Hierarchical Dynamic Simplification for Interactive Visualization of Complex Scenes

Raimund Leitner
ray@oeh.tu-graz.ac.at

Institute of Computer Graphics
University of Technology Graz
Austria

Abstract

The paper describes the implementation of an asynchronous view dependent simplification algorithm based on hierarchical vertex clustering. The clustering is performed by using an octree and observing a screenspace error over all tree nodes. The simplification neither requires nor preserves manifold topology and allows drastic simplification of large polygonal models.

KEYWORDS: view dependent simplification, mesh simplification, level of detail, non-manifold, asynchronous, multi-threaded.

1 Introduction

Polygon meshes are currently the most used model representations for visualization of three-dimensional computer graphics. These meshes are mathematically simple to manage and polygon mesh rendering hardware is widely available. Most continous model representations (splines, bezier curves, volume data) can be approximated with arbitrary accuracy.

Figure 1: Model of the Cassini space probe (415.000 polygons)

The complexity of models for many applications (CAD, medical or scientific visualizations) is too high to render it with available graphic hardware interactively (appropriate framerate). Figure 1 with 415.000 polygons and figure 2 with 500.000 polygons are examples of such very complex 3D scenes. The scenes of the Michelangelo project contain up to 500 millions triangles and current hardware can not display them properly [4].

Generally, three different approaches for simplifying polygon meshes can be distinguished:

Augmenting: Additional details are added to the raw polygon data to achieve a better visual quality. Some examples are Gouraud-shading and texture-mapping.

Culling: Using additional informations about the model, parts of the model are not displayed. These parts can be objects out of view or behind the spectator.

Polygon simplification: Polygons of small, not important or far objects are simplified or temporally erased without a significant loss in visual quality.

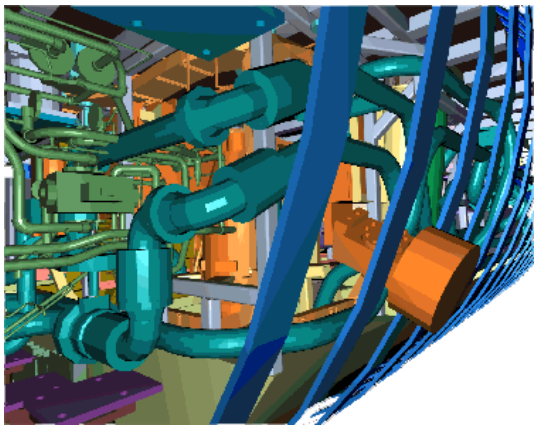

Figure 2: Submarine auxiliary room (500.000 polygons)

2 Simplification Algorithms

In the following some algorithms for polygon mesh simplification are discussed. Quite more algorithms exist than can be mentioned here and, hence, some important and recent algorithms had been chosen [5].

Simplification algorithms can be divided into two classes:

View-independent: The simplification is not dependent on the spectator's position and viewing angle and need not to be recalculated if the spectator moves. The spectator can not be included into the simplification process causing that invisible or far parts of the scene can not be considered differently than near parts.

View-dependent: The simplification considers the spectator's position and viewing angle and has to be partially or fully recalculated if the spectator moves. The advantage of view-dependent algorithms is that invisible or far parts of the scene can be simplified more than near parts. This sometimes increases the visual quality of the result incredibly.

The classification of simplification algorithms by quality appears to be as difficult as the simplification problem itself, because subject impressions of the spectator rarely can be described quantitatively. Jerky moving faces or points disturb the human eye very strongly even if the mortion is quite small. Further, the human eye is very sensitive to discontinuities of the silhouette although a more coarse structuring inside the model is noticed less strongly. Some algorithms already provide ways to consider these specialities of the human perception.

2.1 Triangle Decimation

Triangle Decimation is one of the first simplification algorithms [8]. It was mainly developed for simplifying the results of the Marching Cubes algorithm for isosurface extraction from volume data. The Marching Cubes algorithm often produces too fine triangulated polygon meshes with many coplanar faces. This fact implies that a following triangle decimation seems reasonable.

The Triangle decimation algorithm considers all vertices during some passes. Each pass will delete a vertex, if this does not change the local topology in the neighborhood and the difference of the original mesh and the resulting mesh is not more than an user definable distance. All triangles containing this vertex are deleted and the hole is re-triangulated.

The Triangle Decimation algorithm has the noticeable property not to create new vertices. This property simplifies the reuse of surface normals and texture coordinates, but limits the simplification, because stronger simplification sometimes require the displacement of vertices and/or changing topology. The algorithm can process non-manifold meshes, but does not try to simplify these (non-manifold) parts.

