View-Dependent Simplification

of Arbitrary Polygonal Environments

By

David P. Luebke

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

1998

Approved by:

Advisor: Professor Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Reader: Professor Anselmo Lastra
University of North Carolina at Chapel Hill

Reader: Professor Greg Turk,
Georgia Institute of Technology

ABSTRACT

DAVID P. LUEBKE: Hierarchical Dynamic Simplification
(Under the direction of Professor Frederick P. Brooks, Jr.)

This dissertation describes *hierarchical dynamic simplification (HDS)*, a new approach to the problem of simplifying arbitrary polygonal environments. HDS is *dynamic*, retessellating the scene continually as the user's viewing position shifts, and *global*, processing the entire database without first decomposing the environment into individual objects. The resulting system enables real-time display of very complex polygonal CAD models consisting of thousands of parts and millions of polygons. HDS supports various preprocessing algorithms and various run-time criteria, providing a general framework for dynamic view-dependent simplification.

Briefly, HDS works by clustering vertices together in a hierarchical fashion. The simplification process continually queries this hierarchy to generate a scene containing only those polygons that are important from the current viewpoint. When the volume of space associated with a vertex cluster occupies less than a user-specified amount of the screen, all vertices within that cluster are collapsed together and degenerate polygons filtered out. HDS maintains an *active list* of visible polygons for rendering. Since frame-to-frame movements typically involve small changes in viewpoint, and therefore modify this list by only a few polygons, the method takes advantage of temporal coherence for greater speed.

To Steven Janke, who introduced me to the world of computer graphics, and
to Emily Luebke, who keeps me grounded in the larger world beyond.

ACKNOWLEDGMENTS

I would like to thank my advisor and mentor Dr. Fred Brooks, who somehow saw promise in a very green young chemistry major with a single computer science course to his name. I have worked for and with Dr. Brooks during my entire stay at Carolina, and would not have it any other way. He is an excellent scientist, a wonderful teacher, a skillful manager, a talented writer, and the wisest student of human nature I have ever met. I have learned more from Dr. Brooks than I can possibly condense into these acknowledgements.

I would also like to thank Greg Turk for his invaluable technical advice, especially in the early stages of my research when my ideas for view-dependent simplification were only vaguely formed. Greg exemplifies the kind of professor I would like to be; he is creative, knowledgeable, excited, fun to work with, and a consummate researcher. Dinesh Manocha has been a constant source of good advice, and his unflagging energy has never ceased to amaze and inspire me. When my students find me in the lab at 3 A.M. the night of the deadline, they can thank (or curse) Dinesh. Lastly, I would like to thank Nick England and Anselmo Lastra for serving on my committee. Both have provided the valuable perspective of experts in computer graphics but not simplification.

I have spent three interesting, informative, and enjoyable summers at IBM's T.J. Watson Research Center. During those summers I had the privilege of working with and learning from Paul Borrel, Josh Mittleman, Jai Menon, and Fausto Bernardini. I owe my interest in polygonal simplification to my work on the 3DIX project, my dissertation topic to a casual conversation with Josh in the cafeteria, and my very education to IBM and the IBM Cooperative Fellowship program. My sincere thanks to everyone at IBM.

Special thanks go to my friends, teammates, and fellow students Carl Erikson, Bill Mark, and Mark Parris. Dan Aliaga, Chris Georges, Stefan Gottschalk, Terry Hopkins, and Jon McAllister have all given me their ideas, encouragement, and friendship. Finally, my deepest thanks and all my love to Emily Luebke, who has given me these and the world besides.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION...1

1.1. Polygons in Computer Graphics 1

1.2. Polygonal Simplification 3

1.3. Motivation For a New Approach 4

1.4. Hierarchical Dynamic Simplification 9

1.5. HDS as a Framework 12

1.6. Thesis Statement 13

1.7. Outline of Argument 13

CHAPTER 2 PREVIOUS WORK 14

2.1. Taxonomy 15

2.2. Catalog of Important Papers 18

CHAPTER 3 STRUCTURES AND METHODS 29

3.1. The Vertex Tree 29

3.2. The Active Triangle List 32

3.3. Methods 34

CHAPTER 4 VIEW-DEPENDENT SIMPLIFICATION 36

4.1. Screenspace Error Threshold 37

4.2. Silhouette Preservation 38

4.3. Triangle-Budget Simplification 41

CHAPTER 5 OPTIMIZING THE ALGORITHM......................................43

5.1. Exploiting Temporal Coherence 43

5.2. Visibility: Accelerating Rendering 47

5.3. Visibility: Accelerating Simplification 50

5.4. Streamlining the Math 50

5.5. Parallelization: Asynchronous Simplification 52

CHAPTER 6 CONSTRUCTING THE VERTEX TREE 54
6.1. Simplest: Spatial Subdivision 54
6.2. Prettiest: Simplification Envelopes, Progressive Mesh Algorithm 56
6.3. A Hybrid Approach 57
CHAPTER 7 RESULTS AND ANALYSIS 59
7.1. The Platform 59
7.2. The Models 59
7.3. The Path 65
7.4. Visual Results 66
7.5. Run-time Performance 74
7.6. Vertex Tree Characteristics 77
7.7. Preprocessing Performance 80
7.8. Artifacts 81
7.9. Preprocessing Complexity 84
CHAPTER 8 CONTEMPORARY RELATED WORK 88
8.1. Recent published algorithms 88
8.2. Issues and Trends 94
CHAPTER 9 SUMMARY AND FUTURE WORK 98
9.1. Summary 98
9.2. Future Work 99
CHAPTER 10 REFERENCES 103
APPENDIX 106

LIST OF FIGURES

FIGURE 1: THE TRADITIONAL APPROACH TO POLYGONAL SIMPLIFICATION 4

FIGURE 2: TWO MASSIVE CAD MODELS 5

FIGURE 3: PRESERVING GENUS LIMITS DRASTIC SIMPLIFICATION 6

FIGURE 4: A DIESEL ENGINE MODEL WITH OVER 200 PARTS 8

FIGURE 5: A SIMPLE MESH AND ASSOCIATED VERTEX TREE 10

FIGURE 6: A SEQUENCE OF FOLD OPERATIONS 11

FIGURE 7: TRIS AND SUBTRIS OF A NODE IN THE VERTEX TREE 30

FIGURE 8: THE VERTEX TREE, ACTIVE TREE, AND BOUNDARY NODES 31

FIGURE 9: CORNERS AND PROXIES 33

FIGURE 10: VIEW-DEPENDENT SIMPLIFICATION 37

FIGURE 11: SILHOUETTE PRESERVATION 39

FIGURE 12: SILHOUETTE PRESERVATION AND BACKFACE SIMPLIFICATION 40

FIGURE 13: TRIANGLES ADDED, DELETED, AND ADJUSTED DURING A 700-FRAME PATH ... 44

FIGURE 14: VISIBLE, INVISIBLE, AND IRRELEVANT NODES 48

FIGURE 15: CALCULATING THE SCREENSPACE EXTENT OF A NODE 50

FIGURE 16: PLOT OF TIME SPENT IN ADJUSTTREE( ) 52

FIGURE 17: A 2-D EXAMPLE OF OCTREE VS. TIGHT-OCTREE CLUSTERING 55

FIGURE 18: THE SPHERE MODEL 60

FIGURE 19: THE BUNNY MODEL 61

FIGURE 20: THE SIERRA TERRAIN 61

FIGURE 21: THE CASSINI SPACE PROBE MODEL 62

FIGURE 22: THE AUXILIARY MACHINE ROOM MODEL 62

FIGURE 23: THE TORPEDO ROOM MODEL 63

FIGURE 24: A CLOSE-UP OF THE TORP MODEL 63
FIGURE 25: THE BONE6 MODEL 64
FIGURE 26: THE POWERPLANT_4M MODEL 64
FIGURE 27: INTERIOR VIEW OF THE POWERPLANT_4M MODEL 65
FIGURE 28: THE AMR SHOWN AT ORIGINAL RESOLUTION 66
FIGURE 29: THE AMR MODEL AT 0.7% SCREENSPACE ERROR TOLERANCE 67
FIGURE 30: THE AMR MODEL AT 2.5% SCREENSPACE ERROR TOLERANCE 68
FIGURE 31: THE TORP MODEL AT ORIGINAL RESOLUTION 69
FIGURE 32: THE TORP MODEL SHOWN AT 0.8% SCREENSPACE ERROR TOLERANCE 70
FIGURE 33: THE TORP MODEL SHOWN AT 1.5% SCREENSPACE ERROR TOLERANCE 71
FIGURE 34: THE BUNNY MODEL SHOWN AT 1% SCREENSPACE ERROR TOLERANCE 72
FIGURE 35: THE BUNNY MODEL SHOWN AT 5% SCREENSPACE ERROR TOLERANCE 73
FIGURE 36: THE BUNNY MODEL SHOWN WITH SILHOUETTE PRESERVATION 74
FIGURE 37: TIMINGS AND TRIANGLE COUNT FOR THE TORP MODEL 75
FIGURE 38: TIMINGS AND TRIANGLE COUNT FOR THE CASSINI MODEL 76
FIGURE 39: A HISTOGRAM OF THE DEPTHS OF LEAF NODES IN THE VERTEX TREE 79
FIGURE 40: PREPROCESSING TIME IN SECONDS VS. NUMBER OF VERTICES. 81
FIGURE 41: AN EXAMPLE OF MESH FOLDING 83
FIGURE 42: A PLOT OF VERTICES VS. TRIANGLES FOR THE SAMPLE MODELS 84
FIGURE 43: ENFORCING A BALANCED TREE CAN SACRIFICE GOODNESS OF FIT 87
FIGURE 44: HOPPE’S DEVIATION SPACE IN CROSS-SECTION 90

CHAPTER 1

INTRODUCTION

1.1. Polygons in Computer Graphics

Computer graphics is a science of simulation. Its practitioners attempt to create realistic images by simulating the optics and physics of a virtual world, or *model*, defined within the computer. In general, the more accurate the simulation and the more precisely defined the model, the more time and memory will be required to create such images. For example, the process called *ray tracing* explicitly simulates the optics of a scene using the particle approximation of light, tracing light paths in reverse from the eye into the scene and ultimately to the light sources. Ray tracing processes commonly run for minutes or hours, but can generate some very realistic-looking images. This realism, however, is fundamentally static. Changes in scene or viewpoint can only be reflected in the resulting images by re-running the entire ray tracing process.

Interactive computer graphics deals with *dynamic realism*. Images must not only look correct, but also move correctly in response to interactive input. This amounts to an additional constraint: images must be generated quickly, preferably twenty or more times per second. The careful procedures just described must be rejected in favor of faster methods. Interactive graphics, then, is a science of approximation. Its practitioners are concerned less with accurately simulating the physics of a scene than with finding better and faster ways to approximate the results of such a simulation. The techniques of texture mapping, Gouraud shading, and the Phong lighting model are all approximations essential to modern interactive graphics. But perhaps the most fundamental approximation underlying the field of interactive graphics is the use of polygons to model three-dimensional surfaces.

Polygonal models currently dominate interactive computer graphics. This is chiefly due to their mathematical simplicity: by providing a piecewise linear approximation to shape, polygonal models lend themselves to simple, regular rendering algorithms in which the visibility and colors of most pixels are determined by interpolating across the polygon's surface. Such algorithms embed well in hardware, which has in turn led to widely available polygon rendering accelerators for every platform. In addition, polygons serve as a sort of lowest common denominator for computer models. Almost any surface representation may be converted with arbitrary accuracy to a polygonal mesh, including splines, implicit mathematical surfaces, and volumetric isosurfaces. For these and other reasons, polygonal models remain the most common representation for interactive rendering of medical, scientific, and computer-aided design (CAD) datasets.

The polygonal complexity of such models often exceeds the ability of graphics hardware to render them interactively. Three basic approaches are used to alleviate this problem:

- Augmenting the raw polygonal data to convey more visual detail per polygon, so that fewer polygons can represent the model. For example, Gouraud shading can use smooth interpolation of color across a polygon to create the illusion of a curved surface across the flat polygon. Texture mapping takes this notion further, stretching an actual image across the polygonal face. A well-chosen texture map can provide a stunning increase in realism.

- Large regions that are completely occluded from the current viewpoint can be quickly culled away using information about the model. The visibility processing approaches described by John Airey and Seth Teller are excellent examples, using the structure of architectural models to divide the world into cells that are mostly mutually invisible [Airey 90, Teller 91]. Ned Greene, Hansong Zhang, and others have tackled the more difficult problem of general polygonal environments [Greene 93, Zhang 97].

- Simplifying the polygonal geometry of small or distant portions of the model to reduce the rendering cost without a significant loss in the visual content of the scene. Such methods are known collectively as *polygonal simplification* algorithms. This thesis describes a novel approach to the polygonal simplification problem.

1.2. Polygonal Simplification

The goal of polygonal simplification in rendering is to reduce the complexity of a polygonal model to a level that can be rendered at interactive rates. The key observation that makes this possible is that much of the complexity is unnecessary in a typical model *for a given viewpoint*. An elaborate table lamp, for example, may require thousands of polygons to faithfully depict its every curve and bevel. However, if the table lamp is just one object in an architectural walkthrough of an entire house, and the user views the lamp from the opposite side of the house, the lamp will only occupy a few pixels of the final image. In this case, rendering those thousands of polygons is a waste of time; a crude version of the lamp comprising fifty polygons would suffice. The same is true, to varying degrees, of every object in the house. Only those portions of the model quite near the viewer need to be rendered in their original full detail. For the rest of the scene, a simplified approximation of the original geometry can be used to bring the polygon count down to manageable levels.

Figure 1 summarizes the traditional approach to polygonal simplification. In an offline preprocessing step, multiple versions of each object are created at progressively coarser *levels of detail*, or LODs. Once the LODs have been created and stored for every object in the model, complexity can be regulated at run-time by choosing for each frame which LOD will represent each object. As an object grows more and more distant, the system switches to coarser and coarser LODs.

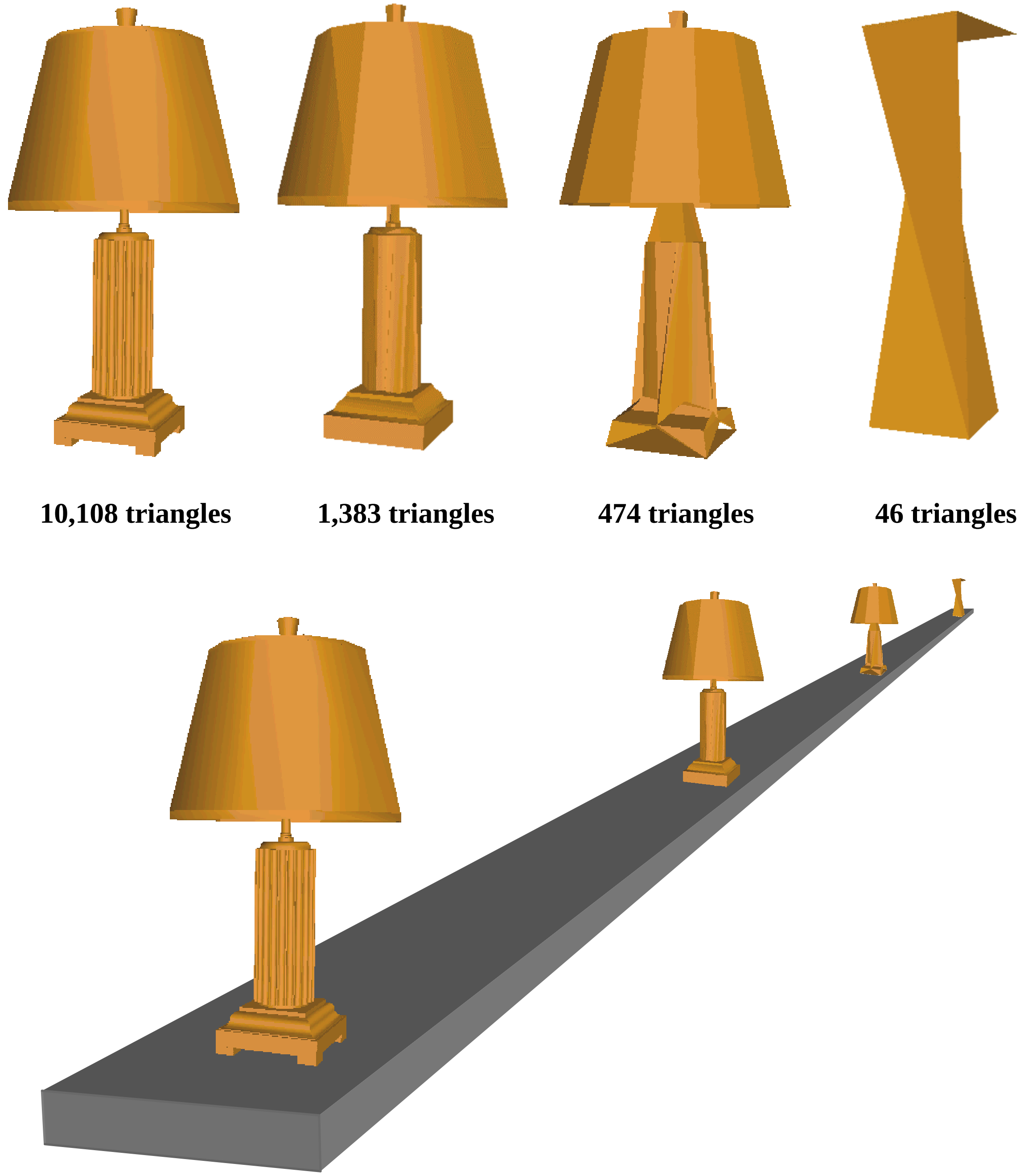
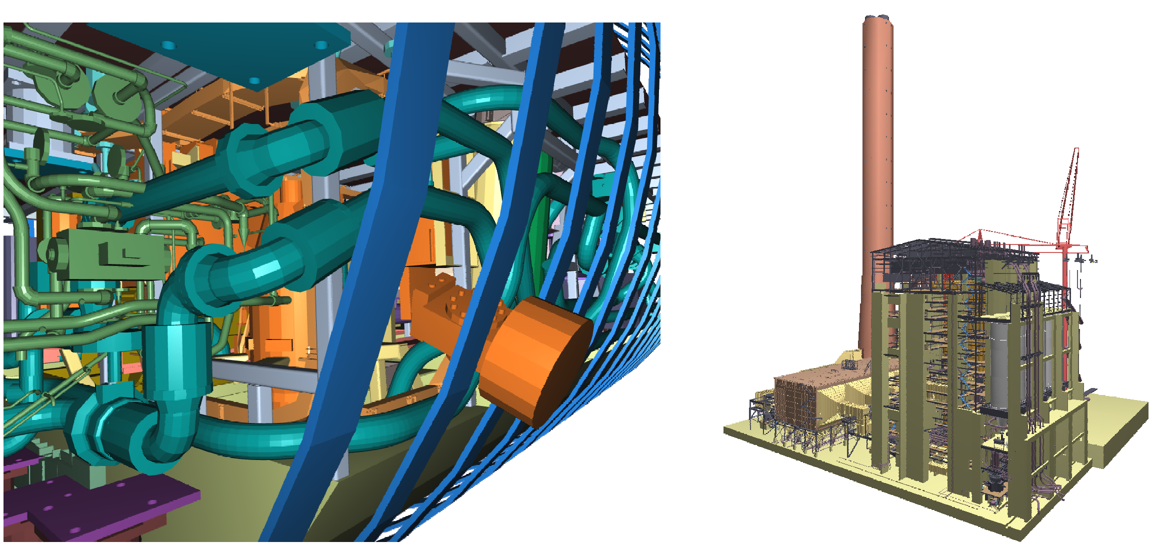


Figure 1: The traditional approach to polygonal simplification creates several *levels of detail*, or *LODs*, for each object. Which LOD is drawn depends on the object's distance.

1.3. Motivation For a New Approach

This LOD-based approach to polygonal simplification has been in use since the flight simulator systems of the 1970's [Cosman 81]. Levels of detail were originally created by hand, but in the 1990s there began to appear in the literature a flurry of algorithms for automatically generating LODs from a detailed original object. The field of polygonal simplification, surveyed in Chapter 2, now appears to be approaching maturity. Many excellent algorithms have been published, each with its particular advantages. Some approaches are best suited to curved, organic forms, while others excel at preserving the sharp corners and flat faces of mechanical objects. Some methods make global guarantees about the topology of the simplified object, while others specialize in quickly reducing the redundant geometry often found on volumetric isosurfaces.

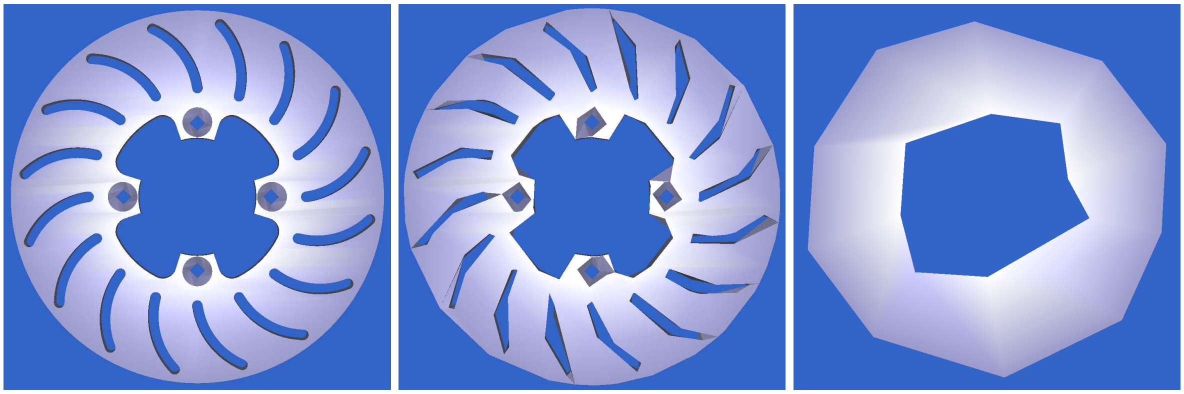
The algorithm described in this thesis was conceived for complex, large-scale CAD databases, a class of models for which earlier simplification methods often prove inadequate. Several features of such models make simplification a difficult task. To begin with, large-scale CAD models are by their nature handcrafted, often by many different CAD operators working independently on subsections of the larger model. As a result, the models tend to be messy, often containing topological degeneracies of every sort. The sheer complexity of these models can also be daunting. Massive models consisting of thousands of parts and millions of polygons are not uncommon. Finally, such massive CAD models often represent scenes rather than objects, that is, physically large environments through which a viewer would walk or fly rather than small intricate objects to be inspected from various angles. Figure 2 shows two such models.



(a) Submarine auxiliary machine room (500,000 polygons)
Courtesy Electric Boat Division, General Dynamics Corp.

(b) Coal-fired power plant (13,000,000 polygons)
Courtesy ABB Engineering & Jim Close

Figure 2: Two massive CAD models.



(a) 4,736 triangles, 21 holes *(b) 1,006 triangles, 21 holes* *(c) 46 triangles, 1 hole*

Figure 3: Preserving genus limits drastic simplification. The original model of a brake rotor (a) is shown simplified with a topology-preserving algorithm (b) and a topology-modifying algorithm (c). Rotor model courtesy Alpha_1 Project, University of Utah.

1.3.1. Limitations of Traditional LOD

Three factors make such models particularly difficult to simplify and render using traditional LOD-based simplification algorithms. First, most traditional algorithms are quite slow, taking minutes or even hours to create LODs for a complex object. For models containing thousands of parts and millions of polygons, creating LODs becomes a batch process that can take hours or days to complete. Depending on the application, such long preprocessing times may be a slight inconvenience or a fundamental handicap. In a design-review setting, for instance, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours prevent the rapid turnaround desirable in this scenario.

Second, most traditional LOD algorithms both require and preserve manifold topology in the polygonal mesh. Requiring clean mesh topology hinders the usefulness of such algorithms on handcrafted CAD models, which as noted above often contain topological degeneracies. Preserving mesh topology implies preserving the overall genus, which as Figure 3 shows, can limit the amount of simplification possible.

Third, traditional algorithms all work on a per-object basis, limiting the amount of drastic simplification possible. The problem boils down to an implicit assumption in traditional LOD about the *size* of objects in the scene. This limitation is perhaps best illustrated anecdotally, with two examples.

1.3.2. Drastic Simplification: the Problem with Large Objects

IBM's 3-D Interaction Accelerator, or 3DIX for short, is an excellent product that uses the Rossignac-Borrel algorithm [Rossignac 92] to enable interactive flythroughs of complex 3-D models. Like other traditional systems, 3DIX generates exactly one sequence of LODs for every object in the model. IBM, as a major financial and technological sponsor of the Atlanta Olympic Games, gave the research team that developed 3DIX an opportunity to demonstrate their product publicly, shortly before the Games commenced in the summer of 1996. The subject was an AutoCAD model of the Olympic Stadium in Atlanta. Although the model was well over a million polygons, 3DIX had already been successfully used on even larger datasets, so the 3DIX team did not anticipate any problems. When they actually received the model, however, they were horrified to learn that all the seats in all the bleachers comprised one giant stadium-sized object! The problem, of course, was that creating a sequence of LODs for such a huge object is useless. The viewer will always be near some portion of the bleachers and quite distant from other portions. Using a high level-of-detail would mean high fidelity but low frame rates and jerky motion; using a low level-of-detail would provide smooth motion but terrible fidelity for the nearby bleachers. The 3DIX demo did take place, and it was indeed impressive, but only after an AutoCAD expert had spent much of the intervening two weeks subdividing the bleachers into small chunks.

The solution proposed here for the Problem with Large Objects involves a *dynamic simplification* algorithm that incrementally changes the level of detail of a model at run time. Dynamic simplification in turn enables *view-dependent simplification*, in which the level of detail is varied across the model according to interactive viewing parameters such as the view position and orientation. A large and complex object, such as the stadium bleachers, presents no problem to a view-dependent simplification algorithm because only the portions of the object near the user need to be rendered in high detail. The bulk of the stadium stands can still be simplified drastically, rescuing frame rate while preserving fidelity.

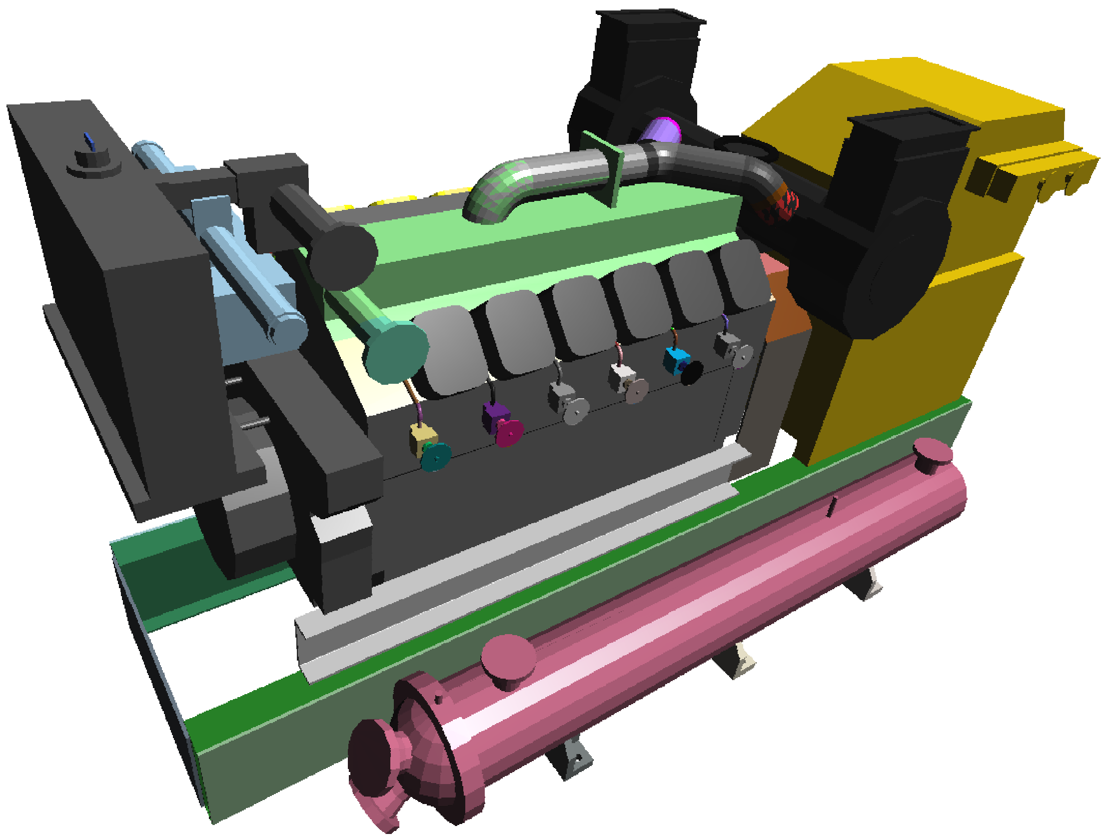


Figure 4: A diesel engine model with over 200 parts.
Courtesy Electric Boat Division, General Dynamics Corp.

1.3.3. Drastic Simplification: the Problem with Small Objects

The diesel engine shown in Figure 4 demonstrates another difficulty with the traditional per-object approach. This engine model, from the submarine Auxiliary Machine Room dataset, contains over two hundred small parts. Assume an excellent LOD algorithm, which at the lowest level can with good fidelity reduce each of these parts to a single cube. The entire assembly still requires over 2,400 triangles to render! From a great distance (say the other end of the submarine) the whole diesel engine may cover only a few pixels on the viewer's screen. In this situation a single, fifty-polygon, roughly engine-shaped block makes a better approximation than two hundred small cubes.

The solution proposed here for the Problem with Small Objects involves a *global simplification* algorithm that treats the entire scene rather than individual objects within the scene. With knowledge about the entire scene, the algorithm can decide at an appropriate level of detail to start combining the various parts of the diesel engine. At a low enough level of detail, the whole engine (and perhaps nearby portions of the walls and floor) can be merged and represented by that fifty-polygon block. Note that the idea of global simplification dovetails nicely with a dynamic, view-dependent approach. Since view-dependence allows different portions of an object to be represented at different levels of detail, the entire scene can be treated as a single all-inclusive object. The result is a global simplification algorithm that can automatically merge objects within the scene as needed for drastic simplification.

This point is important and bears repeating: for drastic simplification using traditional level-of-detail based algorithms, large objects must be subdivided and small objects must be combined. As the experience of the 3DIX team shows, doing this manually can mean a great deal of work. A global, dynamic, view-dependent algorithm is better suited to drastic simplification than the traditional approach of separate LODs for each object.

1.4. Hierarchical Dynamic Simplification

These considerations led to the new approach, called *hierarchical dynamic simplification*, presented in this thesis. Hierarchical dynamic simplification (HDS) is a framework for polygonal simplification via *vertex merging*. This operation, in which several polygon vertices are collapsed together into a single vertex, provides the fundamental mechanism for removing polygonal detail. When two vertices sharing an edge of a triangle are merged, that triangle becomes redundant and may be removed. Note the use of *triangle* rather than *polygon*. The constant memory requirements and guaranteed planarity of triangles make them preferable to generic polygons, and like most simplification algorithms, HDS assumes that polygonal models have been fully triangulated.

