* 395-402, 1997.

[Cohen et al. 98] Cohen, J., Olano, M., and Manocha, D., "Appearance-Preserving Simplification," *Computer Graphics (SIGGRAPH '98 Proceedings),* 115-122, 1998.

[Coorg and Teller 96] Coorg, S., and Teller, S., "Temporally Coherent Conservative Visibility," *Proceedings of 12th ACM Symposium on Computational Geometry,* 1996.

[Cormen et al. 94] Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms,* MIT Press and McGraw-Hill, 1994.

[Eck et al. 95] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W., "Multiresolution Analysis of Arbitrary Meshes," *Computer Graphics (SIGGRAPH '95 Proceedings),* 173-182, 1995.

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

[Erikson 96] Erikson, C., "Polygonal Simplification: An Overview," *UNC Chapel Hill Computer Science Technical Report TR96-016,* 1996.

[Erikson 99] Erikson, E., *Personal communication with Erik Erikson of Red Storm Entertainment, Rendering Software Engineer of Rainbow Six,* 1999.

[Eyles et al. 97] Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N., and Westover, L., "PixelFlow: The Realization," *SIGGRAPH/Eurographics Workshop on Graphics Hardware '97 Proceedings,* 57-68, 1997.

[Funkhouser and Séquin 93] Funkhouser, T., and Séquin, C., "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments," *Computer Graphics (SIGGRAPH '93 Proceedings),* 247-254, 1993.

[Garland 97] Garland, M., "QSlim Simplification Software," *http://www.cs.cmu.edu/~garland/quadrics/qslim.html,* 1997.

[Garland and Heckbert 97] Garland, M., and Heckbert, P., "Surface Simplification Using Quadric Error Metrics," *Computer Graphics (SIGGRAPH '97 Proceedings),* 209-216, 1997.

[Garland and Heckbert 98] Garland, M., and Heckbert, P., "Simplifying Surfaces with Color and Texture using Quadric Error Metrics," *IEEE Visualization '98 Proceedings,* 263-269, 1998.

[Gopi and Manocha 98] Gopi, M., and Manocha, D., "A Unified Approach for Simplifying Polygonal and Spline Models," *IEEE Visualization '98 Proceedings,* 271-278, 1998.

[Greene et al. 93] Greene, N., Kass, M., and Miller, G., "Hierarchical Z-Buffer Visibility," *Computer Graphics (SIGGRAPH '93 Proceedings),* 231-238, 1993.

[He et al. 95] He, T., Hong, L., Kaufman, A., Varshney, A., and Wang, S., "Voxel-Based Object Simplification," *IEEE Visualization '95 Proceedings,* 296-303, 1995.

[Heckbert and Garland 94] Heckbert, P., and Garland, M., "Multiresolution Modeling for Fast Rendering," *Graphics Interface '94 Proceedings,* 43-50, 1994.

[Heckbert and Garland 97] Heckbert, P., and Garland, M., "Survey of Polygonal Surface Simplification Algorithms," *Draft of Carnegie Mellon University Computer Science Technical Report (http://www.cs.cmu.edu/~garland/Papers/simp.pdf),* 1997.

[Hoppe et al. 93] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W., "Mesh Optimization," *Computer Graphics (SIGGRAPH '93 Proceedings),* 19-26, 1993.

[Hoppe 96] Hoppe, H., "Progressive Meshes," *Computer Graphics (SIGGRAPH '96 Proceedings),* 99-108, 1996.

[Hoppe 97] Hoppe, H., "View-Dependent Refinement of Progressive Meshes," *Computer Graphics (SIGGRAPH '97 Proceedings)*, 189-198, 1997.

[Hudson et al. 97] Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., and Zhang, H., "Occlusion Culling using Shadow Volumes," *Proceedings of 13th ACM Symposium on Computational Geometry*, 1997.

[Knuth 73] Knuth, D., *Sorting and Searching, volume 3 of The Art of Computer Programming*, Addision-Wesley, 1973.

[Lindstrom and Turk 98] Lindstrom, P., and Turk,G., "Fast and Memory Efficient Polygonal Simplification," *IEEE Visualization '98 Proceedings*, 279-286, 1998.

[Lorensen and Cline 87] Lorensen, W., and Cline, H., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics (SIGGRAPH '87 Proceedings)*, 163-169, 1987.

[Low and Tan 97] Low, K., and Tan, T., "Model Simplification Using Vertex-Clustering," *Symposium on Interactive 3D Graphics '97 Proceedings*, 75-82, 1997.

[Luebke and Georges 95] Luebke, D. and Georges, C., "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets," *Symposium on Interactive 3D Graphics '95 Proceedings*, 105-106, 1995

[Luebke 97] Luebke, D., "A Survey of Polygonal Simplification Algorithms," *UNC Chapel Hill Computer Science Technical Report TR97-045*, 1997.

