A Developer's Survey of Polygonal Simplification Algorithms

David P. Luebke
University of Virginia

This article surveys polygonal simplification algorithms, identifies the issues in picking an algorithm, relates the strengths and weaknesses of different approaches, and describes several published algorithms.

Polygonal models currently dominate interactive computer graphics. This is chiefly because of their mathematical simplicity: polygonal models lend themselves to simple, regular rendering algorithms that embed well in hardware, which has in turn led to widely available polygon rendering accelerators for every platform. Unfortunately, the complexity of these models—measured by the number of polygons—seems to grow faster than the ability of our graphics hardware to render them interactively. Put another way, the number of polygons we want always seems to exceed the number of polygons we can afford.

Polygonal simplification techniques (see Figure 1) offer one solution for developers grappling with complex models. These methods simplify the polygonal geometry of small, distant, or otherwise unimportant portions of the model, seeking to reduce the rendering cost without a significant loss in the scene's visual content. 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,[1] and flight simulators have long used hand-crafted multiresolution airplane models to guarantee a constant frame rate. Recently, a flurry of research has targeted generating such models automatically. If you're considering using polygonal simplification to speed up your 3D application, this article should help you choose among the bewildering array of published algorithms.

The first questions

The first step in picking the right simplification algorithm is defining the problem. Ask yourself the following questions. (Note that Table 1 gives some informal and highly subjective recommendations for developers, organized according to the criteria I present in this section.)

Why do I need to simplify polygonal objects?

What's your goal? Are you trying to eliminate redundant geometry? For example, the volumetric isosurfaces generated by the marching cubes algorithm[10] tile the model's flat regions with many small, coplanar triangles. Merging these triangles into larger polygons can often decrease the model complexity drastically without introducing any geometric error. Similarly, you may need to subdivide a model for finite-element analysis; afterwards a simplification algorithm could remove unnecessary geometry.

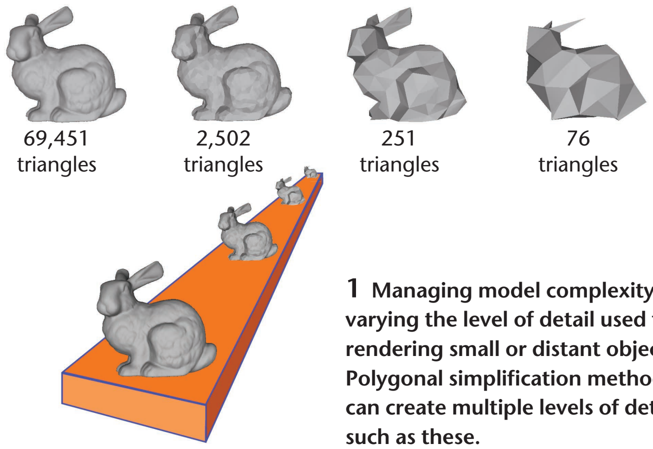
Or are you trying to reduce model size, perhaps creating downloadable models for a Web site? Here the primary concern becomes optimizing bandwidth, which means minimizing storage requirements. A simplification algorithm can take the original highly detailed model—whether created by a CAD program, laser scanner, or other source—and reduce it to a bandwidth-friendly level of complexity. If reducing the size required to store or transmit your 3D models is important, you should also investigate algorithms for geometric compression (see the "Further Reading" sidebar on p. 26).

Or are you trying to improve runtime 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 LOD and nearby objects with a higher LOD, applications from video games to CAD visualization packages can accelerate rendering and increase interactivity. Similar techniques let applications manage the rendering complexity of flying over a large terrain database. This leads naturally to the next important question.

0272-1716/01/$10.00 © 2001 IEEE

What are my models like?

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



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.

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 *Hausdorff distance* of the simplified vertices or surface to the original. (Informally, two point sets A and B are within Hausdorff distance *d* of each other if every point in A is within distance *d* of a point in B, and vice versa.) Other algorithms bound the volumetric deviation of the simplified mesh from the original. Some algorithms preserve the model's topological genus; others attempt to reduce the genus in a controlled fashion.[11,12]

Do you simply want high visual fidelity? This unfortunately is much harder to pin down; perception is more difficult to quantify than geometry. Nonetheless, some algorithms empirically provide higher visual fidelity than others do: one measures the simplification against rendered images[4] and another bounds, in pixels, the

Table 1. Assorted recommendations for "The First Questions" section.

Questions and Answers	Recommendation
Why do I need to simplify polygonal objects?	
Eliminate redundant geometry	Decimation[2] excels at this.
Reduce model size	For a one-shot application, use a high-fidelity algorithm like appearance-preserving simplification (APS)[3] or image-driven simplification (IDS).[4] Also consider geometry compression techniques (see the "Further Reading" sidebar).
Improve runtime performance (by managing levels of detail)	Depends, see below.
What are my models like?	
Complex organic forms	Decimation is often used, for example, for medical datasets, but quadric error metrics (QEM)[5] provide better fidelity at drastic rates.
Mechanical objects	Progressive meshes[6] for fidelity; vertex clustering for speed and simplicity.
Lots of textures or precomputed lighting	APS preserves fidelity best, with guaranteed bounds on deviation.
A few high-complexity objects	Use a view-dependent algorithm such as progressive meshes or hierarchical dynamic simplification (HDS).[7]
Multiple moderately complex objects	Use LODs. QEM is the best overall algorithm for producing LODs.
Large assemblies of many small objects	Merge objects into hierarchical assemblies using a topology-tolerant algorithm such as QEM or HDS.
So complex they don't fit in memory	Out-of-core simplification[8] is your best bet.
What matters to me most?	
Geometric accuracy	Use simplification envelopes (SE)[9] for manifold models, otherwise use QEM.
Visual fidelity	APS provides strong fidelity guarantees but is limited on most current hardware. IDS is driven by rendered images and has high visual fidelity.
Preprocess time	QEM provides high fidelity at high speed.
Drastic simplification	QEM if view-independent simplification suffices, otherwise use HDS.
Completely automatic	HDS works well for this.
Simple to code	Use publicly available code if possible, otherwise code up vertex clustering.