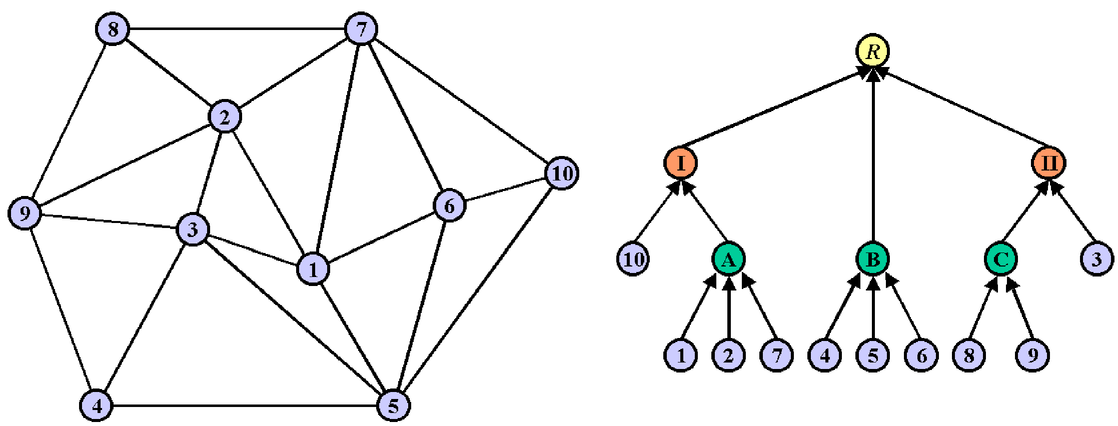


Figure 5: A simple mesh and associated vertex tree. Vertices are clustered hierarchically to root *R*.

As a polygonal simplification algorithm, HDS has some novel features. HDS is global: whereas traditional LOD algorithms represent the scene as a collection of objects, each at several levels of detail, HDS uses a single large data structure that constitutes the entire model. This structure is the *vertex tree*, a hierarchy of vertex merge operations that encodes a continuum of possible levels of detail across the whole model. Figure 5 shows a simple two-dimensional example model and its associated vertex tree. Applying a node's vertex merge operation collapses all of the vertices within the node together to a single *representative vertex*. Triangles whose corners have been collapsed together become redundant and can be eliminated, decreasing the total triangle count. This is called *folding* the node. Likewise, a node may be *unfolded* by splitting its representative vertex into the node's constituent vertices. Triangles filtered out when the node was collapsed become visible again when the node is expanded, increasing the triangle count. Figure 6 illustrates the result of folding and unfolding different nodes in the vertex tree. Chapter 3 describes the vertex tree in detail, and Chapter 6 discusses ways to construct it.

Note that the vertex tree contains information only about the vertices and triangles of the model. The algorithm makes no assumptions about the connectivity of those primitives. In particular, the triangles are not assumed to form a manifold mesh or approximate a smooth surface. This is another uncommon feature of the HDS framework: because the simplification operates on the level of triangles and vertices rather than meshes and surfaces, manifold topology is not required and need not be preserved.

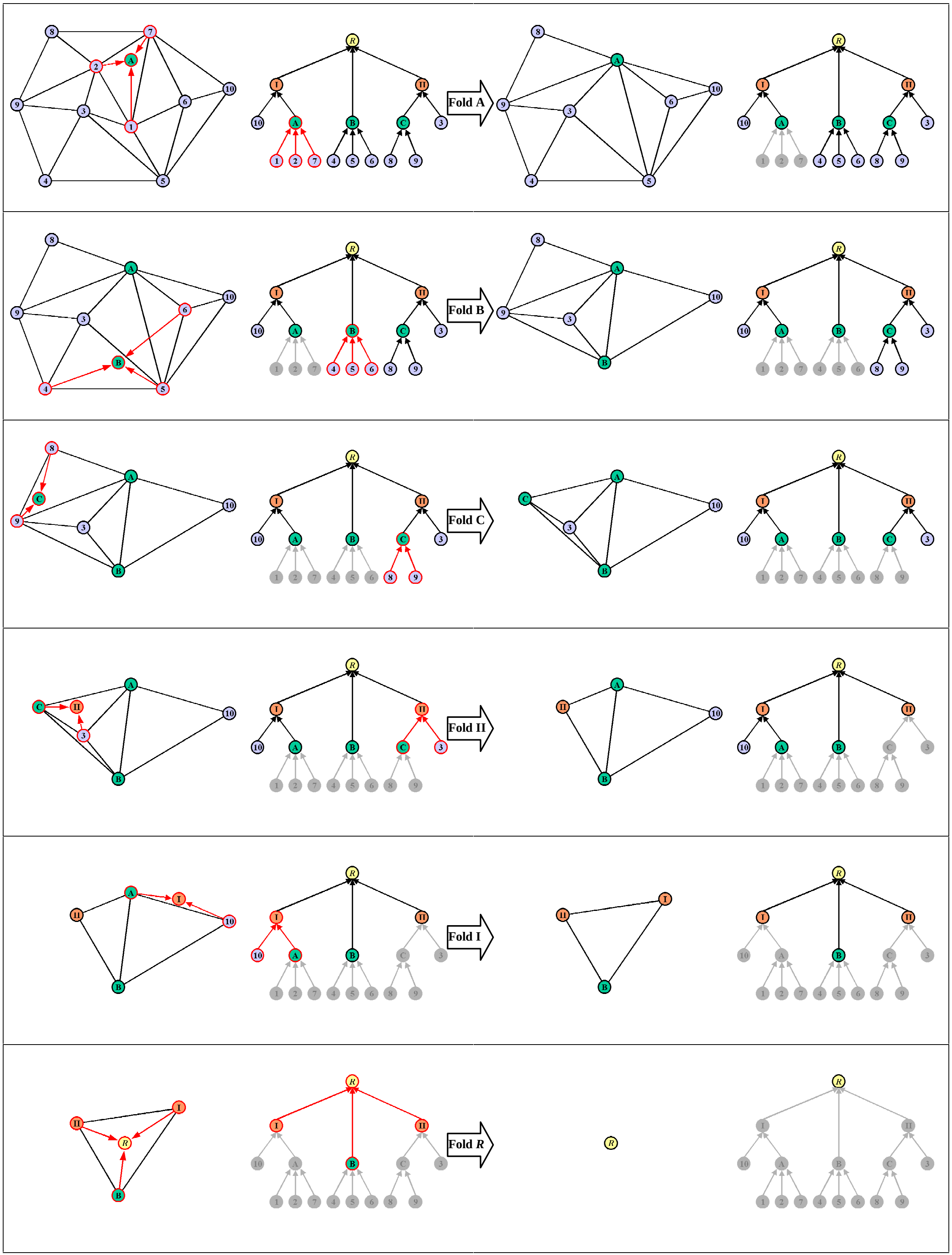


Figure 6: A sequence of fold operations. Folding each node removes some triangles from the scene, reducing the scene to a single triangle and finally to the root node *R*.

Finally, the entire system is dynamic and view-dependent. Nodes to be folded or unfolded are continually chosen at run-time based on their current projected screen size. In the simplest mode, the user sets a screenspace-size threshold, say two pixels, before flying the viewpoint interactively around the model. The screenspace extent of each node is monitored: as the viewpoint shifts, certain nodes in the vertex tree will shrink in apparent size, falling below the two-pixel threshold. These nodes will be folded into their parent nodes and the now-redundant triangles removed from the scene. Other nodes will increase in apparent size and will be unfolded into their constituent child nodes, introducing new vertices and new triangles into the display list. The user may adjust the screenspace size threshold throughout a viewing session for interactive control over the degree of simplification. Since nodes are folded and unfolded each frame, efficient methods for finding, adding, and removing affected triangles are crucial and form the subject of Chapter 3.

1.5. HDS as a Framework

It is important to emphasize that HDS is not a single algorithm so much as a general framework from which algorithms can be constructed. The only essential invariants of the HDS framework are the vertex tree and its associated methods supporting dynamic view-dependent simplification. Decisions such as how the vertex tree is constructed, which view-dependent criteria are used, and how the error metric guides simplification all flesh out the framework into a specific algorithm. The chart below illustrates this flexibility, showing some of the ways these parameters can be varied at run time in the current system to customize a dynamic view-dependent algorithm.

	Dynamic, View-dependent Simplification Algorithm					
Choices to make	**Vertex Tree Construction**		**Use of Error Metric**		**View-Dependent Criteria**	
Possible Alternatives	**Tight-octree based spatial subdivision**	**Tight-octree/ edge-collapse hybrid**	**Fidelity-based simplification**	**Polygon-budget simplification**	**Screen-space error threshold**	**Silhouette preservation**
See Section:	6.1	6.3	2.1.2	2.1.2, 4.3	4.1	4.2

1.6. Thesis Statement

A global, dynamic, and view-dependent approach to polygonal simplification can provide a powerful, general framework for visualizing polygonal environments, even those too complex or degenerate for other simplification schemes.

1.7. Outline of Argument

The first clause of the thesis statement refers to a “global, dynamic, and view-dependent approach to polygonal simplification.” The bulk of this dissertation is devoted to describing the design and implementation of such an approach. Sections 1.3.2 and 1.3.3 have already addressed “global, dynamic, and view-dependent,” explaining how each attribute enables drastic simplification despite very large and small objects. Together they provide a “powerful, general framework” for rendering polygonal scenes. Chapter 7 demonstrates the power and generality of HDS anecdotally: HDS enables interactive walkthroughs of models of very high complexity, and successfully runs without modification on models that can crash other algorithms. The models tested include some extremely complex real-world maritime and aerospace CAD datasets.

The rest of this dissertation is here outlined on a chapter-by-chapter basis. Chapter 2 surveys the previous work in traditional LOD-based polygonal simplification. Chapter 3 then turns to the new approach, presenting the data structures and methods that enable dynamic simplification. Chapter 4 describes how to use those structures and methods for view-dependent simplification, and Chapter 5 describes a set of optimizations that allow the algorithm to run in real time even on very complex models. Chapter 6 discusses how the vertex tree is constructed and how this step might take advantage of other polygonal simplification research. Chapter 7 presents results of the algorithm in action on several different models, describes some artifacts inherent in the simplification process (and how to avoid them), and analyzes the space and time complexity of the algorithm. Chapter 8 returns to the literature, describing some recent related work and commenting on issues and trends in the field of polygonal simplification. Finally, Chapter 9 summarizes the contributions of this thesis and suggests some avenues for future research.

CHAPTER 2

PREVIOUS WORK: A SURVEY OF TRADITIONAL LEVEL-OF-DETAIL ALGORITHMS

Polygonal simplification is at once a very current and a very old topic in computer graphics. As early as 1976 James Clark described the benefits of representing objects within a scene at several resolutions, and flight simulators have long used hand-crafted multi-resolution models of airplanes to guarantee a constant frame rate [Clark 76, Cosman 81]. With the mainstream debut of 3-D CAD and workstation-based computer graphics, recent years have seen a flurry of research into generating such multi-resolution representations of objects automatically by simplifying the polygonal geometry of the object. This chapter surveys the field of polygonal simplification, describing some historically important work as well as relating the current state of the art in traditional LOD-based polygonal simplification. The goal of this chapter is not only to provide a backdrop for the dissertation but also to identify some major issues and trends in the field to date.

Note that terrains, or tessellated height fields, are a special category of polygonal models. The regularity and two-dimensional nature of these models simplify some aspects of the simplification problem; most of the problems facing researchers in polygonal simplification have been solved a year or two earlier for the restricted domain of terrain datasets. At the risk of injustice to some elegant work on terrains, this survey focuses on solutions that apply to the more general realm of polygonal meshes.

A bewildering variety of simplification techniques have appeared in the recent literature; the next section attempts to classify the important similarities and differences among these techniques. A catalog of nine published algorithms follows, briefly describing each approach and placing it into this taxonomy.

2.1. Taxonomy

The various simplification approaches described in the computer graphics literature of the last five years can be categorized along many axes. Some algorithms iteratively remove polygons while others collapse vertices, some algorithms preserve topology while others ignore it, and so on. This section essays a useful taxonomy for comparing the multifarious published simplification algorithms by enumerating three important areas in which existing solutions differ or resemble each other.

2.1.1. Mechanism of Polygon Elision

Nearly every simplification technique in the literature uses some variation or combination of four basic polygon elision mechanisms: sampling, adaptive subdivision, decimation, and vertex merging.

- *Sampling* schemes begin by sampling the geometry of the initial model. These samples can be points on the 2-D manifold surfaces in the model or voxels in a 3-D grid superimposed upon the model. The algorithm then tries to create a polygonal simplification that closely matches the sampled data. Varying the number of samples regulates the accuracy of the created simplification.

- *Adaptive subdivision* approaches create a very simple polygonal approximation called the *base model*. The base model consists of triangles or squares, shapes that lend themselves to recursive subdivision. This process of subdivision is applied until the resulting surface lies within some user-specified threshold of the original surface. Conceptually simple, adaptive subdivision methods suffer two disadvantages. First, creating the base model involves the very problem of polygonal simplification that the algorithm is attempting to solve. For this reason adaptive subdivision approaches have been more popular for the specialized case of terrains, whose base model is typically just a rectangle. Second, a recursive subdivision of the base model may not be able to capture the exact geometry of the original model, especially around sharp corners and creases in the mesh [Hoppe 96].

- *Decimation* techniques iteratively remove vertices or faces from the mesh, retriangulating the resulting hole after each step. This process continues until it reaches a user-specified degree of simplification. If decimation algorithms do not permit a vertex or face removal that will change the local topology of the mesh, the decimation process may be unable to effect high degrees of simplification.

- *Vertex merging* schemes operate by merging two or more vertices of a triangulated model together into a single vertex, which can in turn be merged with other vertices. Merging two corners of a triangle makes that triangle degenerate. Such triangles can then be eliminated, decreasing the total polygon count. Vertex merging approaches do not necessarily require manifold topology, though some algorithms use a limited vertex merge called an *edge collapse*, in which only the two vertices sharing an edge are collapsed in each operation. These algorithms generally assume manifold topology implicitly.

2.1.2. Use of Error Metric

Simplification methods can be characterized by how they use an error metric to regulate the quality of the simplification. A surprising number of algorithms use no metric at all, but simply require the user to run the algorithm with different settings and explicitly select appropriate LOD switching distances. For large databases, however, this degree of user intervention is simply not practical. Those algorithms that utilize an error metric to guide simplification fall into two categories:

- *Fidelity-based simplification* techniques allow the user to specify the desired fidelity of the simplification in some form, then attempt to minimize the number of polygons, subject to that fidelity constraint.

- *Polygon-budget simplification* systems attempt to maximize the fidelity of the simplified model without exceeding a specified polygon budget.

For example, adaptive subdivision algorithms lend themselves nicely to fidelity-based simplification, simply subdividing the base model until the fidelity requirement is met. Polygon-budget simplification is a natural fit for decimation techniques, which are designed to remove vertices or faces one at a time and merely need to halt upon reaching the target number of polygons. As mentioned above, however, topology constraints often prevent decimation algorithms from reducing the polygon count below a certain level. To be most useful, a simplification algorithm should be capable of either fidelity-based or polygon-budget operation. Fidelity-based approaches are crucial for generating accurate images, whereas polygon-budget approaches are important for time-critical rendering. The user may well require both of these possibilities in the same system.

2.1.3. Preservation of Topology

In the context of polygonal simplification, *topology* refers to the structure of the connected polygonal mesh. The *local topology* of a face, edge, or vertex refers to the connectivity of that feature's immediate neighborhood. The mesh forms a *2-D manifold* if the local topology is everywhere homeomorphic to a disc, that is, if the neighborhood of every feature consists of a connected ring of triangles forming a single surface. Every edge in a mesh displaying manifold topology is shared by exactly two triangles, and every triangle has exactly three neighboring triangles, all distinct (a *2-D manifold with boundary* allows the local neighborhoods to be homeomorphic to a half-disc, which means some edges can belong to only one triangle). A *topology-preserving* simplification algorithm preserves manifold connectivity. Such algorithms do not close holes in the mesh, and they therefore preserve the genus of the simplified surface. *Global topology* refers to the connectivity of the entire surface. A simplification algorithm preserves global topology if it preserves local topology and does not create self-intersections within the simplified object [Erikson 96]. A self-intersection, as the name implies, occurs when two non-adjacent faces intersect each other.

Many real-world CAD models contain objects that violate manifold local topology, global topology, or both. Since interactive visualization of CAD databases is a primary application of polygonal simplification, the behavior of the various approaches when encountering such models is an important characteristic. Simplification algorithms can be separated into two camps:

- *Topology-preserving algorithms* preserve the genus of the simplified object, so no holes will appear or disappear during simplification. The opacity of the object seen from any distance thus tends to remain roughly constant. This constraint limits the simplification possible, however, since objects of high genus cannot be simplified below a certain number of polygons without closing holes in the model. Moreover, a topology-preserving approach requires a mesh with valid topology to begin with. Some algorithms, such as [Schroeder 92], are *topology-tolerant*: they ignore regions in the mesh with invalid local topology, leaving those regions unsimplified. Other algorithms faced with such regions may simply crash.

- *Topology-modifying algorithms* do not necessarily preserve local or global topology. The algorithms can therefore close up holes in the model as simplification progresses, permitting drastic simplification beyond the scope of topology-preserving schemes. This drastic simplification often comes at the price of poor visual fidelity, however, and distracting popping artifacts as holes appear and disappear from one LOD to the next. Some topology-modifying algorithms do not require valid topology in the initial mesh, which greatly increases their utility in real-world CAD applications. Some topology-modifying algorithms attempt to regulate the change in topology, but most are *topology-insensitive*, paying no heed to the initial mesh connectivity at all.

2.2. Catalog of Important Papers

The intent of this section is not to provide an exhaustive list of work in the field of polygonal simplification, nor to select the “best” published papers, but rather to briefly describe a few important algorithms that span the taxonomy presented above. Most of the papers chosen represent influential advances in the field; a few provide more careful treatment of existing ideas.

Table 1 summarizes the catalog. Each algorithm is broken down according to which mechanism or combination of mechanisms it uses, whether it supports fidelity-based simplification or polygon-budget simplification, and whether the algorithm preserves or modifies topology. The final column indicates whether the algorithm is *topology-tolerant*, that is, whether the presence of non-manifold mesh regions will catastrophically affect the algorithm. An asterix (*) under fidelity-based or polygon-budget simplification indicates that the algorithm can be easily extended to support that use of the error metric, even though the algorithm's original publication does not mention it.

In addition to the algorithms presented here, a few particularly recent and relevant papers are discussed in Chapter 8: Recent Related Work.

Algorithm	**Reference**	**Mechanism**				**Use of Error Metric**		**Topology**		
		Sampling	Adaptive Subdivision	Decimation	Vertex Merging	Fidelity-based	Polygon-budget	Preserving	Modifying	Tolerant
Multi-Resolution 3D Approximations…	Rossignac 92				X	*			X	X
Decimation of Triangle Meshes	Schroeder 92			X		X	*	X		X
Re-tiling Polygonal Surfaces	Turk 92	X		X			X	X		X
Mesh Optimization	Hoppe 93	X		X	X		X	X		
Multiresolution Analysis of Arbitrary Meshes	Eck 95		X			X		X		
Voxel-Based Object Simplification	He 95	X				X			X	
Simplification Envelopes	Cohen 96			X		X		X		
Progressive Meshes	Hoppe 96				X		X	X		
Model Simplification Using Vertex Clustering	Low 97				X	X			X	X

Table 1: Nine simplification algorithms classified by mechanism, use of error metric, and treatment of topology. An asterix means that the algorithm could easily be extended to include the specified use of error metric.

2.2.1. Rossignac and Borrel

Multi-Resolution 3D Approximations for Rendering Complex Scenes (1992)

This vertex-merging algorithm by Jarek Rossignac and Paul Borrel is one of the few schemes that neither requires nor preserves valid topology. The algorithm can therefore deal robustly with degenerate models with which other approaches have little or no success. This is a tremendous advantage for simplification of handcrafted CAD databases, a notoriously messy category of models.

The algorithm begins by assigning a perceptual importance to each vertex based upon two factors. Vertices associated with large faces are considered more important than vertices associated only with small faces, and vertices of high curvature (measured by the inverse of the maximum angle between any pair of edges incident to the vertex) are considered more important than vertices of low curvature. Next a three-dimensional grid is overlaid on the model and all vertices within each cell of the grid are collapsed to a single *representative vertex* for the cell, chosen according to the importance weighting calculated in the first step. The resolution of this grid determines the quality of the resulting simplification; a coarse grid will aggressively simplify the model whereas a fine grid will perform only minimal reduction. In the process of clustering, triangles whose corners are collapsed together become degenerate and disappear.

One unique feature of the Rossignac-Borrel algorithm is the fashion in which it treats these triangles. Reasoning that a triangle with two corners collapsed is simply a line and a triangle with three corners collapsed is simply a point, the authors choose to render such triangles using the line and point primitives of the graphics hardware, filtering out redundant lines and points. Thus a simplification of a polygonal object will generally be a collection of polygons, lines, and points. The resulting simplifications are therefore more accurate from a schematic than a strictly geometric standpoint. For the purposes of drastic simplification, however, the lines and points can contribute significantly to the recognizability of the object.

In addition to its inherent robustness, the Rossignac-Borrel algorithm can be implemented very efficiently and is one of the fastest algorithms known. However, the method suffers several disadvantages. Since topology is not preserved and no explicit error bounds with respect to the surface are guaranteed, the resulting simplifications are often less pleasing visually than those of slower algorithms. In addition, the simplification is sensitive to the orientation of the clustering grid, so two identical objects at different orientations can produce quite different simplifications. Finally, the algorithm does not lend itself to either fidelity-based or polygon-budget simplification, since the only way to predict how many triangles an LOD will have using a specified grid resolution is to perform the simplification.

2.2.2. Schroeder, Zarge, and Lorenson

Decimation of Triangle Meshes (1992)

One of the first published algorithms to simplify general polygonal models, this paper coined the term "decimation" for iterative removal of vertices. Schroeder's decimation scheme is designed to operate on the output of the Marching Cubes algorithm for extracting isosurfaces from volumetric data [Lorenson 87], and is still commonly used for this purpose. Marching Cubes output is often heavily overtessellated, with coplanar regions divided into many more polygons than necessary, and Schroeder's algorithm excels at removing this redundant geometry.

The algorithm operates by making multiple passes over all the vertices in the model. During a pass, each vertex is considered for deletion. If the vertex can be removed without violating the local topology of the neighborhood, and if the resulting surface would lie within a user-specified distance of the unsimplified geometry, the vertex and all its associated triangles are deleted. This leaves a hole in the mesh, which is then retriangulated. The algorithm continues to iterate over the vertices in the model until no more vertices can be removed.

Simplifications produced by the decimation algorithm possess an interesting feature: the vertices of the simplified model are a subset of the vertices of the original model. This property is convenient for reusing normals and texture coordinates at the vertices, but it can limit the fidelity of the simplifications, since minimizing the geometric error introduced by the simplified approximation to the original surface can at times require changing the positions of the vertices [Garland 97]. The decimation algorithm is topology tolerant, accepting models with non-manifold vertices, but does not attempt to simplify those regions of the model.

2.2.3. Turk

Re-Tiling Polygonal Surfaces (1992)

Another of the first papers to address simplification of arbitrary polyhedral objects, this algorithm combines elements of the sampling and decimation mechanisms. The re-tiling algorithm works best on smoothly curved surfaces without sharp edges or discontinuities, working better for organic forms such as people or animals than for mechanical shapes such as furniture or machine parts. Re-tiling provides a form of polygon-budget simplification by allowing the user to specify the number of vertices in the simplified model, but it is not obvious how to modify the algorithm to provide a fidelity metric.

The algorithm begins by randomly distributing the user-specified number of vertices over the surface of the model. The algorithm then simulates repulsion forces between the vertices, allowing nearby vertices to repel each other. Since the vertices are constrained to move within the surface, this repulsion tends to redistribute the randomly scattered vertices evenly across the surface. Next, the algorithm uses a method called *mutual tessellation* to construct an intermediate surface that contains both the new and original vertices. All the original vertices are removed, leaving the re-tiled surface with only the new vertices. Finally, a local re-triangulation is applied to improve the aspect ratio of the resulting triangles.

Among the contributions of this paper was the introduction of a method to interpolate smoothly between different levels of detail, a process which Hughes Hoppe calls geomorphing [Hoppe 96].

2.2.4. Hoppe, DeRose, Duchamp, McDonald, and Stuetzle

Mesh Optimization (1993)

This paper describes a complex sampling approach, which evolved out of the authors' work on surface reconstruction of laser-scanned datasets [Hoppe 92]. Surface reconstruction is the problem of creating a three-dimensional mesh from a collection of sample points. Mesh optimization, as the name suggests, treats simplification as an optimization problem. The number of vertices in the simplification and its deviation from the original are explicitly modeled as an energy function to be minimized.

The algorithm begins by sampling the mesh, taking a number of randomly placed samples in addition to the vertices of the original mesh. These sample points are maintained as the algorithm modifies the mesh; the distance moved by the sample points during the algorithm is used to measure deviation from the original surface. Next a random edge of the mesh is picked and one of three operations attempted at random: edge collapse, edge split, or edge swap. An inner loop then adjusts the positions of vertices to minimize the energy function for the next configuration. If the overall energy is not reduced or the topology is violated, the randomly selected edge operation is undone. Another random edge is picked and the process repeats, iterating until repeated attempts suggest that the energy function has reached a local minimum.

The careful simplification performed by the mesh optimization algorithm produces models of very high fidelity. The algorithm seems to be especially well suited for mechanical CAD models, capturing sharp features very nicely. Though topology is preserved, with the consequent limits on simplification, mesh optimization appears excellent at simplifying right up to those limits. Unfortunately, the algorithm is somewhat slow; for example, the authors report a simplification time of 47 minutes of one 18,272-polygon object. Moreover, the mesh optimization algorithm seems complex enough to make implementation a daunting task, though fortunately Hughes Hoppe has made his code for the algorithm available at *http://www.research.microsoft.com/~hoppe/Recon.940503b.tar.gz*.

2.2.5. Eck, DeRose, Duchamp, Hoppe, Lounsbery, and Stuetzle

Multiresolution Analysis of Arbitrary Meshes (1995)

This adaptive subdivision algorithm uses a compact wavelet representation to guide the recursive subdivision process. By adding or subtracting wavelet coefficients the algorithm can smoothly interpolate between levels of detail. The algorithm provides fidelity-based simplification by using enough wavelet coefficients to guarantee that the simplified surface lies within a user-specified distance of the original model.

A chief contribution of this paper is a method for finding a simple base mesh that exhibits *subdivision connectivity*, which means that the original mesh may be recovered by recursive subdivision. As mentioned above, finding a base mesh is simple for terrain datasets but difficult for general polygonal models of arbitrary topology. Eck's algorithm creates the base mesh by growing Voronoi-like regions across the triangles of the original surface. When these regions can grow no more, a Delauney-like triangulation is formed from the Voronoi sites, and the base mesh is formed in turn from the triangulation.

This algorithm possesses the disadvantages of strict topology-preserving approaches: manifold topology is absolutely required in the input model, and the shape and genus of the original object limit the potential for drastic simplification. The fidelity of the resulting simplifications is quite high for smooth organic forms, but the algorithm is fundamentally a low-pass filtering approach and has difficulty capturing sharp features in the original model unless the features happen to fall along a division in the base mesh [Hoppe 96].

2.2.6. He, Hong, Kaufman, Varshney, and Wang

Voxel-Based Object Simplification (1995)

Topology-preserving algorithms must retain the genus of the original object, which often limits their ability to perform drastic simplification. Topology-insensitive approaches such as the Rossignac-Borrel algorithm do not suffer from these constraints, but reduce the topology of their models in a haphazard and unpredictable fashion. Voxel-based object simplification is an intriguing attempt to simplify topology in a gradual and controlled manner using the robust and well-understood theory of signal processing.

The algorithm begins by creating a volumetric representation of the model, superimposing a three-dimensional grid of voxels over the polygonal geometry. Each voxel is assigned a value of 1 or 0, according to whether the sample point of that voxel lies inside or outside the object. Next the algorithm applies a low-pass filter and resamples the volume. The result is another volumetric representation of the object with lower resolution. Sampling theory guarantees that small, high-frequency features will be eliminated in the low-pass filtered volume. The Marching Cubes algorithm [Lorenson 87] is applied to this volume to generate a simplified polygonal model. Since Marching Cubes can create redundant geometry, a standard topology-preserving algorithm is required as a postprocess.

Unfortunately, high-frequency details such as sharp edges and squared-off corners seem to contribute greatly to the perception of shape. As a result, the voxel-based simplification algorithm performs poorly on models with such features. This greatly restricts its usefulness on mechanical CAD models. Moreover, the algorithm as presented in the paper is not topology-tolerant, since deciding whether sample points lie inside or outside the object requires well-defined closed-mesh objects with manifold topology.

2.2.7. Cohen, Varshney, Manocha, Turk, Weber, Agrawal, Brooks, and Wright *Simplification Envelopes (1996)*

Simplification envelopes provide a method of guaranteeing fidelity bounds while enforcing global as well as local topology. Simplification envelopes *per se* are more of a framework than an individual algorithm, and the authors of this paper present two examples of algorithms within this framework.

The simplification envelopes of a surface consist of two *offset surfaces*, or copies of the surface offset no more than some distance ε from the original surface. The *outer envelope* is created by displacing each vertex of the original mesh along its normal by ε. Similarly, the *inner envelope* is created by displacing each vertex by $-\varepsilon$. The envelopes are not allowed to self-intersect; where the curvature would create such self-intersection, ε is locally decreased.

Once created, these envelopes can guide the simplification process. The algorithms described in the paper both take decimation approaches that iteratively remove triangles or vertices and re-triangulate the resulting holes. By keeping the simplified surface within the envelopes, these algorithms can guarantee, first, that global topology is respected, and second, that the simplified surfaces never deviate by more than ε from the original surface. The resulting simplifications tend to have very good fidelity.

Where fidelity and topology preservation are crucial, simplification envelopes are an excellent choice. The ε error bound is also an attractive feature of this approach, providing a natural means for calculating LOD switching distances. Though the algorithms presented in the paper are based on a decimation approach, a vertex-merging algorithm based on simplification envelopes is easy to imagine. However, the very strengths of simplification envelopes technique are also their weaknesses. The strict preservation of topology and the careful avoidance of self-intersections curtail the approach's capability for drastic simplification. The construction of offset surfaces also demands an orientable manifold; topological imperfections in the initial mesh can hamper or prevent simplification. Finally, the algorithms for simplification envelopes are intricate; writing a robust system based on simplification envelopes seems a substantial undertaking. The authors have made their implementation available at *http://www.cs.unc.edu/~geom/envelope.html*.

2.2.8. Hoppe

Progressive Meshes (1996)

This vertex-merging algorithm by Hughes Hoppe follows up on the mesh optimization approach. As described above, mesh optimization used the three techniques of edge collapse, edge split, and edge swap in random order to reduce an explicitly modeled energy function. The progressive meshes paper builds on the discovery that the edge collapse operation alone suffices to achieve high-quality simplification. The main contributions of the paper are the *progressive mesh*, a new representation for polygonal models based on edge collapses, and a topology-preserving simplification algorithm for generating progressive meshes.

A progressive mesh consists of a simple *base mesh*, created by a sequence of edge collapse operations, followed by a stream of *vertex split* records. A vertex split (or *vsplit*) is the dual of an edge collapse (or *ecol*). Each vsplit replaces a vertex by two edge-connected vertices, creating one additional vertex and two additional triangles. The vsplit records in a progressive mesh correspond to the edge collapse operations used to create the base mesh. Applying all of the vsplit records to the associated base mesh will recapture the original model exactly; applying a subset of the vsplit records will create an intermediate simplification. Since each vertex split creates two triangles (one for boundary edges), triangle-budget simplification is easily implemented by applying the vsplit records in order until the specified triangle budget is reached. In fact, the stream of vsplit records encodes a continuum of simplifications from the base mesh up to the original model. The vertex split and edge collapse operations are quite fast and may be applied at run-time to transition between levels of detail.

The quality of the intermediate simplifications depends entirely on the order of ecol operations used to create the base mesh. Hoppe describes a careful simplification algorithm to generate these edge collapses. The algorithm, like the mesh optimization algorithm, models fidelity explicitly as an energy function to be minimized. All edges that can be collapsed are evaluated according to their effect on this energy function and sorted into a priority queue. The energy function can then be minimized in a greedy fashion by performing the ecol operation at the head of the queue, which will most decrease the energy function. Since this may change how collapsing nearby edges will affect the energy function, those edges are re-evaluated and resorted into the queue. This process repeats until topological constraints prevent further simplification. The remaining edges and triangles comprise the base mesh, and the sequence of ecol operations performed becomes (in reverse order) the stream of vsplit operations.

