Concepts and Algorithms for Polygonal Simplification

Jonathan D. Cohen

Department of Computer Science, The Johns Hopkins University

1. INTRODUCTION

1.1 Motivation

In 3D computer graphics, polygonal models are often used to represent individual objects and entire environments. Planar polygons, especially triangles, are used primarily because they are easy and efficient to render. Their simple geometry has enabled the development of custom graphics hardware, currently capable of rendering millions or even tens of millions of triangles per second. In recent years, such hardware has become available even for personal computers. Due to the availability of such rendering hardware and of software to generate polygonal models, polygons will continue to play an important role in 3D computer graphics for many years to come.

However, the simplicity of the triangle is not only its main advantage, but its main disadvantage as well. It takes many triangles to represent a smooth surface, and environments of tens or hundreds of millions of triangles or more are becoming quite common in the fields of industrial design and scientific visualization. For instance, in 1994, the UNC Department of Computer Science received a model of a notional submarine from the Electric Boat division

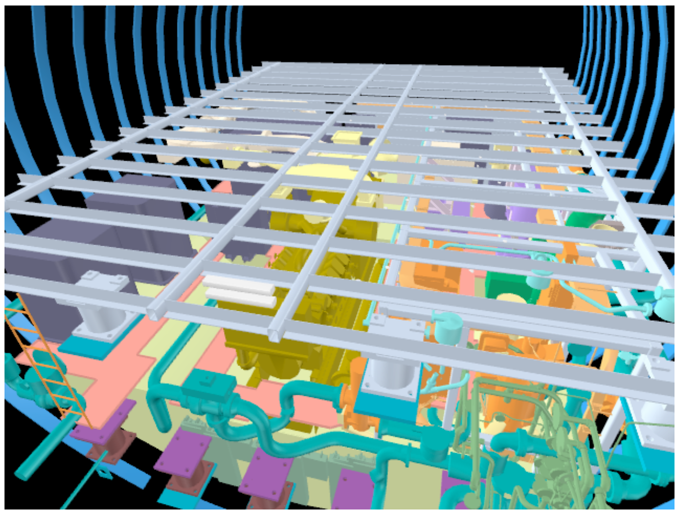


Figure 1: The auxiliary machine room of a notional submarine model: 250,000 triangles

of General Dynamics, including an auxiliary machine room composed of 250,000 triangles (see Figure 1) and a torpedo room composed of 800,000 triangles. In 1997, we received from ABB Engineering a coarsely-tessellated model of an entire coal-fired power plant, composed of over 13,000,000 triangles. It seems that the remarkable performance increases of 3D graphics hardware systems cannot yet match the desire and ability to generate detailed and realistic 3D polygonal models.

1.2 Polygonal Simplification

This imbalance of 3D rendering performance to 3D model size makes it difficult for graphics applications to achieve *interactive* frame rates (10-20 frames per second or more). Interactivity is an important property for applications such as architectural walkthrough, industrial design, scientific visualization, and virtual reality. To achieve this interactivity in spite of the enormity of data, it is often necessary to trade fidelity for speed.

We can enable this speed/fidelity tradeoff by creating a *multi-resolution* representation of our models. Given such a representation, we can render smaller or less important objects in the scene at a lower resolution (i.e. using fewer triangles) than the larger or more important objects, and thus we render fewer triangles overall. Figure 2 shows a widely-used test model: the Stanford bunny. This model was acquired using a laser range-scanning device; it contains over 69,000 triangles. When the 2D image of this model has a fairly large area, this may be a reasonable number of triangles to use for rendering the image. However, if the image is smaller, like Figure 3 or Figure 4, this number of triangles is probably too large. The right-most image in each of these figures shows a bunny with fewer triangles. These complexities are often more appropriate for image of these sizes. Each of these images is typically some small piece of a much larger image of a complex scene.

For CAD models, such representations could be created as part of the process of building the original model. Unfortunately, the robust modeling of 3D objects and environments is already a difficult task, so we would like to explore solutions that do not add extra burdens to the original modeling process. Also, we would like to create such representations for models acquired by other means (e.g. laser scanning), models that already exist, and models in the process of being built.

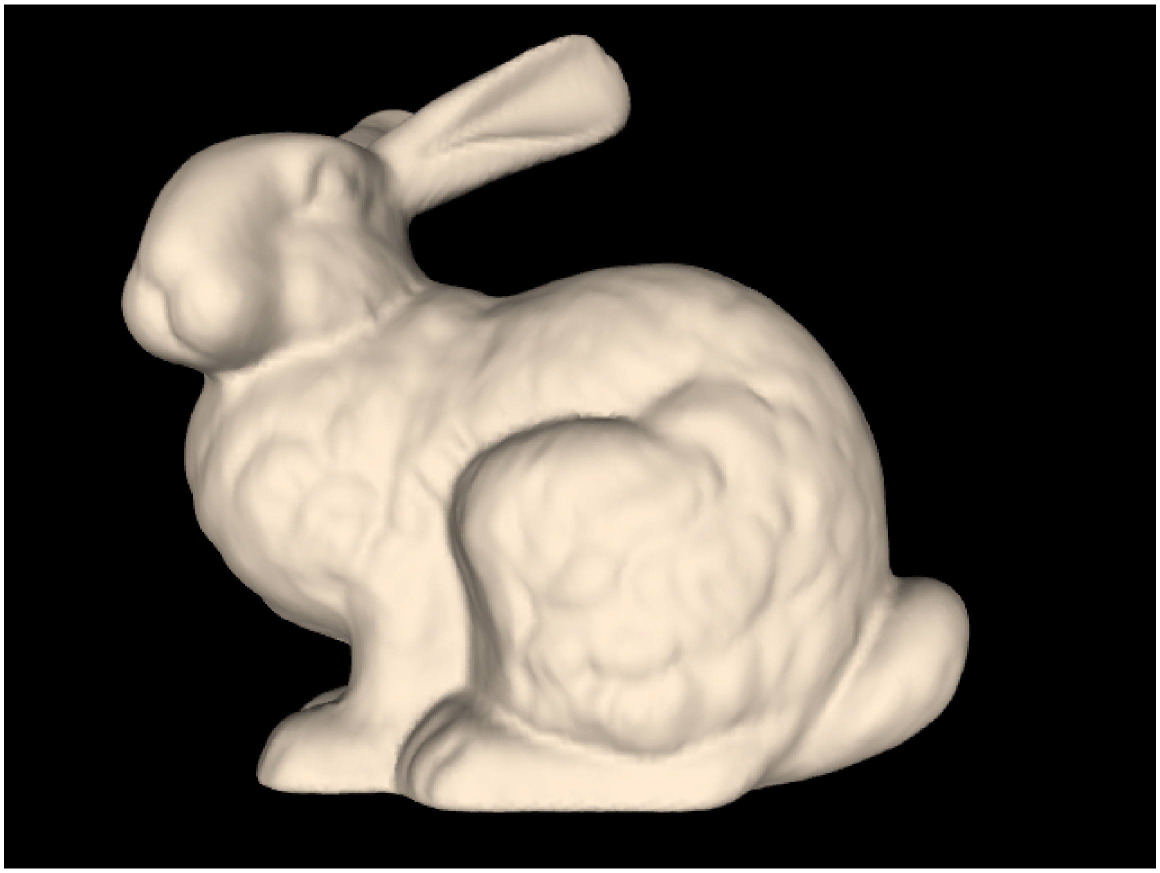
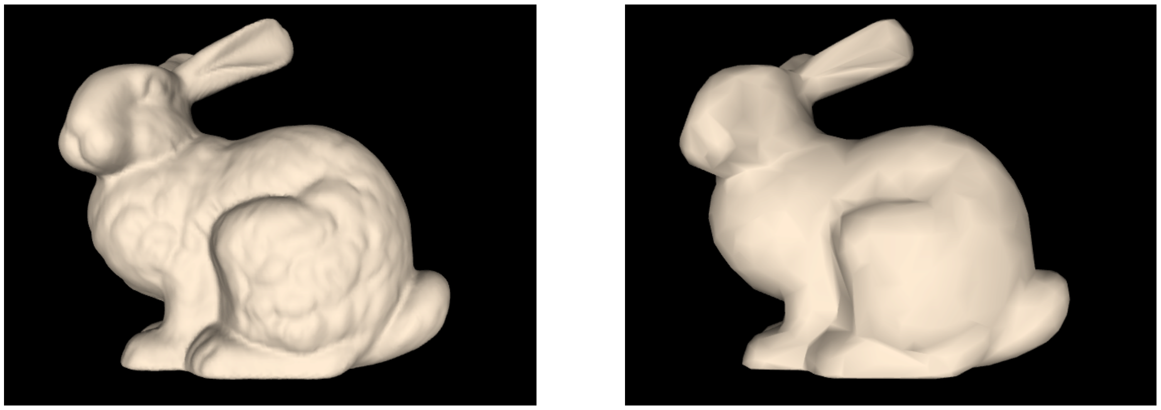


Figure 2: The Stanford bunny model: 69,451 triangles


69,451 triangles

2,204 triangles

Figure 3: Medium-sized bunnies.

69,451 triangles

575 triangles

Figure 4: Small-sized bunnies.

Simplification is the process of automatically reducing the complexity of a given model. By creating one or more simpler representations of the input model (generally called *levels of detail*), we convert it to a multi-resolution form. This problem of automatic simplification is rich enough to provide many interesting and useful avenues of research. There are many issues related to how we represent these multi-resolution models, how we create them, and how we manage them within an interactive graphics application. This dissertation is concerned primarily with the issues of level-of-detail quality and rendering performance. In particular, we explore the question of how to preserve the appearance of the input models to within an intuitive, user-specified tolerance and still achieve a significant increase in rendering performance.

1.3 Topics Covered

This paper reviews some fundamental concepts necessary to understand algorithms for simplification of polygonal models at a high level. These concepts include optimal/near-optimal solutions for the simplification problem, the use of local simplification operations, topology preservation, level-of-detail representations for polygonal models, error measures for surface deviation, and the preservation of appearance attributes. This is not a complete survey of the field of polygonal model simplification, which has grown to be quite large (for more information, several survey papers are available [Erikson 1996, Heckbert and Garland 1997]). In particular, this paper does *not* provide much coverage of algorithms specialized for simplifying polygonal terrains, nor does it cover simplification and compression algorithms geared towards progressive transmission applications.

2. OPTIMALITY

There are two common formulations of the simplification problem, described in [Varshney 1994], to which we may seek optimal solutions:

- **Min-# Problem**: Given some error bound, ε, and an input model, I, compute the minimum complexity approximation, A, such that no point of A is farther than ε distance away from I and vice versa (the complexity of A is measured in terms of number of vertices or faces).

- **Min-ε Problem**: Given some target complexity, n, and an input model, I, compute the approximation, A, with the minimum error, ε, described above.

In computational geometry, it has been shown that computing the min-# problem is NP-hard for both convex polytopes [Das and Joseph 1990] and polyhedral terrains [Agarwal and Suri 1994]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_0 be the size of a min-# approximation. An algorithm has been given in [Mitchell and Suri 1992] for computing an ε-approximation of size $O(k_0 \log n)$ for convex polytopes of initial complexity n. This has been improved by Clarkson in [Clarkson 1993]; he proposes a randomized algorithm for computing an approximation of size $O(k_0 \log k_0)$ in expected time $O(k_0 n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [Brönnimann and Goodrich 1994] Brönnimann and Goodrich observed that a variant of Clarkson's algorithm yields a polynomial-time deterministic algorithm that computes an approximation of size $O(k_0)$. Working with polyhedral terrains, [Agarwal and Suri 1994] present a polynomial-time algorithm that computes an ε-approximation of size $O(k_0 \log k_0)$ to a polyhedral terrain.

Because the surfaces requiring simplification may be quite complex (tens of thousands to millions of triangles), the simplification algorithms used in practice must be $o(n^2)$ (typically $O(n \log n)$) for the running time to be reasonable. Due to the difficulty of computing near-optimal solutions for general polygonal meshes and the required efficiency, most of the algorithms described in the computer graphics literature employ local, greedy heuristics to achieve what appear to be reasonably good simplifications with no guarantees with respect to the optimal solution.

3. LOCAL SIMPLIFICATION OPERATIONS

Simplification is often achieved by performing a series of local operations. Each such operation serves to coarsen the polygonal model by some small amount. A simplification algorithm generally chooses one of these operation types and applies it repeatedly to its input surface until the desired complexity is achieved for the output surface.

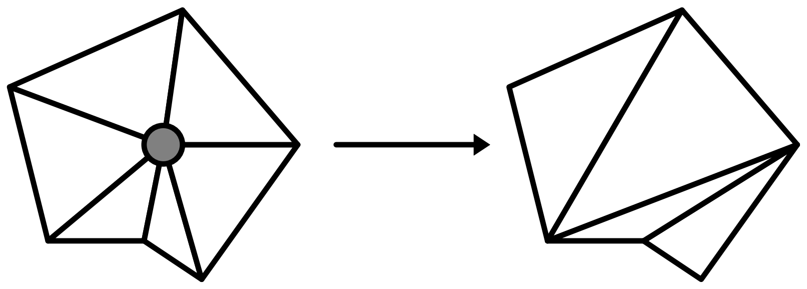


Figure 5: Vertex remove operation

3.1 Vertex Remove

The vertex remove operation involves removing from the surface mesh a single vertex and all the triangles touching it. This removal process creates a hole that we then fill with a new set of triangles. Given a vertex with *n* adjacent triangles, the removal process creates a hole with *n* sides. The hole filling problem involves a discrete choice from among a finite number of possible retriangulations for the hole. The *n* triangles around the vertex are replaced by this new triangulation with *n*-2 triangles. The Catalan sequence,

$$\mathrm{C}(i) = \frac{1}{i+1} * \binom{2i}{i} = \frac{1}{i+1} * \frac{(2i)!}{i!(2i-i)!} = \frac{1}{i+1} * \frac{(2i)!}{i!i!} = \frac{(2i)!}{(i+1)!i!} \ , \qquad (1)$$

describes the number of unique ways to triangulate a convex, planar polygon with *i*+2 sides [Dörrie 1965, Plouffe and Sloan 1995]. This provides an upper bound on the number of non-self-intersecting triangulations of a hole in 3D. For example, holes with 3 sides have only 1 triangulation, and holes with 4, 5, 6, 7, 8, and 9 sides have up to 2, 5, 14, 42, 132, and 429 triangulations, respectively.

Both [Turk 1992] and [Schroeder et al. 1992] apply the vertex remove approach as part of their simplification algorithms. Turk uses point repulsion (weighted according to curvature) to distribute some number of new vertices across the original surface, then applies vertex remove operations to remove most of the original vertices. Holes are retriangulated using a planar projection approach. Schroeder also uses vertex remove operations to reduce mesh complexity, employing a recursive loop splitting algorithm to fill the necessary holes.

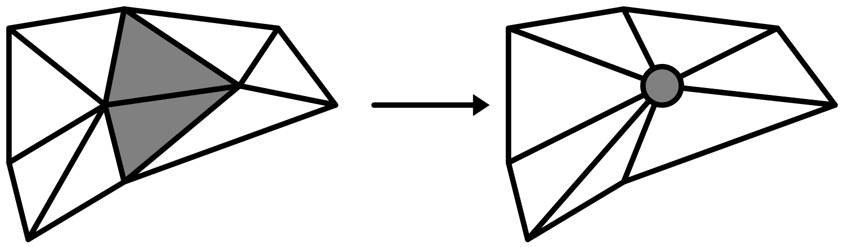


Figure 6: Edge collapse operation

3.2 Edge Collapse

The edge collapse operation has become popular in the graphics community in the last several years. The two vertices of an edge are merged into a single vertex. This process distorts all the neighboring triangles. The triangles that contain both of the vertices (i.e. those that touch the entire edge) degenerate into 1-dimensional edges and are removed from the mesh. This typically reduces the mesh complexity by 2 triangles.

Whereas the vertex remove operation amounts to making a discrete choice of triangulations, the edge collapse operation requires us to choose the coordinates of the new vertex from a continuous domain. Common choices for these new coordinates include the coordinates of one of the two original vertices, the midpoint of the collapsed edge, arbitrary points along the collapsed edge, or arbitrary points in the neighborhood of the collapsed edge.

Not only is the choice of new vertex coordinates for the edge collapse a continuous problem, but the actual edge collapse operation may be performed continuously in time. We can linearly interpolate the two vertices from their original positions to the final position of the new vertex. This allows us to create smooth transitions as we change the mesh complexity. As described in [Hoppe 1996], we can even perform *geomorphs*, which smoothly transition between versions of the model with widely varying complexity by performing many of these interpolations simultaneously.

In terms of the ability to create identical simplifications, the vertex removal and edge collapse operations are not equivalent. If we collapse an edge to one of its original vertices, we can create *n* of the triangulations possible with the vertex remove, but there are still C(*n*+2)-*n* triangulations that the edge collapse cannot create. Of course, if we allow the edge collapse to choose arbitrary coordinates for its new vertex, it can create infinitely many simplifications that the vertex remove operation cannot create. For a given input model and

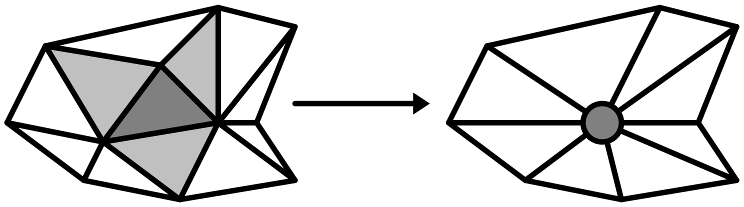


Figure 7: Face collapse operation

desired output complexity, it is not clear which type of operation can achieve a closer approximation to the input model.

The edge collapse was used by [Hoppe et al. 1993] as part of a mesh optimization process that employed the vertex remove and edge swap operations as well (the edge swap is a discrete operation that takes two triangles sharing an edge and swaps which pair of opposite vertices are connected by the edge). In [Hoppe 1996], the vertex remove and edge swaps are discarded, and the edge collapse alone is chosen as the simplification operation, allowing a simpler system that can take advantage of the features of the edge collapse. Although systems employing multiple simplification operations might possibly result in better simplifications, they are generally more complex and cannot typically take advantage of the inherent features of any one operation.

3.3 Face Collapse

The face collapse operation is similar to the edge collapse operation, except that it is more coarse-grained. All three vertices of a triangular face are merged into a single vertex. This causes the original face to degenerate into a point and three adjacent faces to degenerate into line segments, removing a total of four triangles from the model. The coarser granularity of this operation may allow the simplification process to proceed more quickly, at the expense of the fine-grained local control of the edge collapse operation. Thus, the error is likely to accumulate more quickly for a comparable reduction in complexity. [Hamann 1994, Gieng et al. 1997] use the face collapse operation in their simplification systems. The new vertex coordinates are chosen to lie on a local quadratic approximation to the mesh. Naturally, it is possibly to further generalize these collapse operations to collapse even larger connected portions of the input model. It may even be possible to reduce storage requirements by grouping nearby collapse operations with similar error bounds into larger collapse operations.

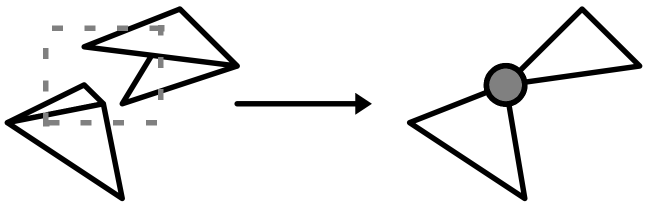


Figure 8: Vertex Cluster operation

Thus, the fine-grained control may be traded for reduced storage and other overhead requirements in certain regions of the model.

3.4 Vertex Cluster

Unlike the preceding simplification operations, the vertex cluster operation relies solely on the geometry of the input (i.e. the vertex coordinates) rather than the topology (i.e. the adjacency information) to reduce the complexity. Like the edge and face collapses, several vertices are merged into a single vertex. However, rather than merging a set of topologically adjacent vertices, a set of “nearby” vertices are merged [Rossignac and Borrel 1992]. For instance, one possibility is to merge all vertices that lie within a particular 3D axis-aligned box. The new, merged vertex may be one of the original vertices that “best represents” the entire set, or it may be placed arbitrarily to minimize some error bound. An important property of this operation is that it can be robustly applied to arbitrary sets of triangles, whereas all the preceding operations assume that the triangles form a connected, manifold mesh.

The effects of this vertex cluster are similar to those of the collapse operations. Some triangles are distorted, whereas others degenerate to a line segment or a point. In addition, there may be coincident triangles, line segments, and points originating from non-coincident geometry. One may choose to render the degenerate triangles as line segments and points, or one may simply not render them at all. Depending on the particular graphics engine, rendering a line or a point may not be much faster than rendering a triangle. This is an important consideration, because achieving a speed-up is one of the primary motivations for simplification.

There is no point in rendering several coincident primitives, so multiple copies are filtered down to a single copy. However, the question of how to render coincident geometry is complicated by the existence of other surface attributes, such as normals and colors. For instance, suppose two triangles of wildly different colors become coincident. No matter what color we render the triangle, it may be noticeably incorrect.

[Rossignac and Borrel 1992] use the vertex clustering operation in their simplification system to perform very fast simplification on arbitrary polygonal models. They partition the model space with a uniform grid, and vertices are collapsed within each grid cell. [Luebke and Erikson 1997] build an octree hierarchy rather than a grid at a single resolution. They dynamically collapse and split the vertices within an octree cell depending on the current size of the cell in screen space as well as silhouette criteria.

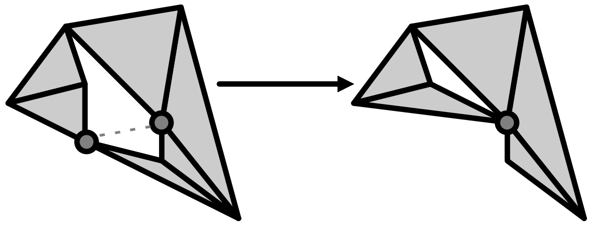


Figure 9: Generalized edge collapse operation

3.5 Generalized Edge Collapse

The generalized edge collapse (or vertex pair) operation combines the fine-grained control of the edge collapse operation with the generality of the vertex cluster operation. Like the edge collapse operation, it involves the merging of two vertices and the removal of degenerate triangles. However, like the vertex cluster operation, it does not require that the merged vertices be topologically connected (by a topological edge), nor does it require that topological edges be manifold.

[Garland and Heckbert 1997] apply the generalized edge collapse in conjunction with error quadrics to achieve simplification that gives preference to the collapse of topological edges, but also allows the collapse of virtual edges (arbitrary pairs of vertices). These virtual edges are chosen somewhat heuristically, based on proximity relationships in the original mesh.

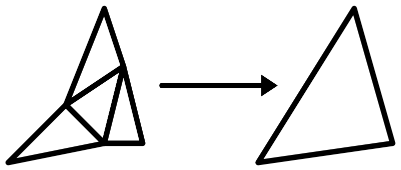


Figure 10: Unsubdivide operation

3.6 Unsubdivide

Subdivision surface representations have also been proposed as a solution to the multi-resolution problem. In the context of simplification operations, we can think of the “unsubdivide” operation (the inverse of a subdivision refinement) as our simplification operation. A common form of subdivision refinement is to split one triangle into four triangles. Thus the unsubdivide operation merges four triangles of a particular configuration into a single triangle, reducing the triangle count by three triangles.

[DeRose et al. 1993] shows how to represent a subdivision surface at some finite resolution as a sequence of wavelet coefficients. The sequence of coefficients is ordered from lower to higher frequency content, so truncating the sequence at a particular point determines a particular mesh resolution. [Eck et al. 1995] presents an algorithm to turn an arbitrary topology mesh into one with the necessary subdivision connectivity. They construct a base mesh of minimal resolution and guide its refinement to come within some tolerance of the original mesh. This new refined subdivision mesh is used in place of the original mesh, and its resolution is controlled according to the wavelet formulation.

4. TOPOLOGICAL CONSIDERATIONS

4.1 Manifold vs. Non-manifold Meshes

Polygonal simplification algorithms may be distinguished according to the type of input they accept. Some algorithms require the input to be a *manifold* triangle mesh, while others accept more general triangle sets. In the continuous domain, a manifold surface is one that is everywhere homeomorphic to an open disc. In the discrete domain of triangle meshes, such a surface has two topological properties. First, every vertex is adjacent to a set of triangles that form a single, complete cycle around the vertex. Second, each edge is adjacent to exactly two triangles. For a manifold mesh with *borders*, these restrictions are slightly relaxed. A border is simply a chain of edges with adjacent triangles only to one side. In a manifold mesh with borders, a vertex may be surrounded by a single, incomplete cycle (i.e. the beginning need not meet the end). Also, an edge may be adjacent to either one or two triangles.

A mesh that does not have the above properties is said to be *non-manifold*. Such meshes may occur in practice by accident or by design. Accidents are possible, for example, during either the creation of the mesh or during conversions between representation, such as the conversion from a solid to a boundary representation. The correction of such accidents is a subject of much interest [Barequet and Kumar 1997, Murali and Funkhouser 1997]. They may occur by design because such a mesh may require fewer triangles to render than a visually-comparable manifold mesh or because such a mesh may be easier to create in some situations. If the non-manifold portions of a mesh are few and far between, we may refer to the mesh as *mostly manifold*.

At the extreme, some data sets take the form of a set of triangles, with no connectivity information whatsoever (sometimes referred to as a “triangle soup”). Such data might turn out to be manifold or non-manifold if we were to attempt to reconstruct the connectivity information. In general, if any conversion has been performed on the original data, it’s safe to assume that a naïve reconstruction will result in at least some non-manifold regions.

The most robust algorithms, based on vertex clusters, operate as easily on a triangle soup as on a perfectly manifold mesh [Rossignac and Borrel 1992], [Luebke and Erikson 1997]. This advantage cannot be stressed enough and is extremely important in the case where the simplification user has no control over the data. The ability to view an large, unfamiliar data set interactively is invaluable in the process of learning its ins and outs, and these algorithms allow one to get up and running quickly.

However, these very general algorithms do not typically create simplifications that look as attractive as those produced by algorithms that operate on manifold meshes. These algorithms, which rely on operations such as the vertex remove or edge collapse, respect the topology of the original mesh and avoid catastrophic changes to the surface and its appearance. The manifold input criterion does limit the applicability of these algorithms to some real-world models, but many of these algorithms may be modified to handle mostly manifold meshes by avoiding simplification of the non-manifold regions. This can be an effective strategy until the non-manifold regions begin to dominate the surface complexity.

The vertex pair and edge collapse operations can both operate on non-manifold meshes as well as manifold ones. Vertex-pair algorithms must deal with the non-manifold meshes they are bound to create by merging non-adjacent vertices. Edge collapse algorithms can operate on non-manifold meshes, but it may be difficult to adapt the most rigorous error metrics for manifold meshes to use on non-manifold meshes.

4.2 Topology Preservation

The topological structure of a polygonal surface typically refers to features such as its *genus* (number of topological holes, e.g. 0 for a sphere, 1 for a torus or coffee mug) and the number and arrangement of its borders. These features are fully determined by the adjacency graph of the vertices, edges, and faces of a polygonal mesh. For manifold meshes with no borders (i.e. closed surfaces), the Euler equation holds:

$$F - E + V = 2 - G, \qquad (2)$$

where F is the number of faces, E is the number of edges, V is the number of vertices, and G is the genus.

In addition to this combinatorial description of the topological structure, the embedding of the surface in 3-space impacts its perceived topology in 3D renderings. Generally, we expect the faces of a surface to intersect only at their shared edges and vertices.

Most of the simplification operations described in section 3 (all except the vertex cluster and the generalized edge collapse) preserve the connectivity structure of the mesh. If a simplification algorithm uses such an operation and also prevents local self-intersections (intersections within the adjacent neighborhood of the operation), we say the algorithm *preserves local topology*. If the algorithm prevents any self-intersections in the entire mesh, we say it *preserves global topology*.

If the simplified surface is to be used for purposes other than rendering (e.g. finite element computations), topology preservation may be essential. For rendering applications, however, it is not always necessary. In fact, it is often possible to construct simplifications with fewer polygons for a given error bound if topological modifications are allowed.

However, some types of topological modifications may have a dramatic impact on the appearance of the surface. For instance, many meshes are the surfaces of solid objects. For example, consider the surface of a thin, hollow cylinder. When the surface is modified by more than the thickness of the cylinder wall, the interior surface will intersect the outer surface. This can cause artifacts that cover a large area on the screen. Problems also occur when polygons with different color attributes become coincident.

Certain types of topological changes are clearly beneficial in reducing complexity, and have a smaller impact on the rendered image. These include the removal of topological holes and thin features (such as the antenna of a car). Topological modifications are encouraged in [Rossignac and Borrel 1992], [Luebke and Erikson 1997], [Garland and Heckbert 1997] and [Erikson and Manocha 1998] and controlled modifications are performed in [He et al. 1996] and [El-Sana and Varshney 1997].

5. LEVEL-OF-DETAIL REPRESENTATIONS

We can classify the possible representations for level-of-detail models into two broad categories: *static* and *dynamic*. Static levels of details are computed totally off-line. They are fully determined as a pre-process to the visualization program. Dynamic levels of detail are typically computed partially off-line and partially on-line within the visualization program. We now discuss these representations in more detail.

5.1 Static Levels of Detail

The most straightforward level-of-detail representation for an object is a set of independent meshes, where each mesh has a different number of triangles. A common heuristic for the generation of these meshes is that the complexity of each mesh should be reduced by a factor of two from the previous mesh. Such a heuristic generates a reasonable range of complexities, and requires only twice as much total memory as the original representation.

It is common to organize the objects in a virtual environment into a hierarchical *scene graph* [van Dam 1988, Rohlf and Helman 1994]. Such a scene graph may have a special type of node for representing an object with levels of detail. When the graph is traversed, this level-of-detail node is evaluated to determine which child branch to traverse (each branch represents one of the levels of detail). In most static level-of-detail schemes, the children of the level-of-detail nodes are the leaves of the graph. [Erikson and Manocha 1998] presents a scheme for generating *hierarchical levels of detail*. This scheme generates level-of-detail nodes throughout the hierarchy rather than just at the leaves. Each such interior level-of-detail node involves the merging of objects to generate even simpler geometric representations. This overcomes one of the previous limitations of static levels of detail — the necessity for choosing a single scale at which objects are identified and simplified.

The transitions between these levels of detail are typically handled in one of three ways: discrete, blended, or morphed. The discrete transitions are instantaneous switches; one level of detail is rendered during one frame, and a different level of detail is rendered during the following frame. The frame at which this transition occurs is typically determined based on the distance from the object to the viewpoint. This technique is the most efficient of the three transition types, but also results in the most noticeable artifacts.

Blended transitions employ alpha-blending to fade between the two levels of detail in question. For several frames, both levels of detail are rendered (increasing the rendering cost during these frames), and their colors are blended. The blending coefficients change gradually to fade from one level of detail to the other. It is possible to blend over a fixed number of frames when the object reaches a particular distance from the viewpoint, or to fade over a fixed range of distances [Rohlf and Helman 1994]. If the footprints of the objects on the screen are not identical, blending artifacts may still occur at the silhouettes.

Morphed transitions involve gradually changing the shape of the surface as the transition occurs. This requires the use of some correspondence between the two levels of detail. Only one representation must be rendered for each frame of the transition, but the vertices require some interpolation each frame. For instance, [Hoppe 1996] describes the *geomorph* transition for levels of detail created by a sequence of edge collapses. The simpler level of detail was originally generated by collapsing some number of vertices, and we can create a transition by simultaneously interpolating these vertices from their positions on one level of detail to their positions on the other level of detail. Thus the number of triangles we render during the transition is equal to the maximum of the numbers of triangles in the two levels of detail. It is also possible to morph using a mutual tessellation of the two levels of detail, as in [Turk 1992], but this requires the rendering of more triangles during the transition frames.

5.2 Dynamic Levels of Detail

Dynamic levels of detail provide representations that are more carefully tuned to the viewing parameters of each particular rendered frame. Due to the sheer number of distinct representations this requires, each representation cannot simply created and stored independently. The common information among these representations is used to create a single representation for each simplified object. From this unified representation, a geometric representation that is tuned to the current viewing parameters is extracted. The coherence of the viewing parameters enables incremental modifications to the geometry rendered in the previous frame; this makes the extraction process feasible at interactive frame rates.

[Hoppe 1996] presents a representation called the *progressive mesh*. This representation is simply the original object plus an ordered list of the simplification operations performed on the object. It is generally more convenient to reverse the order of this intuitive representation, representing the simplest *base mesh* plus the inverse of each of the simplification operations. Applying all of these inverse operations to the base mesh will result in the original object representation. A particular level of detail of this progressive mesh is generated by performing some number of these operations.

