

# Parallel Processing City Models for Real-time Visualization

Patrick Cozzi and Dr. Boon Thau Loo  
The University of Pennsylvania

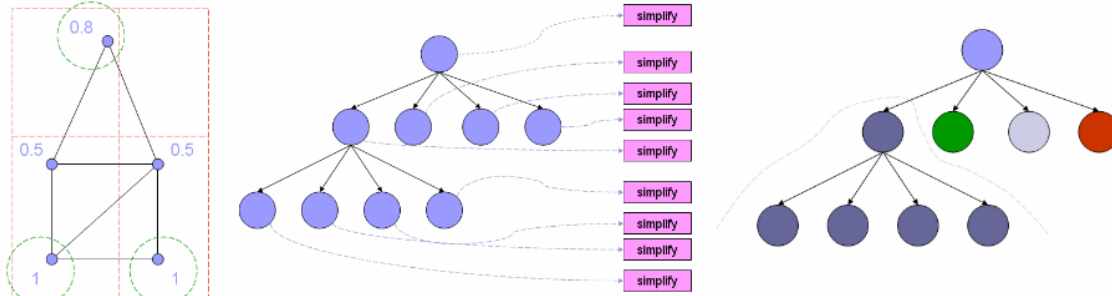


Figure 1 – vertex clustering, polygonal simplification parallelism, and sub-tree parallelism.

## Abstract

We present a parallel algorithm for creating hierarchical levels of detail (HLODs). Our algorithm exploits two orthogonal degrees of parallelism. First, polygonal simplification executes on separate nodes in parallel. In addition, sub-trees are processed and spatially subdivided in parallel. We demonstrate our algorithm with a multi-threaded C++ implementation that takes advantage of multicore processors.

## 1. Introduction

3D models representing cities contain massive amounts of geometry and textures. City models can be acquired using aerial imagery and ground laser scans or generated procedurally [Mueller06]. The resulting models can have polygon counts in the billions; e.g. a procedurally generated, high detail model of Pompeii has about 1.4 billion polygons.

Brute force rendering of city models cannot achieve interactive frame rates even on today's fastest GPUs. Furthermore, entire models are unlikely to fit into system memory. For these reasons, massive models are rendered using hierarchical levels of detail (HLODs) [Erikson01] and out-of-core memory management. Parts of the model

closer to the viewer are rendered with high detail and those further away are rendered with less detail. Besides reducing the load on the GPU, HLODs reduce system memory requirements since only a subset of the model is required to be resident at any one time.

Once a model is acquired, HLODs are computed offline before real-time visualization can take place. This process is slow because of the massive amounts of geometry and textures that needs to be simplified. Preprocessing commonly takes several hours. The main contribution of this work is a parallel algorithm for efficiently creating HLODs.

Our algorithm builds the HLOD in a top-down fashion using vertex clustering [Rossignac93] for polygonal simplification of nodes. Given the input model, the root node is created by drastically simplifying the model in parallel to spatially subdividing the model into 8 children. Each child is the root of a sub-tree that is processed in parallel. The sub-tree is simplified in parallel to its spatial subdivision. The recursion continues until a stopping condition, such maximum tree height, is satisfied

The rest of this paper is structured as follows: we review related polygonal simplification and HLOD work in section 2. Section 3 provides an overview of the vertex clustering algorithm. Section 4 uses vertex

clustering and spatial subdivision to create HLODs. Section 5 extends this serial algorithm to a parallel one. In section 6, we present performance results. Finally, section 7 concludes and suggests future work.

## 2. Related Work

Polygonal simplification is a well studied area in the field of computer graphics. The reader is referred to [Luebke01] for an excellent survey of simplification algorithms. The fastest known algorithm, vertex clustering, was introduced by [Rossignac93] and extended by [Low97] who improved fidelity at the cost of increasing the runtime from linear to  $n \log(n)$ . We chose the original algorithm for its speed and simplicity.

[Erikson01] presents the use of HLODs to accelerate display of massive static and dynamic environments. They use asynchronous simplification for rebuilding parts of the tree when objects move. More recently, [Borgeat05] decouples simplification and segmentation and provide a parallel simplification algorithm using “Quadric with attributes.”

