Proc. VIIth Digital Image Computing: Techniques and Applications, Sun C., Talbot H., Ourselin S. and Adriaansen T. (Eds.), 10-12 Dec. 2003, Sydney

Techniques for Accelerated View-Dependent Mesh Refinement

James Strauss and Amitava Datta

School of Computer Science and Software Engineering,
The University of Western Australia,
35 Stirling Highway, Crawley, W.A. 6009, Australia
jamess,datta@csse.uwa.edu.au

Abstract. View-dependent mesh refinement techniques typically pre-compute a hierarchical data structure that is queried at run-time to produce an approximation of a given mesh. This approximation, or level-of-detail (LOD), can be used in place of the original object so long as the viewpoint does not change. This should result in no loss of visible detail, yet should be less computationally expensive.
The approach of existing techniques is to collapse or expand nodes in the hierarchical data structure level by level. We propose a new method for fast generation of view-dependent level-of-details when the frame-to-frame coherence is low, such as when an object moves or rotates at a rapid rate with respect to the viewpoint.
Our method is based on two new techniques for aggressive detection of visible parts of the merge tree data structure. The jump split and jump collapse move the active nodes front up or down the tree many generations at a time, reducing the number and expense of iterations required to refine the mesh.

1 Introduction

Most three dimensional (3D) objects in computer graphics are represented by polygonal meshes. These polygonal meshes are often obtained using laser scanning equipment and may consist of millions of triangles that describe an object very accurately. Rendering these complex meshes at interactive frame-rates requires significant rendering resources.

One popular strategy for reducing the load on rendering hardware is to avoid rendering the fine details of an object that may be so fine that they do not contribute visually to the image produced by the renderer. In such cases, an approximation of the original object may be used so that only the visible parts of the mesh are rendered at full detail and other parts are rendered at coarser resolution. Widely used techniques for view-dependent refinement of meshes are the *progressive mesh* by Hoppe [6], the *merge tree* by Xia *et al.* [12, 11] and the *half-edge collapse hierarchy* by Pajarola [9].

The key requirement for any data structure storing the continuous level-of-details for a mesh is that it should be possible to quickly identify the parts of the mesh to be rendered when the viewpoint changes. Our technique improves detection of the visible parts of a model with high level-of-detail when the viewpoint changes rapidly.

The rest of the paper is organized as follows. We discuss some previous work in section 2, data structures in section 3, our method for determination of level-of-details in section 4, some optimizations in our implementation in section 5 and finally, some results in section 6.

2 Previous work

Several mesh refinement techniques exist that focus upon producing a simplified level-of-detail from a mesh of arbitrary shape and size.

2.1 Progressive Meshes and the Vertex Split

The Progressive Mesh (PM), developed by Hoppe [6], is a data structure that captures a continuous sequence of mesh approximations. A progressive mesh is stored as a coarse base model together with a sequence of detail records that indicate how to incrementally refine the base mesh into the original complex model. The PM is optimized for view-independent LOD control.

Each detail record in the progressive mesh carries information for a vertex split operation that adds one vertex to the model. To produce a simplification, the base mesh is retrieved from the progressive mesh and a sub-sequence of detail records are used to perform vertex splits on the base mesh to produce the desired resolution. Each vertex split operation can be used in reverse as an edge collapse.

2.2 View-dependent Simplification

View-dependent simplification uses characteristics of the scene, such as the viewpoint of the camera and the location of light sources, to determine the most appropriate level-of-detail for an object. The main advantage of a view-dependent system is that different regions of a single object can be displayed at different resolution. For example, regions of the object at close range can be displayed at higher complexity than regions at greater distance.

The difficulty in view-dependent simplification arises from the fact that a typical simplification produces only one sequence of vertex splits to refine the coarse base mesh into the complete complex mesh. The detail required for a view-dependent simplification may occur anywhere in this sequence, hence all vertex splits before that record in the sequence must be performed. This results in a significant number of vertex splits that are not necessary but are forced by precedence in the single sequence of detail records.

