A Developer's Survey of Polygonal Simplification Algorithms

David P. Luebke

Department of Computer Science

University of Virginia

TYPE OF PAPER: **Tutorial**

SUBMITTED: **1-18-99**

ABSTRACT

Polygonal simplification, a.k.a. level of detail, is an important tool for anyone doing interactive rendering, but how is a developer to choose among the dozens of published algorithms? This article surveys the field from a developer's point of view, attempting to identify the issues in picking an algorithm, relate the strengths and weaknesses of different approaches, and describe a number of published algorithms as examples.

A Developer's Survey of Polygonal Simplification Algorithms

Polygonal simplification, a.k.a. level of detail, is an important tool for anyone doing interactive rendering, but how is a developer to choose among the dozens of published algorithms? This article surveys the field from a developer's point of view, attempting to identify the issues in picking an algorithm, relate the strengths and weaknesses of different approaches, and describe a number of published algorithms as examples.

1 Introduction

Polygonal models currently dominate interactive computer graphics. This is chiefly due to their mathematical simplicity: by providing a piecewise linear approximation to surface shape, polygonal models lend themselves to simple, regular rendering algorithms in which the visibility and color of pixels are determined by interpolating across the polygon's surface. Such algorithms embed well in hardware, which has in turn led to widely available polygon rendering accelerators for every platform. Unfortunately, the complexity of our polygonal models seems to grow faster than the ability of our graphics hardware to rendering them interactively. Put another way, the number of polygons we *want* to render always seems to exceed the number of polygons we can *afford* to render.

Polygonal simplification techniques [Figure 1] offer one solution for developers grappling with over-complex models. These methods seek to simplify the polygonal geometry of small, distant, or otherwise unimportant portions of the model to reduce the rendering cost without a significant loss in the visual content of the scene. This is at once a very current and a very old idea in computer graphics. As early as 1976 James Clark described the benefits of representing objects within a scene at several resolutions, and flight simulators have long used hand-crafted multi-resolution models of airplanes to guarantee a constant frame rate.[1,2] Recent years have seen a flurry of research into generating such models automatically. If you are considering using polygonal simplification to speed up your 3D application, this article should help you choose among the bewildering array of published algorithms.

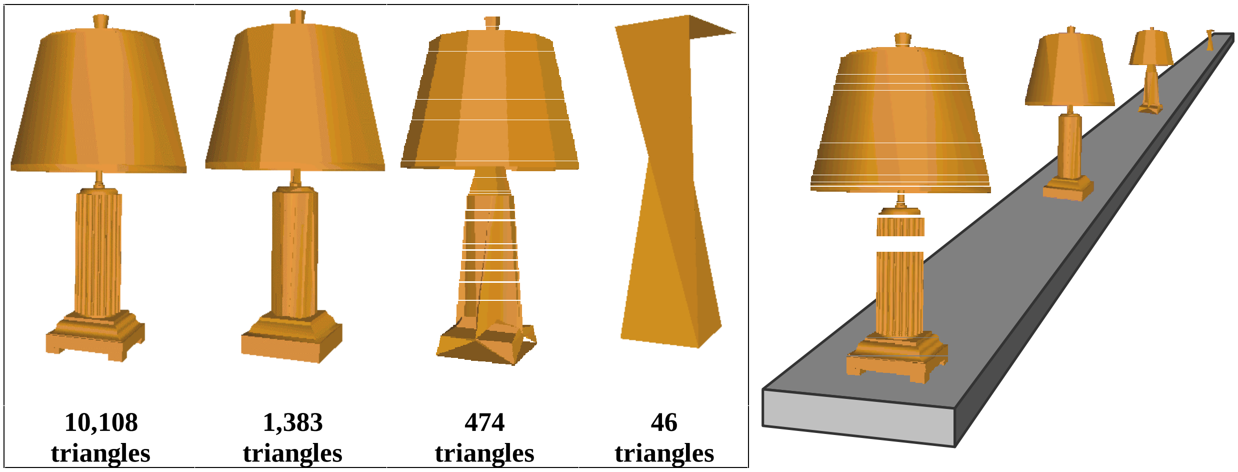


Figure 1: Managing model complexity by varying the level of detail used for rendering small or distant objects. Polygonal simplification methods can create multiple levels of detail such as these.

The first section asks some relevant questions; the next categorizes the various simplification algorithms and discusses the advantages and disadvantages of each category. Short descriptions of some example algorithms follow, along with pointers to additional detail and a few remarks on general issues facing the polygonal simplification field. The conclusion returns to the questions asked initially with some specific recommendations.

2 The First Questions

The first step in picking the right simplification algorithm is defining the problem. Ask yourself the following questions:

2.1 Why do I need to simplify polygonal objects?

What is your goal? Are you trying to **eliminate redundant geometry**? For example, the volumetric isosurfaces generated by the Marching Cubes algorithm[3] tile planar regions of the model with many small co-planar triangles. These triangles can be merged into larger polygons, often drastically decreasing the model complexity without introducing any geometric error. Similarly, a model may need to be subdivided for finite-element analysis; afterwards a simplification algorithm could remove unnecessary geometry.

