Parallel Processing for View-Dependent Polygonal Virtual Environments

Jihad El-Sana Amitabh Varshney

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

ABSTRACT

This paper presents a parallel algorithm for preprocessing as well as real-time navigation of view-dependent virtual environments on shared memory multiprocessors. The algorithm proceeds by hierarchical spatial subdivision of the input dataset by an octree. The parallel algorithm is robust and does not generate any artifacts such as degenerate triangles and mesh foldovers. The algorithm performance scales linearly with increase in the number of processors as well as increase in the input dataset complexity. The resulting visualization performance is fast enough to enable interleaved acquisition and modification with interactive visualization.

1. INTRODUCTION

Interactive visualization of large geometric datasets in computer graphics is a challenging task due to several reasons. One of the main reasons is that the sizes of several present-day geometric datasets are one or more orders of magnitude larger than what the current graphics hardware can display at interactive rates. Further, the rate of growth in the complexity of such geometric datasets is greater than the advances in graphics hardware rendering capabilities. As a result, several software and algorithmic solutions have been proposed to bridge this gap between the actual and desired rendering performances on large datasets. These include visibility-based culling, geometric multiresolution hierarchies, levels of detail in illumination and shading, texture mapping, and image-based rendering.

Traditional level-of-detail-based rendering approaches substitute low-detail representations of objects that are perceptually less significant and high-detail representations of objects that are perceptually more significant. However, these approaches do not allow one to vary detail across different regions of the same object based on illumination and viewing parameters. Recently, Xia *et al*,[10] Hoppe,[7] and Luebke *et al*[9] have proposed the idea of view-dependent navigation of large polygonal datasets in virtual environments. These approaches allow real-time modification of the level of detail of an object based on run-time parameters such as local illumination and distance from the viewer. However, these techniques require non-trivial pre-processing times, making them better suited for off-line pre-processing followed by interactive visualization. This paper presents a technique for parallelizing the pre-processing as well as navigation of such datasets. This has the potential to influence the generation and acquisition of such datasets based on visualization-assisted human feedback.

In this paper we first give an overview of a related work on view-dependent simplification. We then discuss our preprocessing algorithm followed by our real-time navigation algorithm and results.

2. RELATED WORK

Related work in the area of polygonal simplification has been well surveyed in several recent papers.[1,2,6,9] In this paper we shall overview related work on view-dependent simplification.

E-mail: jihad,varshney@cs.sunysb.edu

2.1. View-Dependent Simplifications

Most of the previous work on generating multiresolution hierarchies for level-of-detail-based rendering has concentrated on computing a fixed set of view-independent levels of detail. At runtime an appropriate level of detail is selected based on viewing parameters. Such methods are overly restrictive and do not take into account finer image-space feedback such as light position, visual acuity, silhouettes, and view direction. Recent advances to address some of these issues in a view-dependent manner take advantage of the temporal coherence to adaptively refine or simplify the polygonal environment from one frame to the next. In particular, adaptive levels of detail have been used in terrains by Gross *et al*[3] and Lindstrom *et al.*[8] Gross *et al* define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Lindstrom *et al* define a quadtree-based block data structure that provides a continuous level of detail representation. In these approaches, the level of detail around any region can adaptively refine in real-time. These lines of research provide elegant solutions for terrains and other datasets that are defined on a grid. Most of the work for view-dependent simplifications for general polygonal models is closely related to the concept of progressive meshes that are summarized next.

2.1.1. Progressive Meshes

Progressive meshes have been introduced by Hoppe[6] 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.

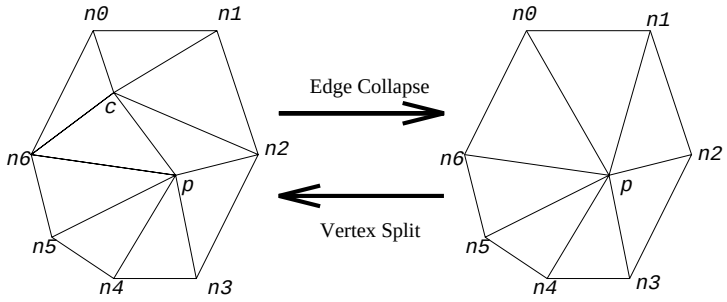


Figure 1. Edge collapse and vertex split

A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses:

$$M^k \overset{collapse_{k-1}}{\rightarrow} M^{k-1} \overset{collapse_{k-2}}{\rightarrow} \ldots M^1 \overset{collapse_0}{\rightarrow} M^0 \tag{1}$$

One 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:

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

The sequence $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ is referred to as a *progressive mesh* representation.

