EUROGRAPHICS '99 / P. Brunet and R. Scopigno
(Guest Editors)

Volume 18 (1999), Number 3

Generalized View-Dependent Simplification

Jihad El-Sana Amitabh Varshney

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We propose a technique for performing view-dependent geometry and topology simplifications for level-of-detail-based renderings of large models. The algorithm proceeds by preprocessing the input dataset into a binary tree, the view-dependence tree *of general vertex-pair collapses. A subset of the Delaunay edges is used to limit the number of vertex pairs considered for topology simplification. Dependencies to avoid mesh foldovers in manifold regions of the input object are stored in the view-dependence tree in an implicit fashion. We have observed that this not only reduces the space requirements by a factor of two, it also highly localizes the memory accesses at run time. The view-dependence tree is used at run time to generate the triangles for display. We also propose a cubic-spline-based distance metric that can be used to unify the geometry and topology simplifications by considering the vertex positions and normals in an integrated manner.*

1. Introduction

Recent advances in three-dimensional shape acquisition, simulation, and design technologies have led to generation of datasets that are beyond the interactive rendering capabilities of the current graphics hardware. To bridge the increasing gap between hardware capabilities and graphics dataset sizes, the complexity of the graphics dataset is reduced such that its visual appearance is similar to the original. This reduction is achieved through several algorithms and techniques such as level-of-detail rendering with multi-resolution hierarchies, occlusion culling, and image-based rendering. Preserving the topology of the input dataset is an important criterion for some graphics application such as tolerancing, drug design, geological, and medical volume visualization. However, for several real-time graphics applications where interactivity is essential and preserving the topology is not required, topology simplification has been shown to yield significant benefits with little difference in visual appearance [7].

Recently, view-dependent simplifications have been introduced to enable various levels of detail to seamlessly co-exist over different regions of the same surface. These levels of detail depend on parameters such as view location, illumination, and speed of motion and are determined per-frame. However, most of these view-dependent simplification algorithms, with the notable exception of the work by Luebke and Erikson [19], are based on edge-collapse, which by itself is inadequate for topology simplification. In order to be able to change the topology of a model we should be able to merge different objects, hence, vertices which are not connected via an edge. This leads us to the first issue that we address in this paper – how can one limit the potentially n^2 such vertex pairs under consideration (where n is the number of vertices of the model).

Second, we propose a unified distance metric that allows measurement of the distance between two vertices taking into account their coordinates as well as normals. Our spline-based distance metric can also be used to provide a reasonable measure of the distance from the view point to the vertices of the object. Since our metric handles the normals and the coordinates in a unified manner, we can use the same distance function during construction of the vertex hierarchy as we use during the run-time view-dependent simplifications. This was not possible in the past.

Dependencies lists were introduced by Xia *et al* [24] and improved later by Hoppe [16] in order to prevent foldovers at run time. But they are expensive to store and to test at run time due to the memory overhead and several non-local accesses. These non-local accesses lead to unnecessary paging for large datasets or on computers with less memory. In addition, it is very hard to extend explicit dependencies lists to handle the view-dependent topology simplification. In this paper we introduce the concept of *implicit dependencies* that ensure run-time consistency in the generated triangulations, with a very small memory overhead and purely local accesses. We expect implicit dependencies to be useful for view-dependent visualization applications dealing with networks external memory prefetching as well as over networks.

2. Related Work

Since related work on geometry simplification has been well surveyed in several recent papers [15, 9, 5] in this paper we shall overview only the related work in the areas of topology and view-dependent simplifications.

2.1. Topology Simplification

Rossignac and Borrel [21] use a global grid to subdivide a model. This approach can simplify the topology if the desired simplification regions fall within a grid cell. He *et al* [14] used low-pass filtering to perform a controlled simplification of the topology of volumetric datasets; however polygonal objects need to be voxelized. El-Sana and Varshney [7] perform genus simplification by extending the concept of α-hulls to triangles. Their approach is based on convolving individual triangles with a L_∞ cube of side α and computing their union. The convolution operation effectively eliminates all holes less than α.

Several edge-collapse-based schemes have been proposed for topology simplification, though not in the context of view-dependent renderings. Schroeder [23] has introduced vertex-split and vertex-merge operations on polygonal meshes for modifying the topology of polygonal models. Vertex splits are performed along feature lines and at corners. Garland and Heckbert [9] present a quadric error metric that can be used to perform genus as well as geometric simplifications by using vertex-pair collapses. Popović and Hoppe [20] introduce the operator of a generalized vertex split to represent progressive changes to the geometry as well as topology.

2.2. View-Dependent Simplification

Adaptive, view-dependent levels of detail were first introduced in the context of terrains by Gross *et al* [11]. Gross *et al* define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Several other researchers have since then presented other methods for view-dependent rendering of terrains. However, keeping with the focus of this paper, we shall only overview previous work done in the area of view-dependent simplifications of generalized meshes.

Progressive meshes have been introduced by Hoppe [15] to provide a continuous resolution representation of polygonal meshes. Progressive meshes are based upon two fundamental operators – edge collapse and its dual, the vertex split as shown in Figure 1. A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses. The sequence $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ is referred to as a *progressive mesh* representation.

Merge trees have been introduced by Xia *et al* [24] as a data-structure built upon progressive meshes to enable real-time view-dependent rendering of an object. These trees encode the vertex splits and edge collapses for an object in a hierarchical manner. The edge collapses are performed using the shortest-edge-first heuristic. Each vertex stores the distance from the user beyond which it will be collapsed to its parent and a distance at which it will be split. The distance metric is defined using view position, view angle, variations in surface normal, local illumination, silhouettes, and front/back-facing regions.

Hoppe [16] has independently developed a view-dependent simplification algorithm. This algorithm proceeds to construct a vertex hierarchy over a progressive mesh in a top-down fashion by minimizing an energy function. Screen-space projection and orientation of the polygons is then used to guide the run-time view-dependent simplifications. Guéziec *et al* [13] demonstrate a surface partition scheme for a progressive encoding scheme for surfaces in the form of a directed acyclic graph (DAG). The DAG represents the partial ordering of the edge collapses with path compression. De Floriani *et al.* [6] have introduced the multi-triangulation(MT). The decimation and refinement in MT is achieved through a set of local operators that affect fragments of the mesh. Then the dependencies between these fragments are used to construct a DAG of these fragments. This DAG is used at run time to guide the change of the resolution of each fragment.

Klein *et al* [18] have developed an illumination-dependent refinement algorithm for multiresolution meshes. The algorithm stores maximum deviation from Phong interpolated normals and introduces correspondence between the normals during the simplification algorithm. Schilling and Klein [22] have introduced a refinement algorithm that is texture dependent. Their algorithm measures the texture distortion in the simplified mesh by mapping the triangulation into the texture space and then measuring the error at vertices and edge intersections. Giang *et al* [10] have presented a method to produce a hierarchy of triangle meshes that can be used to blend different levels of detail in a smooth fashion.

Luebke and Erikson [19] use a scheme based on defining a *tight octree* over the vertices of the given model to generate hierarchical view-dependent simplifications. This approach can simplify topology if the desired simplification regions fall within one cell. However, fine control over the simplification of the topology is not easy to achieve.

