University of Virginia Technical Report CS-99-33:

ROBUST VIEW-DEPENDENT SIMPLIFICATION FOR VERY LARGE-SCALE CAD VISUALIZATION

David P. Luebke

Department of Computer Science

University of Virginia, Thornton Hall

Charlottesville, VA 22903 (U.S.A.)

804-924-1021 (Work) 804-982-2214 (Fax)

ABSTRACT

View-dependent simplification (VDS) is a novel polygonal simplification algorithm uniquely suited to the interactive visualization of very large-scale CAD datasets. VDS adjusts the simplification continually according to view-dependent parameters such as the viewpoint position and orientation. As a result, objects can span several levels of detail, degrading smoothly from high fidelity where necessary to low fidelity where possible. VDS is also global, able to process the entire database without first decomposing the environment into individual objects. The resulting system enables interactive display of very complex polygonal CAD models consisting of thousands of parts and millions of polygons. VDS supports various preprocessing algorithms and various view-dependent criteria, providing a general framework for dynamic view-dependent simplification.

Keywords: *polygonal simplification, multiresolution methods, interactive visualization*

1 INTRODUCTION

Interactive visualization of very large-scale CAD models is an increasingly crucial problem. More and more enterprises and industries are embracing full-system CAD processes, producing and employing CAD models of unprecedented detail and completeness. These models span the spectrum of products from submarines to power plants, from airplanes to offshore oilrigs. Such large-scale CAD databases can serve as a single unified resource for simulation and design, cutting costs, streamlining the design process, and speeding up production. Interactive computer graphics is an integral component of the full-system CAD process, enabling the design, visualization, and manipulation of these datasets. Despite tremendous strides in computer graphics hardware, however, the growth in complexity of large-scale models continues to outstrip our capability to render them interactively. When converted to polygonal form for interactive rendering, today's large-scale models easily reach 100 million polygons, two to three orders of magnitude beyond what a high-end commercial graphics platform can render interactively. Tomorrow's models will undoubtedly measure billions of polygons. To achieve interactive rendering rates on such large-scale datasets clearly requires some algorithmic means of managing geometric complexity.

Polygonal simplification provides a powerful tool for managing this complexity. These techniques simplify the polygonal geometry of small, distant, or otherwise unimportant portions of the model, reducing the rendering cost without a significant loss of visual detail. This article presents a novel polygonal simplification approach uniquely suited to interactive rendering of very large-scale CAD databases.

1.1 Traditional Polygonal Simplification

Polygonal simplification is at once a very current and a very old topic 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 [1,3]. Recent years have seen a flurry of research into generating such multi-resolution representations of objects automatically by simplifying the polygonal geometry of the object. Figure 1 illustrates

traditional polygonal simplification. Multiple versions of each object are created at progressively coarser *levels of detail* or *LODs* in a preprocess. At run time the system picks which LOD will represent the object based on criteria such as distance from the viewer.

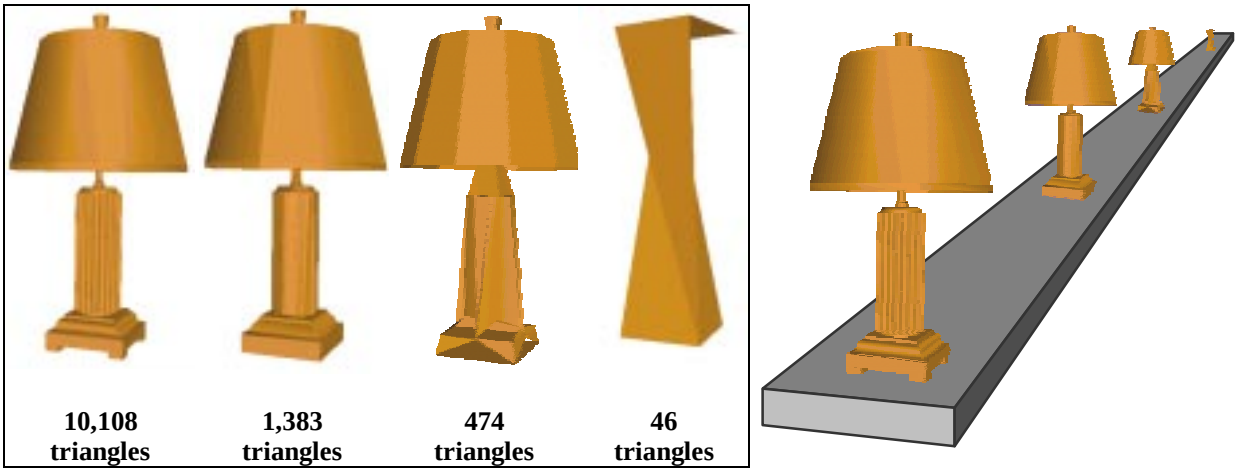


Figure 1: The traditional approach to polygonal simplification creates multiple levels of detail (LODs) of each object in a preprocess, and picks an LOD at run time based on distance.

1.2 Motivation for a Different Approach

This article describes an algorithm conceived for very large-scale CAD databases, a class of models for which earlier simplification methods often prove inadequate. Several features of such models make simplification a difficult task. To begin with, large-scale CAD models are by their nature handcrafted, often by many designers with different styles and levels of expertise. As a result, the models tend to be messy, often containing topological degeneracies of every sort. The sheer complexity of these models can also be daunting. Massive models consisting of thousands of parts and millions of polygons are not uncommon. Such massive CAD models often represent entire scenes rather than objects, and typically exhibit a high dynamic range, containing structural elements as large as the model as well as small, complex parts and assemblies.

An ideal polygonal simplification algorithm for large-scale CAD models should possess a few key attributes. Such an algorithm should be:

Fast. Many traditional algorithms are quite slow, taking minutes or even hours to create LODs for a complex object. 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.

Capable of topology reduction. Most traditional LOD algorithms both require and preserve manifold topology in the polygonal mesh. Requiring clean mesh topology hinders the usefulness of such algorithms on handcrafted CAD models, which as noted above often contain topological degeneracies. In addition, preserving mesh topology implies preserving the overall genus, which as Figure 2 shows, can limit the amount of simplification possible.

Capable of drastic simplification. Visualizing truly large-scale CAD datasets requires reducing the polygonal complexity of those datasets by three to four orders of magnitude. Traditional methods work on a per-object basis, creating separate levels of detail for each object in the model. As the next sections argue, this limits the amount of drastic simplification possible.

(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.

1.3 Drastic Simplification: the Problem With Large Objects

Creating multiple LODs per object is useless for very large objects. Such an object—for example, the terrain landform in an civil engineering project, or the hull of a ship in maritime CAD—presents a fundamental problem: parts of the object will always be near the viewer, while other parts will always be distant. If the object contains many polygons, using a highly detailed LOD would mean high fidelity but low frame rates and jerky motion; using a low

detail would provide smooth motion but terrible fidelity. The usual solution is to break up large objects by hand into smaller objects that can be simplified separately. This can entail a great deal of work, however, and gives rise to the problem of *cracks* between adjacent objects simplified to different levels of detail.

The solution presented here involves *view-dependent simplification*, in which the level of detail varies across the object according to interactive viewing parameters such as viewer position and orientation. The terrain landform and ship hull present no problem to a view-dependent simplification algorithm because only the portions of the object near the viewpoint need to be rendered in high detail. The bulk of the object can still be simplified drastically, rescuing frame rates while preserving visual fidelity.

1.4 Drastic Simplification: the Problem With Small Objects

Complex assemblies of many small objects present another problem for traditional per-object LOD. The diesel engine model shown in Figure 3, contains over two hundred small parts (and at that is not particularly detailed). Assume an excellent LOD algorithm, which can reduce with good fidelity each of these parts to a single cube: the entire assembly still requires over 2,400 triangles to render. From a distance, the whole engine may cover only a few pixels on the viewer's screen. In this situation a single, fifty-polygon, roughly engine-shaped block makes a better approximation than two hundred small cubes.

Our solution is to use a *global simplification* algorithm that treats the entire scene rather than individual objects within the scene. With knowledge about the entire scene, the algorithm can decide when to start combining the various parts of the diesel engine. At a low enough level of detail, the whole engine (and perhaps nearby portions of the walls and floor) can be merged and represented by that fifty-polygon block. Note that the idea of global simplification dovetails nicely with a view-dependent approach. Since view-dependence allows different portions of an object to be represented at different levels of detail, *the entire scene can be treated as a single all-inclusive object for view-dependent simplification*.

These points are important and bear repeating: for drastic simplification using per-object LOD algorithms, large objects must be subdivided and small objects must be combined. Doing this manually can mean a great deal of work. A global, view-dependent algorithm is

better suited to drastic simplification, and thus better suited for very large-scale CAD models, than the traditional approach of creating separate LODs for each object.