Or are you trying to **reduce model size**, perhaps creating VRML models for a web site? Here the primary concern becomes optimizing bandwidth, which means minimizing model size. A simplification algorithm can take the original highly detailed model, whether created by CAD program, laser scanner, or other source, and reduce it to a bandwidth-friendly level of complexity.

Or are you trying to **improve run-time performance** by simplifying the polygonal scene being rendered? The most common use of polygonal simplification is to generate *levels of detail* (LODs) of the objects in a scene. By representing distant objects with a lower level of detail and nearby objects with a higher level of detail, applications from video games to CAD visualization packages can accelerate rendering and increase interactivity. Similar techniques can allow applications to manage the rendering complexity of flying over a large terrain databases. This leads naturally to the next important question:

2.2 What are my models like?

No algorithm excels at simplifying all categories of models. Some approaches are best suited to **curved, organic forms**, while others work best at preserving the sharp corners, flat faces, and regular curves of **mechanical objects**. Many models, such as radiositized scenes or scientific visualization datasets, have **precomputed colors or lighting** which must be taken into account. Some scenes, such as terrain datasets and volumetric isosurfaces from medical or scientific visualization, comprise a **few large, high-complexity individual objects**. The bad guys in a shoot-em-up video game, on the other hand, might consist of **multiple objects of moderate complexity**, mostly in isolation. As a final example, a CAD model of an automobile engine involves **large assemblies of many small objects**. Which simplification algorithm you choose depends on which of these descriptions applies to your models.

2.3 What matters to me most?

Ask yourself what you care about in a simplification algorithm. Do you need to preserve and regulate **geometric accuracy** in the simplified models? According to what criterion? Some algorithms control the Hausdorf distance of the simplified vertices to the original; others bound the volumetric deviation of the simplified mesh from the original. Some algorithms preserve the topological genus of the model; others attempt to reduce the genus in a controlled fashion.[4,5] Or do you simply want high **visual fidelity**? This unfortunately is much harder to pin down, since perception is much harder to quantify than geometry. Nonetheless, some algorithms empirically provide higher visual fidelity than others do, and at least one algorithm is able to bound, in pixels, the visual disparity between an object and its simplification.[6]

Is **preprocess time** an issue? For models containing thousands of parts and millions of polygons, creating LODs becomes a batch process that can take hours or days to complete. Depending on the application, such long preprocessing times may be a slight inconvenience or a fundamental handicap. In a design-review setting, for instance, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours prevent the rapid turnaround desirable in this scenario. On the other hand, when creating LODs for a video game or a product demonstration, it makes sense to take the time necessary to get the highest-quality simplifications.

If run-time performance is crucial, or your models are extremely complex, you may need an algorithm capable of **drastic simplification**. As the following sections explain, drastic simplification may require drastic measures such as view-dependent simplification and topology-reducing simplification. If you need to simplify many different models, a **completely automatic** algorithm may be most important. Or perhaps—let's face it—you just want something **simple to code**. Once you decide what matters to you most, you're ready to pick an algorithm.

3 Different Kinds of Algorithms, and Why You'd Want to Use Them

The computer graphics literature is replete with excellent simplification algorithms. Dozens of approaches have been proposed, each with strengths and weaknesses. This section attempts a useful taxonomy for comparing simplification algorithms by listing some important ways algorithms can differ or resemble each other, and describing what these mean to the developer.

3.1 Mechanism

Nearly every simplification technique in the literature uses some variation or combination of four basic polygon elision mechanisms: *sampling*, *adaptive subdivision*, *decimation*, and *vertex merging*. Although most surveys consider the underlying mechanism of primary importance when dividing up simplification algorithms, it hardly matters to a developer choosing which algorithm to use. Nonetheless, the various mechanisms may affect other features of an algorithm, and so are worth a few comments:

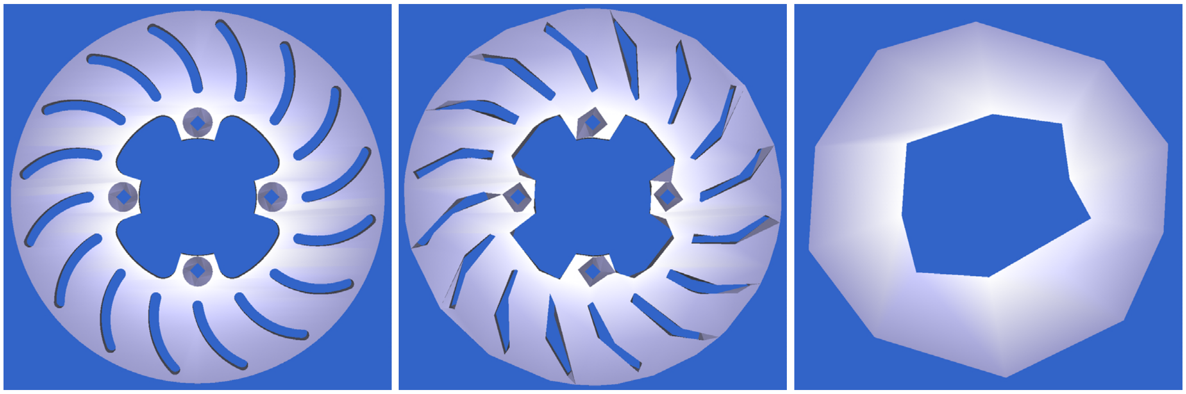
- **Sampling** algorithms sample the geometry of the initial model, either with points upon the model's surface or voxels superimposed on the model in a 3D grid. These are among the more elaborate (read: difficult to code) approaches. They may have trouble achieving high fidelity since high-frequency features are inherently difficult to sample accurately. These algorithms work best on smooth organic forms with no sharp corners.
- **Adaptive subdivision** algorithms find a simple *base mesh* that can be recursively subdivided to more and more closely approximate the initial model. This approach works best when the base model is easily found; for example, the base model for a terrain is typically a rectangle. Achieving high fidelity on general polygonal models requires creating a base model that captures important features of original model, which can be tricky (it is in fact a variation on polygonal simplification problem). Adaptive subdivision methods preserve the surface topology, which as the next section describes may limit their capacity for drastic simplification of complex objects, but they are well suited for multiresolution modeling, since changes made at low levels of subdivision propagate naturally to higher levels.
- **Decimation** techniques iteratively remove vertices or faces from the mesh, retriangulating the resulting hole after each step. These algorithms are relatively simple to code and can be very fast. Most use strictly local changes that tend to preserve genus, which again could restrict drastic simplification ability, but these algorithms excel at removing redundant geometry such as coplanar polygons.
- **Vertex merging** schemes operate by collapsing two or more vertices of a triangulated model together into a single vertex, which can in turn be merged with other vertices. Merging a triangle's corners eliminates it, decreasing the total polygon count. Vertex merging is a fairly simple and easy-to-code mechanism, but algorithms use techniques of varying sophistication to

