Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models

Julie C. Xia, Jihad El-Sana, Amitabh Varshney

Abstract— **We present an algorithm for performing adaptive real-time level-of-detail-based rendering for triangulated polygonal models. The simplifications are dependent on viewing direction, lighting, and visibility and are performed by taking advantage of image-space, object-space, and frame-to-frame coherences. In contrast to the traditional approaches of precomputing a fixed number of level-of-detail representations for a given object our approach involves statically generating a continuous level-of-detail representation for the object. This representation is then used at run-time to guide the selection of appropriate triangles for display. The list of displayed triangles is updated incrementally from one frame to the next. Our approach is more effective than the current level-of-detail-based rendering approaches for most scientific visualization applications where there are a limited number of highly complex objects that stay relatively close to the viewer. Our approach is applicable for scalar (such as distance from the viewer) as well as vector (such as normal direction) attributes.**

I. INTRODUCTION

The scientific visualization and virtual reality communities have always faced the problem that their "desirable" visualization dataset sizes are one or more orders of magnitude larger than what the hardware can display at interactive rates. Recent research on graphics acceleration for the navigation of such three-dimensional environments has been motivated by attempts to bridge the gap between the desired and the actual hardware performance, through algorithmic and software techniques. This research has involved reducing the geometric and rendering complexities of the scene by using

- statically computed level-of-detail hierarchies [10], [14], [22], [25], [31], [32], [35],
- visibility-based culling that is statically computed [1], [34] and dynamically computed [17], [18], [27],
- various levels of complexity in shading and illumination models[4],
- texture mapping [5], [6], and
- image-based rendering [7], [8], [13], [29], [33].

In this paper we will focus on reducing the geometric complexity of a three-dimensional environment by using dynamically computed level-of-detail hierarchies. Research on simplification of general three-dimensional polygonal objects (non-convex, non-terrain, possibly high genus) has spanned the entire gamut of highly local to global algorithms, with several approaches in between that have both local and global steps.

Local algorithms work by applying a set of local rules, which primarily work under some definition of a *local neighborhood*, for simplifying an object. The local rules are iteratively applied under a set of constraints and the algorithm terminates when it is no longer possible to apply the local rule without violating some constraint. The global algorithms optimize the simplification process over the whole object, and are not necessarily limited to

the small neighborhood regions on the object. Some of the local approaches have been – vertex deletion by Schroeder *et al* [32], vertex collapsing by Rossignac and Borrel [31], edge collapsing by Hoppe *et al* [26] and Guéziec [20], triangle collapsing by Hamann [21], and polygon merging by Hinker and Hansen [24]. Some of the global approaches have been – redistributing vertices over the surface by Turk [35], minimizing global energy functions by Hoppe *et al* [26], using simplification envelopes by Varshney [36] and Cohen *et al* [10], and wavelets by DeRose *et al* [14]. The issue of preservation or simplification of the genus of the object is independent of whether an algorithm uses local rules, or global rules, or both, to simplify. Recent work by He *et al* [22] provides a method to perform a controlled simplification of the genus of an object.

Simplification algorithms such as those mentioned above are iteratively applied to obtain a hierarchy of successively coarser approximations to the input object. Such multiresolution hierarchies have been used in level-of-detail-based rendering schemes to achieve higher frame update rates while maintaining good visual realism. These hierarchies usually have a number of distinct levels of detail, usually 5 to 10, for a given object. At run time, the perceptual importance of a given object in the scene is used to select its appropriate level of representation from the hierarchy [9], [11], [12], [16], [28], [30]. Thus, higher detail representations are used when the object is perceptually more important and lower detail representations are used when the object is perceptually less significant. Transitions from one level of detail to the next are typically based on simple image-space metrics such as the ratio of the image-space area of the object (usually implemented by using the projected area of the bounding box of the object) to the distance of the object from the viewer.

Previous work, as outlined above, is well-suited for virtual reality walkthroughs and flythroughs of large and complex structures with several thousands of objects. Examples of such environments include architectural buildings, airplane and submarine interiors, and factory layouts. However, for scientific visualization applications where the goal often is to visualize one or two highly detailed objects at close range, most of the previous work is not directly applicable. For instance, consider a biochemist visualizing the surface of a molecule or a physician inspecting the iso-surface of a human head extracted from a volume dataset. It is very likely during such a visualization session, that the object being visualized will not move adequately far away from the viewer to allow the rendering algorithm to switch to a lower level of detail. What is desirable in such a scenario is an algorithm that can allow several different levels of details to co-exist across different regions of the same object. Such a scheme needs to satisfy the following two important criteria:

1. It should be possible to select the appropriate levels of detail

Contact Address: Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, Email: chaoyu|jihad|varshney@cs.sunysb.edu

across different regions of the same object in real time.

2. Different levels of detail in different regions across an object should merge seamlessly with one another without introducing any cracks and other discontinuities.