## 3. Vertex Clustering

We chose vertex clustering for polygonal simplification because of its unmatched speed, robustness in handling degenerate meshes, lack of topology assumptions and ability to create drastic simplifications. Although its fidelity is not the best among simplification algorithms, we feel that the runtime system can manage LODs such that the user doesn’t significantly notice.

Figure 2 shows the progression of vertex clustering. Although it is shown operating on a planar mesh, the algorithm runs in 3-space.

An indexed triangle list is provided as input. In addition, the cluster size is provided to determine the degree of simplification. Larger clusters result in more drastic simplifications. The output is a simplified model represented by either a new index list into the original vertices or a new list of vertices and indices. The former is useful for sharing vertex buffers with multiple levels of detail and the later is used for storing the mesh in the nodes of a HLOD tree.

Given the indexed triangle list, weights are computed for each vertex as shown in figure 2.b. Vertices with higher weights are less likely to be removed during simplification. Weights are based on the probability of the vertex being on a silhouette edge and the vertex’s longest incident edge. We stray from the original algorithm and use the  $\cos(\theta / 2)$  metric suggested by [Low97]. We factor out the trigonometry, resulting in the only significant operations being square roots and divides. Although this is expected linear time, section 6 demonstrates that computing the weights is expensive relative to the rest of the algorithm.

After per-vertex weights are computed, vertices are assigned to clusters uniformly distributed within the model’s axis aligned bounding box (AABB) as shown in figure 2.c. To eliminate the cubic memory requirements of a naïve 3D array, we use a spatial hash table with the hash function presented in [Teschner03].

In figure 2.d, the representative

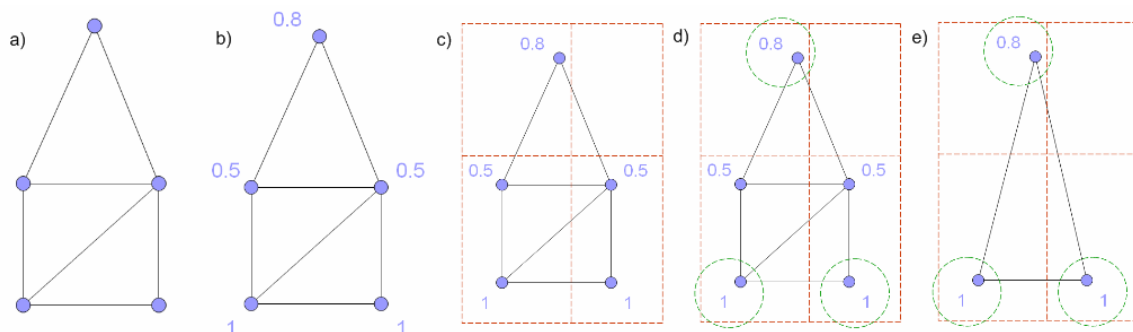


Figure 2 – The steps of vertex clustering.

vertex for each cluster is found. The representative vertex is the vertex with the highest weight. Every vertex in a cluster is then collapsed to the representative vertex. Degenerate triangles are removed to finish the simplification. Figure 2.e shows the simplified model.

#### 4. HLOD Creation

This section describes a straightforward top-down algorithm for creating HLODs. Section 5 extends this algorithm to exploit its nature parallelism. Although vertex clustering is used, the HLOD creation algorithm is largely independent of the polygonal simplification algorithm.

The input is an indexed triangle list and resolution representing the degree of simplification. In addition, the maximum tree height is provided as a stopping condition. Alternative stopping conditions include number of triangles per node and

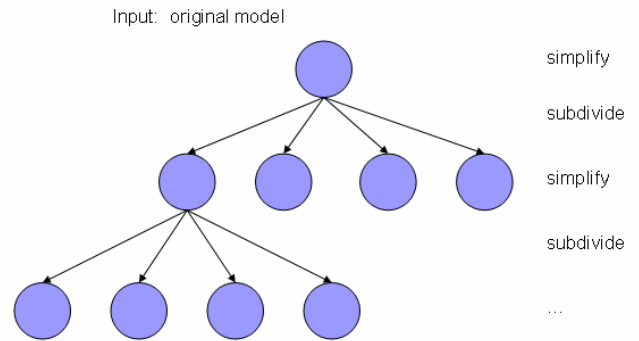


Figure 3 – HLOD tree.