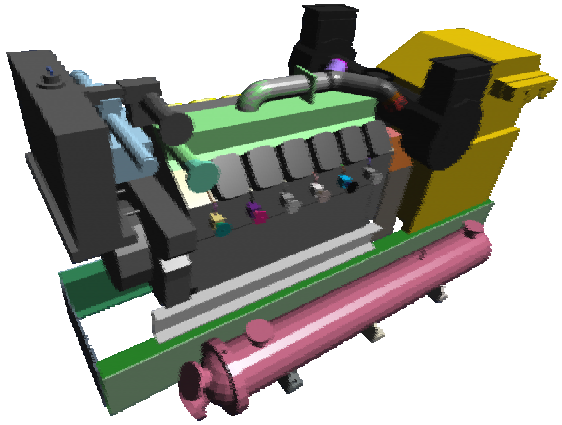


Figure 3: A diesel engine model with over 200 parts.
Courtesy Electric Boat Division, General Dynamics Corp.

1.5 The VDS algorithm

These considerations led to the algorithm presented in this article, called simply *view-dependent simplification* or *VDS*. VDS provides a framework for polygonal simplification via *vertex merging*. This operation, in which several polygon vertices are collapsed together into a single vertex, provides the fundamental mechanism for removing polygonal detail. Merging vertices that share an edge of a triangle makes that triangle redundant, allowing it to be removed. Note the use of *triangle* rather than *polygon*. The constant size and guaranteed planarity of triangles make them preferable to generic polygons, and like most simplification algorithms, VDS assumes that polygonal models have been fully triangulated.

VDS was designed explicitly to address the particular demands of large-scale CAD visualization. The algorithm permits global simplification, with a single large data structure comprising (if the user so desires) the entire model. This structure is the *vertex tree*, a hierarchy of vertex merge operations that encodes a continuum of possible levels of detail across the whole model. Applying a node's vertex merge operation collapses all of the vertices within the node together to a single vertex, eliminating triangles whose corners have been collapsed together. This is called *folding* the node. Likewise, a node may be *unfolded* by splitting that single vertex into the vertices of the node's children. Triangles filtered out when the node was folded become visible again when the node is unfolded, increasing the

triangle count. Figure 4 illustrates a simple two-dimensional example mesh and vertex tree, along with a sequence of folding operations.

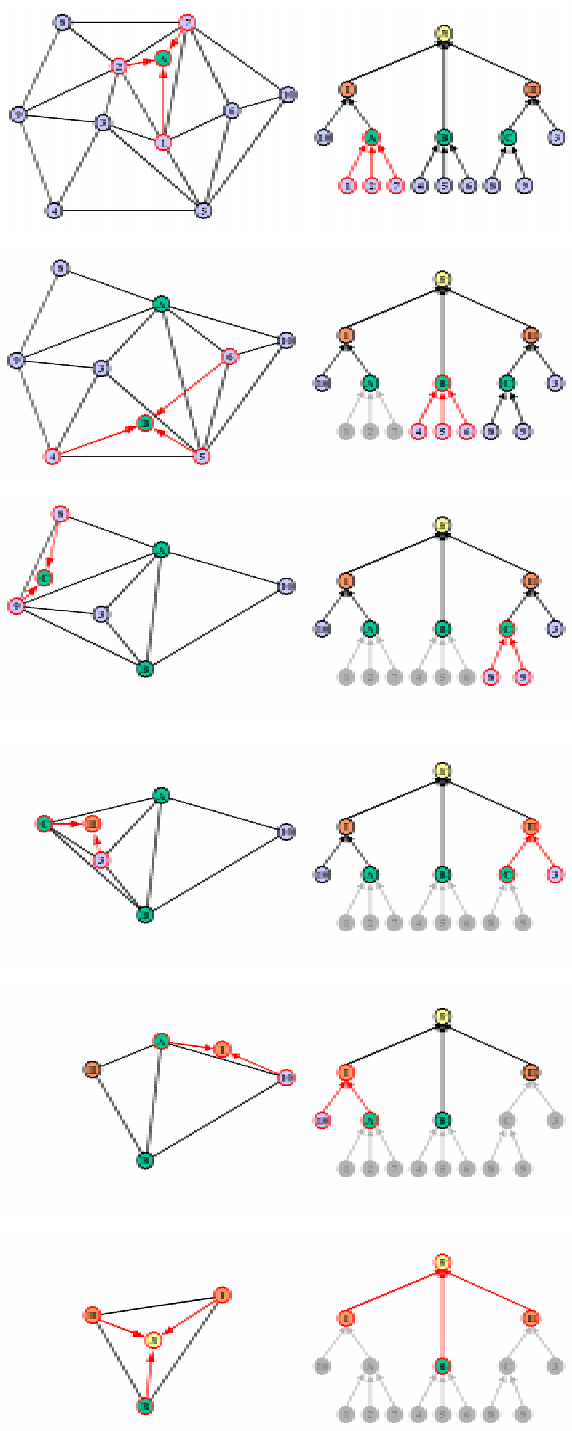


Figure 4: A sequence of fold operations. Folding each node removes some triangles from the scene, reducing the scene to a single triangle and finally to the root node *R*.

Note that the vertex tree contains information only about the vertices and triangles of the model. The algorithm makes no assumptions about the connectivity of those primitives. In particular, the triangles are not assumed to form a manifold mesh or approximate a smooth surface. This is another important feature of the VDS framework: because the simplification operates on the level of triangles and vertices rather than meshes and surfaces, manifold topology is not required and need not be preserved.

The entire system is dynamic and view-dependent. Nodes to be folded or unfolded are continually chosen at run-time based according to user-specified criteria. For example, a common criterion is to set a threshold on projected node screen size. In this mode, the user sets a screenspace-size threshold—say two pixels—before flying the viewpoint interactively around the model. The screenspace extent of each node is monitored: as the viewpoint shifts, certain nodes in the vertex tree will shrink in apparent size, falling below the two-pixel threshold. These nodes will be folded and redundant triangles removed from the scene. Other nodes will increase in apparent size and will be unfolded into their constituent child nodes, introducing new vertices and new triangles into the display list. Adjusting the threshold lets the user interactively control the degree of simplification and select the right balance of fidelity and performance.

Other criteria can be used to drive the system; VDS is less a particular algorithm than a general framework from which algorithms can be constructed. The only essential invariants of the VDSlib framework are the vertex tree and its associated methods (e.g., folding and unfolding nodes). Decisions such as how the vertex tree is constructed and which view-dependent criteria are used to fold and unfold nodes flesh out the framework into a specific algorithm.

2 STRUCTURES AND METHODS

2.1 The Vertex Tree

The VDS vertex tree spans the entire model, organizing every vertex of every polygon into one global hierarchy encoding all simplifications VDS can produce. Leaf nodes each represent a single vertex of the original model; internal nodes represent the merging of multiple vertices from the original model into a single vertex called the *proxy*. A proxy is

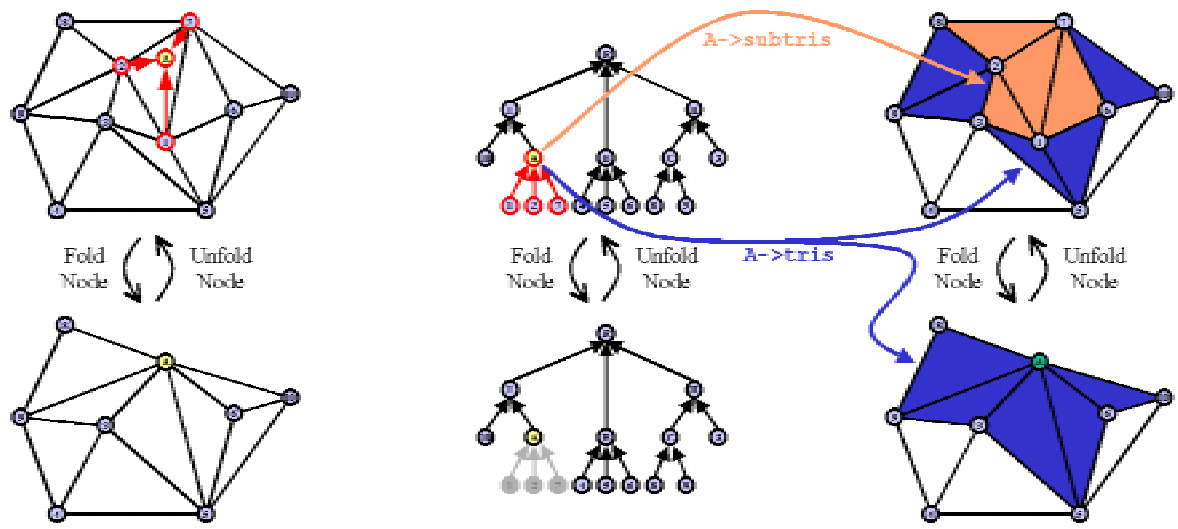
associated with each node in the vertex tree. We say a node **N** *supports* a vertex **V** of the original model if the leaf node representing **V** is a descendent of **N**. Each node in the vertex tree, then, supports a subset of the vertices in the original model; the root node supports every vertex of the entire model. A node supports a triangle **T** of the original model if it supports one or more of the vertices that form **T**'s corners.