In this paper we present a general scheme that can construct such seamless and adaptive level-of-detail representations on-the-fly for polygonal objects. Since these representations are view-dependent, they take advantage of view-dependent illumination, visibility, and frame-to-frame coherence to maximize visual realism and minimize the time taken to construct and draw such objects. Our approach shows how one can adaptively define such levels of detail based on (a) scalar attributes such as distance from the viewpoint and (b) vector attributes such as the direction of vertex normals. An example using our approach is shown in Figure 1.
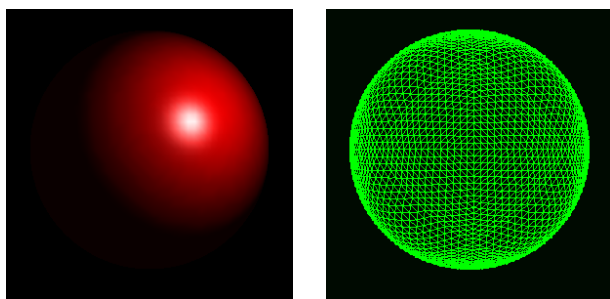
(a) Sphere with 8192 triangles (uniform LOD)
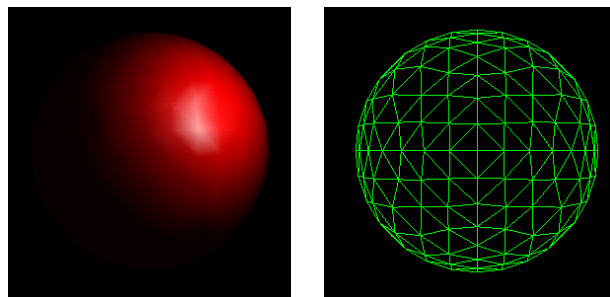
(b) Sphere with 512 triangles (uniform LOD)
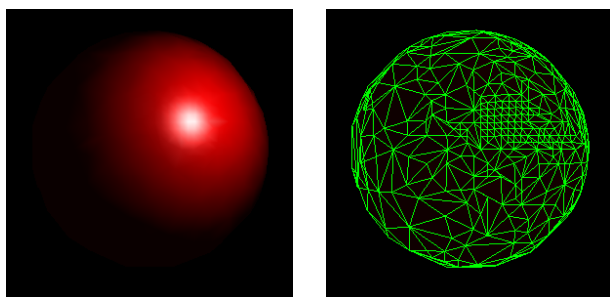
(c) Sphere with 537 triangles (adaptive LOD)

Fig. 1. Uniform and adaptive levels of detail

II. PREVIOUS WORK

Adaptive levels of detail have been used in terrains by Gross *et al* [19] by using a wavelet decomposition of the input data samples. They define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Thus, the level of detail around any region can adaptively refine in real-time. This work provides a very elegant solution for terrains and other datasets that are defined on a regular grid.

Some of the previous work in the area of general surface simplification has addressed the issue of adaptive approximation of general polygonal objects. Turk [35] and Hamann [21] have proposed curvature-guided adaptive simplification with lesser simplification in the areas of higher surface curvature. In [10], [36], adaptive surface approximation is proposed with different amounts of approximation over different regions of the object. Guéziec [20] proposes adaptive approximation by changing the tolerance volume in different regions of the object. However in all of these cases, once the level of approximation has been fixed for a given region of the object, a discrete level of detail corresponding to such an approximation is statically generated. No methods have been proposed there that allow free intermixing of different levels of detail across an object in real time in response to changing viewing directions.

Work on surface simplification using wavelets [14], [15] and progressive meshes [25] goes a step further. These methods produce a continuous level-of-detail representation for an object in contrast to a set of discrete number of levels of detail. In particular, Hoppe [25] outlines a method for selective refinement – i.e. refinement of a particular region of the object based upon view frustum, silhouette edges, and projected screen-space area of the faces. Since the work on progressive meshes by Hoppe [25] is somewhat similar to our work we overview his method next and discuss how our method extends it.

Progressive meshes offer an elegant solution for a continuous resolution representation of polygonal meshes. A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses. An edge collapse transformation and its dual, the vertex split transformation, is shown in Figure 2.
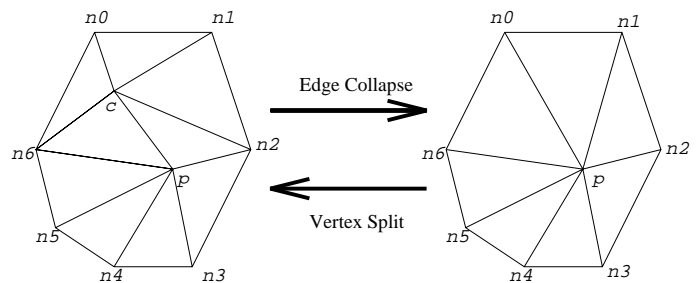

Fig. 2. Edge collapse and vertex split

Thus, a sequence of k successive edge collapse transformations yields a sequence of successively simpler meshes:

$$M^k \stackrel{collapse_{k-1}}{\rightarrow} M^{k-1} \stackrel{collapse_{k-2}}{\rightarrow} \ldots M^1 \stackrel{collapse_0}{\rightarrow} M^0 \quad (1)$$

We can retrieve the successively higher detail meshes from the simplest mesh M^0 by using a sequence of vertex-split transformations that are dual to the corresponding edge collapse transformations:

$$M^0 \stackrel{split_0}{\rightarrow} M^1 \stackrel{split_1}{\rightarrow} \ldots M^{k-1} \stackrel{split_{k-1}}{\rightarrow} (\hat{M} = M^k) \quad (2)$$

Hoppe [25] refers to $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ as a *progressive mesh* representation. Progressive meshes present a novel approach to storing, rendering, and transmitting meshes by using a continuous-resolution representation. However we feel that there is some room for improvement in adapting them for performing selective refinement in an efficient manner. In particular, following issues have not yet been addressed by progressive meshes:

1. The sequence of edge collapses is aimed at providing good approximations M^i to $(\hat{M} = M^k)$. However, if a sequence of meshes M^i are good approximations to $\hat{M}$ under some distance metric, it does not necessarily mean that they also provide a "good" sequence of edge collapse transformations for selective refinement. Let us consider a two-dimensional analogy of a simple polygon as shown in Figure 3. Assume that vertices v_0, v_6, v_7, and v_8 are "important" vertices (under say some perceptual criteria) and can not be deleted. An approach that generates approximations based on minimizing distances to the original polygon will collapse vertices in the order $v_1 \rightarrow v_2, v_2 \rightarrow v_3, v_3 \rightarrow v_4, v_4 \rightarrow v_5, v_5 \rightarrow v_6$ to get a coarse polygon (v_0, v_6, v_7, v_8). Then if selective refinement is desired around vertex v_1, vertices v_6, v_5, v_4, v_3, v_2 will need to be split in that order before one can get to vertex v_1. An approach that was more oriented towards selective refinement might have collapsed $v_1 \rightarrow v_2, v_3 \rightarrow v_4, v_5 \rightarrow v_6, v_2 \rightarrow v_4, v_4 \rightarrow v_6$ for better adaptive results, even though the successive approximations are not as good as the previous ones under the distance metric.
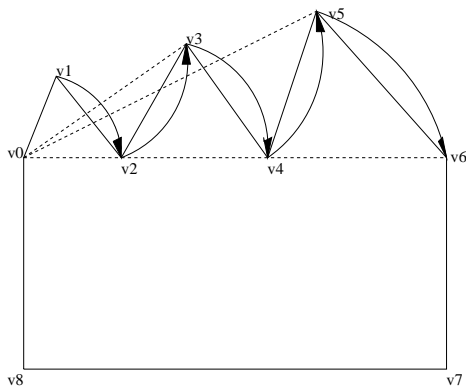

Fig. 3. Good versus efficient selective simplification

2. Since the edge collapses are defined in a linear sequence, the total number of child links to be traversed before reaching the desired node is $O(n)$.

3. No efficient method for incrementally updating the selective refinements from one frame to the next is given. The reverse problem of selective refinement – selective simplification too is not dealt with.

In this paper we provide a solution to the above issues with the aim of performing real-time adaptive simplifications and refinements. We define a criterion for performing edge collapses that permits adaptive refinement around any vertex. Instead of constructing a series of sequential edge collapses we construct a *merge tree* over the vertices of mesh $\hat{M}$ so that one can reach any child vertex in $O(\log n)$ links. We then describe how one can perform incremental updates within this tree to exploit frame-to-frame coherence, view-dependent illumination, and visibility computations using both scalar and vector attributes.

III. SIMPLIFICATION WITH IMAGE-SPACE FEEDBACK

Level-of-detail-based rendering has thus far emphasized object-space simplifications with minimal feedback from the image space. The feedback from the image space has been in the form of very crude heuristics such as the ratio of the screen-space area of the bounding box of the object to the distance of the object from the viewer. As a result, one witnesses coarse image-space artifacts such as the distracting "popping" effect when the object representation changes from one level of detail to the next [23]. Attempts such as alpha-blending between the old and the new levels of detail during such transitions serve to minimize the distraction at the cost of rendering two representations. However alpha blending is not the solution to this problem since it does not address the real cause – lack of sufficient image-space feedback to select the appropriate local level of detail in the object space; it merely tries to cover-up the distracting artifacts.

Increasing the feedback from the image space allows one to make better choices regarding the level of detail selection in the object-space. We next outline some of the ways in which image-space feedback can influence the level of detail selection in the object-space.

A. Local Illumination

Increasing detail in a direction perpendicular to, and proportional to, the illumination gradient across the surface is a good heuristic [2]. This allows one to have more detail in the regions where the illumination changes sharply and therefore one can represent the highlights and the sharp shadows well. Since surface normals play an important role in local illumination one can take advantage of the coherence in the surface normals to build a hierarchy over a continuous resolution model that allows one to capture the local illumination effects well. We outline in Section IV-C how one can build such a hierarchy.

B. Screen-Space Projections

Decision to keep or collapse an edge should depend upon the length of its screen-space projection instead of its object-space length. At a first glance this might seem very hard to accomplish in real-time since this could mean checking for the projected lengths of all edges at every frame. However, usually there is a significant coherence in the ratio of the image-space length to the object-space length of edges across the surface of an object and from one frame to the next. This makes it possible to take advantage of a hierarchy built upon the the object-space edge lengths for an object. We use an approximation to the screen-space projected edge length that is computed from the object-space edge length. We outline in Section IV-B how one can build such a hierarchy.

C. Visibility Culling

During interactive display of any model there is usually a significant coherence between the visible regions from one frame to the next. This is especially true of the back-facing polygons

that account for almost half the total number of polygons and do not contribute anything to the visual realism. A hierarchy over a continuous resolution representation of an object allows one to significantly simplify the invisible regions of an object, especially the back-facing ones. This view-dependent visibility culling can be implemented in a straightforward manner using the hierarchy on vertex normals discussed in Section IV-C.

D. Silhouette boundaries

Silhouettes play a very important role in perception of detail. Screen-space projected lengths of silhouette edges (i.e., edges for which one of the adjacent triangles is visible and the other is invisible), can be used to very precisely quantify the amount of smoothness of the silhouette boundaries. A hierarchy built upon a continuous-resolution representation of a object allows one to do this efficiently.

IV. CONSTRUCTION OF MERGE TREE

We would like to create a hierarchy that provides us a continuous-resolution representation of an object and allows us to perform real-time adaptive simplifications over the surface of an object based upon the image-space feedback mechanisms mentioned in Section III. Towards this end we implement a *merge tree* over the vertices of the original model. In our current implementation, the merge tree stores the edge collapses in a hierarchical manner. However, as we discuss in Section VII the concept of a merge tree is a very general one and it can be used with other local simplification approaches as well. Note that the merge tree construction is done as an off-line preprocessing step before the interactive visualization.

A. Basic Approach

In Figure 2, the vertex c is merged with the vertex p as a result of collapsing the edge (pc). Conversely, during a vertex split the vertex c is created from the vertex p. We shall henceforth refer to c as the child vertex of the parent vertex p. The merge tree is constructed upwards from the high-detail mesh $\hat{M}$ to a low-detail mesh M^0 by storing these parent-child relationships in a hierarchical manner over the surface of an object.