2.2 Re-Tiling Polygonal Surfaces

This method is well applicable for smooth, curved surfaces without sharp edges like organic models [10].

The algorithm starts by distributing an user defined number of vertices onto the polygon mesh. Afterwards repulsion forces between these vertices are simulated, causing that the vertices become almost equally distributed (vertices are only allowed to move on the surface). Now, mutual tesselation creates a surface consisting of old and new vertices. A local triangulation improves the aspect ratio of the triangles. Finally, the original vertices are decimated from the surface to obtain a *re-tiled* model with the new vertices.

2.3 Multi-Resolution 3D-Approximations for Rendering Complex Scenes

[7] describes a vertex-clustering algorithm which neither requires nor preserves valid topology. The algorithm is well suited for degenerated or messy models (e.g. hand-drawn CAD models). Most simplification algorithms rarely produce good results for such models or do not work at all.

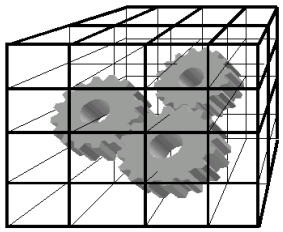

Figure 3: Vertex Clustering

The algorithm starts with assigning each vertex a weight. Vertices of large triangles get larger weights and vertices of smaller appropriate smaller ones. The model's curvature at a vertex is considered with the inverse of the maximum angle between all edge pairs of this vertex. Then a three-dimensional grid divides the scene in many equally sized cells (Figure 3). All vertices inside a cell are clustered to one representative vertex, which is determined from the pre-caclulated weights. The quality of the simplification can be determined by the resolution of the grid.

The algorithm is very stable and one of the fastest of its class, but has some disadvantages. Because topology is not preserved and no explicit error bounds in relation to the model exist, the result can be visually less satisfying than the result of slower algorithms. Additionally the result depends on the orientation of the grid and identical objects with different orientations are simplified differently.

2.4 Simplification Envelopes

Simplification Envelopes are a method for reliable preservation of local and global topology [1].

The Simplification Envelope of a surface consists of two offset surfaces, which are not more than an user defined ε distant from the original surface. The outer offset surface is created by a displacement along the normal vector by ε and the inner offset surface is created by displacing by $-\varepsilon$ (Figure 4). The envelopes are not allowed to self-intersect, which can be avoided by local reduction of ε (numerical more stable) or analytical calculation (computational expense) and correction of the self-intersection.

Simplification Envelopes achieve very good visual quality, because the simplified model can not be more distant than ε. This big advantage is also the biggest disadvantage, because drastic simplification often requires the change of topology.

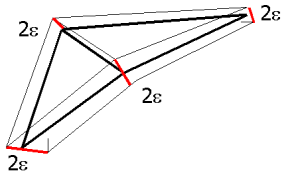

Figure 4: Simplification Envelopes

Further, algorithms for the calculation of Simplification Envelopes are tricky and difficult in programming. This makes robust systems based on Simplification Evelopes difficult.

2.5 Progressive Meshes

Progressive Meshes are a representation of polygon meshes based on *edge collapses* and *vertex splits* and a topology-preserving simplification algorithm for the creation of Progressive Meshes itself [3].

A Progressive Mesh consist of a simple base mesh, which was conducted by consecutive *edge collapses* from the original mesh. A series of *vertex splits* (inverse operation to *edge collapse*) creates stepwise more accurate models until the original model is reached, if all *vertex splits* have beeen executed. The *edge collapse* and *vertex splits* can be used for a continous changing between different simplification levels, because they can be executed very quickly. Further it is possible to interpolate the *edge collapse* and *vertex splits* to achieve softer transitions (geomorphing).

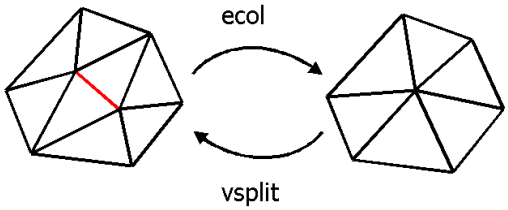

Figure 5: Progressive Meshes

The quality of Progressive Meshes depends strongly on the order of the *ecol* bzw. *vsplit* operations (Figure 5). Hoppe describes a very accurate and expense algorithm for creating the order of the *edge collapses*. The method models the approximation error as an explicit energy function, which is minimized. The influence of all *edge collapses* on the energy function is determined and sorted into a queue. The edge, which minimizes the energy function by the largest amount, is removed from the queue. This changes the influence of *edge collapses* close to this edge and therefore these edges are recalculated and inserted correctly into the queue. This continues until no further simplification due to topological constraints is possible or an user defined threshold is exceeded.