2.1.2. Vertex Hierarchies

Merge trees have been introduced by Xia *et al*[10] 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[7] has independently developed a view-dependent simplification algorithm that works with progressive meshes. This algorithm proceeds to construct a vertex hierarchy over a progressive mesh in a top-down fashion by

minimizing an energy function.[6] Screen-space projection and orientation of the polygons is then used to guide the run-time view-dependent simplifications. Luebke and Erikson[9] define a *tight octree* over the vertices of the given model to generate hierarchical view-dependent simplifications. In a tight octree, each node of the octree is tightened to the smallest axis-aligned bounding cube that encloses the relevant vertices before subdividing further. If the screen-space projection of a given cell of an octree is too small, all the vertices in that cell are collapsed to one vertex and the parent cell is then considered for the same test. Guéziec *et al*[5] 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.

3. PARALLEL PREPROCESSING FOR VIEW-DEPENDENT NAVIGATION

Merge trees have been introduced by Xia *et al*[10] as a data-structure built upon progressive meshes to enable real-time view-dependent rendering of an object. Let the vertex p that arises from the collapse of edge (p, c) be considered the parent and the vertex c be the child. The merge tree is constructed in a bottom-up fashion from the high-detail mesh to a low-detail mesh by storing these parent-child relationships in a hierarchical manner over the surface of an object. At each level l of the tree a maximal set of edge-collapses is selected in the shortest-edge-first order and with the constraint that their neighborhoods do not overlap. The vertices remaining after these edge collapses are promoted to level $l + 1$. Each node of the merge tree represent a vertex in some level of detail.

View-dependent simplification is achieved by performing edge-collapses and vertex-splits on the triangulation used for display depending upon view-dependent parameters such as lighting (detail is directly proportional to intensity gradient), polygon orientation, (high detail for silhouettes and low detail for backfacing regions) and screen-space projection. Since there is a high temporal coherence the selected levels in the merge tree change only gradually from frame to frame. Unconstrained edge-collapses and vertex-splits during runtime can be shown to result in mesh foldovers resulting in visual artifacts such as shading discontinuities. To avoid these artifacts Xia *et al*[10] propose the concept of dependencies or constraints that necessitate the presence of the entire neighborhood of an edge before it is collapsed (or its parent vertex is split).

3.1. Sequential Initialization

Our algorithm performs the construction of the merge tree in a parallel fashion. It was designed with the following goals in mind:

- divide the input polygonal mesh into mutually exclusive, collectively exhaustive subsets such that the processing of each subset could be carried out in parallel, and
- the load should be balanced across multiple processors.

As our results demonstrate in Section 5, we were able to achieve both of these objectives.

In the initialization step of our algorithm we create the cells in a sequential manner. Cell creation is simple and fast. Input mesh is subdivided into the cells of a regular grid which is the lowest level of a uniform octree. At the end of this process each cell contains a list of pointers to vertices that belong to it. We subdivide edges in a disjoint manner across cells: an edge $e_i = (v_0, v_1)$ belongs to the cell which contains the vertex v_0. This stage of our algorithm is performed only once, for the initial set of vertices and for the lowest level (level 0) of the octree. From our experience with the sample datasets we have tested this algorithm on, this stage is currently fast enough without parallelization.

3.2. Cell Processing

In processing each cell we determine the sequence of edge collapses that this cell contributes in the final merge tree. We are interested in a sequence which keeps the visual appearance of an object as close as possible to its original. Each edge collapse introduces additional errors to the simplified model. These errors are also influenced by the sequence of the previous collapses. We chose to use a greedy algorithm that carry out the edge collapses that introduce the least error first. However, an edge collapse may lead to a foldover or a sliver triangle, which introduces visualization artifacts in computing lighting and shading. Hence, we want to prevent the collapse of such edges. We shall refer to a collapse of an edge that does not introduce the above artifacts as a *safe collapse*. To determine the