Along with progressive meshes, Hoppe introduces a nice framework for handling surface attributes of a mesh during simplification. Such attributes are categorized as *discrete* attributes, associated with faces in the mesh, and *scalar* attributes, associated with corners of the faces in the mesh. Common discrete attributes include material and texture identifiers; common scalar attributes include color, normal, and texture coordinates. Hoppe also describes how to model some of these attributes in the energy function, allowing normals, color, and material identifiers to guide the simplification process.

2.2.9. Low and Tan

Model Simplification Using Vertex Clustering (1997)

Kok-Lim Low and Tiow-Seng Tan have invented a revised version of the Rossignac-Borrel algorithm. Observing that the spatial binning invoked by the 3-D grid is simply a form of vertex clustering, Low and Tan introduce a different clustering approach they call *floating-cell clustering*. In this approach the vertices are ranked by importance, and a cell of user-specified size is centered on the most important vertex. All vertices falling within the cell are collapsed to the representative vertex and degenerate triangles are filtered out as in the Rossignac-Borrel scheme. The most important remaining vertex becomes the center of the next cell, and the process is repeated. By eliminating the underlying grid, floating-cell clustering greatly reduces the sensitivity of the simplification to the position and orientation of the model. In addition, floating-cell simplification results vary less with cell size than the results of the uniform-subdivision approach.

Low and Tan also improve upon the criteria used for calculating vertex importance. Let θ be the maximum angle between all pairs of edges incident to a vertex. Though Rossignac and Borrel used $1/\theta$ to estimate the probability that the vertex lies on the silhouette, Low and Tan argue that $\cos(\theta/2)$ provides a better estimate.

Moreover, Low and Tan extend the concept of drawing degenerate triangles as lines, calculating an approximate width for those lines based on the vertices being clustered and drawing the line using the thick-line primitive present in most graphics systems. The appearance of these lines is further improved by giving the line a normal to be shaded by the standard graphics lighting computations. This normal is dynamically assigned at run-time to give the line a cylinder-like appearance.

Low and Tan address the lack of a fidelity metric in the original algorithm by noting that the clustering size used to create an LOD can be related to the maximum number of pixels each cluster can cover. This provides a rough fidelity metric, allowing the user to specify that no LOD will be used unless it clusters only vertices within *n* pixels of each other.

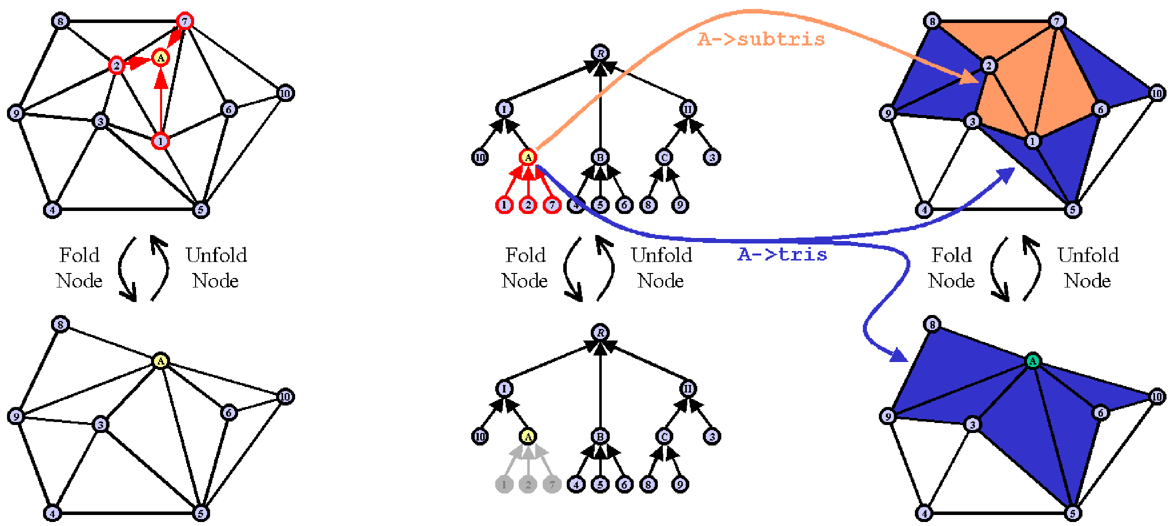
CHAPTER 3

STRUCTURES AND METHODS

3.1. The Vertex Tree

The most fundamental data structure used by HDS is the *vertex tree*. The vertex tree spans the entire model, organizing every vertex of every polygon into one global hierarchy that encodes all possible simplifications of the model. Internal nodes in the vertex tree represent the merging of multiple vertices from the original model into a single vertex. This is the representative vertex, or *repvert*. A repvert is associated with each node in the vertex tree. At the leaves of the tree, each node contains exactly one vertex from the original model; in this case, that vertex is the node's repvert. Each node in the vertex tree, then, represents a subset of the vertices in the original model; the root node represents the vertices of the entire model.

Section 1.4 defined *folding* a node as the process of merging the vertices represented by a node together into the node's repvert, and *unfolding* a node as the reverse process. Here the terms are defined more carefully. For simplicity, assume that a node's children must all be folded before the node can be folded (since those children can first be folded recursively if necessary, this assumption does not limit the power of the fold operation). This assumption reduces the process of merging all the vertices represented by a node to the process of merging the repverts of that node's children. Similarly, unfolding a node assumes that the node's parent is unfolded, and splits a node's representative vertex into just the few representative vertices of the node's folded children. Defined this way, fold and unfold are local operations that make only incremental changes to the vertex tree.



(a) Nodes 1, 2, 7 merge to form A *(b) The local vertex tree* *(c) The tris and subtris of node A*

Figure 7: Tris and subtris of a node in the vertex tree. The highlighted node A represents the clustering of nodes 1, 2, and 7.

When a node is folded, vertices are merged together and triangles change shape or degenerate into lines or points. In fact, given the definitions of the fold and unfold operations above, the same triangles are affected by the two dual operations. One set of triangles, called the node's *tris*, will change in shape as a corner shifts during fold and unfold operations. Another set of triangles, called the node's *subtris*, will disappear when the node is folded and reappear when the node is unfolded (Figure 7). This leads to the key observation behind dynamic simplification: *since a node's tris and subtris do not depend on the state of other nodes in the vertex tree, they can be computed offline and accessed very quickly at runtime.*

Unfolded nodes are labeled *active*; folded nodes are labeled *inactive*. If the entire vertex tree (excepting the root node) is labeled inactive to begin with, the definitions above ensure that after any sequence of fold and unfold operations the active nodes will constitute a cut of the vertex tree, rooted at the root node, called the *active tree*. Folded nodes with active parents are a special case; these nodes form the boundary of the active tree and are labeled *boundary* (Figure 8). Since the location of the boundary nodes determines which vertices in the original model have been collapsed together, the path of the boundary nodes across the vertex tree completely determines the current simplification. Notice that by definition, only boundary nodes can be unfolded and only active nodes whose children are all boundary nodes can be folded. The optimizations of Chapter 5 will take advantage of this fact.

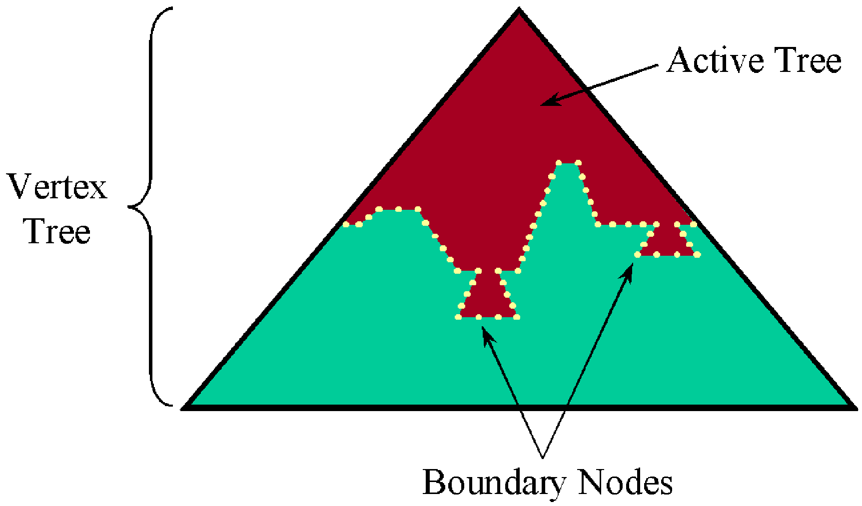


Figure 8: The vertex tree, active tree, and boundary nodes.

Each node in the vertex tree includes the basic structure described below; explanations of the individual fields follow.

```
struct Node {
   Byte          path[];
   Byte          depth;
   NodeStatus    label;
   Coord         repvert;
   Coord         center;
   float         radius;
   Tri           *tris;
   Tri           *subtris;
   Node          *parent;
   Byte          numchildren;
   Node          **children;
};
```

- **`path`**: an array of digits that specifies the path from the root of the vertex tree to the node. The *n*th element of the array specifies which branch to take at level *n*.
- **`depth`**: the depth of the node in the vertex tree[1].
- **`label`**: the node's status: *active*, *boundary*, or *inactive*.

[1] Of course, **`node->depth`** = || **`node->path`** ||, but storing the depth separately turns out to be convenient for optimizing the **`firstActiveAncestor()`** function (see Section 5.1).

- `repvert`: the coordinates of the node's representative vertex. All vertices in boundary and inactive nodes are collapsed to this vertex.
- `center`, `radius`: the center and radius of a bounding sphere containing all vertices in this node. This bounding sphere will be used for determining if the node is within the view frustum and for estimating its screenspace extent.
- `tris`: a list of triangles with exactly one corner in the node. These are the triangles whose corners must be adjusted when the node is folded or unfolded.
- `subtris`: a list of triangles with an edge spanning two children of the node, or two edges spanning three children of the node. These triangles will be filtered out if the node is folded, and re-introduced if the node is unfolded.
- `parent`: a pointer to the parent of this node in the vertex tree.
- `numchildren`, `children`: the number of, and pointers to, the children of this node.

3.2. The Active Triangle List

If the vertex tree represents every simplification of the model possible in the HDS system, the *active triangle list* represents the current simplification being rendered. Such a structure may seem unnecessary, for any simplification can be extracted from the vertex tree by a series of unfold operations starting with the root node. In practice, however, creating each frame's simplification from scratch is cumbersome. The chief purpose of the active triangle list is to take advantage of temporal coherence. Frames in an interactive viewing session typically exhibit only incremental shifts in viewpoint, so the set of visible triangles remains largely constant. In its simplest form, the active triangle list is just a sequence of those visible triangles. Expanding a node appends some triangles to the active triangle list; collapsing the node removes them. The active triangle list is maintained in the current implementation as a doubly-linked list of triangle structures, each with the following basic structure:

```
struct Tri {
    Node        *corners[3];
    Node        *proxies[3];
    Tri         *prev, *next;
};
```

The **`corners`** field represents the triangle at its highest resolution, pointing to the three nodes whose representative vertices are the original corners of the triangle. The **`corners`** of a triangle therefore remain fixed. The **`proxies`** field represents the triangle in the current simplification, pointing to the *first active ancestor* of each corner node (Figure 9). In other words, the **`proxies`** field of a triangle in the active triangle list represents the three vertices into which the **`corners`** of the triangle have been merged. The first active ancestor need not be a proper ancestor; if the corner node *N* is labeled active, the first active ancestor of *N* is just *N*. Boundary nodes are considered active for the purposes of the first active ancestor test. If the node *N* is inactive, its first active ancestor is the boundary node on the path from *N* to the root. Note that the definitions of the fold and unfold operations ensure that exactly one boundary node will exist along the path from each inactive node to the root.

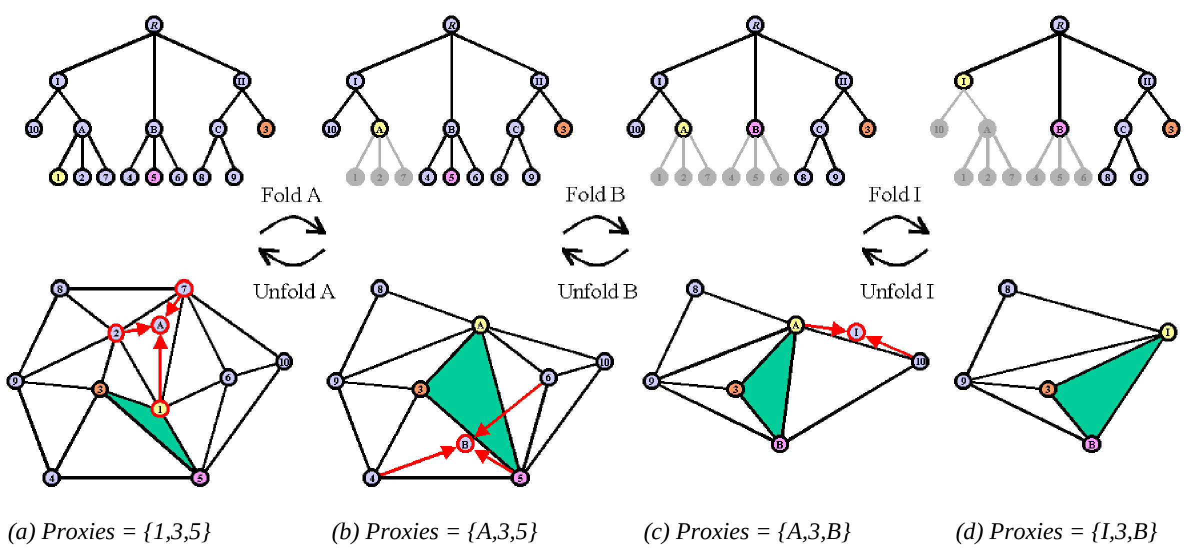


Figure 9: Corners and proxies. The proxies of a triangle are the *first active ancestors* of its corners. Here the proxies of triangle 1-3-5 shift as nodes fold and unfold.

3.3. Methods

The fundamental methods associated with the active triangle list are **`addTri()`** and **`removeTri()`**. As the names imply, these operations add or remove a triangle from the active triangle list. Using a doubly-linked list with sentinels before and after the list simplifies this process considerably:

```
// global dummy structures start & end active triangle list:
Tri *startTriList, *endTriList;

addTri (Tri *T)
   // append to end of list
   T->next = endTriList;
   T->prev = endTriList->prev;
   T->prev->next = T;

removeTri (Tri *T)
   // sentinels ensure prev & next fields won't be NULL
   T->next->prev = T->prev;
   T->prev->next = T->next;
```

Note that this scheme maintains the active triangle list entirely in place. All triangles in the model are kept in an array with their `prev` and `next` fields initialized to `NULL`. As **`addTri()`** and **`removeTri()`** are called, they thread the doubly-linked list through the array of triangles. Though simple, this approach is less than optimal with regard to memory access patterns: after a long series of **`addTri()`** and **`removeTri()`** calls, the linked list is likely to hop around the array of triangles seemingly at random. If the entire array does not fit into cache (or even into main memory), this can greatly degrade performance. Chapter 9 will discuss possible optimizations to avoid this problem.

The fundamental methods associated with a node in the vertex tree are **`foldNode()`** and **`unfoldNode()`**. These functions add or remove the subtris of the specified node from the active triangle list and update the proxies of the node's tris:

```
foldNode (Node *N)
   N->label = boundary;
   foreach child C of N
      // all children should be labeled boundary; change to inactive
         if (C->label == boundary)
               C->label = inactive;
         else
            // validity check:
               reportError();
   foreach triangle T in N->tris
      // update tri proxies
         foreach corner c of {1,2,3}
               T->proxies[c] = firstActiveAncestor(T->corners[c]);
   foreach triangle T in N->subtris
      // remove subtris from active list
         removeTri(T);

unfoldNode (Node *N)
   if (N->parent != boundary)
      // validity check:
         reportError();
   foreach child C of N
         C->label = boundary;
   N->label = active;
   foreach triangle T in N->tris
      // update tri proxies
         foreach corner c of {1,2,3}
               T->proxies[c] = firstActiveAncestor(T->corners[c]);
   foreach triangle T in N->subtris
      // add subtris to active list
         addTri(T);
```

Note that a properly debugged system should only call **`unfoldNode()`** on boundary nodes and should never call **`foldNode()`** on a node whose children are not boundary nodes, so the validity checks above can be removed for greater speed.

CHAPTER 4

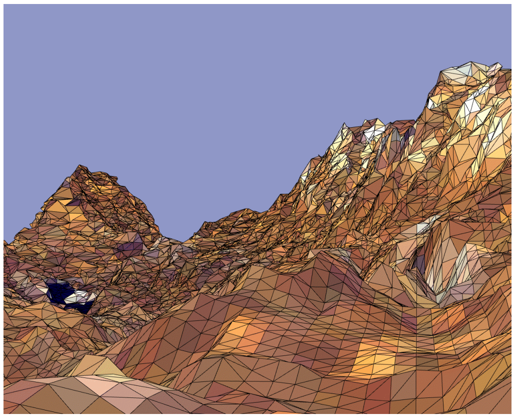
VIEW-DEPENDENT SIMPLIFICATION

The structures and methods described provide a framework for dynamic simplification, since nodes can be folded and unfolded and triangles added and removed fast enough to respond to run-time events. Any criterion for run-time simplification may be plugged into this framework; each criterion takes the form of a function to choose which nodes are folded and unfolded each frame.

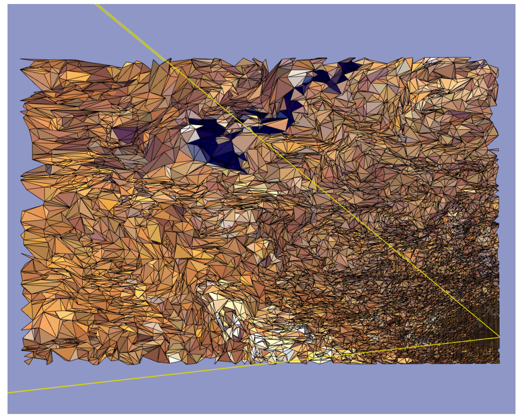
Dynamic simplification in turn provides a framework for view-dependent simplification, since these criteria may use data such as the precise location and orientation of the viewpoint. Traditional LOD, by contrast, uses only a general eye-to-object distance metric, or at best an approximate screen-space extent for the object.

The sections below describe three view-dependent criteria supported by the prototype implementation of HDS: a screenspace error threshold, a silhouette test, and a triangle budget.

4.1. Screenspace Error Threshold



(a) The displayed image, with triangles highlighted.



(b) Overhead view, with view frustum in yellow.

Figure 10: View-dependent simplification using a screenspace error threshold of 2%. The full resolution is used near the viewer and smoothly degrades as the model recedes into the distance. Terrain model courtesy Herman Towles, Sun Microsystems.

The underlying philosophy of HDS is to remove triangles that are not important to the scene. Since importance usually diminishes with size on the screen, an obvious run-time strategy is to collapse vertices that occupy a small amount of the screen. To formulate this strategy more precisely, consider a node in the vertex tree. Folding this node, which represents multiple vertices in the original model, clusters those vertices together into the node's repvert. The error introduced by collapsing the vertices can be thought of as the maximum distance a vertex can be shifted during the fold operation, which equals the length of the vector between the node's repvert and the vertex farthest from the repvert in the cluster. The extent of this vector when projected onto the screen is the *screenspace error* of the node. By unfolding exactly those nodes whose screenspace error exceeds a user-specified threshold *t*, HDS enforces a quality constraint on the simplification: no vertex shall move by more than *t* pixels on the screen.

Determining the exact screenspace extent of a vertex cluster can be a time-consuming task, but a conservative estimate can be efficiently obtained by associating a bounding volume with each node in the vertex tree. The current implementation uses bounding spheres, which allow an extremely fast screenspace extent test but often provide a poor fit to the vertex cluster. The function **`nodeSize()`** tests the bounding sphere of a node and returns its extent on the screen as a fraction of viewport size. The recursive procedure **adjustTree()** uses **nodeSize()** in a top-down fashion, evaluating which nodes to fold and unfold. Nodes with extent greater than *t* are unfolded and smaller nodes are folded:

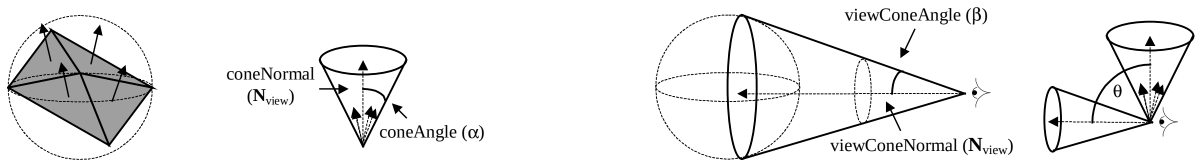
```
adjustTree (Node *N)
   size = nodeSize(N);
   if (size >= threshold)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // size < threshold
         foldSubtree(N);
```

The recursive function **foldSubtree()**, as the name suggests, folds the entire subtree rooted at node N:

```
foldSubtree (Node *N)
   if (node->label == active)
         foreach child C of N
               foldSubtree(C);
               foldNode(C);
```

4.2. Silhouette Preservation

Silhouettes and contours are particularly important visual cues for object recognition. Detecting nodes along object silhouettes and allocating more detail to those regions can therefore disproportionately increase the perceived quality of a simplification [Xia 96]. A conservative but efficient silhouette test can be plugged into the HDS framework by adding two fields to the **Node** structure: `coneNormal` is a vector and `coneAngle` is a floating-point scalar. These fields together specify a *cone of normals* [Shirman 93] for the node, which bounds all the normals of all the triangles supported by the node. At run time a viewing cone is created that originates from the viewer position and tightly encloses the bounding sphere of the node (Figure 11). Testing the viewing cone against the cone of normals determines whether the node is completely frontfacing, completely backfacing, or (if any normal in the cone of normals is orthogonal to any direction contained within the viewing cone) potentially on the silhouette.



(a) A node containing four triangles, shown with its bounding sphere, and the node's cone of normals.

(b) The viewing cone originates from the viewer and tightly encloses the node's bounding sphere. The angle between $\mathbf{N}_{cone}$ and $\mathbf{N}_{view}$ is denoted θ.

Figure 11: Silhouette preservation. If any vector within the viewing cone is at right angles to any vector within the cone of normals, the node may be on the silhouette.

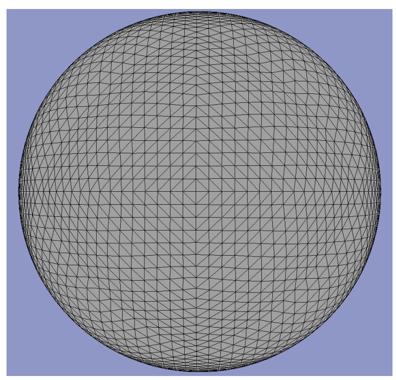
```
testSilhouette(Node *node, Coord eyePt)
   α = node->coneAngle;
   N_cone = node->coneNormal;
   β = calcViewConeAngle(eyePt, node);
   N_view = calcViewConeNormal(eyePt, node);
   θ = cos^-1(N_view • N_cone);
   if (θ - α - β > π/2)
         return FrontFacing;
   if (θ + α + β < π/2)
         return BackFacing;
   return OnSilhouette;
```

Silhouette preservation dovetails nicely with the screenspace error metric approach presented above: the operation determines which nodes may be on the silhouette, and these nodes are then tested against a tighter screenspace error threshold (*Ts*) than interior nodes (T_I). Also, nodes that **`testSilhouette()`** evaluates as entirely backfacing can be collapsed, aggressively simplifying portions of the model oriented away from the viewer. This is called *backface simplification*. Figure 12 illustrates silhouette preservation and backface simplification using a simple model of a sphere. The **`adjustTree()`** operation is easily modified to incorporate this test:

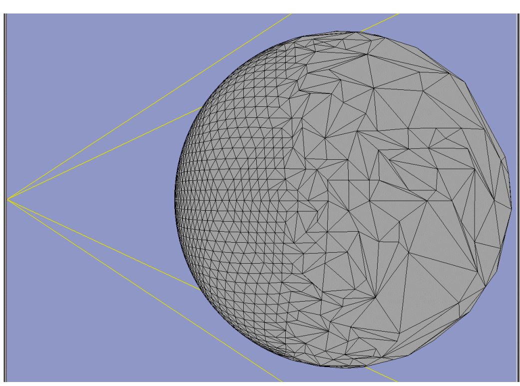
```
adjustTree(Node *N)
   size = nodeSize(N);
   result = testSilhouette(N);
   if (result == OnSilhouette)
         testThreshold = T_s;
   else if (result == FrontFacing)
         testThreshold = T_I;
   else  // result == BackFacing
         foldSubtree(N);
         return;

   if (size >= testThreshold)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else    // size < testThreshold
         foldSubtree(N);
```

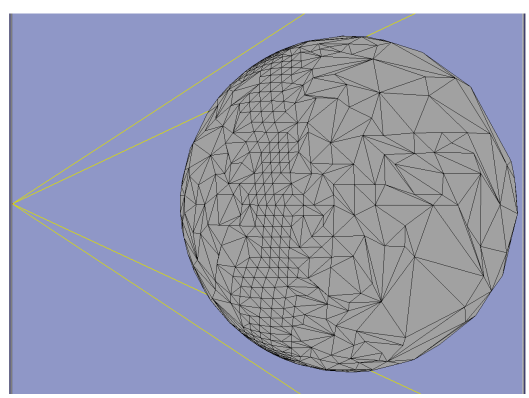
Note that *hierarchical backface culling* falls out of the silhouette preservation test if polygons of backfacing nodes are not rendered [Kumar 96].



(a) Original resolution (8,192 faces)



(b) 1% error threshold, backface simplification enabled (3,388 faces)



(c) 1% silhouette error threshold, 20% interior error threshold (1,950 faces)

Figure 12: Silhouette preservation and backface simplification.

4.3. Triangle-Budget Simplification

The screenspace error threshold and silhouette test allow the user to set a bound on the fidelity of the simplified scene, but often a bound on the complexity (and thus rendering time) is desired instead. *Triangle budget simplification* allows the user to specify how many triangles the scene should contain. HDS then minimizes the maximum screenspace error of all boundary nodes within this triangle budget constraint. The intuitive meaning of this process is easily put into words: "Vertices on the screen can move as far as *t* pixels from their original position. Using no more than *n* triangles, minimize *t*."

The current system performs triangle budget simplification using a priority queue of boundary nodes, sorted by screenspace error. The node N with the greatest error is unfolded, removing N from the top of the queue and inserting the children of N back into the queue. This process iterates until unfolding the top node of the queue would exceed the triangle budget, at which point the maximum error has been minimized. The simplification could further refine the scene by searching the priority queue for the largest nodes that can still be unfolded without violating the triangle budget, but this is unnecessary in practice. The initial minimization step works extremely well on all models tested, and always terminates within a few triangles of the specified budget. Pseudocode for this procedure is straightforward, using a standard heap to implement the priority queue:

```
budgetSimplify(Node *rootnode)
// Initialize priority queue Q to contain just the rootnode
Heap *Q(rootnode);

while (Q->topnode->nsubtris < tribudget)
   unfoldNode(Q->topnode);
   // insert children, sorted by screenspace error:
   foreach child C of Q->topnode
         Q->insert(C);
   tribudget = tribudget - Q->topnode->nsubtris;
   Q->removeTopnode();
```

Note that simply bounding the number of triangles in a scene does not guarantee a constant frame rate on most modern graphics hardware. Triangle count directly affects the amount of geometric computation, or *transformation cost*, of a scene, but equally important is the *fill rate*, or speed with which the graphics hardware can write pixels to the framebuffer. Since large triangles require filling many more pixels than small triangles, overall frame rate depends not only on the number but also the size of the triangles rendered. Models complex enough to warrant polygonal simplification techniques, however, tend to consist mainly of small polygons. Chapter 7 will demonstrate that fill rate is not a rendering bottleneck for any sizable model tested in HDS. In practice, then, regulating the number of triangles provides good control and a nearly constant frame rate.

CHAPTER 5

OPTIMIZING THE ALGORITHM

An initial naïve implementation of the HDS algorithm ran at 10-20 frames per second on small models, no larger than 20,000 triangles or so[2]. The final system has been demonstrated on models more than two orders of magnitude larger. Four kinds of optimizations together made this possible: exploiting temporal coherence, using visibility information, streamlining the math, and parallelizing the algorithm.

5.1. Exploiting Temporal Coherence

Interactive viewing sessions exhibit a high degree of frame-to-frame coherence in the position of the viewer, and HDS takes advantage of this fact throughout. The design of the active triangle list, for example, is based on the assumption that relatively few triangles will be added to, or removed from, the scene each frame. An alternative would be to generate the list of visible triangles anew each frame by traversing from the active nodes of the vertex tree, but this is wasteful if most triangles remain visible from one frame to the next. As Figure 13 shows, less than 2.5% of the triangles were added, deleted, or adjusted each frame during a typical path through the Torp model at a 1% screenspace error threshold, and less than 1.2% of the triangles were added, deleted, or adjusted each frame with a 5% error threshold. The active triangle list exploits temporal coherence by storing the unchanging triangles from frame to frame and supporting efficient add, delete, and update operations for the rest.

---

[2] On an SGI Onyx computer with InfiniteReality graphics. Chapter 7 has details.

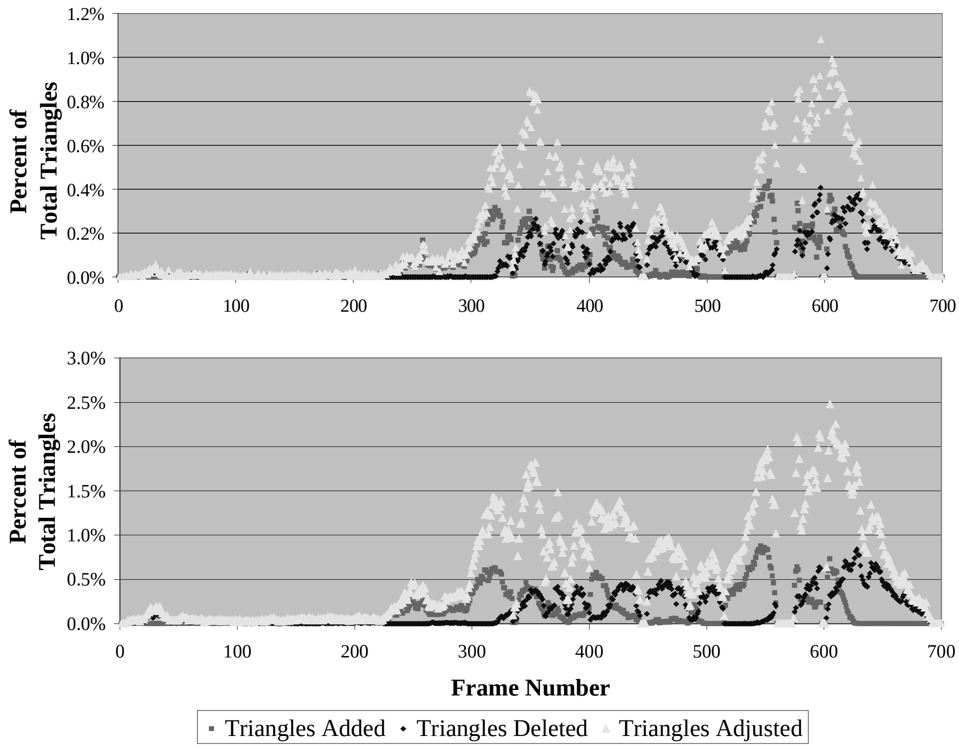


Figure 13: Triangles added, deleted, and adjusted during a 700-frame path through the 699,000-triangle Torp model, using screenspace error thresholds of 1% (top) and 5% (bottom). This path is used for all plots presented in this dissertation.