Further Reading

A number of excellent surveys review the field of polygonal simplification. For example, Cignoni et al.[1] supplied some comparative performance statistics, while Heckbert and Garland[2] and Puppo and Scopigno[3] summarized many of the methods not mentioned here. A tutorial by De Floriani et al.[4] provides a nice overview of LOD techniques in both surface and volume modeling.

Since polygonal simplification methods reduce the amount of geometry required to represent a model, they can are a form of geometry compression. Simplification provides lossy compression: simplified LODs require less memory to store and less bandwidth to deliver across a network, but at the cost of lower fidelity. Since 1995, researchers have proposed many geometry-compression techniques—both lossy and lossless—and the state of the art is rapidly advancing. Developers interested in reducing their 3D models' storage or bandwidth requirements may wish to investigate this burgeoning field. A full survey lies beyond the scope of this article, but a brief list of important papers follows (for more detail, see Taubin and Rossignac's excellent overview[5]):

- *Geometry compression.*[6] This seminal paper introduced the generalized triangle mesh representation, which caches the most recent *n* vertices for reference by triangles. The algorithm also optimizes the encoding of normals using a table-based approach and applies standard compression techniques to colors and coordinates.
- *Geometric compression through topological surgery*.[7] This work focuses on compressing connectivity and coordinate information using a vertex spanning tree. It forms the basis for 3D geometry compression in the MPEG-4 standard.
- *Progressive forest split compression.*[8] This algorithm combines aspects of progressive meshes and topological surgery, providing a highly efficient encoding of models that may be transmitted progressively across a network. Each stage of decompression doubles the number of vertices by splitting each vertex into two and stitching together the split regions according to an encoded triangulation.
- *Edgebreaker.*[9] This algorithm uses a finite-state machine to traverse and label triangles in a manifold mesh, compressing the triangle connectivity of zero-genus (that is, no holes) objects to less than 2 bits per triangle.
- *Progressive geometry compression.*[10] This recent and sophisticated progressive coding algorithm uses semiregular meshes, wavelet transforms, and zero-tree coding to achieve extremely high compression rates with excellent visual fidelity. The algorithm, designed for densely sampled meshes produced by geometry scanning, doesn't attempt to recreate the connectivity or vertex locations of the original mesh, focusing instead on the underlying surface's shape.

References

1. P. Cignoni, C. Montani, and R. Scopigno, "A Comparison of Mesh Simplification Algorithms," *Computers & Graphics*, vol. 22, no. 1, 1998, pp. 37-54.
2. P. Heckbert and M. Garland, "Survey of Polygonal Surface Simplification Algorithms," *Siggraph 97 Course Notes,* no. 25, ACM Press, New York, 1997.
3. E. Puppo and R. Scopigno, *Simplification, LOD, and Multiresolution-Principles and Applications*, tech. report C97-12, National Research Council of Italy, Pisa, Italy, 1997.
4. L. DeFlorani, E. Puppo, and R. Scopigno, "Level-of-Detail in Surface and Volume Modeling," *IEEE Visualization 98 Tutorials,* no. 6, IEEE CS Press, Los Alamitos, Calif., 1998.
5. G. Taubin and J. Rossignac, "3D Geometry Compression," *Siggraph 99 Course Notes,* no. 21, ACM Press, New York, 1999.
6. M. Deering, "Geometry Compression," *Computer Graphics* (Proc. Siggraph 95), vol. 29, ACM Press, New York, 1995.
7. G. Taubin and J. Rossignac, "Geometric Compression through Topological Surgery," *ACM Trans. Graphics*, vol. 17, no. 2, 1998, pp. 84-115.
8. G. Taubin et al.,"Progressive Forest Split Compression," *Computer Graphics* (Proc. Siggraph 98), vol. 32, ACM Press, New York, 1998, pp. 123-132.
9. J. Rossignac, "Edgebreaker: Connectivity Compression for Triangle Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 5, no. 1, 1999, pp. 47-61.
10. A. Khodakovsky, P. Schroder, and W. Sweldens, "Progressive Geometry Compression," *Computer Graphics* (Proc. Siggraph 2000), vol. 34, ACM Press, New York, 2000, pp. 271-278.

visual disparity between an object and its simplification.[3]

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. Hours of preprocessing 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.

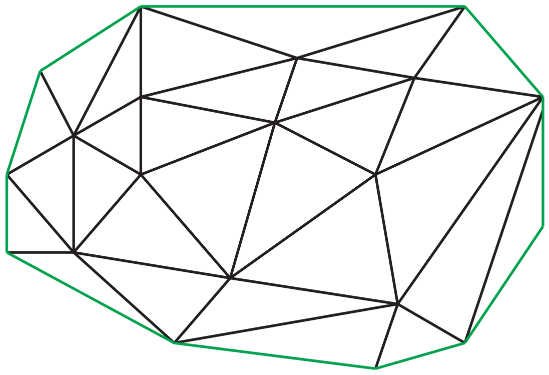
If runtime performance is crucial or your models are extremely complex, you may need an algorithm capable of drastic simplification. As the following sections explain, drastic simplification may require drastic measures such as view-dependent simplification and topology-reducing simplification. If you need to simplify many different models, a completely automatic algorithm may be most important. 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.