determine which vertices to merge in what order. Accordingly, vertex merging algorithms range from simple, fast, and crude to complex, slow, and accurate. *Edge collapse* algorithms tend to preserve local topology, but algorithms permitting general vertex merge operations can modify topology and aggregate objects, enabling drastic simplification of complex objects and assemblies of objects. Most view-dependent algorithms are based on vertex merging.

3.2 Topology

The treatment of mesh topology during simplification provides an important distinction among algorithms. First, a few terms: in the context of polygonal simplification, *topology* refers to the structure of the connected polygonal mesh. The *local topology* of a face, edge, or vertex refers to the connectivity of that feature's immediate neighborhood. The mesh forms a *2-D manifold* if the local topology is everywhere homeomorphic to a disc, that is, if the neighborhood of every feature consists of a connected ring of polygons forming a single surface. In a triangulated mesh displaying manifold topology, every edge is shared by exactly two triangles, and every triangle has exactly three neighboring triangles. A *2-D manifold with boundary* also allows the local neighborhoods to be homeomorphic to a half-disc, meaning some edges can belong to only one triangle.

A **topology-preserving** simplification algorithm preserves manifold connectivity at every step. Such algorithms do not close holes in the mesh, and therefore preserve the overall genus of the simplified surface. Since no holes are appearing or disappearing during simplification, the fidelity of the simplified object tends to be relatively good. This constraint limits the simplification possible, however, since objects of high genus cannot be simplified below a certain number of polygons without closing holes in the model [Figure 2]. Moreover, a topology-preserving approach requires a mesh with valid topology to begin with. Some algorithms are **topology-tolerant**: they ignore regions in the mesh with non-manifold local topology, leaving those regions unsimplified. Other algorithms, faced with non-manifold regions, may simply crash.



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

Figure 2: Preserving genus limits drastic simplification. The original model of a brake rotor (a) is shown simplified with a topology-preserving algorithm (b) and a topology-modifying algorithm (c). Rotor model courtesy Alpha_1 Project, University of Utah

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

As a rule, topology-preserving algorithms work best when visual fidelity is crucial, or with an application such as finite-element analysis, in which surface topology can affect results. Preserving topology also simplifies some applications, such as multi-resolution modeling, which require a correspondence between high- and low-detail representations of an object. Real-time visualization of complex enough scenes, however, requires drastic simplification, and here topology-modifying algorithms have the edge. Either way, pick a topology-tolerant algorithm unless you are certain that your models will always have valid manifold topology. A surprising number of real-world models are full of T-intersections, three-way edges, mesh foldovers, and other topological degeneracies that will crash many algorithms.

3.3 Static, dynamic, and view-dependent algorithms

The traditional approach to accelerating rendering with polygonal simplification creates several discreet versions of each object in a preprocess, each at a different level of detail. At run-time the appropriate level of detail, or LOD, is chosen to represent the object. Since distant objects use much coarser LODs, the total number of polygons is reduced and rendering speed increased. Because LODs are computed offline during preprocessing, this approach can be called *static polygonal simplification*.

Static simplification has many advantages. Decoupling the simplification and rendering makes this the simplest model to program under. The simplification algorithm can generate LODs without regard to real-time rendering constraints and the run-time task of the rendering algorithm reduces to simply choosing which LODs to render. Furthermore, modern graphics hardware lends itself to the multiple model versions created by static simplification, since each LOD can be converted to triangle strips and compiled as a separate display list. Rendering such display lists will be much faster than simply rendering the unprocessed polygonal model using most graphics hardware.