3. Our Approach

We present an algorithm that enables geometry and topology simplification in a view-dependent fashion. In the preprocessing stage we construct a hierarchy of vertex-pair collapses to build a view-dependence tree. Among the n^2 vertex-pair candidates we choose the vertex pairs that are connected by the mesh edges and an additional set of vertex pairs determined using a Voronoi diagram (details are in section 4). Details of how the view-dependence tree is constructed are given in section 5. Our view-dependence tree differs from previous work [24, 16] in that it enables topology simplification, does not store explicit dependencies, and handles non-manifold cases. At run-time the view-dependence tree is used to guide the selection of the appropriate level of detail based on factors such as view and illumination parameters.

We have found that distance metrics that combine vertex position and normal perform better than metrics that take into account only position, such as the Euclidean distance metric.

In section 7 we discuss a new metric that combines the normals and positions of the collapsed vertex using cubic splines. Our spline metric tests view frustum, foveation, backfacing regions, and local illumination in a natural fashion.

4. Virtual Edges

A polygonal mesh can be simplified into successively coarser meshes by applying a sequence of edge collapses or vertex-pair collapses. In an *edge collapse* the two vertices which are connected by this edge are collapsed into one vertex and the adjacent triangles are updated appropriately as shown in Figure 1. A *vertex-pair collapse* is a generalization of the edge collapse and involves merging two vertices that may or may not be connected by an edge. The adjacent triangles are updated in a manner similar to that for the edge collapse.

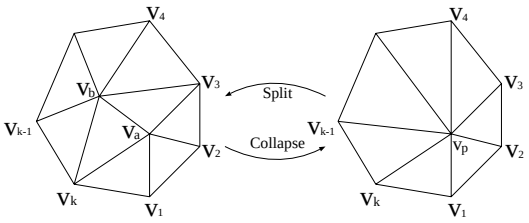


Figure 1: *Edge collapse and vertex split*

For our algorithm to allow topology simplification it needs to collapse vertex pairs that are not connected by an edge. Such vertex-pair collapses allow merging of unconnected components. We say that such a vertex pair is connected by a *virtual edge* while the original model edges are referred to as *real edges*. Virtual edges have been used by Garland and Heckbert [9], Popović and Hoppe [20], and Schroeder [23]. We rely on the Delaunay triangulation to restrict potentially n^2 such virtual edges. The set of virtual edges that we consider is a subset of the Delaunay edges over the dataset vertices. It is important to note here that the Delaunay edges do not necessarily suffice to represent all vertex-pair collapses that might be considered desirable by an application. However, we have found that in practice, the Delaunay edges suffice for most topology simplifications, particularly, when they are supplemented by the real edges of the model as candidates for collapse. We generate the virtual edges for consideration through a Voronoi diagram in which the dataset vertices are the Voronoi sites. We construct the virtual edges from the given set of Voronoi sites by connecting every pair of vertices by a virtual edge if their corresponding Voronoi cells share a Voronoi face and are not connected via a real edge.

Three-dimensional Voronoi diagram can be constructed in $O(n^2 \log n)$ time for n points. Several research groups have developed software for computing three-dimensional Voronoi diagrams and Delaunay triangulations and released it in the public domain. We chose to use *Qhull* [2] to construct the three-dimensional Voronoi diagram since it is robust and can handle degeneracies well.

4.1. Optimization

We have found that a large fraction of the Delaunay edges connect vertices which are already connected via real edges. One way to reduce this fraction is to reduce the number of Voronoi sites, that is the number of vertices we consider. Recently, Amenta *et al* [1] have proposed the concept of Voronoi filtering to reduce such connections between points during surface reconstruction from a set of unorganized sample points. In our case we achieve this reduction by combining the faces to superfaces and computing the Voronoi diagram for the vertices that form the boundary of these superfaces. Thus the virtual edges that are generated are a subset of the edges connecting the boundary vertices of such superfaces. To allow a rich collection of such virtual edges for consideration, we require that all the triangles of any superface form a manifold patch. We define a superface normal as the average of the normals of all the triangles that form it.

Some algorithms have been suggested to compute superfaces [4, 17]. Our algorithm for this construction is very similar to others. In the initialization step every triangle forms a superface. Then in a recursive greedy fashion we combine the two adjacent superfaces that have the minimum angle between their normals and all the shared vertices are manifold. The algorithm stops when the minimum angle between the normals of any two adjacent superfaces is larger than a given threshold. We are interested in superfaces that do not contain sharp edges, hence we use threshold of 75° in our current implementation.

After we have constructed the superfaces, we compute the Voronoi diagram of the vertices which form the boundary of these superfaces. We consider the virtual edges which connect any two vertices that (a) are not connected via a real edge, (b) share a Voronoi face, and (c) do not lie on the boundary of the same superface.

It is important to note that superfaces are used only to reduce the number of vertices we consider to generate the virtual edges. As we pointed out earlier, the virtual edges are used to establish connectivity across different components that would otherwise not have readily merged. Beyond a certain size, the shape and extent of superfaces does not affect the run-time view-dependent simplification to a great extent since they are being primarily used to limit redundant connectivity. Hence simple heuristics to define superfaces, like the one we use here, work well in practice.

5. View-Dependence Tree

View-dependence tree is a generalization of the merge tree introduced by Xia *et al* in following ways:

- View-dependence tree is capable of performing topology and geometry simplifications whereas a merge tree can only perform topology-preserving geometric simplifications.
- The construction of the view-dependence tree is based on a generalized vertex-pair collapse method to combine two vertices. In addition to the real edges, a good representative subset of virtual edges is used for performing edge collapses. This subset is more general than simply considering all virtual edges with lengths less than a threshold.

- View-dependence tree does not store dependencies lists, since it uses implicit dependencies to ensure runtime consistency in the generated triangulations. This allows highly localized memory accesses during run-time.
- It is not limited to manifold surfaces. It can handle arbitrary polygonal meshes.

In a vertex-pair collapse we define the vertices that are collapsed as the *children* and the newly created vertex as the *parent*. Each node of the view-dependence tree keeps vertex information and pointers to (a) its two children, (b) its parent, and (c) two adjacent triangle lists: *permanent* and *current*. The *permanent adjacent triangle* (PAT) list holds pointers to the triangles that are removed after the collapse of the vertex represented by this node. The *current adjacent triangle* (CAT) list exists only when the node is active and it holds a list of pointers to the current adjacent triangles. Each item $(t : s)$ of the PAT list consists of a pointer to a triangle t and an offset s. The offset s is the index of the triangle t on the CAT list of this node before the collapse took place. This is illustrated in Figure 2. We shall refer to a view-dependence-tree node as *node* and a split or collapse of the vertex represented by a node as the split or collapse of the node.

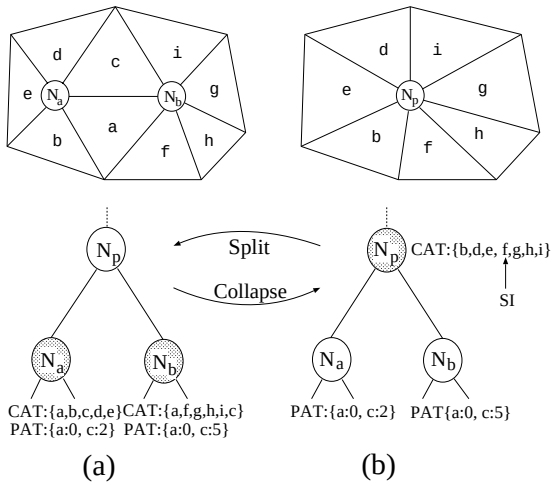


Figure 2: *Adjacent triangle update after split and collapse*

The vertices of the full-resolution mesh are represented by the leaves of the view-dependence tree. In the initialization step of constructing the view-dependence tree, we store all real and virtual edges in a priority heap. The priority heap is based on the distance between the two vertices of the edge with respect to some distance metric (shortest-distance-first). Then while the priority heap is not empty we perform the following: (a) we remove the edge on the top of the priority heap, (b) we test whether the two vertices of the removed edge can be safely collapsed, and (c) if they can be safely collapsed we execute the collapse operation.