The vertex tree traversal can also profit from temporal coherence. Just as few triangles change status from frame to frame, few nodes in the vertex tree change status from frame to frame. Most nodes labeled Inactive this frame will remain Inactive next frame; most Active nodes will remain Active; most Boundary nodes will remain Boundary. Under these conditions the **`adjustTree()`** function is inefficient, visiting many nodes unnecessarily. The root node (for example) will typically be active the entire viewing session, so why test it each frame? One improvement would be to modify **`adjustTree()`** to evaluate only boundary nodes and their parents, skipping quickly over the majority of unchanging nodes, but this still suffers from the disadvantage that those nodes have to be loaded into memory, if only to check their status.

A better scheme is to traverse, not down the vertex tree as **`adjustTree()`** does, but across the vertex tree along the path formed by boundary nodes. In this way active nodes far from the action are never considered and need not even be resident in memory. This path, called the *boundary path*, is maintained as a doubly-linked list by adding **`prev`** and **`next`** fields to the **`Node`** structure. The function **`adjustPath()`** traverses the boundary path, folding and unfolding nodes as necessary[3]:

```
adjustPath ()
   Node *current;               // node currently being tested
   Node *parent;                // parent of current node
   Node *lastparent;            // parent node last iteration

   // beforepath and afterpath are dummy nodes bracketing the path
   current = beforepath->next;

   while (current->next != afterpath)
         current = current->next
         parent = current->parent;
         if (parent != lastparent)
               lastparent = parent;
               // check parent's size first
               if (nodeSize(parent) < threshold)
                     // parent falls below threshold; fold parent
                     foldSubtree(parent);
                     current = parent;
                     continue;
         // parent is fine, check current node
         if (nodeSize(current) >= threshold)
               // current node too large; unfold
               unfoldNode(current);
```

This traversal scheme requires modifying the **`foldNode()`**and **`unfoldNode()`** functions to maintain the boundary path:

---

[3] Note that this pseudocode uses the **`nodeSize()`** function for clarity, but the actual implementation uses the streamlined expression described in section 5.4

```
foldNode (Node *N)
   N->label = boundary;
   foreach child C of N
         // children should be labeled boundary; change to inactive
         C->label = inactive;
   foreach triangle T in N->tris
         // update tri proxies
         foreach corner c of {1,2,3}
               T->proxies[c] = firstActiveAncestor(T->corners[c]);
   foreach triangle T in N->subtris
         // remove subtris from active list
         removeTri(T);
   N->prev = N->firstChild->prev;
   N->next = N->lastChild->next;
   N->prev->next = N;
   N->next->prev = N;
```

```
unfoldNode (Node *N)
   Node *pred = N->prev;
   Node *succ = N->next;

   foreach child C of N
         C->label = boundary;
         C->prev = pred;
         pred->next = C;
         pred = C;
   N->label = active;
   foreach triangle T in N->tris
         // update tri proxies
         foreach corner c of {1,2,3}
               T->proxies[c] = firstActiveAncestor(T->corners[c]);
   foreach triangle T in N->subtris
         // add subtris to active list
         addTri(T);
   pred->next = succ;
   succ->prev = pred;
```

Another frequent operation that can take advantage of coherence is the **`firstActiveAncestor()`** function, used heavily by **`foldNode()`** and **`unfoldNode()`**. **`FirstActiveAncestor(N)`** returns the nearest ancestor of node N which is labeled active or boundary. A simple implementation is straightforward:

```
Node *firstActiveAncestor (Node *N)
    Node *tmp = N;

    while (tmp->label == Inactive)
          tmp = tmp->parent;
    return tmp;
```

However, if the node N lies far below the boundary path in the vertex tree, every call to **`firstActiveAncestor(N)`** will traverse the same path from N to its first active ancestor. From frame to frame, this ancestor is likely to remain fixed; when it does change, the first active ancestor of a node tends to move up or down the tree by only a node or two. The **`firstActiveAncestor(N)`** operation can exploit this by storing the result of each search and starting the next search from the previous result. The modified function uses a new **`ancestor`** field to the Node structure, and uses the **`path`** field to guide the search down the tree along the correct path:

```
Node *firstActiveAncestor (Node *N)
    Node *tmp = N->ancestor;      // the current node being examined
    Byte whichchild;              // which child leads from tmp to N?

    if (tmp->label == boundary or tmp == N)
          // previous FAA still on the boundary; return it
          return tmp;
    else if (tmp->label == Inactive)
          // previous FAA Inactive, search up the tree
          while (tmp->label == Inactive)
                tmp = tmp->parent;
          N->ancestor = tmp;
          return tmp;
    else
          // previous FAA active, search down tree for boundary node
          do
                if (tmp->label == boundary or tmp == N)
                      N->ancestor = tmp;
                      return tmp;
                whichchild = N->path[tmp->depth];
                tmp = tmp->children[whichchild];
          loop
          N->ancestor = tmp;
          return tmp;
```

5.2. Visibility: Accelerating Rendering

For many applications, most of the model is invisible most of the time. In architectural CAD, for example, a user might want a *walkthrough*, steering the viewpoint interactively through the interior of a virtual building. Since the horizontal field of view in such an application is typically 90° or less, a quarter of the floorplan or less will be visible to a viewer from the center of the model. A limited vertical field of view restricts the visible portion of the model still further. The process of quickly identifying and rejecting objects outside the visible field of view is called *view-frustum culling*. In applications such as architectural walkthroughs, view-frustum culling can greatly decrease rendering time by not rendering invisible portions of the model.

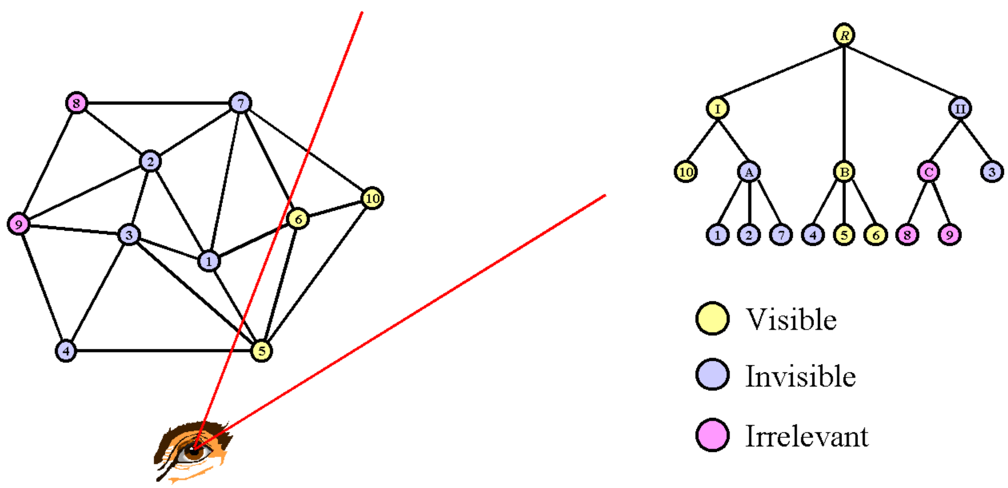


Figure 14: Visible, invisible, and irrelevant nodes. Invisible nodes lie outside the view frustum. Irrelevant nodes are invisible *and* support no vertices of visible triangles.

Enabling efficient view-frustum culling in HDS requires modifying the active triangle list slightly. The problem is that the active triangle list as described exploits temporal coherence but not spatial coherence. Triangles are added to and removed from the list in a haphazard fashion as nodes are folded and unfolded, so triangles near each other in the model are unlikely to be near each other in the active triangle list. Without spatial coherence, view-frustum culling can only be done on a per-triangle basis. Unfortunately, determining the visibility of an individual triangle takes as long as rendering the triangle, so applying view-frustum culling techniques to every triangle in the model fails to speed up rendering.

The solution is to impose spatial coherence by splitting the active triangle list into many lists, each representing a portion of the complete model. When unfolding a node creates a triangle, it is added to whichever list corresponds to the portion of the model containing the triangle. View-frustum culling is applied to the lists themselves, or rather to the portions of the model they represent. The rendering process tests a bounding volume associated with each list and skips any lists determined to be invisible. Organizing the lists and bounding volumes hierarchically can speed up this process even further by allowing the rendering process to reject large chunks of the scene without examining the individual lists containing the triangles that constitute portions of those large chunks.

The vertex tree provides a ready-made hierarchy for these multiple active triangle lists. Each node in the vertex tree represents a subset of the vertices in the model, and for every triangle, one or more nodes exist that represent all three of that triangle's corners. Of these nodes, the triangle's *cull node* is the farthest from the root of the vertex tree. The cull node is invisible if its bounding volume lies outside the view frustum. Since that volume bounds all three corners of the triangle, the triangle need not be rendered if its cull node is invisible. This property holds hierarchically: the descendants of an invisible node are invisible, and no triangles having the node or a descendant as a cull node need be rendered.

A naïve implementation of view-frustum culling, then, simply involves creating an active triangle list for every node of the vertex tree and modifying **`unfoldNode()`** to add each triangle to the active triangle list of its cull node. Rendering the scene then involves a top-down traversal of the active tree, testing each node against the view frustum. If the node is visible, its active triangle list is rendered and its children traversed in turn. If the node is invisible, nothing is rendered and traversal of that branch of the vertex tree terminates. This scheme enables efficient visibility culling, but sacrifices some of the advantages of temporal coherence, since *every* visible active node must be visited every frame. On complex models the overhead of traversing a deep active tree undermines the benefit of rendering fewer triangles, and rendering the hierarchy of lists takes longer than rendering a single global list.

In practice a hybrid approach works well. Cull nodes are restricted to high levels of the vertex tree, providing a coarse-grained culling without the overhead of a full active tree traversal. The resulting system thus exploits both visibility culling and temporal coherence. For all the results presented in this thesis, cull nodes were restricted to nodes of depth ≤ 4. A more sophisticated approach might be to restrict cull nodes to those nodes representing vertices supporting more than some particular number of triangles.

5.3. Visibility: Accelerating Simplification

Distributing the active list across multiple cull nodes speeds up rendering by quickly discarding triangles contained by invisible nodes. HDS may still need to examine such nodes, however, since the **`tris`** and **`subtris`** fields of an invisible node may refer to visible triangles (Figure 14). This fact gives rise to a stronger condition: some nodes are not only invisible but also *irrelevant*. Irrelevant nodes are defined as invisible nodes that support no vertices of visible triangles. Folding or unfolding an irrelevant node therefore cannot possibly affect the scene, and the simplification traversal can save time by not visiting these nodes. In a walkthrough session, the vast majority of invisible nodes are usually irrelevant, so testing for irrelevance provides a significant speedup. An exact test is difficult, but a conservative test for irrelevant nodes is easily constructed by adding a **`container`** field to the **`Node`** structure. The container node C of a node N is the smallest node that contains every tri and subtri of N and N's descendants. C thus contains every triangle that might be affected by operations on the subtree rooted at N. If C is invisible, N is irrelevant and can be safely ignored by the simplification traversal.

5.4. Streamlining the Math

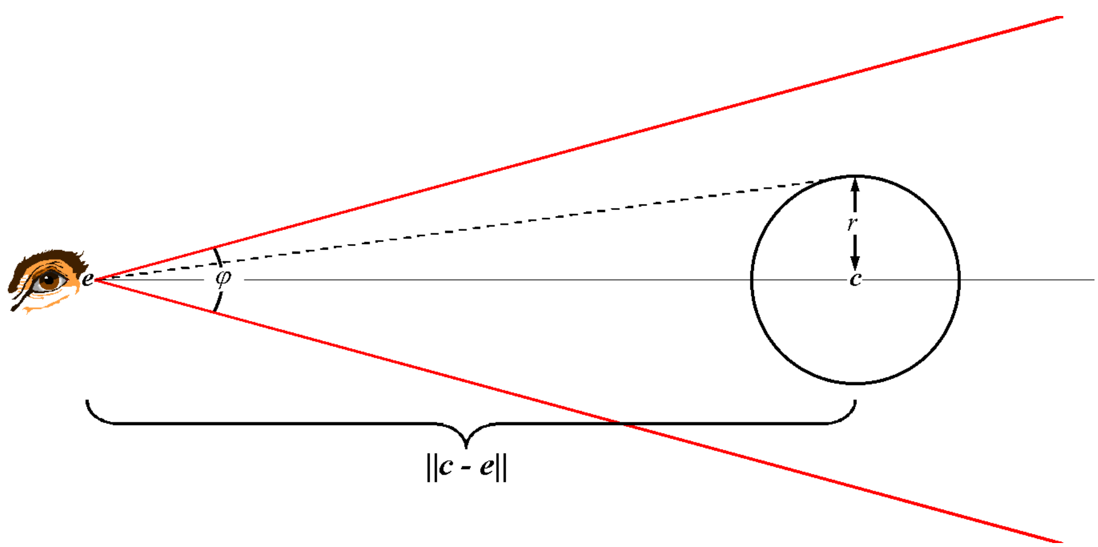


Figure 15: Calculating the screenspace extent of a node. The node is approximated by a bounding sphere of radius *r* centered at *c*. The eyepoint is *e* and the field of view is φ.

Some of the geometric operations in HDS are quite complex; appropriate use of approximations and careful attention to implementation details can greatly streamline the computation involved. For example, the **`nodeSize()`** function from Section 4.1 finds the extent of a cluster of vertices when projected onto the screen. An exact solution would presumably involve projecting the vertices (or their convex hull) and comparing the resulting screen coordinates, a dauntingly expensive operation. Since **`nodeSize()`** is typically called thousands of times per frame, an approximate solution based on bounding spheres is used instead. Figure 15 shows a sphere with center $\boldsymbol{c}$ and radius r, seen from the eyepoint $\boldsymbol{e}$ with field-of-view angle φ. The fraction of viewport F occupied by the sphere is estimated by:

$$F = \frac{r}{\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2)} \qquad \text{(Equation 1)}$$

Note that this approximation assumes that the sphere lies in the center of the field of view, and slightly underestimates F for nodes near the edges of the viewport. Already much simpler than the exact calculation, this fairly terse expression can be optimized still further in context. The function **`adjustTree()`**, for instance, compares each node's screenspace extent F to a user-specified threshold t. This amounts to evaluating the inequality:

$$F \geq t \qquad \text{(Equation 2)}$$

which reduces to:

$$r \geq t\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2) \qquad \text{(Equation 3)}$$

Squaring both sides and dividing by $\tan^2(\varphi/2)$ yields:

$$r^2 \cot^2(\varphi/2) \geq t^2 \|\boldsymbol{c} - \boldsymbol{e}\|^2 \qquad \text{(Equation 4)}$$

The $\cot^2(\varphi/2)$ term is precalculated at the beginning of each frame. This expression is well suited for rapid evaluation; the lack of division or square root operations makes it particularly palatable. Fixing the field-of-view angle throughout the viewing session enables a further optimization: compute and store the entire $r^2\cot^2(\varphi/2)$ term for each node, instead of just the radius r of the node's bounding sphere. Though these rearrangements may seem minor, the final expression evaluates much faster. As Figure 16 shows, this optimization alone more than tripled the speed of the simplification process in practice.

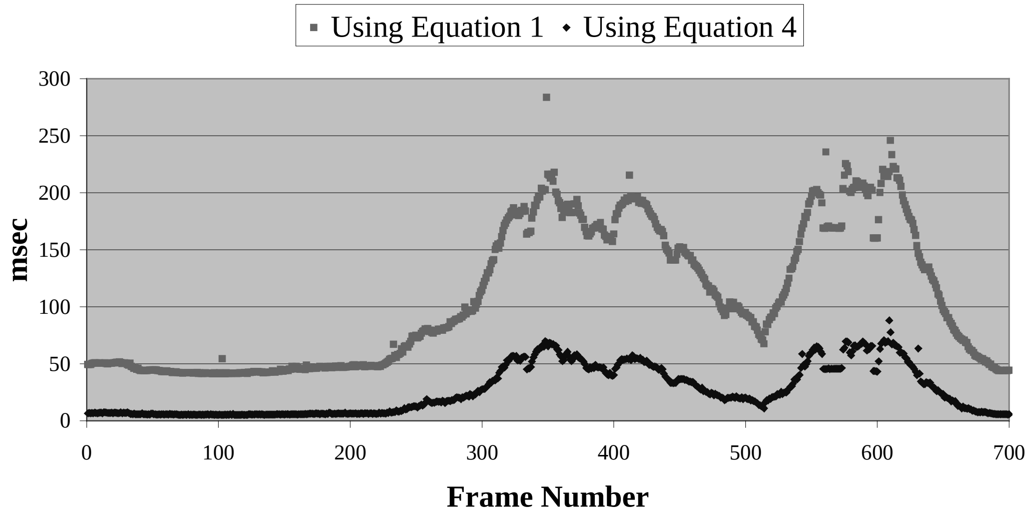


Figure 16: Plot of time spent in `adjustTree()` using Equation 1 versus Equation 4.

Here is the modified **`adjustTree()`** function, with silhouette tests omitted for clarity. The `threshold2` term, as the name suggests, holds the user-specified threshold, squared, and the new `r2cot2` field of the **`Node`** structure stores $r^2\cot^2(\varphi/2)$ for the node. Modifying the **`adjustPath()`** function along the same lines is straightforward.

```
adjustTree (Node *N)
   distance2 = (N->center[X] – eyept[X])^2 +
               (N->center[Y] – eyept[Y])^2 +
               (N->center[Z] – eyept[Z])^2
   if (N->r2cot2 >= threshold2 * distance2)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // node size is below threshold
         foldSubtree(N);
```

5.5. Parallelization: Asynchronous Simplification

An important strategy for speeding up any algorithm is to parallelize it, distributing the work over multiple processors. Computer graphics applications most commonly accomplish this by performing the major stages of the rendering computation concurrently in a pipeline fashion. A traditional level-of-detail system might be divided into SELECT and RENDER stages: the SELECT stage decides which resolution of which objects to render and compiles them into a display list, which the RENDER process then renders. Meanwhile, the SELECT process prepares the display list for the next frame [Funkhouser 93, Rohlf 94]. If S is the time taken to select levels of detail and R is the time taken to render a frame, performing the two processes as a pipeline reduces the total time per frame from R+S to max(R,S).

HDS divides naturally into two basic tasks, SIMPLIFY and RENDER. The SIMPLIFY task traverses the vertex tree, folding and unfolding nodes as needed. The RENDER task cycles over the active triangle list rendering each triangle. Let the time taken by SIMPLIFY to traverse the entire tree be S and the time taken by RENDER to draw the entire active list be R. The frame time of a uniprocessor implementation will then be R+S, and the frame time of a pipelined implementation will again be max(R,S). The rendering task usually dominates the simplification task, so the effective frame time often reduces to R. The exception is during large shifts of viewpoint, when the usual assumption of temporal coherence fails and many triangles must be added and deleted from the active triangle list. This can have the distracting effect of slowing down the frame rate exactly when the user speeds up the rate of motion.

Asynchronous simplification provides a solution: let the SIMPLIFY and RENDER tasks run asynchronously, with the SIMPLIFY process writing to the active triangle list and the RENDER process reading it. This decouples the tasks for a total frame time of R, eliminating the slowdown artifact associated with large viewpoint changes. When the viewer's velocity outpaces the simplification rate in asynchronous mode, the SIMPLIFY process simply falls behind. Typically, this results in a temporary coarsening of the scene quality. Under HDS, the portions of the scene near the viewer are refined to high detail whereas distant portions are simplified to coarse detail. If the user moves forward too quickly for the SIMPLIFY process to keep up, the viewpoint will leave the highly detailed region behind and move into a coarsely represented region. The scene rendered for the viewer remains coarse in quality until the SIMPLIFY process catches up, at which point the scene gradually refines back to the expected quality. This graceful degradation of fidelity is less distracting than sudden drops in frame rate.

CHAPTER 6

CONSTRUCTING THE VERTEX TREE

The previous chapters have described the vertex tree and how it enables dynamic view-dependent simplification, but have left open the question of how the vertex tree is constructed in the first place. This is an important issue, for the care with which the vertex tree is constructed directly affects the quality of the simplifications extracted from it. The vertex tree is completely determined by the order in which vertices are grouped. Once the hierarchical grouping of vertices is established, the matter of calculating subtris, container nodes, and so on becomes a purely mechanical process. The problem, then, is how to perform this hierarchical vertex clustering.

Possible algorithms for clustering vertices form a spectrum, ranging from fast, simple approaches whose resulting simplifications have moderate fidelity to slower, more sophisticated methods with superb fidelity. The choice of algorithm for constructing the vertex tree is heavily application-dependent. In a design-review setting, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours are unacceptable in this scenario. On the other hand, a walkthrough of the completed model might be desired for demonstration purposes. Here it makes sense to use a slower, more careful algorithm to optimize the quality of simplifications and prevent any distracting artifacts.

6.1. Simplest: Spatial Subdivision

One of the simplest techniques is to classify the vertices of the model with a hierarchical space-partitioning structure. Recall the spatial binning approach introduced by Rossignac and Borrel, which clustered vertices according to a uniform grid [Rossignac 92]. The first versions of HDS used a straightforward extension of the Rossignac-Borrel algorithm to construct the vertex tree, clustering vertices in a top-down fashion with an *octree*. In this method vertices are first ranked by importance using local criteria such as edge length and curvature. The root node of the octree is an axis-aligned cube large enough to contain every vertex in the model. Beginning with this root node, the most important vertex within each node is chosen as that node's repvert. The node is then divided exactly in half along the X, Y, and Z directions into 8 cubical subnodes (hence the name "octree"). The vertices are then partitioned among the node's eight children and the process is recursively repeated for any subnode with more than one vertex. In this way vertices are clustered roughly according to proximity. Neighboring vertices are likely to get clustered near the leaves of the tree, whereas distant vertices tend to merge only at higher levels of the tree.

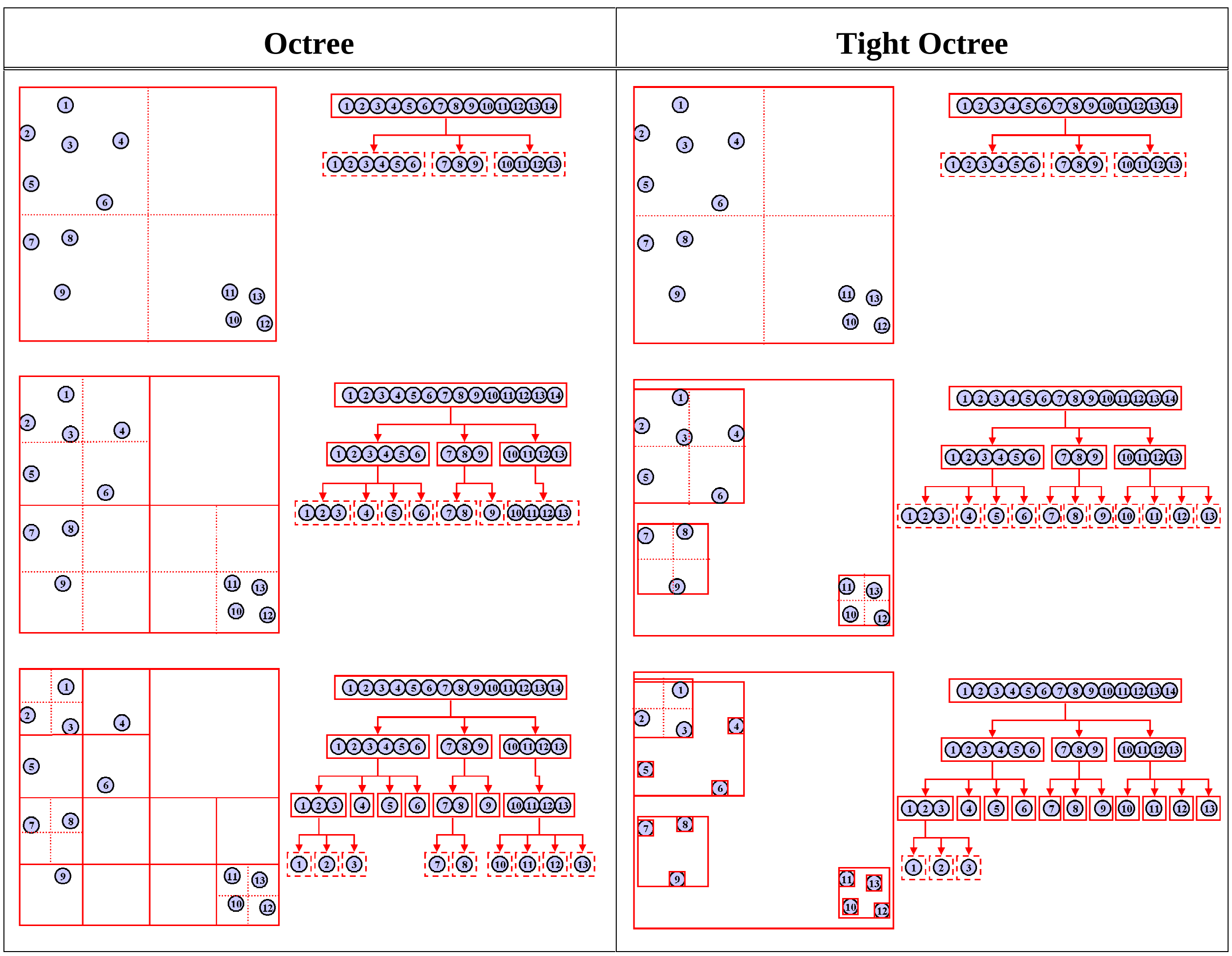


Figure 17: A 2-D example of octree vs. tight-octree clustering. Note that the tight octree produces two fewer nodes than the octree even on this small example.

Unless the vertices of the model are uniformly distributed, the straightforward approach just described will result in unbalanced octrees with more nodes than necessary, which wastes storage space and traversal time. CAD models are often locally dense but globally sparse, consisting of highly detailed components separated by large areas of low detail or empty space. In this situation a more adaptive partitioning structure is desired. The *tight octree* is a modified octree in which each node is tightened to the smallest axis-aligned cube that encloses the relevant vertices before the node is subdivided (Figure 17). This approach seems to adapt very well to CAD models, and most results presented in this thesis used tight-octree spatial subdivision to cluster vertices.

Top-down spatial subdivision clustering schemes possess many advantages. Their simplicity makes an efficient, robust implementation relatively easy to code. In addition, spatial partitioning of vertices is typically very fast, bringing the preprocess time of even large models down to manageable levels. Preprocessing the 700,000-polygon torpedo room model, for example, takes only 143 seconds using a tight-octree clustering scheme. Finally, spatial-subdivision vertex clustering is inherently very general. No knowledge of the polygon mesh is used; manifold topology is neither assumed nor preserved. In the CAD domain, meshes with degeneracies such as cracks, T-junctions, and missing polygons are regrettably common. Spatial-subdivision vertex clustering schemes will operate despite the presence of degeneracies incompatible with more complex schemes.

6.2. Prettiest: Simplification Envelopes, Progressive Mesh Algorithm

On the other end of the spectrum, some very sophisticated simplification algorithms could be used to build the vertex cluster tree. Section 2.2.7 described the *Simplification Envelopes* approach of Cohen et al., which uses offset surfaces of a polygonal mesh bounded to a distance ε of the mesh and modified to prevent self-intersection. By generating a simpler triangulation of the surface without intersecting the simplification envelopes, the authors guarantee a simplification that preserves global topology and varies from the original surface by no more than ε [Cohen 96]. Simplification envelopes could be used to construct the vertex tree in HDS by applying successively larger values of ε, and at each stage clustering those vertices that do not cause the mesh to intersect the envelopes. The value of ε used to generate each cluster would then become the error metric associated with that node in the vertex tree. The resulting simplifications should have excellent fidelity. Unfortunately, it is not clear how to extend simplification envelopes to allow merging between different objects, or to allow drastic topology-discarding collapse operations at high levels of the tree.

Hoppe describes an optimization approach that creates a series of edge collapses for the *Progressive Meshes* representation [Hoppe 96]. The stream of edge collapse records in a progressive mesh contains an implicit hierarchy that maps directly to the HDS vertex tree. Each edge collapse corresponds to a node in HDS with two children and one or two subtris. A progressive mesh could thus be viewed without modification in an HDS system, though this has disadvantages. A progressive mesh never collapses more than two vertices together at a time, which may result in an unnecessarily deep vertex tree. A modified optimization step that could collapse multiple vertices seems possible, and would address this problem. Also, progressive meshes collapse only vertices within a mesh, so separate objects never merge together. Finally, restricting edge collapses to those that preserve the manifold topology of the mesh limits the amount of simplification possible[4]. For these reasons, directly embedding a progressive mesh into the HDS vertex tree does not lend itself to drastic simplification, and may not be optimal for visualizing complex CAD models.

However, Hoppe's method of maintaining discrete and scalar attributes as vertices are collapsed extends directly to HDS, and is used without modification in the current implementation.

6.3. A Hybrid Approach

Sophisticated, high-fidelity methods such as the simplification envelope and progressive mesh approaches can be combined with top-down spatial subdivision to allow drastic simplification and merging of objects. Since neither approach allows vertices from different meshes to merge, the result of either on a collection of objects in a scene is a collection of vertex trees. When the vertex tree produced by the high-fidelity algorithm for each object is

[4] For example, our implementation could not reduce the 69,451-triangle bunny model beyond 520 triangles.

judged adequate, the spatial subdivision algorithm unifies this “vertex forest” into a single tree. A tight octree or similar structure merges nearby vertex clusters, without regard to topology or source mesh. The final vertex tree exhibits both high fidelity (at low levels of the tree) and drastic simplification (at high levels).

The simplifications used to illustrate silhouette preservation as a run-time criterion were generated with this type of hybrid approach (see Figure 12). The nature of the silhouette test made a hybrid approach more attractive than the usual tight-octree clustering for two reasons. First, effective silhouette preservation requires clustering vertices of coplanar regions in preference to clustering vertices across a crease in the mesh. This means merging vertices so as to minimize the normal cones of the resulting vertex cluster rather than merging vertices according to simple proximity. Second, the curvature (and therefore silhouette) of a non-manifold mesh is not well defined. To preserve manifold topology, only certain adjacent vertices in the mesh should be collapsed.

These considerations led to a two-stage clustering algorithm. First, a progressive mesh-like algorithm was applied, in which edges were collapsed so as to chosen to minimize the resulting normal cones and to maintain a balanced tree. The vertex resulting from each edge collapse was simply chosen to be the midpoint of the collapsed edge. Collapses that resulted in normal cone angles greater than 135^{o} were disallowed. When the model could be simplified no further with these restrictions, a tight octree was applied to the remaining vertex clusters to produce a single HDS vertex tree.

CHAPTER 7

RESULTS AND ANALYSIS

7.1. The Platform

Unless otherwise noted, all results reported in this thesis were obtained on a four-processor Silicon Graphics Onyx^2 computer with 195 Mhz MIPS R10000 processors, 1152 megabytes of main memory, 4 megabytes of secondary cache, and InfiniteReality graphics.

7.2. The Models

Model	Category	Vertices	Triangles
Sphere	Procedural	4,098	8,192
Bunny	Scanned	35,947	69,451
Sierra	Terrain	81,920	162,690
Cassini	Aerospace CAD	189,615	415,257
AMR	Maritime CAD	280,544	504,969
Torp	Maritime CAD	411,778	698,872
Bone6	Medical	569,685	1,136,785
Powerplant_3M	Structural CAD	1,303,162	2,796,984
Powerplant_4M	Structural CAD	1,794,082	3,879,736

Table 2: The names, categories and complexity of the models used to test HDS.