Dynamic polygonal simplification is a relatively recent departure from the traditional static approach. Where a static simplification algorithm creates individual LODs in a preprocess, a dynamic simplification system creates a data structure from which any desired level of detail can be extracted *at run time*. A major advantage of this approach is better granularity: since the level of detail for each object is specified exactly rather than selected from a few pre-created options, no more polygons than necessary are used. This frees up more polygons for rendering other objects, which in turn use only as many polygons as needed for the desired level of detail. Better granularity thus leads to better use of resources and higher overall fidelity for a given polygon count. Dynamic simplification also supports progressive transmission of polygonal models, in which a base model is transmitted followed by a stream of refinements to be integrated dynamically.[7]

View-dependent simplification extends dynamic simplification, using view-dependent criteria to select the most appropriate level of detail *for the current view*. Thus in a view-dependent system a single object can span multiple levels of simplification, with nearby portions of the object shown at higher resolution than distant portions, or silhouette regions of the object at higher resolution than interior regions. By allocating polygons where they are most needed, view-dependent simplification offers the best distribution of this scarce resource yet.

Indeed, very complex models representing physically large objects often cannot be adequately simplified without view-dependent techniques. Terrain models are a classic example. Large terrain databases are well beyond the interactive rendering abilities of even high-end graphics hardware. Creating traditional LODs does not help: the viewpoint is typically quite close to part of the terrain and distant from other parts, so a high level of detail will provide good fidelity at unacceptable frame rates, while a low level of detail will provide good frame rates but terrible fidelity. Breaking up the terrain into smaller chunks, each comprising multiple LODs, addresses both problems but introduces discontinuities between chunks. These discontinuities become apparent as cracks when two adjacent chunks are represented at different levels of detail. A view-dependent simplification system, however, can use a high level of detail to represent the terrain near the viewpoint and a low level of detail for parts distant, with a smooth degradation of detail between.

To summarize, the static simplification approach of creating multiple LODs in a preprocess is simple and works best with most current graphics hardware. Dynamic simplification supports progressive transmission of polygonal models and provides better granularity, which in turn can provide better fidelity. View-dependent simplification can provide even better fidelity for a given polygon count, and can handle models (such as terrains) containing very complex individual objects that are physically large with respect to the viewer. An obvious disadvantage of view-dependent systems is the increased run-time load of choosing and extracting an appropriate level of detail. If the rendering system is CPU-bound, this additional load will decrease the frame rate, cutting into the speedup provided by regulating level of detail.

4 A Brief Catalog of Algorithms

Several published algorithms follow, each classified according to their underlying mechanism, how they treat topology, and whether they use static, dynamic, or view-dependent simplification. The intent of this section is not to provide an exhaustive list of work in the field of polygonal simplification, nor to select the "best" published papers, but rather to briefly describe a few important algorithms that span the taxonomy presented above. You may choose to implement one of the algorithms here (or download the publicly available code, if applicable), or you may choose another algorithm from the literature, or you may come up with your own. Hopefully this article will help you make an informed decision.

4.1 Triangle Mesh Decimation: Schroeder, Zarge, and Lorenson[8]

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

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

Simplifications produced by the decimation algorithm possess an interesting feature: the vertices of the simplified model are a subset of the vertices of the original model. This property is convenient for reusing normals and texture coordinates at the vertices, but it can limit the fidelity of the simplifications, since minimizing the geometric error introduced by the simplified approximation to the original surface can at times require changing the positions of the vertices.[9] The decimation algorithm, designed to reduce isosurfaces containing millions of polygons, is quite fast. It is also topology tolerant, accepting models with non-manifold vertices but not attempting to simplify around those vertices. Public-domain decimation code is available as part of the Visualization Tool Kit at *http://www.kitware.com/vtk.html*.

4.2 Vertex Clustering: Rossignac and Borrel[10], Low and Tan[11]

This vertex-merging algorithm, first proposed by Jarek Rossignac and Paul Borrel and later refined by Kok-Lim Low and Tiow-Seng Tan, is topology-insensitive, neither requiring nor preserving valid topology. The algorithm can therefore deal robustly with degenerate models with which other approaches have little or no success. The Rossignac-Borrel algorithm begins by assigning a perceptual importance to each vertex based upon two factors. Vertices attached to large faces and vertices of high curvature are considered more important than vertices attached to small faces and vertices of low curvature. Next a three-dimensional grid is overlaid on the model and all vertices within each cell of the grid are collapsed to the single most important vertex within the cell. The resolution of the grid determines the quality of the resulting simplification; a coarse grid will aggressively simplify the model whereas a fine grid will perform only minimal reduction. In the process of clustering, triangles whose corners are collapsed together become degenerate and disappear.

Low and Tan introduce a different clustering approach they call *floating-cell clustering*. In this approach the vertices are ranked by importance, and a cell of user-specified size is centered on the most important vertex. All vertices falling within the cell are collapsed to the representative vertex and degenerate triangles are filtered out as in the Rossignac-Borrel scheme. The most important remaining vertex becomes the center of the next cell, and the process is repeated. Eliminating the underlying grid greatly reduces the sensitivity of the simplification to the position and orientation of the model. Low and Tan also improve upon the criteria used for calculating vertex importance. Let θ be the maximum angle between all pairs of edges incident to a vertex. Rossignac and Borrel use $1/\theta$ to estimate the probability that the vertex lies on the silhouette, but Low and Tan argue that $\cos(\theta/2)$ provides a better estimate.