The execution of a vertex-pair collapse involves creating a new node (parent), and moving the adjacent triangles that have become degenerate from the two children's CAT lists to their respective PAT lists. The parent's CAT list is formed by appending the CAT list of the left child to the CAT list of the right child. Note that the triangles that have just become degenerate and are represented by the PAT lists of the two child nodes are not present in the parent's CAT list. We then mark the position of the start of the right-child CAT list in the parent's CAT list by a *split index* (SI). This is illustrated in Figure 2. We note that in some cases the PAT list can have more than two triangles. At the time of creation of a new node, we also store with it the distance between the vertex-pair used in the collapse. We refer to this distance as the *switch-value* of the node. This distance can be computed using any reasonable distance metric. In our current implementation we use the cubic spline metric introduced in section 7. These switch values are used to determine the level of detail in the view-dependence tree at run time.

5.1. Preventing Foldovers

As a result of edge or vertex-pair collapses a triangle may "fold back" on itself or change its normal by about π. We refer to this as a *mesh fold-over* or just a *foldover*. In the construction of a view-dependence tree we would like to prevent foldovers and long sliver triangles. We define a vertex-pair collapse as *safe* if it does not lead to any foldovers or long sliver triangles. To determine the safety of a vertex-pair collapse we use two heuristics. First, for any triangle adjacent to any of the two collapsed vertices, the difference between the normal of this triangle before and after the collapse is bounded by some user-specified threshold. Second, for any triangle adjacent to any of the two collapsed vertices the quality of this triangle should not drop below some user-determined threshold. We quantify the quality of a triangle with area a and lengths of the three sides $l_0, l_1,$ and l_2 to be $\frac{4\sqrt{3}a}{l_0^2+l_1^2+l_2^2}$ as proposed by Guéziec [12]. We exclude from these tests triangles which become degenerate after a vertex-pair collapse such as the two triangles adjacent to the collapsed edge.

Since we use Delaunay tetrahedralizations to construct virtual edges it might happen that a virtual edge is piercing a face of the original mesh. A collapse of such an edge can produce undetected self-intersection. To detect such self-intersections a global search is required after each collapse. However, such self-intersections are limited in our case because we use superfaces and the relative difference between normals of the collapsing vertices. In a test we conducted on the Auxiliary Machine Room dataset we found that only 0.005% of the collapses resulted in such self-intersections.

The safety of a collapse or split can be lost during the traversal of the vertex hierarchy tree at run-time. Figure 3 shows an example of how an undesirable folding in the adaptive mesh can arise even though all the vertex-pair collapses that were determined statically were correct. Figure 3(a) shows the initial state of the mesh. During constructing the view-dependence tree, we first collapsed vertex v_b to v_a and then we collapsed vertex v_c to v_d. Now suppose at run-time we determined that we needed to display vertices v_a, v_b, and v_d and could possibly collapse vertex v_c to v_d, then this will lead to a foldover (as shown in Figure 3(b)).

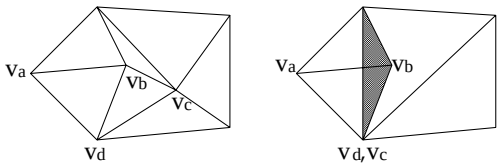


Figure 3: *Foldover case*

In order to prevent foldovers at run-time, previous algorithms [24, 16] have used *explicit dependencies* amongst the nodes of the vertex-hierarchy tree. Let us define the *region of influence* as the set of triangles which are adjacent to one of the collapsed vertices. We define the *collapse boundary* to be the set of vertices that form the boundary of the region of influence of the collapsed vertices. Explicit dependencies permit the collapse of an edge *e* only when all the vertices defining the boundary of the region of influence of this collapse exist and are adjacent to the edge *e*. As an example, consider Figure 1. Vertex v_a can collapse with vertex v_b only when the vertices $v_0, v_1, \ldots, v_k$ exist and are adjacent to v_a and v_b. Hence, the explicit dependencies can be stated as in Definition 1.

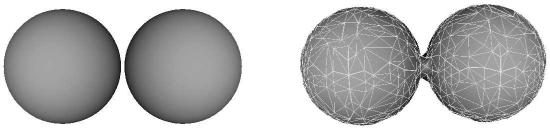
Definition 1 *Explicit dependencies* for the collapse of two vertices v_a and v_b in Figure 1.

i. *Collapse:* Vertex v_a can collapse to vertex v_b, only when all the vertices v_0, v_1, ..., v_k are present as neighbors of vertices v_a and v_b.
ii. *Split:* Vertex v_p can safely split to vertices v_a and v_b only if vertices $v_0, v_1, \ldots, v_k$ are present and adjacent to v_p.

In our view-dependence tree we use implicit dependencies (discussed in section 6) to prevent foldovers at run time.

5.2. Component Merge

One of the advantages of topology simplification is the ability to connect previously unconnected components. In our algorithm this connection is established through the collapse of virtual edges. If these two components are connected through one vertex only (one virtual edge collapse), further collapses of real edges will not increase the shared area between these components. However, if the two components share three vertices that form a triangle, then we remove the two identical triangles (since they face each other) and thereby establish a *tunnel* between these two components.



(a) Original (b) Simplified

Figure 4: *Tunnel established between two spheres*

Establishing such a tunnel enables simplification of the topology as well as reduces the number of triangles of the model. For example, the two spheres in Figure 4(a) are close enough that a tunnel is established between them as shown in Figure 4(b). The two spheres connected with a tunnel have the topology of a single sphere which is different from the topology of two unconnected spheres as well as for two spheres that just share a few vertices.

Some genus simplifying algorithms in the past [7] have proposed eliminating the interior triangles after genus simplification. The detection of such triangles is based on a region-growing approach that is quite efficient for generating static levels of details. However, for view-dependent simplifications with strict frame-to-frame timing constraints, such techniques are too expensive to be conducted in real-time. Therefore, in our present approach we only remove the redundant triangles that can be rapidly detected by a local search around the collapsed vertices.

6. Implicit Dependencies

Previous work on view-dependent simplification has used explicit dependencies to prevent foldovers at run-time. These constraints result in memory overhead and several non-local memory accesses during testing of the possibility for split or collapse.

We next propose the concept of *implicit dependencies*. Implicit dependencies rely on the enumeration of vertices generated after each collapse. If the model has *n* vertices at the highest level of detail they are assigned vertex-ids $0, 1, \ldots, n-1$. Every time a vertex pair is collapsed to generate a new vertex, the id of the new vertex is assigned to be one more than the greatest vertex-id thus far. This process is continued till the entire view-dependence tree has been constructed.

Before split or collapse operation is executed at runtime we make a few simple tests based on vertex ids to ensure the consistency of the generated triangulations and to avoid mesh foldovers. These tests are given in Definition 2.