At each level l of the tree we determine parent-child relationships amongst as many vertices at level l as possible. In other words, we try to determine all vertices that can be safely merged based on criterion defined in Section IV-D. The vertices that are determined to be the children remain at level l and all the other vertices at level l are promoted to level $l+1$. Note that the vertices promoted to level $l+1$ are a proper superset of the parents of the children left behind at level l. This is because there are vertices at level l that are neither parents nor children. We discuss this in greater detail in the context of *regions of influence* later in this section. We apply the above procedure recursively at every level until either (a) we are left with a user-specified minimum number of vertices, or (b) we cannot establish any parent-child relationships amongst the vertices at a given level. Case (b) can arise because in determining a parent-child relationship we are essentially collapsing an edge and not all edge collapses are considered legal. For a detailed discussion on legality of edge collapses the interested reader can refer to [26].

Since in an edge collapse only one vertex merges with another, our merge tree is currently implemented as a binary tree.

To construct a balanced merge tree we note that the effects of an edge collapse are local. Let us define the *region of influence* of an edge (v_0, v_1) to be the union of triangles that are adjacent to either v_0 or v_1 or both. The region of influence of an edge is the set of triangles that can change as an edge is gradually collapsed to a vertex, for example, in a morphing. Thus, in Figure 2 as vertex c merges to vertex p, (or p splits to c), the changes to the mesh are all limited to within the region of influence of edge (pc) enclosed by $n_0, n_1, \ldots n_6$. Note that all the triangles in region of influence will change if vertices p and c are merged to form an intermediate vertex, say $(p+c)/2$. In our current implementation, the position of the intermediate vertex is the same as the position of the parent vertex p. However our data-structures can support other values of the intermediate vertex too. Such values could be used, for example, in creating intermediate morphs between two level-of-detail representations.

To create a reasonably balanced merge tree we try to collapse as many edges as possible at each level such that there are no common triangles in their respective regions of influence. Since this step involves only local checks, we can accomplish this step in time linear in the number of triangles at this level. If we assume that the average degree (i.e. the number of neighboring triangles) of a vertex is 6, we can expect the number of triangles in an edge's region of influence to be 10. After the collapse this number of triangles reduces to 8. Thus the number of triangles can be expected to reduce roughly by a factor of $4/5$ from a higher-detail level to a lower-detail level. Thus, in an ideal situation, the total time to build the tree will be given by $n + \frac{4n}{5} + \frac{16n}{25} + \ldots = O(n)$. However, this assumes that we arbitrarily choose the edges to be collapsed. A better alternative is to sort the edges by their edge lengths and collapse the shortest edges first. Collapsing an edge causes the neighboring edges to change their lengths. However as mentioned above, since changes are local we can maintain the sorted edge lengths in a heap for efficient updates. With this strategy one can build the merge tree in time $O(n \log n)$.

B. Scalar Subtree Attributes

To allow real-time refinement and simplification we can store at every parent node (i.e. a node that splits off a child vertex) of the merge tree, a range of scalar attributes of the children in the subtree below it. Then image-space feedback can be used to determine if this range of scalar attributes merits a refinement of this node or not. We explain this process of incremental refinement and simplification in greater details in Section V-A.

In our current implementation every merge tree node v stores the Euclidean distances to its child and parent that determine when v's child will merge into v and when v will merge into its parent. The former is called the *downswitch distance* and the latter is called the *upswitch distance*. These distances are built up during the merge tree creation stage. If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold, we permit refinement at v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold, it means that this region

occupies very little screen space and can be simplified.

C. Vector Subtree Attributes

Our implementation also allows incremental simplification and refinement based upon the coherences of the surface normals. This allows us to implement view-dependent real-time simplifications based on local illumination and visibility. The regions with low intensity gradients are drawn in lower detail, while the regions with high intensity gradients are drawn in higher detail. Similarly, regions of the object that are back-facing are drawn at a much lower detail then the front-facing regions.

a) Child Cones   b)Merged Parent Cone

Fig. 5. Cone merging
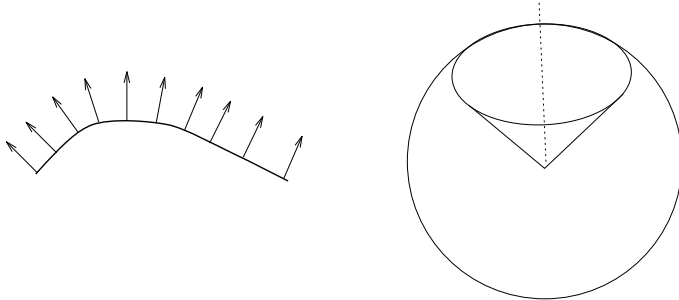
Fig. 4. Bounding cone for normal vectors

Since we are using frame-to-frame coherences in computing the levels of detail we need to adopt a data-structure that represents the variation in the normal vectors amongst all the descendents of any given vertex. To identify a possible representation, let us consider the idea behind a Gauss map. A Gauss map is a mapping of the unit normals to the corresponding points on the surface of a unit sphere. Thus, all the normal variations in a sub-tree will be represented by a closed and connected region on the surface of a sphere using a Gauss map. To simplify the computations involved, we have decided to approximate such regions by circles on the surface of the unit sphere, i.e. bounding cones containing all the subtree normal vectors. This is demonstrated in Figure 4 where the normal vectors in the surface shown on the left are contained within the cone (i.e. a circle on the Gauss map) on the right.

At the leaf-level, each vertex is associated with a normal-cone whose axis is given by its normal vector and whose angle is zero. As two vertices merge, the cones of the child and parent vertices are combined into a new normal cone that belongs to the parent vertex at the higher level. The idea behind this merging of cones is shown in Figure 5.