In [Hoppe 1997], the progressive mesh is reorganized into a vertex hierarchy. This hierarchy is a tree that captures the dependency of each simplification operation on certain previous operations. Similar representations include the *merge tree* of [Xia et al. 1997], the *multiresolution model* of [Klein and Krämer 1997], the *vertex tree* of [Luebke and Erikson 1997], and the *multi-triangulation* of [DeFloriani et al. 1997]. Such hierarchies allow selective refinement of the geometry based on various metrics for screen-space deviation, normal deviation, color deviation, and other important features such as silhouettes and specular highlights. A particular level of detail may be expressed as a *cut* through these graphs, or a *front* of vertex nodes. Each frame, the nodes on the current front are examined, and may cause the graph to be refined at some of these nodes.

[DeFloriani et al. 1997] discuss the properties of such hierarchies in terms of graph characteristics. Examples of these properties include compression ratio, linear growth, logarithmic height, and bounded width. They discuss several different methods of constructing such hierarchies and test these methods on several benchmarks. For example, one common heuristic for building these hierarchies is to choose simplification operations in a greedy fashion according to an error metric. Another method is to choose a set of operations with disjoint areas of influence on the surface and apply this entire set before choosing the next set. The former method does not guarantee logarithmic height, whereas the latter does. Such height guarantees can have practical implications in terms of the length of the chain of dependent operations that must be performed in order to achieve some particular desired refinement.

[DeRose et al. 1993] present a wavelet-based representation for surfaces constructed with subdivision connectivity. [Eck et al. 1995] make this formulation applicable to arbitrary triangular meshes by providing a remeshing algorithm to approximate an arbitrary mesh by one with the necessary subdivision connectivity. Both the remeshing and the filtering/reconstruction of the wavelet representation provide bounded error on the surfaces generated. [Lee et al. 1998] provide an alternate remeshing algorithm based on a smooth, global parameterization of the input mesh. Their approach also allows the user to constrain the parameterization at vertices or along edges of the original mesh to better preserve important features of the input.

5.3 Comparison

Static levels of detail allow us to perform simplification entirely as a pre-process. The real-time visualization system performs only minimal work to select which level of detail to render at any given time. Because the geometry does not change, it may be rendered in retained mode (i.e. from cached, optimized *display lists*). Retained-mode rendering should always be at least as fast as immediate mode rendering, and is much faster on most current high-end hardware. Perhaps the biggest shortcoming of using static levels of detail is that they require that we partition the model into independent “objects” for the purpose of simplification. If an object is large with respect to the user or the environment, especially if the viewpoint is often contained inside the object, little or no simplification may be possible.

This may require that such objects be subdivided into smaller objects, but switching the levels of detail of these objects independently causes visible cracks, which are non-trivial to deal with.

Dynamic levels of detail perform some of simplification as a pre-process, but defer some of the work to be computed by the real-time visualization system at run time. This allows us to provide more fine-tuning of the exact tessellation to be used, and allows us to incorporate more view-dependent criteria into the determination of this tessellation. The shortcoming of such dynamic representations is that they require more computation in the visualization system as well as the use of immediate mode rendering. Also, the memory requirements for such representations are often somewhat larger than for the static levels of detail.

6. SURFACE DEVIATION ERROR BOUNDS

Measuring the deviation of a polygonal surface as a result of simplification is an important component of the simplification process. This surface deviation error gives us an idea of the quality of a particular simplification. It helps guide the simplification process to produce levels of detail with low error, determine when it is appropriate to show a particular level of detail of a given surface, and optimize the levels of detail for an entire scene to achieve a high overall image quality for the complexity of the models actually rendered.

6.1 Distance Metrics

Before discussing the precise metrics and methods used by several researchers for measuring surface deviation, we consider two formulations of the distance between two surfaces. These are the Hausdorff distance and the mapping distance. The Hausdorff distance is a well-known concept from topology, used in image processing as well as surface modeling, and the mapping distance is a commonly used metric for parametric surfaces.

6.1.1 Hausdorff Distance

The Hausdorff distance is a distance metric between point sets. Given two sets of points, *A* and *B*, the Hausdorff distance is defined as

$$\mathrm{H}(A,B) = \max(\mathrm{h}(A,B), \mathrm{h}(B,A)), \qquad (3)$$

where

$$\mathrm{h}(A,B) = \max_{a \in A} \min_{b \in B} \| a - b \| . \tag{4}$$

Thus the Hausdorff distance measures the farthest distance from a point in one point set to its closest point in the other point set (notice that h(A,B) ≠ h(B,A)). Because a surface is a particular type of continuous point set, the Hausdorff distance provides a useful measure of the distance between two surfaces.

6.1.2 Mapping Distance

The biggest shortcoming of the Hausdorff distance metric for measuring the distance between surfaces is that it makes no use of the point neighborhood information inherent in the surfaces. The function h(A,B) implicitly assigns to each point of surface A the closest point of surface B. However, this mapping may have discontinuities. If points i and j are "neighboring" points on surface A (i.e. there is a path on the surface of length no greater than ε that connects them), their corresponding points, i' and j', on surface B may not be neighboring points. In addition, the mapping implied by h(A,B) is not identical to the mapping implied by h(B,A).

For the purpose of simplification, we would like to establish a continuous mapping between the surface's levels of detail. Ideally, the correspondences described by this mapping should coincide with a viewer's perception of which points are "the same" on the surfaces. Given such a continuous mapping

$$\mathrm{F}: A \rightarrow B$$

the mapping distance is defined as

$$\mathrm{D}(\mathrm{F}) = \max_{a \in A} \| a - \mathrm{F}(a) \| . \tag{5}$$

Because there are many such mappings, there are many possible mapping distances. The minimum mapping distance is simply

$$\mathrm{D}_{\min} = \min_{\mathrm{F} \in M} \mathrm{D}(\mathrm{F}) , \tag{6}$$

where *M* is the set of all such continuous mapping functions. Note that although D_{min} and its associated mapping function may be difficult to compute, all continuous mapping functions provide an upper bound on D_{min}.

6.2 Surface Deviation Algorithms

We now classify several simplification algorithms according to how they measure the surface deviation error of their levels of detail.

6.2.1 Mesh Optimization

[Hoppe et al. 1993] pose the simplification problem in terms of optimizing an energy function. This function has terms corresponding to number of triangles, surface deviation error, and a heuristic spring energy. To quantify surface deviation error, they maintain a set of point samples from the original surface and their closest distance to the simplified surface. The sum of squares of these distances is used as the surface deviation component of the energy function. The spring energy term is required because the surface deviation error is only measured in one direction: it approximates the closest distance from the original surface to the simplified surface, but not vice versa. Without this term, small portions of the simplified surface can deviate quite far from the original surface, as long as all the point samples are near to some portion of the simplified surface.

6.2.2 Vertex Clustering

[Rossignac and Borrel 1993] present a simple and general algorithm for simplification using vertex clustering. The vertices of each object are clustered using several different sizes of uniform grid. The surface deviation in this case is a Hausdorff distance and must be less than or equal to the size of grid cell used in determining the vertex clusters. This is a very conservative bound, however. A slightly less conservative bound is the maximum distance from a vertex in the original cluster to the single representative vertex after the cluster is collapsed. Even this bound is quite conservative in many cases; the actual maximum deviation from the original surface to the simplified surface may be considerably smaller than the distance the original vertices travel during the cluster operation.

[Luebke and Erikson 1997] take a similar approach, but their system uses an octree instead of a single-resolution uniform grid. This allows them to take a more dynamic approach, folding and unfolding octree cells at run-time and freely merging nearby objects. The measure of surface deviation remains the same, but they allow a more flexible choice of error tolerances in their run-time system. In particular, they use different tolerances for silhouette and non-silhouette clusters.

6.2.3 Superfaces

[Kalvin and Taylor 1996] present an efficient simplification algorithm based on merging adjacent triangles to form polygonal patches, simplifying the boundaries of these patches, and finally retriangulating the patches themselves. This algorithm guarantees a maximum deviation from vertices of the original surface to the simplified surface and from vertices of the simplified surface to the original surface. Unfortunately, even this bidirectional bound does not guarantee a maximum deviation between points on the simplified surface and points on the original surface. For instance, suppose we have two adjacent triangles that share an edge, forming a non-planar quadrilateral. If we retriangulate this quadrilateral by performing an edge swap operation, the maximum deviation between these two surfaces is non-zero, even though their four vertices are unchanged (thus the distance measured from vertex to surface is zero).

6.2.4 Error Tolerance Volumes

[Guéziec 1995] presents a simplification system that measures surface deviation using error volumes built around the simplified surface. These volumes are defined by spheres, specified by their radii, centered at each of the simplified surface's vertices. We can associate with any point in a triangle a sphere whose radius is a weighted average of the spheres of the triangle's vertices. The error volume of an entire triangle is the union of the spheres of all the points on the triangle, and the error volume of a simplified surface is the union of the error volumes of its triangles. As edge collapses are performed, not only are the coordinates of the new vertex computed, but new sphere radii are computed such that the new error volume contains the previous error volume. The maximum sphere radius is a bound on the Hausdorff distance of the simplified surface from the original, and thus provides a bound for surface deviation in both 3D and 2D (after perspective projection).

6.2.5 Simplification Envelopes

The simplification envelopes technique of [Cohen and Varshney et al. 1996] bounds the Hausdorff distance between the original and simplified surfaces without actually making measurements during the simplification process. For a particular simplification, the input surface is surrounded by two envelope surfaces, which are constructed to deviate by no more than a specified tolerance, ε, from the input surface. As the simplification progresses, the modified triangles are tested for intersection with these envelopes. If no intersections occur, the simplified surface is within distance ε from the input surface. Similar constructions are built to constrain error around the borders of bordered surfaces. By including extensive self-intersection testing as well, the algorithm provides complete global topology preservation. This algorithm does an excellent job at generating small-triangle-count surface approximations for a given error bound. The biggest limitations are the up-front processing costs required for envelope construction (for each level of detail to be generated) and the conservative nature of the envelopes themselves, which do not expand beyond the point of self-intersection.

6.2.6 Error Quadrics

[Ronfard and Rossignac 1996] describe a fast method for approximating surface deviation. They represent surface deviation error for each vertex as a sum of squared distances to a set of planes. The initial set of planes for each vertex are the planes of its adjacent faces. As vertices are merged, the sets of planes are unioned. This metric provides a useful and efficient heuristic for choosing an ordering of edge collapse operations, but it does not provide any guarantees about the maximum or average deviation of the simplified surface from the original.

[Garland and Heckbert 1997] present some improvements over [Ronfard and Rossignac 1996]. The error metric is essentially the same, but they show how to approximate a vertex's set of planes by a quadric form (represented by a single 4x4 matrix). These matrices are simply added to propagate the error as vertices are merged. Using this metric, it is possible to choose an optimal vertex placement that minimizes the error. In addition, they allow the merging of vertices that are not joined by an edge, allowing increased topological modification. [Erikson and Manocha 1998] further improve this technique by automating the process of choosing which non-edge vertices to collapse and by encouraging such merging to preserve the local surface area.

6.2.7 Mapping Error

[Bajaj and Schikore 1996] perform simplification using the vertex remove operation, and measure surface deviation using local, bijective (one-to-one and onto) mappings in the plane between points on the surface just before and just after the simplification operation. This approach provides a fairly tight bound on the maximum deviation over all points on the surface, not just the vertices (as does [Guéziec 1995]) and provides pointwise mappings between the original and simplified surfaces.

A similar technique is employed by [Cohen et al. 1997], who perform mappings in the plane for the edge collapse operation. They present rigorous and efficient techniques for finding a plane in which to perform the mapping, as well as applying the mapping and propagating error from operation to operation. The computed mappings are used not only to guide the simplification process in its choice of operations, but also to assign texture coordinates to the post-collapse vertices and to control the switching of levels of detail in interactive graphics applications.

6.2.8 Hausdorff Error

[Klein et al. 1996] measure a one-sided Hausdorff distance (with appropriate locality restrictions) between the original surface and the simplified surface. By definition, this approach produces the smallest possible bound on maximum one-sided surface deviation, but the one-sided formulation does not guarantee a true bound on overall maximum deviation. At each step of the simplification process, the Hausdorff distance must be measured for each of the original triangles mapping to the modified portion of the surface. The computation time for each simplification operation grows as the simplified triangles cover more and more of the mesh, but of course, there are also fewer and fewer triangles to simplify. [Klein and Krämer 1997] present an efficient implementation of this algorithm.

6.2.9 Memory-efficient Simplification

[Lindstrom and Turk 1998] demonstrate the surprising result that good simplifications are possible without measuring anything with respect to the original model. All errors in this method are measured purely as incremental changes in the local surface. The error metric used preserves the total volume while minimizing volume changes of each triangle. Another interesting aspect of this work is that they perform after-the-fact measurements to compare the "actual" mean and maximum simplification errors of several algorithm implementations. These measurement use the *Metro* geometric comparison tool [Cignoni et al. 1996], which uniformly samples the simplified surface, computes correspondences with the original surface, and measures the error of the samples.

7. APPEARANCE ATTRIBUTE PRESERVATION

We now classify several algorithms according to how they preserve the appearance attributes of their input models.

7.1 Scalar Field Deviation

The mapping algorithm presented in [Bajaj and Schikore 1996] allows the preservation of arbitrary scalar fields across a surface. Such scalar fields are specified at the mesh vertices and linearly interpolated across the triangles. Their approach computes a bound on the maximum deviation of the scalar field values between corresponding points on the original surface and the simplified surface.

7.2 Color Preservation

[Hughes et al. 1996] describes a technique for simplifying colored meshes resulting from global illumination algorithms. They use a logarithmic function to transform the vertex colors into a more perceptually linear space before applying simplification. They also experiment with producing mesh elements that are quadratically- or cubically-shaded in addition to the usual linearly-shaded elements.

[Hoppe 1996] extends the error metric of [Hoppe et al. 1993] to include error terms for scalar attributes and discontinuities as well as surface deviation. Like the surface deviation, the scalar attribute deviation is measured as a sum of squared Euclidean distances in the attribute space (e.g. the RGB color cube). The distances are again measured between sampled points on the original surface and their closest points on the simplified surface. This metric is useful for prioritizing simplification operations in order of increasing error. However, it does not provide much information about the true impact of attribute error on the final appearance of the simplified object on the screen. A better metric should incorporate some degree of area weighting to indicate how the overall illuminance of the final pixels may be affected.

[Erikson and Manocha 1998] present a method for measuring the maximum attribute deviation in Euclidean attribute spaces. Associated with each vertex is an attribute volume for each attribute being measured. The volume is a disc of the appropriate dimension (i.e. an interval in 1D, a circle in 2D, a sphere in 3D, etc.). Each attribute volumes is initially a point in the attribute space (an *n*-disk with radius zero). As vertex pairs are merged, the volumes grow to contain the volumes of both vertices.

[Garland and Heckbert 1998] extend the algorithm of [Garland and Heckbert 1997] to consider color and texture coordinate error as well as geometry. The error quadrics are lifted to higher dimensions to accommodate the combined attribute spaces (e.g. 3 dimensions for RGB color and 2 dimensions for texture coordinates). The associated form matrices grow quadratically with the dimension, but standard hardware-accelerated rendering models typically require a dimension of 9 or less. The error is thus measured and optimized for all attributes simultaneously. The method makes the simplifying assumption that the errors in all these attribute values may be measured as in a Euclidean space.

[Certain et al. 1996] present a method for preserving vertex colors in conjunction with the wavelet representation for subdivision surfaces [DeRose et al. 1993]. The geometry and color information are stored as two separate lists of wavelet coefficients. Coefficients may be added or deleted from either of these lists to adjust the complexity of the surface and its geometric and color errors. They also use the surface parameterization induced by the subdivision to store colors in texture maps to render as textured triangles for machines that support texture mapping in hardware.

[Bastos et al. 1997] use texture maps with bicubic filtering to render the complex solutions to radiosity illumination computations. The radiosity computation often dramatically increases the number of polygons in the input mesh in order to create enough vertices to store the resulting colors. Storing the colors instead in texture maps removes unnecessary geometry, reducing storing requirements and rasterization overhead.

The appearance-preserving simplification technique of [Cohen et al. 1998] is in some sense a generalization of this "radiosity as textures" work. Colors are stored as texture maps before the simplification is applied. Mappings are computed as in [Cohen et al. 1997], but this time in the 2D texture domain, effectively measuring the 3D displacements of a texture map as a surface is simplified. Whereas [Bastos et al. 1997] reduces geometry complexity to that of the pre-radiositized mesh, [Cohen et al. 1998] simplify complex geometry much farther, quantifying the distortions caused by the simplification of non-planar, textured surfaces. [Cignoni et al. 98] describe a method for compactly storing attribute values into map structures that are customized to a particular simplified mesh.

7.3 Normal Vector Preservation

[Xia et al. 1997] associate a cone of normal vectors with each vertex during their simplification preprocess. These cones initially have an angle of zero, and grow to contain the cones of the two vertices merged in an edge collapse. Their run-time, dynamic simplification scheme uses this range of normals and the light direction to compute a range of reflectance vectors. When this range includes the viewing direction, the mesh is refined, adapting the simplification to the specular highlights. The results of this approach are visually quite compelling, though they do not allow increased simplification of the highlight area as it gets smaller on the screen (i.e. as the object gets farther from the viewpoint).

[Klein 1998] maintains similar information about the cone of normal deviation associated with each vertex. The refinement criterion takes into account the spread of reflected normals (i.e. the specular exponent, or shininess) in addition to the reflectance vectors themselves. Also, refinement is performed in the neighborhood of silhouettes with respect to the light sources as well as specular highlights. Again, this normal deviation metric does not allow increased simplification in the neighborhood of the highlights and light silhouettes as the object gets smaller on the screen.

[Cohen et al. 1998] apply their appearance-preserving technique to normals as well as colors by storing normal vectors in normal maps. Figure 11 shows a view of a complex "armadillo" model. Applying the appearance-preserving algorithm to this model generates the simplified versions of Figure 12 and Figure 13, in which it is nearly impossible to distinguish the simplifications from the original. Compared this to the bunnies in Figure 3 and Figure 4. Although the positions of the surfaces are preserved quite well, as evidenced by the similarity of the silhouettes of the bunnies, the shading makes it quite easy to tell which bunnies have been simplified and which have not (i.e. the appearance has not been totally preserved).

The appearance-preserving approach to normal preservation has the advantage that the normal values need not be considered in the simplification process – only texture distortion error constrains the simplification process. In fact, the error in the resulting images can be characterized entirely by the number of pixels of deviation of the textured surface on the screen. The major disadvantage to this approach is that it assumes a per-pixel lighting model is applied to shade the normal-mapped triangles. Per-pixel lighting is still too computationally expensive for most graphics hardware, though support for such lighting is making its way into standard graphics APIs such as OpenGL.

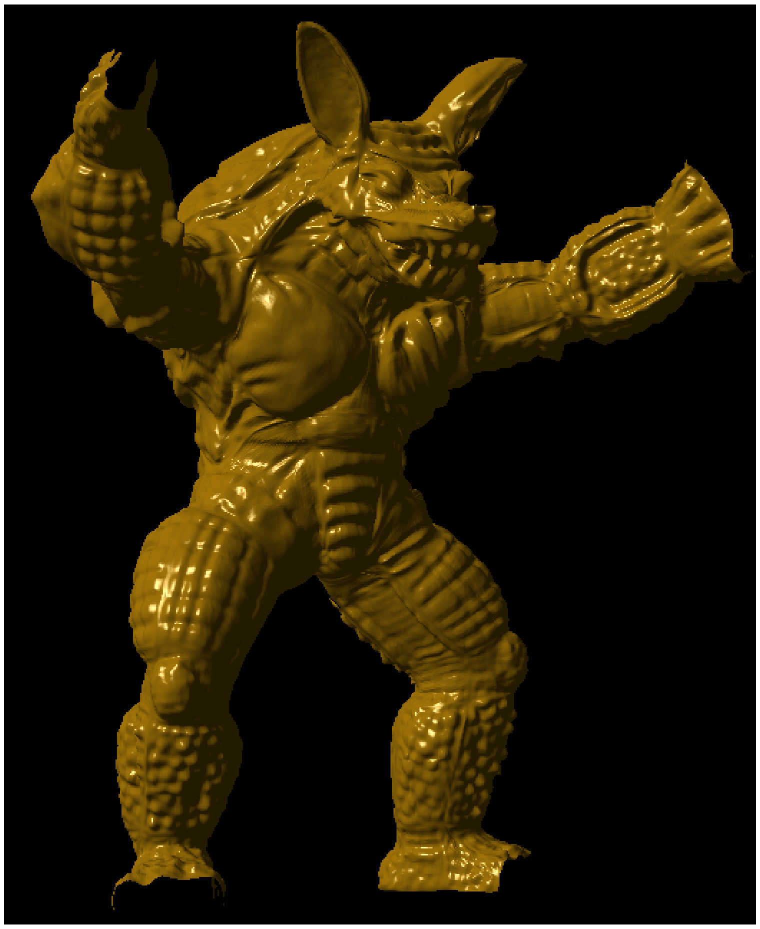


Figure 11: “Armadillo” model: 249,924 triangles

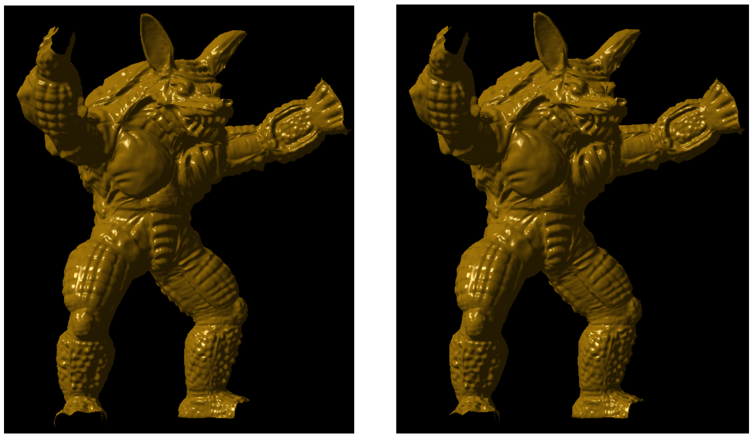

249,924 triangles 7,809 triangles

Figure 12: Medium-sized “armadillos”

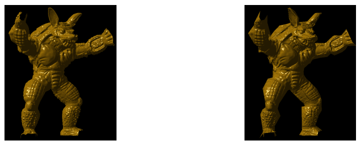

249,924 triangles 975 triangles

Figure 13: Small-sized “armadillos”

8. CONCLUSIONS

As is the case for many classes of geometric algorithms, there does not seem to be any single best simplification algorithm or scheme. An appropriate scheme depends not only on the characteristics of the input models, but also the final application to which the multi-resolution output will be applied.

For poorly-behaved input data (mostly non-manifold or triangle soups), the vertex clustering algorithms [Rossignac and Borrel 1992], [Luebke and Erikson 1997] should yield the fastest and most painless success. For cleaner input data, one of the many methods which respect topology will likely produce more appealing results.

When even pre-computation time is of the essence, a fast algorithm such as [Garland and Heckbert 1997] may be appropriate, while applications required better-controlled visual fidelity should invest some extra pre-computation time in an algorithm such as [Cohen et al. 1998], [Guéziec 1995], or [Hoppe 1996], to achieve guaranteed or at least higher quality.

For applications and machines with extra processing power to spare, dynamic level of detail techniques such as [Hoppe 1997] and [Luebke and Erikson 1997] can provide smooth level-of-detail transitions with minimal triangle counts. However, for applications requiring maximal triangle throughput (including display lists) or need to actually employ their CPU(s) for application-related processing, static levels of detail (possibly with geomorphs between levels of detail) are often preferable (they also add less complexity to application code).

The construction and use of levels of detail have become essential tools for accelerating the rendering process. The field has now reached a level of maturity at which there is a rich “bag of tricks” from which to choose when considering the use of levels of detail for a particular application. Making sense of the available techniques as well as when and how well they work is perhaps the next step towards answering the question, “What is a good simplification?”, both statically, and over the course of an interactive application.

9. ACKNOWLEDGMENTS

We gratefully acknowledge Greg Angelini, Jim Boudreaux, and Ken Fast at the Electric Boat division of General Dynamics for the submarine model; the Stanford Computer Graphics Laboratory for the bunny and model; and Venkat Krishnamurthy and Marc Levoy at the Stanford Computer Graphics Laboratory and Peter Schröder for the “armadillo” model. Thanks to Amitabh Varshney of the State University of New York at Stonybrook for the original material for the section on optimality. Finally, thanks to Dinesh Manocha, my advisor, for the continued guidance that led to the completion of my dissertation.

10. REFERENCES

Agarwal, Pankaj K. and Subhash Suri. Surface Approximation and Geometric Partitions. *Proceedings of 5th ACM-SIAM Symposium on Discrete Algorithms.* 1994. pp. 24-33.

Bajaj, Chandrajit and Daniel Schikore. Error-bounded Reduction of Triangle Meshes with Multivariate Data. *SPIE.* vol. 2656. 1996. pp. 34-45.

Barequet, Gill and Subodh Kumar. Repairing CAD Models. *Proceedings of IEEE Visualization '97.* October 19-24. pp. 363-370, 561.

Bastos, Rui, Mike Goslin, and Hansong Zhang. Efficient Rendering of Radiosity using Texture and Bicubic Interpolation. *Proceedings of 1997 ACM Symposium on Interactive 3D Graphics.*

Brönnimann, H. and Michael T. Goodrich. Almost Optimal Set Covers in Finite VC-Dimension. *Proceedings of 10th Annual ACM Symposium on Computational Geometry.* 1994. pp. 293-302.

Certain, Andrew, Jovan Popovic, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive Multiresolution Surface Viewing. *Proceedings of SIGGRAPH 96.* pp. 91-98.

Cignoni, Paolo, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. A General Method for Recovering Attribute Values on Simplified Meshes. *Proceedings of IEEE Visualization '98.* pp. 59-66, 518.

Cignoni, Paolo, Claudio Rocchini, and Roberto Scopigno. Metro: Measuring Error on Simplified Surfaces. Technical Report B4-01-01-96, Instituto I. E. I.- C.N.R., Pisa, Italy, January 1996.

Clarkson, Kenneth L. Algorithms for Polytope Covering and Approximation. *Proceedings of 3rd Workshop on Algorithms and Data Structures.* 1993. pp. 246-252.

Cohen, Jonathan, Dinesh Manocha, and Marc Olano. Simplifying Polygonal Models using Successive Mappings. *Proceedings of IEEE Visualization '97.* pp. 395-402.

Cohen, Jonathan, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. *Proceedings of ACM SIGGRAPH 98.* pp. 115-122.

Cohen, Jonathan, Amitabh Varshney, Dinesh Manocha, Gregory Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification Envelopes. *Proceedings of SIGGRAPH 96.* pp. 119-128.

Das, G. and D. Joseph. The Complexity of Minimum Convex Nested Polyhedra. *Proceedings of 2nd Canadian Conference on Computational Geometry.* 1990. pp. 296-301.

DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. *Proceedings of IEEE Visualization '97.* pp. 103-110.

DeRose, Tony, Michael Lounsbery, and J. Warren. Multiresolution Analysis for Surfaces of Arbitrary Topology Type. Technical Report TR 93-10-05. Department of Computer Science, University of Washington. 1993.

Dörrie, H. Euler's Problem of Polygon Division. *100 Great Problems of Elementary Mathematics: Their History and Solutions.* Dover, New York.1965. pp. 21-27.

Eck, Matthias, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. *Proceedings of SIGGRAPH 95.* pp. 173-182.

El-Sana, Jihad and Amitabh Varshney. Controlled Simplification of Genus for Polygonal Models. *Proceedings of IEEE Visualization'97.* pp. 403-410.

Erikson, Carl. Polygonal Simplification: An Overview. Technical Report TR96-016. Department of Computer Science, University of North Carolina at Chapel Hill. 1996.

Erikson, Carl and Dinesh Manocha. Simplification Culling of Static and Dynamic Scene Graphs. Technical Report TR98-009. Department of Computer Science, University of North Carolina at Chapel Hill. 1998.

Garland, Michael and Paul Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. *Proceedings of IEEE Visualization '98*. pp. 263-269, 542.

Garland, Michael and Paul Heckbert. Surface Simplification using Quadric Error Bounds. *Proceedings of SIGGRAPH 97.* pp. 209-216.

Gieng, Tran S., Bernd Hamann, Kenneth I. Joy, Gregory L. Schlussmann, and Isaac J. Trotts. Smooth Hierarchical Surface Triangulations. *Proceedings of IEEE Visualization '97.* pp. 379-386.

Guéziec, André. Surface Simplification with Variable Tolerance. *Proceedings of Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS '95).* pp. 132-139.

Hamann, Bernd. A Data Reduction Scheme for Triangulated Surfaces. *Computer Aided Geometric Design.* vol. 11. 1994. pp. 197-214.

He, Taosong, Lichan Hong, Amitabh Varshney, and Sidney Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics.* vol. 2(2). 1996. pp. 171-814.

Heckbert, Paul and Michael Garland. Survey of Polygonal Simplification Algorithms. *SIGGRAPH 97 Course Notes.*1997.

Hoppe, Hugues. Progressive Meshes. *Proceedings of SIGGRAPH 96.* pp. 99-108.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97.* pp. 189-198.

Hoppe, Hugues, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH 93.* pp. 19-26.

Hughes, Merlin., Anselmo Lastra, and Eddie Saxe. Simplification of Global-Illumination Meshes. *Proceedings of Eurographics '96, Computer Graphics Forum.* pp. 339-345.

Kalvin, Alan D. and Russell H. Taylor. Superfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications.* vol. 16(3). 1996. pp. 64-77.

Klein, Reinhard. Multiresolution Representations for Surface Meshes Based on the Vertex Decimation Method. *Computers and Graphics.* vol. 22(1). 1998. pp. 13-26.

Klein, Reinhard and J. Krämer. Multiresolution Representations for Surface Meshes. *Proceedings of Spring Conference on Computer Graphics 1997.* June 5-8. pp. 57-66.

Klein, Reinhard, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. *Proceedings of IEEE Visualization '96.*

Krishnamurthy, Venkat and Marc Levoy. Fitting Smooth Surfaces to Dense Polygon Meshes. *Proceedings of SIGGRAPH 96.* pp. 313-324.

Lindstrom, Peter and Greg Turk. Fast and Memory Efficient Polygonal Simplification. *Proceedings of IEEE Visualization '98*. pp. 279-286, 544.

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97.* pp. 199-208.

Mitchell, Joseph S. B. and Subhash Suri. Separation and Approximation of Polyhedral Surfaces. *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms.* 1992. pp. 296-306.

Murali, T. M. and Thomas A. Funkhouser. Consistent Solid and Boundary Representations from Arbitrary Polygonal Data. *Proceedings of 1997 Symposium on Interactive 3D Graphics.* April 27-30. pp. 155-162, 196.

O'Rourke, Joseph. *Computational Geometry in C.* Cambridge University Press 1994. 357 pages.

Plouffe, Simon and Neil James Alexander Sloan. The Encyclopedia of Integer Sequences. Academic Press 1995. pp. 587.

Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real--Time 3D Graphics. *Proceedings of SIGGRAPH 94.* July 24-29. pp. 381-395.

Ronfard, Remi and Jarek Rossignac. Full-range Approximation of Triangulated Polyhedra. *Computer Graphics Forum.* vol. 15(3). 1996. pp. 67-76 and 462.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering. *Modeling in Computer Graphics.* Springer-Verlag1993. pp. 455-465.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. Technical Report RC 17687-77951. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958. 1992.

Schikore, Daniel and Chandrajit Bajaj. Decimation of 2D Scalar Data with Error Control. Technical Report CSD-TR-95-004. Department of Computer Science, Purdue University. 1995.

Schroeder, William J., Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. *Proceedings of SIGGRAPH 92.* pp. 65-70.

Turk, Greg. Re-tiling Polygonal Surfaces. *Proceedings of SIGGRAPH 92.* pp. 55-64.

van Dam, Andries. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics.* vol. 22(3). 1988. pp. 125-218.

Varshney, Amitabh. Hierarchical Geometric Approximations. Ph.D. Thesis. Department of Computer Science. University of North Carolina at Chapel Hill. 1994.

Xia, Julie C., Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics.* vol. 3(2). 1997. pp. 171-183.a

Hybrid Simplification: Combining Multi-resolution Polygon and Point Rendering

Jonathan D. Cohen
cohen@cs.jhu.edu
Johns Hopkins University

Daniel G. Aliaga
aliaga@bell-labs.com
Lucent Technologies Bell Labs

Weiqiang Zhang
zhangwq@cs.jhu.edu
Johns Hopkins University

Abstract

Multi-resolution hierarchies of polygons and more recently of points are familiar and useful tools for achieving interactive rendering rates. We present an algorithm for tightly integrating the two into a single hierarchical data structure. The trade-off between rendering portions of a model with points or with polygons is made automatically.