Definition 2 *Implicit Dependencies Tests*

i. *Vertex-Pair Collapse:* A vertex-pair (a, b) can be collapsed if the vertex-id of their parent is less than the vertex-ids of the parents of the collapsed boundary vertices.
ii. *Vertex Split:* A vertex *p* can be safely split at runtime if its vertex-id is greater than the vertex-ids of all its neighbors.

Consider the example in Figure 5. Figure 5(a) shows the original mesh, Figures 5 (b), (c), and (d) show the sequence of the collapses (v_9, v_{10}), (v_7, v_8), and (v_{11}, v_{12}) respectively. The explicit dependencies list for the collapse (v_7, v_8) is $E(v_7, v_8) = \{v_1, v_2, v_3, v_4, v_{11}\}$. This means that the nodes adjacent to v_7 and v_8 must be exactly the elements of the list $E(v_7, v_8)$ before the collapse (v_7, v_8). When using implicit dependencies, we rephrase the above as: the collapse (v_7, v_8) can occur only after the collapse of (v_9, v_{10}) to v_{11} and before the collapse of any of the vertices $\{v_1, v_2, v_3, v_4, v_{11}\}$. In terms of

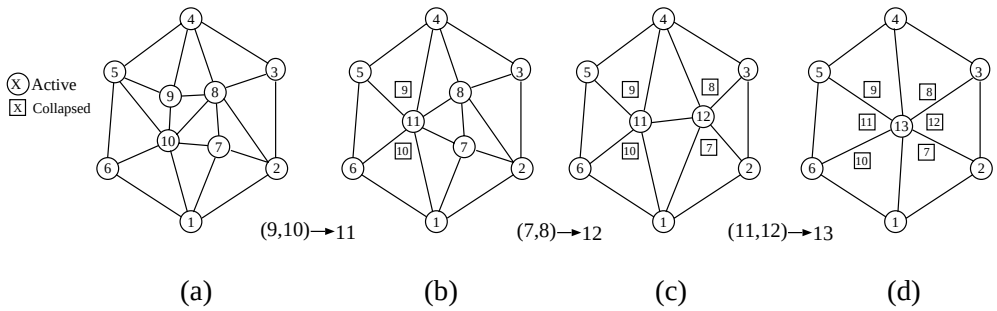


Figure 5: *Explicit dependencies and implicit dependencies*

carrying out the implicit dependency tests, we note that the vertex ids of the parents of the collapse boundary for (v_7, v_8) are all going to be ≥ 12 (since (v_7, v_8) collapses to v_{12}). Therefore, by Definition 2(i), (v_7, v_8) collapse should occur before any of its other collapse boundary vertices. Similarly, split of node v_{12} has to occur before the split of node v_{11}.

In our current implementation of implicit dependencies we store two integers with each node which are (i) the maximum vertex-id M of the adjacent vertices and (ii) the minimum vertex-id m of the parents of the collapse boundary vertices. The two integers are updated after each change of the collapse boundary as a result of split or collapse. As an example in Figure 5, the integer pair (M, m) for nodes v_7 as well as v_8 for the mesh state shown in Figure 5(a) is $(10, 11)$ while it is $(11, 13)$ for Figure 5(b). The readers might wish to verify for themselves the applicability of the implicit dependency tests given above for various split and collapse operations.

Before we proceed with the proof of the correctness of the implicit dependencies let us define some terms that help in understanding this proof.

Definition 3 Two collapse boundaries are *adjacent* if a vertex of one vertex-pair collapse exists in the other collapse boundary, otherwise they are *disjoint*.

Definition 4 A view-dependence tree is called *foldover-free* if during its construction no executed collapse leads to a foldover. A sequence of vertex-pair collapses and vertex split operations is called a *foldover-free sequence* if no foldover occurs during the execution of this sequence.

Definition 5 A sequence of vertex-pair collapses and vertex split operations on a view-dependence tree T *preserves the local order of the collapses* if every two adjacent collapses are carried out in the same order that they were executed during the construction of T.

Lemma 1 *For a given view-dependence tree, implicit dependencies preserve the local order of the collapses, which means that any two adjacent collapses occur in the same order as in the construction sequence.*

Proof: Note that there is a one-to-one correspondence between the enumeration of the newly created nodes and the collapses during the construction of the view-dependence tree. For example, in the collapse $C_\alpha : (v_i, v_j)$ to create the new node v_p, the *id* of the node v_p is equal to $n + \alpha$, where n is the number of leaves of the view-dependence tree (which is the number of vertices of the dataset) and α is the number of collapses before C_α. Thus, Definition 2(i) is equivalent to "*Select the collapse with the minimum id among the set of the given adjacent collapses* ", but this statement imposes a local order on the adjacent collapses. Following the same terminology, Definition 2(ii) can be stated as "*Select the collapse with the maximum id among the set of the given adjacent collapses*", but again this imposes a local order on the collapses. These local orders are the same as in the construction sequence of the view-dependence tree by the way the node ids are assigned. □

Theorem 1 *For any foldover-free tree, a sequence of vertex-pair collapses that preserves the local order of the collapses is a foldover-free sequence.*

Proof: Let us assume by contradiction that a foldover occurs in a foldover-free tree even though the collapse sequence preserves the local order of the collapses. Let $C_f : (v_b, v_a)$ be the first collapse which leads to a foldover. Now, we proceed with the proof in two stages. In the first stage, we extend the sequence of vertex-pair collapses such that every collapse in the range $C_0, \ldots, C_f$ exists in the sequence. We achieve this by executing the missing collapses in that range, without violating the implicit dependencies. The execution of such collapses is achievable, since it is always possible to execute the collapse with the minimum index. However, during the execution of the missing collapses we may encounter other foldovers. In such cases, we rename the one with the lowest index to C_f.

In the second stage, we sort the complete sequence of collapses $C_0, C_1, \ldots, C_f$, with respect to their index order. We carry out this sort by only exchanging pair of consecutive collapses which are not in an index order (such as C_i, C_j and $i > j$). Now, we show that this exchange preserves the implicit dependencies. C_i and C_j can be either adjacent or disjoint collapses. However such an adjacent pair can not exist because it contradicts the implicit dependencies. If the two collapses are

disjoint, then they can not affect each other by the definition of collapse boundary.

The resulting sequence $C_0, C_1, \ldots, C_f$ is a prefix of the construction collapse sequence, which means that this foldover occurs during the construction too. But the given view dependence tree was foldover-free. This contradiction proves the theorem. □

In the above proofs we have only dealt with the sequences of collapses. If we treat a split operation as undoing a collapse and removing the corresponding collapse from the sequence, it suffices to show that the above proofs cover both splits and collapses. Theorem 1 proves that just maintaining the local order of the collapses ensures a foldover-free mesh and Lemma 1 proves that the implicit dependencies maintain the local order of the collapses. Thus, we can say that the implicit dependencies suffice to ensure a foldover-free mesh.

7. Spline-Based Distance Metric

Three-dimensional curves and surfaces play an important role in design and manufacture of various products. Since it is faster to render triangulated surfaces, most of these surfaces are triangulated before rendering. The triangulated datasets can be classified as the following main types with respect to their acquisition: (a) smooth surfaces such as parametric CAD/CAM (b) range data such as terrains and laser-scanned data, and (c) hand-digitized or designed polygonal data.