Nine sample models were chosen to span several model categories and a large range of polygon counts. *Sphere* is a simple 8,200-triangle sphere generated procedurally to illustrate the effect of silhouette preservation and backface simplification. *Bunny* comes from the Stanford 3-D Scanning Repository at *http://www-graphics.stanford.edu/data/3Dscanrep*, and was generated by combining several separate meshes generated by a laser scan of a clay bunny figurine. The bunny contains 70,000 triangles and has become an unofficial emblem of the polygonal simplification field. *Sierra* is a terrain model, courtesy Sun Microsystems, and consists of 163,000 triangles in a regular height field. *Cassini* is an aerospace CAD model of the Cassini space probe, provided courtesy of the Jet Propulsion Laboratory. It contains over 415,000 triangles. *AMR* depicts the auxiliary machine room of a notional nuclear submarine, containing approximately 505,000 triangles. *Torp* is another maritime CAD model, representing the torpedo room of the same dataset with approximately 699,000 triangles. The Electric Boat Division of General Dynamics Corporation provided both submarine models. *Bone6* is a 1.1-million triangle medical model created from an isosurface of the Visible Man volumetric dataset [Lorenson 95]. Finally, *Powerplant_3M* is a 3-million triangle subset of a coal-fired powerplant model, and *Powerplant_4M* is a 4-million triangle subset of the same model. The entire powerplant dataset, provided courtesy of ABB Engineering, comprises over 13,000,000 triangles in all its original detail. These subsets were used because lack of memory prevents HDS from preprocessing the full model in a reasonable time.

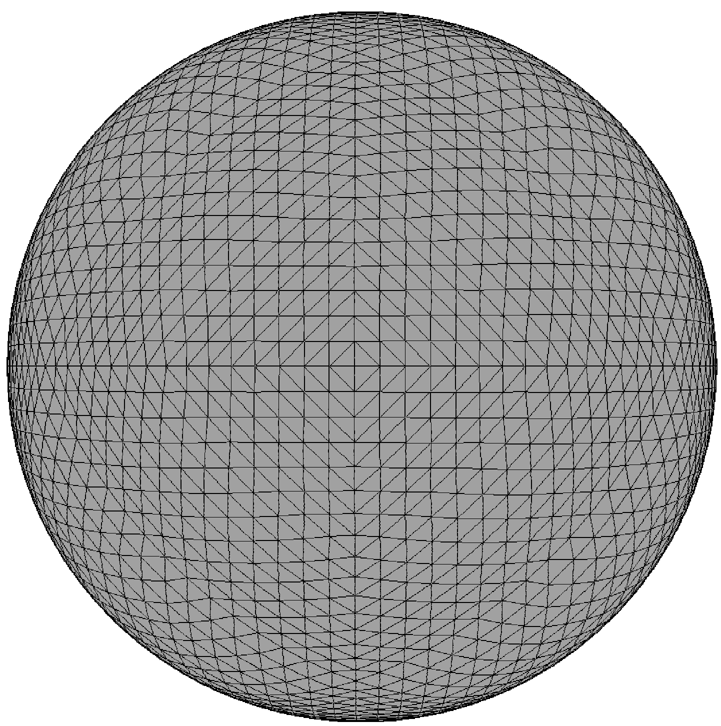


Figure 18: The Sphere model is densely tesselated, containing 8,192 triangles.

Figure 19: The Bunny model consists of 69,451 triangles.

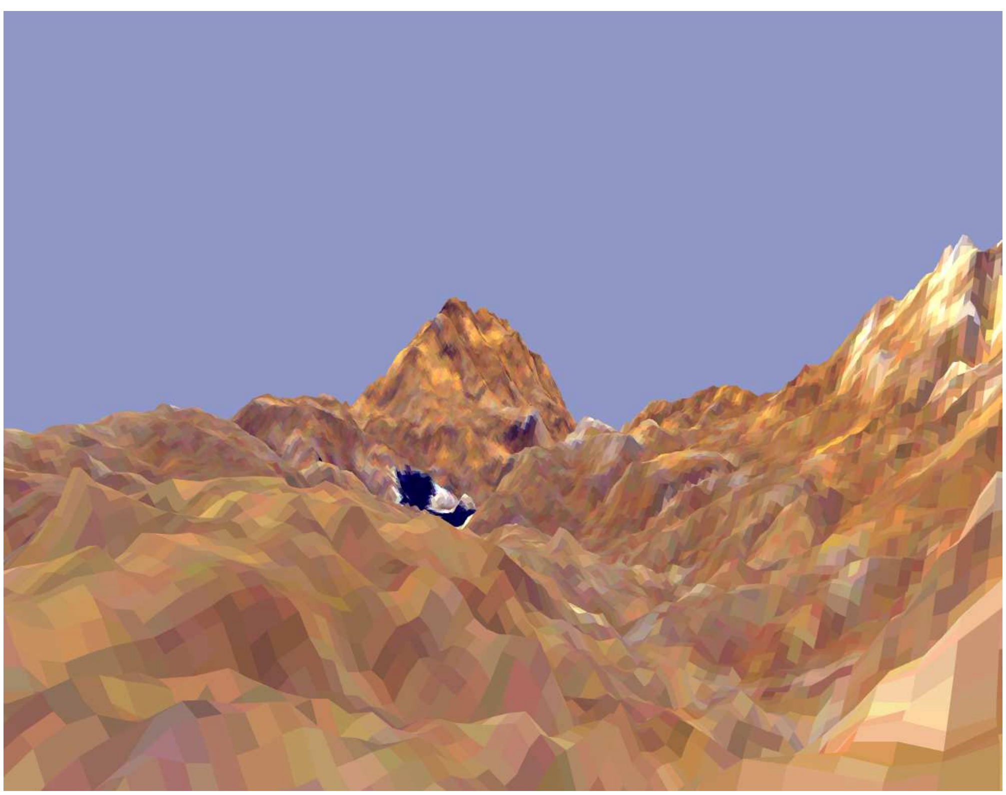


Figure 20: The Sierra terrain contains 162,690 triangles.

Figure 21: The Cassini space probe model contains 415,257 triangles.

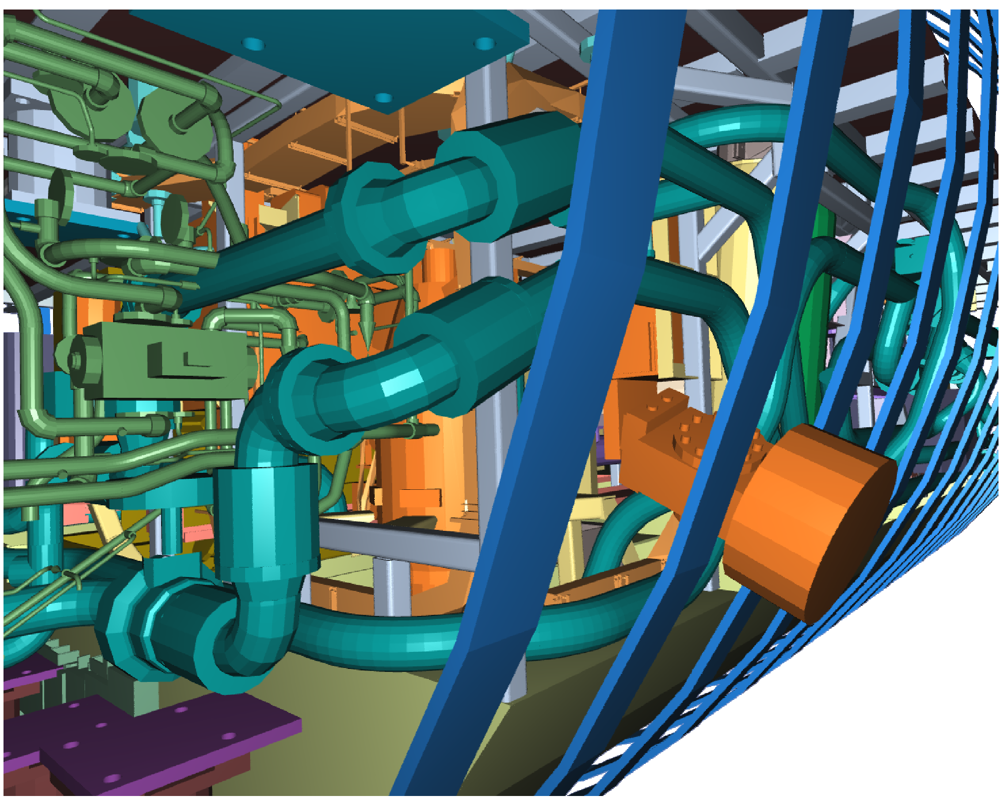


Figure 22: A close-up view of the AMR model, which contains 504,969 triangles in all.

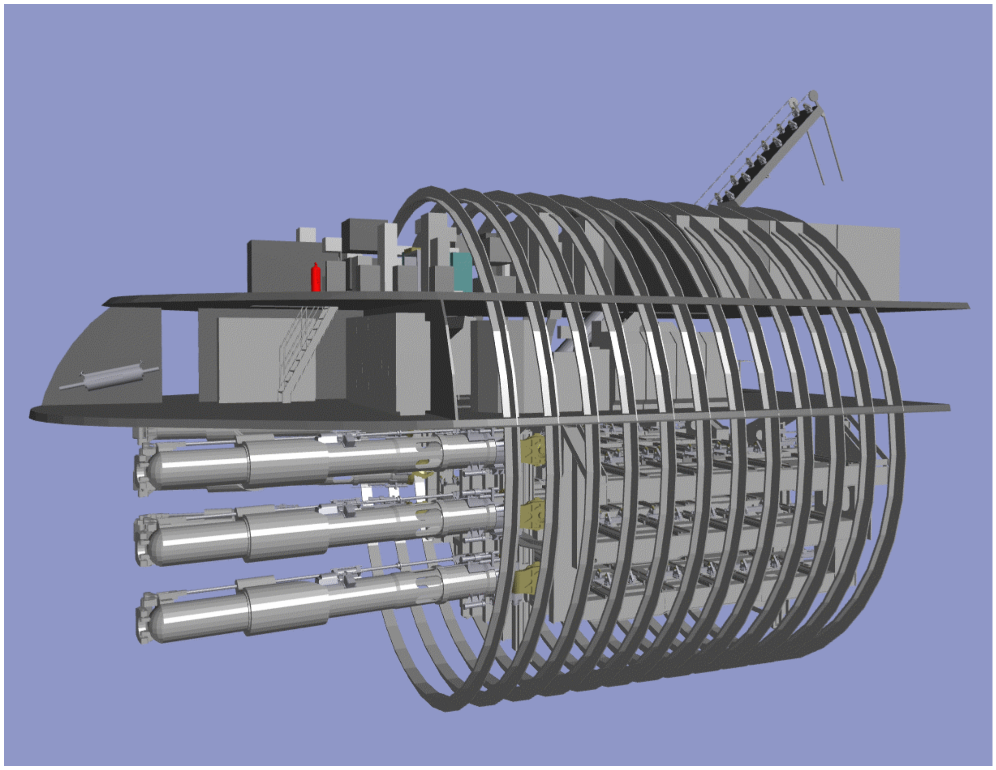


Figure 23: The Torp model comprises 698,872 triangles.

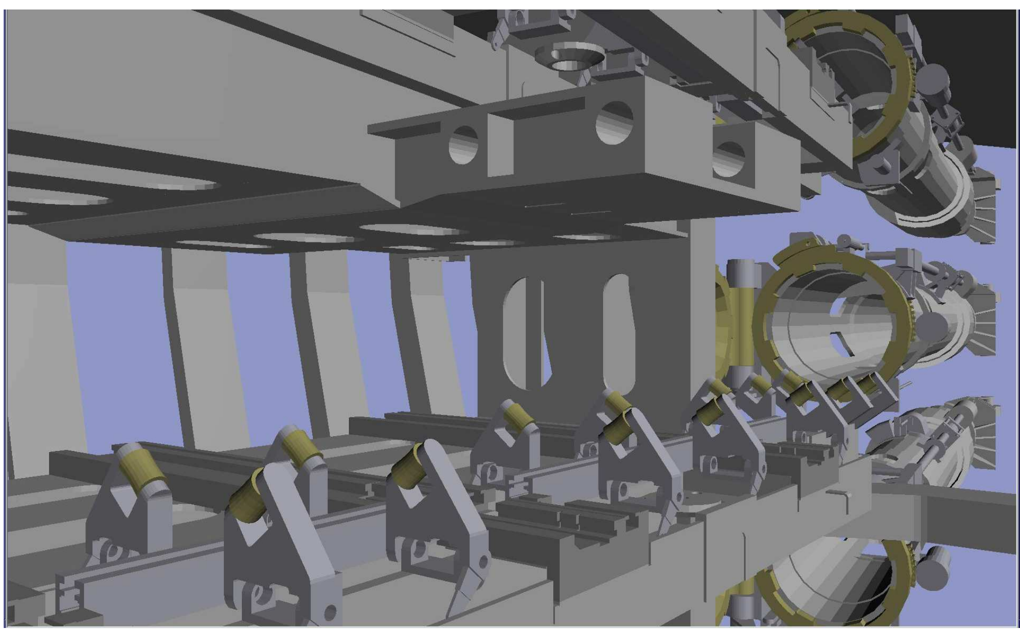


Figure 24: A close-up of the Torp model, showing the port pivot structures and torpedo tubes. Much of the model's complexity is in these rollers.

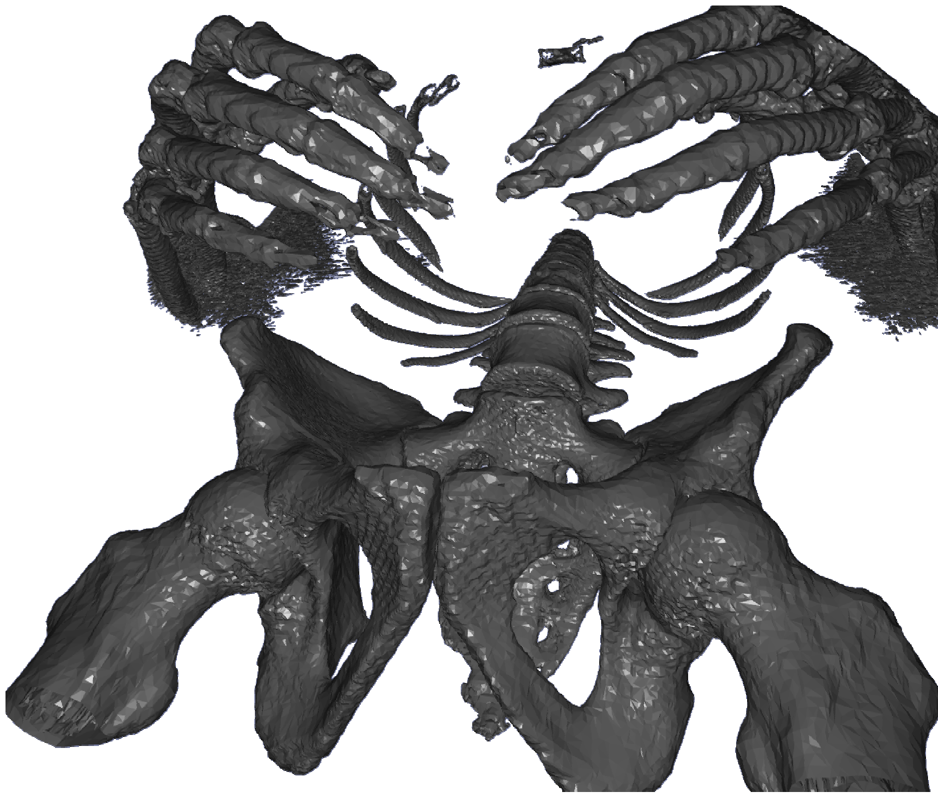


Figure 25: The Bone6 model is the bone-tissue interface from part of the Visible Man volumetric dataset. The initial Marching Cubes model was partially simplified by the decimation algorithm (see Section 2.2.2) but still contains over 1.1 million triangles.

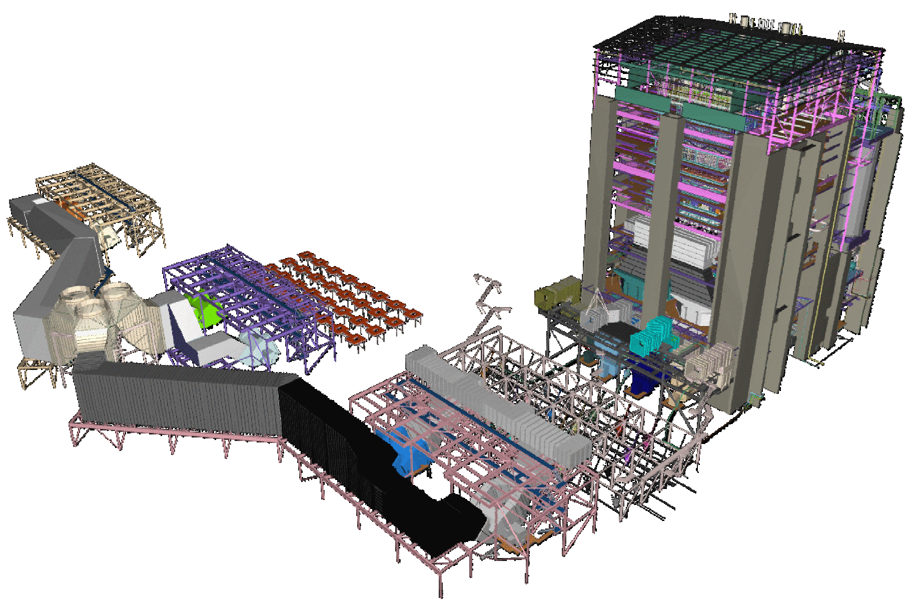


Figure 26: The Powerplant_4M model is a 3,879,736-triangle subset of the full powerplant model.

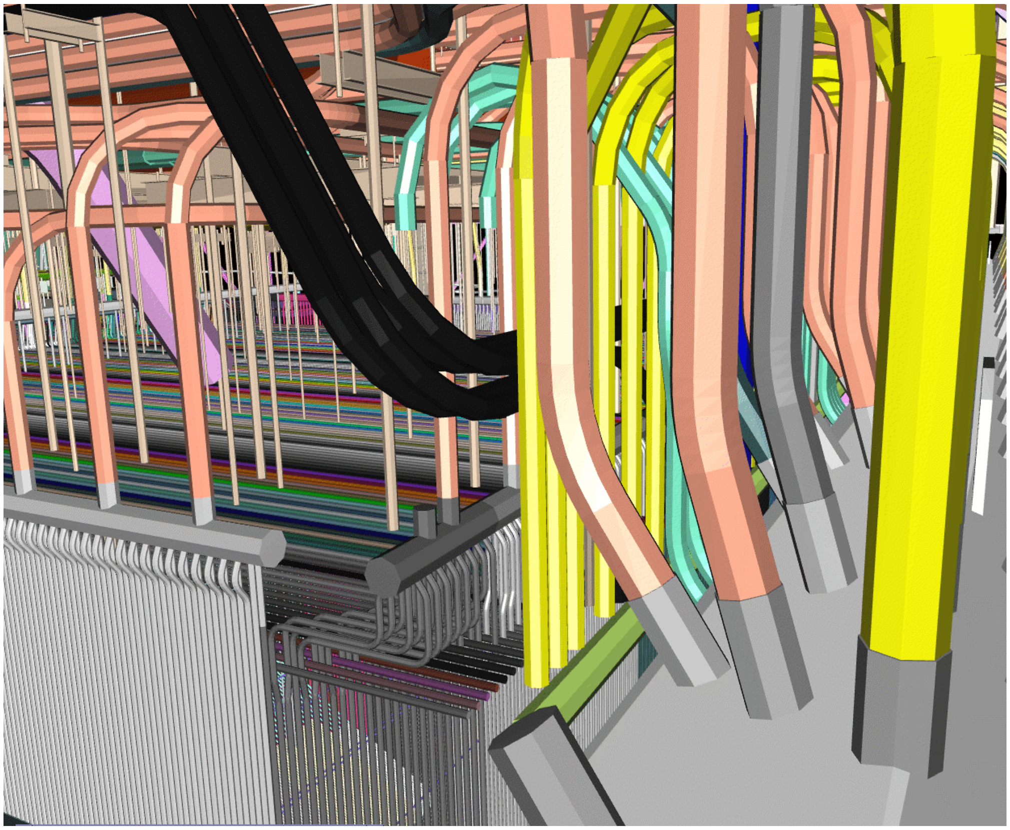


Figure 27: Interior view of the Powerplant_4M model. Piping accounts for most of the complexity of the model, as this view into the 46^{th} level of the main structure shows.

7.3. The Path

For consistency, all results presented below use a single 700-frame canonical path, captured from a flythrough of the Torp model by recording the view matrix used each frame. The path attempts to provide a variety of viewing conditions. Initially, the viewpoint is well outside the model, looking into the center as the viewpoint orbits the model in the X-Y plane. The viewpoint then dives into the interior of the model, inspecting the pivot structures and torpedo tubes shown in Figure 24, then tilting up and flying up through the model to inspect the torpedo loading mechanism at the top of the submarine. Most of the model is out of the field of view at this point. The viewpoint then tilts down to look into the model as it rises further to bring most of the model into view again, this time from above. The viewpoint then dives down into the model again, once more inspecting the structures in the lower level of the torpedo room. Finally, the viewpoint backs out of the model, moving backwards until the entire model is once more in view. The same path was applied to models other than Torp by scaling the viewpoints along the path according to the size of the model. Although some of the motions along the path no longer resemble realistic viewing actions on that model (for example, the viewpoint tends to fly through objects rather than around them), scaling the path to match each model enables a direct comparison of the run-time performance of HDS on multiple models.

7.4. Visual Results

This section shows a few examples of the HDS system in action, comparing some of the original models to run-time simplifications created with various screenspace error tolerances.

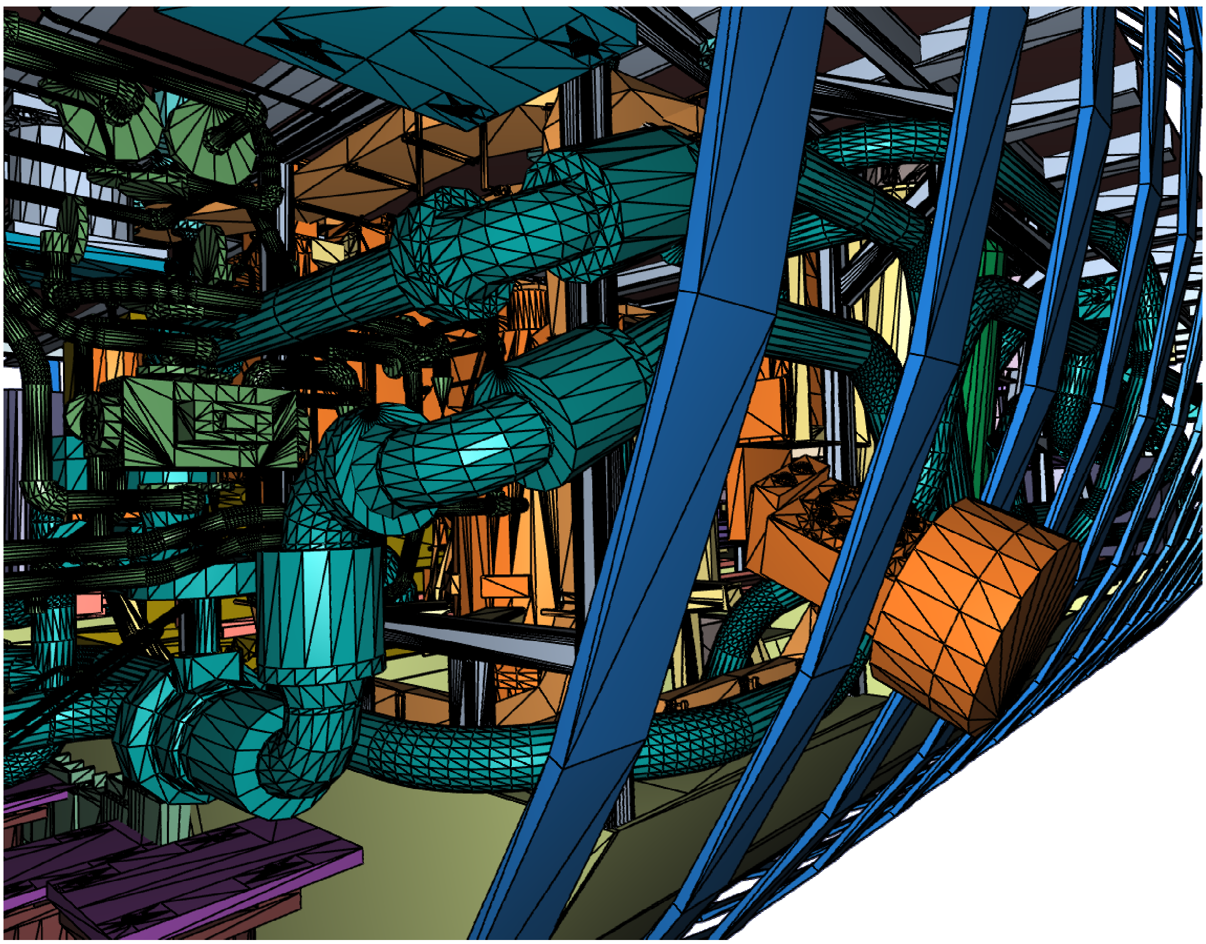


Figure 28: The AMR shown at original resolution. The entire model has 504,969 faces.

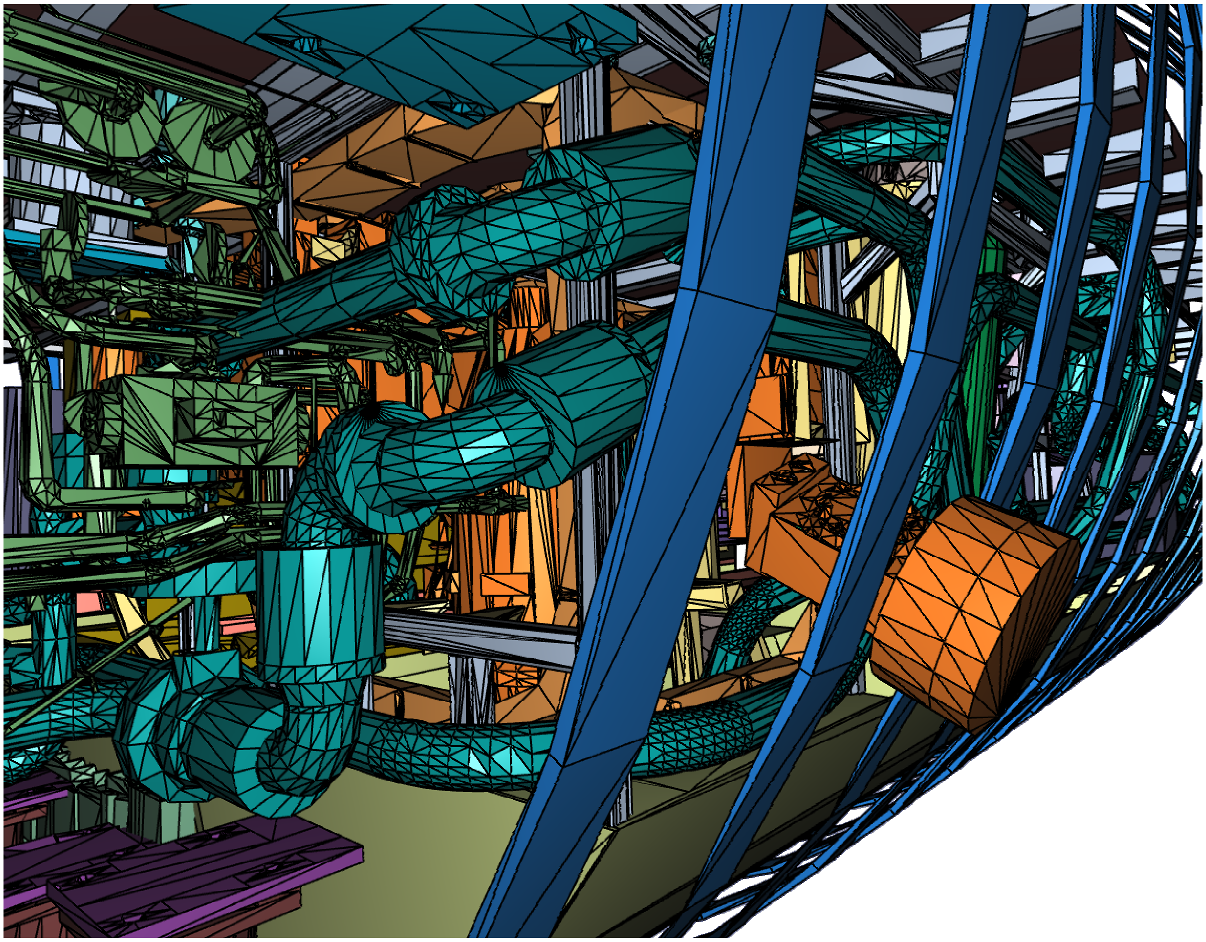


Figure 29: The AMR model at 0.7% screenspace error tolerance (123,106 faces). This level of run-time simplification filters out much geometry that is redundant for all but the closest views, as seen in the green piping on the left.

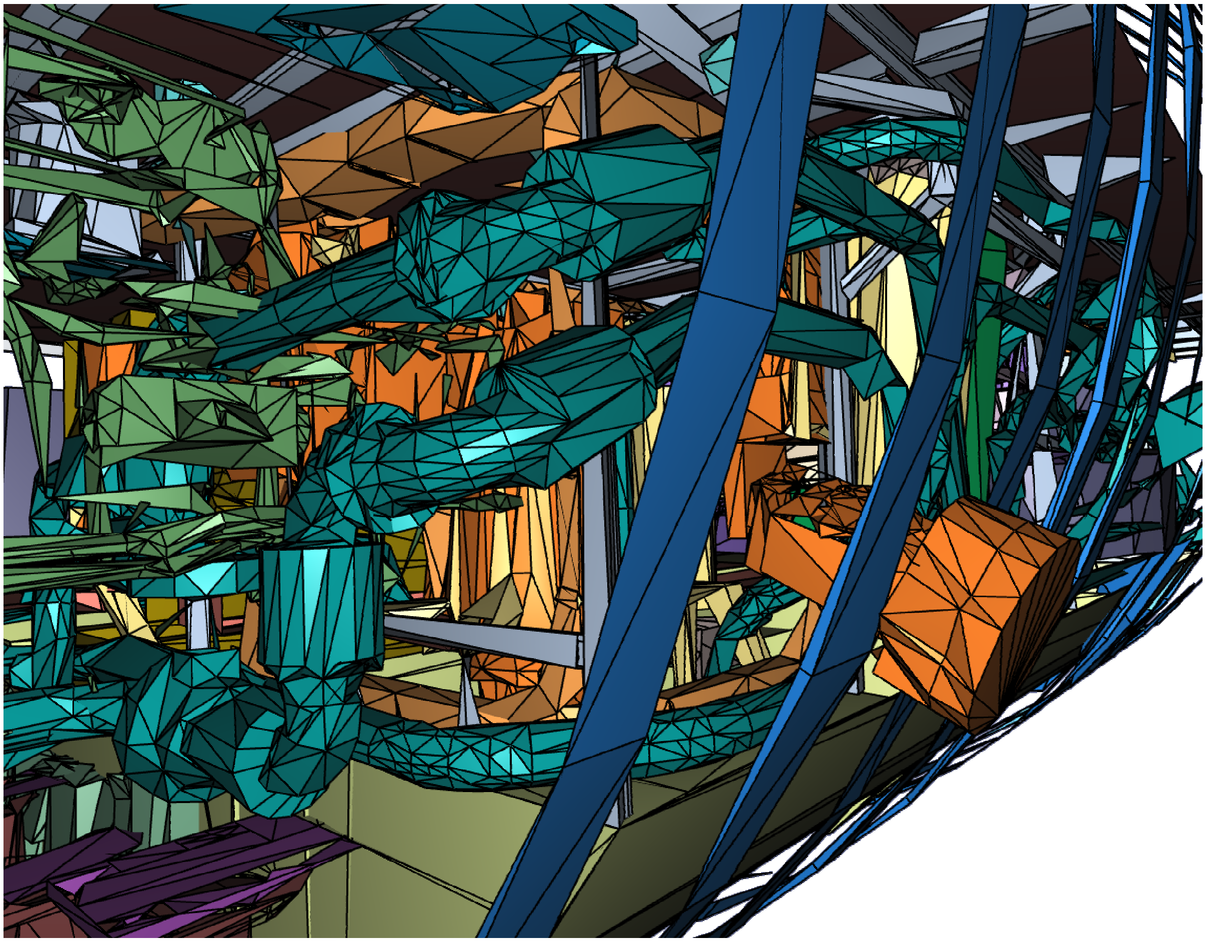


Figure 30: The AMR model at 2.5% screenspace error tolerance (34,128 faces). This level of simplification begins to introduce visual artifacts, but preserves the broad shape of objects in the scene using a tenth as many polygons as the original model.