One unique feature of the Rossignac-Borrel algorithm is the fashion in which it treats triangles whose corners have merged. Reasoning that a triangle with two corners collapsed is simply a line and a triangle with three corners collapsed is simply a point, the authors choose to render such triangles using the line and point primitives of the graphics hardware, filtering out redundant lines and points. Thus a simplification of a polygonal object will in general be a collection of polygons, lines, and points. The resulting simplifications are therefore more accurate from a schematic than a strictly geometric standpoint. For the purposes of drastic simplification, however, the lines and points can contribute significantly to the recognizability of the object. Low and Tan extend the concept of drawing degenerate triangles as lines, calculating an approximate width for those lines based on the vertices being clustered, and drawing the line using the thick-line primitive present in most graphics systems. They further improve the line's appearance by giving it a normal to be shaded by the standard graphics lighting computations. This normal is dynamically assigned at run-time to give the line a cylinder-like appearance.

The original Rossignac-Borrel algorithm, which clusters vertices to a 3-D grid, is extremely fast; the Low-Tan variation is also quite fast. However, the methods suffer some disadvantages. Since topology is not preserved and no explicit error bounds with respect to the surface are guaranteed, the resulting simplifications are often less pleasing visually than those of slower algorithms. Also, it is difficult to specify the output of these algorithms, since the only way to predict how many triangles an LOD will have using a specified grid resolution is to perform the simplification.

4.3 Multiresolution Analysis of Arbitrary Meshes: Eck et al.[12]

This adaptive subdivision algorithm uses a compact wavelet representation to guide a recursive subdivision process. Multiresolution analysis, or MRA, adds or subtracts wavelet coefficients to interpolate smoothly between levels of detail. This process is fast enough to do at run time, enabling dynamic simplification. By using enough wavelet coefficients, the algorithm can guarantee that the simplified surface lies within a user-specified distance of the original model.

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

This algorithm possesses the disadvantages of strict topology-preserving approaches: manifold topology is absolutely required in the input model, and the shape and genus of the original object limit the potential for drastic simplification. The fidelity of the resulting simplifications is quite high for smooth organic forms, but the algorithm is fundamentally a low-pass filtering approach and has difficulty capturing sharp features in the original model unless the features happen to fall along a division in the base mesh.[7]

4.4 Voxel-Based Object Simplification: He et al.[4]

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

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

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

4.5 Simplification Envelopes: Cohen et al.[13]

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

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

Once created, these envelopes can guide the simplification process. The algorithms described in the paper both take decimation approaches that iteratively remove triangles or vertices and re-triangulate the resulting holes. By keeping the simplified surface within the envelopes, these algorithms can guarantee that the simplified surface never deviates by more than ε from the original surface, and furthermore that the surface does not self-intersect. The resulting simplifications tend to have very good fidelity.

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

4.6 Appearance-Preserving Simplification: Cohen, Olano, and Manocha[6]

This rigorous approach takes the error-bounding approach of simplification envelopes a step further, providing the best guarantees on fidelity of any simplification algorithm. Fidelity is expressed in terms of maximum screenspace deviation: when rendered onto the screen, the appearance of the simplification should deviate from the appearance of the original by no more than a user-specified number of pixels. The authors identify three attributes that affect the appearance of the simplification:

- **Surface position**: the coordinates of the polygon vertices.
- **Surface color**: the color field across the polygonal mesh.
- **Surface curvature**: the field of normal vectors across the mesh.

Algorithms that guarantee a limit on the deviation of surface position (such as simplification envelopes) may nonetheless introduce errors in surface color and curvature that exceed that limit [Figure 3]. Appearance-preserving simplification *decouples* surface position from color and curvature by storing the latter in texture maps and normal maps, respectively (a normal map is similar to a bump map). The model then reduces to a simple polygonal mesh with texture coordinates, from which the simplification algorithm computes LODs. Surface position is thus filtered by the simplification process, while the color and curvature information are filtered by the graphics hardware at run time by mip-mapping the texture and normal maps.

The simplification process uses edge collapses, guided by a *texture deviation metric* that bounds the deviation of a mapped attribute value from its correct position on the original surface. This deviation metric is applied to both the texture and normal maps; edge collapses that cause surface color, normals, or position to shift by more than the maximum user-specified distance ε are disallowed. Of course, this requirement constrains the degree of simplification possible, making appearance-preserving simplification less suitable for drastic simplification.