D. Merge Tree Dependencies

By using techniques outlined in Section V-A, one can determine which subset of vertices is sufficient to reconstruct an adaptive level-of-detail for a given object. However, it is not simple to define a triangulation over these vertices and guarantee that the triangulation will not "fold back" on itself or otherwise represent a non-manifold surface (even when the original was not so). Figure 6 shows an example of how an undesirable folding in the adaptive mesh can arise even though all the edge collapses that were determined statically were correct. A shows the initial state of the mesh. While constructing the merge tree,
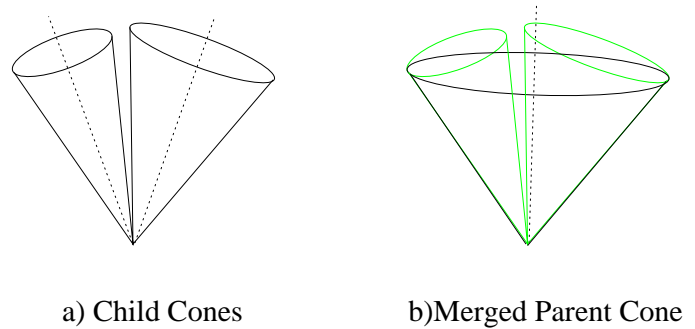
we first collapsed vertex v_2 to v_1 to get mesh B and then collapsed vertex v_3 to v_4 to get mesh C. Now suppose at run-time we determined that we needed to display vertices $v_1, v_2,$ and v_4 and could possibly collapse vertex v_3 to v_4. However, if we collapse v_3 to v_4 directly, as in mesh D, we get a mesh fold where there should have been none. One could devise elaborate procedures for checking and preventing such mesh fold-overs at run-time. However, such checks involve several floating-point operations and are too expensive to be performed on-the-fly.
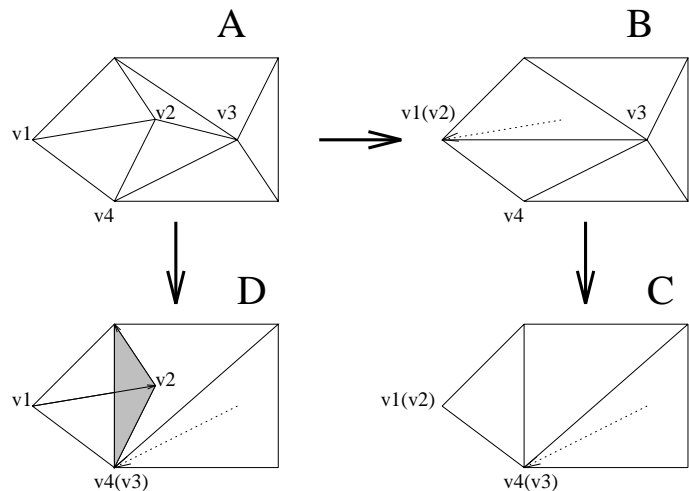

Fig. 6. Mesh folding problem

To solve the above problem we introduce the notion of dependencies amongst the nodes of a merge tree. Thus, the collapse of an edge e is permitted only when all the vertices defining the boundary of the region of influence of the edge e exist and are adjacent to the edge e. As an example, consider Figure 2. Vertex c can merge with vertex p only when the vertices $n_0, n_1, \ldots, n_k$ exist and are adjacent to p and c. From this we determine the following edge collapse dependencies, restricting the level difference between adjacent vertices:

1. c can collapse to p, only when $n_0, n_1, \ldots, n_k$ are present as neighbors of p and c for display.

2. $n_0, n_1, \ldots, n_k$ can not merge with other vertices, unless c first merges with p.

Similarly, to make a safe split from p to p and c, we determine the following vertex split dependency:

1. p can split to c and p, only when $n_0, n_1, \ldots, n_k$ are present as neighbors of p for display.

2. $n_0, n_1, \ldots, n_k$ can not split, unless p first splits to p and c.

The above dependencies are followed during each vertex-split or edge collapse during real-time simplification. These dependencies are easily identified and stored in the merge tree during its creation. Considering Figure 6 again, we can now see that collapse of vertex v_3 to v_4 depends upon the adjacency of vertex v_1 to v_3. If vertex v_2 is present then v_1 will not be adjacent to v_3 and therefore v_3 will not collapse to v_4. Although having dependencies might sometimes give lesser simplification than otherwise, it does have the advantage of eliminating the expensive floating-point run-time checks entirely. The basic idea behind merge tree dependencies has a strong resemblance to creating *balanced subdivisions* of quad-trees as presented by Baum *et al* in [3] where only a gradual change is permitted from regions of high simplifications to low simplifications. Details of how these merge tree dependencies are used during run-time are given in Section V-A.

The pseudocode outlining the data-structure for a merge tree node is given in Figure 7. The pseudocode for building and traversing the merge tree is given in Figure 8. We are representing the triangular mesh by the winged-edge data-structure to maintain the adjacency information.

```
struct NODE {
        struct VERTEX  *vert   ;   /* associated vertex */
        struct NODE    *parent ;   /* parent node for merging   */
        struct NODE    *child[2] ; /* child nodes for refinement */
        float          upswitch ;  /* threshold to merge */
        float          downswitch; /* threshold to refine */
        struct CONE    *cone   ;   /* range of subtree normals */
        struct VERTEX  **adj_vert ; /* adjacent vertices */
        int            adj_num ;   /* number of adjacent vertices */
        struct VERTEX  **depend_vert;/* dependency list for merge */
        int            depend_num  ;/* number of vertices in the */
        };                          /* dependency list */
```