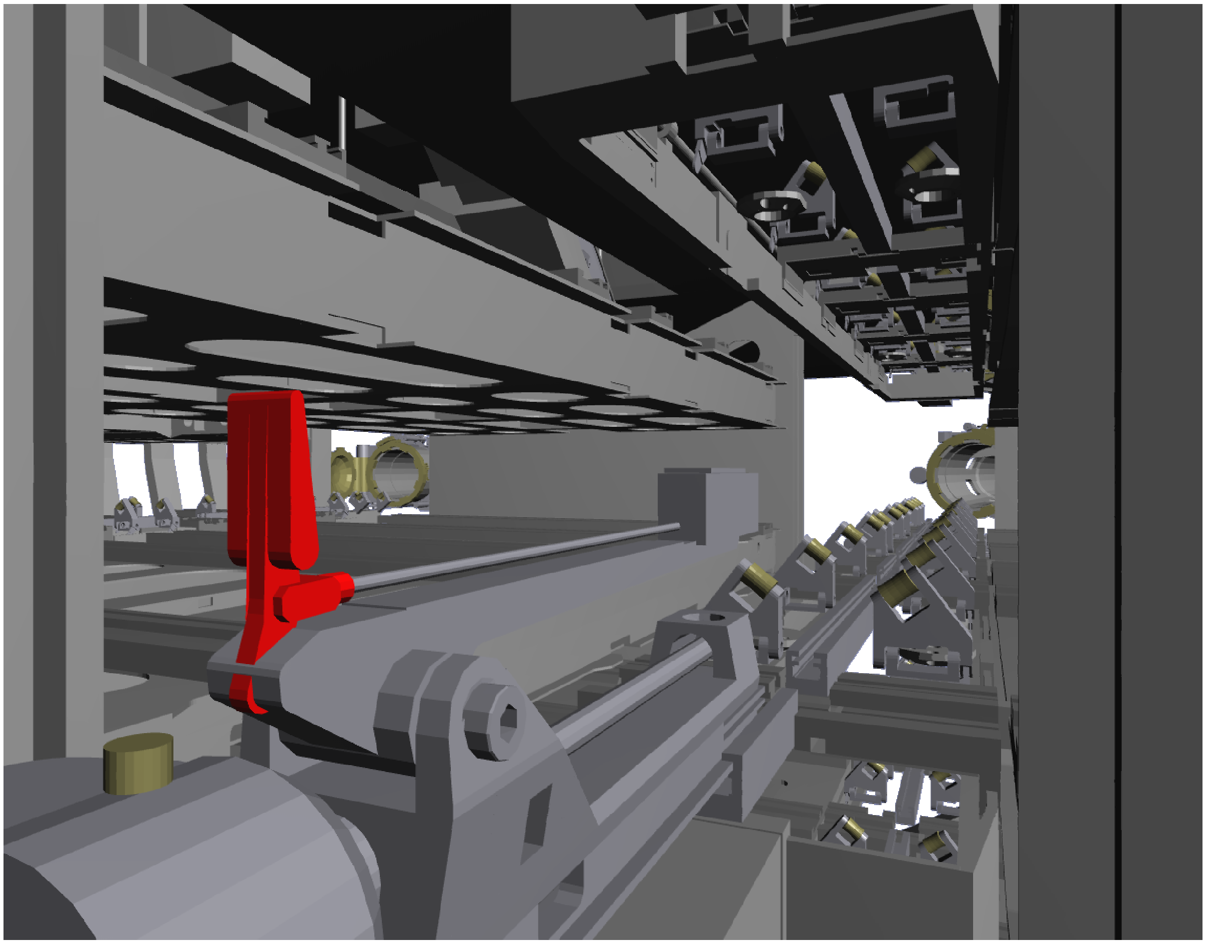


Figure 31: The Torp model at original resolution, comprising 698,872 faces in total.

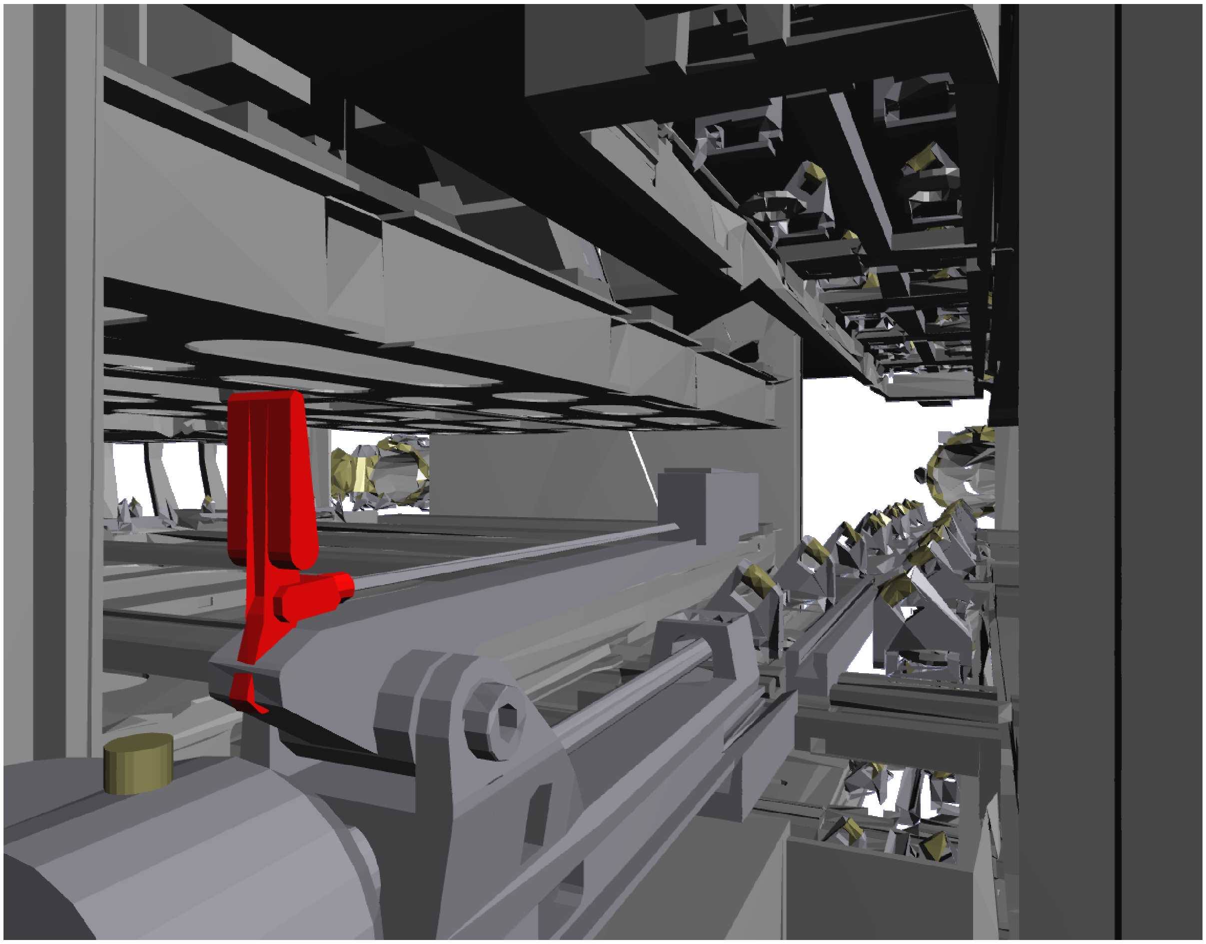


Figure 32: The Torp model shown at 0.8% screenspace error tolerance (129,446 faces). Most visual artifacts are still reasonably subtle.

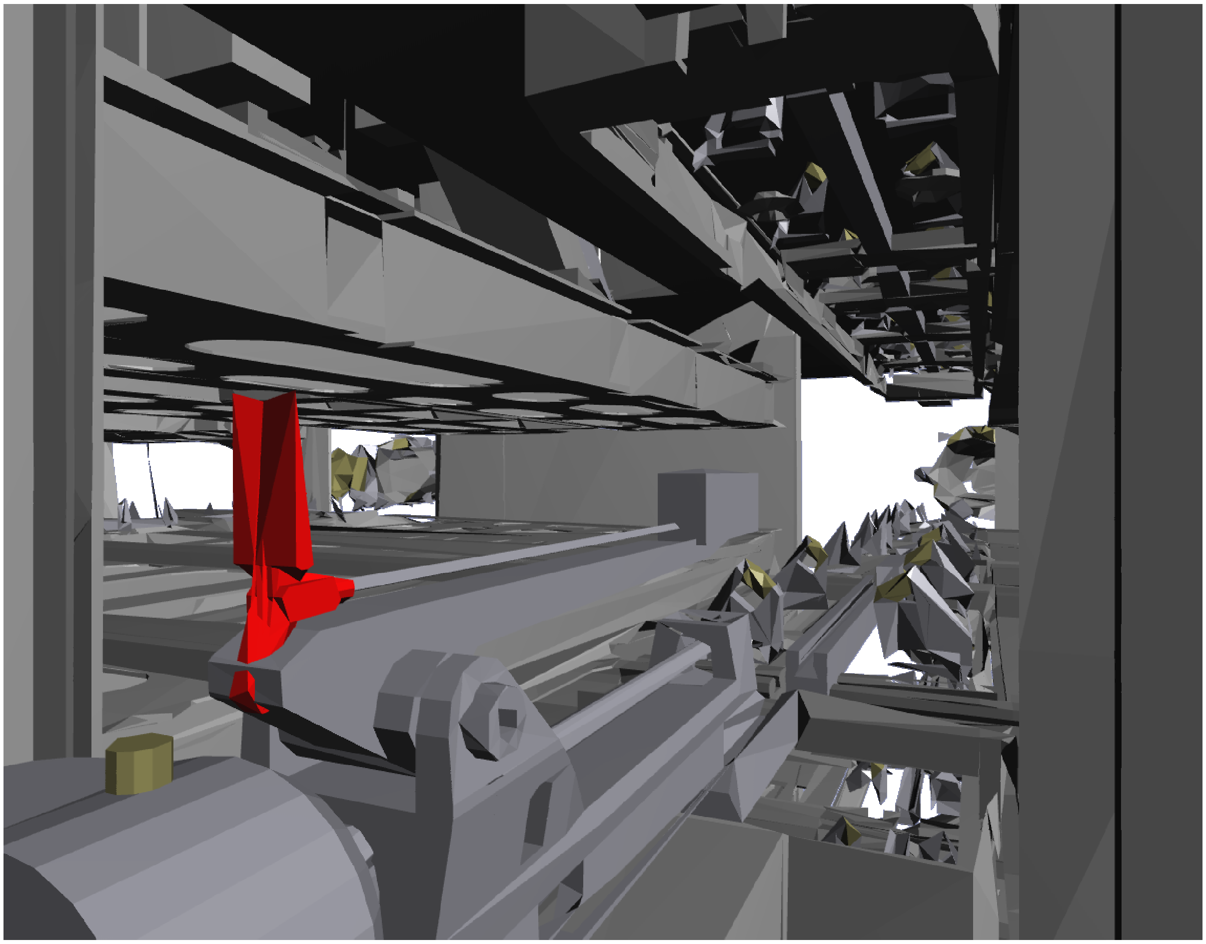


Figure 33: The Torp model shown at 1.5% screenspace error tolerance (76,404 faces). Notice that distant objects are simplified to almost schematic levels, while nearby features such as the circular yellow knob on the left still possess reasonable fidelity.

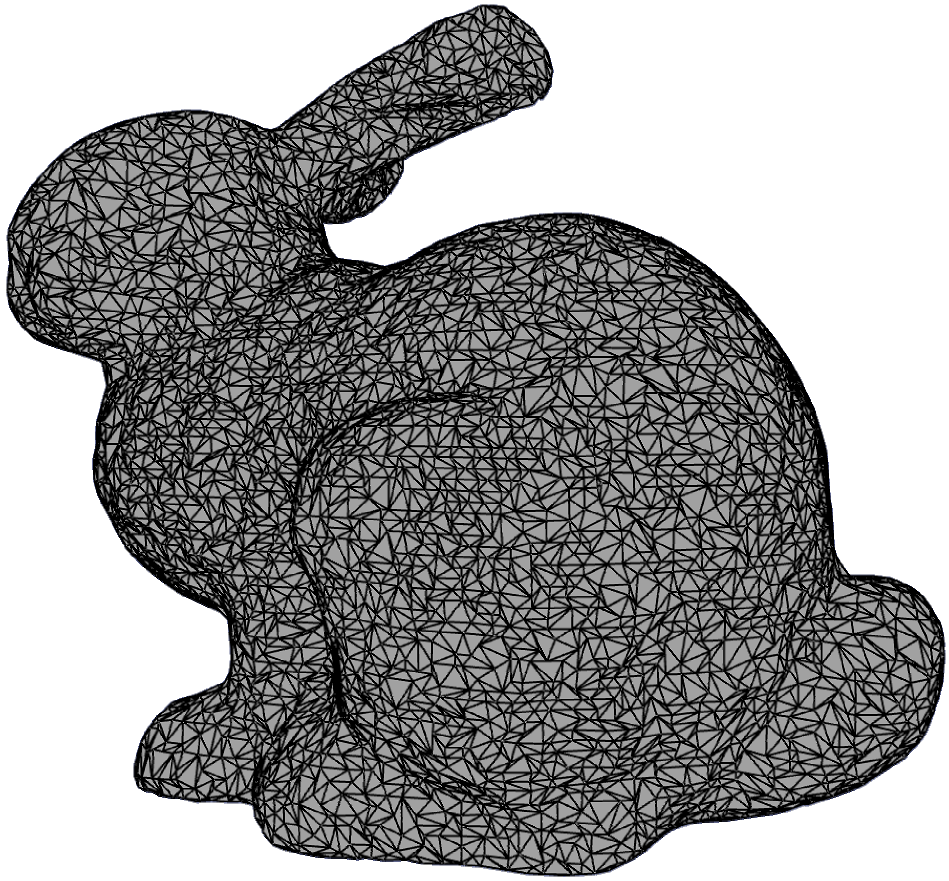


Figure 34: The Bunny model shown simplified to a 1% screenspace error tolerance (19,598 faces).

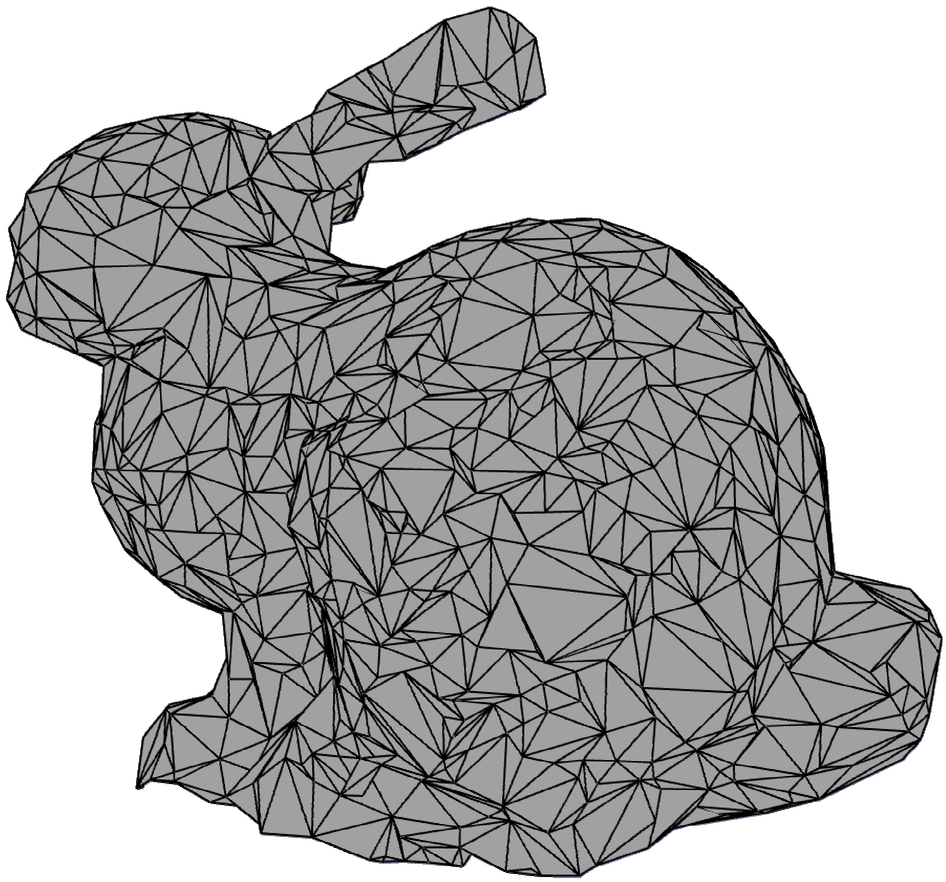


Figure 35: The Bunny model shown simplified to a 5% screenspace error tolerance (2,901 faces).

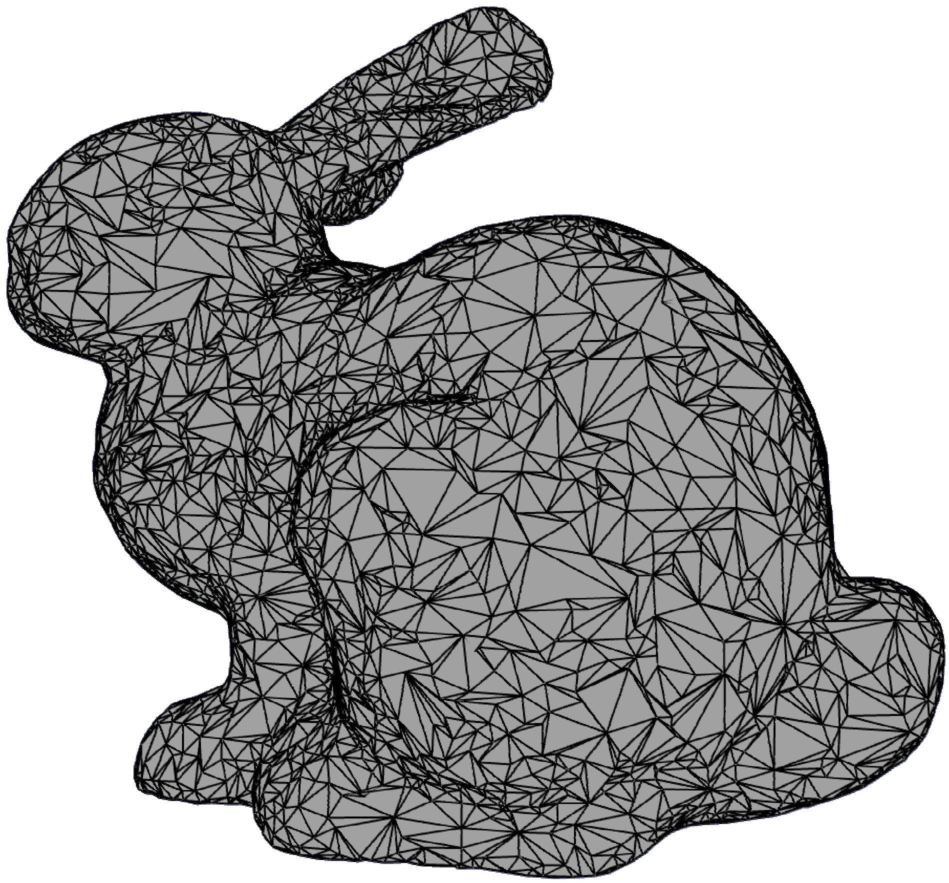


Figure 36: The Bunny model shown with silhouette preservation. Silhouette nodes are tested against a 1% screenspace error threshold and interior nodes are tested against a 5% error threshold. The simplification comprises 13,135 faces.

7.5. Run-time Performance

This section presents plots illustrating the run-time performance of HDS on the Torp and Cassini models. Charts plot the simplification time, rendering time, and number of triangles displayed along the canonical path. Plots are given for each model running under screenspace error thresholds of 1% and 5% of the vertical field of view. Since these runs were all done with a 640x480-pixel viewport, these correspond to 5-pixel and 25-pixel error tolerances.

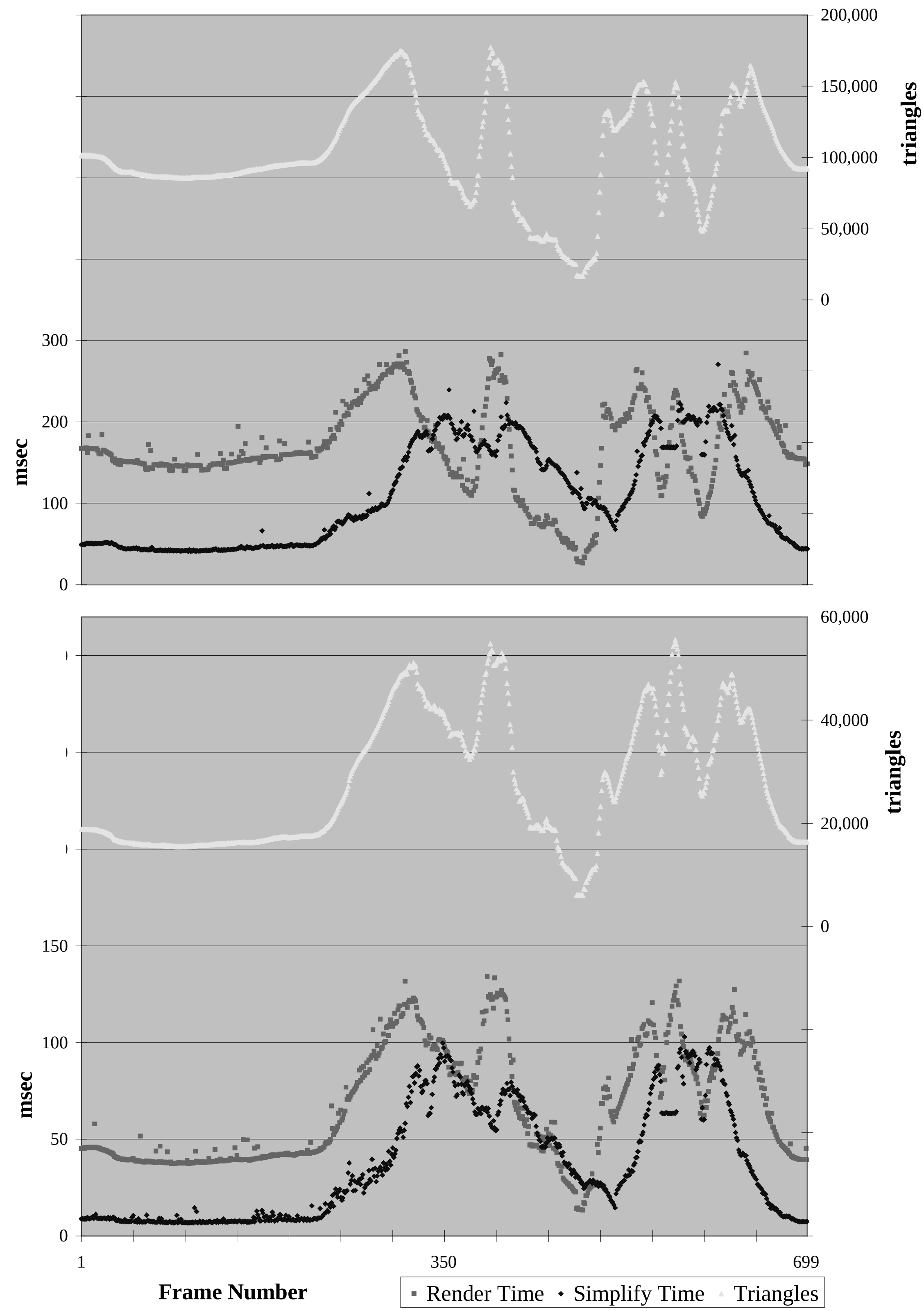


Figure 37: Timings and triangle count for the Torp model at 1% (top) and 5% (bottom) screenspace error thresholds during the course of the canonical path.

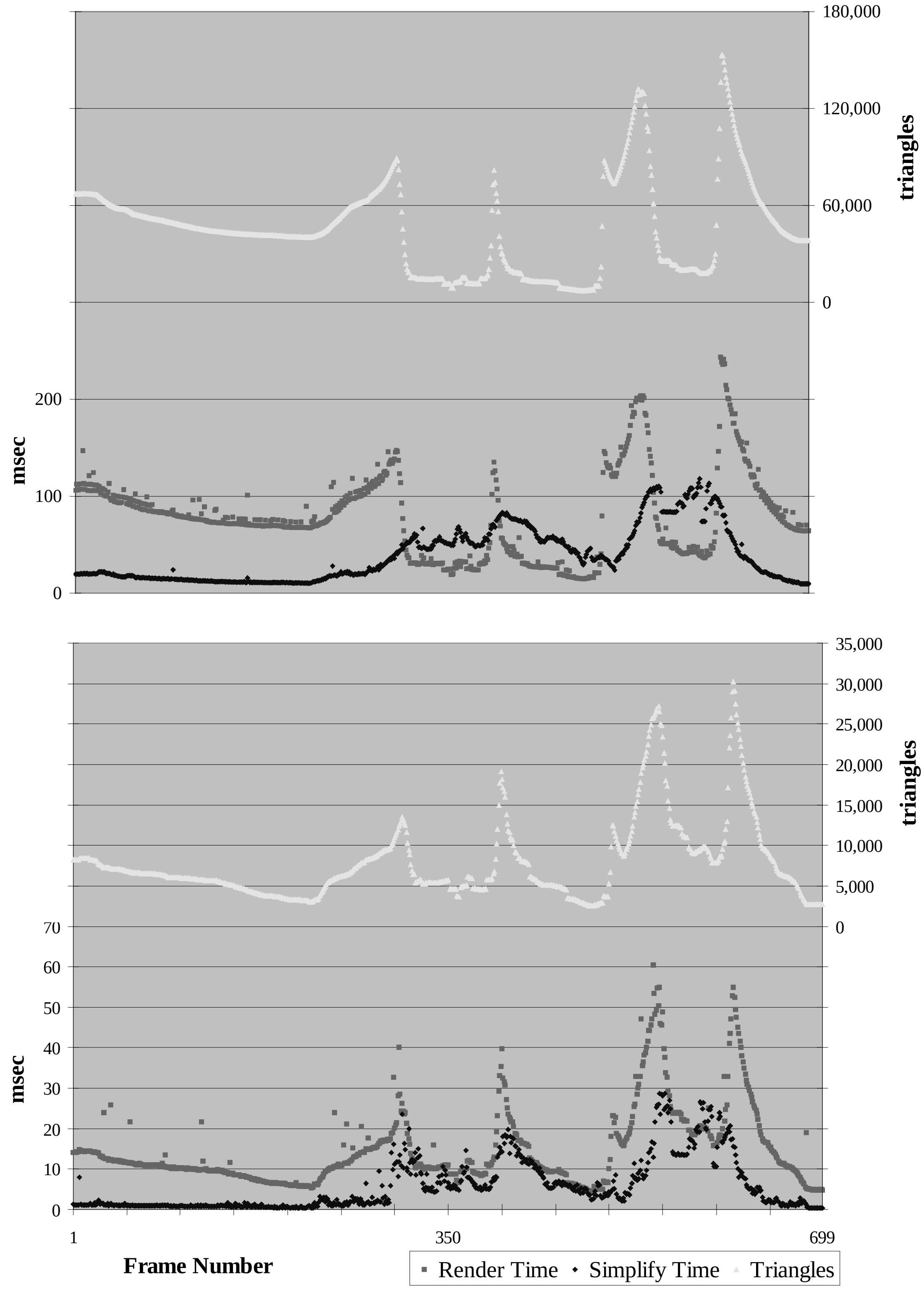


Figure 38: Timings and triangle count for the Cassini model at 1% (top) and 5% (bottom) screenspace error thresholds during the course of the canonical path.

7.6. Vertex Tree Characteristics

This section examines the vertex trees created by the HDS tight octree preprocess for the Cassini and Bone6 models; the Appendix contains data for the remaining models. Tables break down the vertex trees by depth, describing for different levels of the tree:

- The number of nodes at that level.
- The number of leaf nodes at that level.
- The mean degree of all nodes at that level, that is, the average number of children per node, along with the standard deviation of that mean.
- The mean degree of interior nodes at that level, along with the standard deviation.
- The average number of tris per node.
- The average number of subtris per interior node (since by definition only interior nodes have subtris).

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	7.00		7.00		0	82.00
1	7	0	4.57	1.27	4.57	1.27	2475.57	46.14
2	32	0	6.03	1.84	6.03	1.84	914.25	26.53
3	193	24	5.04	2.45	5.76	1.64	243.25	21.30
4	973	92	4.39	2.42	4.85	2.06	88.92	15.51
5	4,275	999	3.39	2.51	4.42	1.91	39.83	10.49
6	14,486	4,997	2.64	2.42	4.04	1.82	21.52	7.74
7	38,292	17,579	1.88	2.11	3.47	1.65	13.56	5.58
8	71,923	44,963	1.13	1.68	3.02	1.35	9.91	4.10
9	81,503	63,084	0.57	1.15	2.54	0.93	8.26	2.79
10	46,827	41,354	0.26	0.76	2.27	0.59	7.44	1.50
11	12,399	10,004	0.43	0.90	2.21	0.48	5.36	0.97
12	5,286	4,398	0.36	0.80	2.12	0.33	3.17	0.73
13	1,880	1,757	0.13	0.50	2.02	0.13	2.74	0.56
14	248	246	0.02	0.18	2.00	0.00	2.52	1.00

Table 3: Vertex tree statistics for the Cassini model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	20.00
1	8	0	6.88	1.13	6.88	1.13	1471.50	26.88
2	55	0	6.45	1.36	6.45	1.36	631.60	24.20
3	355	1	6.78	1.41	6.80	1.36	234.71	23.05
4	2,406	133	5.93	2.21	6.28	1.73	85.75	16.38
5	14,268	1,817	4.79	2.61	5.49	1.99	33.65	10.70
6	68,359	16,873	3.28	2.45	4.35	1.82	15.07	6.74
7	223,891	109,314	1.58	1.75	3.08	1.17	8.61	3.90
8	352,814	290,424	0.42	0.93	2.35	0.61	6.49	2.50
9	146,927	143,383	0.05	0.34	2.17	0.42	5.93	1.99
10	7,705	7,683	0.01	0.11	2.00	0.00	5.72	1.86

Table 4: Vertex tree statistics for the Bone6 model.

Examining the vertex tree statistics for the various models provides some interesting insights. The complexity of the model affects the height of the vertex tree, since each leaf node represents a single unique vertex and the interior nodes can have at most eight children under the tight-octree clustering scheme. The vertex tree for the Cassini model, however, is four levels deeper than the vertex tree for the Bone6 model, which has three times as many vertices. The vertex trees for the AMR and Torp models are similarly deeper than the vertex tree for the Bone6 model, despite the greater complexity of the latter. The reason lies in the regular structure of the Bone6 model, which was extracted as a volumetric isosurface by the Marching Cubes algorithm. Marching Cubes samples the volume at regular intervals called *voxels*, creating isosurfaces with vertices spaced at regular intervals. No two vertices are ever closer than the length of this interval. This regularity lends itself nicely to the tight octree subdivision, which never has to divide the vertices too many times to isolate each vertex.