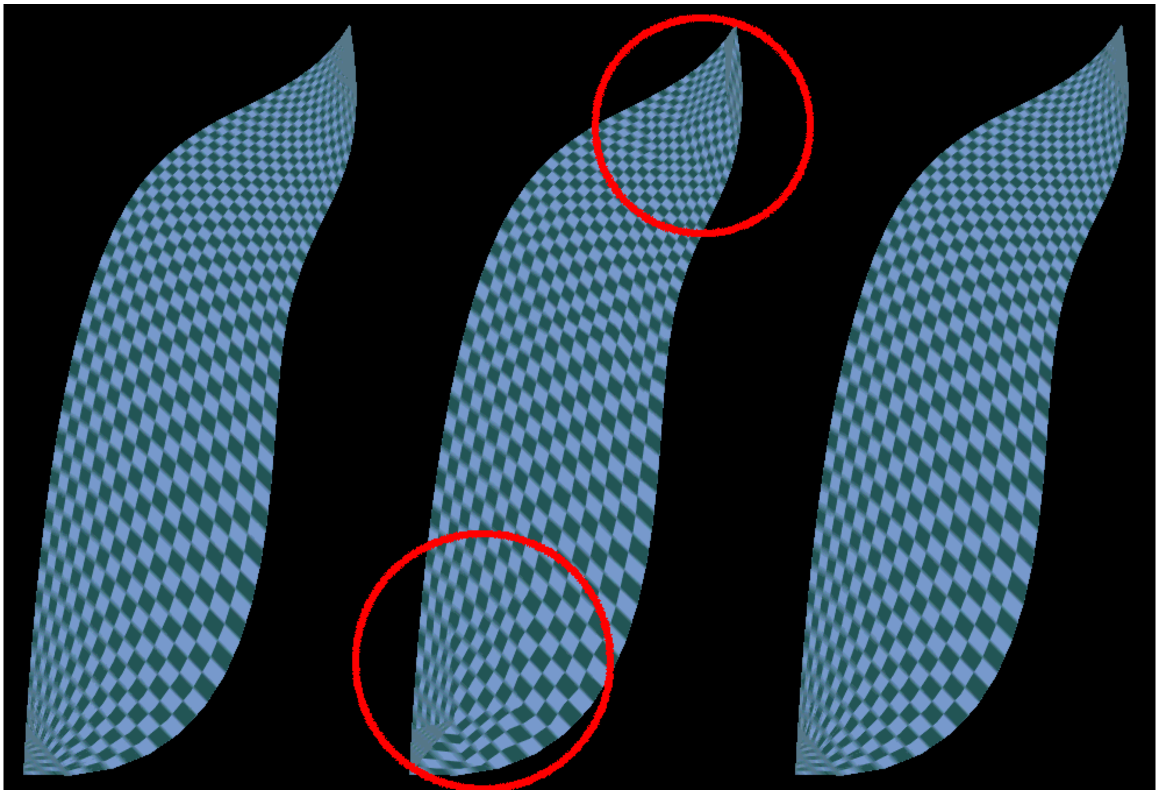


Figure 3: Texture coordinate deviation. The original model (left, 1740 polygons) has a checkerboard texture applied. Simplifying to a single pixel tolerance without taking texture deviation into effect (middle, 108 polygons) results in an accurate silhouette but noticeable texture distortion. Applying the texture deviation metric (right, 434 polygons) guarantees texture as well as silhouette correctness. Courtesy Jon Cohen, University of North Carolina at Chapel Hill.

While texture-mapping graphics hardware is commonplace, hardware support for normal- or bump-mapping is currently available only in a few prototype systems. Appearance-preserving simplification is thus most useful today on models that do not require dynamic lighting, such as radiositized datasets. In the future, as more sophisticated shading becomes available in hardware, appearance-preserving simplification may well become the standard for high-fidelity simplification.

4.7 Progressive Meshes: Hoppe[7,14]

The progressive mesh is a representation for polygonal models based on edge collapses. Hoppe introduced progressive meshes in a SIGGRAPH 96 paper as the first dynamic simplification algorithm for general polygonal manifolds, and followed up with a SIGGRAPH 97 paper extending them to support view-dependent simplification. A progressive mesh consists of a simple *base mesh*, created by a sequence of edge collapse operations, and a series of *vertex split* operations. A vertex split (*vsplit*) is the dual of an edge collapse (*ecol*). Each vsplit replaces a vertex by two edge-connected vertices, creating one additional vertex and two additional triangles. The vsplit operations in a progressive mesh correspond to the edge collapse operations used to create the base mesh. Applying every vsplit to the base mesh will recapture the original model exactly; applying a subset of the vsplits will create an intermediate simplification. In fact, the stream of vsplit records encodes a continuum of simplifications from the base mesh up to the original model. The vertex split and edge collapse operations are fast enough to be applied at run-time. Progressive meshes thus support dynamic simplification and (using the criteria for mesh refinement described below) view-dependent simplification.

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

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

Hoppe describes three view-dependent simplification criteria. A view frustum test aggressively simplifies regions of the mesh out of view, a backfacing test aggressively simplifies regions of the mesh not facing the viewer, and a screenspace error threshold guarantees that the geometric error in the simplified mesh is never greater than a user-specified screen-space tolerance. Since deviation tangent to the surface is measured separately from deviation perpendicular to the surface, silhouette preservation falls out of this error test naturally. Clever streamlining of the math involved makes these tests surprisingly efficient. Hoppe reports that evaluating all three criteria, which share several subexpressions, takes only 230 CPU cycles on average.[14]

The assumption of manifold topology is latent in the progressive mesh structure, which may be a disadvantage for some applications. Preserving topology prevents holes from closing and objects from aggregating, which can limit drastic simplification, and representing non-manifold models as a progressive mesh might present difficulties. Still, the progressive mesh representation provides a powerful and elegant framework for polygonal simplification. Any algorithm based on edge collapses can be used to generate a progressive mesh; Hoppe's energy-minimization approach produces high-fidelity simplifications but is relatively slow and seems somewhat intricate to code. Although the progressive mesh code is not publicly available, Hoppe has published a paper describing their efficient implementation in some detail.[15]