Fig. 7. Data-structure for a merge tree node

```
/* Given a mesh, build_mergetree() constructs a list of merge trees - one
 * for every vertex at the coarsest level of detail.
 */
build_mergetree(struct MESH *mesh, struct NODE **roots)
{ struct HEAP  *current_heap, *next_heap ;
  int level ;

  current_heap = InitHeap(mesh);
  next_heap = InitHeap(nil) ;
  for ( level = 0 ; HeapSize(current_heap) > MIN_RESOLUTION_SIZE; level ++ )
  { while ( HeapSize(current_heap) > 0 )
        { edge = ExtractMinEdge(current_heap);
          node = CreatNode(edge);
          SetDependencies(node);
          SetCone(node);            /* Set vector attributes */
          SetSwitchDistances(node); /* Set scalar attributes */
          InsertHeap(next_heap, node);
        }
    FreeHeap(current_heap);
    current_heap = next_heap ;
    next_heap = InitHeap(nil);
  }
  FlattenHeap(roots, current_heap);
}

/* Given a list of nodes of the merge tree that were active in the previous
 * frame, traverse_mergetree() constructs a list of new merge tree nodes by
 * either refining or simplifying each of the active merge tree nodes.
 */
traverse_mergetree(struct NODE **current_list,
                   struct VIEW view, struct LIGHTS *lights )
{ int switch ;

  for each node in current_list do
  { switch = EvalSwitch(node, view, lights);
    if ( switch == REFINE )
       RefineNode(node);
    else if ( switch == SIMPLIFY )
       MergeNode(node);
  }
}
```

Fig. 8. Pseudocode for building and traversing the merge tree

V. REAL-TIME TRIANGULATION

Once the merge tree with dependencies has been constructed off-line it is easy to construct an adaptive level-of-detail mesh representation at run-time. Real-time adaptive mesh reconstruction involves two phases – determination of vertices that will be needed for reconstruction and determination of the triangulation amongst them. We shall refer to the vertices selected for display at a given frame as *display vertices* and triangles for display as *display triangles*. The phases for determination of display vertices and triangles are discussed next.

A. Determination of display vertices

In this section we outline how we determine the display vertices using the scalar and vector attribute ranges stored with the nodes of the merge tree. We first determine the *primary display vertices* using the screen-space projections and the normal vector cones associated with merge tree nodes. These are the only vertices that would be displayed if there were no triangulation constraints or mesh-folding problems. Next, from these primary display vertices we determine the *secondary display vertices* that are the vertices that need to be displayed due to merge tree dependencies to avoid the mesh fold-overs in run-time triangulations.

A.1 Primary Display Vertices

Screen-Space Projection

As mentioned earlier, every merge tree node v stores a Euclidean distance for splitting a vertex to its child (downswitch distance) as well as the distance at which it will merge to its parent (upswitch distance). If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold T, we permit refinement at v and recursively check the children of v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold T, it means that this region occupies very little screen space and can be simplified, so we mark v as *inactive* for display.

Normal Vectors

We need to determine the direction and the extent of the normal vector orientation within the subtree rooted at a display vertex, with respect to the viewing direction as well as light source, to accomplish view-dependent local illumination and visibility-based culling.

To determine silhouettes and the back-facing regions of an object, we check to see if the normal vector cone at a vertex lies entirely in a direction away from the viewer. If so, this vertex can be marked inactive for display. If not, this vertex is a display vertex and is a candidate for further refinement based on other criteria such as screen-space projection, illumination gradient, and silhouette smoothness. In such a case we recursively check its children. The three possible cases are shown in Figure 9.

Similarly, for normal-based local illumination, such as Phong illumination, we use the range of the reflection vectors and determine whether they contain the view direction or not to determine whether to simplify or refine a given node of the merge tree.

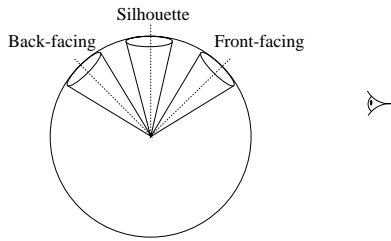We follow the procedures outlined above to select all those

Fig. 9. Selective refinement and simplification using normal cones

Fig. 10. Display triangle determination

vertices for display that either (a) are leaf nodes and none of their parents have been marked as inactive, or (b) have their immediate child marked as inactive. This determines the list of primary display vertices.

A.2 Secondary Display Vertices

We follow the merge dependencies from the list of primary display vertices to select the final set of display vertices in the following manner. If a vertex v is in the initial list of display vertices and for it to be created (via a vertex split), the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ had to be present, we add the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ to the list of display vertices and recursively consider their dependencies. We continue this process until no new vertices are added.

When determining the vertices for display in frame $i + 1$ we start from the vertex list for display used in frame i. We have found a substantial frame-to-frame coherence and the vertex display list does not change substantially from one frame to the next. There are minor local changes in the display list on account of vertices either refining or merging with other vertices. These are easily captured by either traversing the merge tree up or down from the current vertex position. The scalar and vector attribute ranges stored in merge tree nodes can be used to guide refinements if the difference in the display vertex lists from one frame to the next becomes non-local for any reason. We compute the list of display vertices for first frame by initializing the list of display vertices for frame 0 to be all the vertices in the model and then proceeding as above.

B. Determination of display triangles

If the display triangles for frame i are known, determination of the display triangles for frame $i+1$ proceeds in an interleaved fashion with the determination of display vertices for frame $i+1$ from frame i. Every time a display vertex of frame i merges in frame $i+1$ we simply delete and add appropriate triangles to the list of display triangles as shown in Figure 10. The case where a display vertex in frame i splits for frame $i+1$ is handled analogously. Incremental determination of display triangles in this manner is possible because of the dependency conditions mentioned in Section IV-D. The list of display triangles for the first frame is obtained by initializing the list for frame 0 to be all the triangles in the model and then following the above procedure.