Folding a node merges the vertices supported by the node into its proxy, and *unfolding* a node reverses the process. To define these terms more carefully, assume for simplicity that a node's children must all be folded before the node can be folded (since those children can first be folded recursively if necessary, this assumption does not limit the power of the fold operation). This requirement reduces the fold process from merging all vertices supported by a node to merging the proxies of that node's children. Similarly, unfolding a node assumes that the node's parent is unfolded, and splits a node's representative vertex into just the few representative vertices of the node's folded children. Defined this way, fold and unfold are local operations that make only incremental changes to the vertex tree.

Folding and unfolding a node always affects certain triangles. One set of triangles, called the node's relevant triangles or *reltris*, will change in shape as a corner shifts during fold and unfold operations. Another set of triangles, called the node's *subtris*, will disappear when the node is folded and reappear when the node is unfolded (Figure 5). Since reltris and subtris do not depend on the state of other nodes in the vertex tree, they can be computed offline and accessed very quickly at runtime.[1] This is the key observation behind VDS.

---

1 In fact, calculating reltris can be postponed until run-time or avoided altogether; these optimizations are discussed later.



(a) Nodes 1, 2, 7 merge to form A *(b) The local vertex tree* *(c) Reltris and subtris of node A*

Figure 5: Reltris and subtris of a node in the vertex tree. The highlighted node A represents the clustering of nodes 1, 2, and 7.

Unfolded nodes are labeled *active*; folded nodes are labeled *inactive*. During operation the active nodes constitute a cut of the vertex tree, rooted at the root node, called the *active tree*. Folded nodes with active parents are a special case; these nodes form the boundary of the active tree and are labeled *boundary* (Figure 6). Since the location of the boundary nodes determines which vertices in the original model have been collapsed together, the path of the boundary nodes across the vertex tree completely determines the current simplification. Notice that by definition, only boundary nodes can be unfolded and only active nodes whose children are all boundary nodes can be folded.

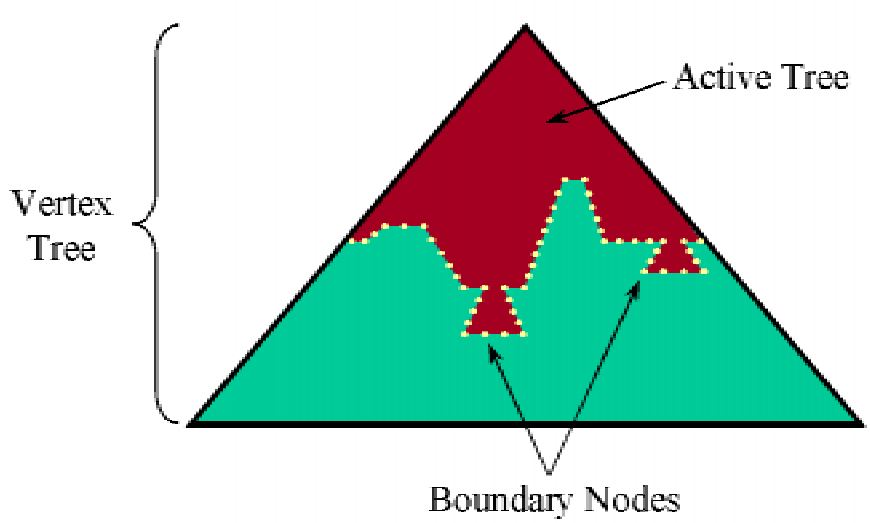


Figure 6: The vertex tree, active tree, and boundary nodes.

Each node in the vertex tree includes the basic structure described below; explanations of the individual fields follow.

```
struct Node {
   Int         depth;      // depth of the node in vertex tree
   NodeStatus  label;      // status: active, boundary, or inactive
   Coordinate  proxy;      // node's representative vertex
   BoundingVol bound;      // bounding volume of all tris supported
   Node *      parent;     // parent node
   Node *      children[]; // child nodes
   Tri *       reltris[];  // triangles that change shape upon folding
   Tri         subtris[];  // triangles that disappear completely
};
```

- `depth`: the depth of the node in the vertex tree.
- `label`: the node's status: *active, inactive, or boundary*.
- `proxy`: the coordinates of the node's representative vertex, to which all vertices represented by the subtree rooted at this node are collapsed.

`bound`: a bounding volume that contains all triangles supported by this node. For simplicity the current implementation uses spheres.

- `parent`: the parent of the node in the vertex tree
- `children`: a list of the node's children in the vertex tree.

`reltris`: a list of triangles with exactly one corner supported by the node. These are the triangles whose corners must be adjusted when the node is folded or unfolded.

`subtris`: a list of triangles with two or three corners supported by the node, but no more than one corner supported by each child of the node. These triangles will be filtered out if the node is folded, and re-introduced if the node is unfolded.

Note that for memory efficiency, fields such as `depth` and `label` can in fact be combined into a single field, and that lazy evaluation of triangle corners (described below) can be used to eliminate the `reltris` field entirely.

2.2 The Active Triangle List

While the vertex tree represents every simplification of the model possible in the VDS system, the *active triangle list* represents the current simplification being rendered. The chief

purpose of the active triangle list is to take advantage of temporal coherence. Frames in an interactive viewing session typically exhibit only incremental shifts in viewpoint, so the set of visible triangles remains largely constant. In its simplest form, the active triangle list is just a sequence of those visible triangles. Unfolding a node adds its subtris to the active triangle list; folding the node removes them. The active triangle list is maintained in the current implementation as a doubly-linked list of triangle structures, each with the following basic structure:

```
struct Tri {
   NodePath    corners[3];
   Node        *proxies[3];
   Tri         *prev, *next;
};
```

The `corners` field encodes the triangle at its highest resolution, referencing the three leaf nodes representing the original corners of the triangle. The `proxies` field represents the triangle in the current simplification, pointing to the *boundary ancestor* of each corner node. If a node **N** is inactive, its boundary ancestor is the boundary node on the path from **N** to the root. The `proxies` of a triangle in the active triangle list therefore encode the three nodes whose proxies currently represent the `corners` of the triangle.

Rather than referencing triangle `corners` directly via pointers, VDS uses the `NodePath` structure, a bit vector which stores the path to a node from the root of the vertex tree. In a binary vertex tree, for instance, each bit would represent a single branch; in an octree, each three-bit sequence would represent an 8-way branch. The *n*th element of the vector specifies which branch to take at level *n*. Referencing nodes in this fashion has advantages over simple pointers. First, simple bitwise operations can be used to compute the first common ancestor of two nodes, or to determine whether one node is a direct ancestor of another. Equivalent tests without the NodePath bit vector would involve hopping through the vertex tree traversing parent pointers. This will typically exhibit very poor memory coherence and correspondingly poor cache behavior.

A related—and perhaps more important—consideration comes into play when the vertex tree as a whole does not fit in memory, and must be paged in as necessary from disk. In this case accessing a triangle's original full-resolution corner nodes may be an extremely

expensive operation, to be avoided at all costs during runtime. Storing the path to each corner node enables the triangle `proxies` field to be updated during fold and unfold operations using purely local information, never referencing the original corner nodes.

2.3 Methods

The fundamental methods associated with the active triangle list are **`addTri()`**, **`removeTri()`**, and **`renderTriList()`**. A simple implementation uses a doubly-linked list with sentinels:

```
// Dummy sentinel structures start and end active triangle list:
Tri *startTriList, *endTriList;

addTri (Tri *T)
   // append to end of list
   T->next = endTriList;
   T->prev = endTriList->prev;
T->prev->next = T;

removeTri (Tri *T)
   // sentinels ensure prev & next fields won't be NULL
   T->next->prev = T->prev;
   T->prev->next = T->next;

RenderTriList ()
   Tri *T = startTriList->next;
   while (T != endTriList)
         renderTri(T);
         T = T->next;
```

Note that this scheme maintains the active triangle list entirely in place. All triangles in the model are kept in an array; as **`addTri()`** and **`removeTri()`** are called, they thread the doubly-linked list through the array. Though simple, this approach exhibits poor memory coherence: after a long series of **`addTri()`** and **`removeTri()`** calls, the linked list is likely to hop around the array of triangles seemingly at random. If the entire array does not fit into cache (or even into main memory), this can greatly degrade performance. Later we will discuss possible optimizations to avoid this problem.