4.8 Hierarchical Dynamic Simplification: Luebke and Erikson[16]

This vertex-merging approach was another of the first to provide view-dependent simplification of arbitrary polygonal scenes. Hierarchical dynamic simplification (HDS) is similar in many ways to progressive meshes, with a hierarchy of vertex merges applied selectively at run time to effect view-dependent simplification. The approaches differ mostly in emphasis: progressive meshes emphasize fidelity and consistency of the mesh, whereas HDS emphasizes speed and robustness. Rather than representing the scene as a collection of objects, each at several levels of detail, in the HDS algorithm the entire model comprises a single large data structure. This structure is the *vertex tree*, a hierarchy of vertex clusters that is queried to generate a simplified scene.

The system is dynamic: for example, clusters to be collapsed or expanded can be chosen continuously based on their projected size. As the viewpoint shifts, the screenspace extent of some nodes in the vertex tree will fall below a size threshold. These nodes will be *folded* into their parent nodes, merging their vertices together and removing some now-degenerate triangles. Other nodes will increase in apparent size and will be *unfolded* into their constituent child nodes, introducing new vertices and new triangles. The user may adjust the screenspace size threshold for interactive control over the degree of simplification. HDS maintains an *active list* of visible polygons for rendering. Since frame-to-frame movements typically involve small changes in viewpoint, and therefore modify the active list by only a few polygons, the method takes advantage of temporal coherence for greater speed.

Any set of run-time criteria can be plugged into the HDS framework in the form of a function that folds and unfolds the appropriate nodes. Three such criteria have been demonstrated: a screenspace error threshold, a silhouette test, and a triangle budget. The screenspace error threshold, just described, monitors the projected extent of vertex clusters. The silhouette test uses a pre-calculated *cone of normals* to determine whether a vertex cluster is currently on the silhouette. Clusters on the silhouette are tested against a tighter screenspace threshold than clusters in the interior. Finally, HDS implements triangle-budget simplification by maintaining a priority queue of vertex clusters. The cluster with the largest screenspace error is unfolded and its children placed in the queue. This process is repeated until unfolding a cluster would violate the triangle budget.

Since construction of the vertex tree disregards triangle connectivity, HDS neither requires nor preserves manifold topology. Since the vertex tree spans the entire scene, objects may be merged as simplification proceeds. Together these properties make HDS topology tolerant and suitable for drastic simplification. The structures and methods of HDS are also probably the simplest among view-dependent algorithms to code. On the other hand, the fidelity of the resulting simplifications tends to be lower than the fidelity of more careful algorithms.

4.9 Quadric Error Metrics: Garland and Heckbert [9]

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

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

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extraordinarily fast, simplifying the 70,000 triangle Bunny model to 100 triangles in 15 seconds. The visual fidelity of the resulting simplifications is quite high, especially at high levels of simplification. Since disconnected vertices closer than *t* are allowed to merge, the algorithm does not require manifold topology. This allows holes to close and objects to merge, enabling more drastic simplification than topology-preserving schemes. One disadvantage of the algorithm is that the number of candidate vertex pairs, and hence the running time of the algorithm, approaches $O(n^2)$ as *t* approaches the size of the model. Moreover, the choice of a good value for *t* is very model-specific and seems difficult to automate for general models. Within these limitations, however, this simple-to-implement algorithm appears to be the best combination of efficiency, fidelity, and generality currently available for creating traditional view-independent LODs. The authors have kindly released their implementation as a free software package called *qslim*, at *http://www.cs.cmu.edu/afs/cs/user/garland/www/quadrics/qslim.html*.

5 Conclusion and Recommendations

Dozens of simplification algorithms have been published over the last few years, and dozens more have undoubtedly been whipped up by developers who were unaware of, or confused by, the bazaar of polygonal simplification techniques. Hopefully this article will help developers consider the right issues and pick an algorithm that best suits their needs.

The field of polygonal simplification seems to be approaching maturity. For example, researchers are converging on vertex merging as the underlying mechanism for polygon reduction. Hierarchical vertex-merging representations such as progressive meshes and the HDS vertex tree provide very general frameworks for experimenting with different simplification strategies, including view-dependent criteria. Settling on this emerging standard will hopefully allow the simplification field to make faster strides in other important issues, such as determining a common error metric. The lack of an agreed-upon definition of fidelity seriously hampers comparison of results among algorithms. Most simplification schemes use some sort of distance- or volume-based metric in which fidelity of the simplified surface is assumed to vary with the distance of that surface from the original mesh. Probably the most common use of polygonal simplification, however, is to speed up rendering for visualization of complex databases. For this purpose, the most important measure of fidelity is not geometric but perceptual: does the simplification *look* like the original?