Different kinds of algorithms, and why you'd want to use them

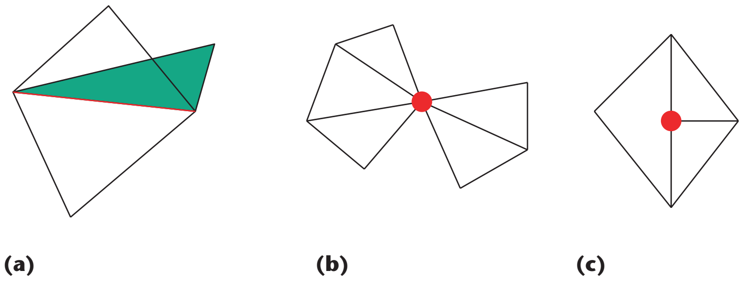
The computer graphics literature is replete with excellent simplification algorithms. Researchers have proposed dozens of approaches, each with strengths and weaknesses. Here, I attempt a useful taxonomy for simplification algorithms, listing some important ways algorithms can differ or resemble each other and describing what these mean to the developer.

Topology

The treatment of mesh topology during simplification provides an important distinction among algorithms. First, let me introduce a few terms. In the context of polygonal simplification, *topology* refers to the connected polygonal mesh's structure. The *genus* is the number of holes in the mesh surface. For example, a sphere and a cube have a genus of zero, while a doughnut and a coffee cup have a genus of one. The *local topology* of a face, edge, or vertex refers to the connectivity of that feature's immediate neighborhood. The mesh forms a *2D manifold* if the local topology is everywhere equivalent to a disc—that is, if the neighborhood of every feature consists of a connected ring of polygons forming a single surface (see Figure 2). In a triangulated mesh displaying manifold topology, exactly two triangles share every edge, and every triangle shares an edge with exactly three neighboring triangles. A *2D manifold with boundary* permits boundary edges, which belong to only one triangle.


2 A 2D manifold with a boundary (boundary edges in green). One or two triangles share each edge and a connected ring of triangles shares each vertex.

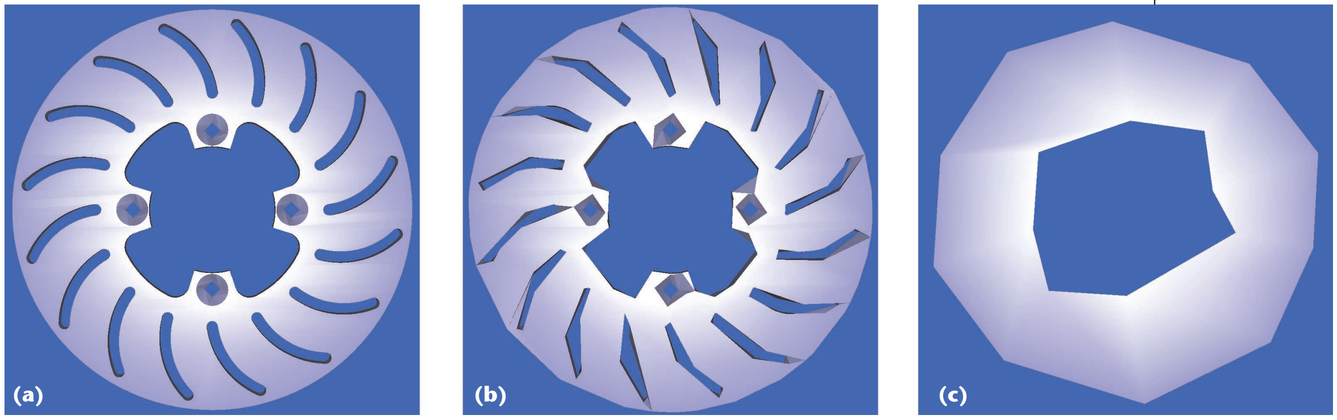
Manifold meshes result in well-behaved models. Virtually any simplification algorithm can successfully operate on any manifold object. Manifold meshes are also desirable for many other applications, such as finite-element analysis and radiosity. Some algorithms and modeling packages guarantee manifold output. For example, the marching-cubes algorithm constructs manifold volumetric isosurfaces. Unfortunately, in actual practice many models aren't perfectly manifold, with topological flaws such as cracks, T-junctions, and nonmanifold points or edges (see Figure 3). Such defects are particularly problematic in CAD, which by definition involves handmade models.



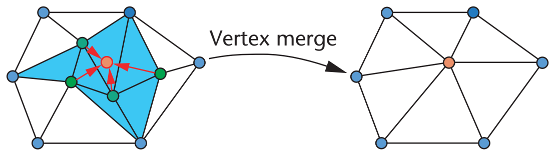
3 Examples of nonmanifold meshes: (a) An edge shared by three triangles, (b) a vertex shared by two otherwise unconnected sets of triangles, and (c) a T-vertex.

A *topology-preserving* simplification algorithm preserves manifold connectivity at every step. Such algorithms don't close holes in the mesh and therefore preserve the overall genus. Because no holes appear or disappear during simplification, the simplified object's visual fidelity tends to be relatively good. This constraint limits the simplification possible, however, since objects of a high genus can't be simplified below a certain number of polygons without closing holes in the model (see Figure 4). Moreover, a topology-preserving approach requires beginning with a mesh with manifold topology. Some algorithms are *topology tolerant*—they ignore regions in the mesh with nonmanifold local topology, leaving those regions unsimplified. Other algorithms, faced with nonmanifold regions, may simply fail.