The fundamental methods of the vertex tree are **`foldNode()`** and **`unfoldNode()`**. These functions add or remove the subtris of the specified node from the active triangle list, update the active boundary, and update the proxies of the node's reltris:

```
foldNode (Node *N)
   N->label = boundary;
   // all children should be labeled boundary; change to inactive
   foreach child C of N
         assert(C->label == boundary)
         C->label = inactive;
   // update tri proxies
   foreach triangle T in N->tris
         // which corner of T does N support?
         foreach corner c of {1,2,3}
               if (T->proxies[c]->parent == N) break;
         T->proxies[c] = N;
   // remove subtris from active triangle list
   foreach triangle T in N->subtris
         removeTri(T);


unfoldNode (Node *N)
   assert (N->label == boundary)
   foreach child C of N
         C->label = boundary;
   N->label = active;
   // update tri proxies
   foreach triangle T in N->tris
         // which corner of T is currently represented by N?
         foreach corner c of {1,2,3}
               if (T->proxies[c] == N) break;
         // which child of N supports T? Check NodePath in T->corners[]
         whichchild = T->corners[c][N->depth];
         T->proxies[c] = T->proxies[c]->children[whichchild];
   // add subtris to active triangle list
   foreach triangle T in N->subtris
         addTri(T);
```

3 View-Dependent Simplification Criteria

These structures and methods enable nodes to be folded or unfolded and triangles added or removed fast enough to respond to run-time events. Criteria for view-dependent simplification in this framework take the form of a function to choose which nodes are folded and unfolded each frame. This section describes three such criteria in our current implementation.

3.1 Screenspace Error Threshold

Our first goal was to remove small and distant triangles from the scene. To formulate this approach more precisely, consider a node in the vertex tree. Folding this node, which represents multiple vertices in the original model, clusters those vertices together into the node's proxy. The error introduced by collapsing the vertices can be thought of as the

maximum distance that a vertex can be shifted during the fold operation. This distance equals the length of the vector between the node's proxy and the clustered vertex farthest from the proxy. The extent of this vector when projected onto the screen is the *screenspace error* of the node. By unfolding exactly those nodes whose screenspace error exceeds a user-specified threshold *t*, VDS enforces a quality constraint on the simplification: no vertex shall move by more than *t* pixels on the screen.

Determining the exact screenspace extent of a vertex cluster can be a time-consuming task, but a conservative estimate can be efficiently obtained by associating a bounding volume with each node in the vertex tree. Our current implementation uses bounding spheres, which allow an extremely fast screenspace extent test but often provide a poor fit to the vertex cluster. The function **`nodeSize()`** tests the bounding sphere of a node and returns its extent on the screen as a fraction of viewport size. The recursive procedure **`adjustTree()`** uses **`nodeSize()`** in a top-down fashion, evaluating which nodes to fold and unfold. Nodes with extent greater than the threshold are unfolded and smaller nodes are folded:

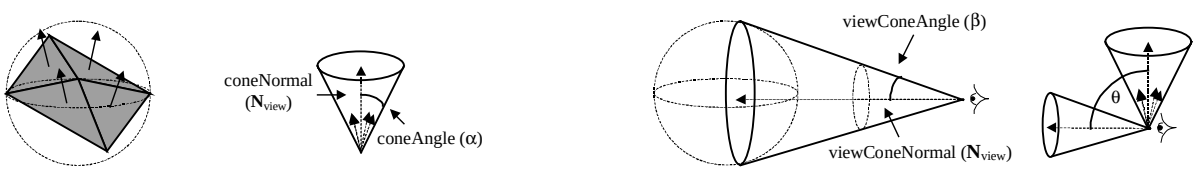
```
adjustTree (Node *N)
   size = nodeSize(N);
   if (size >= threshold)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // size < threshold
         foldSubtree(N);
```

The recursive function **`foldSubtree()`**, as the name suggests, folds the entire subtree rooted at node N:

```
foldSubtree (Node *N)
   if (node->label == active)
         foreach child C of N
               foldSubtree(C);
               foldNode(C);
```

3.2 Silhouette Preservation

Silhouettes and contours are particularly important visual cues for object recognition. Detecting nodes along object silhouettes and allocating more detail to those regions can therefore disproportionately increase the perceived quality of a simplification [15]. A conservative but efficient silhouette test can be plugged into the VDS framework by adding two fields to the `Node` structure: `coneNormal` is a vector and `coneAngle` is a floating-point scalar. These fields together specify a *cone of normals* [14] for the node, which bounds all the normals of all the triangles supported by the node. At run time a viewing cone is created that originates from the viewer position and tightly encloses the bounding sphere of the node (Figure 7). Testing the viewing cone against the cone of normals determines whether the node is completely frontfacing, completely backfacing, or potentially on the silhouette.



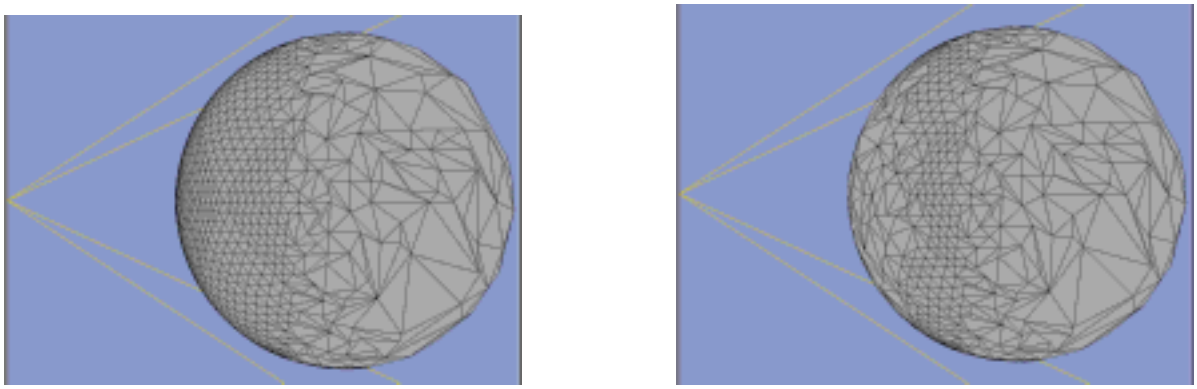
(a) A node containing four triangles, shown with its bounding sphere, and the node's cone of normals.
(b) The viewing cone originates from the viewer and tightly encloses the node's bounding sphere. The angle between N_{cone} and N_{view} is denoted θ.

Figure 7: Silhouette preservation. If any vector within the viewing cone is at right angles to any vector within the cone of normals, the node may be on the silhouette.

```
testSilhouette(Node *node, Coord eyePt)
   α = node->coneAngle;
   N_cone = node->coneNormal;
   β = calcViewConeAngle(eyePt, node);
   N_view = calcViewConeNormal(eyePt, node);
   θ = cos^-1(N_view • N_cone);
   if (θ - α - β > π/2)
         return FrontFacing;
   if (θ + α + β < π/2)
         return BackFacing;
   return OnSilhouette;
```

Silhouette preservation fits easily into the screenspace error metric approach presented above: the silhouette test determines which nodes may be on the silhouette, and these nodes are then tested against a tighter screenspace error threshold (Ts) than interior nodes (T_I).

Following Hoppe [8], we fold nodes that **`testSilhouette()`** evaluates as backfacing, aggressively simplifying portions of the model oriented away from the viewer. This is called *backface simplification*. The **`adjustTree()`** operation is easily modified to incorporate these tests; Figure 8 illustrates silhouette preservation and backface simplification.



(a) 1% error threshold, backface simplification enabled (3,388 faces)

(b) 1% silhouette error threshold, 20% interior error threshold (1,950 faces)

Figure 8: Silhouette preservation and backface simplification. The original model contains 8,192 faces.

3.3 Triangle Budget Simplification

The screenspace error threshold and silhouette test allow the user to set a bound on the fidelity of the simplified scene, but often a bound on the complexity (and thus rendering time) is desired instead. *Triangle budget simplification* allows the user to specify how many triangles the scene should contain. VDS then minimizes the maximum screenspace error of all boundary nodes within this triangle budget constraint. The intuitive meaning of this process is easily put into words: "Vertices on the screen can move as far as *t* pixels from their original position. Using no more than *n* triangles, minimize *t*."

We perform triangle budget simplification using a priority queue of boundary nodes, sorted by screenspace error. The node **N** with the greatest error is unfolded, removing **N** from the top of the queue and inserting the children of **N** back into the queue. This process iterates until unfolding the top node of the queue would exceed the triangle budget, at which point the maximum error has been minimized. Pseudocode for this procedure is straightforward, using a standard heap to implement the priority queue:

```
budgetSimplify(Node *rootnode)
// Initialize priority queue Q to contain just the rootnode
Heap *Q(rootnode);

while (Q->topnode->nsubtris < tribudget)
   unfoldNode(Q->topnode);
   // insert children, sorted by screenspace error:
   foreach child C of Q->topnode
         Q->insert(C);
   tribudget = tribudget - Q->topnode->nsubtris;
   Q->removeTopnode();
```

3.4 Related Work

View-dependent simplification criteria have been proposed by several researchers. Xia and Varshney [15], for instance, describe a novel criterion that uses local illumination information to preserve detail around regions such as specular highlights and shadow boundaries. The screen-space error threshold described above was initially presented in [10]. It is similar in principle to the uniform component of Hoppe's *deviation space* [8]. Deviation space is an ingenious error metric, evaluated at each node, that by its shape also incorporates a form of silhouette preservation. Deviation space thus provides an elegant solution to two view-dependent simplification criteria at once. The advantages of the simpler schemes presented here over Hoppe's deviation space are their speed and robustness, enabling fast preprocessing times and simplification of non-manifold meshes. For the specific domain of large-scale CAD datasets, with messy models and rapid turnaround times, these can be significant advantages indeed.