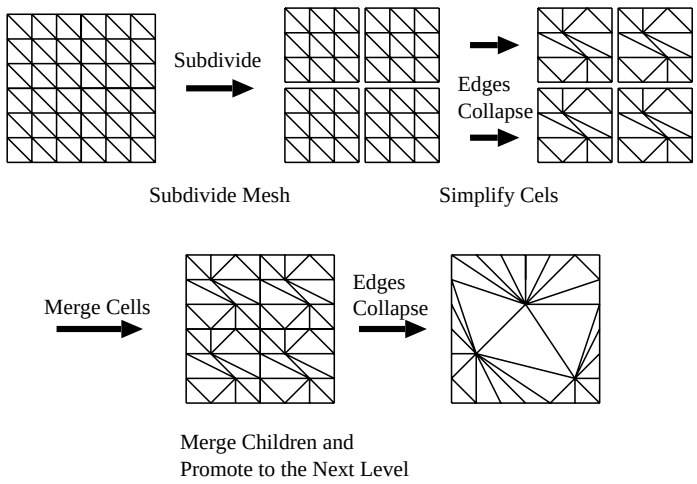


Figure 2. Algorithm Flow

safety of an edge collapse we use two heuristics. First, for any triangle adjacent to any of the two collapsed vertices, the difference between the normals 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 does not drop beyond 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 based on the equation (3) by Guéziec.[4] Using equation (3) the quality of a degenerate triangle evaluates to 0 and that of an equilateral triangle to 1.

$$Quality = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \tag{3}$$

Processing each cell involves building an edge heap of all the edges associated with that cell. This heap is ordered by the length of the edges (shortest edges first). Then for each edge that is removed from this heap, a series of tests are performed to determine if it can be safely collapsed. If the edge passes the tests, the collapse operation of this edge is executed and the resulting vertex is added to the merge sub-tree associated with the cell. In addition, the triangles adjacent to the two vertices, of the collapsed edges, are updated and degenerate triangles are removed.

3.3. Parallel Processing

Typical datasets of interest to the visualization community contain hundreds of thousands to millions of triangles. These numbers are much larger than the number of processors that will likely be available for parallel processing for these applications. This necessitates the presence of a priority queue from which the relatively smaller number of processors can select from the relatively large number of cells for processing.

The number of edges in each cell mainly determines the running time for processing that cell. We shall refer to cells that contain a large number of edges as *heavily-loaded* and cells that contain a small number of edges as *lightly-loaded*. From the load balancing point of view, we would like to avoid the case where the priority queue is empty and one processor is working on a *heavily-loaded* cell while the rest of the processors are idle. We can reduce the penalty for such case by processing the *lightly-loaded* cells last. We achieve this by processing the *heavily-loaded* cells first by using a **priority queue**.

The pseudo-code for constructing the edge heap for each cell in parallel is given in Figure 3. *ParallelHeapConstruct()* receives the list of cells sorted by the number of vertices in each cell. Each idle process gets the next cell from the list, builds its edge heap, and inserts it into the priority queue. Note that the actual construction of the edge heap for each cell in *ConstructEdgeHeap()* is performed in a lock-free manner.

```
1.ParallelHeapConstruct()
2.{
3.    CELL *cell ;
4.    forever do {
6.       acquire lock;
7.       if ( cell )
8.           PriorityQueue.Insert(cell) ;
9.       if ( CellsList.Empty() ) {
10.          release lock;
11.          return ;
12.      }
13.      cell = GetNextCell();
14.      release lock;
15.      ConstructEdgeHeap(cell);
16.   }
17.}
```

Figure 3. Pseudo-code for constructing edge heaps in parallel.

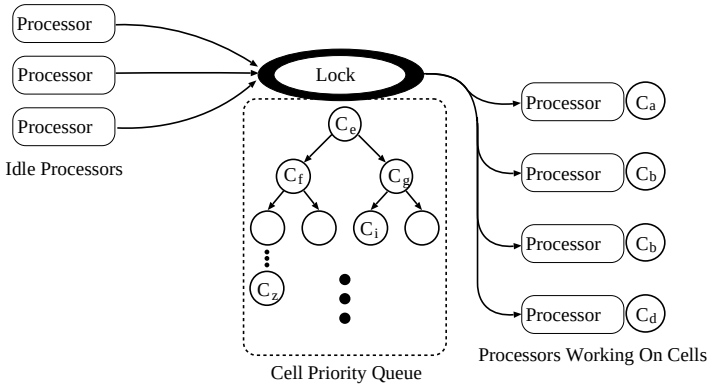


Figure 4. Algorithm Flow

In the procedure *ParallelMergeTreeConstruct()* each idle processor accesses the priority queue which is protected by an exclusive lock as shown in Figure 4. The processor first acquires a lock to access the queue, removes the first safe cell from the queue, and then releases the lock.