Our approach to this problem is to apply a bottom-up simplification process involving not only polygon simplification operations, but point replacement and point simplification operations as well. Given one or more surface meshes, our algorithm produces a hybrid hierarchy comprising both polygon and point primitives. This hierarchy may be optimized according to the relative performance characteristics of these primitive types on the intended rendering platform. We also provide a range of aggressiveness for performing point replacement operations. The most conservative approach produces a hierarchy that is better than a purely polygonal hierarchy in some places, and roughly equal in others. A less conservative approach can trade reduced complexity at the far viewing ranges for some increased complexity at the near viewing ranges.

We demonstrate our approach on a number of input models, achieving primitive counts that are 1.3 to 4.7 times smaller than those of triangle-only simplification.

Keywords: rendering, simplification, multi-resolution, triangles, points, hybrid.

1 INTRODUCTION

Interactive visualizations, which maintain a steady feedback loop with the application user, rely on the ability of the computer and, in particular, the graphics engine to produce images at a high frame rate. Applications with this requirement include the exploration of data through scientific visualization, enhancement of medical procedures through computer-integrated surgery, terrain visualization, production of mechanical systems through CAD visualization and rapid prototyping, and of course the pursuit of entertainment through the high-end video games which have driven the consumer graphics market in recent years.

Most such applications today employ some form of multi-resolution rendering to achieve the necessary balance between the conflicting goals of smooth, interactive performance and useful, high-quality imagery. Multi-resolution rendering uses a hierarchy of rendering primitives, allowing the application to distribute its rendering budget across a complex geometric model to produce such an optimized result.

The rendering primitives used generally depend on the application domain and the method of model design or acquisition. For example, models built from complex polygonal meshes lend themselves to the construction of polygonal hierarchies (some forms are often referred to as levels of detail), built through a process of polygonal simplification. On the other hand, models acquired as a set of points in some form, such as from a camera, laser range-finder, or other device for sampling the physical world, lend themselves naturally to the construction of a point hierarchy through the use of octree-based or other proximity-based point merging schemes. Although these points are in a pure geometric sense infinitesimal, they are usually defined with a radius of extent. Thus, they can be thought of as spheres in world space and, as a matter of rendering efficiency, are typically rasterized as circles or squares in screen space.

In some sense, these representations are interchangeable; both are capable of representing and rendering the same data given a sufficiently high representational resolution. Some applications do, in fact, choose to switch from one domain to another. Point samples may be meshed to produce polygonal models, and polygonal models may be point-sampled and these samples stored to facilitate future rendering. The process of rasterization is itself a conversion from polygons to a set of point samples, so we can clearly establish useful correspondences between triangles and their associated samples, sometimes using them interchangeably.

Both of these representations have merit, but neither is superior for all geometric models under all viewing conditions. Adaptive, view-dependent refinement schemes already employ these multi-resolution representations to adjust the number of primitives used across the model environment to suit the needs of the application's current viewing parameters. So it is natural to consider adapting not only the *number* of primitives but also the *type* of primitives rendered for a particular set of viewing parameters to produce a well-optimized balance of performance and quality. Such a hybrid approach to rendering may produce a system with improved scalability and a wider range of applicability.

1.1 Main Contribution

In this paper, we present a simplification paradigm to *tightly integrate* polygon-based and point-based rendering. Our approach begins with a polygonal model as input, which we proceed to simplify using a standard, greedy simplification procedure (our system employs edge collapse operations). The same optimization criteria that guide the polygon simplification process also trigger the substitution of one or more points for individual triangles as the situation warrants. These points are also merged to produce a complete, hybrid hierarchy. This hierarchy, built entirely as a preprocess, may then be used to perform interactive rendering using adaptive, view-dependent refinement.

Our algorithm provides the following capabilities:

- **Automatic selection:** The algorithm automatically determines where and when a subset of a model is better rendered as triangles or as points.
- **Seamless transition:** The adaptive refinement procedure transitions between triangles and points at a fine granularity.
- **Topology modification:** Multiple manifold surfaces may be merged (and thus more drastically reduced) during the point simplification phase.
- **Error management:** Polygon simplification and point merging are selected as appropriate to reduce geometric error growth within the hierarchy, and guaranteed geometric error bounds from the original surface are provided throughout.

The broader goal of this research is to explore the relative strengths and weaknesses of polygon and point representations, as well as the situations where each is most useful. We also consider how the relative capability of graphics hardware in rendering points versus polygons affects the hierarchies we build. In the long term, we aim to bridge the gap between polygon-based and image-based rendering. Images are essentially specially organized collections of points, and so this research is a stepping stone along the way, providing some useful tools and insights.

1.2 Paper Organization

We proceed by describing in Section 2 some related work in the areas of polygonal simplification and point-based rendering, followed by an overview of our integrated approach in Section 3. After that we review our central data structure, the *multi-resolution graph*, and the off-line and on-line portions of our algorithm in Sections 4, 5, and 6. We conclude with a look at our results, and a discussion of the issues they raise.

2 RELATED WORK

This research draws on previous work in the areas of polygonal simplification and point-based rendering. We now review the most relevant topics in each of these fields.

2.1 Polygonal Simplification

A number of existing polygonal simplification algorithms use priority queue driven, bottom-up decimation strategies [Guéziec 1995, Hoppe 1996, Cohen et al. 1997, Garland and Heckbert 1997]. Of these algorithms, several provide guaranteed bounds on the resulting error between all points on the original surface and all points on the simplified surface (the tightest possible measure being the Hausdorff distance) [Guéziec 1995, Klein et al. 1996, Cohen et al. 1997, Lee et al. 1998]. Our error measure happens to be based on the projection algorithm of [Cohen et al. 1997], but any of this class of guaranteed error measures would do equally well for the purpose of this research.

Several algorithms and hierarchical data structures allow for fine-grained, view-dependent refinement of polygonal models in an interactive setting [Rossignac and Borrel 1993, DeFloriani et al. 1997, Hoppe 1997, Luebke and Erikson 1997, Xia et al. 1997]. Of these, we have found the multi-triangulation data structure of [DeFloriani et al. 1997] to be the most compatible with our current research goals.

The benefits of this research and the properties of our hierarchical models bear some resemblance to those of simplification algorithms that provide for topological modification [Rossignac and Borrel 1993, El-Sana and Varshney 1997, Schroeder 1997, Garland and Heckbert 1998] and merging of low-resolution objects [Erikson and Manocha 1999]. However, none of the existing algorithms provides both fine-grained progressive control and guaranteed surface-to-surface error bounds (other than the most conservative approach of tracking the maximum separation between collapsed vertices).

An interesting piece of research that seems quite similar to ours is the progressive simplicial complex [Popovic and Hoppe 1997]. This data structure, like our multi-resolution graph, also allows for primitives of different types, namely simplices of arbitrary dimension. This general approach allows the simplices to collapse to progressively lower dimension. Our work, in contrast, does not require that triangle vertices be merged to become points, but rather allows this conversion to take place at an arbitrary sampling rate. This allows the hierarchy to be tuned according to a system's relative polygon and point rendering performance characteristics.

2.2 Point-based Rendering

Using points as rendering primitives has a long history in computer graphics [Levoy and Whitted 1985]. Early computer graphics systems used points to render clouds, explosions, and other fuzzy objects [Reeves 1983]. More recently, together with the advent of faster general-purpose CPUs, point rendering has been used to model and render trees, polygonal meshes, and volumetric models [Hoppe et al. 1992, Max and Ohsaki 1995].

The fundamental difficulty of using points is to create a continuous (on-screen) reconstruction of the underlying model. Algorithms leverage the simple rendering calculations of points [Grossman and Dally 1998] to cover surfaces with a sufficient number to samples. Image-Based Rendering (IBR) exploits the screen-coherence of projected points to further accelerate point rendering. By ordering points on a grid and performing incremental computations [McMillan and Bishop 1995], IBR methods can re-project a large number of points (or pixels) each frame.

In our case, we have full knowledge of the underlying model and can choose, a priori, the points to render a model at a desired error tolerance. Thus, we do not need to reconstruct the model. Furthermore, by establishing an error metric over the surface of the model, we have a criterion to sample the model and generate points for an interactive point rendering system [Pfister et al. 2000, Rusinkiewicz and Levoy 2000]. The challenge for our work is to compute, for every neighborhood of a model, when we achieve a win with polygonal rendering and when point rendering is more advantageous.

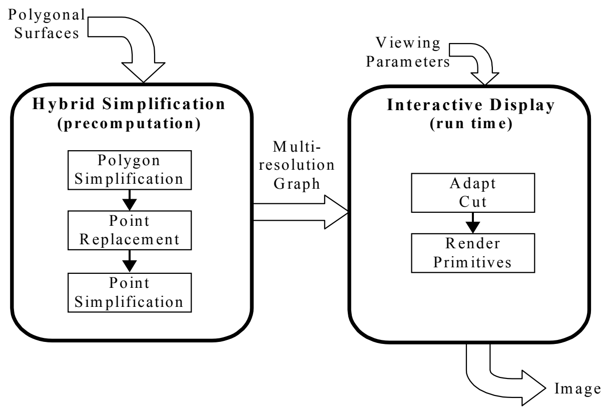


Figure 1: Components of a hybrid multi-resolution rendering system.

3 OVERVIEW

There are many approaches one could take to produce a tight integration of polygons and points in a multi-resolution framework. For example, one could construct two complete hierarchies, one for polygons and one for points, with some type of links describing the mappings between the two. Then all this information would be available at the time of rendering for the best combination of polygons and points to be selected in a view-dependent fashion.

We have opted, at the expense of some flexibility, to pursue a more practical approach of using view-independent information to construct a single hierarchy comprising both polygons and points. Thus all of the important decisions regarding the tradeoffs between the two primitive types have been made before the rendering even begins. This predetermination of the tradeoffs could have negative consequences on how well the decisions are made for any given viewing parameters, but it allows us to build a simple run-time system based on a foundation of well-known algorithms and data structures.

Figure 1 depicts the components of our system. Our hybrid simplification process may be seen at a high level as a simplification algorithm supporting three different simplification operations: polygon simplification (e.g. edge collapse), point replacement, and point simplification. These operations are performed repeatedly in an appropriate order to ultimately produce a hierarchy. Each operation replaces some subset of the model primitives with a new set of primitives, reducing their complexity and perhaps changing their type. In particular, the point replacement operation converts a triangle into one or more points, which may then be further reduced through point merging operations. The set of operations performed, along with the affected primitives and associated error bounds are all stored in a multi-resolution graph data structure.

The interactive rendering system uses the viewing parameters for a given rendering frame to select an appropriate set of primitives from the multi-resolution graph. This set of primitives completely covers the original model (i.e. the entire model is represented by this set) and provides an appropriate resolution. Our current system allows the user to choose a screen-space error tolerance, and the primitives are chosen to be just complex enough to avoid exceeding this tolerance. Because the set of primitives selected is that which lies along a particular *cut* through the graph, and this cut may be modified incrementally from frame to frame, this selection process is referred to as "adapting the cut". Once the primitives are selected, they are rendered to produce the final image. We next present the essential details of this multi-resolution graph before proceeding to the description of the simplification algorithm.

4 MULTI-RESOLUTION GRAPH

Our *multi-resolution graph* (MRG) data structure is an extension to the elegant *multi-triangulation* (MT) data structure described in detail in [DeFloriani et al. 1997, DeFloriani et al. 1998] (Because the extension is to permit the inclusion of new primitive types, the original name is no longer appropriate). The MRG is a simplification hierarchy in the form of a directed acyclic graph. The graph is represented by a set of nodes, *N*, connected by a set of arcs, *A*. There is a unique *source* node at the root of the graph, and a unique *drain* node at the bottom. A small example is shown in Figure 2.

Each node of the MRG represents a change to an underlying geometric model – a refinement if we are traversing downward, or a simplification if we are traversing upward. Thus, as we build this graph (from drain to source, in bottom-up simplification), each of our simplification operations is stored along with its associated error bound as a node.

The primitives of the model are stored with the arcs. The primitives removed from the model by an operation are associated with the child arcs of the operation's node, and those inserted by the operation are associated with its parent arcs (one or more primitives may be stored with an arc). From the arc's perspective, the node beneath it (its *end* node) produces its primitives, and the node above it (its *start* node) consumes them (assuming we are traversing upward).

The arcs represent the dependencies of one mesh operation on another. So, for example, if we wish to perform the refinement indicated by a node, we must first perform the refinement indicated by all of the node's parents. Performing the node's operation amounts to replacing the primitives of a node's parent arcs with those of its child arcs, or vice versa. The model coverage of these two sets of primitives are generally the same to avoid local cracks (i.e. missing surface coverage) or multiple coverage (which

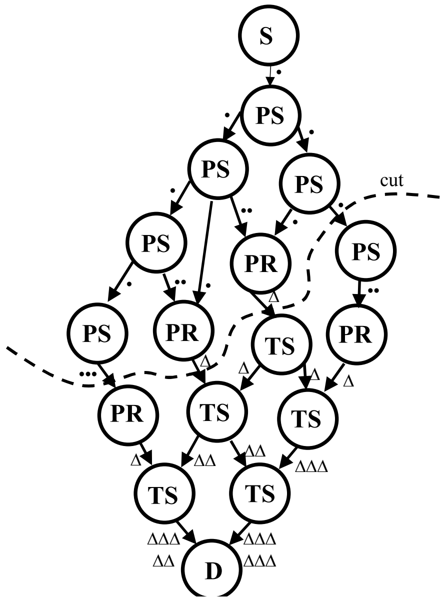


Figure 2: A small multi-resolution graph (11 original triangles). The letters indicate the various node types: D (drain), S (source), TS (triangle simplification), PR (point replacement), and PS (point simplification). The cut contains 2 triangles and 4 points.

may be inefficient, but not necessarily incorrect) across the model.

To extract a connected, consistent representation of the surface, we generate a *cut* of the graph. A cut is a set of arcs that partitions the nodes of the MRG, leaving the source node above the cut, and the drain node below it. In addition, if the cut contains arc *a*, then it must not contain any ancestor or descendent of *a*. The triangles of such a cut represent our input surface at some resolution. In general, we find such a cut by performing a graph traversal starting from the source, testing the error of each visited node against a particular error threshold to decide whether to continue. We can also begin the traversal with an existing cut and move portions of the nodes upward or downward across the cut to modify the local resolution of the surface.

We choose to use this MRG as our simplification graph representation because it has a couple of desirable properties which it inherits directly from the MT.

First, the graph fully specifies all the primitives to be used for all surface resolutions as well as the dependencies between all changes in resolution. Because all the primitives that may be used as part of the rendered model are known in advance, we can provide rigorous bounds on their quality. Not all simplification hierarchies have this property. For example, the well-known simplification hierarchies of [Hoppe 1997] and [Luebke and Erikson 1997] do not have these properties. Looser dependencies may give these hierarchies greater flexibility, because the ordering of vertex merges is not quite so fixed. But the particular triangles that can be extracted from these hierarchies are not known in advance, and vary depending on the order of vertex merges.

Second, the MRG allows for explicit representation of its primitives and a single, general replacement operator. This is incredibly convenient for research purposes when the goal is to explore different primitive types. This very general operation specifies to replace the primitives in set *A* with the primitives in set *B*, without any specific knowledge of the primitive types involved. Operators that allow a more implicit representation of this primitive conversion may produce a more compact data structure, but they are not so convenient for exploration.

5 HYBRID SIMPLIFICATION

As mentioned in Section 3, our simplification process comprises three simplification operations: polygon simplification, point replacement, and point simplification. Although the simplification optimization process could be implemented directly using a single priority queue or queue for each type of operation, we actually separate the simplification process into three distinct components, performing them one after the other in their entirety.

The polygon simplification process explicitly maintains a priority queue of edge collapses that can be used to replace a set of triangles with a smaller set of triangles. A point replacement queue is maintained implicitly in the following way. After computing the optimization value of an edge collapse operation, we evaluate the optimization value of the point replacement operations associated with each of its triangles. If any of these point replacements takes precedence over the edge collapse, we remove the edge collapse from the queue. When the polygon simplification process is finished, point replacement operations are performed on all the remaining triangles. This produces the same result as would an explicit point replacement queue.

Once all the point replacements have been performed, the point simplification process begins. We apply an octree-guided point merging process to simplify the points produced by the replacement operations. The result may differ from a priority-driven point simplification process, but it is efficient and works well in practice.

As the entire simplification process proceeds, we build an MRG from the bottom up (as shown in Figure 2). Each operation we perform adds a node to the graph, and the geometric error bound for the operation is stored with that node. The operation also enables us to create and connect the node's child arcs. The creation of parent arcs is delayed, however, until we know which nodes will consume the primitives created by this operation. If multiple nodes consume these new primitives, then the node gets multiple parents. Notice that each point replacement node has a single child arc containing one triangle, and produces one or more points. Each point simplification node, on the other hand, has at least two child points and produces only a single point. Thus the point simplification nodes can have only one parent arc, and the top portion of our graph is actually a tree.

Now we shall discuss the optimization function used to determine the order of triangle simplification and point replacement operations. We follow this with a more detailed description of each of the three simplification operations.

5.1 Queue Optimization Function

For a given simplification operation, we compute its optimization value as its cost divided by its benefit. The cost is the increase in error, $\Delta\varepsilon$, and the benefit is the decrease in number of primitives, $-\Delta p$, that would occur as a result of performing the operation in question. In fact, we can plot the number of primitives versus the error for the entire simplification process (as in Figure 7), and this optimization function is just the slope of that curve. Thus we attempt to produce a curve in which the error grows as slowly as possible by choosing the operation with the smallest optimization value.

For simplification algorithms that perform only one type of operation, the benefit factor is often unnecessary because it is a constant for all the operations in the process (although for models with borders, operations taking place on the borders generally provide less benefit than those on the interior do).

By ordering both triangle simplification and point replacement operations according to this optimization function, we generally produce error curves that stay entirely below that of a triangle-only simplification process. This outcome relies on the fact that the slope of the point simplification portion of the curve is generally the lowest of all.

However, this conservative ordering delays the introduction of points into the hierarchy, leaving less time to benefit from the small slope of the point simplification. Thus in the interest of producing the best overall curve rather than one which is everywhere beneath the triangle-only simplification curve, we may wish to allow a more aggressive schedule for initiating point replacement operations. To achieve this, we introduce a user-specifiable *transition factor*, τ, which scales the optimization values of all the triangle simplification operations. Setting τ to 1 achieves the same result as the cost/benefit optimization we have already described, whereas setting it to a value greater than 1 will introduce points sooner.

The parameter τ is used to trade an increased primitive count in the lower error ranges for a decreased primitive count in the higher error ranges (seen as a hump in the curve in Figure 8). This is desirable for models of large environments where efficiency for distant portions of the model may be almost as important or more important than efficiency for near portions of the model (because respecting a constant screen-space error tolerance across the model allows greater world-space error for distant portions of the model).

5.2 Triangle Simplification

Our triangle simplification operation is an edge collapse, which merges the two existing vertices of an edge into a single, new vertex. Our implementation is based on the algorithm described in [Cohen et al. 1997], which measures the error of an edge collapse using planar projections. This error is a bound on the Hausdorff distance (a max of min distances) between the original triangles and the simplified triangles. This particular algorithm operates only on manifold surfaces and preserves topology, but this is not a requirement for our hybrid simplification. The only properties we require of a triangle simplification operation is that it provide a guaranteed error bound. (Unfortunately, most operations that allow topology modification or non-manifold inputs do not provide guaranteed error bounds). Thus, a number of other existing algorithms are applicable and may compare favorably [Guéziec 1995, Klein et al. 1996, Lee et al. 1998].

As described earlier, for each edge collapse operation, we also compute the optimization cost of the point replacement of each of its triangles. If any of these point replacements has priority over the edge collapse, the edge collapse is removed from consideration.

5.3 Point Replacement

The point replacement operation provides a means of transitioning from triangles to points in our multi-resolution hierarchy. An important question to consider is how many points we should use to replace a triangle. In a correct MRG, the error always increases as we move upward in the hierarchy. As a matter of principle, we wish to guarantee that the rendering complexity always decreases at the same time. Thus one can always move down the hierarchy to increase quality or up the hierarchy to increase performance. With this in mind, we should never replace a triangle with so many points that the performance is decreased.

To help our system meet this performance constraint, we introduce a system-specific performance ratio, κ, which is the ratio of point-rendering performance to triangle-rendering performance on a particular system. Specifying this ratio correctly should make it possible to generate a hierarchy that is well tuned for the system in question. Now it is clear that we never wish to replace a triangle with more than $\lfloor\kappa\rfloor$ points, because this would actually decrease performance. Notice that if $\kappa \le 1$, then a point replacement can never directly increase performance. Using this definition of κ, we can now refer to a number of primitives in this system, as:

$$primitives = triangles + \frac{points}{\kappa}$$

In practice, we want to perform the point replacement such that it minimizes the optimization value for the operation. Using κ we find that the benefit for a point replacement operation is:

$$benefit = 1 - \frac{replacement\ points}{\kappa}$$

This benefit is just the resulting change in primitive count as we replace a single triangle primitive with a number of points equivalent to a fraction of a triangle primitive. The cost of the point replacement is the associated increase in error due to replacing a triangle with points. We can show that if we completely cover the triangle, this cost is just the point radius.

The main tool we have to work with to solve this discrete optimization problem is a procedure for generating a set of replacement samples, given a specified sampling distance. We next describe this sampling procedure and then a method for optimizing the choice of sampling distance for a triangle.

5.3.1 Sampling Procedure

The sampling procedure takes as input a sampling distance and generates a set of point samples that entirely covers a triangle. For a given sampling distance, we wish to generate as few points as possible to optimize performance. These are the main considerations for the sampling algorithm to work well.

As mentioned above, each sampling point can be treated as a sphere, specified by a center (the sample location) and a radius (the sample's range of coverage). It is intuitively clear that a set of spheres (or circles in the plane of the triangle) must overlap somewhat to completely cover the triangle. Squares, on the other hand, tile the plane quite nicely without overlap. Thus we can tile the plane with circular discs that circumscribe these square tiles.

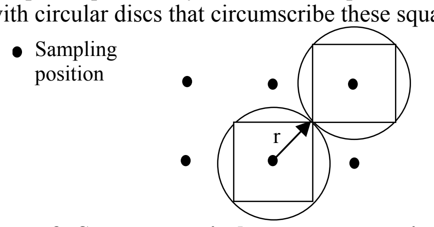


Figure 3: Squares vs. circles to cover a region.

Figure 3 depicts the case in which squares (or circles) just cover the region (i.e. any further separation would lead to a hole). In this sense, it's quite reasonable to use squares to address the sampling problem.

To make the algorithm easy to implement as well as to minimize the number of points, we proceed to sample the triangle one row at a time, as shown in Figure 4. We begin by choosing a coordinate frame such that the largest edge is considered to be horizontal at the bottom of the triangle.

We start the sampling from the left side of edge **AB**, and make the first sampling point cover as much of the triangle as possible but without introducing any gap or hole at the bottom and left side. The following points will be sampled in the same row, which is parallel to edge **AB**, until all the triangle space in that row is covered. This procedure is repeated until the whole triangle is covered. Notice that each row may shift left-to-right with respect to the previous rows, so our sampling does not exactly follow a 2D grid.

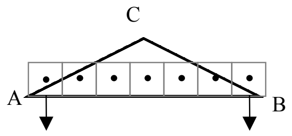


Figure 4: The first row of a triangle is sampled.

In addition to computing the sample's center, we may also wish to sample other attributes, such as color. To make it possible to interpolate rather than extrapolate, we ensure that all the sample centers are located inside or on the edge of the triangle. In Figure 4, if the sampling square has a center outside of the triangle (above edge **AC**), we will push down the square to make the center just located on the edge **AC**. The similar case will occur when the sampling point is close to edge **BC**. To the right side sampling point, we will first move it as left as possible (just cover vertex **B**), then we will adjust it up or down to make it inside the triangle. It is important to get these special boundary conditions right, because when we sample a triangle with a small number of points, all the points may exhibit some boundary condition.

Finally, we need to compute the error bound for a point. In terms of the two-sided Hausdorff distance measure, we know that every point on the triangle is zero distance from a sample sphere, and every point on a sample sphere is within the sphere's radius from a point on the triangle. Thus the incremental error due to sampling is the sphere radius, r. If the sampled triangle already has some error bound ε_Δ, then the total error bound of the point is:

$$\varepsilon_p = r + \varepsilon_\Delta$$

Given the neighboring sample distance, d, the radius of the circumscribing circle is just:

$$r = d/\sqrt{2}$$

5.3.2 Computing Optimum Sample Distance

Now that we can generate a set of samples or, using the same procedure, *count* the number of samples generated for a given triangle and sampling distance, it is possible to effectively optimize the sample distance using a fairly simplistic approach.

First, we would like the ability to find the smallest sampling distance, d, which produces no more than a given number of samples, s. We can initialize d to a value that makes the total area of s square samples equal to the triangle area:

$$Area_\Delta = sd^2 \rightarrow d = \sqrt{\frac{Area_\Delta}{s}}$$

This is the theoretical minimum sampling distance to generate no more than s samples. Then we double d until it produces more than s samples and finally binary search within the resulting interval to find the best sample distance to within some relative tolerance.

Given this discrete function for d as a function of s, we can now optimize for the d that produces the smallest cost/benefit value. The cost value is just the radius, which we have seen in Section 5.3.1 is just a constant multiple of d. The benefit value varies with the number of replacement points, s. Because s as a function of d is a step function, this optimum d will occur just at the top of one of these steps. Thus the most straightforward way to compute the optimum sampling distance is to find d for each integer step with $s<\kappa$ (which we have just described above), and choose the one resulting in the smallest cost/benefit. This works in practice for small values of κ. For larger values, we may wish to estimate a derivative of this step function to provide a faster optimization.

5.4 Point Simplification

After the priority queue has emptied and all remaining point replacements have been applied, we begin the point simplification process by inserting all the original sampling points into the cells of an octree according to the position of the sample center. We then use the octree cells to indicate which sets of points to merge. Each merge can combine up to eight children, and produces exactly one parent point. The center and radius of the new point are chosen such that the parent sphere contains all the child spheres, as shown in Figure 5 (we do not currently use the optimal algorithm, but a simple heuristic). The color of the parent point is a weighted average of those of its children with weights assigned according to surface area.

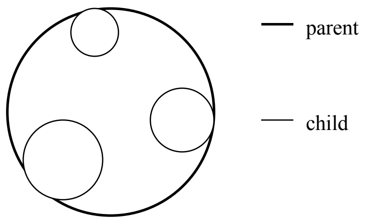


Figure 5: 3 children points are merged into 1 parent point.

Each node of the octree corresponds to one node of the MRG with a single parent arc. As in the case of the original point samples, the radius is only a part of the error bound for the merged point and its generating node; it must be combined with any existing triangle error to compute the total error bound. The total error bound for a point is just its radius plus the maximum triangle error bound component associated with its children (this component is just the child point's error minus its radius). The creation of the root node completes the MRG data structure and the hybrid simplification process.

6 INTERACTIVE DISPLAY

Our interactive display system allows the user to navigate through a 3D environment described by a multi-resolution graph. The model is statically pre-lit with diffuse illumination so that no normals are required for the point primitives (this is still common practice in the image-based rendering community, although current research is gradually reducing this limitation).

The user can select either a screen-space error tolerance, in terms of pixels of deviation, or an object-space error tolerance in terms of a percentage of the length of the environment's bounding box diagonal. Choosing a screen-space tolerance invokes view-dependent refinement every frame as the user navigates the environment. Alternatively, selecting an object-space tolerance causes a one-time refinement to the specified tolerance. This is useful for looking at the various model resolutions up close, and allows navigation without any changes in the model primitives.

As described in Section 4, the set of primitives to be rendered is determined by finding a cut through the graph; the primitives associated with the arcs on the cut are rendered. The cut is adapted by evaluating an error criterion to determine if a node is above or below the current error threshold. For an object-space error tolerance, the stored error is divided by the length of the MRG's bounding box diagonal for comparison with the threshold. For a screen-space threshold, the arc's screen-space depth is computed using a conservative bounding sphere approximation (a bounding sphere is stored with each arc). From this depth, a scaling factor is computed to convert the error length from object-space coordinates to a screen-space pixel distance, which is now comparable with the specified pixel threshold. The same scaling factor is used to convert a point primitive's radius to a screen space radius for rendering as a circle.

Our current implementation renders OpenGL points on a SGI InfiniteReality II platform. Other more efficient point rendering systems have been developed [Grossman and Dally 1998, Pfister et. al 2000, Rusinkiewicz and Levoy 2000], some of which include texture filtering. Any of these can readily be used in the context of our hybrid simplification framework. We hope to take more of a 3D image warping approach in the future, incorporating some form of LDI tree [Chang et al. 1999] or a derivative of it, in order to achieve greater point performance (and thus a larger κ).

7 RESULTS

We have implemented both the hierarchical simplification and interactive display algorithms described in Sections 5 and 6, and tested them on several models. These models are listed in Figure 6. Most of the preprocessing time is spent in the evaluation and prioritization of potential edge collapses. This time is increased somewhat by the optimization of sampling distances for potential point replacement operations, but not excessively for small values of the input parameter κ. The time required for actually generating the samples and performing the octree-based point simplification is typically quite small compared to the rest of the algorithm.

Model	Input Tris	MRG Tris	MRG Points	Simp Time
Armadillo	1,999,404	8,962.427	154,651	111:29
Bunny	69,451	308,738	10,825	1:56
Bronco	74,308	257,694	70,841	2:29
Horse	96,966	432,878	10,429	2:45

Figure 6: Test Models (data reported for κ=3, τ=1; simplification time in minutes:seconds).

It is informative to observe the behavior of the curves produced by plotting the number of primitives versus the object-space error for various choices of the κ and τ parameters. Figure 7 shows such a plot for the Bronco model, with a fixed value of κ=3 and a several different values of the transition factor, τ. Notice that setting τ=1 achieves a curve that is everywhere less than or equal to the curve of triangle simplification alone (this is not actually guaranteed by this non-optimal greedy process). However, if we relax this constraint by increasing τ, we reduce the error values for the lower primitive counts at the expense of increasing it for some of the higher primitive range.

Looking at Figure 8 shows us that the bunny model does not benefit as much from point replacement as much as the Bronco. This is not surprising, because the bunny is a single, highly tessellated manifold surface, while the Bronco comprises 339 individual manifolds, which are not so highly tessellated to begin with. These manifolds are not merged in the triangle domain by our polygon simplification algorithm, so for this reason as well, they benefit from the transition to points.

Figure 9 examines changing the system performance ratio, κ, for a fixed value of τ (τ=1.0). This plot demonstrates, not surprisingly, that if we develop systems with increased levels of point-rendering performance, our hierarchies will directly benefit by switching to points sooner in the simplification process.

In Figures 10 and 11, we report the simplification results for a fixed screen-space error of 3 pixels. We fly-through an environment consisting of 73 Bronco models and record the number of primitives for hybrid simplification with κ=3, τ=1, for hybrid simplification with κ=3, τ=5, and for triangle-simplification only. We are able to reduce the primitive count by an average factor of 1.6 (minimum 1.3, maximum 4.7). As expected, the simplification for τ=5 is slightly better when most of the environment is in the distance (e.g. beginning of the path) and slightly worse when the viewer is mostly surrounded by the environment (e.g. middle of the path).

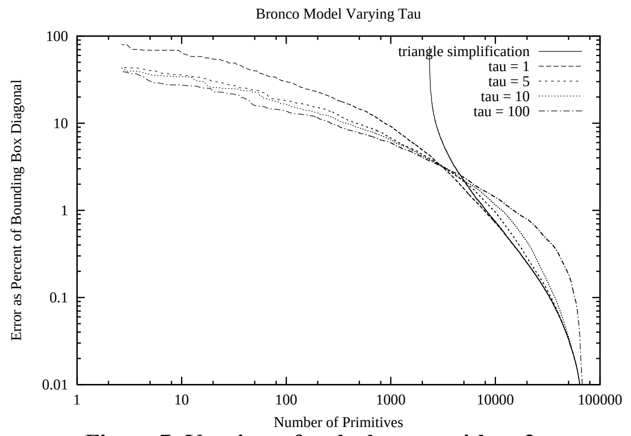


Figure 7: Varying τ for the bronco with κ=3.