4 OPTIMIZING THE ALGORITHM

Our initial implementation ran at interactive rates on small models, on the order of 20,000 triangles. The current system has been demonstrated on models more than two orders of magnitude larger. Four sorts of optimization made this possible: exploiting temporal coherence, using visibility information, streamlining the math, and parallelizing the algorithm.

4.1 Exploiting Temporal Coherence

Interactive viewing sessions exhibit a high degree of frame-to-frame coherence, and VDS exploits this coherence throughout. The active triangle list, for example, is based on the assumption that relatively few triangles will be added or removed each frame. As Figure 9 shows, less than 2% of the triangles were added, deleted, or adjusted each frame during a typical path through the Torp model at a 5-pixel screenspace error threshold. The active triangle list exploits this temporal coherence by storing the unchanged triangles from frame to frame, and by supporting efficient add, delete, and update operations for the rest.

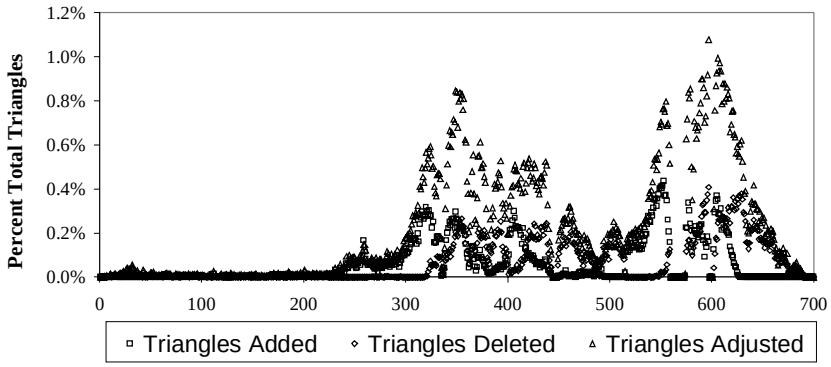


Figure 9: Triangles added, deleted, and adjusted during a 700-frame path through the 699,000-triangle Torp model, using a screenspace error thresholds of 5 pixels.

Vertex tree traversal can also profit from temporal coherence. Just as few triangles change status from frame to frame, few nodes in the vertex tree change status from frame to frame. Under these conditions the **`adjustTree()`** function is inefficient, visiting many nodes unnecessarily. A better scheme is to traverse, not down the vertex tree as **`adjustTree()`** does, but across the vertex tree along the path formed by boundary nodes. This path, called the *boundary path*, is maintained as a doubly-linked list by adding `prev` and `next` fields to the `Node` structure. In this way nodes far from the boundary are never considered and need not even be resident in memory. The function **`adjustPath()`** traverses the boundary path, folding and unfolding nodes as necessary:

```
adjustPath (Node *root)
   Node *currentNode;                  // node currently being tested
   Node *parentNode, *lastParent;      // parent of current & last node

   currentNode = root->next;
   parentNode = lastParent = NULL;
   repeat
         parentNode = currentNode->parent;
         if (parentNode != lastParent)
               lastParent = parentNode;
               // check parent's size first
               if (nodeSize(parentNode) < threshold)
                     // parent falls below threshold; fold
                     foldSubtree(parentNode);
                     currentNode = parentNode;
                     continue;
         // parent is fine, check current node
         if (nodeSize(current) >= threshold)
               // current node too large; unfold
               unfoldNode(current);
         current = current->next;
   until (current == root)
```

Note that `currentNode` is initialized to the root node. The boundary path actually forms a circular linked list, going through the root. This simplifies the maintenance of the boundary path in the **`foldNode()`** and **`unfoldNode()`** functions:

```
foldNode (Node *N)
   Node *pred, *succ;            // predecessor and successor nodes

   N->label = boundary;
   pred = N->children[0]->prev;  // set pred to N's first child's prev
   // all children should be labeled boundary; change to inactive
   foreach child C of N
         C->label = inactive;
         succ = C->next;
   // update tri proxies as before
   // remove subtris from active triangle list as before
   // adjust active boundary
   N->prev = pred;
   N->next = succ;
   N->prev->next = N;
   N->next->prev = N;
```

```
unfoldNode (Node *N)
   Node *pred = N->prev;
   Node *succ = N->next;

   foreach child C of N
         C->label = boundary;
         C->prev = pred;
         pred->next = C;
         pred= C;
   prev->next = succ;
   next->prev = pred;
   N->label = active;
   // update tri proxies as before
   // add subtris to active triangle list as before
```

4.2 Using Visibility Information

For many applications, most of the model is invisible most of the time. VDS can use this visibility information to reduce simplification as well as rendering time. For example, the process of quickly identifying and rejecting objects outside the visible field of view is called *view-frustum culling*. In applications such as architectural walkthroughs, view-frustum culling can greatly decrease rendering time by not rendering invisible portions of the model.

Efficient view-frustum culling in VDS requires modifying the active triangle list. Triangles are added to and removed from the list in a haphazard fashion as nodes are folded and unfolded, so triangles near each other in the model are unlikely to be near each other in the active triangle list. The solution is to impose *spatial coherence* by splitting the active triangle list into a hierarchy of lists, each representing a region of the complete model. Each triangle created by **`unfoldNode()`** is added to the appropriate list. View-frustum culling is applied to the lists themselves; the rendering process tests a bounding volume associated with each list, and skips any lists determined to be invisible.

The vertex tree provides a ready-made hierarchy in which to organize these multiple active triangle lists. Each triangle can be associated with a *cull node* that supports all three corners of the triangle. The cull node is invisible if its bounding volume lies outside the view frustum. Since that volume includes all three corners of the triangle, the triangle need not be rendered if its cull node is invisible. This property holds hierarchically: the descendants of an invisible node are themselves invisible. View-frustum culling, then, can be easily incorporated into VDS by creating an active triangle list for each node in the first few levels

of the vertex tree. When **`unfoldNode()`** creates a triangle, that triangle is added to the list of the appropriate node, which can be quickly calculated via bitwise operations on the triangle's corners.

Not rendering triangles contained by invisible nodes speeds rendering, but an invisible node may still support visible triangles (Figure 10). This fact gives rise to a stronger condition: some nodes are not only invisible but *irrelevant*, meaning that they support no visible triangles. An irrelevant node therefore cannot affect the visible scene, and the simplification traversal can choose to fold the node or simply ignore it. The large majority of invisible nodes are typically irrelevant, so testing for irrelevance provides a significant speedup. A simple test is to extend the node's bounding sphere to include all triangles supported by the node, storing an additional radius.

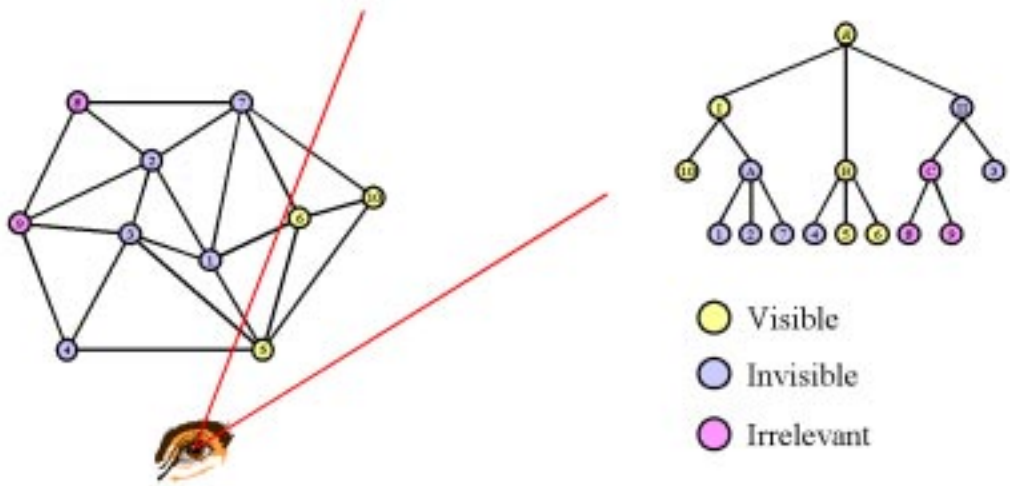


Figure 10: Visible, invisible, and irrelevant nodes. Invisible nodes lie outside the view frustum. Irrelevant nodes are invisible and support no vertices of visible triangles.