Unfortunately, the human visual system remains imperfectly understood, and no well-defined perceptual metric exists to guide simplification. Current distance- and volume-based metrics, while certainly useful approximations, suffer one definite deficiency by not taking the surface normal into account. Since lighting calculations are usually interpolated across polygons, for example, deviation in a vertex's normal can be even more visually distracting than deviation in its position. As another example, consider a pleated polygonal sheet. A single polygon spanning the width of the sheet may have minimal distance error but can have very different reflectance properties. In an early survey of multiresolution rendering methods, Garland and Heckbert propose that fidelity of simplification methods should be measured with perceptual tests using human viewers, or with a good *image-based* error metric. As a starting point for such a metric, they suggest the sum of the squared distances in RGB color space between corresponding pixels.[17]

Table 1 closes with some informal recommendations for developers, organized according to the "First Questions" criteria presented above. Of course, these opinions are highly subjective; your mileage may vary.

Questions and answers	Recommendation
Why do I need to simplify polygonal objects?	
Eliminate redundant geometry	Decimation [4.1] excels at this
Reduce model size	For a one-shot application, use a high-fidelity algorithm like APS [4.6]
Improve run-time performance (by managing level of detail)	Depends, see below
What are my models like?	
Curved, organic forms	Decimation [4.1] works well
Mechanical objects	Progressive Meshes [4.7] for fidelity, Vertex Clustering [4.2] for speed and simplicity
Precomputed colors or lighting	APS [4.6] preserves fidelity best, with guaranteed bounds on deviation
A few high-complexity objects	A view-dependent algorithm such as HDS [4.8] or Progressive Meshes [4.7]
Multiple moderate-complexity objects	Use LODs. QEM [4.9] is the best overall algorithm for producing LODs
Large assemblies of many small objects	Merge objects into hierarchical assemblies using QEM [4.9] (specify objects to merge manually) or HDS [4.8] (does it automatically)
What matters to me most?	
Geometric accuracy	Use SE for manifold models, QEM otherwise [4.5]
Visual fidelity	APS [4.6] provides strong fidelity guarantees, but is limited on most current hardware
Preprocess time	QEM [4.9] provides high fidelity at high speed
Drastic simplification	QEM [4.9] if view-independent simplification suffices, otherwise HDS [4.8]
Completely automatic	HDS [4.8] works well for this
Simple to code	Use publicly available code if possible, otherwise code up Vertex Clustering [4.2]
APS – Appearance-Preserving Simplification	**QEM** – Quadric Error Metrics
HDS – Hierarchical Dynamic Simplification	**SE** – Simplification Envelopes

Table 1: Assorted recommendations to the questions of Section 2

6 References

1. Clark, James. “Hierarchical Geometric Models for Visible Surface Algorithms,” *Communications of the ACM*, Vol. 19, No 10, pp 547-554.
2. Cosman, M., and R. Schumacker. “System Strategies to Optimize CIG Image Content”. *Proceedings Image II Conference* (Scotsdale, Arizona), 1981.
3. Lorenson, William, and H. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, *Computer Graphics*, Vol. 21 (SIGGRAPH 87).
4. Taosong He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. “Voxel-Based Object Simplification”. *Proceedings Visualization 95*, IEEE Computer Society Press (Atlanta, GA), 1995, pp. 296-303.
5. El-Sana, Jihad, and A. Varshney. “Controlled Simplification of Genus for Polygonal Models”, *Proceedings Visualization 97*, IEEE Computer Society Press (Phoenix, AZ), 1997.
6. Cohen, Jon, and D. Manocha. “Appearance Preserving Simplification”, *Computer Graphics*, Vol. 32 (SIGGRAPH 98).
7. Hoppe, Hughes. “Progressive Meshes”, *Computer Graphics*, Vol. 30 (SIGGRAPH 96).
8. Schroeder, William, J. Zarge and W. Lorenson, “Decimation of Triangle Meshes”, *Computer Graphics*, Vol. 26 (SIGGRAPH 92).
9. Garland, Michael, and P. Heckbert, “Simplification Using Quadric Error Metrics”, *Computer Graphics*, Vol. 31 (SIGGRAPH 97).
10. Rossignac, Jarek, and P. Borrel. “Multi-Resolution 3D Approximations for Rendering Complex Scenes”, pp. 455-465 in *Geometric Modeling in Computer Graphics*, Springer-Verlag (1993), Genova, Italy. Also published as IBM Research Report RC17697 (77951) 2/19/92.
11. Low, Kok-Lim, and T.S. Tan. “Model Simplification Using Vertex Clustering”. In *1997 Symposium on Interactive 3D Graphics* (1995), ACM SIGGRAPH, pp. 75-82.
12. Eck, Matthias, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle. “Multiresolution Analysis of Arbitrary Meshes”, *Computer Graphics*, Vol. 29 (SIGGRAPH 95).
13. Cohen, Jon, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright. “Simplification Envelopes”, *Computer Graphics*, Vol. 30 (SIGGRAPH 96).
14. Hoppe, Hughes. “View-Dependent Refinement of Progressive Meshes”, *Computer Graphics*, Vol. 31 (SIGGRAPH 97).
15. Hoppe, Hugues. “Efficient implementation of progressive meshes”, *Computers & Graphics*, Vol. 22, No. 1, pages 27-36, 1998.
16. Luebke, David, and C. Erikson. “View-Dependent Simplification of Arbitrary Polygonal Environments”, *Computer Graphics*, Vol. 31 (SIGGRAPH 97).
17. Garland, Michael, and P. Heckbert. “Multiresolution Modeling for Fast Rendering”. *Proceedings of Graphics Interface ‘94* (1994).