The processor then processes the cell that it just removed in a lock-free manner. If the cell was the last cell in the group of eight cells that could be combined to form the next higher level cell of the octree, then the edge heaps of the eight cells of the group are merged to form the parent cell's edge heap. This new (parent) cell is then inserted in the priority queue. This is outlined in the pseudo-code for *ParallelMergeTreeConstruct()* shown in Figure 5. Note that the variable *cell* is local to the current thread.

The lock-free processing carried out in the procedure ProcessCell() is possible because of the following constraints:

- Only those edges of a cell are considered for collapse and subsequent testing for validity whose length is less than that of the bounding box of the cell, and
- Two spatially adjacent cells are not allowed to be processed concurrently.

Note that the first constraint leaves uncollapsed edges on the edge heap after the cell processing has been completed. These edges are considered at a higher level in the octree after the current edge heap has been merged

```
1.ParallelMergeTreeConstruct()
2.{
3.   CELL *cell ;
4.   forever do
5.   {
6.      acquire lock;
7.      if ( LastCellInGroup(cell)) {
8.         cell = MergeGroupCells();
9.         PriorityQueue.Insert(cell);
10.     }
11.     if ( PriorityQueue.Empty() ) {
12.        release lock;
13.        return;
14.     }
15.     cell = RemoveFirstSafeCell(PriorityQueue) ;
16.     release lock ;
17.     ProcessCell(cell);
18.   }
19.}
```

Figure 5. Pseudo-code for constructing and combining merge sub-trees in parallel.

one or more times and the cell size becomes larger. We merge the remainder of the edge heap on the eight children cells to construct the heap on the parent cell. To simplify the testing of the second condition we group each of the eight cells that share the same parent cell together. At a given time each group receives only one processor.

4. PARALLEL VIEW-DEPENDENT NAVIGATION

The view-dependent merge tree constructed using the above algorithm is used by the navigation process to visualize the 3D object. The navigation algorithm consists of two parallel components: *Adapt* and *Display*. The *Adapt* component updates the list of triangles to be displayed by taking into account the changes in illumination, viewer position, location of silhouettes, and front and back-facing regions of the object. Since there is a high temporal coherence the selected levels in the merge tree and consequently the displayed triangles change only gradually from frame to frame. The *Display* component sends the display list of triangles to the graphics engine. The two components share the display list buffers, which are protected by an exclusive lock.