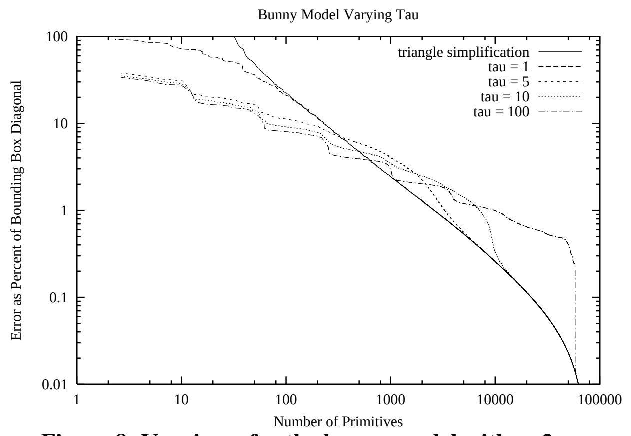


Figure 8: Varying τ for the bunny model with κ=3.

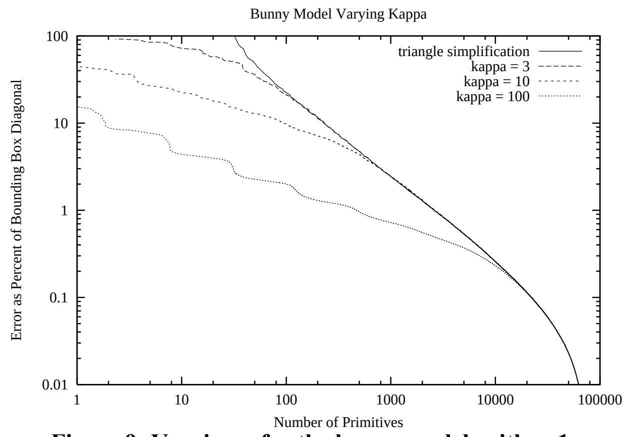


Figure 9: Varying κ for the bunny model, with τ=1.

These graphs show us the nature of error growth in the hierarchies, but they cannot portray the localization of point replacements or the geometric configurations where point replacement is called for. We get a much better intuition for these characteristics from rendered images of the test models.

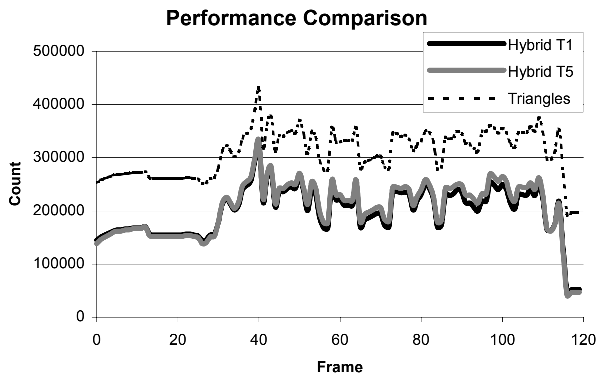


Figure 10: Total primitive counts for a fly-through of the Bronco environment. Hybrid T1/T5 refers to a hybrid simplification using κ=3, τ=1 and κ=3, τ=5, respectively. Triangles refer to a pure triangle simplification.

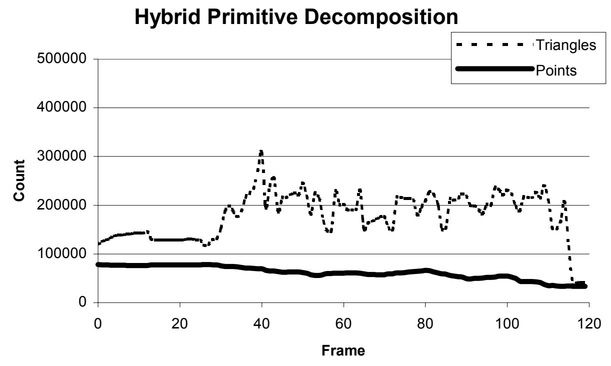


Figure 11: We show the decomposition of the hybrid primitives during the example fly-through of the Bronco model.

Figure 12 shows the bunny being gradually covered by points as it recedes into the distance. The first points appear around the sharp tips of the ears and the curves of the toes, then clusters of points appear in the ridges of the neck and hindquarters, and finally the rest of the triangles are consumed by this phenomenon. The last places to remain triangles are the back and rear of the bunny, which are the flattest. It is pretty clear that the points benefit the most in the regions of high curvature, where simplification is most limited. This occurs primarily when the dihedral angles between faces become small and the mesh is the coarsest. Figure 15 shows the simplification by increasing an object-space error tolerance; thus the rendered points nearer to the eye position are larger, with larger screen-space error.

Figures 13, 14, and 17 show similar transitions for the Bronco, horse, and armadillo models. The Bronco in particular suffers from self-penetration artifacts due to many surfaces with different colors and tight tolerances. We have shown it for a system with κ=10 rather than κ=3 because such a system can reduce these artifacts somewhat by using finer point samples. Triangle simplification alone also suffers from such artifacts on a model like this, but they may be less pronounced than those of the fatter point primitives are.

Figure 16 shows captured frames from the Bronco environment fly-through and also a comparison to a frame with a screen-space error of one pixel rendered using triangle-only simplification.

8 DISCUSSION AND FUTURE WORK

To our knowledge, the hybrid simplification framework presented is the first system that tightly integrates polygon and point rendering into a single multi-resolution hierarchy. This hierarchy is optimized according to the relative performance characteristics of the primitive types on a particular architecture. For a given error bound, it achieves a greater reduction in the overall primitive count as compared to a single-representation hierarchy. As part of this research, we have explored two parameters that influence the characteristics of the hierarchy we build, and we have investigated how the replacement of triangles with points manifests itself in the context of several models.

As future work, we plan to use our hierarchy with a data structure such as an LDI tree. The exact arrangement of the images we warp may differ from other 3D image warping applications because our points are being dynamically added to and removed from inclusion in the warp. One challenge will be to keep our images dense enough with points for warping that we still benefit from the additional efficiency it provides over the transformation of individual points.

Another interesting avenue for exploration is the construction of such a hybrid system in conjunction with a topology-modifying simplification operator. Allowing topological modifications in the polygon domain may "level the playing field" due to their ability to continue simplifying longer. It will be interesting to see how the comparison plays out when both operators can merge objects.

And finally, the system we have presented here only takes into account the geometric deviation of the simplification. It does not account for color error, texture error, nor does it provide for illumination of points. Although measuring the error of attributes is somewhat well understood in their own domains, it is much less clear how to combine them into a useful screen-space metric. Thus attribute errors may be used to guide the off-line optimization, but their use in the interactive display system is a more open-ended problem.

9 ACKNOWLEDGMENTS

We would like to thank Subodh Kumar for help with the MT library and viewer applications and the Bell Labs Multimedia Lab for their support of this research. We also extend our gratitude to the Stanford Computer Graphics Lab for the bunny model, Viewpoint Labs for the Bronco model, Venkat Krishnamurthy, Marc Levoy, and Peter Schröder for the armadillo model, and Cyberware for the horse model.

REFERENCES

Chang, Chun-Fa, Gary Bishop, and Anselmo Lastra. LDI Tree: A Hierarchical Representation for Image-Based Rendering. *Proceedings of SIGGRAPH '99*. pp. 291-298.

Cohen, Jonathan, Dinesh Manocha, and Marc Olano. Simplifying Polygonal Models using Successive Mappings. *Proceedings of IEEE Visualization '97*. pp. 395-402.

Curless, B and M Levoy. A Volumetric Method for Building Complex Models from Range Images. *Proceedings of SIGGRAPH '96*. pp. 303-312.

DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. *Proceedings of IEEE Visualization '97*. pp. 103-110.

DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Efficient Implementation of Multi-Triangulations. *Proceedings of IEEE Visualization '98*. pp. 43-50.

El-Sana, Jihad and Amitabh Varshney. Controlled Simplification of Genus for Polygonal Models. *Proceedings of IEEE Visualization'97*. pp. 403-410.

Erikson, Carl and Dinesh Manocha. GAPS: General and Automatic Polygonal Simplification. *ACM Symposium on Interactive 3D Graphic s '99*. pp. 79-88.

Garland, Michael and Paul Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. *Proceedings of IEEE Visualization '98*. pp. 263-270.

Garland, Michael and Paul Heckbert. Surface Simplification using Quadric Error Bounds. *Proceedings of SIGGRAPH '97*. pp. 209-216.

Grossman, J P and W Dally. Point Sample Rendering. *9th Eurographics Workshop on Rendering '98*, pp. 181-192.

Guéziec, André. Surface Simplification with Variable Tolerance. *Proceedings of Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*. pp. 132-139.

Hoppe, Hugues. Progressive Meshes. *Proceedings of SIGGRAPH '96*. pp. 99-108.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH '97*. pp. 189-198.

Hoppe, H, T DeRose, T Duchamp, J McDonald, and W Stuetzle. Surface Reconstruction from Unorganized Points. *Proceedings of SIGGRAPH '92*. pp. 71-78.

Klein, Reinhard, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. *Proceedings of IEEE Visualization '96*.

Lee, Aaron W. F., Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. *Proceedings of SIGGRAPH '98*. pp. 95-104.

Levoy, Marc and Turner Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022. University of North Carolina at Chapel Hill. 1985.

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH '97*. pp. 199-208.

Max, N and K Ohsaki. Rendering Trees from Precomputed Z-Buffer Views. *Proceedings of Rendering Techniques '95*. pp. 45-54.

McMillan, L and G Bishop. Plenoptic Modeling: An Image-Based Rendering System. *Proceedings of SIGGRAPH '95*. pp. 39-46.

Pfister, H, M Zwicker, J van Baar, and M Gross. Surfels: Surface Elements as Rendering Primitives. *Proceedings of SIGGRAPH 2000*. pp. 335-342.

Popovic, Jovan and Hugues Hoppe. Progressive Simplicial Complexes. *Proceedings of SIGGRAPH '97*. pp. 217-224.

Reeves, W T. Particle Systems: A Technique for Modeling a Class of Fuzzy Objects. *Proceedings of SIGGRAPH '83*. pp. 359-376.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering. *Modeling in Computer Graphics*. Springer-Verlag 1993. pp. 455-465.

Rusinkiewicz, S and M Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH 2000*. pp. 336-352.

Schroeder, W. A Topology-Modifying Progressive Decimation Algorithm. *Proceedings of IEEE Visualization'97*. pp. 205-212.

Xia, Julie C., Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*. vol. 3(2). 1997. pp. 171-183.

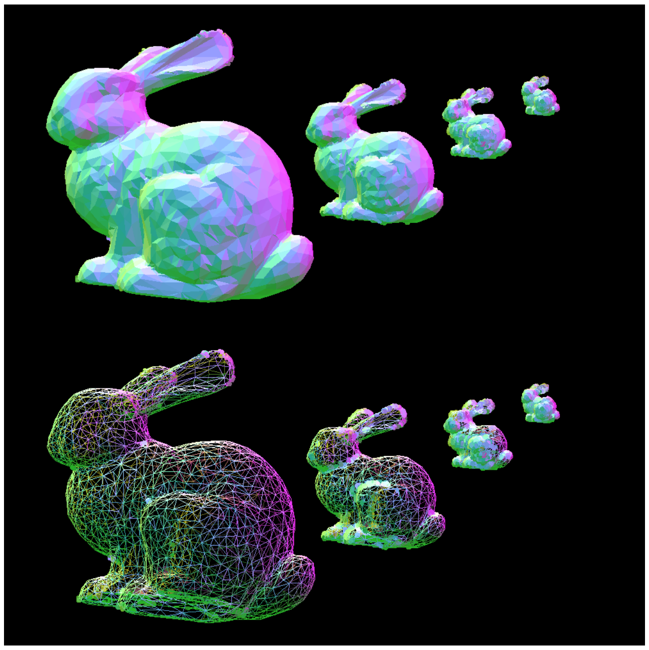


Figure 12: Bunny model with screen-space error of 5 pixels (κ=3, τ=5). (wireframe indicates triangles)

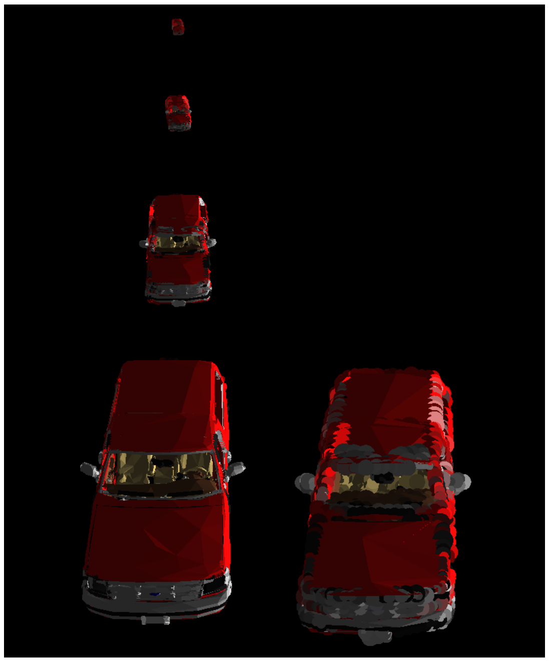


Figure 13: Bronco model with 5 pixels of deviation and zoomed in at 20 pixels of deviation (κ=10, τ=1).

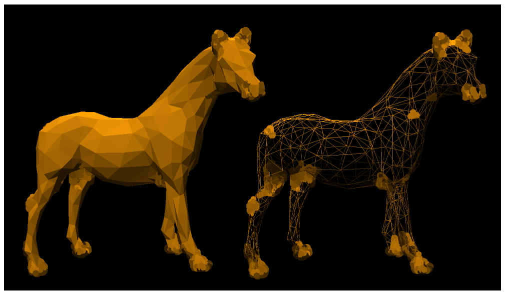


Figure 14: View of the horse model with 20 pixels of deviation (κ=10, τ=1).

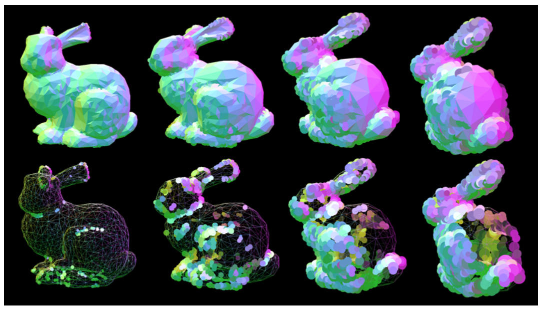
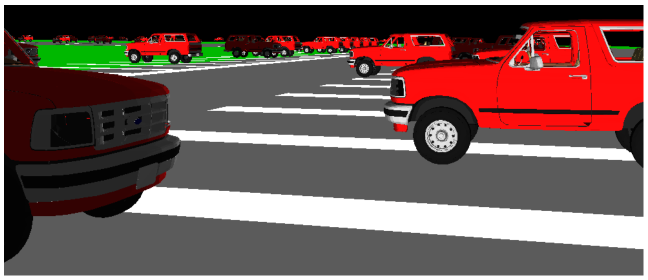
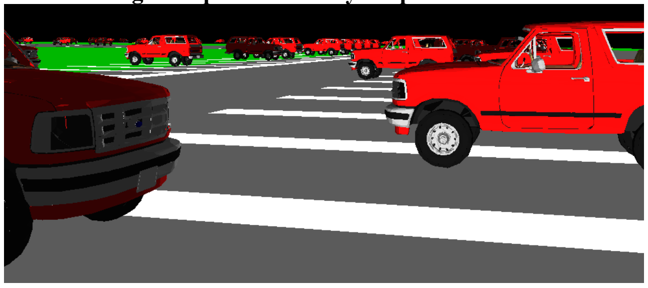


Figure 15: Bunny model with object space deviation of 1%, 2%, 3%, and 4% of its bounding box diagonal (κ=3, τ=5).


Triangle simplification only – 1 pixel of deviation


Triangle simplification only – 3 pixels of deviation

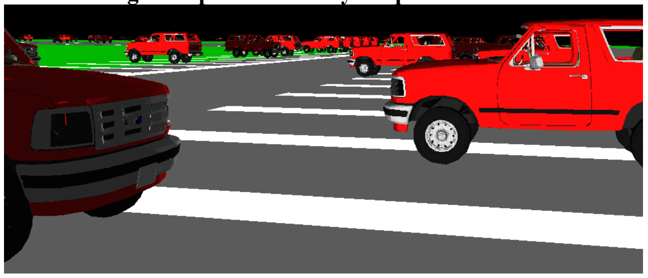

Hybrid simplification – 3 pixels of deviation (κ=3, τ=1)

Figure 16: One frame from the Bronco environment fly-through.

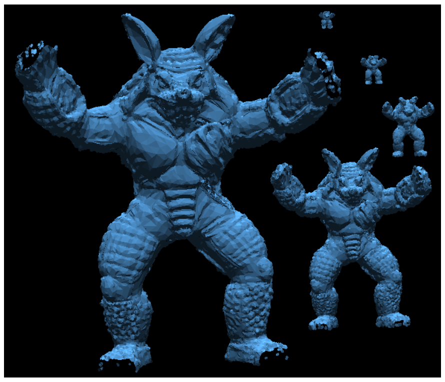


Figure 17: Hybrid simplification of armadillo with 3 pixels of deviation (κ=3, τ=5).

Appearance-Preserving Simplification

Jonathan Cohen Marc Olano Dinesh Manocha

University of North Carolina at Chapel Hill

Abstract

We present a new algorithm for appearance-preserving simplification. Not only does it generate a low-polygon-count approximation of a model, but it also preserves the appearance. This is accomplished for a particular display resolution in the sense that we properly sample the surface position, curvature, and color attributes of the input surface. We convert the input surface to a representation that decouples the sampling of these three attributes, storing the colors and normals in texture and normal maps, respectively. Our simplification algorithm employs a new *texture deviation metric,* which guarantees that these maps shift by no more than a user-specified number of pixels on the screen. The simplification process filters the surface position, while the run-time system filters the colors and normals on a per-pixel basis. We have applied our simplification technique to several large models achieving significant amounts of simplification with little or no loss in rendering quality.

CR Categories: I.3.5: Object hierarchies, I.3.7: Color, shading, shadowing, and texture

Additional Keywords: simplification, attributes, parameterization, color, normal, texture, maps

1 INTRODUCTION

Simplification of polygonal surfaces has been an active area of research in computer graphics. The main goal of simplification is to generate a low-polygon-count approximation that maintains the high fidelity of the original model. This involves preserving the model's main features and overall appearance. Typically, there are three *appearance attributes* that contribute to the overall appearance of a polygonal surface:

1. **Surface position**, represented by the coordinates of the polygon vertices.
2. **Surface curvature**, represented by a field of normal vectors across the polygons.
3. **Surface color**, also represented as a field across the polygons.

The number of samples necessary to represent a surface accurately depends on the nature of the model and its area in screen pixels (which is related to its distance from the viewpoint). For a simplification algorithm to preserve the appearance of the input surface, it must guarantee adequate sampling of these three attributes. If it does, we say that it has preserved the appearance with respect to the display resolution.

e-mail: {cohenj,dm}@cs.unc.edu, olano@engr.sgi.com
WWW: http://www.cs.unc.edu/~geom/APS

The majority of work in the field of simplification has focused on *surface approximation* algorithms. These algorithms bound the error in surface position only. Such bounds can be used to guarantee a maximum deviation of the object's silhouette in units of pixels on the screen. While this guarantees that the object will cover the correct pixels on the screen, it says nothing about the final colors of these pixels.

Of the few simplification algorithms that deal with the remaining two attributes, most provide some threshold on a maximum or average deviation of these attribute values across the model. While such measures do guarantee adequate sampling of all three attributes, they do *not* generally allow increased simplification as the object becomes smaller on the screen. These threshold metrics do not incorporate information about the object's distance from the viewpoint or its area on the screen. As a result of these metrics and of the way we typically represent these appearance attributes, simplification algorithms have been quite restricted in their ability to simplify a surface while preserving its appearance.

1.1 Main Contribution

We present a new algorithm for appearance-preserving simplification. We convert our input surface to a *decoupled representation*. Surface position is represented in the typical way, by a set of triangles with 3D coordinates stored at the vertices. Surface colors and normals are stored in texture and normal maps, respectively. These colors and normals are mapped to the surface with the aid of a surface parameterization, represented as 2D texture coordinates at the triangle vertices.

The surface position is filtered using a standard surface approximation algorithm that makes local, complexity-reducing simplification operations (e.g. edge collapse, vertex removal, etc.). The color and normal attributes are filtered by the run-time system at the pixel level, using standard mip-mapping techniques [1].

Because the colors and normals are now decoupled from the surface position, we employ a new *texture deviation metric,* which effectively bounds the deviation of a mapped attribute value's position from its correct position on the original surface. We thus guarantee that each attribute is appropriately sampled and mapped to screen-space. The deviation metric necessarily constrains the simplification algorithm somewhat, but it is much less restrictive than retaining sufficient tessellation to accurately represent colors and normals in a standard, per-vertex representation. The preservation of colors using texture maps is possible on all current graphics systems that supports real-time texture maps. The preservation of normals using normal maps is possible on prototype machines today, and there are indications that hardware

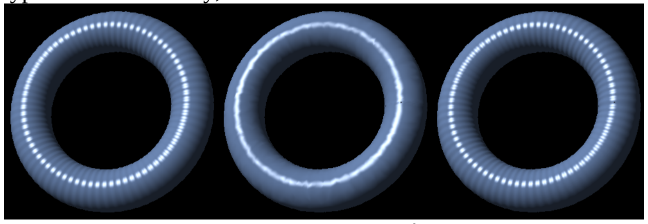


Figure 1: Bumpy Torus Model. *Left*: 44,252 triangles full resolution mesh. *Middle and Right*: 5,531 triangles, 0.25 mm maximum image deviation. *Middle*: per-vertex normals. *Right*: normal maps

support for real-time normal maps will become more widespread in the next several years.

One of the nice properties of this approach is that the user-specified error tolerance, ε, is both simple and intuitive; it is a screen-space deviation in pixel units. A particular point on the surface, with some color and some normal, may appear to shift by at most ε pixels on the screen.

We have applied our algorithm to several large models. Figure 1 clearly shows the improved quality of our appearance-preserving simplification technique over a standard surface approximation algorithm with per-vertex normals. By merely controlling the switching distances properly, we can discretely switch between a few statically-generated levels of detail (sampled from a progressive mesh representation) with no perceptible artifacts. Overall, we are able to achieve a significant speedup in rendering large models with little or no loss in rendering quality.

1.2 Paper Organization

In Section 2, we review the related work from several areas. Section 3 presents an overview of our appearance-preserving simplification algorithm. Sections 4 through 6 describe the components of this algorithm, followed by a discussion of our particular implementation and results in Section 7. Finally, we mention our ongoing work and conclude in Section 8.

2 RELATED WORK

Research areas related to this paper include geometric levels-of-detail, preservation of appearance attributes, and map-based representations. We now briefly survey these.

2.1 Geometric Levels-Of-Detail

Given a polygonal model, a number of algorithms have been proposed for generating levels-of-detail. These methods differ according to the local or global error metrics used for simplification and the underlying data structures or representations. Some approaches based on vertex clustering [2, 3] are applicable to all polygonal models and do not preserve the topology of the original models. Other algorithms assume that the input model is a valid mesh. Algorithms based on vertex removal [4, 5] and local error metrics have been proposed by [6-10]. Cohen et al. [11] and Eck et al. [12] have presented algorithms that preserve topology and use a global error bound. Our appearance-preserving simplification algorithm can be combined with many of these.

Other simplification algorithms include decimation techniques based on vertex removal [4], topology modification [13], and controlled simplification of genus [14]. All of these algorithms compute static levels-of-detail. Hoppe [15] has introduced an incremental representation, called the *progressive mesh*, and based on that representation view-dependent algorithms have been proposed by [16, 17]. These algorithms use different view-dependent criteria like local illumination, screen-space surface approximation error, and silhouette edges to adaptively refine the meshes. Our appearance preserving simplification algorithm generates a progressive mesh, which can be used by these view-dependent algorithms.

2.2 Preserving Appearance Attributes

Bajaj and Schikore [18] have presented an algorithm to simplify meshes with associated scalar fields to within a given tolerance. Hughes et al. [19] have presented an algorithm to simplify radiositized meshes. Erikson and Manocha[20] grow error volumes for appearance attributes as well as geometry. Many algorithms based on multi-resolution analysis have been proposed as well. Schroeder and Sweldens [21] have presented algorithms for simplifying functions defined over a sphere. Eck et al. [12] apply multi-resolution analysis to simplify arbitrary meshes, and Certain et al. [22] extend this to colored meshes by separately analyzing surface geometry and color. They make use of texture mapping hardware to render the color at full resolution. It may be possible to extend this approach to handle other functions on the mesh. However, algorithms based on vertex removal and edge collapses [11, 15] have been able to obtain more drastic simplification (in terms of reducing the polygon count) and produce better looking simplifications [15].

Hoppe [15] has used an optimization framework to preserve discrete and scalar surface appearance attributes. Currently, this algorithm measures a maximum or average deviation of the scalar attributes across the model. Our approach can be incorporated into this comprehensive optimization framework to preserve the appearance of colors and normals, while allowing continued simplification as an object's screen size is reduced.

2.3 Map-based Representations

Texture mapping is a common technique for defining color on a surface. It is just one instance of mapping, a general technique for defining attributes on a surface. Other forms of mapping use the same texture coordinate parameterization, but with maps that contain something other than surface color. *Displacement maps* [23] contain perturbations of the surface position. They are typically used to add surface detail to a simple model. *Bump maps* [24] are similar, but instead give perturbations of the surface normal. They can make a smooth surface appear bumpy, but will not change the surface's silhouette. *Normal maps* [25] can also make a smooth surface appear bumpy, but contain the actual normal instead of just a perturbation of the normal.

Texture mapping is available in most current graphics systems, including workstations and PCs. We expect to see bump mapping and similar surface shading techniques on graphics systems in the near future [26]. In fact, many of these mapping techniques are already possible using the procedural shading capabilities of PixelFlow[27].

Several researchers have explored the possibility of replacing geometric information with texture. Kajiya first introduced the "hierarchy of scale" of geometric models, mapping, and lighting[28]. Cabral et. al. [29] addressed the transition between bump mapping and lighting effects. Westin et. al. [30] generated BRDFs from a Monte-Carlo ray tracing of an idealized piece of surface. Becker and Max [31] handle transitions from geometric detail in the form of displacement maps to shading in the form of bump maps. Fournier [25] generates maps with normal and shading information directly from surface geometry. Krishnamurthy and Levoy [32] fit complex, scanned surfaces with a set of smooth B-spline patches, then store some of the lost geometric information in a displacement map or bump map. Many algorithms first capture the geometric complexity of a scene in an image-based representation by rendering several different views and then render the scene using texture maps [33-36].

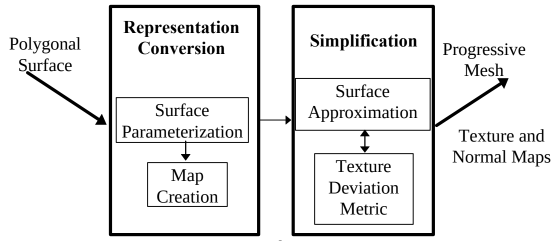


Figure 2: Components of an appearance-preserving simplification system.

3 OVERVIEW

We now present an overview of our appearance-preserving simplification algorithm. Figure 2 presents a breakdown of the algorithm into its components. The input to the algorithm is the polygonal surface, $\mathbb{M}_0$, to be simplified. The surface may come from one of a wide variety of sources, and thus may have a variety of characteristics. The types of possible input models include:

- **CAD models**, with per-vertex normals and a single color
- **Radiositized models**, with per-vertex colors and no normals
- **Scientific visualization models**, with per-vertex normals and per-vertex colors
- **Textured models**, with texture-mapped colors, with or without per-vertex normals

To store the colors and normals in maps, we need a parameterization of the surface, $\mathbf{F}_0(\mathbf{X})$: $\mathbb{M}_0 \rightarrow \mathbb{P}$, where $\mathbb{P}$ is a 2D texture domain (*texture plane*), as shown in Figure 3. If the input model is already textured, such a parameterization comes with the model. Otherwise, we create one and store it in the form of per-vertex texture coordinates. Using this parameterization, per-vertex colors and normals are then stored in texture and normal maps.

The original surface and its texture coordinates are then fed to the surface simplification algorithm. This algorithm is responsible for choosing which simplification operations to perform and in what order. It calls our texture deviation component to measure the deviation of the texture coordinates caused by each proposed operation. It uses the resulting error bound to help make its choices of operations, and stores the bound with each operation in its progressive mesh output.

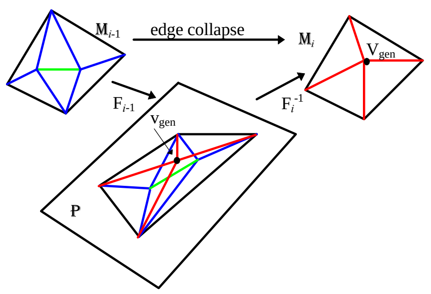
We can use the resulting progressive mesh with error bounds to create a static set of levels of detail with error bounds, or we can use the progressive mesh directly with a view-dependent simplification system at run-time. Either way, the error bound allows the run-time system to choose or adjust the tessellation of the models to meet a user-specified tolerance. It is also possible for the user to choose a desired polygon count and have the run-time system increase or decrease the error bound to meet that target.

4 REPRESENTATION CONVERSION

Before we apply the actual simplification component of our algorithm, we perform a representation conversion (as shown in Figure 2). The representation we choose for our surface has a significant impact on the amount of simplification we can perform for a given level of visual fidelity. To convert to a form which decouples the sampling rates of the colors and normals from the sampling rate of the surface, we first parameterize the surface, then store the color and normal information in separate maps.

4.1 Surface Parameterization

To store a surface's color or normal attributes in a map, the surface must first have a 2D parameterization. This function, $\mathbf{F}_0(\mathbf{X})$: $\mathbb{M}_0 \rightarrow \mathbb{P}$, maps points, $\mathbf{X}$, on the input surface, $\mathbb{M}_0$, to points, $\mathbf{x}$,[*] on the texture plane, $\mathbb{P}$ (see Figure 3). The surface is typically decomposed into several *polygonal patches*, each with its own parameterization. The creation of such parameterizations has been an active area of research and is fundamental for shape transformation, multi-resolution analysis, approximation of meshes by NURBS, and texture mapping. Though we do not present a new algorithm for such parameterization here, it is useful to consider



$\mathbb{M}_{i\text{-}1}$
edge collapse
$\mathbb{M}_i$
$\mathrm{V}_{\mathrm{gen}}$
$\mathrm{F}_{i\text{-}1}$
F_i^{-1}
$\mathrm{v}_{\mathrm{gen}}$
$\mathbb{P}$

Figure 3: A look at the *i*th edge collapse. Computing $\mathbf{V}_{\text{gen}}$ determines the shape of the new mesh, $\mathbb{M}_i$. Computing $\mathbf{v}_{\text{gen}}$ determines the new mapping $\mathbf{F}_i$, to the texture plane, $\mathbb{P}$.

the desirable properties of such a parameterization for our algorithm. They are:

1. **Number of patches**: The parameterization should use as few patches as possible. The triangles of the simplified surface must each lie in a single patch, so the number of patches places a bound on the minimum mesh complexity.
2. **Vertex distribution**: The vertices should be as evenly distributed in the texture plane as possible. If the parameterization causes too much area compression, we will require a greater map resolution to capture all of our original per-vertex data.
3. **One-to-one mapping**: The mapping from the surface to the texture plane should be one-to-one. If the surface has folds in the texture plane, parts of the texture will be incorrectly stored and mapped back to the surface

Our particular application of the parameterization makes us somewhat less concerned with preserving aspect ratios than some other applications are. For instance, many applications apply $\mathbf{F}^{-1}(\mathbf{x})$ to map a pre-synthesized texture map to an arbitrary surface. In that case, distortions in the parameterization cause the texture to look distorted when applied to the surface. However, in our application, the color or normal data originates on the surface itself. Any distortion created by applying $\mathbf{F}(\mathbf{X})$ to map this data onto $\mathbb{P}$ is reversed when we apply $\mathbf{F}^{-1}(\mathbf{x})$ to map it back to $\mathbb{M}$.

Algorithms for computing such parameterizations have been studied in the computer graphics and graph drawing literature.

Computer Graphics: In the recent computer graphics literature, [12, 37, 38] use a spring system with various energy terms to distribute the vertices of a polygonal patch in the plane. [12, 32, 38, 39] provide methods for subdividing surfaces into separate patches based on automatic criteria or user-guidance. This body of research addresses the above properties one and two, but unfortunately, parameterizations based on spring-system algorithms do not generally guarantee a one-to-one mapping.

Graph Drawing: The field of graph drawing addresses the issue of one-to-one mappings more rigorously. Relevant topics include straight-line drawings on a grid [40] and convex straight-line drawings [41]. Battista et al. [42] present a survey of the field. These techniques produce guaranteed one-to-one mappings, but the necessary grids for a graph with V vertices are worst case (and typically) O(V) width and height, and the vertices are generally unevenly spaced.

