# 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

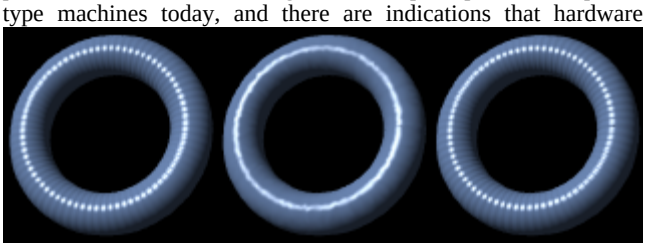

Figure 1: Bumpy Torus Model. *Left*: 44,252 triangles full resolution mesh. *Middle and Right*: 5,531 triangles, 0.25 mm maximum image deviation. *Middle*: per-vertex normals. *Right*: normal maps

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].

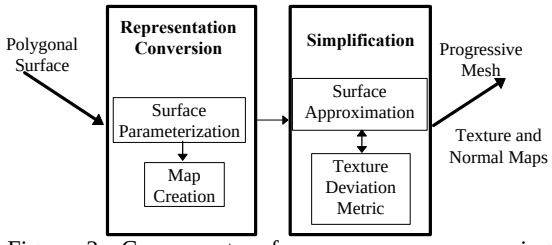


Figure 2: Components of an appearance-preserving simplification system.

# 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

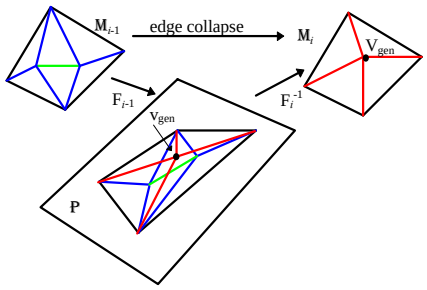


Figure 3: A look at the *i*th edge collapse. Computing $\mathbf{V}_{gen}$ determines the shape of the new mesh, $\mathbb{M}_i$. Computing $\mathbf{v}_{gen}$ determines the new mapping $\mathbf{F}_i$, to the texture plane, $\mathbb{P}$.

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.

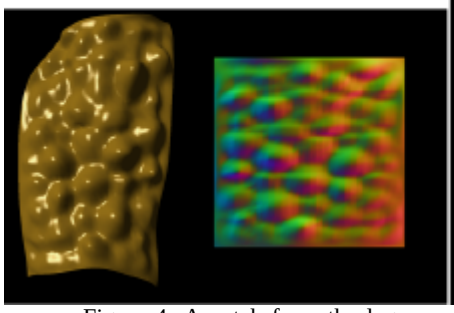

Figure 4: A patch from the leg of an armadillo model and its associated normal map.

Figure 5: Lion model.

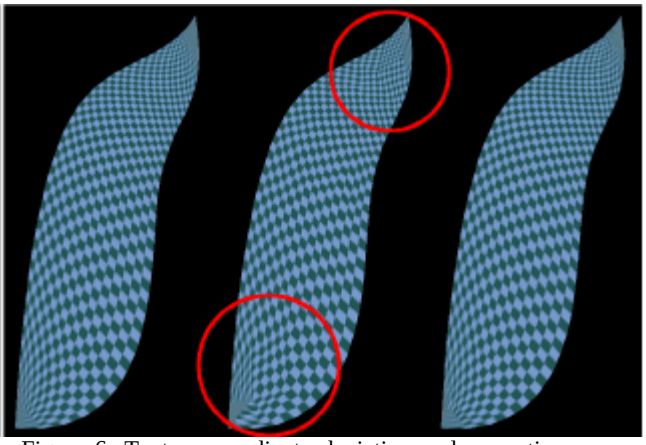

Figure 6: Texture coordinate deviation and correction on the lion's tail. *Left*: 1,740 triangles full resolution. *Middle and Right*: 0.25 mm maximum image deviation. *Middle*: 108 triangles, no texture deviation metric. *Right*: 434 triangles with texture metric.