VI. Results and Discussion

We have tried our implementation on several large triangulated models and have achieved encouraging results. These are summarized in Table I The images of teapot, bunny, crambin,
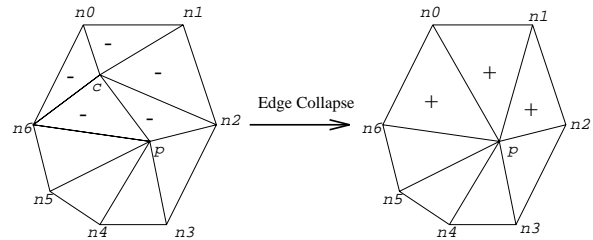
phone, sphere, buddha, and dragon models that were produced for the times in Table I are shown in Figures 1, 11, 13, and 14 respectively. All of these timings are in milliseconds on a Silicon Graphics Onyx with RE2 graphics, a 194MHz R10000 processor, and 640MB RAM. It is easy to see that the time to traverse the merge tree and construct the list of triangles to be displayed from frame to frame is relatively small. This is because of our incremental computations that exploit image-space, object-space, and frame-to-frame coherences. The above times hold as the user moves through the model or moves the lights around. The triangulation of the model changes dynamically to track the highlights as well as the screen-space projections of the faces.
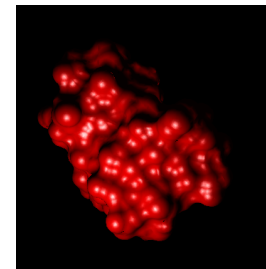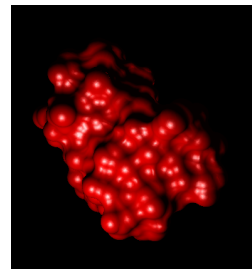
Highest detail Crambin surface　　　Simplified Crambin surface
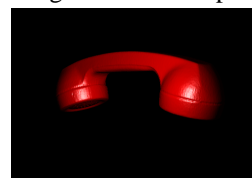

Highest detail Bunny　　　Simplified Bunny


Highest detail Teapot　　　Simplified Teapot


Highest detail Phone　　　Simplified Phone

Fig. 11. Dynamic adaptive simplification

| Dataset	Highest Detail		Adaptive Detail					Reduction Ratio	
	Display Tris	Display Time	Display Tris	Tree Levels	Traverse Tree	Display Time	Total Time	Display Tris	Display Time
Teapot	3751	57	1203	36	10	17	27	32.0%	47.3 %
Sphere	8192	115	994	42	8	16	24	12.1%	20.8 %
Bunny	69451	1189	13696	65	157	128	285	19.7%	23.9 %
Crambin	109884	1832	19360	61	160	194	354	17.6%	19.3 %
Phone	165963	2629	14914	63	112	144	256	8.9 %	9.7 %
Dragon	202520	3248	49771	66	447	394	842	24.5%	25.9 %
Buddha	293232	4618	68173	69	681	546	1227	23.2%	26.5 %

TABLE I

ADAPTIVE LEVEL OF DETAIL GENERATION TIMES
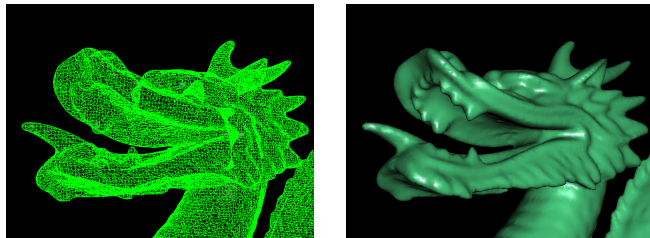
Highest detail model – bottom light source
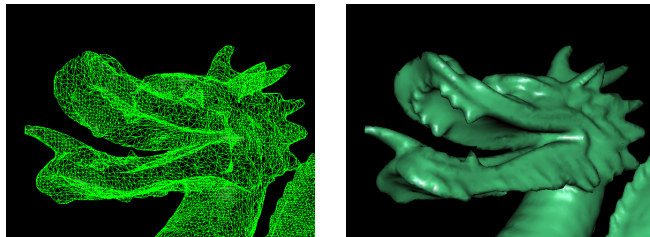
Dynamic adaptive simplification – top light source
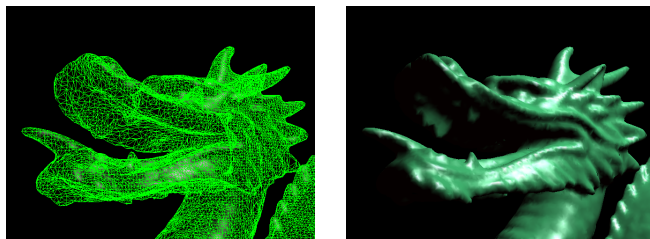
Dynamic adaptive simplification – top light source

Fig. 12. Dynamic adaptive simplification for the head of the Dragon

As can be seen from the merge tree depths, the trees are not perfectly balanced. However, they are still within a small factor of the optimal depths. This factor is the price that has to be paid to incorporate dependencies and avoid the expensive run-time floating-point checks for ensuring good triangulations. For each dataset, we continued the merge tree construction till 8 or fewer vertices were left. As expected, the factor by which the number of vertices decreases from one level to the next tapers off as we reach lower-detail levels since there are now fewer alternatives left to counter the remaining dependency constraints. As an ex-

ample, for sphere, only 64 vertices were present at level 30 and it took another 12 levels to bring down the number to 8. If the tree depth becomes a concern one can stop sooner, trading-off the tree traversal time for the display time.

An interesting aspect of allowing dependencies in the merge tree is that one can now influence the characteristics of the run-time triangulation based upon static edge-collapse decisions during pre-processing. As an example, we have implemented avoidance of slivery (long and thin) triangles in the run-time triangulation. As Guéziec [20], we quantify the quality of a triangle with area a and lengths of the three sides l_0, l_1, and l_2 based on the following formula:

$$\text{Quality} = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \qquad (3)$$

Using Equation 3 the quality of a degenerate triangle evaluates to 0 and that of an equilateral triangle to 1. We classify all edge collapses that result in slivery triangles to be invalid, trading-off quantity (amount of simplification) for quality.