To break a surface into polygonal patches, we currently apply an automatic subdivision algorithm like that presented in [12]. Their application requires a patch network with more constraints than ours. We can generally subdivide the surface into fewer patches. During this process, which grows Voronoi-like patches, we simply require that each patch not expand far enough to touch itself. To produce the parameterization for each patch, we employ

[*] Capital letters (e.g. **X**) refer to points in 3D, while lowercase letters (e.g. **x**) refer to points in 2D.

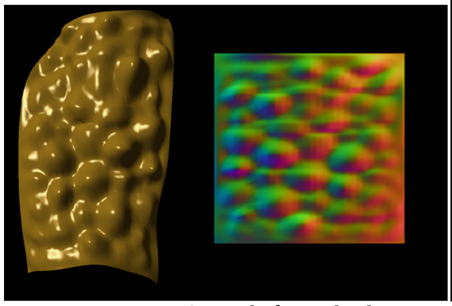


Figure 4: A patch from the leg of an armadillo model and its associated normal map.

Figure 5: Lion model.

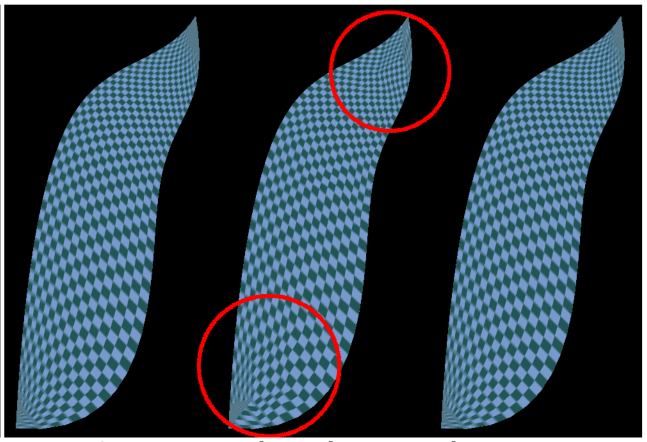


Figure 6: Texture coordinate deviation and correction on the lion's tail. *Left*: 1,740 triangles full resolution. *Middle and Right*: 0.25 mm maximum image deviation. *Middle*: 108 triangles, no texture deviation metric. *Right*: 434 triangles with texture metric.

a spring system with uniform weights. A side-by-side comparison of various choices of weights in [12] shows that uniform weights produce more evenly-distributed vertices than some other choices. For parameterizations used only with one particular map, it is also possible to allow more area compression where data values are similar. While this technique will generally create reasonable parameterizations, it would be better if there were a way to *also* guarantee that $\mathbf{F}(\mathbf{X})$ is one-to-one, as in the graph drawing literature.

4.2 Creating Texture and Normal Maps

Given a polygonal surface patch, $\mathbb{M}_0$, and its 2D parameterization, $\mathbf{F}$, it is straightforward to store per-vertex colors and normals into the appropriate maps using standard rendering software. To create a map, scan convert each triangle of $\mathbb{M}_0$, replacing each of its vertex coordinates, $\mathbf{V}_j$, with $\mathbf{F}(\mathbf{V}_j)$, the texture coordinates of the vertex. For a texture map, apply the Gouraud method for linearly interpolating the colors across the triangles. For a normal map, interpolate the per-vertex normals across the triangles instead (Figure 4).

The most important question in creating these maps is what the maximum resolution of the map images should be. To capture all the information from the original mesh, each vertex's data should be stored in a unique texel. We can guarantee this conservatively by choosing $1/d$ x $1/d$ for our map resolution, where d is the minimum distance between vertex texture coordinates:

$$d = \min_{\mathbf{V}_i,\mathbf{V}_j \in \mathbb{M}_0, i \neq j} \left\| \mathbf{F}(\mathbf{V}_i) - \mathbf{F}(\mathbf{V}_j) \right\| \qquad (1)$$

If the vertices of the polygonal surface patch happen to be a uniform sampling of the texture space (e.g. if the polygonal surface patch was generated from a parametric curved surface patch), then the issues of scan conversion and resolution are simplified considerably. Each vertex color (or normal) is simply stored in an element of a 2D array of the appropriate dimensions, and the array itself is the map image.

It is possible to trade off accuracy of the map data for run-time texturing resources by scaling down the initial maps to a lower resolution.

5 SIMPLIFICATION ALGORITHM

Once we have decomposed the surface into one or more parameterized polygonal patches with associated maps, we begin the actual simplification process. Many simplification algorithms perform a series of edge collapses or other local simplification operations to gradually reduce the complexity of the input surface. The order in which these operations are performed has a large impact on the quality of the resulting surface, so simplification algorithms typically choose the operations in order of increasing error according to some metric. This metric may be local or global in nature, and for surface approximation algorithms, it provides some bound or estimate on the error in surface position. The operations to be performed are typically maintained in a priority queue, which is continually updated as the simplification progresses. This basic design is applied by many of the current simplification algorithms, including [6-8, 15].

To incorporate our appearance-preservation approach into such an algorithm, the original algorithm is modified to use our texture deviation metric in addition to its usual error metric. When an edge is collapsed, the error metric of the particular surface approximation algorithm is used to compute a value for $\mathbf{V}_{gen}$, the surface position of the new vertex (see Figure 3). Our texture deviation metric is then applied to compute a value for $\mathbf{v}_{gen}$, the texture coordinates of the new vertex.

For the purpose of computing an edge's priority, there are several ways to combine the error metrics of surface approximation along with the texture deviation metric, and the appropriate choice depends on the algorithm in question. Several possibilities for such a *total error metric* include a weighted combination of the two error metrics, the maximum or minimum of the error metrics, or one of the two error metrics taken alone. For instance, when integrating with Garland and Heckbert's algorithm [6], it would be desirable to take a weighted combination in order to retain the precedence their system accords the topology-preserving collapses over the topology-modifying collapses. Similarly, a weighted combination may be desirable for an integration with Hoppe's system [15], which already optimizes error terms corresponding to various mesh attributes.

The interactive display system later uses the error metrics to determine appropriate distances from the viewpoint either for switching between static levels of detail or for collapsing/splitting the edges dynamically to produce adaptive, view-dependent tessellations. If the system intends to guarantee that certain tolerances are met, the maximum of the error metrics is often an appropriate choice.

6 TEXTURE DEVIATION METRIC

A key element of our approach to appearance-preservation is the measurement of the *texture coordinate deviation* caused by the simplification process. We provide a bound on this deviation, to

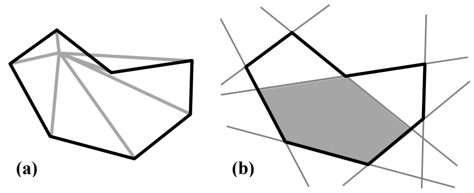


Figure 7: (a) An invalid choice for $\mathbf{v}_{gen}$ in $\mathbb{P}$, causing the new triangles extend outside the polygon. (b) Valid choices must lie in the shaded *kernel*.

be used by the simplification algorithm to prioritize the potential edge collapses and by the run-time visualization system to choose appropriate levels of detail based on the current viewpoint. The lion's tail in Figure 6 demonstrates the need to measure texture coordinate deviation. The center figure is simplified by a surface approximation algorithm without using a texture deviation metric. The distortions are visible in the areas marked by red circles. The right tail is simplified using our texture deviation metric and does not have visible distortions. The image-space deviation bound now applies to the texture as well as to the surface.

For a given point, $\mathbf{X}$, on simplified mesh $\mathbb{M}_i$, this deviation is the distance in 3D from $\mathbf{X}$ to the point on the input surface with the same texture coordinates:

$$\mathrm{T}_i(\mathbf{X}) = \left\| \mathbf{X} - \mathbf{F}_0^{-1}(\mathbf{F}_i(\mathbf{X})) \right\| \qquad (2)$$

We define the texture coordinate deviation of a whole triangle to be the maximum deviation of all the points in the triangle, and similarly for the whole surface:

$$\mathrm{T}_i(\Delta) = \max_{\mathbf{X}\in\Delta} \mathrm{T}_i(\mathbf{X}); \quad \mathrm{T}_i(\mathbb{M}_i) = \max_{\mathbf{X}\in\mathbf{M}_i} \mathrm{T}_i(\mathbf{X}) \qquad (3)$$

To compute the texture coordinate deviation incurred by an edge collapse operation, our algorithm takes as input the set of triangles before the edge collapse and $\mathbf{V}_{gen,}$ the 3D coordinates of the new vertex generated by the collapse operation. The algorithm outputs $\mathbf{v}_{gen}$, the 2D texture coordinates for this generated vertex, and a bound on $\mathrm{T}_i(\Delta)$ for each of the triangles after the collapse.

6.1 Computing New Texture Coordinates

We visualize the neighborhood of an edge to be collapsed in the texture plane, $\mathbb{P}$, as shown in Figure 3. The boundary of the edge neighborhood is a polygon in $\mathbb{P}$. The edge collapse causes us to replace the two vertices of the edge with a single vertex. The 3D coordinates, $\mathbf{V}_{gen}$ of this generated vertex are provided to us by the surface approximation algorithm. The first task of the texture deviation algorithm is to compute $\mathbf{v}_{gen}$, the 2D texture coordinates of this generated vertex.

For $\mathbf{v}_{gen}$ to be valid, it must lie in the convex *kernel* of our polygon in the texture plane [43] (see Figure 7). Meeting this criterion ensures that the set of triangles after the edge collapse covers exactly the same portion of the texture plane as the set of triangles before the collapse.

Given a candidate point in the texture plane, we efficiently test the kernel criterion with a series of dot products to see if it lies on the inward side of each polygon edge. We first test some heuristic choices for the texture coordinates – the midpoint of the original edge in the texture plane or one of the edge vertices. If the heuristic choices fail we compute a point inside the kernel by averaging three corners, found using linear programming techniques [43].

6.2 Patch Borders & Continuity

Unlike an interior edge collapse, an edge collapse on a patch border can change the coverage in the texture plane, either by cutting off some of texture space or by extending into a portion of texture space for which we have no map data. Since neither of these is acceptable, we add additional constraints on the choice of $\mathbf{v}_{gen}$ at patch borders.

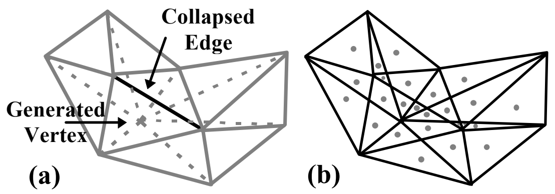


Figure 8: (a) An overlay in $\mathbb{P}$ determines the mapping between $\mathbb{M}_{i-1}$ and $\mathbb{M}_i$. (b) A set of polygonal *mapping cells*, each containing a dot.

We assume that the area of texture space for which we have map data is rectangular (though the method works for any map that covers a polygonal area in texture space), and that the edges of the patch are also the edges of the map. If the entire edge to be collapsed lies on a border of the map, we restrict $\mathbf{v}_{gen}$ to lie on the edge. If one of the vertices of the edge lies on a corner of the map, we further restrict $\mathbf{v}_{gen}$ to lie at that vertex. If only one vertex is on the border, we restrict $\mathbf{v}_{gen}$ to lie at that vertex. If one vertex of the edge lies on one border of the map and the other vertex lies on a different border, we do not allow the edge collapse.

The surface parameterization component typically breaks the input model into several connected patches. To preserve geometric and texture continuity across the boundary between them, we add further restrictions on the simplifications that are performed along the border. The shared border edges must be simplified on both patches, with matching choices of $\mathbf{V}_{\mathbf{gen}}$ and $\mathbf{v}_{\mathbf{gen}}$.

6.3 Measuring Texture Deviation

Texture deviation is a measure of the parametric distortion caused by the simplification process. We measure this deviation using a method similar to the one presented to measure surface deviation in [8]. The main difference is that we now measure the deviation using our mapping in the texture plane, rather than in the plane of some planar projection. While [8] presents an overview of this technique, we present it more formally.

Given the overlay (see Figure 8(a)) in the texture plane, $\mathbb{P}$, of two simplified versions of the surface, $\mathbb{M}_i$ and $\mathbb{M}_j$, we define the *incremental texture deviation* between them:

$$\mathbf{E}_{i,j}(\mathbf{x}) = \left\| \mathbf{F}_i^{-1}(\mathbf{x}) - \mathbf{F}_j^{-1}(\mathbf{x}) \right\| \qquad (4)$$

This is the deviation between corresponding 3D points on the surfaces, both with texture coordinates, $\mathbf{x}$. Between any two sequential surfaces, $\mathbb{M}_i$ and $\mathbb{M}_{i-1}$, differing only by an edge collapse, the incremental deviation, $\mathbf{E}_{i,i-1}(\mathbf{x})$, is only non-zero in the neighborhood of the collapsed edge (i.e. only in the triangles that actually move).

The edges on the overlay in $\mathbb{P}$ partition the region into a set of convex, polygonal *mapping cells* (each identified by a dot in Figure 8(b)). Within each mapping cell, the incremental deviation function is linear, so the maximum incremental deviation for each cell occurs at one of its boundary points. Thus, we bound the incremental deviation using only the deviation at the cell vertices, $\mathbf{v}_k$:

$$\mathbf{E}_{i,i-1}(\mathbb{P}) = \max_{\mathbf{x}\in\mathbf{P}} \mathbf{E}_{i,i-1}(\mathbf{x}) = \max_{\mathbf{v}_k} \mathbf{E}_{i,i-1}(\mathbf{v}_k) \qquad (5)$$

In terms of the incremental deviation, the *total texture deviation*, defined in (2) (the distance from points on $\mathbb{M}_i$ to corresponding points on the original surface, $\mathbb{M}_0$) is

$$\mathbf{T}_i(\mathbf{X}) = \mathbf{E}_{i,0}\left(\mathbf{F}_i(\mathbf{X})\right) \qquad (6)$$

We approximate $\mathbf{E}_{i,0}(\mathbf{x})$ using a set of axis-aligned boxes. This provides a convenient representation of a bound on $\mathbf{T}_i(\mathbf{X})$, which we can update from one simplified mesh to the next without having to refer to the original mesh. Each triangle, Δ_k in $\mathbf{M}_i$, has its own axis-aligned box, $\boldsymbol{b}_{i,k}$ such that at every point on the triangle, the Minkowski sum of the 3D point with the box gives a region that contains the point on the original surface with the same texture coordinates.

$$\forall \mathbf{X} \in \Delta_k,\ \mathbf{F}_0^{-1}\left(\mathbf{F}_i(\mathbf{X})\right) \in \mathbf{X} \oplus \boldsymbol{b}_{i,k} \qquad (7)$$

Figure 9(a) shows an original surface (curve) in black and a simplification of it, consisting of the thick blue and green lines. The box associated with the blue line, $\boldsymbol{b}_{i,0}$, is shown in blue, while the box for the green line, $\boldsymbol{b}_{i,1}$, is shown in green. The blue box slides along the blue line; at every point of application, the point on the base mesh with the same texture coordinate is contained within the translated box. For example, one set of corresponding points is shown in red, with its box also in red.

From (2) and (7), we produce $\tilde{\mathrm{T}}_i(\mathbf{X})$, a bound on the total texture deviation, $\mathrm{T}_i(\mathbf{X})$. This our texture deviation output.

$$\mathrm{T}_i(\mathbf{X}) \ \le\ \tilde{\mathrm{T}}_i(\mathbf{X}) = \max_{\mathbf{X}' \in \mathbf{X} \oplus \boldsymbol{b}_{i,j}} \left\| \mathbf{X} - \mathbf{X}' \right\| \qquad (8)$$

$\tilde{\mathrm{T}}_i(\mathbf{X})$ is the distance from $\mathbf{X}$ to the farthest corner of the box at $\mathbf{X}$. This will always bound the distance from $\mathbf{X}$ to $\mathbf{F}_0^{-1}(\mathbf{F}_i(\mathbf{X}))$. The maximum deviation over an edge collapse neighborhood is the maximum $\tilde{\mathrm{T}}_i(\mathbf{X})$ for any cell vertex.

The boxes, $\boldsymbol{b}_{i,k}$, are the only information we keep about the position of the original mesh as we simplify. We create a new set of boxes, $\boldsymbol{b}_{i+1,k}$, for mesh $\mathbf{M}_{i+1}$ using an incremental computation (described in Figure 10). Figure 9(b) shows the propagation from $\mathbf{M}_i$ to $\mathbf{M}_{i+1}$. The blue and green lines are simplified to the pink line. The new box, $\boldsymbol{b}_{i+1,0}$ is constant as it slides across the pink line. The size and offset is chosen so that, at every point of application, the pink box, $\boldsymbol{b}_{i+1,0}$, contains the corresponding blue or green boxes, $\boldsymbol{b}_{i,0}$ or $\boldsymbol{b}_{i,1}$.

If $\mathbf{X}$ is a point on $\mathbf{M}_i$ in triangle k, and $\mathbf{Y}$ is the point with the same texture coordinate on $\mathbf{M}_{i+1}$, the containment property of (7) holds:

$$\mathbf{F}_0^{-1}\left(\mathbf{F}_{i+1}(\mathbf{Y})\right) \in \mathbf{X} \oplus \boldsymbol{b}_{i,k} \subseteq \mathbf{Y} \oplus \boldsymbol{b}_{i+1,k'} \qquad (9)$$

For example, all three red dots Figure 9(b) have the same texture coordinates. The red point on $\mathbf{M}_0$ is contained in the smaller red box, $\mathbf{X} \oplus \boldsymbol{b}_{i,0}$, which is contained in the larger red box, $\mathbf{Y} \oplus \boldsymbol{b}_{i+1,0}$.

Because each mapping cell in the overlay between $\mathbf{M}_i$ and $\mathbf{M}_{i+1}$ is linear, we compute the sizes of the boxes, $\boldsymbol{b}_{i+1,k'}$, by considering

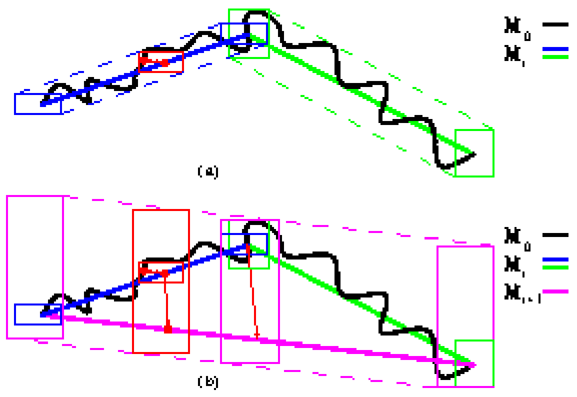


Figure 9: 2D illustration of the box approximation to total deviation error. a) A curve has been simplified to two segment, each with an associated box to bound the deviation. b) As we simplify one more step, the approximation is propagated to the newly created segment.

```
PropagateError():
foreach cell vertex, v
    foreach triangle, $T_{old}$, in $M_{i-1}$ touching v
        foreach triangle, $T_{new}$, in $M_i$ touching v
            PropagateBox(v, $T_{old}$, $T_{new}$)
PropagateBox(v, $T_{old}$, $T_{new}$):
$P_{old}$ = $F_{i-1}^{-1}$(v), $P_{new}$ = $F_i^{-1}$(v)
Enlarge $T_{old}$.box so that $T_{old}$.box applied at
    $P_{old}$ contains $T_{new}$.box applied at $P_{new}$
```

Figure 10: Pseudo-code for the propagation of deviation error from mesh M_{i-1} to mesh M_i.

only the box correspondences at cell vertices. In Figure 9(b), there are three places we must consider. If the magenta box contains the blue and green boxes in all three places, it will contain them everywhere.

Together, the propagation rules, which are simple to implement, and the box-based approximation to the texture deviation, provide the tools we need to efficiently provide a texture deviation for the simplification process.

7 IMPLEMENTATION AND RESULTS

In this section we present some details of our implementation of the various components of our appearance-preserving simplification algorithm. These include methods for representation conversion, simplification and, finally, interactive display.

7.1 Representation Conversion

We have applied our technique to several large models, including those listed in Table 1. The bumpy torus model (Figure 1) was created from a parametric equation to demonstrate the need for greater sampling of the normals than of the surface position. The lion model (Figure 5) was designed from NURBS patches as part of a much larger garden environment, and we chose to decorate it with a marble texture (and a checkerboard texture to make texture deviation more apparent in static images). Neither of these models required the computation of a parameterization. The armadillo (Figure 12) was constructed by merging several laser-scanned meshes into a single, dense polygon mesh. It was decomposed into polygonal patches and parameterized using the algorithm presented in [32], which eventually converts the patches into a NURBS representation with associated displacement maps.

Because all these models were not only parameterized, but available in piecewise-rational parametric representations, we generated polygonal patches by uniformly sampling these representations in the parameter space. We chose the original tessellation of the models to be high enough to capture all the detail available in their smooth representations. Due to the uniform sampling, we were able to use the simpler method of map creation (described in Section 4.2), avoiding the need for a scan-conversion process.

7.2 Simplification

We integrated our texture deviation metric with the successive mapping algorithm for surface approximation [8]. The error metric for the successive mapping algorithm is simply a 3D surface deviation. We used this deviation only in the computation of $\mathbf{V}_{gen}$. Our total error metric for prioritizing edges and choosing switching distances is just the texture deviation. This is sensible because the texture deviation metric is also a measure of surface deviation, whose particular mapping is the parameterization. Thus, if the successive mapping metric is less than the texture deviation metric, we must apply the texture deviation metric, because it is the minimum bound we know that guarantees the bound on our texture deviation. On the other hand, if the successive mapping metric is greater than the texture deviation metric, the texture deviation bound is still sufficient to guarantee a bound on both the surface deviation and the texture.

To achieve a simple and efficient run-time system, we apply a post-process to convert the progressive mesh output to a static set of levels of detail, reducing the mesh complexity by a factor of two at each level.

Our implementation can either treat each patch as an independent object or treat a connected set of patches as one object. If we simplify the patches independently, we have the freedom to switch their levels of detail independently, but we will see cracks between the patches when they are rendered at a sufficiently large error tolerance. Simplifying the patches together allows us to prevent cracks by switching the levels of detail simultaneously.

Table 1 gives the computation time to simplify several models,

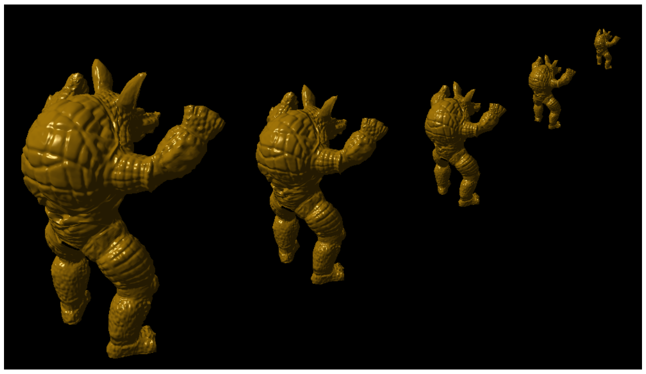


Figure 11: Levels of detail of the armadillo model shown with 1.0 mm maximum image deviation. Triangle counts are: 7,809, 3,905, 1,951, 975, 488

Model	Patches	Input Tris	Time	Map Res.
Torus	1	44,252	4.4	512x128
Lion	49	86,844	7.4	N.A.
Armadillo	102	2,040,000	190	128x128

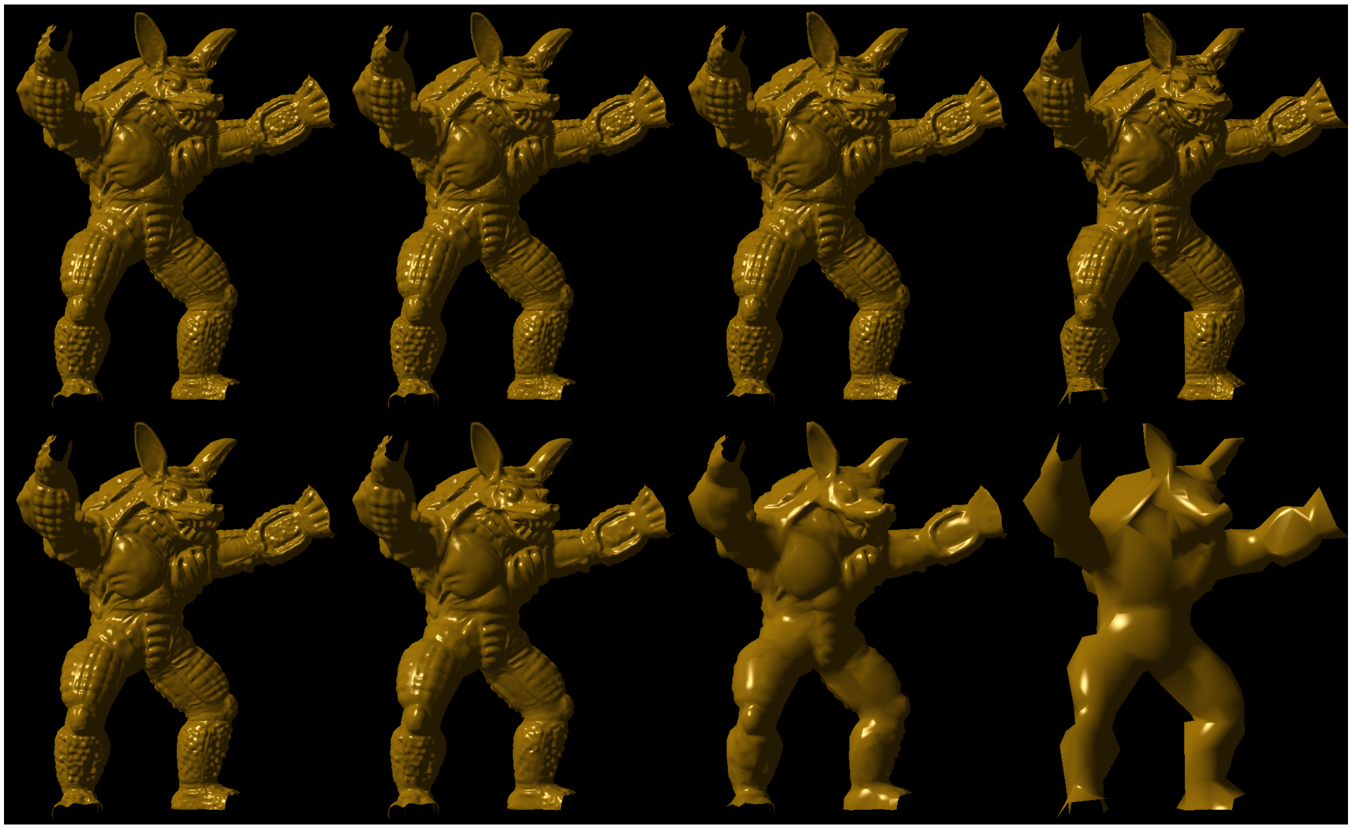
Table 1: Several models used to test appearance-preserving simplification. Simplification time is in minutes on a MIPS R10000 processor.

as well as the resolution of each map image. Figure 11 and Figure 12 show results on the armadillo model. It should be noted that the latter figure is not intended to imply equal computational costs for rendering models with per-vertex normals and normal maps. Simplification using the normal map representation provides measurable quality and reduced triangle overhead, with an additional overhead dependent on the screen resolution.

7.3 Interactive Display System

We have implemented two interactive display systems: one on top of SGI's IRIS Performer library, and one on top of a custom library running on a PixelFlow system. The SGI system supports color preservation using texture maps, and the PixelFlow system supports color and normal preservation using texture and normal maps, respectively. Both systems apply a bound on the distance from the viewpoint to the object to convert the texture deviation error in 3D to a number of pixels on the screen, and allow the user to specify a tolerance for the number of pixels of deviation. The tolerance is ultimately used to choose the primitives to render from among the statically generated set of levels of detail.

Our custom shading function on the PixelFlow implementation performs a mip-mapped look-up of the normal and applies a


249,924 triangles
0.05 mm max image deviation

62,480 triangles
0.25 mm max image deviation

7,809 triangles
1.3 mm max image deviation

975 triangles
6.6 mm max image deviation

Figure 12: Close-up of several levels of detail of the armadillo model. *Top*: normal maps *Bottom*: per-vertex normals

Phong lighting model to compute the output color of each pixel. The current implementation looks up normals with 8 bits per component, which seems sufficient in practice (using [44])

8 ONGOING WORK AND CONCLUSIONS

There are several directions to pursue to improve our system for appearance-preserving simplification. We would like to experiment more with techniques to generate parameterizations that allow efficient representations of the mapped attributes as well as guarantee a one-to-one mapping to the texture plane.

It would be nice for the simplification component to do a better job of optimizing the 3D and texture coordinates of the generated vertex for each edge collapse, both in 3D and the texture plane. Also, it may be interesting to allow the attribute data of a map to influence the error metric. We would also like to integrate our technique with a simplification algorithm like [6] that deals well with imperfect input meshes and allows some topological changes. Finally, we want to display our resulting progressive meshes in a system that performs dynamic, view-dependent management of LODs.

Our current system demonstrates the feasibility and desirability of our approach to appearance-preserving simplification. It produces high-fidelity images using a small number of high-quality triangles. This approach should complement future graphics systems well as we strive for increasingly realistic real-time computer graphics.

ACKNOWLEDGMENTS

We would like to thank Venkat Krishnamurthy and Marc Levoy at the Stanford Computer Graphics Laboratory and Peter Schröder for the armadillo model, and Lifeng Wang and Xing Xing Computer for the lion model. Our visualization system implementation was made possible by the UNC PixelFlow Project and the Hewlett Packard Visualize PxFl team. We also appreciate the assistance of the UNC Walkthrough Project. This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9625217, ONR Young Investigator Award, Honda, Intel, NSF/ARPA Center for Computer Graphics and Scientific Visualization, and NIH/National Center for Research Resources Award 2P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy.

REFERENCES

[1] L. Williams, “Pyramidal Parametrics,” *SIGGRAPH 83 Conference Proceedings*, pp. 1--11, 1983.

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

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

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

[5] G. Turk, “Re-tiling polygonal surfaces,” *Comput. Graphics*, vol. 26, pp. 55--64, 1992.

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

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

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

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

[10] R. Klein, G. Liebich, and W. Straßer, “Mesh Reduction with Error Control,” in *IEEE Visualization '96*: IEEE, October 1996.

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

[12] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle, “Multiresolution Analysis of Arbitrary Meshes,” in *SIGGRAPH'95 Conference Proceedings*, 1995, pp. 173--182.

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

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

[15] H. Hoppe, “Progressive Meshes,” in *SIGGRAPH 96 Conference Proceedings*: ACM SIGGRAPH, 1996, pp. 99--108.

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

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

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

[19] M. Hughes, A. Lastra, and E. Saxe , “Simplification of Global-Illumination Meshes,” *Proceedings of Eurographics '96, Computer Graphics Forum*, vol. 15, pp. 339-345, 1996.

[20] C. Erikson and D. Manocha, “Simplification Culling of Static and Dynamic Scene Graphs,” UNC-Chapel Hill Computer Science TR98-009, 1998.

[21] P. Schroder and W. Sweldens, “Spherical Wavelets: Efficiently Representing Functions on the Sphere,” *SIGGRAPH 95 Conference Proceedings*, pp. 161--172, August 1995.

[22] A. Certain, J. Popovic, T. Derose, T. Duchamp, D. Salesin, and W. Stuetzle, “Interactive Multiresolution Surface Viewing,” in *Proc. of ACM Siggraph*, 1996, pp. 91--98.

[23] R. L. Cook, “Shade trees,” in *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, H. Christiansen, Ed., July 1984, pp. 223--231.

[24] J. Blinn, “Simulation of Wrinkled Surfaces,” *SIGGRAPH '78 Conference Proceedings*, vol. 12, pp. 286--292, 1978.

[25] A. Fournier, “Normal distribution functions and multiple surfaces,” *Graphics Interface '92 Workshop on Local Illumination*, pp. 45--52, 1992.

[26] M. Peercy, J. Airey, and B. Cabral, “Efficient Bump Mapping Hardware,” *SIGGRAPH'97 Conference Proceedings*, pp. 303-306, 1997.

[27] M. Olano and A. Lastra, “A Shading Language on Graphics Hardware: The PixelFlow Shading System,” *SIGGRAPH 98 Conference Proceedings*, 1998.

[28] J. Kajiya, “Anisotropic Reflection Models,” *SIGGRAPH '85 Conference Proceedings*, pp. 15--21, 1985.

[29] B. Cabral, N. Max, and R. Springmeyer, “Bidirectional Reflection Functions From Surface Bump Maps,” *SIGGRAPH '87 Conference Proceedings*, pp. 273--281, 1987.