3 Hierarchical Dynamic Simplification

Hierarchical Dynamic Simplification (HDS) uses a Vertex Tree, which is continually queried to generate a simplified scene [6]. The Vertex Tree contains only information about the vertices and triangles of the scene. Manifold topology is not required and is not preserved. The Vertex Tree is built hierarchically and each node contains one or more vertices..

The model is simplified by clustering all vertices inside a node to one single representative vertex. Degenerated triangles are removed from the scene and reduce the number of triangles to display. Reversely the representative vertex will be split into the vertices of its child nodes if the node is expanded. This forces some triangles, which have been removed during collapsing, to re-appear and increases the number of triangles to display.

One possibility to control simplification is to consider the projected size of each node on the picture plane. While the spectator moves through the scene all nodes which become smaller than an user defined threshold are collapsed. Similar, nodes which exceed the threshold are expanded.

Each frame several node collapses and expansions are necessary; this requires a effective data structure containing the information which triangles must be removed from the scene respectively which triangles must be added.

3.1 Active Triangle List

The Active Triangle List contains all triangles, which have to be rendered for the current level of simplification. If the spectator moves through the scene, triangles must be continually added and removed from the Active Triangle List. This implementation uses a doubly linked list, which is able to insert and remove elements in constant time. To remove a certain triangle quickly from the Active Triangle List, the position (pointer) of the triangle is stored.

The triangles itself are represented by the following structure. `Corners` contains the vertices of the original model and `proxies` the vertices of the current simplification.

```
struct Tri {
    Node*   corners[3];
    Node*   proxies[3];
}
```

3.2 Vertex Tree

The Vertex Tree is generated during a preprocessing stage and controls the order of collapsing and expanding the nodes (Figure 6). Further all necessary informations for the collapsing and expanding of nodes are stored in the Vertex Tree to allow fast access to them.

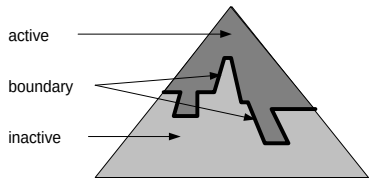

Figure 6: Vertex Tree

This implementation uses an octree-based Vertex Tree. The octree has the advantage of implicit hierarchical spatial information and can be traversed quite fast.

Each node of the Vertex Tree (octree) contains following elements:

```
class CNode {
    int     Label;
    int     Depth;
    Vector  RepVertex;
    Vector  Center;
    float   Radius;
    list    TriList;
    list    SubTriList;
    Node*   Parent;
    Node*   Children;
}
```

Label Indicates the current state of the node. The state can be active, boundary or inactive.

Depth The depth of the node is used for the efficient generation of the TriList and SubTriList during the generation of the Vertex Tree.

RepVertex The coordinates of the representative vertex. All vertices inside a node will be collapsed to this representative vertex, if the node is collapsed.

Center The coordinates of the center of the node.

Radius The radius of the smallest sphere with center *Center*, which contains all vertices of the node. The Screenspace Error caused by collapsing this node is calculated from this radius.

TriList A list of all triangles, which have exactly one vertex inside the node. The vertices of these triangles will be corrected (to the node's representative vertex), if the node is collapsed. Expanding sets the vertices of the triangles to the representative vertex of the first active ancestor node.

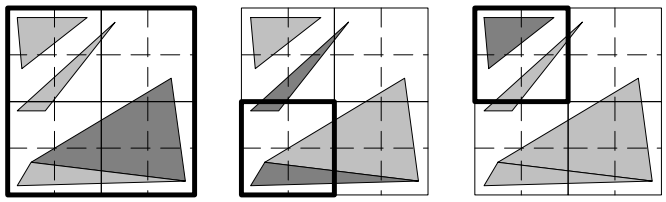

Figure 7: SubTris (dark) of three nodes (bold)

SubTriList A list of all triangles, which have two or three vertices inside the node, but not more than one vertex in every child node (Figure 7). The second condition ensures that each triangle appears at most once in the SubTriList, if an arbitrary path from root to leaf is considered. Hence, the triangles are removed exactly once from the Active Triangle List when the node is collapsed and are added again if the node is expanded. It is impossible that identical triangles are removed or added several times.

Parent contains the parent node.

Children is a list of the child nodes of the node.