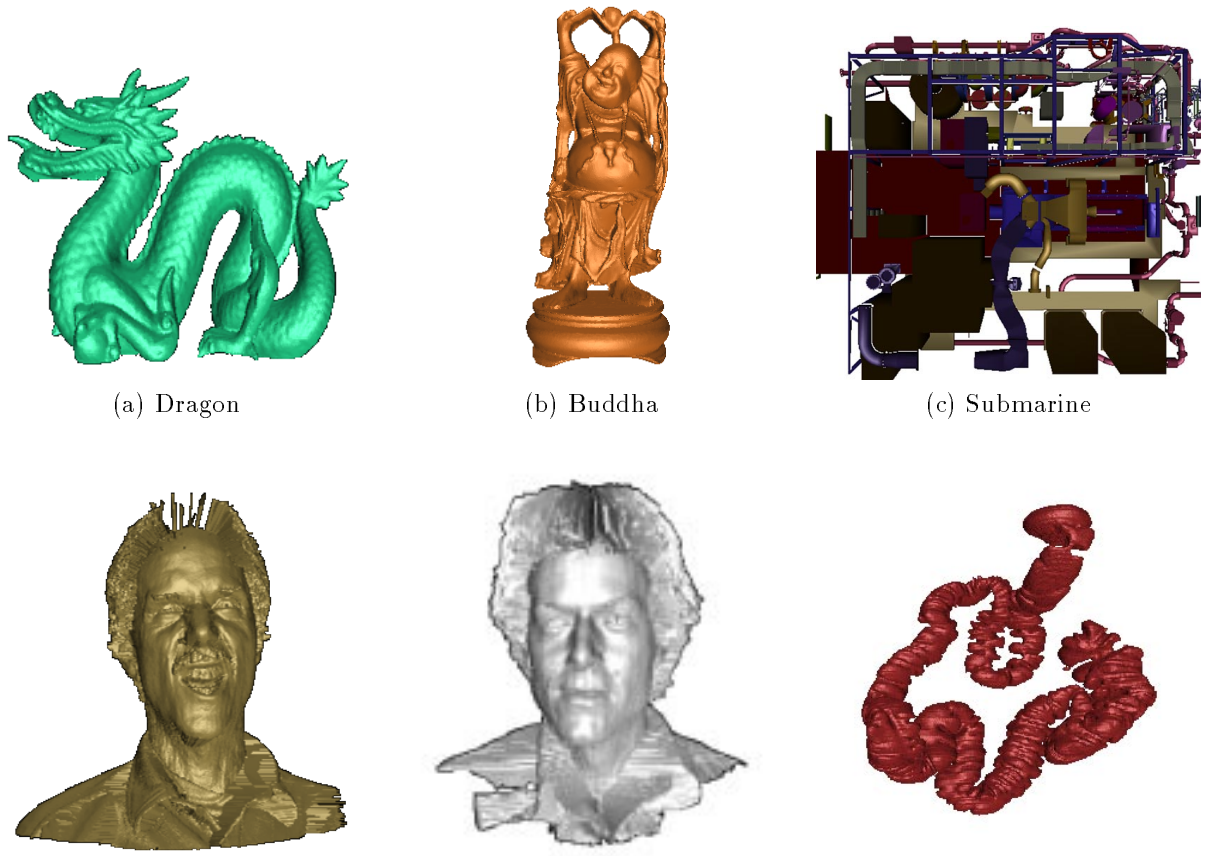
The *Adapt* component holds a list of pointers to the current active nodes of the merge tree; we shall refer to this list as the *active nodes* list. The *Display* component sends an update event each time the view parameters change. On receiving an update event, the *Adapt* scans and updates the active nodes list. Depending on viewing parameters and light, a node in the active list can refine, merge, or stay in its current status. If a node needs to be refined, the current node is replaced by its two children in the active nodes list, and the adjacent triangles are updated appropriately. If a node needs to be merged, this node and its sibling are replaced by their parent node, and the adjacent triangles are updated. The changes on the active nodes list and the triangles list are passed to an auxiliary buffer, which are flushed later into the display buffers.

The *Display* component holds the list of vertices information such as coordinate, normal, and color, of the active nodes and a list of triangles as three indices into the vertices list. The *Display* component deals with following three main events, display, flush , and input. When flush event is received, the auxiliary buffers are flushed into the *Display* component buffers. At input event the view and light parameters are updated. The display event signals the need to resend the vertex and triangle lists to the graphics engine. A display event is sent each time the camera or the display list changes.

The display buffers are updated in an incremental fashion. Since there is typically a very high degree of temporal coherence, the changes to the display buffers stored in the auxiliary buffers are relatively small and the *Adapt* needs to hold the lock on the display buffers for a very short time.

5. RESULTS

We have implemented the above algorithms using POSIX threads on SGI Power Challenge with 16 R10000 processors. We have tested our system on a variety of datasets from different sources and have achieved encouraging results. Table 1 summarizes some of our results. Steve and David models have been provided by Cyberware, Dragon and Buddha have been provided by the Stanford Computer Graphics Laboratory, Submarine represents the Auxiliary Machine Room mechanical CAD model of a notional submarine provided to us by the Electric Boat Corporation, and Colon is the iso-surface of a colon extracted from a CT scan volumetric model provided to us by the VolVis group at SUNY Stony Brook.



(a) Dragon (b) Buddha (c) Submarine

(d) Steve (e) David (f) Colon

Figure 6. Test Models for our Algorithm

Model	Number Of Triangles	Time in seconds				
		1 proc	2 procs	4 procs	8 procs	16 procs
Dragon	202,520	18.90	12.21	07.03	04.67	02.20
Buddha	293,520	26.53	16.89	09.37	05.94	03.39
Submarine	339,444	23.16	14.96	08.44	04.45	02.54
Steve	739,639	75.19	43.13	24.41	15.74	08.33
David	1,172699	122.47	72.01	38.54	24.32	15.04
Colon	1,814,364	48.84	30.04	15.95	08.70	05.99

Table 1. Parallel Preprocessing for Merge Tree Construction

Table 2 below demonstrates the high degree of load balance achieved by our algorithm for the dataset Steve; the

behavior of other datasets is similar. The Min and Max columns show the time spent by the least-loaded and the heaviest-loaded processors, respectively. The final column shows the load imbalance ratio, which as can be seen, is quite small.