Another solution is to provide a large number of independent refinement paths from the coarse mesh to the original mesh. The paths may be followed to different extents in order to simplify different areas of the object. Hierarchical data structures were developed for this purpose.

3 Data Structures

Mesh refinement requires at least two main data structures. The first is a data structure to store the mesh itself. The second stores multiple refinement paths from the original mesh to its coarsest refinement.

3.1 The Half-Edge

Typical mesh data structures are useful for storing static meshes but are not designed to handle meshes that frequently change topology. As vertices are added to the mesh, they must be integrated into the existing topology and the mesh must be locally re-triangulated, requiring polygon adjacency information. We use the half-edge data structure [10] to store the polygon adjacency information of the mesh. Other benefits are the consistent space requirements and the efficiency with which vertex splits and edge collapses can be applied to a mesh stored in this fashion.

3.2 The Modified Merge tree

Our data structure is based on the merge tree data structure by Xia *et al.* [12, 11] with several modifications designed to facilitate the traversal techniques described in section 5.

Building the merge tree begins with the original mesh whose vertices comprise the leaf nodes in the tree. Then the lengths of the half-edges in the mesh are calculated and inserted into a red-black tree, which is queried for the shortest edge to be a candidate for collapse. Shortest length is a simplistic metric for selection of edge collapses but the main focus of this work is elsewhere. In areas of fine detail in the mesh, there are typically many short edges, which are collapsed first.

Upon an edge collapse, two vertices in the mesh are merged into one representative vertex. The two original vertices are considered children and the new merged vertex is considered the parent. This parent is then eligible as a child for future merges with other vertices. The merges build the merge tree bottom-up. Information associated with the edge collapse is stored with each node so that the operation may be reproduced, both as an edge collapse, and in reverse as a vertex split.

Each node also contains information that is used to determine its visibility from an arbitrary viewpoint. A node contains the bounding cone of vertex normals for the complete sub-tree below this node. The bounding cone is used at render-time to quickly check whether the sub-tree below this node is forward-facing. Lastly, each node contains a bounding sphere that encompasses the original positions of all vertices in the sub-tree below.

When all edges have been removed from the red-black tree it is discarded. The final collapsed state of the topology is stored as the lowest LOD for a given mesh. Typically, for closed surface meshes, this final state is two points, since two points are not sufficient to define one triangle.

In addition to the parent-child links that exist in a regular merge tree, we introduce child-child links across the span of the tree. These new links facilitate the jump collapse operation described in section 5.2. The child-child links allow quick access to nearby nodes.

4 Producing a Level-of-detail

Recall that the leaves of the merge tree are the original vertices of the mesh. An internal node of the tree represents a coarser vertex which is a representation of all the vertices in the subtree rooted at that node. If some part of the object is to be rendered at fine detail, then vertex split operations in the corresponding part of the merge tree are performed to add detail to the mesh. For the parts that are non-visible, we can choose a much coarser level-of-detail that corresponds to performing those vertex split operations higher up the tree.

When the viewpoint changes in subsequent renderings, it is typically more efficient to refine this existing LOD that to produce a new LOD from scratch. It is necessary to determine how the visibility of the object has changed and hence what parts of the merge tree are visible. Our main contribution in this paper is fast detection of these visible parts of the merge tree.

Like Xia *et al.* [12, 11] and Pajarola [9], we store a LOD as a list of active nodes in the merge tree. To begin, the root node of the merge tree is the only active node. The active nodes list describes a front in the merge tree, that is, the lowest nodes in the hierarchy that are visible. This front moves up and down the merge tree depending on the visibility of the area of the mesh that the merge tree nodes represent. This is shown in Figure 1.