[Luebke and Erikson 97] Luebke, D., and Erikson, C., "View-Dependent Simplification of Arbitrary Polygonal Environments," *Computer Graphics (SIGGRAPH '97 Proceedings)*, 199-208, 1997.

[Maciel and Shirley 95] Maciel, P., and Shirley, P., "Visual Navigation of Large Environments Using Textured Clusters," *Symposium on Interactive 3D Graphics '95 Proceedings*, 95-102, 1995.

[Molnar et al. 92] Molnar, S., Eyles, J., and Poulton, J., "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics (SIGGRAPH '92 Proceedings)*, 231-240, 1992.

[Montrym et al. 97] Montrym, J., Baum, D., Dignam, D., and Migdal, C., "InfiniteReality: A Real-Time Graphics System," *Computer Graphics (SIGGRAPH '97 Proceedings)*, 293-302, 1997.

[Mueller 95] Mueller, C., "Architectures of Image Generators for Flight Simulators," *UNC Chapel Hill Computer Science Technical Report TR95-015*, 1995.

[Popovic and Hoppe 97] Popovic, J., and Hoppe, H., "Progressive Simplicial Complexes," *Computer Graphics (SIGGRAPH '97 Proceedings)*, 217-224, 1997.

[Rohlf and Helman 94] Rohlf, J., and Helman, J., "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Computer Graphics (SIGGRAPH '94 Proceedings)*, 381-394, 1994

[Ronfard and Rossignac 96] Ronfard, R., and Rossignac, J., "Full-range Approximation of Triangulated Polyhedra," *Computer Graphics Forum (Eurographics '96 Proceedings)*, 67-76, 1996.

[Rossignac and Borrel 93] Rossignac, J., and Borrel, P., "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, 455-465, 1993.

[Rossignac 97] Rossignac, J., "Geometric Simplification and Compression," *Multiresolution Surface Modeling SIGGRAPH '97 Course Notes*, 1997.

[Schaufler and Stuerzlinger 96] Schaufler, G., and Stuerzlinger, W. "Three Dimensional Image Cache for Virtual Reality," *Computer Graphics Forum (Eurographics '96 Proceedings)*, 227-235, 1996.

[Schneider et al. 94] Schneider, B., Borrel, P., Menon, J., Mittleman, J., and Rossignac, J., "Brush as a Walkthrough System for Architectural Models," *Fifth Eurographics Workshop on Rendering*, 389-399, 1994.

[Schroeder et al. 92] Schroeder, W., Zarge, J., and Lorensen, W., "Decimation of Triangle Meshes," *Computer Graphics (SIGGRAPH '92 Proceedings)*, 65-70, 1992.

[Schroeder 97] Schroeder, W., "A Topology Modifying Progressive Decimation Algorithm," *IEEE Visualization '97 Proceedings*, 205-212, 1997.

[Shade et al. 96] Shade, J., Lischinski, D., Salesin, D., DeRose, T., and Snyder, J., "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments," *Computer Graphics (SIGGRAPH '96 Proceedings)*, 75-82, 1996.

[Sudarsky and Gotsman 96] Sudarsky, O., and Gotsman, C., "Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality," *Computer Graphics Forum (Eurographics '96 Proceedings)*, 249-258, 1996.

[Teller and Séquin 91] Teller, S., and Séquin, C., "Visibility Preprocessing for Interactive Walkthroughs," *Computer Graphics (SIGGRAPH '91 Proceedings)*, 61-69, 1991.

[Torres 90] Torres, E., "Optimization of the Binary Space Partitioning Algorithm (BSP) for the Visualization of Dynamic Scenes," *Computer Graphics Forum (Eurographics '90 Proceedings)*, 507-518, 1990.

[Turk 92] Turk, G., "Re-Tiling Polygonal Surfaces," *Computer Graphics (SIGGRAPH '92 Proceedings)*, 55-64, 1992.

[Turk 94] Turk, G., "Ply Model Format Tools," http://www-graphics.stanford.edu/data/3Dscanrep", 1994.

[Xia et al. 97] Xia, J., El-Sana, J., Varshney, A., "Adaptive Real-Time Level-of-Detail-Based-Rendering for Polygonal Models," *IEEE Transactions on Visualization and Computer Graphics*, 171-183, 1997.

[Zhang et al. 97] Zhang, H., Manocha, D., Hudson, T., Hoff, K., "Visibility Culling using Hierarchical Occlusion Maps," *Computer Graphics (SIGGRAPH '97 Proceedings)*, 77-88, 1997.