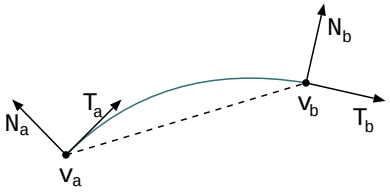


Figure 6: *Hermite Cubic Curve*

Automatic creation of multiresolution hierarchies is a crucial first-step in any level-of-detail-based rendering system. In our approach we use vertex-pair collapses to reduce the complexity of the dataset. It is crucial to find a metric that well approximates the distance between two vertices along the surface and gives a "good" measure of the distance between two unconnected vertices. Since the two vertices may have represented two points on a continuous surface that was sampled at an acceptable rate, splines enable faithful reconstruction of this surface with respect to the sampling rate. A cubic-spline curve constructed using the normal and the coordinates of the two vertices results in a good approximation of the curve that passes through these two vertices along the "original surface".

In order to keep the model as close as possible to its original appearance, we need to carry out the collapses in the order which introduces the least error first. We determine this order by the length of the cubic curve that connects these two vertices. Cubic curves approximate the distances better in polygonal datasets that represent smooth surfaces. The cubic curve is determined by the position and the normal of the two vertices as depicted in Figure 6.

$$P(t) = \sum_{i=0}^{3} H_i t^i \qquad t_a \le t \le t_b \tag{1}$$

$$Length(P(t), v_a, v_b) = \int_{t_a}^{t_b} \left\| \frac{\partial P(t)}{\partial t} \right\| dt \tag{2}$$

The error introduced as a result of the collapse of two vertices is affected by the position of the new vertex (the collapse result). Hence, it important to select the position that minimizes this error. Our spline metric relies on the fact that the mesh represents a smooth surface and tries to keep the updated triangles as close as possible to the "smooth surface". We have found that selecting the position of the resulting vertex c, to be the the point that has the average of the tangents T_a and T_b usually minimizes the distance between the two vertices and the resulting curve as shown in Figure 7.

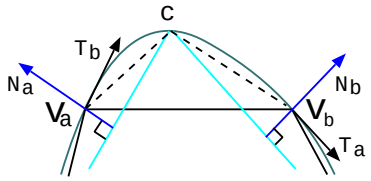


Figure 7: *Computing the position of the collapsed vertex*

The spline metric performs well on models that do not have sharp edges. The ability to combine the normal and the vertex position makes it suitable for datasets that represent smooth surfaces. It is easy to see that for smooth surfaces, a spline-based metric that allows new collapsed vertices to lie above or below the line joining the child vertices will work better than a metric that forces the collapsed vertex to lie on the collapsing (real or virtual) edge as shown in Figure 8.

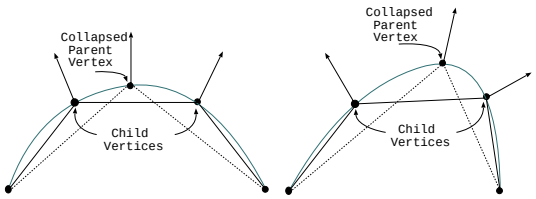
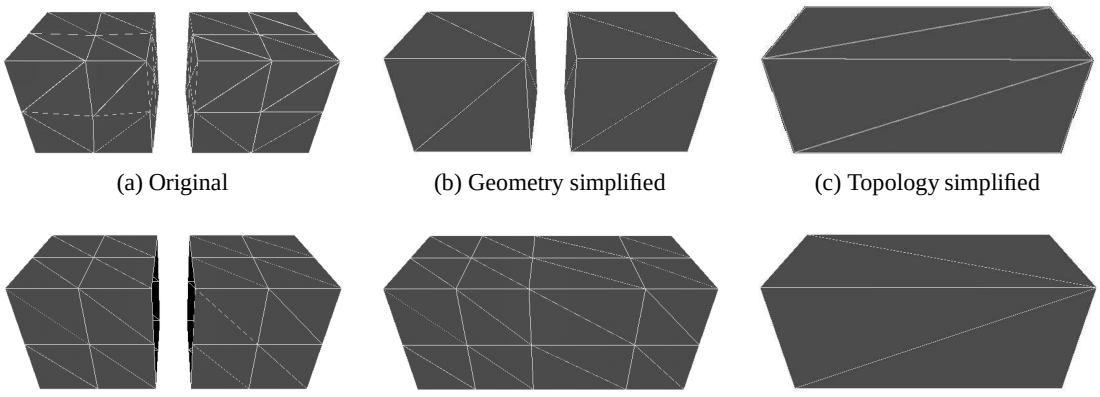


Figure 8: *Effect of normals on the position of the collapsed vertex*

As shown in Figure 9, The cubic-spline-based metric results in different simplifications when two closed boxes are placed near each other as shown in Figure 9(a)–(c) as compared to when two boxes with their closest faces missing are placed near each other as shown in Figure 9(d)–(f). As can be seen in first case the geometry gets simplified before the topology whereas in the second case the topology is simplified by



(a) Original (b) Geometry simplified (c) Topology simplified

(d) Original (e) Topology simplified (f) Geometry simplified

Figure 9: *Simplifying Closed and Open Boxes with Cubic-spline-based Metric*

connecting the two boxes and then the geometry is simplified. This is because the cubic-spline-based metric takes into account the normals of the vertices in addition to their position coordinates.

We use Hermite interpolation to compute the parametric cubic curve as in equation (1), where H_i is determined by the positions and the normals of the two vertices. To compute the length of the curve $P(t)$ that connects the two vertices we use equation (2). We can simplify this distance function in two ways. First, one can approximate the curve length by a sequence of straight lines, which are determined by using the de Boor algorithm [8]. Second, one can analytically simplify equation 2. We have adopted the latter in our implementation; details are in section 7.2).

7.1. Choosing Tangents

Hermite interpolation requires a tangent at each of the two vertices. We compute the tangents at the two vertices using the normals at these vertices. Any vector perpendicular to the vertex normal is a valid tangent at that vertex. Among these infinite number of tangents we pick two for each curve (one from each vertex) as follows: for vertex v_a we choose the tangent that points to the other vertex v_b. For vertex v_b we choose the tangent that points in the direction opposite that of the vertex v_a. The point c (shown in Figure 10) is the intersection of the tangent T_a and the extension of the tangent T_b. The point c is the closest point to v_a and v_b that lies on the intersection of the two planes defined by the two points v_a and v_b and the two normals N_a and N_b, respectively, as shown in Figure 10. A special case arises when the angle between the two normals is π, then any two opposite tangents are valid.

The lengths of the tangent vectors at the two vertices partly determine the behavior of the spline curve passing through these two vertices. Hence, if we do not compute the tangent vector lengths carefully, we may not get the desired spline. For example, as shown in Figure 11, the spline between the two vertices v_a, v_b can have, a *loop*, a *cusp*, or two *inflection points* respectively [3]. We can eliminate the loop and cusp

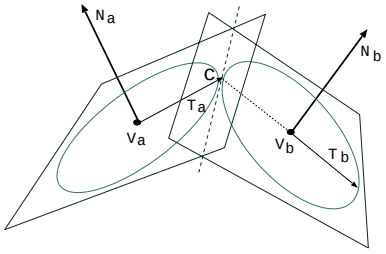


Figure 10: *The curve tangents are computed as the first intersection of the two cylinders*

singularities and unwanted inflection points by changing the magnitudes of the tangent vectors. Su and Liu [3] have found a bound on the length of the tangent that guarantees the elimination of singularities and unwanted inflection points. In the special setting of our spline, assigning the length of the tangent to be less than the Euclidean distance between the two vertices v_a, v_b guarantees the generation of a spline curve without singularities and unwanted inflection points.