4.3 Streamlining the Math

Appropriate use of approximations and careful implementation can greatly streamline the computation involved in evaluating view-dependent criteria. For example, the **`nodeSize()`** function for evaluating screenspace error finds the extent of a cluster of vertices when projected onto the screen. An exact solution would presumably involve projecting the vertices (or their convex hull) and comparing the resulting screen coordinates, a dauntingly expensive operation. Since **`nodeSize()`** is typically called thousands of times per frame, an approximate solution based on bounding spheres is used instead. For a sphere with center ***c***

and radius r, seen from the eyepoint $\boldsymbol{e}$ with field-of-view angle φ, the fraction of viewport F occupied by the sphere is estimated by:

$$F = \frac{r}{\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2)} \tag{1}$$

Note that this approximation assumes that the sphere lies in the center of the field of view, and slightly underestimates F for nodes near the edges of the viewport. This fairly terse expression can be optimized still further in context. The function **`adjustTree()`**, for instance, compares each node's screenspace extent F to a user-specified threshold t. This amounts to evaluating the inequality:

$$F \geq t \tag{2}$$

which reduces to:

$$r \geq t\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2) \tag{3}$$

Squaring both sides and dividing by $\tan^2(\varphi/2)$ yields:

$$r^2 \cot^2(\varphi/2) \geq t^2 \|\boldsymbol{c} - \boldsymbol{e}\|^2 \tag{4}$$

The $\cot^2(\varphi/2)$ term is precalculated at the beginning of each frame. This expression is well suited for rapid evaluation, with no division or square root operations. Fixing the field-of-view angle throughout the viewing session enables further optimization by computing and storing the entire $r^2\cot^2(\varphi/2)$ term for each node, instead of just the radius r of the node's bounding sphere. Though these rearrangements may seem minor, this optimization alone more than tripled the speed of the simplification process in practice.

Here is the modified **`adjustTree()`** function, with silhouette tests omitted for clarity. The `threshold2` term, as the name suggests, holds the user-specified threshold, squared, and the new `r2cot2` field of the `Node` structure stores $r^2\cot^2(\varphi/2)$ for the node. Modifying the **`adjustPath()`** function along the same lines is straightforward.

```
adjustTree (Node *N)
   distance2 = (N->center[X] – eyept[X])² +
               (N->center[Y] – eyept[Y])² +
               (N->center[Z] – eyept[Z])²
   if (N->r2cot2 >= threshold2 * distance2)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // node size is below threshold
         foldSubtree(N);
```

4.4 Parallelization: Asynchronous Simplification

Computer graphics applications commonly parallelize by performing the major rendering stages concurrently in pipeline fashion. A traditional level-of-detail system might be divided into SELECT and RENDER stages: the SELECT stage decides which LOD of which objects to render in frame *n* and compiles them into a display list, while the RENDER process renders frame *n*-1 [4]. If *S* is the time taken to select LODs and *R* is the time taken to render a frame, performing the two processes as a pipeline reduces the total time per frame from $R+S$ to $\boldsymbol{max}(R,S)$.

VDS divides naturally into two basic tasks, SIMPLIFY and RENDER. The SIMPLIFY task traverses the vertex tree, folding and unfolding nodes as needed. The RENDER task loops over the active triangle list rendering each triangle. Let the time taken by SIMPLIFY to traverse the entire tree be *S* and the time taken by RENDER to draw the entire active list be *R*. The frame time of a uniprocessor implementation will then be $R+S$, and the frame time of a pipelined implementation will again be $\boldsymbol{max}(R,S)$. The rendering task usually dominates the simplification task, so the effective frame time often reduces to *R*. The exception is during large shifts of viewpoint, when the usual assumption of temporal coherence fails and many triangles must be added and deleted from the active triangle list. This can have the distracting effect of slowing down the frame rate exactly when the user speeds up the rate of motion.

Asynchronous simplification provides a solution: let the SIMPLIFY and RENDER tasks run asynchronously, with the SIMPLIFY process writing to the active triangle list and the RENDER process reading it. This decouples the tasks for a total frame time of *R*,

eliminating the slowdown artifact associated with large viewpoint changes. When the viewer's velocity outpaces the simplification rate in asynchronous mode, the SIMPLIFY process simply falls behind. Typically, this results in a temporary coarsening of the scene quality. Under VDS, the portions of the scene near the viewer are refined to high detail whereas distant portions are simplified to coarse detail. If the user moves forward too quickly for the SIMPLIFY process to keep up, the viewpoint will leave the highly detailed region behind and move into a coarsely represented region. The scene rendered for the viewer remains coarse in quality until the SIMPLIFY process catches up, at which point the scene gradually refines back to the expected quality. This graceful degradation of fidelity is much less distracting than sudden drops in frame rate.

4.5 Lazy Evaluation of Triangle Corners

A final optimization worth mentioning reduces the space, rather than the time, required by VDS. Recall the `reltris` field of the VDS `Node` structure, which stores a list of triangles that must be adjusted when the node is folded or unfolded. We have discovered that this list can be eliminated by moving the update of triangle corners to the RENDER task, just before the triangle is rendered. Since this is the last possible moment for update, and since only triangles that must in fact be rendered are updated, we refer to this as *lazy evaluation of triangle corners*. In our experiments, evaluating triangle corners in lazy fashion slowed the rendering process by around 5%, while decreasing the memory requirements of the vertex tree by around 15%. The worth of this tradeoff depends on the bottlenecks of the application.

5 CONSTRUCTING THE VERTEX TREE

Thus far we have described the VDS vertex tree and its role in dynamic simplification, but have left open the question of how to construct the vertex tree in the first place. The vertex tree is completely determined by the order in which vertices are grouped. Once the hierarchical grouping of vertices is established, the matter of calculating subtris, bounding volumes, and so on becomes a purely mechanical process. How then do we perform this hierarchical vertex clustering?

The possible algorithms form a spectrum, ranging from fast, simple approaches whose resulting simplifications have moderate fidelity to slower, more sophisticated methods with

superb fidelity. The choice of algorithm for constructing the vertex tree is heavily application-dependent. In a design-review setting, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours are unacceptable in this scenario. On the other hand, a walkthrough of the completed model might be desired for marketing purposes. Here it makes sense to use a slower, more careful algorithm to optimize the quality of simplifications and prevent any distracting artifacts. Since our goal is interactive visualization of very large, potentially messy CAD datasets with rapid turnaround, the clustering scheme we present here emphasizes speed and robustness above all.

5.1 Tight-Octree Vertex Clustering

The *octree* provides a simple top-down approach to vertex clustering. An octree is an 8-way tree in which each node represents an axis-aligned cube; the root node cube is created large enough to contain every vertex in the model. The root node is divided in half along the X, Y, and Z axes into 8 cubical subnodes, the vertices are partitioned among these eight children, and the process is recursively repeated for any subnode with more than one vertex. In this way, vertices are clustered roughly according to proximity. Neighboring vertices are likely to get clustered near the leaves of the tree, whereas distant vertices merge only at higher levels of the tree.

CAD models are often locally dense but globally sparse, consisting of highly detailed components separated by large areas of low detail or empty space. In this situation, a more adaptive partitioning structure is desired. The *tight octree* is a modified octree in which each node is tightened to the smallest axis-aligned cube that encloses the relevant vertices before the node is subdivided. This tightening ensures that every subdivision partitions the vertices, leading to more balanced trees with fewer nodes to traverse and store, and works very well in practice on all the CAD datasets we have tested.

Tight-octree clustering possesses many advantages. Its simplicity makes an efficient, robust implementation relatively easy to code. In addition, the spatial partitioning of vertices is very fast, bringing the preprocess time of even large models down to manageable levels. Preprocessing the 700,000-polygon torpedo room model, for example, takes only 108 seconds using a tight-octree clustering scheme. Finally, spatial-subdivision vertex clustering

is inherently very general. No knowledge of the polygon mesh is used; manifold topology is neither assumed nor preserved. In the CAD domain, meshes with degeneracies such as cracks, T-junctions, and missing polygons are regrettably common, but tight-octree vertex clustering can operate despite the presence of degeneracies incompatible with many schemes.

5.2 Related Work

Hoppe's work on view-dependent refinement of progressive meshes [8, 9] resembles the VDS algorithm presented here in many ways. A *progressive mesh* is a hierarchy of edge collapse operations similar in principle to a binary VDS vertex tree. The view-dependent criteria introduced by Hoppe to simplify the progressive mesh at run time have already been discussed. For construction of the progressive mesh, Hoppe uses a careful optimization approach that sorts possible edge collapse operations into a priority queue based on the error they add to the mesh [7]. This error is estimated by measuring the deviation of the simplified mesh from the original mesh at multiple points scattered across the local neighborhood.