[30] S. Westin, J. Arvo, and K. Torrance, “Predicting Reflectance Functions From Complex Surfaces,” *SIGGRAPH '92 Conference Proceedings*, pp. 255--264, 1992.

[31] B. G. Becker and N. L. Max, “Smooth Transitions between Bump Rendering Algorithms,” in *Computer Graphics (SIGGRAPH '93 Proceedings)*, vol. 27, J. T. Kajiya, Ed., August 1993, pp. 183--190.

[32] V. Krishnamurthy and M. Levoy, “Fitting Smooth Surfaces to Dense Polygon Meshes,” *SIGGRAPH 96 Conference Proceedings*, pp. 313--324, 1996.

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

[34] L. Darsa, B. Costa, and A. Varshney, “Navigating Static Environments using Image-space simplification and morphing,” *Proc. of 1997 Symposium on Interactive 3D Graphics*, pp. 25-34, 1997.

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

[36] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, “Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments,” *SIGGRAPH 96 Conference Proceedings*, pp. 75--82, August 1996.

[37] J. Kent, W. Carlson, and R. Parent, “Shape transformation for polyhedral objects,” *SIGGRAPH '92 Conference Proceedings*, pp. 47--54, 1992.

[38] J. Maillot, H. Yahia, and A. Veroust, “Interactive Texture Mapping,” *SIGGRAPH'93 Conference Proceedings*, pp. 27--34, 1993.

[39] H. Pedersen, “A Framework for Interactive Texturing Operations on Curved Surfaces,” in *SIGGRAPH 96 Conference Proceedings*, *Annual Conference Series*, H. Rushmeier, Ed., August 1996, pp. 295--302.

[40] H. d. Fraysseix, J. Pach, and R. Pollack, “How to Draw a Planar Graph on a Grid,” *Combinatorica*, vol. 10, pp. 41--51, 1990.

[41] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, “A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees,” *J. Comput. Syst. Sci.*, vol. 30, pp. 54--76, 1985.

[42] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, “Algorithms for drawing graphs: an annotated bibliography,” *Comput. Geom. Theory Appl.*, vol. 4, pp. 235--282, 1994.

[43] M. d. Berg, M. v. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*: Springer-Verlag, 1997.

[44] R. F. Lyon, “Phong Shading Reformulation for Hardware Renderer Simplification,” Apple Computer #43, 1993.

Simplifying Polygonal Models Using Successive Mappings

Jonathan Cohen Dinesh Manocha Marc Olano
University of North Carolina at Chapel Hill
{cohenj,dm,olano}@cs.unc.edu

Abstract:
We present the use of mapping functions to automatically generate levels of detail with known error bounds for polygonal models. We develop a piece-wise linear mapping function for each simplification operation and use this function to measure deviation of the new surface from both the previous level of detail and from the original surface. In addition, we use the mapping function to compute appropriate texture coordinates if the original map has texture coordinates at its vertices. Our overall algorithm uses edge collapse operations. We present rigorous procedures for the generation of local planar projections as well as for the selection of a new vertex position for the edge collapse operation. As compared to earlier methods, our algorithm is able to compute tight error bounds on surface deviation and produce an entire continuum of levels of detail with mappings between them. We demonstrate the effectiveness of our algorithm on several models: a Ford Bronco consisting of over 300 parts and $70,000$ triangles, a textured lion model consisting of 49 parts and $86,000$ triangles, and a textured, wrinkled torus consisting of $79,000$ triangles.

CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — Curve, surface, solid, and object representations.
Additional Key Words and Phrases: model simplification, levels-of-detail, surface approximation, projection, linear programming.

1 Introduction

Automatic generation of levels of detail for polygonal data sets has become a task of fundamental importance for real-time rendering of large polygonal environments on current graphics systems. Many detailed models are obtained by scanning physical objects using range scanning systems or created by modeling systems. Besides surface geometry these models, at times, contain additional information such as normals, texture coordinates, color etc. As the field of model simplification continues to mature, many applications desire high quality simplifications, with tight error bounds of various types across the surface being simplified.

Most of the literature on simplification has focused purely on surface approximation. Many of these techniques give guaranteed error bounds on the deviation of the simplified surface from the original surface. Such bounds are useful for providing a measure of the screen-space deviation from the original surface. A few techniques have been proposed to preserve other attributes such as color or overall appearance. However, they are not able to give tight error bounds on these parameters. At times the errors accumulated in all these domains may cause visible artifacts, even though the surface deviation itself is properly constrained. We believe the most promising approach to measuring and bounding these attribute errors is to have a mapping between the original surface and the simplified surface. With such a mapping in hand, we are free to devise suitable methods for measuring and bounding each type of error.

Main Contribution: In this paper we present a new simplification algorithm, which computes a piece-wise linear mapping between the original surface and the simplified surface. The algorithm uses the edge collapse operation due to its simplicity, local control, and suitability for generating smooth transitions between levels of detail. We also present rigorous and complete algorithms for collapsing an edge to a vertex such that there are no local self-intersections. The algorithm keeps track of surface deviation from both the current level of detail as well as from the original surface. The main features of our approach are:

1. **Successive Mapping:** This mapping between the levels of detail is a useful tool. We currently use the mapping in several ways: to measure the distance between the levels of detail before an edge collapse, to choose a location for the generated vertex that minimizes this distance, to accumulate an upper bound on the distance between the new level of detail and the original surface, and to map surface attributes to the simplified surface.

2. **Tight Error Bounds:** Our approach can measure and minimize the error for surface deviation and is extendible to other attributes. These error bounds give guarantees on the shape of the simplified object and screen-space deviation.

3. **Generality:** Portions of our approach can be easily combined with other algorithms, such as simplification envelopes [5]. Furthermore, the algorithm for collapsing an edge into a vertex is rather general and does not restrict the vertex to lie on the original edge.

4. **Surface Attributes:** Given an original surface with texture coordinates, our algorithm uses the successive mapping to compute appropriate texture coordinates for the simplified

mesh. Other attributes such as color or surface normal can also be maintained with the mapping.

5. **Continuum of Levels of Details:** The algorithm incrementally produces an entire spectrum of levels-of-details as opposed to a few discrete levels. Furthermore, the algorithm incrementally stores the error bounds for each level. Thus, the simplified model can be stored as a progressive mesh [12] if desired.

The algorithm has been successfully applied to a number of models. These models consist of hundreds of parts and tens of thousands of polygons, including a Ford Bronco with 300 parts, a textured lion model and a textured wrinkled torus.

Organization: The rest of the paper is organized as follows. In Section 2, we survey related work on model simplification. We give an overview of our algorithm in Section 3. Section 4 discusses the types of mappings computed by the algorithm and describes the algorithm in detail. In Section 5, we present applications of these mapping. The implementation is discussed in Section 6 and its performance in Section 7. Finally, in Section 8 we compare our approach to other algorithms.

2 Previous Work

Automatic simplification has been studied in both the computational geometry and computer graphics literature for several years [1, 3, 5, 6, 7, 8, 9, 10, 12, 11, 15, 16, 17, 18, 19, 21, 22, 24]. Some of the earlier work by Turk [22] and Schroeder [19] employed heuristics based on curvature to determine which parts of the surface to simplify to achieve a model with the desired polygon count. Other work include that of Rossignac and Borrel [16] where vertices close to each other are clustered and a vertex is generated to represent them. This algorithm has been used in the *Brush* walkthrough system [18]. A dynamic view-dependent simplification algorithm has been presented in [24].

Hoppe et al. [12, 11] posed the model simplification problem into a global optimization framework, minimizing the least-squares error from a set of point-samples on the original surface. Later, Hoppe extended this framework to handle other scalar attributes, explicitly recognizing the distinction between smooth gradients and sharp discontinuities. He also introduced the progressive mesh [12], which is essentially a stored sequence of simplification operations, allowing quick construction of any desired level of detail along the continuum of simplifications. However, the algorithm in [12] provides no guaranteed error bounds.

There is considerable literature on model simplification using error bounds. Cohen and Varshney et al. [5, 23] have used envelopes to preserve the model topology and obtain tight error bounds for a simple simplification. But they do not produce an entire spectrum of levels of detail. Guéziec [9] has presented an algorithm for computing local error bounds inside the simplification process by maintaining tolerance volumes. However, it does not produce a suitable mapping between levels of detail. Bajaj and Schikore [1, 17] have presented an algorithm for producing a mapping between approximations and measure the error of scalar fields across the surface based on vertex-removals. Some of the results presented in this paper extend this work non-trivially to edge collapse operation. A detailed comparison with these approaches is presented in Section 8.

An elegant solution to the polygon simplification problem has been presented in [7, 8] where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multiresolution wavelet analysis is used over each patch. These methods preserve global topology, give error bounds on the simplified object and provide a mapping between levels of detail. In [3] they have been further extended to handle colored meshes. However, the initial mesh is not contained in the level of detail hierarchy, but can only be recovered to within an ϵ-tolerance. In some cases this is undesirable. Furthermore, the wavelet based approach can be somewhat conservative and for a given error bound, algorithms based on vertex removal and edge collapses [5, 12] have been *empirically* able to simplify more (in terms of reducing the polygon count).

3 Overview

Our simplification approach may be seen as a high-level algorithm which controls the simplification process with a lower-level cost function based on local mappings. Next we describe this high-level control algorithm and the idea of using local mappings for cost evaluation.

3.1 High-level Algorithm

At a broad level, our simplification algorithm is a generic greedy algorithm. Our simplification operation is the edge collapse. We initialize the algorithm by measuring the cost of all possible edge collapses, then we perform the edge collapses in order of increasing cost. The cost function tries to minimize local error bounds on surface deviation and other attributes. After performing each edge collapse, we locally re-compute the cost functions of all edges whose neighborhoods were affected by the collapse. This process continues until none of the remaining edges can be collapsed.

The output of our algorithm is the original model plus an ordered list of edge collapses and their associated cost functions. This *progressive mesh* [12] represents an entire *continuum* of levels of detail for the surface. A graphics application can choose to dynamically create levels of detail or to statically allocate a set of levels of detail to render the model with the desired quality or speed-up.

3.2 Local Mappings

The edge collapse operation we perform to simplify the surface contracts an edge (the *collapsed edge*) to a single, new vertex (the *generated vertex*). Most of the earlier algorithms position the generated vertex to one of the end vertices or mid-point of the collapse edge. However, these choices for generated vertex position may not minimize the deviation or error bound and can result in a local self-intersection. We choose a vertex position in two dimensions to avoid self-intersections and optimize in the third dimension to minimize error. This optimization of the generated vertex position and measurement of the error are the keys to simplifying the surface without introducing significant error.

For each edge collapse, we consider only the neighborhood of the surface that is modified by the operation (i.e. those faces, edges and vertices adjacent to the collapsed edge). There is a *natural mapping* between the neighborhood of the collapsed edge and the neighborhood of the generated vertex. Most of the triangles incident to the collapsed edge are stretched into corresponding triangles incident to the generated vertex. However, the two triangles that share the collapsed edge are themselves collapsed to edges (see Figure 1). These natural correspondences are one form of mapping

This natural mapping has two weaknesses.

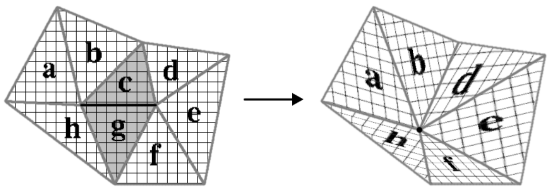


Figure 1: *The natural mapping primarily maps triangles to triangles. The two grey triangles map to edges, and the collapsed edge maps to the generated vertex*

1. The degeneracy of the triangles mapping to edges prevents us from mapping points of the simplified surface back to unique points on the original surface. This also implies that if we have any sort of attribute field across the surface, a portion of it disappears as a result of the operation.
2. The error implied by this mapping may be larger than necessary.

We measure the surface deviation error of the operation by the distances between corresponding points of our mapping. If we use the natural mapping, the maximum distance between any pair of points is defined as:

$$max(distance(\mathbf{v}_1, \mathbf{v}_{generated}), distance(\mathbf{v}_2, \mathbf{v}_{generated})),$$

where the collapsed edge corresponds to $(\mathbf{v}_1, \mathbf{v}_2)$ and $\mathbf{v}_{generated}$ is the generated vertex.

If we place the generated vertex at the midpoint of the collapsed edge, this distance error will be half the length of the edge. If we place the vertex at any other location, the error will be even greater.

We can create mappings that are free of degeneracies and often imply less error than the natural mapping. For simplicity, and to guarantee no self-intersections, we perform our mappings using planar projections of our local neighborhood. We refer to them as *successive mappings*.

4 Successive Mapping

In this section we present an algorithm to compute the mappings and their error bounds, which guide the simplification process. We present efficient and complete algorithms for computing a planar projection, finding a generated vertex in the plane, creating a mapping in the plane, and finally placing the generated vertex in 3D. The resulting algorithms utilize a number of techniques from computational geometry and are efficient in practice.

4.1 Computing a Planar Projection

Given a set of triangles in 3D, we present an efficient algorithm to compute a planar projection which is one-to-one to the set of triangles. The algorithm is guaranteed to find a plane, if it exists.

The projection we seek should be *one-to-one* to guarantee that the operations we perform in the plane are meaningful. For example, suppose we project a connected set of triangles onto a plane and then re-triangulate the polygon described by their boundary. The resulting set of triangles will contain no self-intersections, so long as the projection is one-to-one. Many other simplification algorithms, such as those by Turk [22], Schroeder [19] and Cohen, Varshney et al. [5], also used such projections for

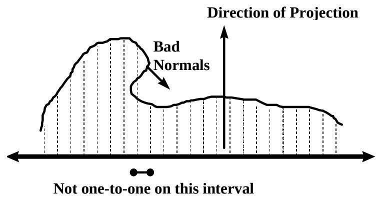


Figure 2: *A 2D example of an invalid projection*

vertex removal. However, they would choose a likely direction, such as the average of the normal vectors of the triangles of interest. To test the validity of the resulting projection, these earlier algorithms would project all the triangles onto the plane and check for self-intersections. This process can be relatively expensive and is not guaranteed to find a one-to-one projecting plane.

We improve on earlier brute-force approaches in two ways. First, we present a simple, linear-time algorithm for testing the validity of a given direction. Second, we present a slightly more complex, but still expected linear-time, algorithm which will find a valid direction if one exists, or report that no such direction exists for the given set of triangles.

4.1.1 Validity Test for Planar Projection

In this section, we briefly describe the algorithm which checks whether a given set of triangles have a one-to-one planar projection. Assume that we can calculate a consistent set of normal vectors for the set of triangles in question (if we cannot, the surface is non-orientable and cannot be mapped onto a plane in a one-to-one fashion). If the angle between a given direction of projection and the normal vector of each of the triangles is less than 90^o, then the direction of projection is valid, and defines a one-to-one mapping from the 3D triangles to a set of triangles in the plane of projection (any plane perpendicular to the direction of projection). Note that for a given direction of projection and a given set of triangles, this test involves only a single dot product and a sign test for each triangle in the set.

The correctness of the validity test can be established rigorously [4]. Due to space limitations, we do not present the detailed proof here. Rather, we give a short overview of the proof.

Figure 2 illustrates our problem in 2D. We would like to determine if the projection of the curve onto the line is one-to-one. Without loss of generality, assume the direction of projection is the y-axis. Each point on the curve projects to its x-coordinate on the line. If we traverse the curve from its left-most endpoint, we can project onto a previously projected location if and only if we reverse our direction along the x-axis. This can only occur when the y-component of the curve's normal vector goes from a positive value to a negative value. This is equivalent to our statement that the normal will be more than 90^o from the direction of projection. With a little more work, we can show that this characterization generalizes to 3D.

4.1.2 Finding a valid direction

The validity test in the previous section provides a quick method of testing the validity of a likely direction as a one-to-one mapping projection. But the wider the spread of the normal vectors of our set of triangles, the less likely we are to find a valid direction by using any sort of heuristic. It is possible, in fact, to

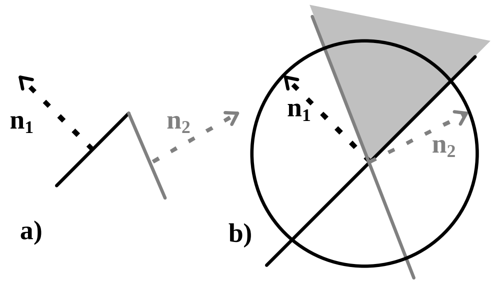


Figure 3: *A 2D example of the valid projection space. a) Two line segments and their normals. b) The 2D Gaussian circle, the planes corresponding to each segment, and the space of valid projection directions.*

compute the set of all valid directions of projection for a given set of triangles. However, to achieve greater efficiency and to reduce the complexity of the software system we choose to find only a single valid direction, which is typically all we require.

The *Gaussian sphere* [2] is the unit sphere on which each point corresponds to a unit normal vector with the same coordinates. Given a triangle, we define a plane through the origin with the same normal as the triangle. For a direction of projection to be valid with respect to this triangle, its point on the Gaussian sphere must lie on the correct side of this plane (i.e. within the correct hemisphere). If we consider two triangles simultaneously (shown in 2D in Figure 3) the direction of projection must lie on the correct side of the planes determined by the normal vectors of both triangles. This is equivalent to saying that the valid directions lie within the intersection of half-spaces defined by these two planes. Thus, the valid directions of projection for a set of N triangles lie within the intersection of N half-spaces.

This intersection of half-spaces forms a convex polyhedron. This polyhedron is a cone, with its apex at the origin and an unbounded base (shown as a triangular region in Figure 3). We can force this polyhedron to be bounded by adding more half-spaces (we use the six faces of a cube containing the origin). By finding a point on the interior of this cone and normalizing its coordinates, we shall construct a unit vector in the direction of projection.

Rather than explicitly calculating the boundary of the cone, we simply find a few corners (vertices) and average them to find a point that is strictly inside. By construction, the origin is definitely such a corner, so we just need to find three more *unique* corners to calculate an interior point. We can find each of these corners by solving a 3D *linear programming* problem. Linear programming allows us to find a point that maximizes a linear objective function subject to a collection of linear constraints [13]. The equations of the half-spaces serve as our linear constraints. We maximize in the direction of a vector to find the corner of our cone that lies the farthest in that direction.

As stated above, the origin is our first corner. To find the second corner, we try maximizing in the positive-x direction. If the resulting point is the origin, we instead maximize in the negative-x direction. To find the third corner, we maximize in a direction orthogonal to the line containing the first two corners. If the resulting point is one of the first two corners, we maximize in the opposite direction. Finally, we maximize in a direction orthogonal to the plane containing the first three corners. Once again, we may need to maximize in the opposite

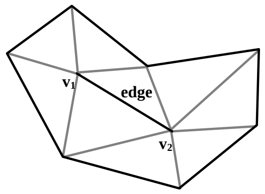


Figure 4: *The neighborhood of an edge as projected into 2D*

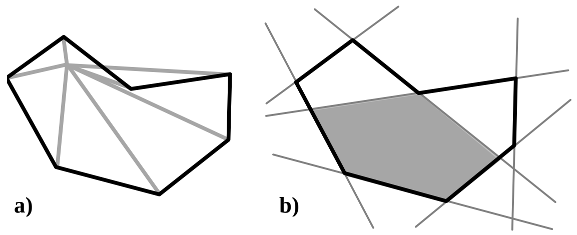


Figure 5: *a) An invalid 2D vertex position. b) The kernel of a polygon is the set of valid positions for a single, interior vertex to be placed. It is the intersection of a set of inward half-spaces.*

direction instead. Note that it is possible to reduce the worst-case number of optimizations from six to four by using the triangle normals to guide the selection of optimization vectors.

We used Seidel's linear time randomized algorithm [20] to solve each linear programming problem. A public domain implementation of this algorithm by Hohmeyer is available. It is very fast in practice.

4.2 Placing the Vertex in the Plane

In the previous section, we presented an algorithm to compute a valid plane. The edge collapse, which we use as our simplification operation, entails merging the two vertices of a particular edge into a single vertex. The topology of the resulting mesh is completely determined, but we are free to choose the position of the vertex, which will determine the geometry of the resulting mesh.

When we project the triangles neighboring the given edge onto a valid plane of projection, we get a triangulated polygon with two interior vertices, as shown in Figure 4. The edge collapse will reduce this edge to a single vertex. There will be edges connecting this generated vertex to each of the vertices of the polygon. In the context of this mapping approach, we would like the set of triangles around the generated vertex to have a one-to-one mapping with our chosen plane of projection, and thus to have a one-to- one mapping with the original edge neighborhood as well.

In this section, we present linear time algorithms both to test a candidate vertex position for validity, and to find a valid vertex position, if one exists.

4.2.1 Validity test for Vertex Position

The edge collapse operation leaves the boundary of the polygon in the plane unchanged. For the neighborhood of the generated vertex to have a one-to-one mapping with the plane, its edges must lie entirely within the polygon, ensuring that no edge crossings occur.

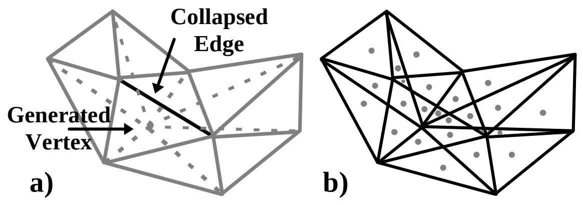


Figure 6: *a) Edge neighborhood and generated vertex neighborhood superimposed. b) A mapping in the plane, composed of 25 polygonal cells (each cell contains a dot). Each cell maps between a pair of planar elements in 3D.*

This 2D visibility problem has been well-studied in the computational geometry literature [14]. The generated vertex must have an unobstructed line of sight to each of the surrounding polygon vertices (unlike the vertex shown in Figure 5a). This condition holds if and only if the generated vertex lies within the polygon's *kernel*, shown in Figure 5b. This kernel is the intersection of inward-facing half-planes defined polygon's edges.

Given a potential vertex position in 2D, we test its validity by plugging it into the implicit-form equation for each of the polygon edges' line. If the position is on the interior with respect to each line, the position is valid, otherwise it is invalid.

4.2.2 Finding a Valid Position

The validity test highlighted above is useful if we wish to test out a likely candidate for the generated vertex position, such as the midpoint of the edge being collapsed. If such a heuristic choice succeeds, we can avoid the work necessary to compute a valid position directly.

Given the kernel definition for valid points, it is straightforward to find a valid vertex position using 2D linear programming. Each of the lines provides one of the constraints for the linear programming problem. Using the same methods as in Section 4.1.2, we can find a point in the kernel with no more than four calls to the linear programming routine. The first and second corners are found by maximizing in the positive- and negative-x directions. The final corner is found using a vector orthogonal to the first two corners.

4.3 Creating a Mapping in the Plane

After mapping the edge neighborhood to a valid plane and choosing a valid position for the generated vertex, we must define a mapping between the edge neighborhood and the generated vertex neighborhood. We shall map to each other the pairs of 3D points which project to identical points on the plane. These correspondences are shown in Figure 6a.

We can represent the mapping by a set of map cells, shown in Figure 6b. Each cell is a convex polygon in the plane and maps a piece of a triangle from the edge neighborhood to a similar piece of a triangle from the generated vertex neighborhood. The mapping represented by each cell is linear.

The vertices of the polygonal cells fall into *four* categories: vertices of the overall neighborhood polygon, vertices of the collapsed edge, the generated vertex itself, and edge-edge intersection points. We already know the locations of the first three categories of cell vertices, but we must calculate the edge-edge intersection points explicitly. Each such point is the intersection of an edge adjacent to the collapsed edge with an edge adjacent to the generated vertex. The number of such points can be quadratic (in the worst case) in the number of neighborhood edges. If we choose to construct the actual cells, we may do so by sorting the intersection points along each neighborhood edge and then walking the boundary of each cell.

4.4 Optimizing the 3D Vertex Position

Up to this point, we have projected the original edge neighborhood onto a plane, performed an edge collapse in this plane, and computed a mapping in the plane between these two local meshes. We are now ready to choose the position of the generated vertex in 3D. This 3D position will completely determine the geometry of the triangles surrounding the generated vertex.

To preserve our one-to-one mapping, it is necessary that all the points of the generated vertex neighborhood, including the generated vertex itself, project back into 3D along the direction of projection (the normal to the plane of projection). This restricts the 3D position of the generated vertex to the line parallel to the direction of projection and passing through the generated vertex's 2D position in the plane. We choose the vertex's position along this line such that it introduces as small a surface deviation as possible, that is it minimizes the maximum distance between any two corresponding points of the edge collapse neighborhood and the generated vertex neighborhood.

4.4.1 Distance function of the map

Each cell of our mapping determines a correspondence between a pair of planar elements. The maximum distance between any pair of planar functions must be at the boundary. For these pairs of polygons, the maximum distance must occur at a vertex. So the maximum distance for the entire mapping will always be at one of the interior cell vertices (because the cell vertices along the boundary do not move).

We parameterize the position of the generated vertex along its line of projection by a single parameter, t. As t varies, the distance between the corresponding cell vertices in 3D varies linearly. Note that these distances will always be along the direction of projection, because the distance between corresponding cell vertices is zero in the other two dimensions (those of the plane of projection). Because the distance is always positive, the distance function of each cell vertex is actually a pair of lines intersecting on the x-axis (shaped like a "V").

4.4.2 Minimizing the distance function

Given the distance function, we would like to choose the parameter t that minimizes the maximum distance between any pair of mapped points. This point is the minimum of the so-called *upper envelope*. For a set of k linear functions, we define the upper envelope function as follows:

$$U(t) = \{f_i(t) \mid f_i(t) > f_j(t) \, \forall \, i, j \;\; 1 \le i, \, j \le k; \;\; i \ne j\}.$$

For linear functions with no boundary conditions, this function is convex. Again we use linear programming to find the t value at which the minima occurs. We use this value of t to calculate the position of the generated vertex in 3D.

4.5 Accommodating Bordered Surfaces

Bordered surface are those containing edges adjacent to only a single triangle, as opposed to two triangles. Such surfaces are quite common in practice. Borders create some complications for the creation of a mapping in the plane. The problem is that the total shape of the neighborhood projected into the plane changes as a result of the edge collapse.

Bajaj and Schikore [1], who employ a vertex-removal approach, deal with this problem by mapping the removed vertex to a length-parameterized position along the border. This solution can be employed for the edge-collapse operation as well. In their case, a single vertex maps to a point on an edge. In ours, three vertices map to points on a chain of edges.

5 Applying Mappings

The previous section described the steps required to compute a mapping using planar projections. Given such a mapping, we would now like to apply it to the problem of computing high-quality surface approximations. We will next discuss how to bound the distance from the current simplified surface to the original surface, and how to compute new values for scalar surface attributes at the generated vertex.

5.1 Approximation of Original Surface Position

In the process of creating a mapping, we have measured the distance between the current surface and the surface resulting from the application of one more simplification operation. What we eventually desire is the distance between this new surface and the *original* surface. One possible solution would be to incorporate the information from all the previous mappings into an increasingly complex mapping as the simplification process proceeds. While this approach has the potential for a high degree of accuracy, the increasing complexity of the mappings is undesirable.

Instead, we associate with every point on the current surface a volume that is guaranteed to contain the corresponding point on the original surface. This volume is chosen conservatively so we can use the same volume for all points in a triangle. Thus the portion of the original surface corresponding to the triangle lies within the convolution of the triangle and the volume.

Possible volume choices include axis-aligned boxes, triangle-aligned prisms and sphere. For computational efficiency, we use axis-aligned boxes. To improve the error bounds, we do not require the box to be centered at the point of application.

The initial box at every triangle has zero size and displacement. After computing the mapping in the plane and choosing the 3D vertex position, we propagate the error by adjusting the size and displacement of the box associated with each new triangle.

For each cell vertex, we create a box that contains the boxes of the old triangles that meet there. The box for each new triangle is then constructed to contain the boxes of all of its cell vertices. By maintaining this containment property at the cell vertices, we guarantee it for all the interior points of the cells.

The maximum error for each triangle is the distance between a point on the triangle and the farthest corner of its associated box. The error of the entire current mesh is the largest error of any of its triangles.

5.2 Computing Texture Coordinates

The use of texture maps has become common over the last several years, as the hardware support for texture mapping has increased. Texture maps provide visual richness to computer-rendered models without adding more polygons to the scene.

Texture mapping requires two *texture coordinates* at every vertex of the model. These coordinates provide a parameterization of the texture map over the surface.

As we collapse an edge, we must compute texture coordinates for the generated vertex. These coordinates should reflect the original parameterization of the texture over the surface. We use linear interpolation to find texture coordinates for the corresponding point on the old surface, and assign these coordinates to the generated vertex.

This approach works well in many cases, as demonstrated in Section 7. However, there can still be some sliding of the texture across the surface. We can extend our mapping approach to also measure and bound the deviation of the texture. This extension, currently under development, will provide more guarantees about the smoothness of transitions between levels of detail.

As we add more error measures to our system, it becomes necessary to decide how to weight these errors to determine the overall cost of an edge collapse. Each type of error at an edge mandates a particular viewing distance based on a user-specified screen-space tolerance (e.g. number of allowable pixels of surface or texel deviation). We conservatively choose the farthest of these. At run-time, the user can still adjust the overall screen-space tolerance, but the relationships between the types of error are fixed.

6 System Implementation

All the algorithms described in this paper have been implemented and applied to various models. While the simplification process itself is only a pre-process with respect to the graphics application, we would still like it to be as efficient as possible. The most time-consuming part of our implementation is the re-computation of edge costs as the surface is simplified (Section 3.1). To reduce this computation time, we allow our approach to be slightly less greedy. Rather than recompute all the local edge costs after a collapse, we simply set a *dirty flag* for these edges. If the next minimum-cost edge we pick to collapse is dirty, we re-compute it's cost and pick again. This *lazy evaluation* of edge costs significantly speeds up the algorithm without much effect on the error across the progressive mesh.

More important than the cost of the simplification itself is the speed at which our graphics application runs. To maximize graphics performance, our display application renders simplified objects only with display lists. After loading the progressive mesh, it takes snapshots to use as levels of detail every time the triangle count decreases by a factor of two. These choices limit the memory usage to twice the original number of triangles, and virtually eliminate any run-time cost of simplification.

7 Results

We have applied our simplification algorithm to four distinct objects: a bunny rabbit, a wrinkled torus, a lion, and a Ford Bronco, with a total of 390 parts. Table 1 shows the total input complexity of each of these objects as well as the time needed to generate a progressive mesh representation. All simplifications were performed on a Hewlett-Packard 735/125 workstation.

Figure 7 graphs the complexity of each object vs. the number of pixels of screen-space error for a particular viewpoint. Each set

Model	Parts	Orig. Triangles	CPU Time (Min:Sec)
Bunny	1	69,451	9:05
Torus	1	79,202	10:53
Lion	49	86,844	8:52
Bronco	339	74,308	6:55

Table 1: *Simplifications performed. CPU time indicates time to generate a progressive mesh of edge collapses until no more simplification is possible.*

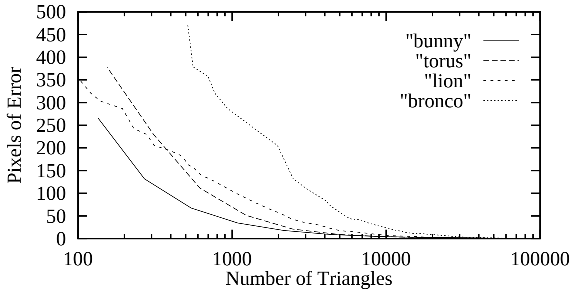


Figure 7: *Continuum of levels of detail for four models*

of data was measured with the object centered in the foreground of a 1000x1000-pixel viewport, with a 45^o field-of-view, like the Bronco in Plates 2 and 3. This was the easiest way for us to measure the continuum. Conveniently, this function of complexity vs. error at a fixed distance is proportional to the function of complexity vs. viewing distance with a fixed error. The latter is typically the function of interest.

Plate 1 shows the typical way of viewing levels of detail – with a fixed error bound and levels of detail changing as a function of distance. Plates 2 and 3 show close-ups of the Bronco model at full and reduced resolution.

Plates 4 and 5 show the application of our algorithm to the texture-mapped lion and wrinkled torus models. If you know how to free-fuse stereo image pairs, you can fuse the torii or any of the adjacent pairs of textured lion. Because the torii are rendered at an appropriate distance for switching between the two levels of detail, the images are nearly indistinguishable, and fuse to a sharp, clear image. The lions, however, are not rendered at their appropriate viewing distances, so certain discrepancies will appear as fuzzy areas. Each of the lion's 49 parts is individually colored in the wire-frame rendering to indicate which of its levels of detail is currently being rendered.