The implementation which is the base for this paper uses triangle lists TriList and SubTriList which are dynamic arrays from the C++ Standard Template Library. In contrast to the Active Triangle List, these lists change only during the generation of the Vertex Tree und remain unchanged afterwards. Dynamic arrays are a good compromise between access speed and memory requirement.

3.3 Collapse Node and Expand Node

The two fundamental operations on the Vertex Tree can be described as follows:

```
CNode::collapseNode()
    Label = BOUNDARY;
    foreach child C
        if (C->Label == ACTIVE)
            C->collapseNode()
        C->Label = INACTIVE;
    foreach triangle T in N->TriList
        foreach corner c of {1,2,3}
            T->proxies[c] = firstActiveAncestor(T->corners[c]);
    foreach triangle T in N->SubTriList
        removeTri(T);

CNode::expandNode()
    foreach child C
```

```
    C->Label = BOUNDARY;
Label = ACTIVE;
foreach triangle T in N->TriList
    foreach corner c of {1,2,3}
        T->proxies[c] = firstActiveAncestor(T->corners[c]);
foreach triangle T in N->SubTriList
    addTri(T);
```

3.4 Screenspace Error

Generally, arbitrary error metrics can be used by HDS; this implementation uses a Screenspace Error Metric. Consequently the size of each node projected to the picture plane is observed. If the node's size shrinks below an user defined threshold the node will be collapsed and if the node's size exceeds the threshold it will be expanded again.

Due to this mechanism smaller respectively distant parts which would be smaller than e.g. 2 pixel on the screen, are not displayed.

4 Multi-threaded Implementation

The algorithm can be easily divided into two threads: a Simplify-Thread and a Render-Thread. The Simplify-Thread traverses the Vertex Tree and changes the Active Triangle List accordingly by inserting or removing triangles. The Render-Thread continually renders the triangles of the Active Triangle List. This approach has the disadvantage that holes appear in some frames, if the Render-Thread draws triangles which vertices had been corrected to fit another triangle inside the hole, but the appropriate triangle was not yet inserted to the Active Triangle List by the Simplify-Thread. Synchronizing the access to the Active Triangle List would disgrace the advantage of two consecutive threads.

A detailed consideration shows, that traversing the Vertex Tree requires much more time than the actual collapse and expand operations. Hence, following approach can be developed:

The Simplify-Thread only traverses the Vertex Tree and decides which nodes should be collapsed or expanded. These nodes and the operations are inserted into a queue, which is read from the Render-Thread. The Render-Thread executes the actual collapse and expand operations, corrects the vertices and adapts the Active Triangle List. Then the scene is rendered. This approach requires only synchronized access to the queue and both threads are able to run almost independently.

To ensure a correct behavior, a copy of the node's label (active, boundary, inactive) has to be stored additionally in each node of the Vertex Tree. This copy of the label indicates the current state of the node for the Simplify-Thread. This ensures that nodes, which are waiting for a collapse or expand operation, are not inserted again into the queue and the Simplify-Thread is able to look for other nodes, which should be collapsed or expanded.

5 Conclusions

The presented results were generated using a multi-threaded implementation written in C++ under Windows NT. The following table shows the memory requirement for some models:

Model	Triangles	Memory
Terrain	39.300	42 MB
Bunny	69.451	64 MB
Crater	199.369	186 MB

Most scenes with one or more objects achieve good results. The models can be displayed in almost arbitrary fine simplification levels by adjusting the threshold for the Screenspace Error. The user can choose the level of simplification by adjusting the threshold and obtains an appropriate frame rate or visual quality.

Figure 8 shows a model of the human foot bones in two simplification levels. The upper figure shows the model without simplification and the figure below shows how the algorithm is able to simplify the model by changing the topology (non-manifold). It can be seen, that certain bones *grow* together.