When the viewpoint changes, our main task is to modify the existing active nodes list until it is suitable for the new viewpoint. Once we have the new list we can determine the changes between lists and hence the geometric operations required to alter the existing LOD into a LOD for the new viewpoint. In practice, these two tasks are performed concurrently, the geometric operations are performed as the active nodes list is iteratively modified.

4.1 Normal Cone Visibility

We can decide whether a node n_i is visible by comparing a *view cone* at the current viewpoint and the *normal cone* at node n_i. Recall that the normal cone at n_i repre-

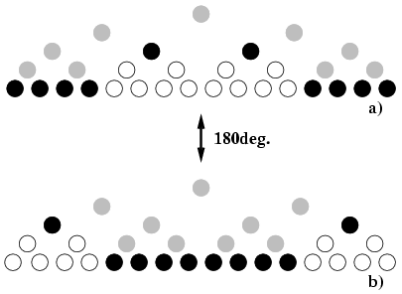

Fig. 1. The movement of the active nodes front in a merge tree. White nodes are invisible, grey are visible and black represent the active nodes. The merge tree in **(a)** and **(b)** represent the same object with a front generated from two viewpoints differing by 180 ˚. The part of the tree representing the visible detail changes with the viewpoint.

sents the union of all the normals of the vertices in the subtree rooted at n_i. We denote the normal cone at n_i as $NC(n_i)$ and the current view cone as VC. There are three possibilities for the comparison of VC and $NC(n_i)$.

i. There is no overlap between VC and $NC(n_i)$. In this case, the part of the mesh in the subtree rooted at n_i is invisible from the current viewpoint. This is typically the case for the back faces of the object. We do not need to split this node further as no node below this node will be visible from the current viewpoint. However, it is possible that this node is not the highest node that is invisible. To check for that case, we collapse this node and its sibling. This collapse is performed using the *COL* operation which is an edge collapse on the half-edge representation of the mesh.
For large changes in viewpoint, many consequtive *COL* operations may be necessary to move the active nodes up to the lowest visible part of the merge tree. However, it is possible to collapse more aggressively to reach a node higher up the tree more quickly. This is done through our *jump collapse* or *JCOL* operation, described in section 5.2.

ii. There is a partial overlap between VC and $NC(n_i)$. In this case, some part of the subtree rooted at n_i is visible from the current viewpoint and some part is invisible. We need to split this node and perform a visibility test on each child. Splitting this node is described by information stored in the node and is performed using a vertex split on the mesh.

iii. VC is completely contained in $NC(n_i)$. In this case, a large part of the subtree rooted at n_i is visible from the current viewpoint. In this case, we need to quickly split many levels in the subtree rooted at n_i. We introduce a new split operation for aggressively generating many nodes from n_i and call it a *jump split* or *JSPLIT* operation, described in section 5.1.

4.2 Bounding Sphere Visibility

We can decide whether a node is visible using the bounding sphere at each merge tree node. Firstly, we can test whether the bounding sphere intersects the view cone. If the bounding sphere is outside the view cone, then the entire sub-tree rooted at this node must also be outside the view cone and the node is declared invisible. Secondly, the bounding sphere can be used to ensure that triangles that are too small to visually contribute to the rendered image are not drawn. The bounding sphere is projected onto the view-plane to determine the upper bound on the projected area of all vertices in the sub-tree rooted at this node. If the area is below a user-defined threshold then the node is declared invisible. Both of these methods are used in FastMesh [9] and we omit the details from this paper.

4.3 Visibility Status

Each of the items in the list of active nodes also has a status record. This status record is used to indicate the operation that was performed to enter this item into the list. The possible states of a node are *COL*, *SPLIT*, *JCOL*, *JSPLIT*, *STABLE* and *UNKNOWN*.

For example, if a node was split into its two children, the children would have the status *SPLIT*. All nodes in the active list are set to the *UNKNOWN* state whenever the viewpoint changes as no actions have been performed on this node. The *STABLE* state is used when it is determined that a node needs no further split or collapse from the current viewpoint.