Number Of Processors	Min	Max	Average	Max/Avg -1
1	75.19	75.19	75.190	0
2	43.01	43.07	43.035	0.0008
4	24.00	24.27	24.150	0.0049
8	14.10	15.50	14.432	0.0740
16	07.73	08.31	08.004	0.0382

Table 2. Load Balance for Parallel Preprocessing.

Figures 7 and 8 below demonstrate that the speedup achieved by our algorithm scales well with increasing number of processors as well as increasing dataset size.

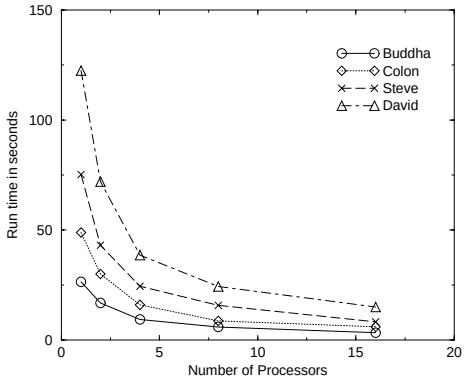


Figure 7. Comparison of Run times with Increasing Number of Processors

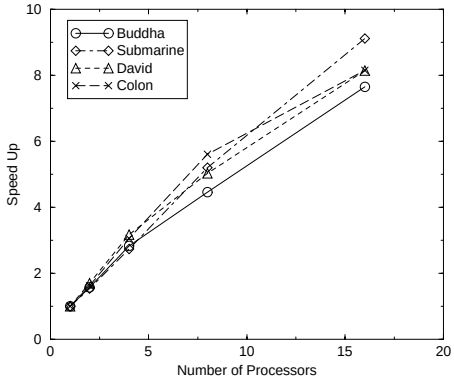


Figure 8. Speedup with Increasing Number of Processors

6. CONCLUSIONS

We have presented an algorithm for parallel pre-processing for view-dependent visualization of large datasets. By parallelization it is possible to bring down the times for pre-processing down to a few seconds even for datasets that are over a million triangles. This has important implications for the way such large datasets are acquired, generated, and designed. For instance, it took Cyberware 17 seconds to acquire datasets such as Steve and David, which is approximately how long it takes us to preprocess them. If the two were to proceed in an interleaved fashion, once

could acquire and visualize in real-time thereby potentially influencing the acquisition process itself. Based on our results, similar arguments may be made for collaborative visualization-assisted CAD design as well as medical volume visualization. We anticipate that parallelization of simplification will also find application in real-time simplification for interactive acquisition of range data for image-based rendering.

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 a Fulbright/Arab-Israeli Scholarship. The Submarine in figure 6(c) is part of the dataset of a notional submarine provided to us by the Electric Boat Division of General Dynamics. The happy Buddha in figure 6 (b) model has been provided by the Stanford Computer Graphics Laboratory and the models Steve and David models in figure 6 (d) and (e) have been provided by Cyberware.

REFERENCES

1. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4-9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 119 – 128. ACM SIGGRAPH, ACM Press, August 1996.
2. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 209 – 216. ACM SIGGRAPH, ACM Press, August 1997.
3. 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.
4. A. Guéziec. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery, MRCAS '95*, 1995.
5. A. Gueziec, F. Lazarus, G. Taubin, and W. Horn. Surface partitions for progressive transmission and display, and dynamic simplification of polygonal surfaces. In S. N. Spencer, editor, *Proceedings VRML 98: third Symposium on the Virtual Reality Modeling Language, Monterey, California, February 16-19, 1998*, pages 25–32, New York, NY, USA, 1998. ACM Press.
6. 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.
7. H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM SIGGRAPH, ACM Press, August 1997.
8. P. Lindstrom, D. Koller, W. Ribarsky, L. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996.
9. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 198 – 208. ACM SIGGRAPH, ACM Press, August 1997.
10. 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.