The most fundamental difference between VDS and view-dependent refinement of progressive meshes is their underlying mechanism of merging vertices. VDS clusters arbitrarily many vertices at once to a single representative vertex, whereas progressive meshes use edge collapse operations that merge exactly two vertices sharing an edge in the mesh. As a result, the vertex hierarchy in a progressive mesh will always be binary, whereas the VDS vertex tree may in principle be *n*-ary[2]. Similarly, applying an edge collapse removes exactly two triangles from the mesh, whereas folding a VDS node may remove many triangles from the scene. Which scheme is better in general is unclear. The binary tree of a progressive mesh will be deeper than the corresponding VDS vertex tree, with many more nodes to traverse and store. On the other hand, the simple and regular structure of the edge collapse operation, which can be represented by a small, constant-size structure, lends itself to efficient storage and traversal. The finer granularity of the edge collapse could be an advantage, since a triangle budget can be specified very precisely, or a disadvantage, since more nodes must be processed to reach a desired level of simplification.

---

[2] In practice, of course, using a tight octree guarantees a vertex tree of maximum degree 8.

In the context of large CAD datasets, for which VDS was designed, the edge collapse operation has some definite disadvantages. Since edge collapses that create non-manifold regions are disallowed, holes in the mesh are not simplified and the genus of the object remains fixed. As argued above, this can limit the potential for drastic simplification of high-genus objects. Moreover, since only vertices that share an edge are merged, each object must be simplified separately, limiting the potential for drastic simplification of complex assemblies of objects. Furthermore, particularly CAD models may contain inherently non-manifold features, such as three triangles meeting at a single shared edge; algorithms based on edge collapses simply cannot represent such models. By clustering based on proximity, without regard to topology or source object, VDS deals well with all of these problem cases.

Once the tight octree has produced a vertex clustering, we use *quadric error metrics* (*QEMs*) to optimize the placement of the representative vertex. QEMs, introduced by Garland and Heckbert [6], provide a simple and fast technique for measuring the sum of the squared distances from a node's proxy to the planes of all of the triangles that node supports. Quadric error metrics require relatively little storage, handle non-manifold surfaces robustly, and produce simplifications of excellent fidelity. We use refinements to the basic QEM algorithm, in particular the surface-area preservation technique used by Erikson and Manocha in the GAPS algorithm [4].

6 RESULTS

All results reported here were obtained on a four-processor SGI Onyx^2 computer with 195 MHz R10K processors, 1152 megabytes of main memory, 4 megabytes of secondary cache, and InfiniteReality graphics.

6.1 Performance

Five sample models were chosen to span several CAD categories and a large range of polygon counts. The models and results are summarized in Table 1. *Engine* is a detailed model of an automobile engine containing over 140,000 triangles. *Cassini* is an aerospace CAD model of the Cassini space probe, provided courtesy of the Jet Propulsion Laboratory. It contains over 415,000 triangles. *AMR* depicts the auxiliary machine room of a notional nuclear submarine, containing approximately 505,000 triangles. *Torp* is another maritime

CAD dataset, representing the torpedo room of the same submarine with approximately 699,000 triangles. The Electric Boat Division of General Dynamics Corporation provided both submarine models. The smallest model, *bunny*, contains 70,000 triangles and comes from the Stanford 3-D Scanning Repository. Though it hardly qualifies as a CAD model, the bunny has become an unofficial benchmark for the polygonal simplification field.

Model	Category	Triangles	Preprocessing time	Vertex tree nodes	Vertex tree storage (w/ gzip)
Bunny	Scanned	69,451	7.2 seconds	50,856	5 Mb
Engine	Mechanical CAD	140,696	25 seconds	102,577	8 Mb
Cassini	Aerospace CAD	415,257	75 seconds	278,329	30 Mb
AMR	Maritime CAD	504,969	86 seconds	394,253	32 Mb
Torp	Maritime CAD	698,872	108 seconds	816,833	48 Mb

Table 1: Names, categories, and complexity of models in the VDS test suite.

6.2 Artifacts

Implementing asynchronous simplification is relatively straightforward, but care must be taken to avoid *dropouts*. Characterized by triangles that disappear for a frame, these transient artifacts occur when the RENDER process sweeps through a region of the active list being affected by the SIMPLIFY process. For example, the **`foldNode()`** operation removes triangles and fills the resulting holes by merging the corners of surrounding triangles. If those neighboring triangles have already been rendered during the frame when **`foldNode()`** adjusts their corners, but the triangle to be removed has not yet been rendered, a hole will appear in the mesh for that frame.

Dropouts prove difficult to eradicate using simple locking schemes without a significant performance penalty. One solution that works well is the *update queue*. Rather than performing the **`foldNode()`** and **`unfoldNode()`** operations, the SIMPLIFY process accumulates these updates into the update queue, marking the node Dirty and placing a Fold or Unfold entry in the queue. At the beginning of every frame, the RENDER process performs the updates in the queue, folding or unfolding each node before marking it Clean again. All changes to the active triangle list take place as a batch before any triangles are rendered; the shared database is thus kept consistent and dropouts are eliminated.

Another visual artifact that VDS can introduce is *mesh folding*. Mesh folding occurs when shifting the position of a vertex causes an attached triangle to flip in orientation (Figure 11). These artifacts are inherent to any vertex-merging or edge-collapse scheme that does not take care to avoid them. The visual effect of mesh folding depends on the rendering parameters. Folding a triangle flips its orientation, so such triangles may not be drawn if backface culling is enabled. If two-sided lighting is enabled, the triangle will be drawn, but since flipping a triangle negates its normal vector, the folded triangle may be shaded differently from the surrounding mesh.

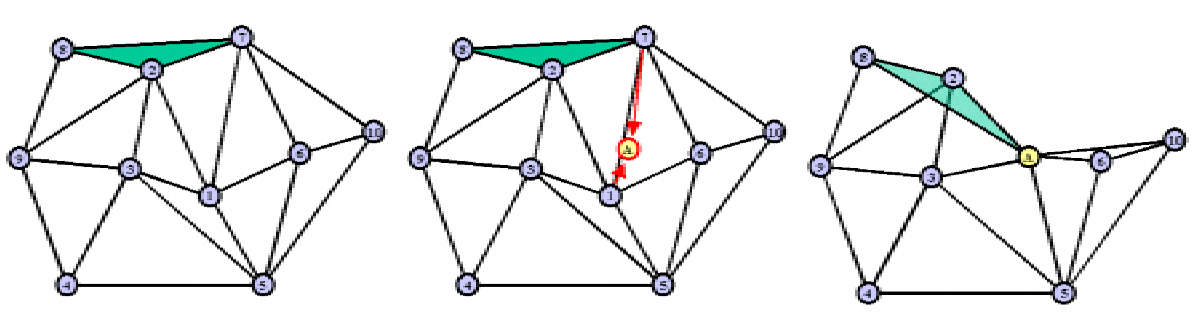


Figure 11: An example of mesh folding. When vertices 1 and 7 are merged to form vertex A, the shaded triangle folds over neighboring triangles, flipping in orientation.

Careful construction of the vertex tree can reduce the likelihood of mesh folding, but to completely eliminate folding artifacts requires additional view-dependent criteria. One possibility is to check the normal of each affected triangle; if a triangle normal is flipped, the fold operation is disallowed. Less expensive tests can be had by enforcing dependency constraints on the mesh, so that certain conditions must be met before a node may be folded or unfolded [8, 15]. Since the artifacts are small, and since additional view-dependent criteria might overly restrict simplification, the current VDS implementation does not attempt to prevent mesh folding. When high fidelity is a concern, however, adding code to prevent these artifacts would certainly be worthwhile.

6.3 Visual Results

Figure 12 shows an examples of the VDS system in action, comparing an original model to run-time simplifications created with various screenspace error tolerances.