7.1 Applications of Projection Algorithm

We have also applied the technique of finding a one-to-one planar projection to the simplification envelopes algorithm [5]. The simplification envelopes method requires the calculation of a vertex normal at each vertex that may be used as a direction to offset the vertex. The criterion for being able to move a vertex without creating a local self-intersection is the same as the criterion for being able to project to a plane. The algorithm presented in [5] used a heuristic based on averaging the face normals.

By applying the projection algorithm based on linear programming (presented in Section 4.1) to the computation of the offset directions, we were able to perform more drastic simplifications. The simplification envelopes method could previously only reduce the bunny model to about 500 triangles, without resulting in any self-intersections. Using the new approach, the algorithm can reduce the bunny to 129 triangles, with no self-intersections.

7.2 Video Demonstration

We have produced a video demonstrating the capabilities of the algorithm and smooth switching between different levels-of-details for different models. It shows the speed-up in the frame rate for eight circling Bronco models (about a factor of six) with almost no degradation in image quality. This is based on mapping the object space error bounds to screen space, which can measure the maximum error in number of pixels. The video also highlights the performance on simplifying textured models, showing smooth switching between levels of detail. The texture coordinates were computed using the algorithm in Section 5.2.

8 Comparison to Previous Work

While concrete comparisons are difficult to make without careful implementation of all the related approaches readily available, we compare some of the features of our algorithm with that of others. The efficient and complete algorithms for computing the planar projection and placing the generated vertex after edge collapse should improve the performance of all the earlier algorithms that use vertex removals or edge collapses.

We compared our implementation with that of the simplification envelopes approach [5]. We generated levels of detail of the Stanford bunny model (70,000 triangles) using the simplification envelopes method, then generated levels of detail with the same number of triangles using the successive mapping approach. Visually, the models were comparable. The error bounds for the simplification envelopes method were smaller by about a factor of two, but the error bounds for the two methods measure different things. Simplification envelopes only bounds the surface deviation in the direction normal to the original surface, while the mapping approach prevents the surface from sliding around as well. Also, simplification envelopes created local creases in the bunnies, resulting in some shading artifacts. The successive mapping approach discourages such creases by its use of planar projections. At the same time, the performance of the simplification envelopes approach (in terms complexity vs. error) has been improved by our new projection algorithm.

Hoppe's progressive mesh [12] implementation is more complete than ours in its handling of colors, textures, and discontinuities. However, this technique provides no guaranteed error bounds, so there is no simple way to automatically choose switching distances that guarantee some visual quality.

The multi-resolution analysis approach to simplification [7, 8] does, in fact, provide strict error bounds as well as a mapping between surfaces. However, the requirements of its subdivision topology and the coarse granularity of its simplification operation do not provide the local control of the edge collapse. In particular, it does not deal well with sharp edges. Hoppe [12] has compared his progressive meshes with the multi-resolution analysis meshes. For a given number of triangles, his progressive meshes provide much higher visual quality. Therefore, for a given error bound, we expect our mapping algorithm to be able to simplify more than the multi-resolution approach.

Guéziec's tolerance volume approach [9] also uses edge collapses with local error bounds. Unlike the boxes used by the successive mapping approach, Guéziec's error volume can grow as the simplified surface fluctuates closer to and farther away from the original surface. This is due to the fact that it uses spheres which always remain centered at the vertices, and the newer spheres must always contain the older spheres. The boxes used by our successive mapping approach are not centered on the surface and do not grow as a result of such fluctuations. Also, the tolerance volume approach does not generate mappings between the surfaces for use with other attributes.

We have made several significant improvements over the simplification algorithm presented by Bajaj and Schikore [1, 17]. First, we have replaced their projection heuristic with a robust algorithm for finding a valid direction of projection. Second, we have generalized their approach to handle more complex operations, such as the edge collapse. Finally, we have presented an error propagation algorithm which correctly bounds the error in the surface deviation. Their approach represented error as infinite slabs surrounding each triangle. Because there is no information about the extent of these slabs, it is impossible to correctly propagate the error from a slab with one orientation to a new slab with a different orientation.

9 Future Work

We are currently working on bounding the screen-space deviation of the texture coordinates. By bounding the error of the texture coordinates, we will provide one type of bound on the deviation of surface colors (from a texture map) or normals (from a bump map). We also plan to measure and bound the deviation of colors and normals specified directly at the polygon vertices.

There are cases where the projection onto a plane produces mappings with unnecessarily large error. We only optimize surface position in the direction orthogonal to the plane of projection. It would be useful to generate and optimize mappings directly in 3D to produce better simplifications.

Our system currently handles non-manifold topologies by breaking them into independent surfaces, which does not maintain connectivity between the components. Handling such non-manifold regions directly may provide higher visual fidelity for large screen-space tolerances.

10 Acknowledgments

We would like to thank Stanford Computer Graphics Laboratory for the bunny model, Stefan Gottschalk for the wrinkled torus model, Lifeng Wang and Xing Xing Computer for the lion model from the Yuan Ming Garden, and Division and Viewpoint for the Ford Bronco model. Thanks to Michael Hohmeyer for the linear programming library. We would also like to thank the UNC Walkthrough Group and Carl Mueller. This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9319957, NSF Grant CCR-9625217, ONR Young Investigator Award, Intel, DARPA Contract DABT63-93-C-0048, NSF/ARPA Center for Computer Graphics and Scientific Visualization, and NIH/National Center for Research Resources Award 2 P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy.

References

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

[2] M.P. Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, 1976.

[3] A. Certain, J. Popovic, T. Derose, T. Duchamp, D. Salesin, and W. Stuetzle. Interactive multiresolution surface viewing. In *Proc. of ACM Siggraph*, pages 91–98, 1996.

[4] J. Cohen, D. Manocha, and M. Olano. Simplifying polygonal models using successive mappings. Technical Report TR97-011, Department of Computer Science, UNC Chapel Hill, 1997.

[5] J. Cohen, A. Varshney, D. Manocha, and G. Turk et al. Simplification envelopes. In *Proc. of ACM Siggraph'96*, pages 119–128, 1996.

[6] M.J. Dehaemer and M.J. Zyda. Simplification of objects rendered by polygonal approximations. *Computer and Graphics*, 15(2):175–184, 1981.

[7] T. Derose, M. Lounsbery, and J. Warren. Multiresolution analysis for surfaces of arbitrary topology type. Technical Report TR 93-10-05, Department of Computer Science, University of Washington, 1993.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proc. of ACM Siggraph*, pages 173–182, 1995.

[9] A. Guéziec. Surface simplification with variable tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, pages 132–139, November 1995.

[10] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface '94*, pages 43–50, May 1994.

[11] H. Hoppe, T. Derose, T. Duchamp, J. Mcdonald, and W. Stuetzle. Mesh optimization. In *Proc. of ACM Siggraph*, pages 19–26, 1993.

[12] Hugues Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996.

[13] B. Kolman and R. Beck. *Elementary Linear Programming with Applications*. Academic Press, New York, 1980.

[14] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[15] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, 462, Aug. 1996. Proc. Eurographics '96.

[16] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[17] D. Schikore and C. Bajaj. Decimation of 2d scalar data with error control. Technical report, Computer Science Report CSD-TR-95-004, Purdue University, 1995.

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

[19] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. In *Proc. of ACM Siggraph*, pages 65–70, 1992.

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

[21] D. C. Taylor and W. A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proc. Graphics Interface '94*, pages 33–42, Banff, Canada, May 1994.

[22] G. Turk. Re-tiling polygonal surfaces. In *Proc. of ACM Siggraph*, pages 55–64, 1992.

[23] A. Varshney. *Hierarchical Geometric Approximations*. PhD thesis, University of N. Carolina, 1994.

[24] J.C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, June 1997.

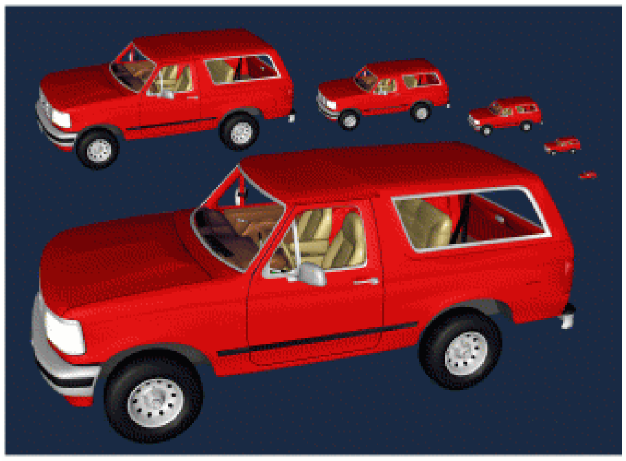


Plate 1: 6 views of the Ford Bronco model, all at 2 pixels of error (0.17 mm)

Triangle counts:	41,855	12,939
	27,970	8,385
	20,922	4,766

Plate 2: Bronco at full resolution 74,000 triangles

Plate 3: 26 pixels of error (4.4 mm) 9,000 triangles

Plate 4: Wrinkled torus model at a transitional distance for 1 pixel of error (0.085 mm)

39,600 triangles 19,800 triangles

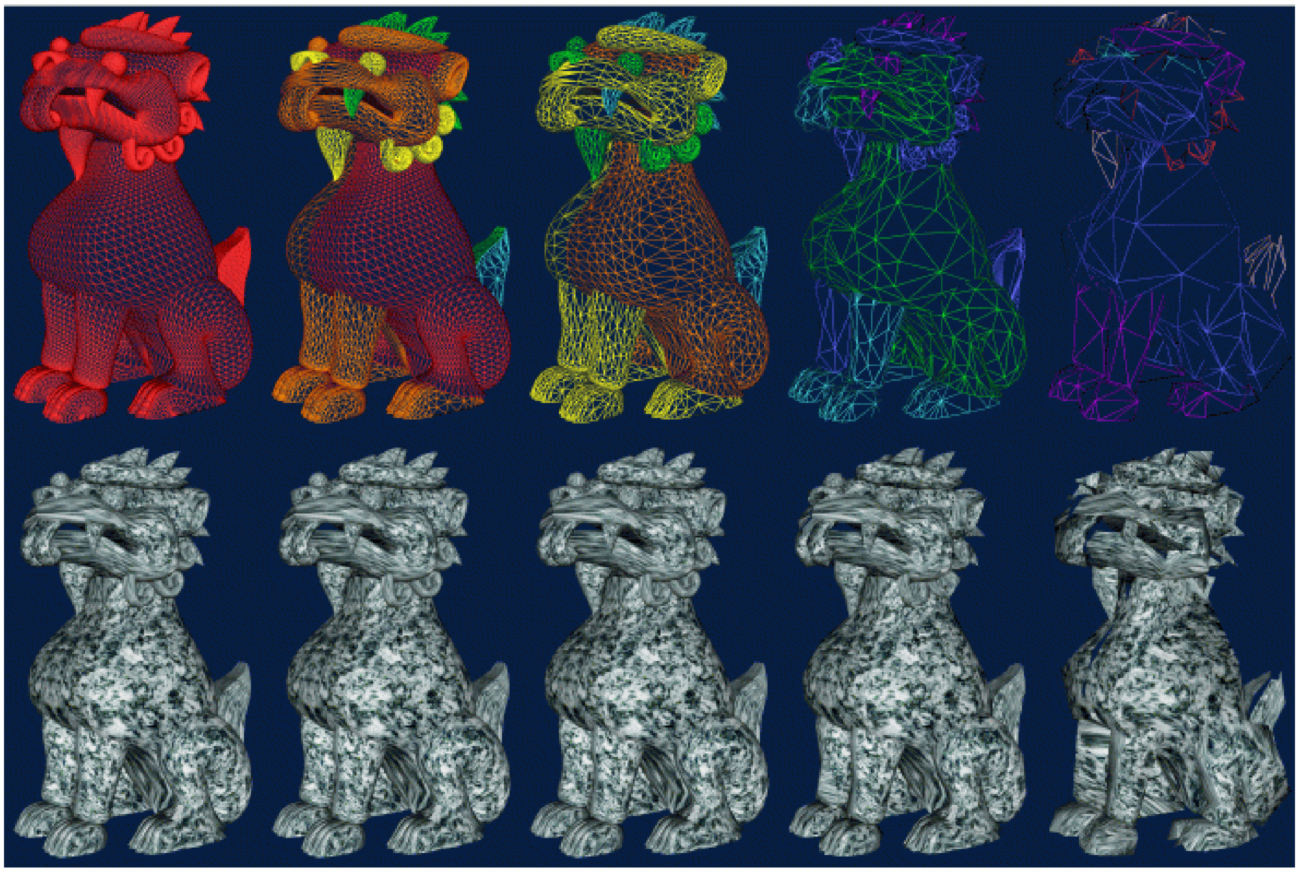


Plate 5: 6 levels of detail for the lion (colors indicate levels of detail of individual parts)

0 pixels of error	1 pixel of error	3 pixels of error	23 pixels of error	76 pixels of error
0 mm of error	0.085 mm of error	0.25 mm of error	1.9 mm of error	6.4 mm of error
86,000 triangles	29,000 triangles	14,000 triangles	4,000 triangles	1,000 triangles

Simplification Envelopes

Jonathan Cohen* Amitabh Varshney† Dinesh Manocha* Greg Turk* Hans Weber*
Pankaj Agarwal‡ Frederick Brooks* William Wright*
http://www.cs.unc.edu/~geom/envelope.html

Abstract

We propose the idea of *simplification envelopes* for generating a hierarchy of level-of-detail approximations for a given polygonal model. Our approach guarantees that all points of an approximation are within a user-specifiable distance ϵ from the original model and that all points of the original model are within a distance ϵ from the approximation. Simplification envelopes provide a general framework within which a large collection of existing simplification algorithms can run. We demonstrate this technique in conjunction with two algorithms, one local, the other global. The local algorithm provides a fast method for generating approximations to large input meshes (at least hundreds of thousands of triangles). The global algorithm provides the opportunity to avoid local "minima" and possibly achieve better simplifications as a result.

Each approximation attempts to minimize the total number of polygons required to satisfy the above ϵ constraint. The key advantages of our approach are:

- General technique providing guaranteed error bounds for genus-preserving simplification
- Automation of both the simplification process and the selection of appropriate viewing distances
- Prevention of self-intersection
- Preservation of sharp features
- Allows variation of approximation distance across different portions of a model

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation — Display algorithms; I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — Curve, surface, solid, and object representations.
Additional Key Words and Phrases: hierarchical approximation, model simplification, levels-of-detail generation, shape approximation, geometric modeling, offsets.

*Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. {cohenj,weberh,manocha,turk,brooks,wright}@cs.unc.edu
†Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400. varshney@cs.sunysb.edu
‡Department of Computer Science, Duke University, Durham, NC 27708-0129. pankaj@cs.duke.edu

1 Introduction

We present the framework of *simplification envelopes* for computing various levels of detail of a given polygonal model. These hierarchical representations of an object can be used in several ways in computer graphics. Some of these are:

- Use in a level-of-detail-based rendering algorithm for providing desired frame update rates [4, 9].
- Simplifying traditionally over-sampled models such as those generated from volume datasets, laser scanners, and satellites for storage and reducing CPU cycles during processing, with relatively few or no disadvantages [10, 11, 13, 15, 21, 23].
- Using low-detail approximations of objects for illumination algorithms, especially radiosity [19].

Simplification envelopes are a generalization of *offset surfaces*. Given a polygonal representation of an object, they allow the generation of minimal approximations that are guaranteed not to deviate from the original by more than a user-specifiable amount while preserving global topology. We surround the original polygonal surface with two envelopes, then perform simplification within this volume. A sample application of the algorithms we describe can be seen in Figure 1.

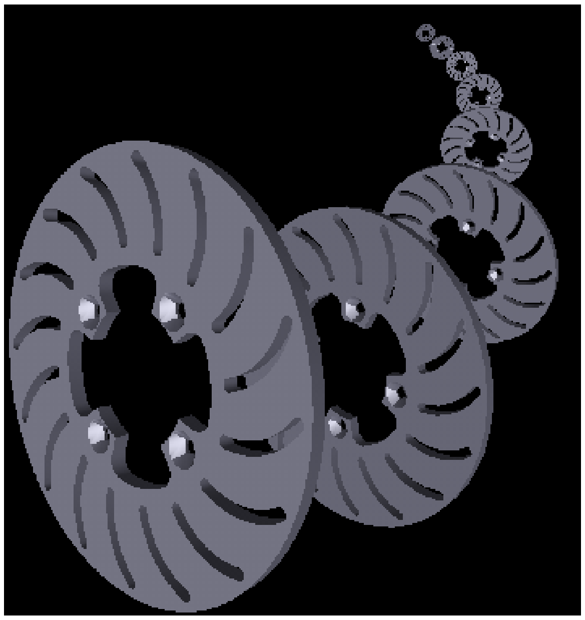


Figure 1: *A level-of-detail hierarchy for the rotor from a brake assembly.*

Such an approach has several benefits in computer graphics. First, one can very precisely quantify the amount of approximation that is tolerable under given circumstances. Given a user-specified error in number of pixels of deviation of an object's silhouette, it is possible to choose which level of detail to view from a particular distance to maintain that pixel error bound. Second, this approach allows one a fine control over which regions of an object should be approximated more and which ones less. This could be used for selectively preserving those features of an object that are *perceptually* important. Third, the user-specifiable tolerance for approximation is the only parameter required to obtain the approximations; fine tweaking of several parameters depending upon the object to be approximated is not required. Thus, this approach is quite useful for *automating the process* of topology-preserving simplifications of a large number of objects. This problem of *scalability* is important for any simplification algorithm. One of our main goals is to create a method for simplification which is not only automatic for large datasets, but tends to preserve the shapes of the original models.

The rest of the paper is organized in the following manner: we survey the related work in Section 2, explain our assumptions and terminology in Section 3, describe the envelope and approximation computations in Sections 4 and 5, present some useful extentions to and properties of the approximation algorithms in Section 6, and explain our implementation and results in Section 7.

2 Background

Approximation algorithms for polygonal models can be classified into two broad categories:

- **Min-# Approximations**: For this version of the approximation problem, given some error bound ϵ, the objective is to minimize the number of vertices such that no point of the approximation $\mathcal{A}$ is farther than ϵ distance away from the input model $\mathcal{I}$.

- **Min-ϵ Approximations**: Here we are given the number of vertices of the approximation $\mathcal{A}$ and the objective is to minimize the error, or the difference, between $\mathcal{A}$ and $\mathcal{I}$.

Previous work in the area of min-# approximations has been done by [6, 20] where they adaptively subdivide a series of bicubic patches and polygons over a surface until they fit the data within the tolerance levels.

In the second category, work has been done by several groups. Turk [23] first distributes a given number of vertices over the surface depending on the curvature and then re-triangulates them to obtain the final mesh. Schroeder et al. [21] and Hinker and Hansen [13] operate on a set of local rules — such as deleting edges or vertices from almost coplanar adjacent faces, followed by local re-triangulation. These rules are applied iteratively till they are no longer applicable. A somewhat different local approach is taken in [18] where vertices that are close to each other are clustered and a new vertex is generated to represent them. The mesh is suitably updated to reflect this.

Hoppe et al. [14] proceed by iteratively optimizing an energy function over a mesh to minimize both the distance of the approximating mesh from the original, as well as the number of approximating vertices. An interesting and elegant solution to the problem of polygonal simplification by using wavelets has been presented in [7, 8] where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multiresolution wavelet analysis is used over each patch. This wavelet approach preserves global topology.

In computational geometry, it has been shown that computing the minimal-facet ϵ-approximation is NP-hard for both convex polytopes [5] and polyhedral terrains [1]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_o be the size of a min-# approximation. An algorithm has been given in [16] for computing an ϵ-approximation of size $O(k_o \log n)$ for convex polytopes. This has recently been improved by Clarkson in [3]; he proposes a randomized algorithm for computing an approximation of size $O(k_o \log k_o)$ in expected time $O(k_o n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [2] Brönnimann and Goodrich observed that a variant of Clarkson's algorithm yields a polynomial-time deterministic algorithm that computes an approximation of size $O(k_0)$. Working with polyhedral terrains, [1] present a polynomial-time algorithm that computes an ϵ-approximation of size $O(k_o \log k_o)$ to a polyhedral terrain.

Our work is different from the above in that it allows adaptive, genus-preserving, ϵ-approximation of arbitrary polygonal objects. Additionally, we can simplify bordered meshes and meshes with holes. In terms of direct comparison with the global topology preserving approach presented in [7, 8], for a given ϵ our algorithm has been *empirically* able to obtain "reduced" simplifications, which are much smaller in output size (as demonstrated in Section 7). The algorithm in [18] also guarantees a bound in terms of the Hausdorff metric. However, it is not guaranteed to preserve the genus of the original model.

3 Terminology and Assumptions

Let us assume that $\mathcal{I}$ is a three-dimensional compact and orientable object whose polygonal representation $\mathcal{P}$ has been given to us. Our objective is to compute a piecewise-linear approximation $\mathcal{A}$ of $\mathcal{P}$. Given two piecewise linear objects $\mathcal{P}$ and $\mathcal{Q}$, we say that $\mathcal{P}$ and $\mathcal{Q}$ are *ϵ-approximations* of each other iff every point on $\mathcal{P}$ is within a distance ϵ of some point of $\mathcal{Q}$ and every point on $\mathcal{Q}$ is within a distance ϵ of some point of $\mathcal{P}$. Our goal is to outline a method to generate two envelope surfaces surrounding $\mathcal{P}$ and demonstrate how the envelopes can be used to solve the following polygonal approximation problem. Given a polygonal representation $\mathcal{P}$ of an object and an approximation parameter ϵ, generate a genus-preserving ϵ-approximation $\mathcal{A}$ with minimal number of polygons such that the vertices of $\mathcal{A}$ are a subset of vertices of $\mathcal{P}$.

We assume that all polygons in $\mathcal{P}$ are triangles and that $\mathcal{P}$ is a well-behaved polygonal model, i.e., every edge has either one or two adjacent triangles, no two triangles interpenetrate, there are no unintentional "cracks" in the model, no T-junctions, etc.

We also assume that each vertex of $\mathcal{P}$ has a single normal vector, which must lie within 90° of the normal of each of its surrounding triangles. For the purpose of rendering, each vertex may have either a single normal or multiple normals. For the purpose of generating envelope surfaces, we shall compute a single vertex normal as a combination of the normals of the surrounding triangles.

The three-dimensional ϵ-offset surface for a parametric surface

$$\mathbf{f}(s,t) = (f_1(s,t), f_2(s,t), f_3(s,t)),$$

whose unit normal to **f** is

$$\mathbf{n}(s,t) = (n_1(s,t), n_2(s,t), n_3(s,t)),$$

is defined as $\mathbf{f}^\epsilon(s,t) = (f_1^\epsilon(s,t), f_2^\epsilon(s,t), f_3^\epsilon(s,t))$, where

$$f_i^\epsilon(s,t) = f_i(s,t) + \epsilon n_i(s,t).$$

Note that offset surfaces for a polygonal object can self-intersect and may contain non-linear elements. We define a simplification envelope $\mathcal{P}(+\epsilon)$ (respectively $\mathcal{P}(-\epsilon)$) for an object $\mathcal{I}$ to be a *polygonal* surface that lies *within* a distance of ϵ from every point p on $\mathcal{I}$ in the same (respectively opposite) direction as the normal to $\mathcal{I}$ at p. Thus, the simplification envelopes can be thought of as an approximation to offset surfaces. Henceforth we shall refer to simplification envelope by simply envelope.

Let us refer to the triangles of the given polygonal representation $\mathcal{P}$ as the *fundamental triangles*. Let $e = (v_1, v_2)$ be an edge of $\mathcal{P}$. If the normals $\mathbf{n}_1, \mathbf{n}_2$ to $\mathcal{I}$ at both v_1 and v_2, respectively, are identical, then we can construct a plane π_e that passes through the edge e and has a normal that is perpendicular to that of v_1. Thus v_1, v_2 and their normals all lie along π_e. Such a plane defines two half-spaces for edge e, say π_e^+ and π_e^- (see Fig 2(a)). However, in general the normals $\mathbf{n}_1$ and $\mathbf{n}_2$ at the vertices v_1 and v_2 need not be identical, in which case it is not clear how to define the two half-spaces for an edge. One choice is to use a *bilinear patch* that passes through v_1 and v_2 and has a tangent $\mathbf{n}_1$ at v_1 and $\mathbf{n}_2$ at v_2. Let us call such a bilinear patch for e as the *edge half-space* β_e. Let the two half-spaces for the edge e in this case be β_e^+ and β_e^-. This is shown in Fig 2(b).

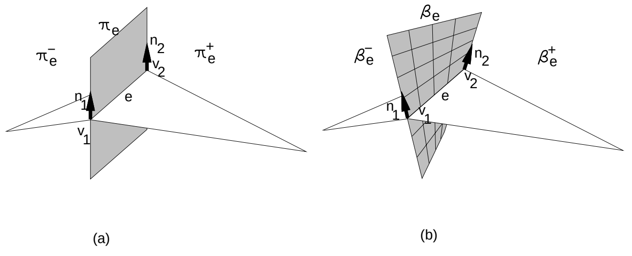


Figure 2: *Edge Half-spaces*

Let the vertices of a fundamental triangle be v_1, v_2, and v_3. Let the coordinates and the normal of each vertex v be represented as $\mathbf{c}(v)$ and $\mathbf{n}(v)$, respectively. The coordinates and the normal of a $(+\epsilon)$-offset vertex v_i^+ for a vertex v_i are: $\mathbf{c}(v_i^+) = \mathbf{c}(v_i) + \epsilon\mathbf{n}(v_i)$, and $\mathbf{n}(v_i^+) = \mathbf{n}(v_i)$. The $(-\epsilon)$-offset vertex can be similarly defined in the opposite direction. These offset vertices for a fundamental triangle are shown in Figure 3.

Now consider the closed object defined by v_i^+ and v_i^-, $i = 1, 2, 3$. It is defined by two triangles, at the top and bottom, and three edge half-spaces. This object contains the fundamental triangle (shown shaded in Figure 3) and we will henceforth refer to it as the *fundamental prism*.

4 Envelope Computation

In order to preserve the input topology of $\mathcal{P}$, we desire that the simplification envelopes do not self-intersect. To meet this criterion we reduce our level of approximation at certain places. In other words, to guarantee that no intersections amongst the envelopes occur, we have to be

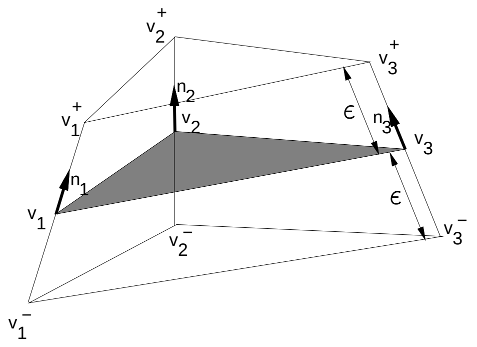


Figure 3: *The Fundamental Prism*

content at certain places with the distance between $\mathcal{P}$ and the envelope being smaller than ϵ. This is how simplification envelopes differ from offset surfaces.

We construct our envelope such that each of its triangles corresponds to a fundamental triangle. We offset each vertex of the original surface in the direction of its normal vector to transform the fundamental triangles into those of the envelope.

If we offset each vertex v_i by the same amount ϵ, to get the offset vertices v_i^+ and v_i^-, the resulting envelopes, $\mathcal{P}(+\epsilon)$ and $\mathcal{P}(-\epsilon)$, may contain self-intersections because one or more offset vertices are closer to some non-adjacent fundamental triangle. In other words, if we define a Voronoi diagram over the fundamental triangles of the model, the condition for the envelopes to intersect is that there be at least one offset vertex lying in the Voronoi region of some non-adjacent fundamental triangle. This is shown in Figure 4 by means of a two-dimensional example. In the figure, the offset vertices b^+ and c^+ are in the Voronoi regions of edges other than their own, thus causing self-intersection of the envelope.

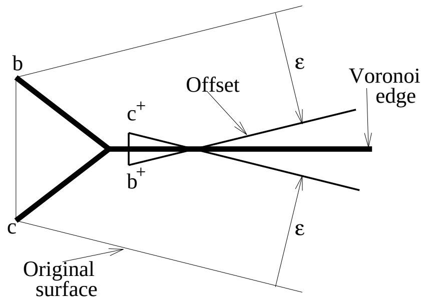


Figure 4: *Offset Surfaces*

Once we make this observation, the solution to avoid self-intersections becomes quite simple — just do not allow an offset vertex to go beyond the Voronoi regions of its adjacent fundamental triangles. In other words, determine the positive and negative ϵ for each vertex v_i such that the vertices v_i^+ and v_i^- determined with this new ϵ do not lie in the Voronoi regions of the non-adjacent fundamental triangles.

While this works in theory, efficient and robust computation of the three-dimensional Voronoi diagram of the fundamental triangles is non-trivial. We now present two methods for computing the reduced ϵ for each vertex, the first method analytical, and the second numerical.

4.1 Analytical ϵ Computation

We adopt a conservative approach for recomputing the ϵ at each vertex. This approach underestimates the values for the positive and negative ϵ. In other words, it guarantees the envelope surfaces not to intersect, but it does not guarantee that the ϵ at each vertex is the largest permissible ϵ. We next discuss this approach for the case of computing the positive ϵ for each vertex. Computation of negative ϵ follows similarly.

Consider a fundamental triangle t. We define a prism t_p for t, which is conceptually the same as its fundamental prism, but uses a value of 2ϵ instead of ϵ for defining the envelope vertices. Next, consider all triangles Δ_i that do not share a vertex with t. If Δ_i intersects t_p above t (the directions above and below t are determined by the direction of the normal to t, above is in the same direction as the normal to t), we find the point on Δ_i that lies within t_p and is closest to t. This point would be either a vertex of Δ_i, or the intersection point of one of its edges with the three sides of the prism t_p. Once we find the point of closest approach, we compute the distance δ_i of this point from t. This is shown in Figure 5.

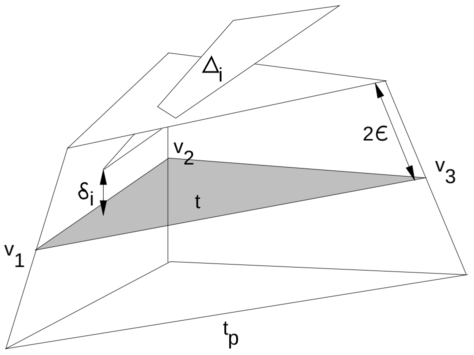


Figure 5: *Computation of* δ_i

Once we have done this for all Δ_i, we compute the new value of the positive ϵ for the triangle t as $\epsilon_{new} = \frac{1}{2}\min_i \delta_i$. If the vertices for this triangle t have this value of positive ϵ, their positive envelope surface will not self-intersect. Once the $\epsilon_{new}(t)$ values for all the triangles t have been computed, the $\epsilon_{new}(v)$ for each vertex v is set to be the minimum of the $\epsilon_{new}(t)$ values for all its adjacent triangles.

We use an octree in our implementation to speed up the identification of triangles Δ_i that intersect a given prism.

4.2 Numerical ϵ Computation

To compute an envelope surface numerically, we take an iterative approach. Our envelope surface is initially identical to the input model surface. In each iteration, we sequentially attempt to move each envelope vertex a fraction of the ϵ distance in the direction of its normal vector (or the opposite direction, for the inner envelope). This effectively stretches or contracts all the triangles adjacent to the vertex. We test each of these adjacent triangles for intersection with each other and the rest of the model. If no such intersections are found, we accept the step, leaving the vertex in this new position. Otherwise we reject it, moving the vertex back to its previous position. The iteration terminates when all vertices have either moved ϵ or can no longer move.

In an attempt to guarantee that each vertex gets to move a reasonable amount of its potential distance, we use an adaptive step size. We encourage a vertex to move at least K (an arbitrary constant which is scaled with respect to ϵ and the size of the object) steps by allowing it to reduce its step size. If a vertex has moved less than K steps and its move is been rejected, it divides its step size in half and tries again (with some maximum number of divides allowed on any particular step). Notice that if a vertex moves i steps and is rejected on the $(i+1)$st step, we know it has moved at least $i/(i+1)$ % of its potential distance, so $K/(K+1)$ which is a lower bound of sorts. It is possible, though rare, for a vertex to move less than K steps, if its current position is already quite close to another triangle.

Each vertex also has its own initial step size. We first choose a global, maximum step size based on a global property: either some small percentage of the object's bounding box diagonal length or ϵ/K, whichever is smaller. Now for each vertex, we calculate a local step size. This local step size is some percentage of the vertex's shortest incident edge (only those edges within 90^o of the offset direction are considered). We set the vertex's step size to the minimum of the global step size and its local step size. This makes it likely that each vertex's step size is appropriate for a step given the initial mesh configuration.