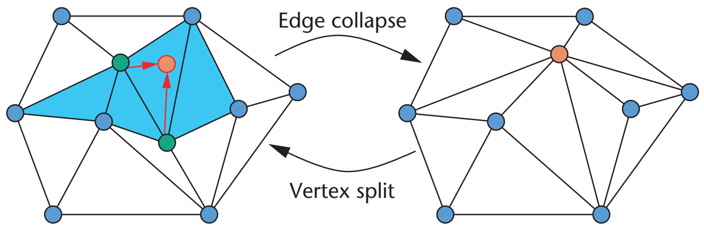
Topology-modifying algorithms don't necessarily preserve manifold topology. The algorithms can therefore close up holes in the model and aggregate separate objects into assemblies as simplification progresses, permitting drastic simplification beyond the scope of topology-preserving schemes. This drastic simplification



4 Preserving genus limits drastic simplification. The original model of a brake rotor with (a) 4,736 triangles and 21 holes is simplified with a topology-preserving algorithm using (b) 1,006 triangles and 21 holes and a topology-modifying algorithm with (c) 46 triangles and one hole. Model courtesy of the Alpha_1 Project, University of Utah.



5 **The vertex-merge operation clusters multiple vertices (green) together into a single representative vertex (orange), eliminating those triangles (aqua) whose corner vertices are merged.**



6 **The edge-collapse operation merges exactly two vertices that share an edge. This eliminates two triangles from the mesh (one if the edge lies on a boundary). A vertex split is the dual of an edge collapse, introducing two triangles.**

often comes at the price of poor visual fidelity, with distracting popping artifacts caused by holes appearing and disappearing from one LOD to the next. Some topology-modifying algorithms don't 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.

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 multiresolution surface editing, which require a correspondence between an object's high- and low-detail representations. Real-time visualization of complex scenes, however, requires drastic simplification and here topology-modifying algorithms have the edge. Either way, pick a topology-tolerant algorithm unless you're certain that your models will always have valid manifold topology.

Mechanism

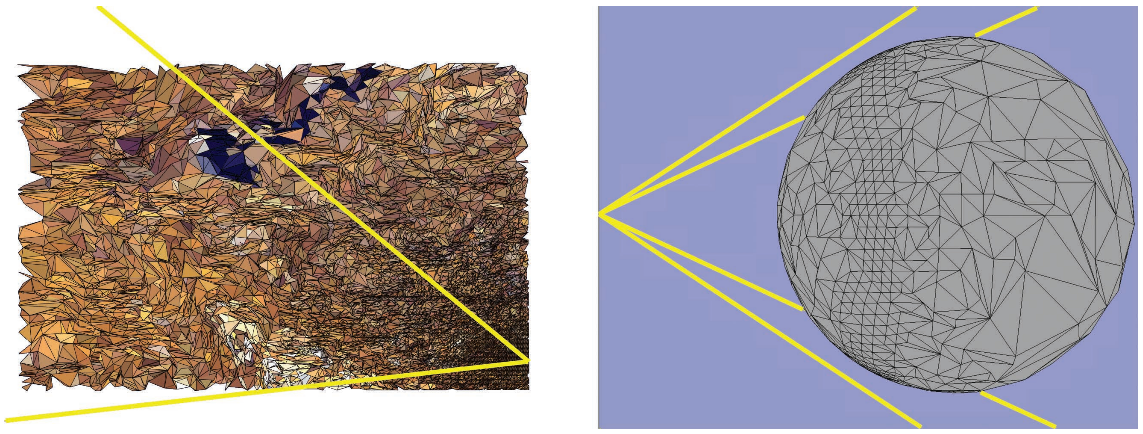
Nearly every simplification technique in the literature uses some variation or combination of four basic polygon removal mechanisms: sampling, adaptive subdivision, decimation, and vertex merging. Because the mechanism you use may affect an algorithm's characteristics, these are worth a few comments.

- Sampling algorithms sample the initial model's geometry, either with points on the model's surface or voxels superimposed on the model in a 3D grid. These are among the more elaborate and difficult to code approaches. They may have trouble achieving high fidelity since high-frequency features are inherently difficult to sample accurately. These algorithms usually work best on smooth organic forms with no sharp corners.
- Adaptive subdivision algorithms find a simple *base mesh* that can be recursively subdivided to 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 the original model's important features, which can be tricky. Adaptive subdivision methods preserve the surface topology, which may limit their capacity for drastic simplification. On the other hand, they suit multiresolution surface editing well, because 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 the 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 in turn can be merged with other vertices. Merging a triangle's corners eliminates it, decreasing the total polygon count (see Figure 5). Vertex merging is a 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 (see Figure 6), which always merge two vertices sharing an edge, 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

Static, dynamic, and view-dependent simplification

The traditional approach to accelerating rendering with polygonal simplification creates several discrete versions of each object in a preprocess, each at a different level of detail. At runtime, rendering algorithms choose the appropriate LOD to represent the object. Because distant objects use much coarser LODs, the total number of polygons is reduced and rendering speed increased. Because we compute LODs offline during preprocessing, this approach can be called *static polygonal simplification*.

Static simplification has many advantages. Decoupling simplification and rendering makes this the simplest model to program. The simplification algorithm generates LODs without regard to real-time rendering constraints, and the rendering algorithm simply chooses which LODs to render. Furthermore, modern graphics hardware lends itself to the multiple model versions created by static simplification, because each LOD can be converted during preprocessing to triangle strips and compiled as a separate display list. Rendering such dis-


7 **Two examples of view-dependent simplification, with the view frustum in yellow. A high-resolution terrain near the viewer is simplified aggressively as distance increases (left). A sphere is simplified aggressively in interior and backfacing regions, while high fidelity is preserved along the silhouette (right).**

play lists will usually be much faster than rendering the LODs as an unordered list of polygons.

Dynamic polygonal simplification departs from the traditional static approach. Whereas a static simplification algorithm creates individual LODs during the preprocessing stage, a dynamic simplification system creates a data structure encoding a continuous spectrum of detail. The desired LOD can be extracted from this structure at runtime. A major advantage of this approach is better granularity. Since the algorithm specifies the LOD for each object exactly, rather than choosing from a few precreated options, it uses no more polygons than necessary. This frees up more polygons for rendering other objects. 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.[6]

View-dependent simplification extends dynamic simplification by using view-dependent criteria to select the most appropriate LOD for the current view. In a view-dependent system, a single object can span multiple levels of simplification. For instance, nearby portions of the object may appear at a higher resolution than distant portions, or silhouette regions of the object may appear at a higher resolution than interior regions (see Figure 7). By allocating polygons where they're most needed, view-dependent simplification optimizes the distribution of this scarce resource.