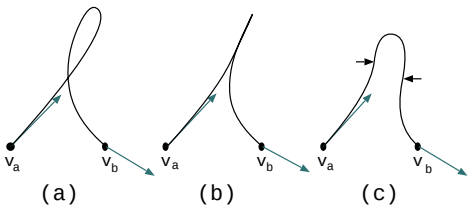


Figure 11: *Curves (a) and (b) have singularities, curve (c) has unwanted inflection points*

7.2. Analytical Approximation

We now explain the analytical approximation for curve length in three-dimensional space, which is derived from (2) and can be written as in equation (3).

$$Length((P(t), v_a, v_b) = \int_{t_a}^{t_b} \sqrt{\dot{X}(t)^2 + \dot{Y}(t)^2 + \dot{Z}(t)^2} dt \quad (3)$$

Since the length of the curve imposes a complete order on the vertex pairs set, we will get the same order if we remove the square root of integral in (3). For spline curve we can always reparameterize t to run from $[0..1]$ instead of $[t_a..t_b]$. After applying these these two changes on equation (3) we get equation (4).

$$Length((P(t), v_a, v_b) = \int_0^1 (\dot{X}(t)^2 + \dot{Y}(t)^2 + \dot{Z}(t)^2) dt \quad (4)$$

The functions $X(t), Y(t)$,and $Z(t)$ are cubic polynomials, hence we can write these functions in a form similar to equation (5) which represent $X(t)$.

$$X(t) = a_x t^3 + b_x t^2 + c_x t + d_x \quad (5)$$

Since the three equations are similar, it is enough to show only the solution of $\int_0^1 \dot{X}(t)^2 dt$, which appears in (6).

$$\begin{aligned} \int_0^1 \dot{X}(t)^2 dt &= \int_0^1 (3a_x t^2 + 2b_x t + c_x)^2 dt \\ &= a(\frac{9}{5}a + 3b) + c(2a + c) + b(\frac{4}{3}b + 2c) \end{aligned} \quad (6)$$

As we see from equation (6), it is fast to compute the length of the curve and the metric derived from it.

8. Real-Time Traversal

The view-dependence tree which is constructed off-line is used at run-time to construct an adaptive level-of-detail mesh representation. In fact, the view-dependence tree is a forest (set of trees) since some nodes can not merge together to form one tree. The view-dependence tree is able to adapt to various levels of detail. Coarse details are associated with nodes that are close to the top of the tree and high details are associated with the nodes that are close to the bottom of the tree as shown in Figure 13. The reconstruction of a real-time adaptive mesh requires the determination of the list of vertices of this adaptive mesh and the list of triangles that connect these vertices. We shall refer to these lists as the list of *active nodes* and the list of *active triangles*.

8.1. Refinement Metric

The light and view parameters determine the level of detail at each region of the scene. We use the spline metric we have introduced in section 7 to construct the view-dependence tree as well as to use it for real-time generation of these levels of detail at run time. Previous approaches have used different distance metrics for the off-line processing and for run-time refinement and simplification. We next discuss how the same spline metric can be used to not only determine the distances amongst vertices but also can be used to determine distances to the light source as well as to the view point.

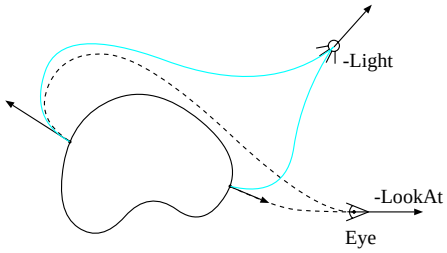


Figure 12: *View and light spline curves*

The level of detail at a vertex v based on the view parameters is computed using the cubic spline that connects the eye position with the vertex v. Here, the normal of the vertex v and the negative look-at vector are used as tangents. The light contribution to the level of detail at the vertex v is similarly computed by using the vertex normal and the negative of the light direction as tangents (as shown in Figure 12). In both of these distance measurements, short curve lengths are associated with high detail and long ones associated with low detail. Note that dark regions (not facing the light) are not displayed in the highest detail even if they are close to the viewer since the light curve is long. As we see in Figure 12 the cubic-spline curves are short for front-facing regions close to the viewer and long for far-away regions that are back-facing. When the normal of a vertex v falls into the silhouette cone, we treat v as special case by displaying it in higher detail.

8.2. Active Nodes

The list of active vertices is a subset of the nodes of the view-dependence tree and is determined by:

- Eye parameters, such as eye position and look-at direction.
- Light Parameters, such as position and direction.
- Distance metric function which determines the level of detail at each vertex.

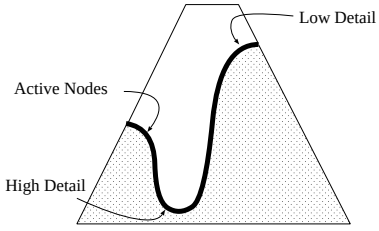


Figure 13: *Active vertices list*

At each frame the set of active nodes is traversed and for each node we use the distance metric to compute a metric value. This metric value represents the distance to the viewer, the light source, and the local geometry. We then compare the metric value at a node with the switch value stored at that node

to determine the next operation to execute. We have discussed how to compute the switch value at a node in section 5. If the metric value is less than the switch value and this node satisfies the implicit dependency conditions for split, we split this node into its two children. If the computed metric value is larger than the switch value stored at the parent of this node and its sibling can collapse, we collapse this node and its sibling. Otherwise, this node stays in the active nodes list.

When a node n can not split as result of the implicit dependency condition, two approaches are possible. In the first approach, we recursively split the nodes which prevent the split of node n (if possible) and then carry out the split of n. In the second approach, we leave node n for the later frames (until the nodes that prevent n's split have split). We have found that the second approach can deliver better frame rates, since the first approach takes longer to update the list of active nodes and triangles. Similar observations have been reported by Hoppe [16] for slowly changing view-parameters.

The split operation involves removing the node from the active nodes list and inserting its two children into this list. The active nodes list is updated after a collapse by replacing the parent of the two nodes with one of them and removing the pointer to the other node.

8.3. Active Triangles

We maintain a global list of all the triangles of the dataset. Each triangle is represented by three pointers to its vertices and its normal. The active triangle list is a list of pointers to the entries in the global triangle list. The adjacent triangle lists (CAT and PAT) in the tree nodes are also represented by a list of pointers to the global triangle list.

As we mentioned earlier, our algorithm relies on the coherence between frames. We assume that the list of active triangles at frame i is given. To determine the list of triangles at frame $i+1$ we either add, remove, or update triangles depending on the split or collapse operation to move from frame i to frame $i+1$. The update of the triangle lists during a vertex-pair collapse is done in two steps. First, we subtract the PAT list at each vertex from the CAT list. Second, we append the resulting CAT list at the right child to the resulting CAT list at the left child to form the CAT list of the parent node. During the append of the two list we assign the *split index* at the parent node to point to the first element of the right child CAT list. The vertex split operation is also carried out in two steps. First, we split the parent's CAT list at the *split index* position into two lists, left list and right list. Second, we merge the left node's PAT list with the left list to form the left node's CAT list. Similarly we merge the right node's PAT list with the right list to form the right node's CAT list. Note that we use the offset of the triangle in the PAT list to perform the correct merge.