The Cassini model, by contrast, has a high *dynamic range*; that is, some high-detail regions of the model are very densely populated with vertices and other low-detail regions of the model are sparsely populated. The tight octree must therefore subdivide further in the high-detail regions, creating a deeper vertex tree. In some sense, then, the handcrafted Cassini, Torp, and AMR models with their high dynamic range are *more* complex than the computer-generated Bone6 model with its regularly spaced triangles, despite the polygon counts of the various models. The issue of dynamic range and its relationship to model complexity seems an interesting point for future reflection.

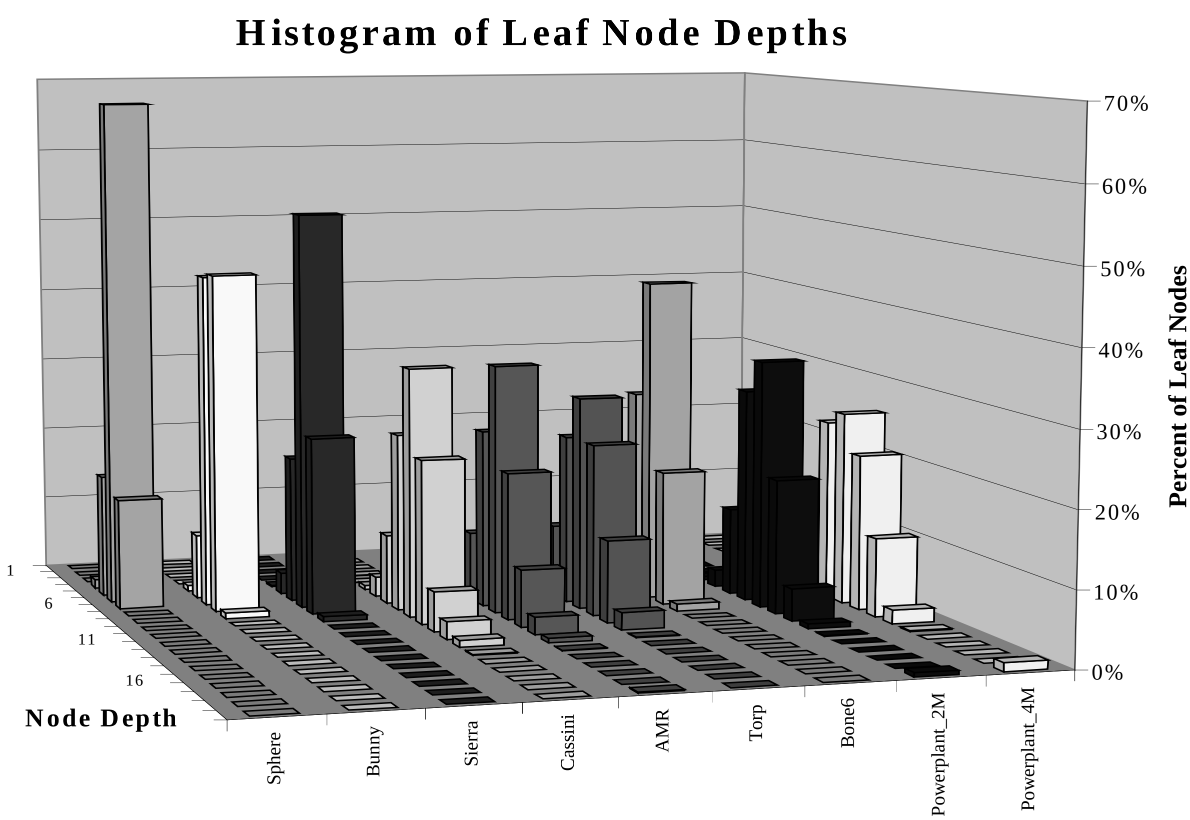


Figure 39: A histogram of the depths of leaf nodes in the vertex tree for each model.

Figure 39 presents for each model a histogram showing what percent of its leaf nodes occur at each level of the vertex tree. This captures in a sense the average depth of the tree and the spread of depths across the tree. The histogram reflects somewhat how such factors as dynamic range affect the construction of the vertex tree. Large spikes in the histogram mean that the tight octree subdivision resolved most vertices into unique nodes at the same depth. This implies a highly regular model. The evidence supports this interpretation: Sphere, for example, is the simplest and most regular model tested and, with 68 percent of the vertices resolved at level 6, shows the largest spike in the histogram. Also showing spikes are Sierra (a uniformly sampled height field), Bunny (stitched together from dense, uniformly sampled laser scans), and Bone6 (a Marching Cubes isosurface from regularly spaced voxels). The five CAD models, on the other hand, all have a high dynamic range and show correspondingly more spread in the histogram.

Figure 39 also helps support the assertion that the vertex trees tend to be reasonably well balanced for real models. A tree in which all the leaf nodes fell within two adjacent levels would certainly be considered well balanced. As the histogram helps show, 70% or more of the leaf nodes occur within three levels even on the widely spread CAD models. Section 7.9 will expand on this observation.

7.7. Preprocessing Performance

Table 5 summarizes the preprocessing performance of HDS on all of these models. The models marked with an asterix were too big to preprocess on the testbed machine used for the rest of the results, and were instead preprocessed on a slower machine with more memory. That machine is a Silicon Graphics Onyx with 4 250 MHz R4400 processors and 2,048 megabytes of main memory. In general this machine took about twice as long to preprocess a model as the faster R10000-processor testbed. Note that the times given here measure "wall clock" time rather than CPU time, and that the I/O operations of reading the original model and writing the finished vertex tree are not included in this table. Also note that the hybrid preprocessing algorithm was developed principally to test silhouette preservation and was not optimized for speed at all.

Model	Vertices	Triangles	Nodes in tight octree	Storage (using gzip)	Preprocessing time (Tight octree)	(Hybrid)
Sphere	4,098	8,192	5,627	0.5 Mb	0.5 seconds	2.5 minutes
Bunny	35,947	69,451	50,856	5 Mb	7.2 seconds	20 minutes
Sierra	81,920	162,690	119,824	12 Mb	22 seconds	—
Cassini	189,615	415,257	278,329	30 Mb	75 seconds	—
AMR	280,544	504,969	394,253	32 Mb	86 seconds	—
Torp	411,778	698,872	621,791	48 Mb	131 seconds	87 minutes
Bone6	569,685	1,136,785	816,833	82 Mb	254 seconds	—
Powerplant_3M	1,303,162	2,796,984	2,031,880	160 Mb	866 seconds *	—
Powerplant_4M	1,794,082	3,879,736	2,930,501	226 Mb	975 seconds *	—

Table 5: HDS Preprocessing times for the various models.

Figure 40 shows a plot of these preprocessing times versus the number of vertices in each model. Though more data points would be desirable, the plot does show the characteristic O(n log n) curve, as expected.

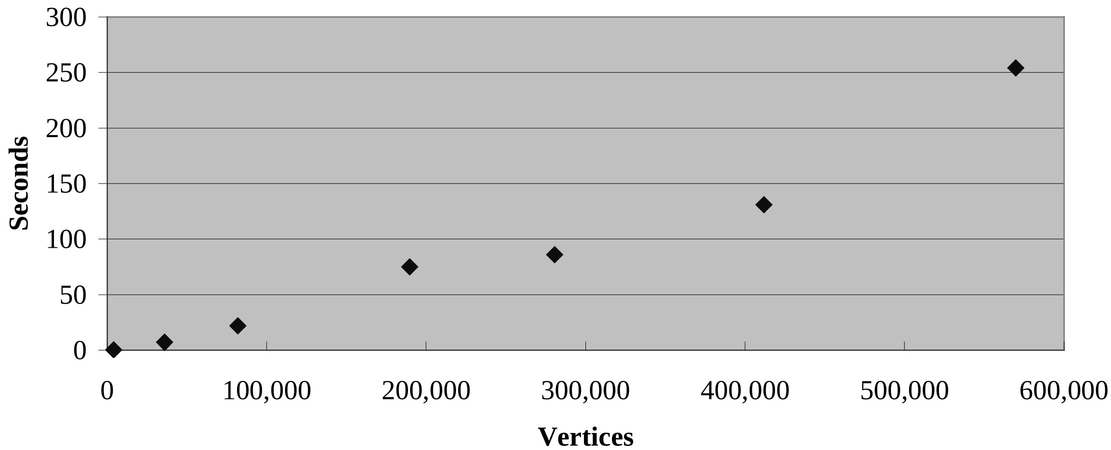


Figure 40: Preprocessing time in seconds vs. number of vertices.

7.8. Artifacts

This section describes two distracting visual artifacts that HDS can introduce: *dropouts* and *mesh folding*. Ways of avoiding both types of artifacts are also discussed.

7.8.1. Dropouts and how to avoid them

A straightforward implementation of the asynchronous simplification scheme presented in Section 5.5 is relatively easy to code on a shared-memory multiprocessor system, but care must be taken to avoid dropouts. Characterized by triangles that disappear for a frame, these transient artifacts occur when the RENDER process sweeps through a region of the active list being affected by the SIMPLIFY process. For example, the **`foldNode()`** operation removes triangles and fills in the resulting holes by adjusting the corner positions of neighboring triangles. If those neighboring triangles have already been rendered during the frame when **`foldNode()`** adjusts their corners, but the triangle to be removed has not yet been rendered, a hole will appear in the mesh for that frame.

Dropouts are fundamentally caused by failure to maintain a consistent shared database in an asynchronous system. They are difficult to eradicate with simple locking schemes. Locking the triangles to be affected before every **`foldNode()`** and **`unfoldNode()`** operation will not suffice, since the triangles may not be near each other in the active triangle list. Since the active triangle list is divided among the high-level nodes for culling purposes, another possibility would be to lock all the nodes affected by the fold or unfold operation[5]. This strategy prevents dropouts, but proves prohibitively expensive in practice.

The *update queue* provides one solution to the dropout problem. The update queue was motivated by the observation that the time spent performing **`foldNode()`** and **`unfoldNode()`** operations is a small fraction of the time taken by the SIMPLIFY process to traverse the vertex tree and determine which nodes to fold and unfold. Rather than actually performing the updates, the SIMPLIFY process accumulates them into the update queue, marking the node Dirty and placing a Fold or Unfold entry in the queue. At the beginning of every frame the RENDER process performs the updates in the queue, folding or unfolding each node before marking it Clean again[6]. All changes to the active triangle list take place as a batch before any triangles are rendered; the shared database is thus kept consistent and dropouts are eliminated.

---

[5] This turns out to be the subtree rooted at **`N->container`**, where **`N`** is the affected node.

[6] Implementation detail: to help smooth out the transition caused by drastic changes in viewpoint, the RENDER process takes only the first 1000 updates each frame.

7.8.2. Mesh folding and how to avoid it

Mesh folding occurs when shifting the position of a vertex causes an attached triangle to flip in orientation (Figure 41). Mesh folding artifacts are inherent to any vertex-merging scheme that does not take care to avoid them, including edge-collapse approaches that merge only two attached vertices at a time. Sometimes better placement of the merged vertex solves the problem, but sometimes merging the vertices will cause a mesh folding artifact regardless of the placement of the new vertex.

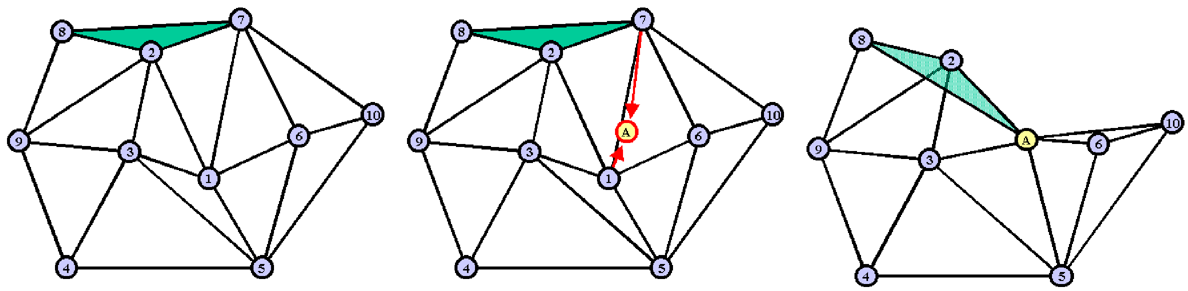


Figure 41: An example of mesh folding. When vertices 1 and 7 are merged to form vertex A, the shaded triangle folds over neighboring triangles, flipping in orientation. Note that placing vertex A closer to vertex 7 would avoid the problem in this example.

How mesh folding artifacts show themselves visually depends on the rendering parameters. Folding a triangle flips its orientation, so such triangles may not be drawn if backface culling is enabled. If two-sided lighting is enabled, the triangle will be drawn, but since flipping a triangle negates its normal vector, the folded triangle may be shaded differently from the surrounding mesh. In the current system, mesh folding artifacts typically appear as small dark slivers in the simplified mesh. Since they are caused by vertex merging, the artifacts are never larger than the screenspace error threshold, which regulates the maximum distance vertices can move during a merge operation.

Constructing the vertex tree carefully can reduce the likelihood of mesh folding, but to eliminate folding artifacts in a view-dependent system requires modifying the view-dependent simplification criteria. For example, a check could be added to ensure that folding a node did not flip the orientation of any triangles supported by the node, and disallowing any folding operations that failed the check. Since the artifacts are small, and since adding such checks to the view-dependent criteria might overly restrict simplification, the current implementation of HDS does not attempt to prevent mesh folding. If high fidelity were a concern, however, adding code to prevent these artifacts would certainly be worthwhile.

7.9. Preprocessing Complexity

In theory, the complexity of polygonal models that HDS might encounter as input is $O(n^3)$ with respect to the number of vertices, for every set of three vertices in the model could form a triangle. Of course, such a model would be an incoherent jumble of intersecting triangles, and in practice, nobody seems likely to want to visualize such a model. For polyhedral models exhibiting manifold topology, on the other hand, the number of triangles is linearly related to the number of edges and vertices by Euler's formula: $V - E + F = 2 - 2g$. Here V is the number of vertices in the model, E the number of edges, F the number of faces, and g the genus of the model.

Although real-world CAD models often have many topological degeneracies, it seems reasonable to assume that on the whole the number of triangles in such models will still grow approximately linearly with the number of vertices. Figure 42 supports this assertion with data from the nine sample models, fitting a straight line to a plot of vertices vs. triangles. This simplifies the complexity analysis of the algorithm considerably, since the numbers of edges, vertices, and triangles can be used interchangeably in asymptotic analysis. In the following paragraphs n represents the number of edges, vertices, or triangles.

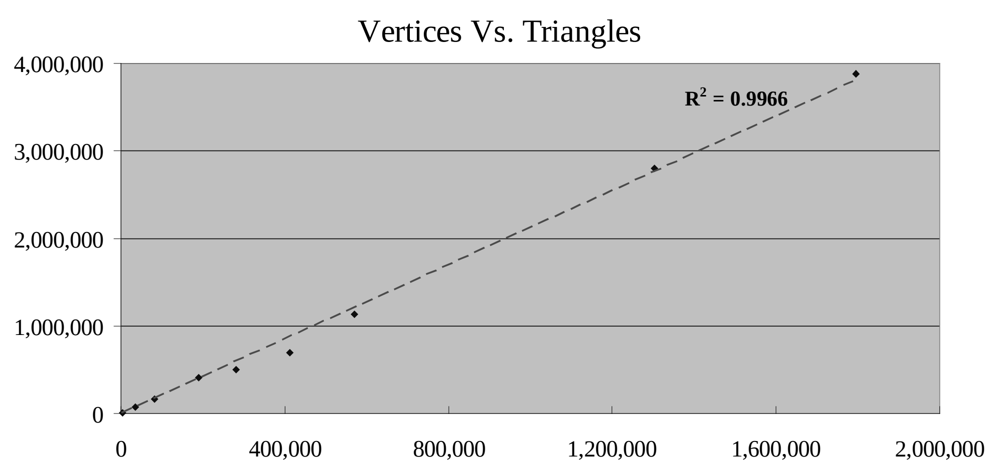


Figure 42: A plot of vertices vs. triangles for the sample models.

7.9.1. Constructing the vertex tree

For the moment, assume that the vertex tree is well balanced; the next section will address this assumption. Since each leaf node represents a unique vertex, a balanced tree will have O(n) leaves, O(n) total nodes, and O(log n) levels. If each node requires constant memory, the complete vertex tree would require O(n) memory. Unfortunately, different nodes require different amounts of storage, since the number of tris and subtris varies with the number and connectivity of a node's vertices. How many tris and subtris will each node have?

Each triangle appears exactly once as a subtri of a node in the vertex tree, so a total of O(n) subtris are scattered about the vertex tree. Each triangle appears in the **`node->tri`** lists of multiple nodes, however. Recall that the triangles in the **`node->tri`** list of a node are those triangles that have exactly one corner supported by that node. Since every leaf node represents a unique vertex, and every triangle has three vertices for corners, every triangle will appear as a tri of at least three nodes. In the models tested, triangles appeared in the **`node->tri`** list of about six nodes on average. In practice, then, it is reasonable to assume that the tris and subtris of nodes in the vertex tree take O(n) storage space, and thus do not contribute asymptotically to the space requirements of the vertex tree.

The time to construct the vertex tree depends on the algorithm used, but it clearly cannot be less than O(n), since that is how many nodes must be created. The tight-octree scheme can be shown to run in O(n log n) time, since at each node of the octree the subdivision algorithm must traverse all of the vertices falling within that node and allocate them among the node's children. Starting at the root node, then, the subdivision process must traverse every vertex in the model once for every level of the tree. A reasonably well balanced tree, which we are assuming at the moment, has O(log n) levels, so building the complete tree takes O(n log n) time. Once the vertex tree is built, a single pass is made over the O(n) triangles in the model. Each triangle is assigned to the **`node->subtri`** list of only one node, but can in the worst case be assigned to the **`node->tri`** lists of O(log n) nodes. Assigning tris therefore takes O(n log n) time, so the total tight-octree vertex tree construction process takes O(n log n) time and space.

7.9.2. Eliminating assumptions

The above arguments make two basic assumptions: first, that the vertex tree is reasonably well balanced, and second, that each triangle appears on average in the **`node->tri`** lists of an approximately constant number of nodes. On the whole, both assumptions seem to hold well in practice. The histogram and charts in Section 7.6 and the Appendix show that the majority of leaves occur within three levels of the tree, and that the average triangle appears in the **`node->tri`** lists of 5-7 nodes, regardless of model size. However, they remain assumptions and could conceivably fail. This section addresses how using more complex structures and methods could obviate these assumptions.

A number of vertex clustering schemes could be used to guarantee a balanced vertex tree. A K-D tree, for example, is a very general subdivision scheme which recursively divides *d*-dimensional space into *k* subspaces by applying *k–1* axis-aligned *splitting planes*. To which axis the splitting planes are aligned can be chosen on a node-by-node basis, or can simply alternate by level (e.g., split along the X axis, then the Y axis, then the Z axis, etc.). The placement of the splitting can also vary, which enables the construction of balanced vertex trees. By placing the splitting planes so as to divide vertices equally among subnodes, the K-D tree construction can allocate roughly the same number of vertices to every node at a particular level of the tree. The tree will thus be balanced. Similar strategies can ensure that other structures such as a *binary space partition* tree (BSP tree) or an *oriented bounding box* tree (OBB tree) are also well balanced.

Such approaches suffer the disadvantage that they ignore the *aspect ratio* of the vertex clusters created. Very long and skinny boxes (or convex polyhedra, in the case of BSP trees) may be created that will be severely overestimated by the bounding spheres used to calculate the screen-space error introduced by folding a node. Furthermore, subdivision criteria that strictly enforce balanced-tree creation may sacrifice goodness of fit (Figure 43). In such cases the tree will be balanced, but the overall fidelity of the simplifications will usually suffer.

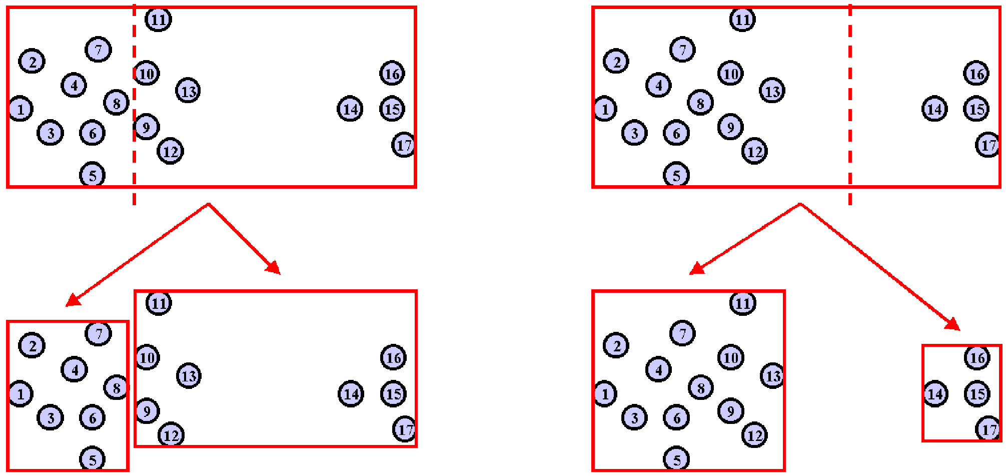


Figure 43: Enforcing a balanced tree can sacrifice goodness of fit. A K-D tree subdivision could place the vertical splitting line between vertices 8 and 9 to ensure a balanced tree (left), but splitting between 13 and 14 clearly fits the data better (right).

The assumption that each triangle appears on average in the **`node->tri`** lists of an approximately constant number of nodes could be made certain by storing only active triangles in those lists. Under this scheme, each active triangle would be referenced exactly three times, in the **`node->tri`** list of the three boundary nodes that represent the corners of the triangle. These lists would be dynamically maintained: the **`foldNode()`** function would create the **`node->tri`** list of the node to be folded by collecting the lists of its children, and the **`unfoldNode()`** function would distribute the tris of the node to be unfolded among the lists of the children. In addition, **`foldNode()`** would have to remove each subtri of the node from the **`node->tri`** lists it appears in, and **`unfoldNode()`** would have to add each subtri to the appropriate **`node->tri`** lists.

With this approach, the **`node->tri`** lists need only be maintained at run-time and need not be stored with the tree at all. Furthermore, the total memory taken up by the lists is proportional to the number of active triangles rather than the number of total triangles, so dynamically maintaining the **`node->tri`** lists will certainly save memory. Whether the scheme will save enough time to justify the added run-time complexity is unclear. The shorter lists mean **`foldNode()`** and **`unfoldNode()`** functions will have fewer triangles to update, but the overhead of maintaining the lists may overwhelm the other savings. In any case, a more elegant solution certainly seems possible and worth exploring.

CHAPTER 8

CONTEMPORARY RELATED WORK

8.1. Recent published algorithms

Algorithm	Reference	Mechanism				Error Metric		Topology			Style	
		Sampling	Adaptive Subdivision	Decimation	Vertex Merging	Fidelity-based	Polygon-budget	Preserving	Modifying	Tolerant	View-dependent	View-independent
View-Dependent Refinement of Progressive Meshes	Hoppe 97				X	X	X[7]	X			X	
Progressive Simplicial Complexes	Popovic 97				X		X		X	X		X
Surface Simplification Using Quadric Error Metrics	Garland 97				X		X		X	X		X
Dynamic View-Dependent Simplification …	Xia 96				X	X		X			X	
Hierarchical Dynamic Simplification	This thesis				X	X	X		X[8]	X	X	

Table 6: Four recent simplification algorithms classified according to the taxonomy of Chapter 2. HDS is presented for comparison.

This section discusses a few recently published papers of particular importance. Both [Xia 96] and [Hoppe 97] independently demonstrate view-dependent polygonal simplification schemes; [Popovic 97] describes a progressive representation that simplifies the topology as well as the geometry of polygonal models; [Garland 97] describes a traditional LOD algorithm of extraordinary speed and accuracy. Table 6 summarizes these

---

[7] This is not quite polygon-budget simplification in the sense of Section 4.3, but a feedback mechanism for regulating triangle count by varying screenspace error tolerance.

[8] Of course, the appropriate choice of vertex tree construction method and view-dependent criteria would enable a topology-preserving HDS algorithm, if desired.

algorithms, placing each into the taxonomy of Chapter 2 with an additional classification for view-dependent versus view-independent approaches. The chart also includes HDS for comparison. More detailed descriptions of the four algorithms follow.

8.1.1. Hoppe

View-Dependent Refinement of Progressive Meshes (1997)

Hoppe has recently extended progressive meshes to perform view-dependent simplification at run time, independently developing an elegant system similar in many ways to HDS. In this paper Hoppe alters the progressive mesh representation in minor ways, extending the structures and methods to enable view-dependent as well as dynamic simplification. The paper goes on to describe three view-dependent simplification criteria:

- A view frustum test to simplify aggressively regions of the mesh out of view.
- A backfacing test to simplify aggressively regions of the mesh not facing the viewer.
- A screenspace error threshold to guarantee that the geometric error in the simplified mesh is never greater than a user-specified screen-space tolerance.

The view frustum test is analogous to avoiding irrelevant nodes in HDS, described in Section 5.3. The chief difference between the algorithms is that Hoppe's approach simplifies irrelevant regions in the mesh, whereas HDS simply ignores them. The backfacing test closely resembles the silhouette preservation criteria used by HDS, though the formulation is better streamlined in Hoppe's test. This test could probably be extended to perform silhouette preservation in the manner of HDS, but Hoppe instead uses an ingenious method of measuring the screenspace error, out of which silhouette preservation falls naturally. Rather than simply putting a bounding volume around all vertices represented by a node in the vertex hierarchy, as HDS does, Hoppe finds the deviation between the mesh before and after folding the node. This deviation is represented as a *deviation space*, shown in Figure 44, consisting of two cones joined with a sphere.

As Hoppe explains, the directional component represented by the cones addresses deviation orthogonal to the surface, whereas the uniform component represented by the sphere captures deviation tangent to the surface and deviation of highly curved surfaces. The deviation introduced by a fold operation is often perpendicular to the surface, in which case the radius of the cones and sphere would be a small fraction of the height of the cones. Thus the screen-space projection of the deviation space for this fold operation is highest when the surface is tangent to the viewer (i.e., on the silhouette) and vanishes when the surface is perpendicular to the view vector (i.e., in the interior of the object). Clever streamlining of the math involved makes the screen-space projection of the deviation space surprisingly efficient. Hoppe reports that evaluating all three criteria, which share several subexpressions, takes only 230 CPU cycles on average [Hoppe 97].

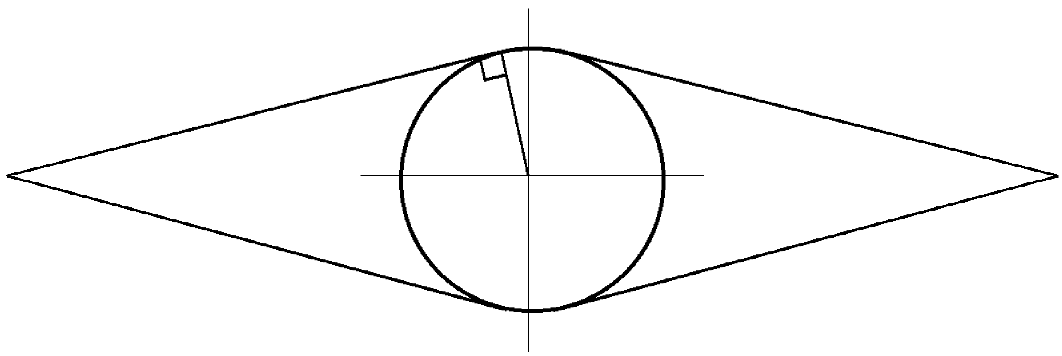


Figure 44: Hoppe's deviation space in cross-section.

The most obvious difference between HDS and view-dependent refinement of progressive meshes is probably their underlying mechanism of merging vertices. HDS evolved conceptually from the spatial binning approach of [Rossignac 92], clustering arbitrarily many vertices at once to a single representative vertex. Progressive meshes grew out of the author's early work on mesh optimization, and rely instead on the *edge collapse* operation [Hoppe 96]. An edge collapse merges exactly two vertices, which must share an edge in the mesh. As a result, the vertex hierarchy in a progressive mesh will always be binary, whereas the HDS vertex tree may in principle be *n*-ary[9]. Similarly, applying an edge collapse removes exactly two triangles from the mesh, whereas folding an HDS node may remove many triangles from the scene[10]. Which scheme is better is unclear. The binary tree

[9] In practice, of course, using a tight octree guarantees a vertex tree of maximum degree 8.

[10] Hoppe treats boundary edges by labeling one of the two triangles *nil*.

of a progressive mesh will typically be deeper than the corresponding HDS vertex tree, with many more nodes to traverse and store. On the other hand, the simple and regular structure of the edge collapse operation, which can be represented by a small, constant-size structure, lends itself to efficient storage and traversal. The finer granularity of the edge collapse could be an advantage, since a triangle budget can be specified very precisely, or a disadvantage, since more nodes must be processed to reach a desired level of simplification.

In the context of large CAD datasets, for which HDS was designed, the edge collapse operation used by Hoppe has some definite disadvantages. Since edge collapses that create non-manifold regions are disallowed, holes in the mesh are not simplified and the genus of the object remains fixed. As Section 1.3.2 argues, this can limit the potential for drastic simplification of high-genus objects. Moreover, since only vertices that share an edge are merged, each object and indeed each connected mesh component comprising each object must be simplified separately. This in turn has been shown in Section 1.3.3 to limit the potential for drastic simplification of complex assemblies of objects. By clustering based on proximity, without regard to topology or source object, HDS deals well with both of these problem cases.

One interesting aspect of Hoppe's system is a dynamic triangle-stripping algorithm which is applied every frame to reduce the rendering time. Most modern computer graphics hardware can render triangle strips faster than the same triangles fully specified, since in an ideal triangle strip the ratio of vertices transformed per triangle rendered approaches one. Hoppe takes advantage of this with a greedy rendering algorithm that attempts to create triangle strips as it traverses the visible triangles. The algorithm is quite simple (as it must be, since it is run anew every frame), but Hoppe reports an average strip length of 4.2 triangles. This should be enough to accelerate rendering, but Hoppe does not report the exact speedup. In any case, the integration of such a dynamic triangle-stripping algorithm into HDS seems worth exploring, though the non-manifold nature of typical HDS simplifications complicates the task. The difficulty is that triangle strips must span adjacent triangles, and HDS currently does not attempt to track triangle adjacencies through node fold and unfold operations.

8.1.2. Xia & Varshney

Dynamic, View-Dependent Simplification for Polygonal Meshes (1996)

Xia and Varshney also propose a view-dependent simplification scheme based on a hierarchy of edge-collapse operations. They discuss several view-dependent criteria for run-time simplification. Most of these have already described in the context of HDS and view-dependent progressive mesh refinement. One novel criterion is the use of local illumination information to add detail around regions such as specular highlights and shadow boundaries. The chief differences between their approach and that of [Hoppe 97] seem to be the method of constructing the vertex hierarchy (which Xia and Varshney call the *merge tree*) and the constraints imposed upon the run-time refinement of that hierarchy. In general, the simplifications given by [Hoppe 97] seem to achieve higher fidelity for a given triangle count. This appears to be partially the fault of those extra run-time constraints, which allow only gradual degradation of detail from high-fidelity to low-fidelity portions of the mesh, and partially the result of Hoppe's simplification process, which carefully optimizes the choice of edges to collapse and the placement of the resulting vertices.