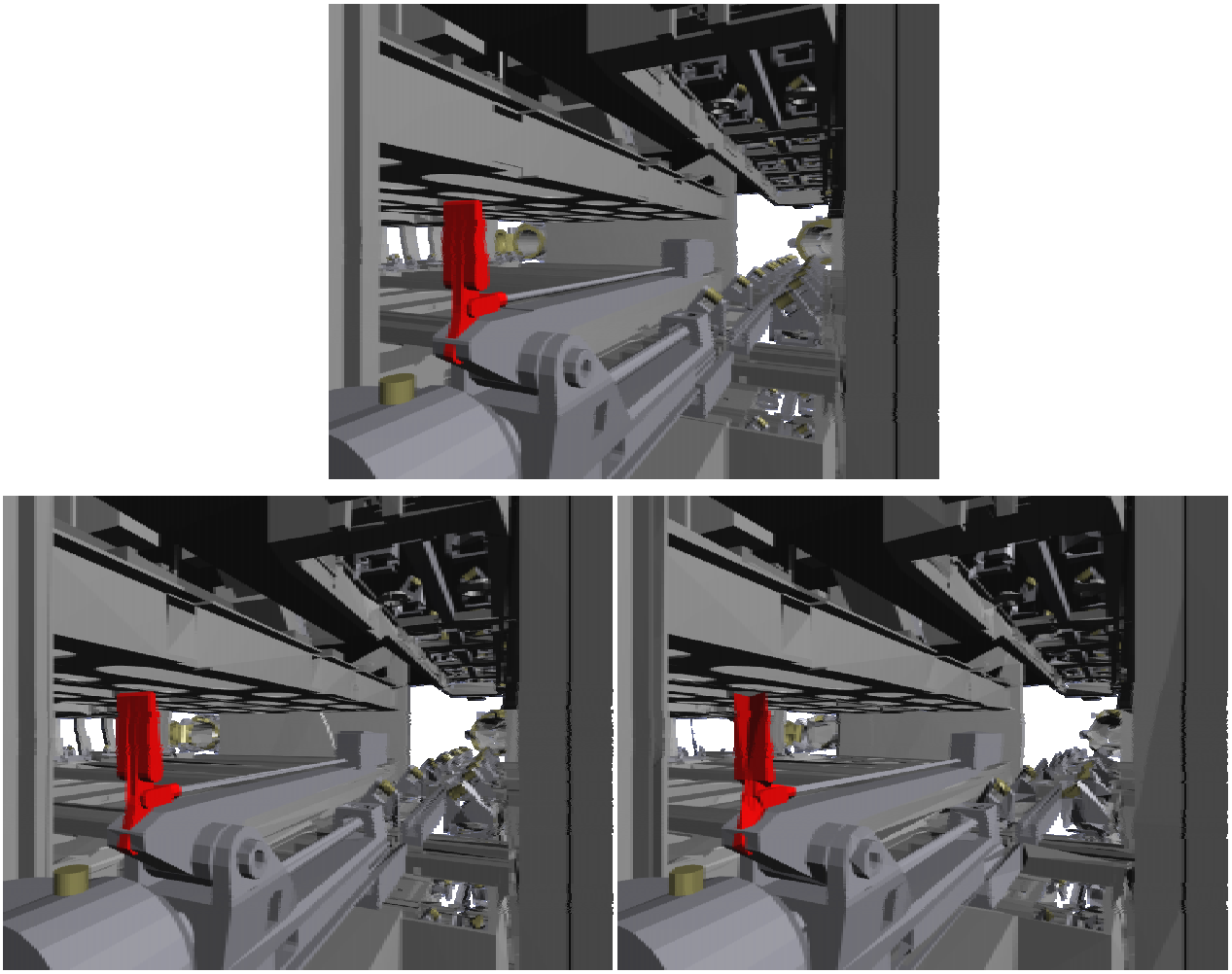


Figure 12: Top: The Torp model at original resolution comprises 698,872 faces.
Left: At 0.8% screenspace error (129,446 faces), visual artifacts are fairly subtle.
Right: At 1.5% screenspace error (76,404 faces), distant objects are simplified to almost schematic levels, while nearby features still possess reasonable fidelity.

7 SUMMARY AND FUTURE WORK

View-dependent simplification can provide a powerful, general framework for visualizing complex polygonal environments. We have described VDS, a view-dependent polygonal simplification algorithm particularly well suited to very large-scale CAD datasets. Such datasets are notoriously difficult for simplification; they are by nature extremely large, complex, and messy models. Key advantages of VDS include the ability to perform drastic simplification—despite the presence of very large objects, very small and numerous objects, or objects of high genus—and the ability to simplify arbitrary polygonal models despite non-manifold mesh topology.

VDS is also flexible, letting the user tailor the vertex tree construction algorithm and view-dependent simplification criteria to the application at hand. We have focused on the tight-octree vertex-clustering algorithm for constructing the vertex tree. The tight-octree is fast, robust, and completely automatic, making it well suited for large-scale CAD datasets with rapid turnaround. We have presented multiple view-dependent criteria for simplification, and discussed many optimizations to the basic VDS system.

The chief disadvantages of VDS are those of any view-dependent polygonal simplification scheme: an increased computational load on the CPU, and a mismatch to current graphics hardware, which is largely oriented towards retained-mode rendering.

Many interesting avenues of future work remain to be explored. One promising criterion for view-dependent simplification would adapt the work of Cohen [2] on *appearance preserving simplification*. Using successive mappings, this approach is able to bound, in screenspace, the distortion of texture maps during simplification. A similar technique can be applied to a *normal map*, which represents surface curvature just as a texture map captures surface color. Appearance preserving simplification thus provides a screenspace bound, not only on geometric deviation, but on deviation in coloration as well. View-dependent appearance-preserving simplification seems a promising area for future research.

Oshima [12] and Reddy [13] describe *gaze-directed simplification* systems, in which level of detail is regulated by the direction of a user's gaze. For example, an object in the center of a user's field of view would be allocated more detail than the same object in the periphery of the user's vision. Both Oshima and Reddy apply these criteria to selection of static LODs. We have begun to experiment with gaze-directed view-dependent simplification, and our initial results are encouraging.

The memory access patterns of VDS could be improved. For example, the current system implements the active triangle list as a doubly linked list to support efficient insert and delete operations. This list is maintained in place, threaded through an array of all triangles in the model. As described above, however, this tends to create a haphazard path that ruins cache coherence as it hops back and forth through the array. A better approach would collect all the active triangles into a single coherent array where they could be rendered with a simple linear pass.

These sort of memory-management issues touch on the larger topic of *out-of-core simplification*, in which the model to be rendered is far larger than main memory. VDS seems well suited to out-of-core simplification, since only the boundary path and active triangle list appear crucial to keep resident in memory. Research problems to be addressed include the out-of-core generation of a vertex tree. Hoppe [9] describes an interesting solution to this problem for the specific domain of terrain rendering; perhaps his approach could be generalized to arbitrary polygonal models.

One important area of future research is the question of how to simplify dynamic polygonal environments. Every simplification algorithm to date assumes that the models to be simplified are static, and must be run from scratch if the model changes. In an active CAD session, however, a designer builds a complex model with a series of incremental, often local changes. For example, many systems are based on constructive solid geometry (CSG) modeling, in which solids are defined by a series of Boolean operations upon simpler solids. Supported operations include union, intersection, and subtraction. A simplification system that supported efficient union, intersection, and subtraction operations upon VDS-style vertex trees could maintain a simplified representation of the model through incremental updates to the design. This should enable the designer to view a larger, more complex portion of the model interactively, which might provide more helpful context. Simplification of dynamic scenes is a challenging problem, and seems likely to be one of the next frontiers of polygonal simplification.

8 REFERENCES

1. Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, Vol. 19, No 10.

2. Cohen, Jon, and D. Manocha. "Appearance Preserving Simplification", *Computer Graphics*, Vol. 32 (SIGGRAPH 98).

3. Cosman, M., and R. Schumacker. "System Strategies to Optimize CIG Image Content". *Proceedings Image II Conference* (Scotsdale, Arizona), 1981.

4. Erikson, Carl, and D. Manocha. "GAPS: General and Automatic Polygonal Simplification", *Proceedings 1999 Symposium on Interactive 3-D Graphics* (Atlanta, Georgia), 1999.

5. Funkhouser, Tom, and C. Sequin. "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments", *Computer Graphics*, Vol. 27 (SIGGRAPH 93).

6. Garland, Michael, and P. Heckbert, "Simplification Using Quadric Error Metrics", Computer Graphics, Vol. 31 (SIGGRAPH 97).

7. Hoppe, Hughes. "Progressive Meshes", *Computer Graphics*, Vol. 30 (SIGGRAPH 96).

8. Hoppe, Hughes. "View-Dependent Refinement of Progressive Meshes", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

9. Hoppe, Hughes. “Smooth view-dependent level-of-detail control and its application to terrain rendering”, *IEEE Visualization `98*, October 1998.

10. Luebke, David, and C. Erikson. “View-Dependent Simplification of Arbitrary Polygonal Environments”, *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

11. Luebke, David. “View-Dependent Simplification of Arbitrary Polygonal Environments”, Ph.D. thesis, Department of Computer Science, University of North Carolina, 1998.

12. Oshima, Toshikazu, H. Yamammoto, and H. Tamura. “Gaze-Directed Adaptive Rendering for Interacting with Virtual Space”, *Proceedings of VRAIS 96*.

13. Reddy, Martin. “Perceptually-Modulated Level of Detail for Virtual Environments”, Ph.D. thesis, University of Edinburgh, 1997.

14. Shirman, L., and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches", *Computer Graphics Forum*, Vol. 12, No 3, 1993.

15. Xia, Julie, and A. Varshney. “Dynamic View-Dependent Simplification for Polygonal Models”, *IEEE Visualization 96*, October 1996.