size of a node's AABB. The output is a HLOD tree, where each node contains an indexed triangle list and its AABB. The root node's AABB is the same as the input model. The root's mesh is the most simplified mesh of all the nodes. As you go down the tree, the node's AABB becomes smaller but the meshes are less simplified. Once the leaf nodes are reached, the AABBs are at their smallest and each node contains an un-simplified portion of the original

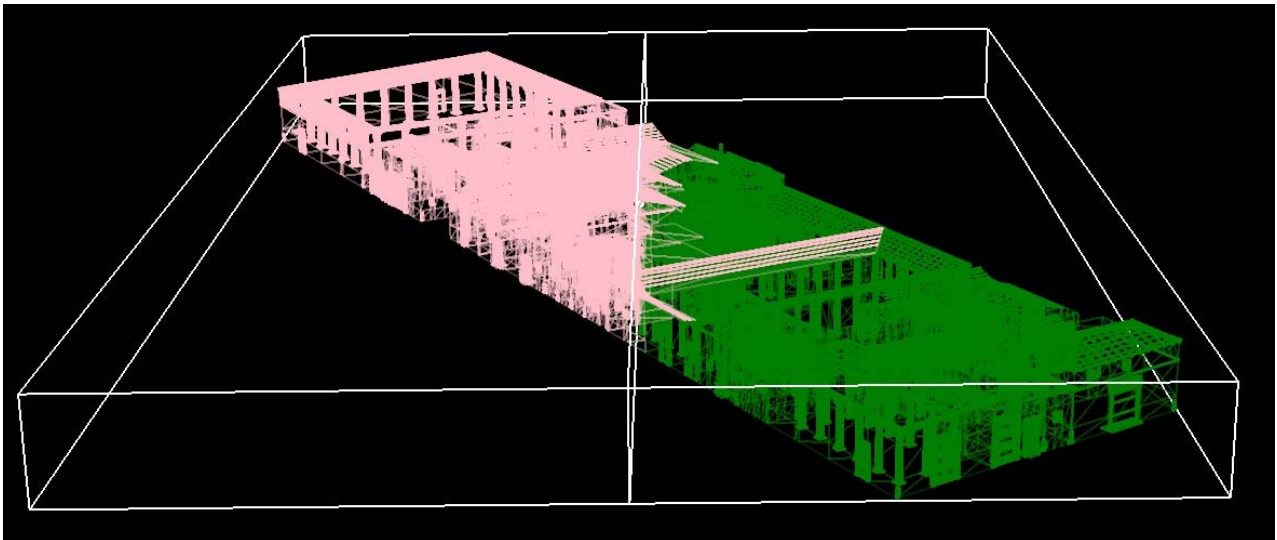


Figure 4 – Two children of a root node.