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

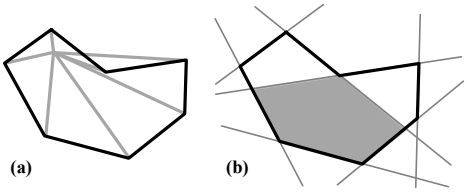


Figure 7: (a) An invalid choice for $\mathbf{v}_{gen}$ in $\mathbf{P}$, causing the new triangles extend outside the polygon. (b) Valid choices must lie in the shaded *kernel*.

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:

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

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:

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

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.

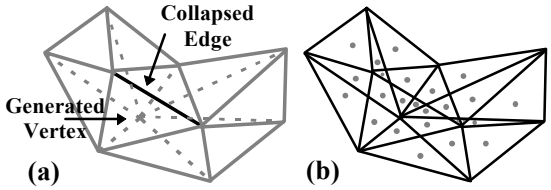


Figure 8: (a) An overlay in $\mathbb{P}$ determines the mapping between $\mathbb{M}_{i-1}$ and $\mathbb{M}_i$. (b) A set of polygonal *mapping cells*, each containing a dot.

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 $\mathbf{V}_{gen}$ and $\mathbf{v}_{gen}$.

## 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:

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

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, $\mathbf{v}_k$:

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

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

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

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, $\Delta_k$ in $\mathbb{M}_i$, 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.

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

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.

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

$\tilde{\mathrm{T}}_i(\mathbf{X})$ 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 $\mathbb{M}_{i+1}$ using an incremental computation (described in Figure 10). Figure 9(b) shows the propagation from $\mathbb{M}_i$ to $\mathbb{M}_{i+1}$. 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 $\mathbb{M}_i$ in triangle $k$, and $\mathbf{Y}$ is the point with the same texture coordinate on $\mathbb{M}_{i+1}$, the containment property of (7) holds:

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

For example, all three red dots Figure 9(b) have the same texture coordinates. The red point on $\mathbb{M}_o$ 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 $\mathbb{M}_i$ and $\mathbb{M}_{i+1}$ is linear, we compute the sizes of the boxes, $\boldsymbol{b}_{i+1,k'}$, by considering

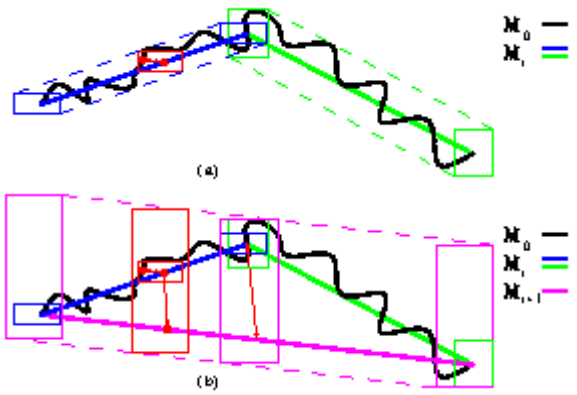

Figure 9: 2D illustration of the box approximation to total deviation error. a) A curve has been simplified to two segment, each with an associated box to bound the deviation. b) As we simplify one more step, the approximation is propagated to the newly created segment.

```
PropagateError():
foreach cell vertex, v
    foreach triangle, Told, in Mi-1 touching v
        foreach triangle, Tnew, in Mi touching v
            PropagateBox(v, Told, Tnew)

PropagateBox(v, Told, Tnew):
Pold = Fi-1^-1(v), Pnew = Fi^-1(v)
Enlarge Told.box so that Told.box applied at
    Pold contains Tnew.box applied at Pnew
```

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 | 62,480 triangles | 7,809 triangles | 975 triangles |
|---|---|---|---|
| 0.05 mm max image deviation | 0.25 mm max image deviation | 1.3 mm max image deviation | 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.