Indeed, complex models representing physically large objects often can't 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, but creating traditional LODs doesn't help. The viewpoint is typically quite close to part of the terrain and distant from other parts, so a high LOD will provide good fidelity at unacceptable frame rates, while a low LOD 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 appear as cracks when two adjacent chunks are represented at different LODs. A view-dependent simplification system, however, can use a high LOD to represent the terrain near the viewpoint and a low LOD for parts distant, with a smooth degradation of detail between. Not surprisingly, early work on view-dependent simplification focused on terrains.[13]

The static simplification approach of creating multiple discrete LODs in a preprocess is simple and works best with most current graphics hardware. Dynamic simplification supports the 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 complex individual objects that are physically large with respect to the viewer. An obvious disadvantage of view-dependent systems is the increased runtime load of choosing and extracting an appropriate LOD. If the rendering system is CPU bound, this additional load will decrease the frame rate, cutting into the speedup provided by regulating LODs.

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. In some cases, I describe a family of algorithms, with reference to papers that improve or extend the original published algorithm. The intent of this section isn't to provide an exhaustive list of work in the field of polygonal simplification, nor to select the best papers, but to briefly describe a few important algorithms that span the taxonomy presented. You may choose to implement one of the algorithms here (or download the code, if available), choose another algorithm from the literature, or come up with your own. Hopefully, this article will help you make an informed decision.

The decimation algorithm, designed to reduce isosurfaces containing millions of polygons, is quite fast. It's also topology tolerant, accepting models with nonmanifold vertices but not attempting to simplify around those vertices.

Triangle mesh decimation

Schroeder, Zarge, and Lorenson published one of the first algorithms to simplify general polygonal models and coined the term *decimation* for iterative removal of vertices.[2] 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, the algorithm considers deleting each vertex. If the vertex can be removed without violating the neighborhood's local topology, and if the resulting surface would lie within a user-specified distance of the unsimplified geometry, the algorithm deletes the vertex and all its associated triangles. This leaves a hole in the mesh, which is then retriangulated. The algorithm continues to iterate over the vertices in the model until it can't remove any more vertices.

The vertices of a model simplified by the decimation algorithm are a subset of the original model's vertices. 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 require changing the vertices' positions.[5] The decimation algorithm, designed to reduce isosurfaces containing millions of polygons, is quite fast. It's also topology tolerant, accepting models with nonmanifold vertices but not attempting to simplify around those vertices. Schroeder, Zarge, and Lorenson have since developed a topology-modifying algorithm.[14] Public-domain decimation code is available as part of the Visualization Tool Kit at http://www.kitware.com/vtk.html.

Vertex clustering

This vertex-merging algorithm, first proposed by Rossignac and Borrel,[15] 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 an importance to each vertex. 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, the algorithm overlays a 3D grid on the model and collapses all vertices within each cell of the grid to the single most important vertex within the cell. The grid's resolution determines the quality of the resulting simplification; a coarse grid will aggressively simplify the model whereas a fine grid will perform only minimal reduction. During the clustering process, triangles whose corners are collapsed together become degenerate and disappear.

Low and Tan[16] introduced a different clustering approach called *floating-cell clustering*. This technique ranks the vertices by importance, and a cell of user-specified size is centered on the most important vertex. The algorithm collapses all vertices falling within the cell to the representative vertex and filters out degenerate triangles as in the Rossignac–Borrel scheme. The most important remaining vertex becomes the center of the next cell, and the process repeats. Eliminating the underlying grid greatly reduces the simplification's sensitivity to the model's position and orientation. Low and Tan also improved on the criteria used for calculating vertex importance. Recently, Lindstrom extended the Rossignac–Borrel algorithm in a different direction, describing an out-of-core implementation that requires only enough memory to store the final (simplified) mesh.[8] Lindstrom also used Garland and Heckbert's quadric error metric (described later) to place the representative vertex for each cell.

One unique feature of the Rossignac–Borrel algorithm is the fashion in which it renders triangles with merged corners. Reasoning that a triangle with two corners collapsed is a line and a triangle with three corners collapsed is a point, Rossignac and Borrel chose to render such triangles using the graphics hardware line and point primitives. Thus, a simplification of a polygonal object will generally be a collection of polygons, lines, and points. The resulting simplifications are therefore more accurate from a schematic than a strictly geometric standpoint. For the purposes of drastic simplification, however, the lines and points can contribute significantly to the recognizability of the object. Low and Tan extended this concept, using thick lines and a dynamically assigned normal to give a cylinder-like appearance to degenerate triangles collapsed to lines.

The original Rossignac–Borrel algorithm and Lindstrom's out-of-core extension, which cluster vertices to a 3D grid, are extremely fast and run in $O(n)$ time for n vertices. The Low–Tan variation is also quite fast, though ranking the vertices by importance slows the algorithm to $O(n \lg n)$. The methods do suffer some disadvantages. Since the algorithms don't preserve topology and don't guarantee the amount of error introduced by the simplified surface, the resulting simplifications are often less pleasing visually than those of slower algorithms. Also, it's 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.

Multiresolution analysis of arbitrary meshes

This adaptive subdivision algorithm by Eck et al.[17] uses a compact wavelet representation to guide a recursive subdivision process. Multiresolution analysis, or MRA, adds or subtracts wavelet coefficients to interpolate smoothly between LODs. This process is fast enough to do at runtime, 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.