4.4 Modifying the Active Nodes List

Modifying the list is an iterative process that continues until the status of all active nodes is *STABLE*. Based on the result of the visibility test and the status of the node we can determine whether further processing of a node is necessary. For example, if a node is found to be partially visible and its status is *SPLIT*, then we have not yet split far enough, children of this node may also be visible and must be split. Nodes with *STABLE* status are skipped in subsequent passes over the active node list. When all active nodes have *STABLE* status, no further passes over the list are necessary. Table 1 details the operations that are performed given the status of the node and the result of the visibility test.

5 Optimizations

For any given viewpoint there is a correct front in the merge tree that should be stabilized upon. During LOD refinement, the active nodes list moves from the front used at the previous viewpoint toward the front that is correct for the new viewpoint. In many cases, such as during small incremental changes in viewpoint, these fronts may not differ by a great amount. However, for larger changes in viewpoint it is important to move quickly from one front and stabilize upon the correct new front.

Using existing edge collapse and vertex split techniques, moving the front from a given node many generations higher or lower in the merge tree will require many geometric operations, with a visibility test performed before each. In the interest of speed, it is important to minimize the number of visibility tests that are performed. The jump split and jump collapse operations are an attempt to reduce the number and expense of iterations required to move the front.

	UNKNOWN	JSPLIT	SPLIT	COL	JCOL
invisible	JCOL	COL	NO OP	COL	JCOL
partial	SPLIT	SPLIT	SPLIT	NO OP	SPLIT
full	JSPLIT	JSPLIT	SPLIT	NO OP	SPLIT

Table 1. Operations performed given the status of the node and the result of the visibility test. For example, if the status of the node is *SPLIT* and the node is fully or partially visible then the front has not yet moved to the lowest extent of the visible nodes, more splits are necessary. The mnemonic *NO OP* means no operation is necessary.

5.1 The Jump Split Operation (JSPLIT)

The *JSPLIT* operates on the assumption that if the change in viewpoint is large, then a given node that is ready to split will be likely to have many descendants that should also be split. We can reduce the number of visibility tests performed by doing only one at the top node and then assuming the same result for successive generations in the tree. If the assumption is too ambitious, too many generations may be split and some collapses will be needed.

For a jump downward of n generations, we perform a breadth first traversal of the tree from the current node to a depth of n generations. Figure 2 i) is a downward jump of one generation from node 2. The result of this traversal is a list of the nodes in the sub-tree beneath the current node, in this case 2, 5 and 6. We attempt to split all of these candidates.

Vertex splits must be performed partially ordered top-down in the hierarchy. The breadth first search is used to seed the split candidates list but some jump candidates may not be able to be split without other non-ancestor nodes being split first. These dependencies are caused by geometric restrictions on the topology of the mesh and are well understood [9]. If a split candidate is unable to be performed due to a dependency on another node, this node is inserted into the split candidate list and immediately attempted. Once all split candidates have been split, the jump split is complete.

Note that the purpose of the *JSPLIT* operation is to quickly reach the visibility front. There is a possibility that aggressive use of the *JSPLIT* operation may push the front beyond the visible internal nodes of the tree. Hence, we only use the *JSPLIT* operation when it is clear that the sub-tree nodes are likely to be visible as described in section 4.1.

5.2 The Jump Collapse Operation (JCOL)

The jump collapse operation seeks to aggressively move the front up the tree by identifying groups of adjacent invisible nodes. Instead of collapsing these nodes higher up the tree level by level, it is more efficient to collapse many levels at a time by replacing them with their common ancestor. The common ancestor will serve as a coarse approximation for these invisible nodes.