One of the advantages of using normal cones for back-face simplification and silhouette definition is that it allows the graphics to focus more on the regions of the object that are perceptually more important. Thus, for instance, for generating the given image of the molecule crambin, 8729 front-facing vertices were traversed as compared to 3361 backfacing vertices; 1372 were classified as silhouette vertices. Similarly, for the model of phone, our approach traversed 6552 front-facing vertices compared to only 1300 backfacing vertices; 900 were classified as silhouette vertices.

Clearly, there is a tradeoff here between image-quality and amount of simplification achieved. The results for our simplifications given in this section correspond to the images that we thought were comparable to the images from the original highest detail models. Higher levels of simplifications (that are faster to incrementally compute and display) with correspondingly lower quality images are obviously possible, allowing easy implementations of progressive refinement for display.

VII. CONCLUSIONS AND FUTURE WORK

We have outlined a simple approach to maintain dynamically adaptive level of detail triangulations. Crucial to this approach
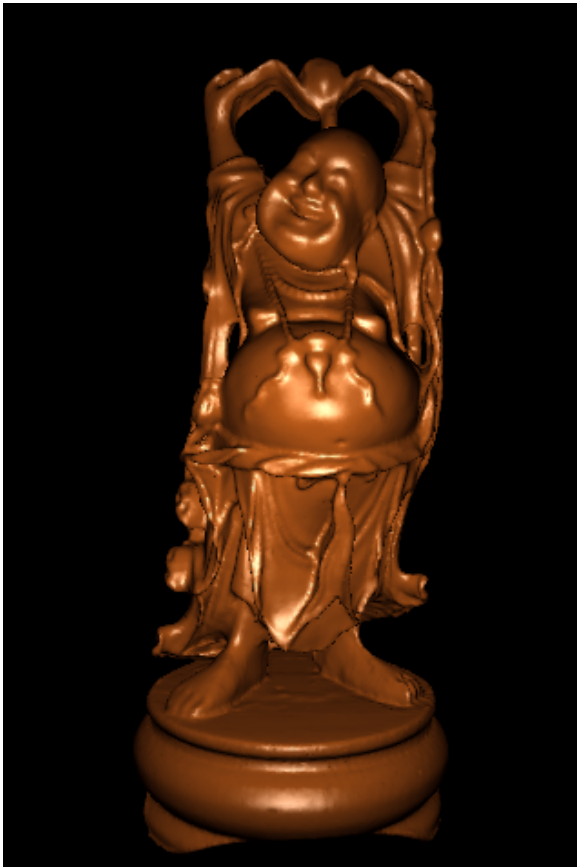
Highest detail   Simplified

Fig. 13. Dynamic adaptive simplification for the Buddha

Highest detail   Simplified

Fig. 14. Dynamic adaptive simplification for the Dragon

is the notion of merge trees that are computed statically and are used during run-time to take advantage of the incremental changes in the triangulation. In our current implementation we are using the method of edge collapses. However the idea behind merge trees is pretty general and can be used in conjunction with other local heuristics for simplification such as vertex deletion and vertex collapsing. We plan to study some of these other heuristics in the future and compare them with our current implementation that uses edge collapses.

At present we do not store color ranges at the nodes of the merge tree. Storing and using these should improve the quality of the visualizations produced using merge trees even further. Also of some interest will be techniques that create better balanced merge trees while still incorporating dependencies. We plan to investigate these issues further.

Of course, our approach also makes dynamically-specified manual simplifications possible, where the user can interactively specify the amounts of approximation desired at various regions of the object. Using this, certain parts of the object can be rendered at lower or higher details than otherwise. However, in this paper we have only considered automatic object simplifications during interactive display.

ACKNOWLEDGEMENTS

REFERENCES

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC 27599-3175, 1990.

[2] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, No. 2, pages 41–50, March 1990.

[3] D. R. Baum, Mann S., Smith K. P., and Winget J. M. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:51–60, 1991.

[4] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics: Proceedings of SIGGRAPH'86*, volume 20, No. 4, pages 29–37. ACM SIGGRAPH, 1986.

[5] J. F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78*, pages 286–292. ACM, 1978.

[6] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *CACM*, 19(10):542–547, October 1976.

[7] S. Chen. Quicktime VR – an image-based approach to virtual environment navigation. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 29–38. ACM, 1995.

[8] S. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[9] J. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[10] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification envelopes. In *Proceed-*

[11] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[12] F. C. Crow. A more flexible image generation environment. In *Computer Graphics: Proceedings of SIGGRAPH'82*, volume 16, No. 3, pages 9–18. ACM SIGGRAPH, 1982.

[13] L. Darsa, B. Costa, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *Proceedings, 1997 Symposium on Interactive 3D Graphics*, pages 25 – 34, 1997.

[14] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[15] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH 95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182. ACM SIGGRAPH, August 1995.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. ACM SIGGRAPH, August 1993.

[17] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 65 – 74. ACM Siggraph, ACM Press, August 1996.

[18] N. Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[19] 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.

[20] A. Guéziec. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery, MRCAS '95*, 1995.

[21] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11:197–214, 1994.

[22] 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.

[23] J. Helman. Graphics techniques for walkthrough applications. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages B1–B25, 1995.

[24] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[25] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[26] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 19–26. ACM SIGGRAPH, August 1993.

[27] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings, 1995 Symposium on Interactive 3D Graphics*, pages 105 – 106, 1995.

[28] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Computer Graphics*, pages 95–102, 1995.

[29] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 39–46. ACM, 1995.

[30] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, July 1994.

[31] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[32] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 65–70. ACM SIGGRAPH, 1992.

[33] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical

image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 75–82. ACM SIGGRAPH, ACM Press, August 1996.

[34] S. Teller and C. H. Séquin. Visibility preprocessing for interactive walk-throughs. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:61–69, 1991.

[35] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 55–64. ACM SIGGRAPH, 1992.

[36] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.