9. Results and Discussion

We have implemented our algorithm in C++ on an SGI Onyx2 with Infinite Reality graphics. We have tested our algorithm on several datasets and have received encouraging results for our non-optimized implementation. The times for our preprocessing appear in Table 1. As can be seen our method to compute the virtual edges limits them to be linear with respect to the number of real edges of the model. For these results we have used the normal difference angle for generation of the superfaces to be less than 75^o.

The model tricycle in Figure 14 consists of multiple unconnected components. Figure 14 shows the different levels of detail of the tricycle as the unconnected components merge and the number of triangles is reduced. Figure 15 shows the result of simplifying a hole on the back-facing region of the sphere. Figure 15(a) shows the original high-detail model of the sphere showing a hole. Figures 15(b) and (c) show one view and its opposite to illustrate the simplifying away of the hole in the region facing away from the viewer. Figure 15(d) shows the side-view of the mesh in wire-frame to illustrate the different levels of detail and the simplifying away of the hole. Using our spline metric we were able to achieve high detail at regions that are close to the viewer and front-facing and low detail at regions that are far from the viewer or back-facing. Figure 16 shows the results of view-frustum-based simplification. The objects outside of the view-frustum are simplified to zero triangles. We test very few nodes of the view-dependence tree to reduce the objects lying outside the view-frustum to low details. Figure 17(a) shows the Auxiliary Machine Room(AMR) of a notional submarine dataset from the Electric Boat Corporation. Figure 17(a) shows the viewer position for Figure 17(b) by the location A and for Figure 17(c) by the location Z. For a given error threshold that produces almost visually indistinguishable images we have found that our generalized view-dependent method produces more aggressive simplification than topology-preserving view-dependent algorithm. For instance, AMR model we were able to achieve 76K triangles using the geometry and topology simplification compared with 91K triangles for the geometry simplification, for the same error bound and viewer position.

Table 2 presents the space requirements for the View-dependence tree that we propose in this paper and the Merge tree. The *adjacency* column indicates the space required to store the adjacency information for all the nodes including the space for the explicit or the implicit dependencies. The *total* column indicates the total space requirements for the entire vertex hierarchy. As can be seen the nodes of the view-dependence tree require approximately half as much space as that for the merge tree. We would like to point out here that for the purposes of comparison the view-dependence tree's space requirements do not represent the space for virtual edge collapses since nothing equivalent exists for the merge tree. We have compared the total amount of memory required for the merge trees [24] and the view-dependence trees as proposed in this paper. For the example of the Buddha dataset, we have found that the merge tree requires 32.1MB of memory while the view-dependence tree requires 22.3MB. In this comparison we had disabled the presence of virtual edges in the view-dependence tree since none exist in the merge tree.

Table 3 demonstrates the range of pointer accesses with explicit dependencies including the average distance of access

Dataset	Vertices	Triangles	Real Edges	Vertices after Superfaces	Virtual Edges	Preprocessing Time(sec)		
						View-Dep. Tree	Superfaces	Qhull
Tricycle	7K	14K	21K	2K	9K	2.1	0.7	2.2
Buddha	145K	293K	435K	29K	126K	51.0	24.9	26.3
Tricycle Lot	178K	340K	517K	55K	217K	84.6	26.9	63.4
AMR	195K	381K	572K	31K	152K	87.0	28.1	83.8
Torpedo Room	465K	737K	1,288K	91K	291K	222.8	42.7	96.2
Dragon	437K	871K	1,309K	82K	336K	273.7	46.4	79.7

Table 1: *Preprocessing times for different datasets*

Dataset	Number of Nodes	View-Dependence Tree(KB)		Merge Tree(KB)		Savings Factor	
		Adjacency	Total	Adjacency	Total	Adjacency	Total
Tricycle	14,086	109	616	626	1,133	5.74	1.83
Buddha	289,265	2,345	12,758	14,788	25,201	6.30	1.97
Tricycle Lot	352,098	2,733	15,408	15,692	28,367	5.74	1.84
AMR	337,791	2,728	14,888	15,373	27,533	5.63	1.84
Torpedo Room	823,761	6,653	36,308	37,489	67,144	5.63	1.84
Dragon	874,927	6,972	38,469	46,906	78,403	6.72	2.03

Table 2: *View-Dependence Tree size vs Merge Tree size*

as well as the standard deviation. This has serious implications for external memory algorithms as well as for performing view-dependent simplifications over networks. If we assume a normal bell-shaped distribution, to get over 95% of the hits in a block we will need a block size to be $\mu + 2\sigma$ to just cover the accesses required to decide on the split/collapse for a *single* node. For a dataset such as the Dragon, this implies that we might need a block of size 21*MB*. This is particularly large if we note that the entire tree for the Dragon dataset is about 38.5 MB. The implicit dependencies on the other hand, since they require only local accesses within the node at run time, do not suffer from this drawback.

We have observed several advantages in using our generalized view-dependent simplification algorithm in terms of memory savings, localized access, and better simplification.

- *Memory Savings:* Implicit dependencies require only two integers to store instead of a list of adjacent triangle pointers in explicit dependencies. In addition our view-dependence tree does not duplicate triangle pointers.
- *Localized Access:* Explicit dependencies in previous work [24, 16] need to visit every neighbor of an active node to test whether it can split or collapse. This may result in non-local accesses causing unnecessary paging (when the dataset is larger that the local memory).
- *Better Simplification:* Our algorithm is able to simplify geometry and topology in a view-dependent manner. It can achieve aggressive simplification for objects that are far from the viewer, are backfacing, or are out of the view frustum. In addition, our algorithm can deal with non-manifold representations. Our spline-based distance metric works in a natural and intuitive manner, especially for smooth surfaces.

Our algorithm accepts arbitrary polygonal datasets. In addition, it is able to achieve better simplification than topology preserving algorithms for models with non-manifold cases such as cracks and T-junctions. However, our algorithm does simplify all the T-junctions, since that requires a connectivity between vertices and edges.

Dataset	Number of Nodes	Average Distance (μ)(MB)	Standard Deviation (σ)(MB)
Tricycle	14,086	0.16	0.11
Buddha	289,265	4.32	2.12
Tricycle Lot	352,098	3.95	3.15
AMR	337,791	3.70	3.20
Torpedo Room	823,761	13.74	6.93
Dragon	874,927	14.26	7.10

Table 3: *Non-Local Access with Explicit Dependencies*

10. Conclusions and Future Work

We have presented the concept of view-dependence trees as a tool to perform geometry and topology simplification for large polygonal datasets. These trees use implicit dependencies and efficient node representation that make them more compact, faster to navigate, and allow them to accept general non-manifold datasets. Further, we have also introduced a spline-based distance metric that can incorporate both the vertex normals and positions in a reasonably intuitive way.

We see the scope for future work in designing external

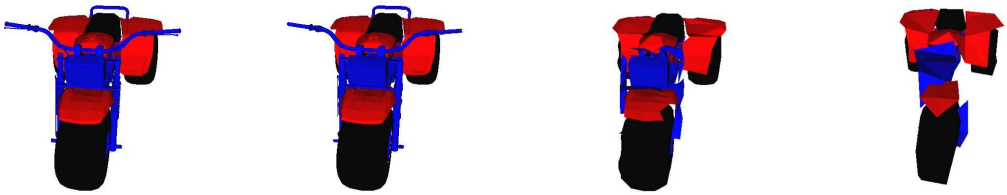
memory algorithms for visualization of datasets whose sizes exceed that of the main memory and collaborative visualization of large dataset that reside in a local or remote server, by taking advantage of the localized and compact structure of view-dependence trees.