This work's chief contribution is that it provides a method for finding a simple base mesh that exhibits subdivision connectivity, so that recursive subdivision will recover the original mesh. As previously mentioned, finding a base mesh is simple for terrain data sets but difficult for general polygonal models of arbitrary topology. MRA creates the base mesh by growing Voronoi-like regions across the original surface's triangles. When these regions can't grow anymore, the Voronoi sites form a Delauney-like triangulation, and the triangulation forms the base mesh.

This algorithm possesses the disadvantages of strict topology-preserving approaches. Manifold topology is absolutely required in the input model, and the original object's shape and genus limit the potential for drastic simplification. The fidelity of the resulting simplifications is quite high for smooth, organic forms. However, 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.[6]

Voxel-based object simplification

Topology-preserving algorithms must retain the original object's genus, which often limits their ability to perform drastic simplification. Topology-insensitive approaches such as the Rossignac–Borrel algorithm don't suffer from these constraints but reduce the topology of their models haphazardly and unpredictably. Voxel-based object simplification by He et al.[11] is an intriguing attempt to simplify topology in a gradual and controlled manner using a signal processing approach.

The algorithm begins by sampling a volumetric representation of the model, superimposing a 3D grid of voxels over the polygonal geometry. It assigns each voxel 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 a lower resolution. Sampling theory guarantees that small, high-frequency features will be eliminated in the low-pass-filtered volume. The algorithm then applies marching cubes to generate a simplified polygonal model. Because marching cubes can create redundant geometry, He et al. used a standard topology-preserving algorithm 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 isn't topology tolerant, since deciding whether sample points lie inside or outside the object requires well-defined, closed-mesh objects with manifold topology.

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extremely fast. The visual fidelity of the resulting simplifications tends to be quite high.

Simplification envelopes

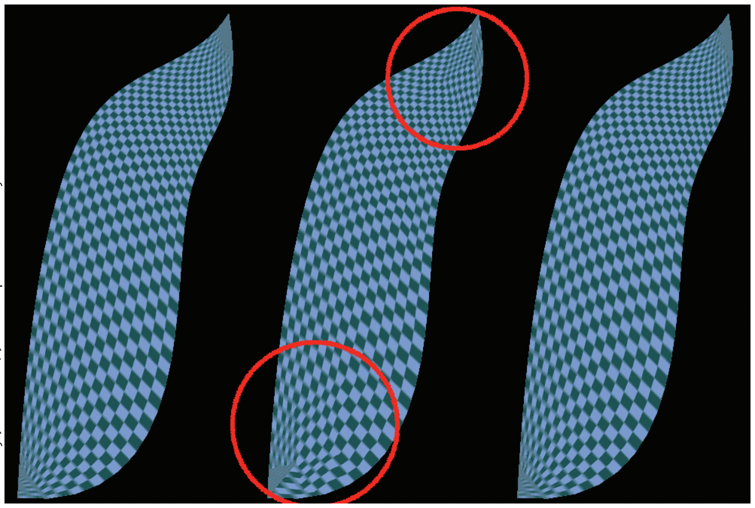
Cohen et al.[9] introduced *simplification envelopes* to guarantee fidelity bounds while enforcing global and local topology. The simplification envelopes of a surface consist of two *offset surfaces*, or copies of the surface offset no more than some distance ε from the original surface. The outer envelope displaces each vertex of the original mesh along its normal by ε, while the inner envelope displaces each vertex by –ε. The envelopes aren't allowed to self-intersect; where the curvature would create such a self-intersection, the algorithm decreases ε in the local neighborhood.

Once created, these envelopes can guide the simplification process. Cohen et al. describe two decimation approaches that iteratively remove triangles or vertices and retriangulate 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 that the surface doesn't self-intersect. The resulting simplifications tend to have excellent fidelity.

Where fidelity and topology preservation are crucial, simplification envelopes prove 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 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 requires an orientable manifold, and 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, Cohen et al. posted their implementation at http://www.cs.unc.edu/~geom/envelope.html.

Appearance-preserving simplification

This rigorous algorithm by Cohen, Olano, and Manocha[3] takes the error-bounding approach of simplification envelopes a step further, providing the best



8 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.

guarantees on fidelity of any simplification algorithm. Fidelity is expressed in terms of maximum screenspace deviation, meaning that the simplification's appearance when rendered should deviate from the original's appearance by no more than a user-specified number of pixels. The authors identified three attributes that affect the simplification's appearance:

- *Surface position*: the coordinates of the vertices.
- *Surface color*: the color field across the 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. For example, simplifying a texture-mapped surface can introduce more distortion in the texture map than in the actual geometry (see Figure 8). Appearance-preserving simplification decouples surface position from color and curvature by storing the latter in texture and normal maps (a normal map resembles a bump map), respectively. The model then reduces to a simple polygonal mesh with texture coordinates, from which the simplification algorithm computes LODs. The simplification process thus filters surface position, while the graphics hardware filters color and curvature information at runtime 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. The algorithm applies this deviation metric 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 aren't allowed. Of course, this requirement constrains the degree of simplification possible, making appearance-preserving simplification less suitable for drastic simplification.

Although texture-mapping graphics hardware is commonplace, hardware support for normal- or bump-mapping is just beginning to appear in consumer-level systems. Appearance-preserving simplification is thus most useful today on models that don't require dynamic lighting, such as radiositized datasets. In the near future, as more sophisticated shading becomes ubiquitous in hardware, appearance-preserving simplification could become the standard for high-fidelity simplification.