This approach to computing an envelope surface is robust, simple to implement (if difficult to explain), and fair to all the vertices. It tends to maximize the minimum offset distance amongst the envelope vertices. It works fairly well in practice, though there may still be some room for improvement in generating maximal offsets for thin objects. Figure 6 shows internal and external envelopes computed for three values of ϵ using this approach.

As in the analytical approach, a simple octree data structure makes these intersection tests reasonably efficient, especially for models with evenly sized triangles.

5 Generation of Approximation

Generating a surface approximation typically involves starting with the input surface and iteratively making modifications to ultimately reduce its complexity. This process may be broken into two main stages: *hole creation*, and *hole filling*. We first create a hole by removing some connected set of triangles from the surface mesh. Then we fill the hole with a smaller set of triangles, resulting in some reduction of the mesh complexity.

We demonstrate the generality of the simplification envelope approach by designing two algorithms. The hole filling stages of these algorithms are quite similar, but their hole creation stages are quite different. The first algorithm makes only local choices, creating relatively small holes, while the second algorithm uses global information about the surface to create maximally-sized holes. These design choices produce algorithms with very different properties.

We begin by describing the envelope validity test used to determine whether a *candidate triangle* is valid for inclusion in the approximation surface. We then proceed to the two example simplification algorithms and a description of their relative merits.

5.1 Validity Test

A *candidate triangle* is one which we are considering for inclusion in an approximation to the input surface. Valid candidate triangles must lie between the two envelopes. Because we construct candidate triangles from the vertices of the original model, we know its vertices lie between the two envelopes. Therefore, it is sufficient to test the candidate triangle for intersections with the two envelope

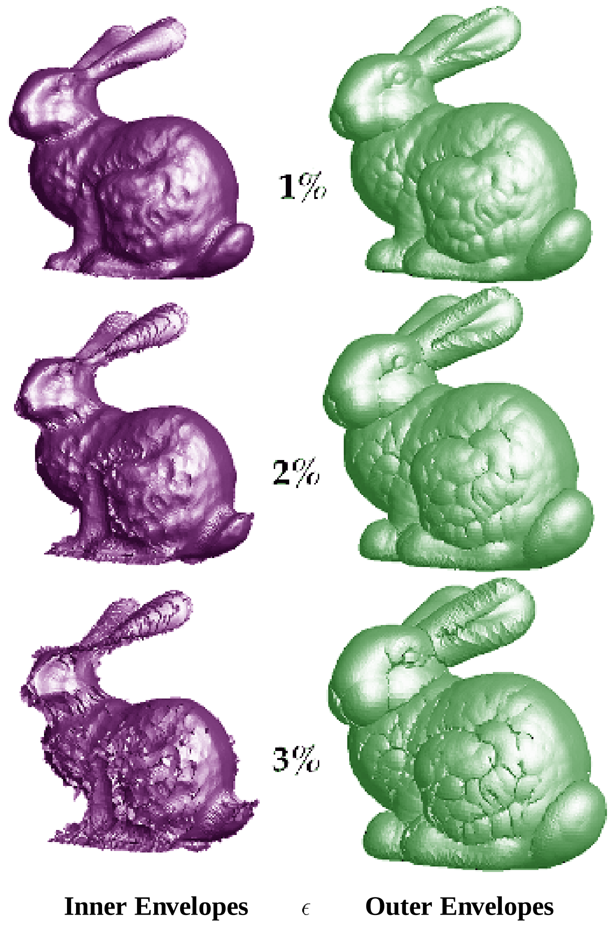


Figure 6: *Simplification envelopes for various* ϵ

surfaces. We can perform such tests efficiently using a space-partitioning data structure such as an octree.

A valid candidate triangle must also not cause a self-intersection in our surface, Therefore, it must not intersect any triangle of the current approximation surface.

5.2 Local Algorithm

To handle large models efficiently within the framework of simplification envelopes we construct a vertex-removal-based local algorithm. This algorithm draws heavily on the work of [21], [23], and [14]. Its main contributions are the use of envelopes to provide global error bounds as well as topology preservation and non-self-intersection. We have also explored the use of a more exhaustive hole-filling approach than any previous work we have seen.

The local algorithm begins by placing all vertices in a queue for removal processing. For each vertex in the queue, we attempt to remove it by creating a hole (removing the vertex's adjacent triangles) and attempting to fill it. If we can successfully fill the hole, the mesh modification is accepted, the vertex is removed from the queue, and its neighbors are placed back in the queue. If not, the vertex is removed from the queue and the mesh remains unchanged. This process terminates when the global error bounds eventually prevent the removal of any more vertices. Once the vertex queue is empty we have our simplified mesh.

For a given vertex, we first create a hole by removing all adjacent triangles. We begin the hole-filling process by generating all possible triangles formed by combinations of the vertices on the hole boundary. This is not strictly necessary, but it allows us to use a greedy strategy to favor triangles with nice aspect ratios. We fill the hole by choosing a triangle, testing its validity, and recursively filling the three (or fewer) smaller holes created by adding that triangle into the hole (see figure 7). If a hole cannot be filled at any level of the recursion, the entire hole filling attempt is considered a failure. Note that this is a single-pass hole filling strategy; we do not backtrack or undo selection of a triangle chosen for filling a hole. Thus, this approach does not guarantee that if a triangulation of a hole exists we will find it. However, it is quite fast and works very well in practice.

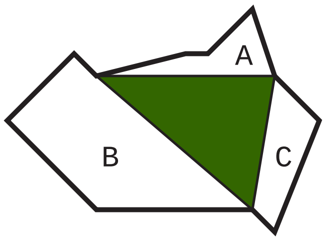


Figure 7: *Hole filling: adding a triangle into a hole creates up to three smaller holes*

We have compared the above approach with an exhaustive approach in which we tried all possible hole-filling triangulations. For simplifications resulting in the removal of hundreds of vertices (like highly oversampled laser-scanned models), the exhaustive pass yielded only a small improvement over the single-pass heuristic. This sort of confirmation reassures us that the single-pass heuristic works well in practice.

5.3 Global Algorithm

This algorithm extends the algorithm presented in [3] to non-convex surfaces. Our major contribution is the use of simplification envelopes to bound the error on a non-convex polygonal surface and the use of fundamental prisms to provide a generalized projection mechanism for testing for regions of multiple covering (overlaps). We present only a sketch of the algorithm here ; see [24] for the full details.

We begin by generating all possible candidate triangles for our approximation surface. These triangles are all 3-tuples of the input vertices which do not intersect either of the offset surfaces. Next we determine how many vertices each triangle *covers*. We rank the candidate triangles in order of decreasing covering.

We then choose from this list of candidate triangles in a greedy fashion. For each triangle we choose, we create a large hole in the current approximation surface, removing all triangles which *overlap* this candidate triangle. Now we begin the recursive hole-filling process by placing this candidate triangle into the hole and filling all the subholes with other triangles, if possible. One further restriction in this process is that the candidate triangle we are testing should not overlap any of the candidate triangles we have previously accepted.

5.4 Algorithm Comparison

The local simplification algorithm is fast and robust enough to be applied to large models. The global strategy is theoretically elegant. While the global algorithm works well for small models, its complexity rises at least quadratically,

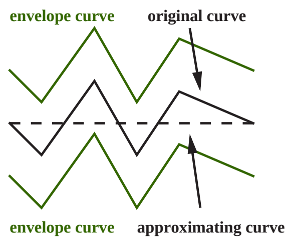


Figure 8: *Curve at local minimum of approximation*

making it prohibitive for larger models. We can think of the simplification problem as an optimization problem as well. A purely local algorithm may get trapped in a local "minimum" of simplification, while an ideal global algorithm will avoid all such minima.

Figure 8 shows a two-dimensional example of a curve for which a local vertex removal algorithm might fail, but an algorithm that globally searches the solution space will succeed in finding a valid approximation. Any of the interior vertices we remove would cause a new edge to penetrate an envelope curve. But if we remove all of the interior vertices, the resulting edge is perfectly acceptable.

This observation motivates a wide range of algorithms of which our local and global examples are the two extremes. We can easily imagine an algorithm that chooses nearby groups of vertices to remove simultaneously rather than sequentially. This could potentially lead to increased speed and simplification performance. However, choosing such sets of vertices remains a challenging problem.

6 Additional Features

Envelope surfaces used in conjunction with simplification algorithms are powerful, general-purpose tools. As we will now describe, they implicitly preserve sharp edges and can be extended to deal with bordered surfaces and perform adaptive approximations.

6.1 Preserving Sharp Edges

One of the important properties in any approximation scheme is the way it preserves any sharp edges or normal discontinuities present in the input model. Simplification envelopes deal gracefully with sharp edges (those with a small angle between their adjacent faces). When the ϵ tolerance is small, there is not enough room to simplify across these sharp edges, so they are automatically preserved. As the tolerance is increased, it will eventually be possible to simplify across the edges (which should no longer be visible from the appropriate distance). Notice that it is not necessary to explicitly recognize these sharp edges.

6.2 Bordered Surfaces

A bordered surface is one containing points that are homeomorphic to a half-disc. For polygonal models, this corresponds to edges that are adjacent to a single face rather than two faces. Depending on the context, we may naturally think of this as the boundary of some plane-like piece of a surface, or a hole in an otherwise closed surface.

The algorithms described in 5 are sufficient for closed triangle meshes, but they will not guarantee our global error bound for meshes with borders. While the envelopes constrain our error with respect to the normal direction of the surface, bordered surfaces require some additional constraints to hold the approximation border close to the original border. Without such constraints, the border of the approximation surface may "creep in," possibly shrinking the surface out of existence.

In many cases, the complexity of a surface's border curves may become a limiting factor in how much we can simplify the surface, so it is unacceptable to forgo simplifying these borders.

We construct a set of border tubes to constrain the error in deviation of the border curves. Each border is actually a cyclic polyline. Intuitively speaking, a border tube is a smooth, non-self-intersecting surface around one of these polylines. Removing a border vertex causes a pair of border edges to be replaced by a single border edge. If this new border edge does not intersect the relevant border tube, we may safely attempt to remove the border vertex.

To construct a tube we define a plane passing through each vertex of the polyline. We choose a coordinate system on this plane and use that to define a circular set of vertices. We connect these vertices for consecutive planes to construct our tube. Our initial tubes have a very narrow radius to minimize the likelihood of self-intersections. We then expand these narrow tubes using the same technique we used previously to construct our simplification envelopes.

The difficult task is to define a coordinate system at each polyline vertex which encourages smooth, non-self-intersecting tubes. The most obvious approach might be to use the idea of Frenet frames from differential geometry to define a set of coordinate systems for the polyline vertices. However, Frenet frames are meant for smooth curves. For a jagged polyline, a tube so constructed often has many self-intersections.

Instead, we use a projection method to minimize the twist between consecutive frames. Like the Frenet frame method, we place the plane at each vertex so that the normal to the plane approximates the tangent to the polyline. This is called the *normal plane*.

At the first vertex, we choose an arbitrary orthogonal pair of axes for our coordinate system in the normal plane. For subsequent vertices, we project the coordinate system from the previous normal plane onto the current normal frame. We then orthogonalize this projected coordinate system in the plane. For the normal plane of the final polyline vertex, we average the projected coordinate systems of the previous normal plane and the initial normal plane to minimize any twist in the final tube segment.

6.3 Adaptive Approximation

For certain classes of objects it is desirable to perform an adaptive approximation. For instance, consider large terrain datasets, models of spaceships, or submarines. One would like to have more detail near the observer and less detail further away. A possible solution could be to subdivide the model into various spatial cells and use a different ϵ-approximation for each cell. However, problems would arise at the boundaries of such cells where the ϵ-approximation for one cell, say at a value ϵ_1 need not necessarily be continuous with the ϵ-approximation for the neighboring cell, say at a different value ϵ_2.

Since all candidate triangles generated are constrained to lie within the two envelopes, manipulation of these envelopes provides one way to smoothly control the level of approximation. Thus, one could specify the ϵ at a given vertex to be a function of its distance from the observer — the larger the distance, the greater is the ϵ.

As another possibility, consider the case where certain features of a model are very important and are not to be approximated beyond a certain level. Such features might have human perception as a basis for their definition or they might have mathematical descriptions such as regions of high curvature. In either case, a user can vary the ϵ associated with a region to increase or decrease the level of approximation. The bunny in Figure 9 illustrates such an approximation.

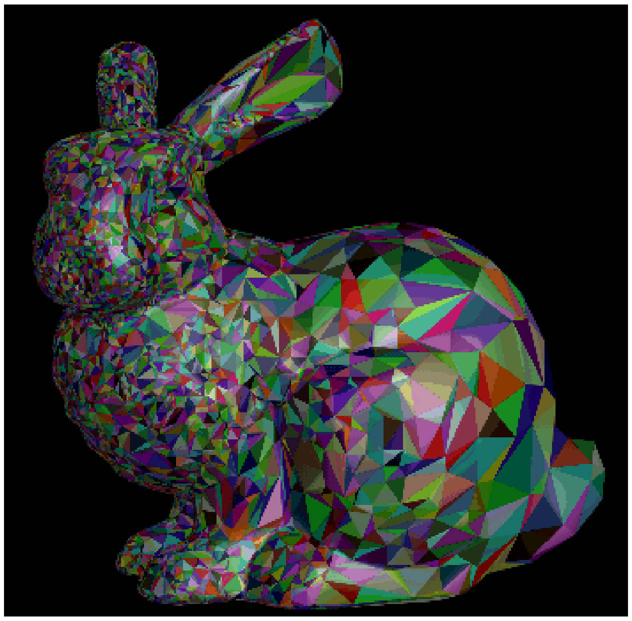


Figure 9: *An adaptive simplification for the bunny model. ϵ varies from 1/64% at the nose to 1% at the tail.*

7 Implementation and Results

We have implemented both algorithms and tried out the local algorithm on several thousand objects. We will first discuss some of the implementation issues and then present some results.

7.1 Implementation Issues

The first important implementation issue is what sort of input model to accept. We chose to accept only manifold triangle meshes (or bordered manifolds). This means that each edge is adjacent to two (one in the case of a border) triangles and that each vertex is surrounded by a single ring of triangles.

We also do not accept other forms of degenerate meshes. Many mesh degeneracies are not apparent on casual inspection, so we have implemented an automatic degeneracy detection program. This program detects non-manifold vertices, non-manifold edges, sliver triangles, coincident triangles, T-junctions, and intersecting triangles in a proposed input mesh. Note that correcting these degeneracies is more difficult than detecting them.

Robustness issues are important for implementations of any geometric algorithms. For instance, the analytical method for envelope computation involves the use of bilinear patches and the computation of intersection points. The computation of intersection points, even for linear elements, is difficult to perform robustly. The numerical method for envelope computation is much more robust because it involves only linear elements. Furthermore, it requires an intersection test but not the calculation of intersection points. We perform all such intersection tests in a conservative manner, using fuzzy intersection tests that may report intersections even for some close but non-intersecting elements.

Another important issue is the use of a space-partitioning scheme to speed up intersection tests. We chose to use an octree because of its simplicity. Our current octree implementation deals only with the bounding boxes of the elements stored. This works well for models with triangles that are evenly sized and shaped. For CAD models, which may contain long, skinny, non-axis-aligned triangles, a simple octree does not always provide enough of a speedup, and it may be necessary to choose a more appropriate space-partitioning scheme.

7.2 Results

We have simplified a total of 2636 objects from the auxiliary machine room (AMR) of a submarine dataset, pictured in Figure 10 to test and validate our algorithm. We reproduce the timings and simplifications achieved by our implementation for the AMR and a few other models in Table 1. All simplifications were performed on a Hewlett-Packard 735/125 with 80 MB of main memory. Images of these simplifications appear in Figures 11 and 12. It is interesting to compare the results on the bunny and phone models with those of [7, 8]. For the same error bound, we are able to obtain much improved solutions.

We have automated the process which sets the ϵ value for each object by assigning it to be a percentage of the diagonal of its bounding box. We obtained the reductions presented in Table 1 for the AMR and Figures 11 and 12 by using this heuristic.

For the rotor and AMR models in the above results, the i^{th} level of detail was obtained by simplifying the $i-1^{th}$ level of detail. This causes to total ϵ to be the sum of all previous ϵ's, so choosing $\epsilon's$ of 1, 2, 4, and 8 percent results in total $\epsilon's$ of 1, 3, 7, and 15 percent. There are two advantages to this scheme:
(a) It allows one to proceed incrementally, taking advantage of the work done in previous simplifications.
(b) It builds a hierarchy of detail in which the vertices at the i^{th} level of detail are a subset of the vertices at the $i-1^{th}$ level of detail.

One of the advantages of the setting ϵ to a percent of the object size is that it provides an a way to automate the selection of switching points used to transition between the various representations. To eliminate visual artifacts, we switch to a more faithful representation of an object when ϵ projects to more than some user-specified number of pixels on the screen. This is a function of the ϵ for that approximation, the output display resolution, and the corresponding maximum tolerable visible error in pixels.

8 Future Work

There are still several areas to be explored in this research. We believe the most important of these to be the generation of correspondences between levels of detail and the moving of vertices within the envelope volume.

Bunny			Phone			Rotor			AMR		
ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time
0	69,451	N/A	0	165,936	N/A	0	4,735	N/A	0	436,402	N/A
1/64	44,621	9	1/64	43,537	31	1/8	2,146	3	1	195,446	171
1/32	23,581	10	1/32	12,364	35	1/4	1,514	2	3	143,728	61
1/16	10,793	11	1/16	4,891	38	3/4	1,266	2	7	110,090	61
1/8	4,838	11	1/8	2,201	32	1 3/4	850	1	15	87,476	68
1/4	2,204	11	1/4	1,032	35	3 3/4	716	1	31	75,434	84
1/2	1,004	11	1/2	544	33	7 3/4	688	1			
1	575	11	1	412	30	15 3/4	674	1			

Table 1: *Simplification ϵ's and run times in minutes*

8.1 Generating Correspondences

A true geometric hierarchy should contain not only representations of an object at various levels of detail, but also some correspondence information about the relationship between adjacent levels. These relationships are necessary for propagating local information from one level to the next. For instance, this information would be helpful for using the hierarchical geometric representation to perform radiosity calculations. It is also necessary for performing geometric interpolation between the models when using the levels of detail for rendering. Note that the envelope technique preserves silhouettes when rendering, so it is also a good candidate for alpha blending rather than geometric interpolation to smooth out transitions between levels of detail.

We can determine which elements of a higher level of detail surface are covered by an element of a lower level of detail representation by noting which fundamental prisms this element intersects. This is non-trivial only because of the bilinear patches that are the sides of a fundamental prism. We can approximate these patches by two or more triangles and then tetrahedralize each prism. Given this tetrahedralization of the envelope volume, it is possible to stab each edge of the lower level-of-detail model through the tetrahedrons to determine which ones they intersect, and thus which triangles are covered by each lower level-of-detail triangle.

8.2 Moving Vertices

The output mesh generated by either of the algorithms we have presented has the property that its set of vertices is a subset of the set of vertices of the original mesh. If we can afford to relax this constraint somewhat, we may be able to reduce the output size even further. If we allow the vertices to slide along their normal vectors, we should be able to simplify parts of the surface that might otherwise be impossible to simplify for some choices of epsilon. We are currently working on a goal-based approach to moving vertices within the envelope volume. For each vertex we want to remove, we slide its neighboring vertices along their normals to make them lie as closely as possible to a tangent plane of the original vertex. Intuitively, this should increase the likelihood of successfully removing the vertex. During this whole process, we must ensure that none of the neighboring triangles ever violates the envelopes. This approach should make it possible to simplify surfaces using smaller epsilons than previously possible. In fact, it may even enable us to use the original surface and a single envelope as our constraint surfaces rather than two envelopes. This is important for objects with areas of high maximal curvature, like thin cylinders.

9 Conclusion

We have outlined the notion of simplification envelopes and how they can be used for generation of multiresolution hierarchies for polygonal objects. Our approach guarantees non-self-intersecting approximations and allows the user to do adaptive approximations by simply editing the simplification envelopes (either manually or automatically) in the regions of interest. It allows for a global error tolerance, preservation of the input genus of the object, and preservation of sharp edges. Our approach requires only one user-specifiable parameter, allowing it to work on large collections of objects with no manual intervention if so desired. It is rotationally and translationally invariant, and can elegantly handle holes and bordered surfaces through the use of cylindrical tubes. Simplification envelopes are general enough to permit both simplification algorithms with good theoretical properties such as our global algorithm, as well as fast, practical, and robust implementations like our local algorithm. Additionally, envelopes permit easy generation of correspondences across several levels of detail.

10 Acknowledgements

Thanks to Greg Angelini, Jim Boudreaux, and Ken Fast at Electric Boat for the submarine model, Rich Riesenfeld and Elaine Cohen of the Alpha_1 group at the University of Utah for the rotor model, and the Stanford Computer Graphics Laboratory for the bunny and telephone models.Thanks to Carl Mueller, Marc Olano, and Bill Yakowenko for many useful suggestions, and to the rest of the UNC Simplification Group (Rui Bastos, Carl Erikson, Merlin Hughes, and David Luebke) for providing a great forum for discussing ideas. The funding for this work was provide by a Link Foundation Fellowship, Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, ARO MURI grant DAAH04-96-1-0013, NSF Grant CCR-9319957, NSF Grant CCR-9301259, NSF Career Award CCR-9502239, ONR Contract N00014-94-1-0738, ARPA Contract DABT63-93-C-0048, NSF/ARPA Center for Computer Graphics and Scientific Visualization, NIH Grant RR02170, an NYI award with matching funds from Xerox Corp, and a U.S.-Israeli Binational Science Foundation grant.

References

[1] P. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proceedings Fifth Symposium on Discrete Algorithms*, pages 24–33, 1994.

[2] H. Brönnimann and M. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proceedings Tenth ACM Symposium on Computational Geometry*, pages 293–302, 1994.

[3] K. L. Clarkson. Algorithms for polytope covering and approximation. In *Proc. 3rd Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science, 1993.

[4] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[5] G. Das and D. Joseph. The complexity of minimum convex nested polyhedra. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 296–301, 1990.

[6] M. J. DeHaemer, Jr. and M. J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers & Graphics*, 15(2):175–184, 1991.

[7] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics: Proceedings of SIGGRAPH'95*, pages 173–182, 1995.

[9] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[10] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics: Proceedings of SIGGRAPH 1993*, pages 231–238. ACM SIGGRAPH, 1993.

[11] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 296–303, 1995.

[12] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface*, 1994.

[13] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[14] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[15] A. D. Kalvin and R. H. Taylor. Superfaces: Polyhedral approximation with bounded error. Technical Report RC 19135 (#82286), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10958, 1993.

[16] J. Mitchell and S. Suri. Separation and approximation of polyhedral surfaces. In *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 296–306, 1992.

[17] Kevin J. Renze and J. H. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. In *Proceedings of SIVE'95*, 1995.

[18] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[19] H. E. Rushmeier, C. Patterson, and A. Veerasamy. Geometric simplification for indirect illumination calculations. In *Proceedings Graphics Interface '93*, pages 227–236, 1993.

[20] F. J. Schmitt, B. A. Barsky, and W. Du. An adaptive subdivision method for surface-fitting from sampled data. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):179–188, 1986.

[21] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[22] G. Taubin. A signal processing approach to fair surface design. In *Proc. of ACM Siggraph*, pages 351–358, 1995.

[23] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[24] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

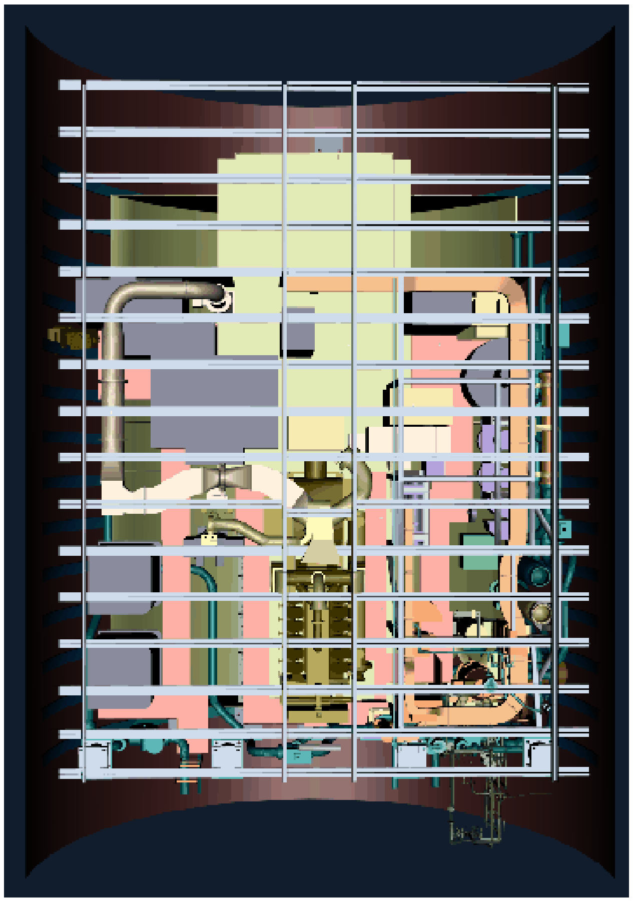


Figure 10: *Looking down into the auxiliary machine room (AMR) of a submarine model. This model contains nearly 3,000 objects, for a total of over half a million triangles. We have simplified over 2,600 of these objects, for a total of over 430,000 triangles.*

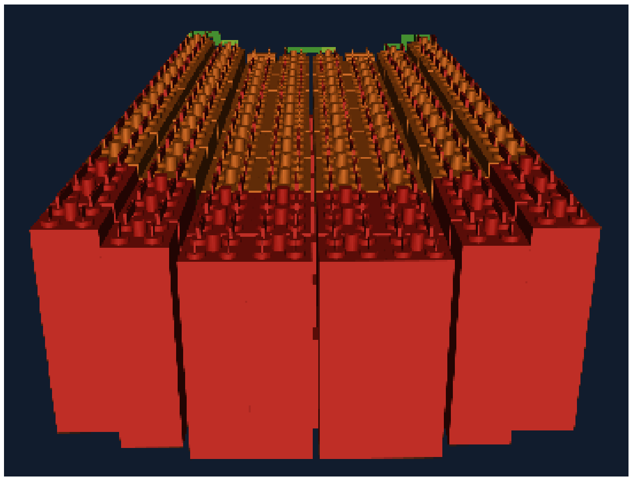
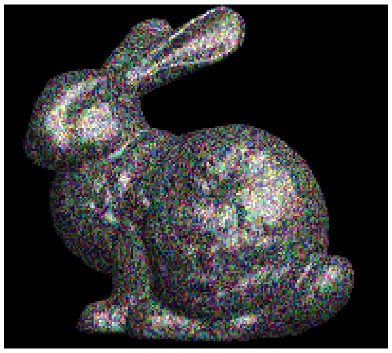
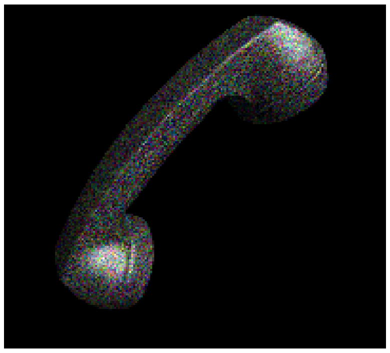
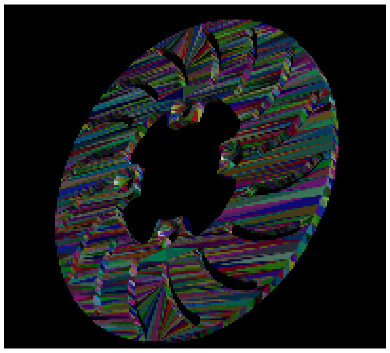
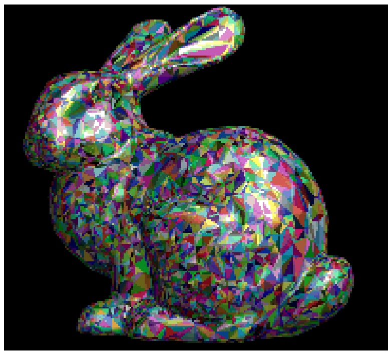


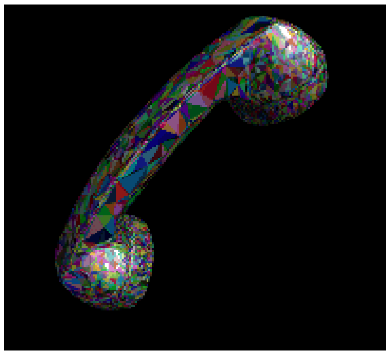
Figure 11: *An array of batteries from the AMR. All parts but the red are simplified representations. At full resolution, this array requires 87,000 triangles. At this distance, allowing 4 pixels of error in screen space, we have reduced it to 45,000 triangles.*

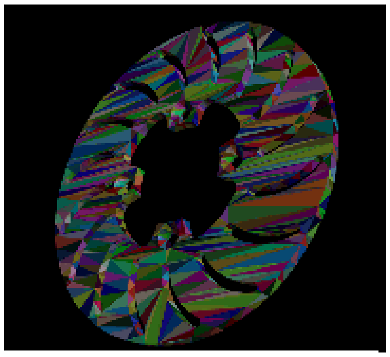

(a) bunny model: 69,451 triangles

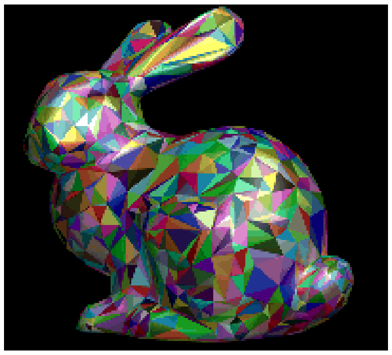

(e) phone model: 165,936 triangles

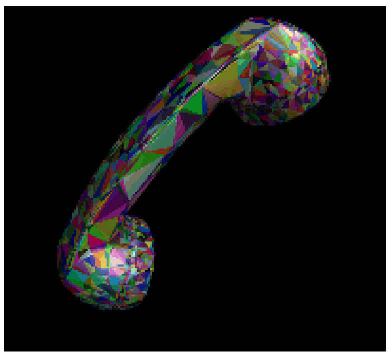

(i) rotor model: 4,736 triangles

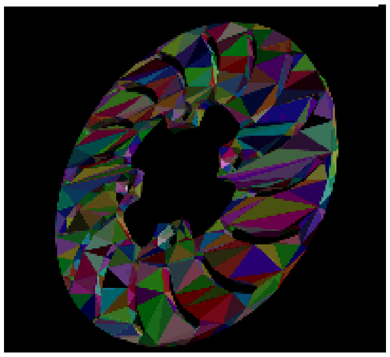

(b) $\epsilon = 1/16\%$, $10,793$ triangles

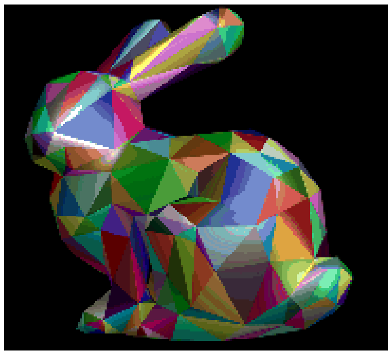

(f) $\epsilon = 1/32\%$, $12,364$ triangles

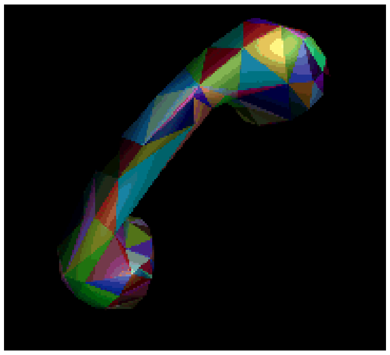

(j) $\epsilon = 1/8\%$, $2,146$ triangles


(c) $\epsilon = 1/4\%$, $2,204$ triangles


(g) $\epsilon = 1/16\%$, $4,891$ triangles


(k) $\epsilon = 3/4\%$, $1,266$ triangles


(d) $\epsilon = 1\%$, 575 triangles


(h) $\epsilon = 1\%$, 412 triangles

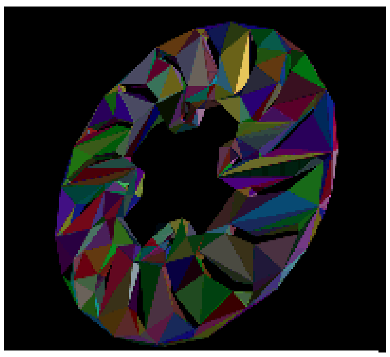

(l) $\epsilon = 3\ 3/4\%$, 716 triangles

Figure 12: *Level-of-detail hierarchies for three models. The approximation distance, ϵ, is taken as a percentage of the bounding box diagonal.*