Acknowledgements

This work has been supported in part by the NSF grants: CCR-9502239, DMI-9800690, ACR-9812572 and a DURIP instrumentation award N00014970362. Jihad El-Sana has been supported in part by the Fulbright/Arab-Israeli Scholarship and the Catacosinos Fellowship for Excellence in Computer Science. The figure 17 shows the Auxiliary Machine Room part from the dataset of a notional submarine provided to us by the Electric Boat Division of General Dynamics. We would like to thank the reviewers for their insightful comments which led to several improvements in the presentation of this paper.

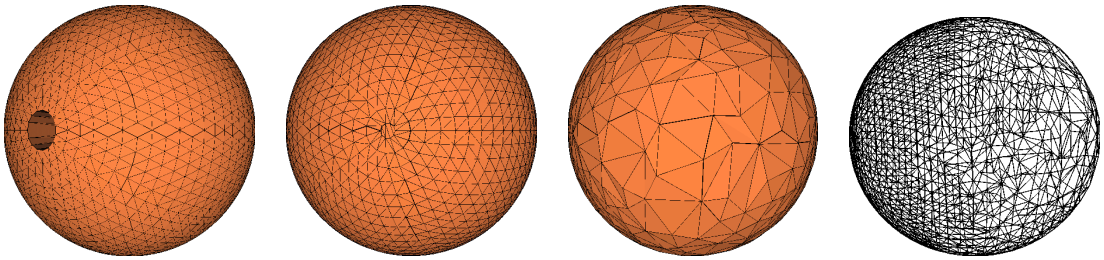
References

1. N. Amenta, M. Bern, and M. Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *Prceedings of SIGGRAPH 98(Orlando, Florida, July 19–24, 1998)*, pages 415–421, 1998.
2. C. B. Barber and H. Huhdanpaa. Qhull 2.6, 1998. http://www.geom.umn.edu/software/qhull.
3. S. Bu-qing and L. Dingyuan. *Computational Geometry-Curve and Surface Modeling*. Academic Press Inc., 1989.
4. B. Chazelle, D. Dobkin, N. Shouraboura, and A. Tal. strategies for polyhedral surface decomposition: An experimental study. *Comput. Geom. Theory Appl.*, 7:327–342, 1997.
5. P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, February 1998. ISSN 0097-8493.
6. L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulation. In H. Rushmeier D. Elbert and H. Hagen, editors, *Proceedings Visualization '98*, pages 43–50, October 1998.
7. J. El-Sana and A. Varshney. Topology simplification for polygonal virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 4, No. 2:133–144, 1998.
8. G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press, San Diego, California, third edition, 1993.
9. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, pages 209 – 216. ACM SIGGRAPH, ACM Press, August 1997.
10. Tran S. Gieng, Bernd Hamann, Kenneth L. Joy, Gregory L. Schussman, and Isaac J. Trotts. Constructing hierarchies for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 4(2), 1998.
11. M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.
12. A. Guéziec. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery, MRCAS '95*, 1995.
13. A. Gueziec, F. Lazarus, G. Taubin, and W. Horn. Surface partitions for progressive transmission and display, and dynamic simplification of polygonal surfaces. In *Proceedings VRML 98, Monterey, California, February 16–19*, pages 25–32, 1998.
14. T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.
15. H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.
16. H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, pages 189 – 197. ACM SIGGRAPH, ACM Press, August 1997.
17. A. D. Kalvin and R. H. Taylor. Superfaces: Polyhedral approximation with bounded error. *IEEE Computer Graphics and Applications*, 16(3):67–77, May 1996.
18. R. Klein, A. Schilling, and W. Straßer. Illumination dependent refinement of multiresolution meshes. In *Computer Graphics International*, June 1998.
19. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, pages 198 – 208. ACM SIGGRAPH, ACM Press, August 1997.
20. J. Popović and H. Hoppe. Progressive simplicial complexes. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, pages 217 – 224. ACM SIGGRAPH, ACM Press, August 1997.
21. J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.
22. A. Schilling and R. Klein. Texture-dependent refinement for multiresolution models. In *Computer Graphics International*, June 1998.
23. W. Schroeder. A topology modifying progressive decimation algorithm. In *IEEE Visualization '97 Proceedings*, pages 205 – 212. ACM/SIGGRAPH Press, October 1997.
24. J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, pages 171 – 183, June 1997.



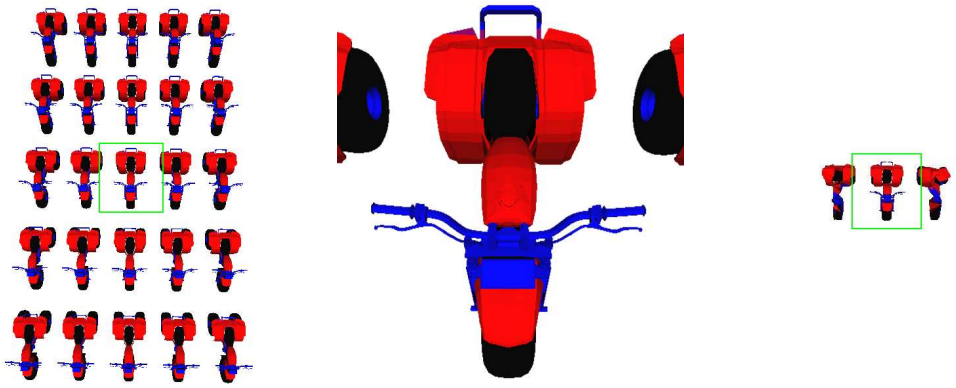
13594 Tris 6960 Tris 2032 Tris 225 Tris

Figure 14: *Different levels of detail for a tricycle model*



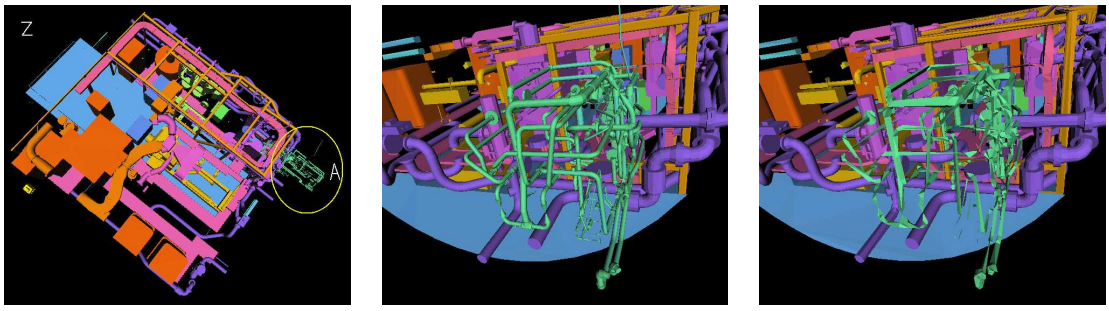
(a)Original (b) Front-Facing (c) Back-Facing (c) Side View

Figure 15: *Hole Simplification on the back-facing region*



(a) The complete model (b) Selected View (c) After Culling

Figure 16: *View Frustum Culling Using View-Dependent Simplification*



(a) Original model (b) Viewer at A (b) Viewer at z

Figure 17: *Close and Far View of the Auxiliary Machine Room Dataset*