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.

0-8186-8262-0/97 $10.00 Copyright 1997 IEEE.

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.

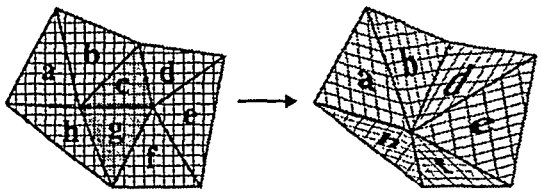

Figure 1: *The natural mapping primarily maps triangles to triangles. The two grey triangles map to edges, and the collapsed edge maps to the generated vertex*

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

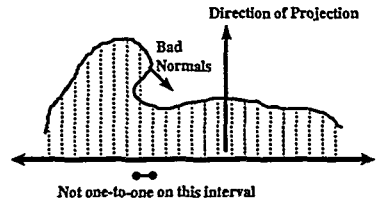

Figure 2: *A 2D example of an invalid projection*

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°, 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° 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

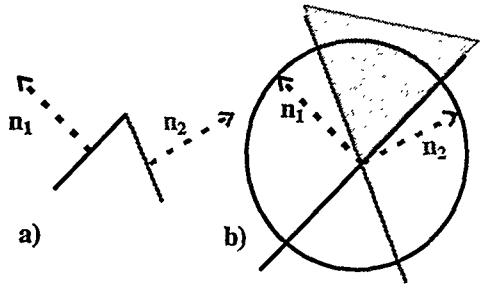

Figure 3: *A 2D example of the valid projection space. a) Two line segments and their normals. b) The 2D Gaussian circle, the planes corresponding to each segment, and the space of valid projection directions.*

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 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.

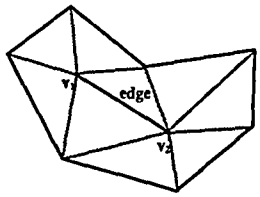

Figure 4: *The neighborhood of an edge as projected into 2D*

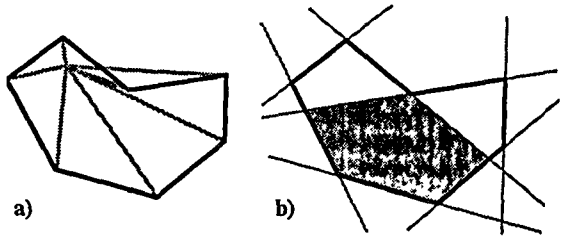

Figure 5: *a) An invalid 2D vertex position. b) The kernel of a polygon is the set of valid positions for a single, interior vertex to be placed. It is the intersection of a set of inward half-spaces.*

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.

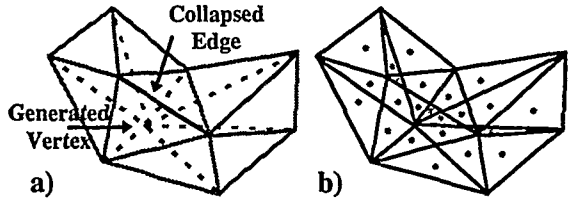

Figure 6: *a) Edge neighborhood and generated vertex neighborhood superimposed. b) A mapping in the plane, composed of 25 polygonal cells (each cell contains a dot). Each cell maps between a pair of planar elements in 3D.*

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:

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

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.*

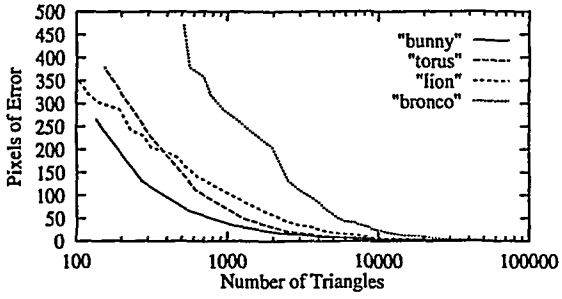

Figure 7: *Continuum of levels of detail for four models*

of data was measured with the object centered in the foreground of a 1000x1000-pixel viewport, with a 45° 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.