Quadric error metrics

This vertex-merging algorithm by Garland and Heckbert[5] strikes perhaps the best balance yet between speed, fidelity, and robustness. The algorithm proceeds by iteratively merging pairs of vertices, which need not be connected by an edge. Candidate vertex pairs include all vertex pairs connected by an edge, plus all vertex pairs separated by less than a user-specified distance threshold t. The algorithm's major contribution is a new way to represent the error introduced by a sequence of vertex merge operations, called the *quadric error metric*. A vertex's quadric error metric is a 4×4 matrix that represents the sum of the squared distances from the vertex to the planes of neighboring triangles. Because 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 merged vertex's quadric error metric. 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 algorithm removes the vertex pair with the lowest merge error from the top of the queue and merges it. The algorithm then updates the errors of all vertex pairs involving the merged vertices and repeats the process.

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extremely fast. The visual fidelity of the resulting simplifications tends to be quite high, even at drastic levels of simplification. Because disconnected vertices closer than t may merge, the algorithm doesn't require manifold topology. This lets holes close and objects 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 algorithm's running time, approaches $O(n^2)$ as t approaches the model's size. Erikson and Manocha[18] proposed an adaptive threshold selection scheme that addresses this problem. They also improved fidelity for certain models by incorporating a surface-area metric, and addressed the lack of support for shading attributes (color, normal, and texture coordinates) in the original Garland–Heckbert algorithm. Garland and Heckbert also described an extension for color in a later paper,[19] but Hoppe probably presented the best extension of quadrics to handle attributes.[20] All told, the simple-to-implement quadric-error-metrics algorithm may be the best combination of efficiency, fidelity, and generality currently available for creating LODs. Even better, Garland and Heckbert released their implementation as a software package called QSlim, available at http://graphics.cs.uiuc.edu/~garland/software/qslim.html.

Image-driven simplification

This unique algorithm by Lindstrom and Turk[4] was the first to address simplification directly in terms of how the simplified model will look when rendered. Like the quadric-error-metrics algorithm, simplification occurs through a sequence of edge collapse operations. The unique feature of image-driven simplification is the error metric used to order the edge collapses. While all other simplification algorithms to date use some form of geometric criteria—possibly modified to account for color, normal, and texture coordinate distortion—Lindstrom and Turk used a purely image-based metric.

Put briefly, we can determine the cost of an edge collapse operation by performing the collapse and rendering the model from multiple viewpoints. The algorithm compares the rendered images to images of the original model and sums up the mean-square error in luminance across all pixels of all images. It sorts all edges under consideration into a priority queue according to the total error they induce in the images and chooses the edge collapse that induces the least error. The algorithm then reevaluates and resorts nearby edges into the queue and continues the process.

Evaluating an edge collapse is clearly an expensive step. Rendering the entire model for every edge and from many viewpoints—Lindstrom and Turk used 20 viewpoints in their implementation—would be almost prohibitively expensive, even with hardware-accelerated rendering. To reduce the cost, Lindstrom and Turk exploited the fact that a typical edge collapse affects only a small region of the screen and thus a small fraction of the total triangles. They used a clever scheme for "unrendering" and rerendering the affected triangles, based on spatial hash tables that record the position of each triangle within each image. In this way, only a small portion of the triangles need be rendered per image to evaluate each edge collapse.

Using an image-based metric addresses several thorny problems in polygonal simplification. Most geometry-based algorithms that account for surface attributes such as color and texture coordinates use an arbitrary user-specified weighting to determine the relative importance of preserving these attributes versus preserving geometric fidelity. Evaluating each simplification operation according to its effect on a rendered image provides a direct, natural way to balance geometric and shading properties. Other advantages of the image-based approach include high-fidelity preservation of silhouette regions coupled with drastic simplification of unseen model geometry, and simplification sensitivity to artifacts caused by shading interpolation as well as to the content of texture maps across the surface.

The primary disadvantage of the image-driven approach for many developers will undoubtedly be the algorithm's speed. Despite the optimizations mentioned, reducing a model comprising tens of thousands of polygons to a few hundred polygons could take hours. To address this, Lindstrom and Turk performed two passes, presimplifying the model with a fast geometry-driven algorithm before applying image-driven simplification. Still, the image-driven stage ranges from several minutes to a few hours, much slower than the fastest geometry-driven algorithms. Depending on the application, however, the high visual quality of the resulting simplifications will undoubtedly be worth the wait to some developers.

Progressive meshes

A *progressive mesh* represents polygonal models as a sequence of edge collapses. Hoppe introduced progressive meshes as the first dynamic simplification algorithm for general polygonal manifolds[6] and later extended them to support view-dependent simplification.[21] 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 vsplit and ecol operations are fast enough to apply at runtime, supporting dynamic and view-dependent simplification.

Along with the new representation, Hoppe described a careful simplification algorithm that explicitly models mesh complexity and fidelity as an energy function to be minimized. The algorithm evaluates all edges that can be collapsed according to their effect on this energy function and sorts them 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 decrease the energy function. The algorithm then reevaluates and resorts nearby edges 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 introduced a nice framework for handling surface attributes of a progressive mesh during simplification. He categorized such attributes 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 described how to model some of these attributes in the energy function, letting normals, color, and material identifiers guide the simplification process.

Hoppe described 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 screenspace tolerance. Because the algorithm measures deviation tangent to the surface 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 reported that evaluating all three criteria, which share several subexpressions, takes only 230 CPU cycles on average.

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 nonmanifold models as a progressive mesh might present difficulties. Still, the progressive mesh representation provides a powerful and elegant framework for polygonal simplification. Hoppe's energy-minimization approach produces high-fidelity simplifications but is relatively slow and seems somewhat intricate to code. Note, however, that any algorithm based on edge collapses can be used to generate a progressive mesh. For example, the quadric-error-metrics approach would be a fast and simple-to-code alternative. Although the progressive mesh code isn't publicly available, Hoppe published a paper describing its efficient implementation in some detail.[22]

Hierarchical dynamic simplification

This vertex-merging approach by Luebke and Erikson[7] was another of the first to provide a view-dependent simplification of arbitrary polygonal scenes. Hierarchical dynamic simplification (HDS) resembles progressive meshes in many ways, with a hierarchy of vertex merges applied selectively at runtime 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 LODs, 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 HDS queries 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 become small. These nodes can be *folded* into their parent nodes, merging 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 triangles.