Typically, when an invisible node is found in the front, it is collapsed along with its sibling, moving the front up the merge tree. When the viewpoint has changed by a large amount, a large part of the object will become invisible and hence some previously visible large areas of the merge tree will have become invisible. The jump collapse takes advantage of this by searching left and right, using the child-child links in the merge tree, for a sequence of adjacent invisible nodes. The search is conducted in both directions only until the first partially or fully visible node is found. The list of invisible nodes are called the collapse candidates. In figure 2, if the jump collapse begins at node 10 it will search left and right in the tree to find the collapse candidates, 8, 9, 10, 11 and 12.

For a collapse to take place, two candidates in the list must have the same parent. If a collapse is performed after consulting the visibility of only one child then the other child may be visible and the jump collapse would be incorrect. Searching the list for

nodes with the same parent is efficient due to the nature of the intra-generational links. Nodes with the same parent occur consecutively, if at all.

The algorithm continues in a loop, looking for sequences in the list, performing the appropriate collapse, then replacing each list item with its parent and looking for longer sequences, until no sequence of sufficient length is found. Then the items in the insufficient sequence are removed from the list and the algorithm considers only the remainder in future iterations. Once all candidates are removed from the list, the jump collapse is complete.

6 Results

Our experiments were performed on a Intel Pentium 4 system at 2.4 Ghz with NVidia GeForce 4 graphics hardware. The implementation runs under Linux, is written in C and uses OpenGL.

At this stage we have experimented with the Stanford Bunny and the Georgia Tech Skeleton Hand. Our main aim was to measure the degree to which use of our *JSPLIT* and *JCOL* operations decreased the time required to refine the mesh after changes in the viewpoint. The timings for refining the mesh are presented in table 2 and represent the duration of 50 viewpoint changes by the specified angle.

Using jump splits of moderate length, two to four generations, decreases the duration of refinement for large changes in viewpoint. When the change in viewpoint decreases, the change in the position of the front in the merge tree is not sufficient to allow the jump split to provide any significant benefit. When jumps of high depth are used, the front is moved too far, past the lowest extent of the visible part. Refinement then uses collapses to undo some of the jump split and the benefit of fewer visibility tests is lost.

Model	Faces	180 ˚	180 ˚ jump	90 ˚	90 ˚ jump	45 ˚	45 ˚ jump
Stanford Bunny	69451	10.82	9.27	7.68	6.91	5.77	5.48
Skeleton Hand	654666	61.58	50.34	47.01	43.39	36.65	33.96

Table 2. Performance of view-dependent refinement of the Stanford Bunny and Skeleton Hand meshes. Times (in seconds) are for 50 rotations of the viewpoint by the specified angle around a vertical axis. Results are shown with and without enhancement by the jump split and jump collapse operations described in this paper.

The jump collapse further reduces the duration of refinements. As the change in viewpoint decreases there is less change in the front and fewer opportunities to save time using the jump collapse. For large changes in viewpoint the jump collapse and jump split reduce the duration of refinement significantly.

7 Acknowledgments

The authors would like to thank both the Stanford 3D Scanning Repository and the Georgia Tech Large Model Archive for making the data sets used in this research pub-

licly available. This work was partially supported by the Western Australian Interactive Virtual Environments Center (IVEC).

References

1. P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, February 1998. ISSN 0097-8493.
2. Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. *Proceedings of SIGGRAPH 98*, pages 115–122, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.
3. Jihad El-Sana and Amitabh Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, September 1999. ISSN 1067-7055.
4. Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization '98*, pages 263–270, October 1998. ISBN 0-8186-9176-X.
5. Ugur Güdükbay, Okan Arikan, and Bülent Özgüç. Visualizer: A mesh visualization system using view-dependent refinement. *Computers & Graphics*, 26(3):491–503, 2002.
6. H. Hoppe. Progressive meshes. *Proceedings of SIGGRAPH 96*, pages 99–108, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.
7. H. Hoppe. View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH 97*, pages 189–198, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.
8. D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics & Applications*, 21(3):24–35, May / June 2001. ISSN 0272-1716.
9. R. Pajarola. Fastmesh: Efficient view-dependent meshing. *9th Pacific Conference on Computer Graphics and Applications*, pages 22–30, 2001.
10. Kevin Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics & Applications*, 5(1):21–40, January 1985.
11. J. C. 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*, 3(2), April - June 1997. ISSN 1077-2626.
12. J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. *IEEE Visualization '96*, pages 327–334, October 1996. ISBN 0-89791-864-9.