8.1.3. Popovic and Hoppe

Progressive Simplicial Complexes (1997)

This vertex-merging algorithm extends progressive meshes to handle scenes of arbitrary topology, and in principle of arbitrary dimension. This involves generalizing the edge collapse and vertex split operations of progressive meshes. The edge collapse is replaced by a *vertex unification* operation, which merges two vertices that need not share an edge. The vertex split is replaced by a *generalized vertex split*, which encodes not only the position of the unified vertices but also their connectivity. This enables the progressive simplicial complex (PSC) representation to collapse triangles into lines and lines into points, like the Rossignac-Borrel and Low-Tan algorithms described in Chapter 2. Unlike those algorithms, however, the PSC performs the collapses selectively in an incremental, reversible fashion. The authors present a very computationally intensive simplification algorithm for ordering the vertex unification operations in an optimal fashion. Generating the PSC takes as long as

22 hours for one 232,974-triangle example. The fidelity of the example simplifications is quite high, taking excellent advantage of the topology-simplification capability. For example, a string of pearls (disconnected tetrahedra) in a model of a chandelier simplifies to a segmented 1-dimensional curve.

8.1.4. Garland and Heckbert

Surface Simplification Using Quadric Error Metrics (1997)

This recent view-independent vertex-merging algorithm strikes perhaps the best balance yet between speed, fidelity, and robustness. The algorithm proceeds by iteratively merging pairs of vertices, which may or may not be connected by an edge. Candidate vertex pairs are selected at the beginning of the algorithm according to a user-specified distance threshold t. Candidate pairs include all vertex pairs connected by an edge, plus all vertex pairs separated by less than t. The major contribution of the algorithm is a new way to represent the error introduced by a sequence of vertex merge operations, called the *quadric error metric*. The quadric error metric of a vertex is a 4x4 matrix that represents the sum of the squared distances from the vertex to the planes of neighboring triangles. Since the matrix is symmetric, 10 floating-point numbers suffice to represent the geometric deviation introduced during the course of the simplification.

The error introduced by a vertex merge operation can be quickly derived from the sum of the quadric error metrics of the vertices being merged, and that sum will become the quadric error metric of the merged vertex. At the beginning of the algorithm, all candidate vertex pairs are sorted into a priority queue according to the error calculated for merging them. The vertex pair with the lowest merge error is removed from the top of the queue and merged. The errors of all vertex pairs involving the merged vertices are then updated and the algorithm repeats.

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extraordinarily fast; the authors report simplifying the 70,000 triangle Bunny model to 100 triangles in 15 seconds. The visual fidelity of the resulting simplifications is quite high, especially at high levels of simplification. Since disconnected vertices closer than t are allowed to merge, the algorithm does not require manifold topology, though it can be made to preserve topology by setting *t* to zero. One disadvantage of the algorithm is that the number of candidate vertex pairs, and hence the running time of the algorithm, approaches $O(n^2)$ as *t* approaches the size of the model. Moreover, the choice of a good value for *t* is very model-specific and seems difficult to automate for general models. Within these limitations, however, this simple-to-implement algorithm appears to be the best combination of efficiency, fidelity, and generality currently available for creating traditional view-independent LODs. Many future algorithms will undoubtedly build on concepts introduced by Garland and Heckbert.

8.2. Issues and Trends

This section attempts to identify and discuss some important issues and trends facing the field of polygonal simplification.

8.2.1. Mechanism

The field appears to be converging on vertex merging as the underlying mechanism for polygon reduction. All four surface simplification papers in the SIGGRAPH '97 conference, for example, present algorithms based on merging vertices [Hoppe 97, Luebke 97, Garland 97, Popovic 97]. The simplicity and robust nature of vertex merging no doubt play a large part in this trend. Earlier work such as [Hoppe 96] and [Ronfard 96] has probably played a part as well by demonstrating that high-quality simplification is possible with an algorithm based entirely on edge collapses. Hierarchical vertex-merging representations such as progressive meshes and the HDS vertex tree provide very general frameworks for experimenting with different simplification strategies, including the relatively new view-dependent criteria. Settling on this emerging standard will hopefully allow the field of polygonal simplification to make faster strides in the other important issues discussed below.

8.2.2. Error metrics

The lack of an agreed-upon definition of fidelity seriously hampers comparison of results among algorithms. Most simplification schemes use some sort of distance-based metric in which fidelity of the simplified surface is assumed to vary with the distance of that surface from the original mesh. The edge-collapsing approach of Guèziec preserves the enclosed volume of the simplified surface to within a user-specified tolerance [Guèziec 97]. Such metrics are useful for certain CAD applications, such as finite element analysis, and for certain medical and scientific applications, such as co-registering surfaces or measuring volumes. Probably the most common use of polygonal simplification, however, is to speed up rendering for visualization of complex databases. For this purpose, the most important measure of fidelity is not geometric but perceptual: does the simplification *look* like the original?

Unfortunately, the human visual system remains imperfectly understood, and no well-defined perceptual metric exists to guide simplification. Existing distance- and volume-based metrics, while certainly useful approximations, suffer one definite deficiency by not taking the surface normal into account. Since lighting calculations are usually interpolated across polygons, for example, deviation in a vertex's normal can be even more visually distracting than deviation in its position. As another example, consider a pleated polygonal sheet. A single polygon spanning the width of the sheet may have minimal distance error but can have very different reflectance properties. In their survey of multiresolution methods for fast rendering, Garland and Heckbert propose that fidelity of simplification methods should be measured with perceptual tests using human viewers, or with a good *image-based* error metric. As a starting point for such a metric, they suggest the sum of the squared distances in RGB color space between corresponding pixels. [Garland 94].

8.2.3. View-dependence

Recent months have seen the publication of view-dependent algorithms from at least three research efforts working independently [Hoppe 97, Luebke 97, Xia 96]. View-dependence appears to be an idea whose time has come. The algorithms possess some definite advantages over their traditional view-independent counterparts. View-dependent methods are more general, making fewer implicit assumptions regarding object size. Under traditional LOD, physically large objects must be subdivided. Consider a model of a ship: the hull of the ship should be divided into several sections, or the end furthest from the user will be tessellated as finely as the nearby hull. View-dependent techniques do not have this problem, since a single object can be rendered at multiple levels of detail. Moreover, view-dependent methods offer the possibility of more sophisticated simplification criteria such as silhouette preservation, preservation of specular highlights [Xia 96], and aggressive simplification of backfacing regions [Hoppe 97]. View-independent algorithms can address none of these criteria.

However, view-dependent algorithms also suffer some significant drawbacks. To begin with, they inherently involve more run-time computation than view-independent approaches. When the CPU rather than the graphics subsystem is the limiting factor in rendering performance, view-dependent approaches become less attractive. Also, view-dependent simplification is by nature an immediate-mode technique, a disadvantage since most current rendering hardware favors retained-mode display lists. Experiments on an SGI Onyx with InfiniteReality graphics, for example, indicate that Gouraud-shaded depth-buffered unlit triangles render two to three times faster in a display list than in a tightly optimized immediate mode display loop [Aliaga 97]. For these reasons view-dependent techniques seem unlikely to completely supplant view-independent techniques in the near future.

8.2.4. Geometry Compression

A field closely related to polygonal simplification is *geometry compression*. Rather than attempting to produce simpler representations of a polygonal model, geometry compression focuses on minimizing the storage requirements of a given mesh. Deering introduced the first geometry compression algorithm for general 3-D polygonal models. His approach applies quantization and lossy compression to attributes such as the position, normal, and color of vertices, achieving compression rates of 6:1 to 10:1 [Deering 95]. Taubin and Rossignac extend this idea by compressing the topological connectivity of polygons in the mesh. Decomposing the triangulated model into a tree of linear triangle strips allows significant compression of connectivity information, down to an average of only two bits per triangle [Taubin 96]. The Taubin-Rossignac algorithm has since been incorporated into a proposal for the next-generation binary format of VRML, the Virtual Reality Modeling Language, and the authors of the proposal report compression ratios of 50:1 or more for large VRML models [Taubin 97].

8.2.5. Progressive Transmission

As the bandwidth and processing power available to home users increase, 3-D graphics seem likely to undergo a mass-market debut similar to that which has recently shaken the Internet. VRML and browser plug-ins may well bring such a revolution about quite soon via the World Wide Web. The evolution of the Web has underscored the importance of *progressive transmission* algorithms. These algorithms transmit a coarse version of the data first, followed by a stream of refinements, which the receiving process uses to reconstruct the original. The progressive mesh representation is by design well suited for progressive transmission of polygonal models [Hoppe 96]. If the mass-market debut of 3-D graphics occurs on the scale of the WWW, polygonal simplification algorithms may well be measured by their ability to support compression and progressive transmission.

CHAPTER 9

SUMMARY AND FUTURE WORK

9.1. Summary

Recall the thesis of this dissertation: *A global, dynamic, and view-dependent approach to polygonal simplification can provide a powerful, general framework for visualizing polygonal environments.* HDS has demonstrated the power and generality of such an approach. The preceding chapters have described the structures and methods to support dynamic simplification, given some examples of run-time criteria that can be used to effect view-dependent simplification, and discussed a number of significant optimizations to the basic system. HDS provides a flexible framework for view-dependent simplification by supporting different vertex-tree construction algorithms and view-dependent criteria; multiple examples of each have been described and implemented. The prototype system has been demonstrated on several models, including some extremely complex real-world CAD datasets, and shown to achieve interactive frame rates at good fidelity.

The chief advantages of HDS include the ability to perform drastic simplification, despite the presence of very large objects, very small and numerous objects, or objects of high genus, and the ability to simplify arbitrary polygonal models, despite the presence of non-manifold mesh topology. Another key advantage is the flexibility to choose the vertex tree construction algorithm and view-dependent simplification criteria to suit the application at hand. For example, the tight-octree vertex-clustering algorithm was designed with a CAD design-review application in mind. Its main advantages are speed, generality, and the ability to produce reasonable simplifications with no user intervention whatever.

The chief disadvantages of HDS are those of any view-dependent polygonal simplification scheme: an increased computational load on the CPU, and a mismatch to current graphics hardware, which is largely oriented towards retained-mode rendering.

9.2. Future Work

HDS has shown the promise of view-dependent simplification techniques, but many possible improvements and experiments suggest themselves. This section discusses some promising avenues of future work.

9.2.1. Better Vertex Trees

The current system uses bounding spheres to determine the visibility and screenspace extent of nodes in the vertex tree. These spheres have the advantage of simplicity, but often provide a poor fit to the vertices represented by the node. Node visibility and screenspace extent can thus be badly overestimated, which results in an overly conservative simplification with more polygons than necessary. Modifying HDS to use tighter-fitting, more adaptive bounding volumes would be an intriguing experiment. For example, bounding volume hierarchies using oriented bounding volumes have been shown to approximate certain models to a given precision with asymptotically fewer bounding volumes than those using axis-aligned bounding volumes. Arbitrarily oriented ellipsoids would also make an interesting bounding volume. Ellipsoids have the nice property of projecting to ellipses on the screen under perspective projection, which might simplify the computation of their screenspace extent.

The tight-octree scheme clusters vertices roughly by proximity; the hybrid edge-collapse/tight octree scheme described in Section 6.3 also takes the normals of adjacent polygons into account. Many other possible factors for vertex-tree construction come to mind. Garland’s quadric error metrics, for example, provide a nice method of tracking and minimizing geometric distortion caused by vertex merging [Garland 97]. [Hoppe 96] allows material attributes such as color to influence the creation of the vertex hierarchy; [Cohen 98] presents a careful algorithm for *appearance-preserving simplification* that regulates not only geometric fidelity but color and shading information.

9.2.2. Better Use of Resources

Display Lists

HDS requires immediate-mode graphics, but today's high-end graphics hardware runs three to five times faster using display lists [Aliaga 97]. The motivation to use that hardware better is clear, but how to do so is not obvious. For example, a bit of thought suggests that many triangles in HDS have a long life span. The subtris of the root node, for example, will likely be present the entire course of an HDS session. Why not incorporate those triangles into a display list? The reason is that, although the triangles are indeed present the entire time, they are changing shape. Even the subtris of the root node may shift slightly as the nodes representing the corners those triangles get folded and unfolded. This prevents the system from capturing long-lived triangles into display lists, which require completely frozen geometry.

This idea is not without merit, however. Perhaps a way could be found to identify triangles with all three corners well above the boundary path, and form them into display lists on the fly. Such triangles should remain fixed until the boundary path rises above the nodes representing their corners, which would invalidate the display list. If multiple display lists corresponding to different parts of the vertex tree were created, the system could invalidate just the display lists containing violating triangles, rendering their associated triangles in immediate mode instead. This would enable the system to use display lists, with their increased rendering performance, as much as possible.

Additional Parallelism

Modern high-end graphics platforms often have more CPUs than rendering subsystems. Only one CPU can typically feed each graphics pipeline, so further parallelizing the algorithm seems worthwhile. For example, a straightforward extension to the asynchronous simplification scheme described in Section 5.5 would split the SIMPLIFY task into multiple tasks. Perhaps every processor could run a SIMPLIFY task, each hunting around the vertex tree for nodes to be folded or unfolded. The dynamic display-list-compiling scheme described above might provide another use for extra processing power. One processor might be in charge of continuously searching the vertex tree for triangles that can be safely frozen, and compiling them into a display list. When a display list was ready, the RENDER process could grab it from memory and send it to the graphics hardware.

Memory management

HDS sorely needs better memory-management techniques. For example, the current implementation only works well when the whole vertex tree fits in main memory, but for truly massive models this will often not be the case. Careful structuring of the algorithms and intelligent prefetching (perhaps by an unused CPU) should make it possible to keep only the immediate region of the boundary path in memory. The rest of the vertex tree could then be loaded off of disk on a just-in-time basis.

The memory access patterns of HDS could also be improved. For example, the current system implements the active triangle list as a doubly linked list to support efficient insert and delete operations. This list is maintained in place, threaded through an array of all triangles in the model. As Section 3.3 describes, however, this tends to create a haphazard path that ruins cache coherence as it hops seemingly at random back and forth through the array. Moreover, if only ten percent of the triangles in the model are active, ninety percent of the cached array is useless anyway. Ideally, all the active triangles would be collected into a single coherent array where they could be rendered with a simple linear pass.

One approach would be to organize the array of all n triangles in the model into two parts, the first containing m active triangles and the second containing n-m inactive triangles. The **`addTri()`** routine would swap the triangle to be made active with the first inactive triangle, at position $m+1$, then increment m. Similarly, the **`removeTri()`** routine need only swap the triangle to be made inactive with the mth triangle and decrement m. Since the active triangles are always the first m entries in the array, the individual triangle structures need not even have an **`active`** flag. The memory access performance of a coherent linear sweep across a packed array should be far superior to the current memory-hopping linked-list traversal, especially with some prefetching to avoid cache misses. This elegant scheme still presents some problems, such as how the **`tris`** and **`subtris`** fields of nodes in the vertex tree can refer to triangles whose positions in memory keep changing, but on the whole it seems well worth exploring.

9.2.3. Dynamic Polygonal Simplification

A final important avenue of future research is the wide-open question of how to simplify dynamic polygonal environments. Every simplification algorithm to date assumes that the models to be simplified are static, and must be run from scratch if the model changes. Even the fastest algorithms, including HDS using tight-octree vertex clustering, report times of several seconds to simplify a medium-sized model such the 70,000 triangle Stanford bunny. It would appear useful to develop an algorithm that supports incremental updates to the simplification in response to incremental changes in the model.

Two particularly interesting categories of dynamic polygonal environments are active CAD sessions and deformable models. In an active CAD session, a designer builds a complex model with a series of incremental, often local changes. For example, many systems are based on constructive solid geometry (CSG) modeling, in which solids are defined by a series of Boolean operations upon simpler solids. Supported operations include union, intersection, and subtraction. A simplification system that supported efficient union, intersection, and subtraction operations upon HDS-style vertex trees could maintain a simplified representation of the model through incremental updates to the design. This should enable the designer to view a larger, more complex portion of the model interactively, which might provide more helpful context.

A deformable model can change shape interactively, typically with a high degree of temporal coherence. A simulation of a bouncing ball or of fabric draped over a moving object might make a good example. Again, these models are changing only incrementally, and an incremental simplification algorithm should be capable of maintaining a simplified representation of the model to speed up rendering and possibly even the simulation. Simplification of dynamic scenes is a challenging problem, and seems likely to be one of the next frontiers of polygonal simplification.

CHAPTER 10

REFERENCES

[Airey 90] Airey, John. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. Ph.D. thesis, UNC-CH CS Department TR #90-027 (July 1990).

[Aliaga 97] Aliaga, Daniel. "SGI Performance Tips" (Talk). For more information see: *http://www.cs.unc.edu/~aliaga/IR.html*.

[Clark 76] Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, Vol. 19, No 10, pp 547-554.

[Cohen 96] Cohen, Jon, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright. "Simplification Envelopes", *Computer Graphics*, Vol. 30 (SIGGRAPH 96).

[Cohen 98] Cohen, Jon, and D. Manocha. "Appearance Preserving Simplification", *Computer Graphics*, Vol. 32 (SIGGRAPH 98).

[Cosman 81] Cosman, M., and R. Schumacker. "System Strategies to Optimize CIG Image Content". *Proceedings Image II Conference* (Scotsdale, Arizona), 1981.

[Deering 95] Deering, Michael. "Geometry Compression", *Computer Graphics*, Vol. 29 (SIGGRAPH 95).

[Eck 95] Eck, Matthias, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle. "Multiresolution Analysis of Arbitrary Meshes", *Computer Graphics*, Vol. 29 (SIGGRAPH 95).

[Erikson 96] Erikson, Carl. "Polygonal Simplification: An Overview". UNC-CH CS Department TR #96-016 (July 1996).

[Garland 94] Garland, Michael, and P. Heckbert. "Multiresolution Modeling for Fast Rendering". *Proceedings of Graphics Interface '94* (1994).

[Greene 93] Greene, Ned, M. Kass, G. Miller. "Hierarchical Z-Buffer Visibility", *Computer Graphics*, Vol. 29 (SIGGRAPH 93).

[He 95] Taosong He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. "Voxel-Based Object Simplification". *Proceedings Visualization 95*, IEEE Computer Society Press (Atlanta, GA), 1995, pp. 296-303.

[Hoppe 92] Hoppe, Hughes, T. DeRose, T. DuChamp, J. McDonald, W. Stuetzle. "Surface Reconstruction from Unorganized Points", *Computer Graphics*, Vol. 26 (SIGGRAPH 92).

[Hoppe 93] Hoppe, Hughes. "Mesh Optimization", *Computer Graphics*, Vol. 27 (SIGGRAPH 93).

[Hoppe 96] Hoppe, Hughes. "Progressive Meshes", *Computer Graphics*, Vol. 30 (SIGGRAPH 96).

[Hoppe 97] Hoppe, Hughes. "View-Dependent Refinement of Progressive Meshes", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

[Lorenson 87] Lorenson, William, and H. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", *Computer Graphics*, Vol. 21 (SIGGRAPH 87).

[Lorenson 95] Lorensen, W., "Marching Through the Visible Man," in *Proceedings of Visualization '95*, IEEE Press, October 1995.

[Luebke 97] Luebke, David, and C. Erikson. "View-Dependent Simplification of Arbitrary Polygonal Environments", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

[Low 97] Low, Kok-Lim, and T.S. Tan. "Model Simplification Using Vertex Clustering". In *1997 Symposium on Interactive 3D Graphics* (1995), ACM SIGGRAPH, pp. 75-82.

[Popovic 97] Popovic, Jovan, and H. Hoppe. "Progressive Simplicial Complexes", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

[Ronfard 96] Ronfard, Rèmi, and J. Rossignac. "Full-range Approximation of Triangulated Polyhedra", *Computer Graphics Forum*, Vol. 15 (Eurographics 96).

[Rossignac 92] Rossignac, Jarek, and P. Borrel. "Multi-Resolution 3D Approximations for Rendering Complex Scenes", pp. 455-465 in *Geometric Modeling in Computer Graphics*, Springer-Verlag, Eds. B. Falcidieno and T.L. Kunii, Genova, Italy, 6/28/93-7/2/93. Also published as IBM Research Report RC17697 (77951) 2/19/92.

[Schroeder 92] Schroeder, William, J. Zarge and W. Lorenson, "Decimation of Triangle Meshes", *Computer Graphics*, Vol. 26 (SIGGRAPH 92).

[Shirman 93] Shirman, L., and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches", *Computer Graphics Forum* (Proc. Eurographics '93) Vol. 12, No 3, (1993), p.p. 261-272.

[Taubin 96] Taubin, Gabriel, and J. Rossignac. "Geometric Compression through Topological Surgery", IBM Research Technical Report RC-20340 (1996).

[Taubin 97] Taubin, Gabriel (Chair). "VRML Compressed Binary Format Working Group Home Page". For more information see: *http://www.vrml.org/WorkingGroups/vrml-cbf/cbfwg.html*.

[Teller 91] Teller, Seth, and C. Sequin. "Visibility Preprocessing for Interactive Walkthroughs", *Computer Graphics*, Vol. 25 (SIGGRAPH 91).

[Turk 92] Turk, Greg. "Re-tiling Polygonal Surfaces", *Computer Graphics*, Vol. 26 (SIGGRAPH 92).

[Xia 96] Xia, Julie, and A. Varshney. "Dynamic View-Dependent Simplification for Polygonal Models", *Visualization 96*.

[Zhang 97] Zhang, Hansong, D. Manocha, T. Hudson, K. Hoff. "Visibility Culling Using Hierarchical Occlusion Maps", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

APPENDIX

This appendix supplies additional performance data for HDS running on all of the models described in Section 7.2. The charts below amplify the information in Section 7.6: Vertex Tree Characteristics. To repeat the explanation given there, the tables break down the vertex trees by depth, describing for each level of the tree:

- The number of nodes at that level.
- The number of leaf nodes at that level.
- The mean degree of all nodes at that level, that is, the average number of children per node, along with the standard deviation of that mean.
- The mean degree of interior nodes at that level, along with the standard deviation.
- The average number of tris per node.
- The average number of subtris per interior node.

Section 7.6 gave tables for the Cassini model and the Bone6 dataset. Tables are given below for the rest of the sample models: Sphere, Bunny, Sierra, AMR, Torp, Powerplant_3M, and Powerplant_4M.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	12.00
1	8	0	7.00	0.00	7.00	0.00	96.00	12.00
2	56	0	6.43	0.50	6.43	0.50	34.71	12.57
3	360	48	4.32	2.00	4.98	1.14	15.27	7.69
4	1,554	672	1.96	1.91	3.45	1.12	8.25	4.88
5	3,042	2,772	0.20	0.66	2.24	0.60	6.20	2.49

Table 7: Vertex tree statistics for the Sphere model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	12.00
1	8	0	7.00	0.76	7.00	0.76	260.62	13.50
2	56	0	6.79	0.89	6.79	0.89	107.16	12.12
3	380	10	5.80	1.61	5.96	1.32	41.04	10.39
4	2,205	250	4.30	2.10	4.85	1.51	17.84	7.81
5	9,481	3,001	2.38	1.94	3.48	1.29	9.65	4.89
6	22,552	15,533	0.71	1.10	2.27	0.58	6.73	2.50
7	15,906	15,773	0.02	0.18	2.01	0.09	5.99	2.02

Table 8: Vertex tree statistics for the Bunny model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	10.00
1	8	0	6.75	1.58	6.75	1.58	420.00	14.25
2	54	1	6.39	1.61	6.51	1.35	204.41	14.21
3	345	10	6.13	1.71	6.32	1.36	83.54	12.92
4	2,116	160	4.90	2.16	5.30	1.71	32.55	9.93
5	10,370	2,318	3.29	2.28	4.24	1.64	14.98	6.76
6	34,128	15,982	1.54	1.66	2.90	1.11	8.56	3.47
7	52,570	43,492	0.37	0.84	2.17	0.48	6.39	2.22
8	19,682	19,407	0.03	0.23	2.00	0.00	5.99	1.99

Table 9: Vertex tree statistics for the Sierra model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	20.00
1	8	0	7.00	1.07	7.00	1.07	724.75	37.00
2	56	0	6.79	1.02	6.79	1.02	377.32	18.75
3	380	1	5.15	1.69	5.16	1.67	119.96	11.54
4	1,957	212	3.69	2.40	4.14	2.14	45.53	8.54
5	7,228	2,445	2.94	2.63	4.44	1.94	25.54	8.08
6	21,245	6,720	2.61	2.24	3.82	1.65	16.28	5.92
7	55,485	23,112	1.85	1.86	3.17	1.33	10.64	4.31
8	102,702	60,852	1.08	1.45	2.64	1.00	7.98	3.10
9	110,616	85,145	0.55	1.07	2.38	0.76	6.75	2.47
10	60,672	50,195	0.38	0.86	2.19	0.54	6.23	1.91
11	22,902	19,492	0.31	0.76	2.06	0.54	5.85	1.77
12	7,027	5,958	0.27	0.67	1.75	0.58	5.69	1.42
13	1,875	1,496	0.23	0.48	1.13	0.34	6.21	0.21
14	427	89	0.81	0.43	1.02	0.13	9.39	0.01
15	344	12	0.97	0.18	1.00	0.00	10.44	0.00
16	332	0	1.00	0.00	1.00	0.00	10.75	0.00
17	332	0	1.00	0.00	1.00	0.00	10.75	0.00
18	332	0	1.00	0.00	1.00	0.00	10.75	0.00

Table 10: Vertex tree statistics for the AMR model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	8.00		8.00		0	62.00
1	8	0	7.25	0.89	7.25	0.89	1044.62	38.62
2	58	0	6.83	1.44	6.83	1.44	564.48	30.09
3	396	15	5.52	2.15	5.73	1.89	154.93	13.18
4	2,185	166	4.43	2.27	4.80	1.96	52.57	9.38
5	9,689	1,900	3.57	2.51	4.44	1.99	24.76	7.20
6	34,581	10,564	2.53	2.17	3.64	1.66	14.01	5.05
7	87,445	38,698	1.73	1.85	3.10	1.36	9.46	3.69
8	151,284	91,707	1.07	1.47	2.73	1.01	7.15	2.81
9	162,349	115,958	0.71	1.18	2.47	0.74	6.03	2.24
10	114,544	93,079	0.43	0.93	2.29	0.56	5.28	1.76
11	49,084	44,495	0.20	0.63	2.12	0.39	4.62	1.40
12	9,731	9,514	0.04	0.30	2.01	0.10	4.03	1.10

Table 11: Vertex tree statistics for the Torp model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	5.00		5.00		0	16.00
1	5	0	6.80	1.64	6.80	1.64	2659.20	96.60
2	34	0	7.03	1.24	7.03	1.24	1927.15	54.35
3	239	1	6.61	1.57	6.64	1.52	789.51	22.48
4	1,580	107	4.89	2.38	5.25	2.05	259.92	12.06
5	7,734	881	4.39	2.36	4.95	1.87	89.62	8.91
6	33,949	5,611	3.62	2.29	4.33	1.78	33.43	7.56
7	122,751	29,694	2.76	2.10	3.64	1.62	16.20	6.01
8	338,628	151,559	1.72	1.88	3.11	1.44	10.57	4.88
9	581,737	373,817	0.95	1.44	2.65	1.14	8.76	3.40
10	551,425	434,303	0.49	1.01	2.30	0.78	8.00	2.13
11	269,161	235,206	0.25	0.70	1.98	0.68	7.58	1.61
12	67,222	56,493	0.22	0.56	1.40	0.59	8.00	1.04
13	15,066	7,754	0.51	0.55	1.06	0.24	9.71	0.17
14	7,724	794	0.90	0.31	1.00	0.04	10.23	0.00
15	6,940	18	1.00	0.05	1.00	0.02	10.29	0.00
16	6,924	4	1.00	0.02	1.00	0.00	10.30	0.00
17	6,920	0	1.00	0.00	1.00	0.00	10.31	0.00
18	6,920	0	1.00	0.00	1.00	0.00	10.31	0.00

Table 12: Vertex tree statistics for the Powerplant_2M model.

Level	Nodes	Leaf Nodes	Children/ Node	(std dev)	Children/ Interior Node	(std dev)	Tris/ Node	Subtris/ Interior Node
0	1	0	4.00		4.00		0	0.00
1	4	0	8.00	0.00	8.00	0.00	6696.25	86.50
2	32	0	7.19	1.31	7.19	1.31	4197.53	63.94
3	230	0	7.10	1.26	7.10	1.26	1349.50	28.46
4	1,634	12	6.05	1.80	6.10	1.73	343.55	13.78
5	9,891	365	4.80	1.95	4.98	1.74	87.90	9.90
6	47,464	5,448	3.72	2.13	4.20	1.76	30.91	8.51
7	176,613	51,820	2.50	2.13	3.53	1.66	16.35	7.07
8	440,744	221,827	1.53	1.88	3.09	1.53	12.08	5.59
9	675,373	430,515	1.00	1.55	2.76	1.33	10.57	3.49
10	676,114	464,121	0.73	1.19	2.33	0.91	9.26	1.64
11	493,058	374,549	0.46	0.85	1.91	0.49	7.84	0.66
12	226,743	188,850	0.24	0.58	1.46	0.51	7.14	0.21
13	55,369	33,383	0.42	0.54	1.05	0.23	7.99	0.01
14	23,173	2,351	0.90	0.30	1.00	0.03	9.99	0.00
15	20,840	35	1.00	0.04	1.00	0.01	10.50	0.00
16	20,806	2	1.00	0.01	1.00	0.00	10.51	0.00
17	20,804	0	1.00	0.00	1.00	0.00	10.51	0.00
18	20,804	0	1.00	0.00	1.00	0.00	10.51	0.00

Table 13: Vertex tree statistics for the Powerplant_4M model.

One interesting feature to note in the above charts is the unnecessary depth of the AMR and Powerplant vertex trees. The AMR, for example, has 332 nodes at level 19, the maximum depth the vertex tree is allowed to reach. However, these nodes are simply the sole children of the nodes at level 18, which are in turn the sole children of the nodes at level 17, and so on. The problem is that some of the vertices in the original model are exactly coincident, so the tight octree cannot resolve them into separate nodes of the vertex tree. The current implementation is flawed, since the spatial subdivision procedure simply continues to recurse until the maximum node depth is reached. A simple solution would be to test the size of each tightened octree node. If a node reaches zero size but has multiple vertices, those vertices are coincident and can be merged. Implementing this simple fix would reduce the depth of the AMR and Powerplant vertex trees, saving storage space and preprocessing time. Powerplant_4M in particular would benefit, since over one percent of its vertices are coincident.

The final table summarizes the depth of leaf nodes in the vertex trees of each of the sample models, and was the source for the histogram in Figure 29:

Level	AMR	Bone6	Bunny	Cassini	Powerplant_2M	Powerplant_4M	Sierra	Sphere	Torp
0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.2%	0.0%
4	0.1%	0.3%	0.7%	0.0%	0.0%	0.0%	0.2%	16.4%	0.0%
5	1.0%	1.7%	8.6%	0.5%	0.1%	0.0%	2.8%	67.6%	0.5%
6	2.6%	8.4%	44.6%	2.6%	0.4%	0.3%	19.5%	14.8%	2.6%
7	9.0%	27.4%	45.3%	9.3%	2.3%	2.9%	53.1%	0.0%	9.5%
8	23.8%	43.2%	0.8%	23.7%	11.6%	12.4%	23.7%	0.0%	22.6%
9	33.3%	18.0%	0.0%	33.3%	28.7%	24.0%	0.7%	0.0%	28.5%
10	19.6%	0.9%	0.0%	21.8%	33.3%	25.9%	0.0%	0.0%	22.9%
11	7.6%	0.0%	0.0%	5.3%	18.0%	20.9%	0.0%	0.0%	10.9%
12	2.3%	0.0%	0.0%	2.3%	4.3%	10.5%	0.0%	0.0%	2.3%
13	0.6%	0.0%	0.0%	0.9%	0.6%	1.9%	0.0%	0.0%	0.1%
14	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	0.0%	0.0%	0.0%
15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
17	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
18	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 14: Leaf node depths for each sample model.