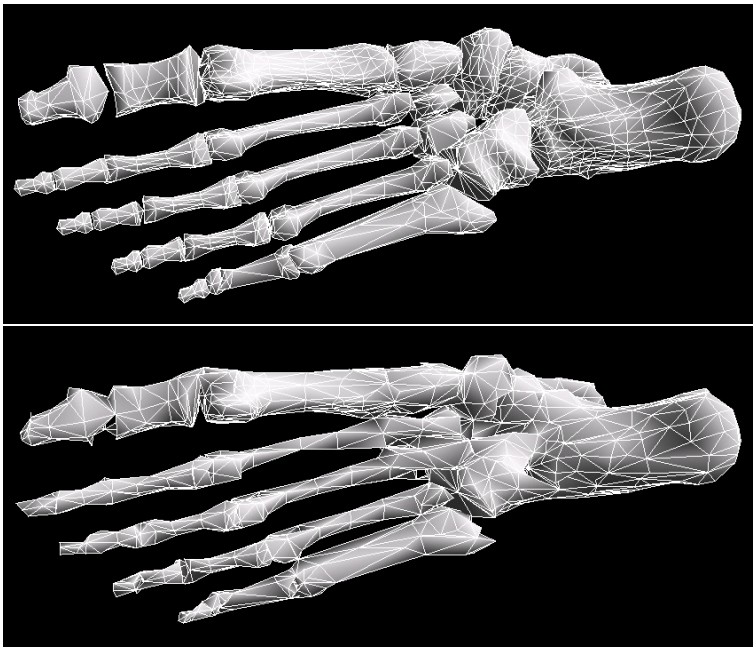

Figure 8: Non mani-fold Simplification

Using a model from the Crate Lake, USA (Figure 9) the performance of the multi-threaded implementation is shown. The test was executed on a Dual-Pentium III 600 MHz system with 256 MB RAM and a NVidia Erazor^2 under Windows 2000 (resolution 1600x1200):

The dual-thread simplification runnning on a dual-processor-system achieves almost twice the framerate as the single-thread variant. Compared to the rendering without simplification a significant higher frame rate is reached, although the visual

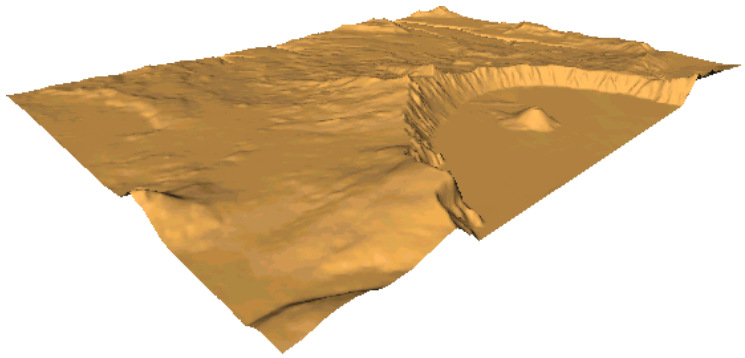

Figure 9: Model of the Crater Lake, USA (199k polygons)

quality is reduced almost unnoticeable. Although the number of rendered triangles is reduced below 20%, the visual quality remains high. The model was displayed to be completely visible and to fill the display window as much as possible.

Model: Crater			
Testcase	Error	Triangles	avg. framerate
Without simplification	-	199k	1.8
Single-thread simplification	1%	30k	11
Dual-thread simplification	1%	30k	21

Problems occur with scenes with several closely located but separated objects (Figure 8). The vertices of different objects are not distinguished during *fold* operations. For high levels of simplification (drastic simplification) this effect is welcome, because the whole scene is displayed very simplified and it is allowed that close object merge. If the effect occurs at high resolutions (slight simplification), the human eye is very sensitive to such simplification artefacts.

Improvements are possible by implementing a silhouette preservation based on a *Cone of Normals* [9] or an advanced error metric like a *Quadrics Error Metric* [2].

References

[1] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *SIGGRAPH 96 Conference Proceedings*, volume Vol. 30. ACM SIGGRAPH, 1996.

[2] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[3] H. Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, volume Vol. 30. ACM SIGGRAPH, 1996.

[4] M. Levoy. The digital michelangelo project. In *Proceedings of the Second International Conference on 3D Digital Imaging and Modeling*, Ottawa, Canada, 1999.

[5] D. Luebke. A survey of polygonal simplification algorithms. Technical Report UNC Technical Report TR97-045, UNC, 1997.

[6] D. Luebke and C. Erikson. View dependent simplification of arbitrary polygonal environments. In *Computer Graphics*, volume Vol. 31. SIGGRAPH 97, 1997.

[7] J. Rossignac and P. Borrel. *Geometric Modelling in Computer Graphics*, chapter Multi-Resolution 3D Approximations for Rendering Complex Scenes, pages 455–465. Springer-Verlag, 1992.

[8] W. Schroeder, J. Zargea, and W. Lorenson. Decimation of triangle meshes. In *SIGGRAPH 92 Conference Proceedings*, volume Vol. 26. ACM SIGGRAPH, 1992.

[9] L. Shirman and S. Abi-Ezzie. The cone of normals technique for fast processing of curved patches. In *Eurographics 93 Conference Proceedings*, volume Vol. 12, pages 261–272. Eurographics 93, 1993.

[10] G. Turk. Re-tiling polygonal surfaces. In *SIGGRAPH 92 Conference Proceedings*, volume Vol. 26. ACM SIGGRAPH, 1992.