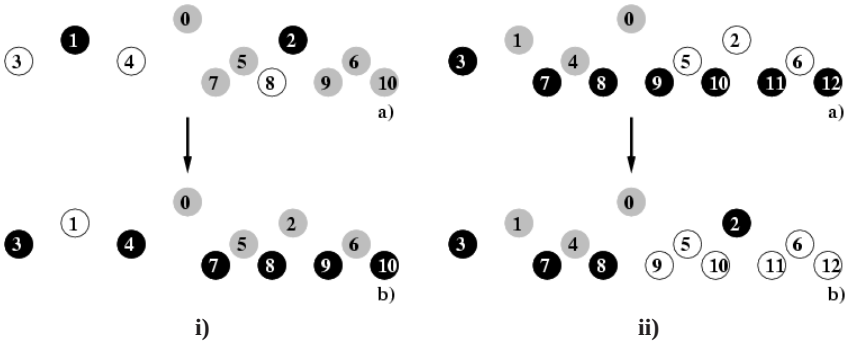

Fig. 2. (i) The Jump Split Operation performed on an example merge tree (a) and the result (b). White nodes are invisible, grey are visible and black represent the active nodes. The jump split is designed to rapidly move the front down the merge tree. The jump split begins at node 2, yet split three nodes $\{2, 5, 6\}$ after performing only one visibility test on node 2. (ii) The Jump Collapse Operation performed on an example merge tree (a) and the result (b). White nodes are invisible, grey are visible and black represent the active nodes. The jump collapse is designed to rapidly move the front up the merge tree. This jump collapse begins at node 10 by searching left and right for other invisible nodes $\{9, 10, 11, 12\}$. The highest common ancestor of only this set of nodes is node 2 so the jump collapse ceases at that node.

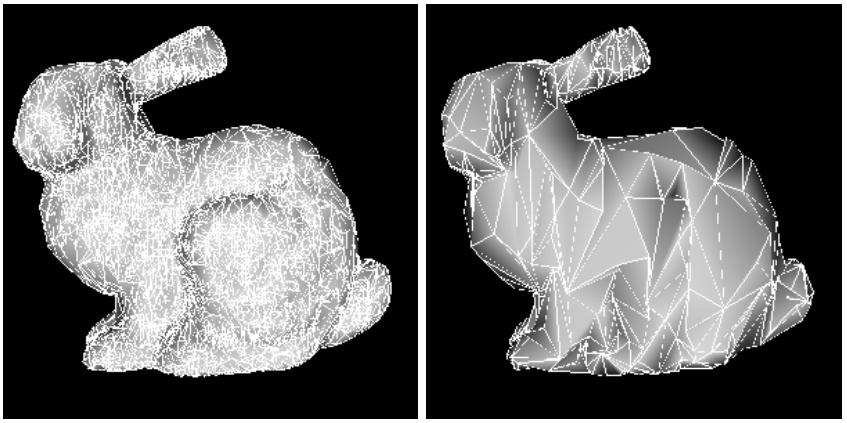

Fig. 3. A view-dependent refinement of the Stanford Bunny. The wireframe of the mesh is also rendered to emphasise geometric complexity. Image a) is a refinement of the bunny seen from the viewpoint, the mesh retains much of its fine detail using 7709 of the 69451 polygons in the original mesh. Image b) is a refinement seen from opposite the viewpoint, the side of the mesh facing away from the viewpoint is coarsely approximated.