Different folding criteria can be plugged into the HDS framework as callbacks that fold and unfold the appropriate nodes. Demonstrated criteria include a screenspace error threshold, a silhouette test, and a triangle budget. The screenspace error threshold monitors the projected extent of vertex clusters and folds nodes smaller than some number of pixels on the screen. The silhouette test uses a precalculated cone of normals to determine whether a vertex cluster currently lies on the silhouette, then tests clusters on the silhouette 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 repeats until unfolding a cluster would violate the triangle budget.

Because constructing 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. HDS' structures and methods are also simple to code. On the other hand, the fidelity of the resulting simplifications tends to be lower than the fidelity of more careful algorithms. Again, it's important to emphasize that any algorithm based on vertex merging (including those using edge collapses) can be used to construct the HDS vertex tree.

This algorithm has been implemented as a public-domain library called VDSlib that provides a framework for view-dependent simplification, using user-defined callbacks to control the construction, simplification, culling, and rendering of the vertex tree. VDSlib is available at http://vdslib.virginia.edu.

Issues, trends, and recommendations

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 general frameworks for experimenting with different simplification strategies, including view-dependent criteria. Settling on this emerging standard will let the simplification field 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. Cignoni et al.[23]

described a nice tool called Metro, available at http://vcg.iei.pi.cnr.it/metro.html, for measuring and visualizing this sort of geometric fidelity.

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 isn't geometric but perceptual: Does the simplification look like the original? To date, only Cohen's appearance-preserving simplification[3] and Lindstrom's image-driven simplification[4] attempt to address this question. Perceptual metrics and perceptually driven simplification seem like crucial topics for further research.

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 plethora of polygonal simplification techniques. Hopefully this article will help developers consider the right issues and pick an algorithm that best suits their needs. ■

References

1. J. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Comm. ACM*, vol. 19, no. 10, 1976, pp. 547-554.
2. W. Schroeder, J. Zarge, and W. Lorenson, "Decimation of Triangle Meshes," *Computer Graphics* (Proc. Siggraph 92), vol. 26, ACM Press, New York, 1992, pp. 65-70.
3. J. Cohen, M. Olano, and D. Manocha, "Appearance Preserving Simplification," *Computer Graphics* (Proc. Siggraph 98), vol. 32, ACM Press, New York, 1998, pp. 115-122.
4. P. Lindstrom and G. Turk, "Image-Driven Simplification," *ACM Trans. Graphics*, vol. 19, no. 3, July 2000, pp. 204-241.
5. M. Garland and P. Heckbert, "Simplification Using Quadric Error Metrics," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 209-216.
6. H. Hoppe, "Progressive Meshes," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 99-108.
7. D. Luebke and C. Erikson, "View-Dependent Simplification of Arbitrary Polygonal Environments," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 199-208.
8. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), vol. 34, ACM Press, New York, 2000, pp. 259-262.
9. J. Cohen et al., "Simplification Envelopes," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 119-128.
10. W. Lorenson and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics* (Proc. Siggraph 87), vol. 21, ACM Press, New York, 1987, pp. 163-169.
11. T. He et al., "Voxel-Based Object Simplification," *Proc. Visualization 95*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 296-303.
12. J. El-Sana and A. Varshney, "Controlled Simplification of Genus for Polygonal Models," *Proc. IEEE Visualization 97*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 403-412.
13. P. Lindstrom et al., "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 109-118.
14. W. Schroeder, "A Topology-Modifying Progressive Decimation Algorithm," *Proc. IEEE Visualization 97*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 205-212.
15. J. Rossignac and P. Borrel, "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, Springer-Verlag, Berlin, 1993, pp. 455-465.
16. K-L. Low and T.S. Tan, "Model Simplification Using Vertex Clustering," *Proc. 1997 Symp. Interactive 3D Graphics*, ACM Press, New York, 1997, pp. 75-82.
17. M. Eck et al., "Multiresolution Analysis of Arbitrary Meshes," *Computer Graphics* (Proc. Siggraph 95), vol. 29, ACM Press, New York, 1995, pp. 173-182.
18. C. Erikson and D. Manocha, "GAPS: General and Automatic Polygonal Simplification," *Proc. 1999 Symp. Interactive 3D Graphics*, ACM Press, New York, 1999, pp. 79-88.
19. M. Garland and P. Heckbert, "Simplifying Surfaces with Color and Texture using Quadric Error Metrics," *Proc. IEEE Visualization 98*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 263-270.
20. H. Hoppe, "New Quadric Metric for Simplifying Meshes with Appearance Attributes," *Proc. IEEE Visualization 99*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 59-66.
21. H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 189-198.
22. H. Hoppe, "Efficient Implementation of Progressive Meshes," *Computers & Graphics*, vol. 22, no. 1, 1998, pp. 27-36.
23. P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: Measuring Error on Simplified Surfaces," *Computer Graphics Forum*, vol. 17, no. 2, 1998, pp. 167-174.

David Luebke *is an assistant professor at the University of Virginia. His research focuses on interactive computer graphics, especially the problem of rendering very complex scenes in real time. His current work includes perceptually guided interactive rendering, automatic acquisition of 3D scenes, occlusion culling, and frameworks for high-speed view-dependent rendering. He received a BA in chemistry from Colorado College in 1993 and a PhD in computer science from the University of North Carolina in 1998.*

Readers may contact Luebke at the Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903-2442, email luebke@cs.virginia.edu, http://www.cs.virginia.edu/~luebke.