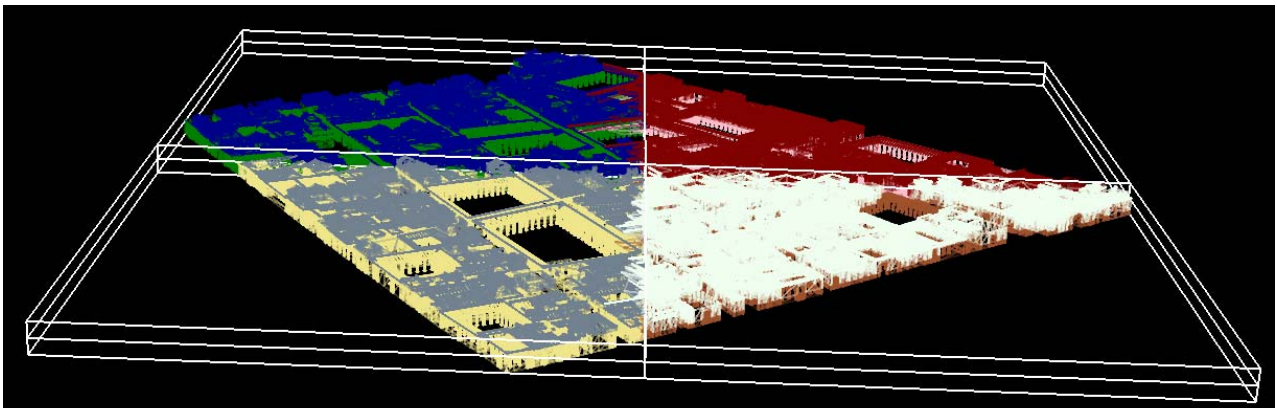


Figure 5 – Eight children of a root node.

mesh. The property of decreasing simplification as you move down the tree is created by using the same resolution, defined as a percentage, for simplification with increasingly smaller AABBs.

Figure 3 shows an example HLOD with 4 children. First, the original model is drastically simplified to create the root node. Next the original model is spatially subdivided into 8 children. In our current implementation, standard octree partitioning is used. 8 child nodes of equal size are created and populated with triangles from the parent that overlap it. When a triangle intersects more than one child, it is inserted into both children. This is evident in figure 4 where only two children are shown. Figure 5 shows the full 8 children of the model used for performance testing.

This recursive simplification and subdivision continues until the desired tree height is reached.

## 5. Parallel HLOD Creation

The top-top HLOD creation algorithm presented in section 4 naturally extends to a parallel implementation. We exploit the fact that separate nodes can be simplified in parallel as well as separate sub-trees can be processed and subdivided in parallel. Furthermore, HLOD creation lends itself to a parallel implementation because of the minimal coordination required among concurrent threads.

### 5.1 Parallel Simplification

Polygonal simplification is executed in parallel as shown in figure 6. The root node starts a thread to simplify the original model. In parallel, the original model is spatially subdivided into its children. The appropriate subset of original model's vertices is copied into each child. This allows polygonal simplification to operation on significantly less data as you progress down the tree. The recursion continues, with each child starting threads for

polygonal simplification. A thread pool is used to limit the number of concurrent simplification threads.

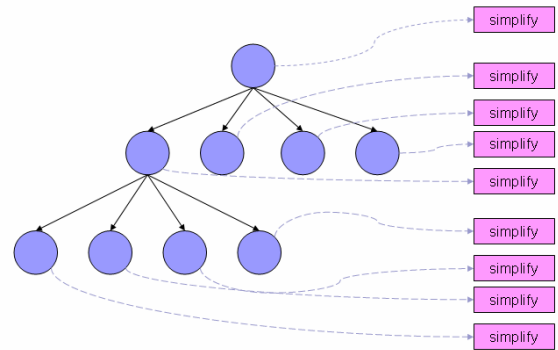


Figure 6 – Parallel Simplification

### 5.2 Parallel Sub-trees

Figure 7 shows an alternative to parallel simplification; processing sub-trees in parallel. First, the root node simplifies the original model. Next 8 children are created. Each child can now be processed in parallel. Subdivision and simplification of the same sub-tree occurs in the same thread but different sub-trees and processed in parallel.

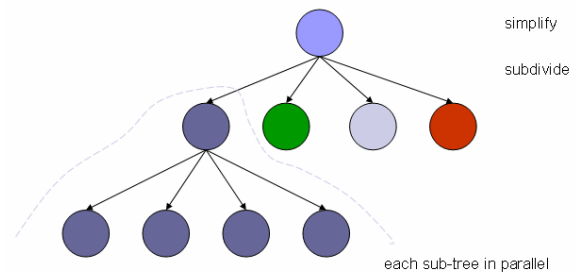


Figure 7- Parallel Sub-trees

### 5.3 Hybrid

Simplification parallelism and sub-tree parallelism can be combined in a straightforward manner to create the hybrid algorithm shown in figure 8. First, the root node starts a thread to simplify the original model. In parallel, the root node is spatially subdivided into 8 children. Each child's

sub-tree is processed in parallel, creating threads for simplification. Since the work at the root is significant, this algorithm exploits the best possible concurrency in the first stage of tree creation as explained in section 6.

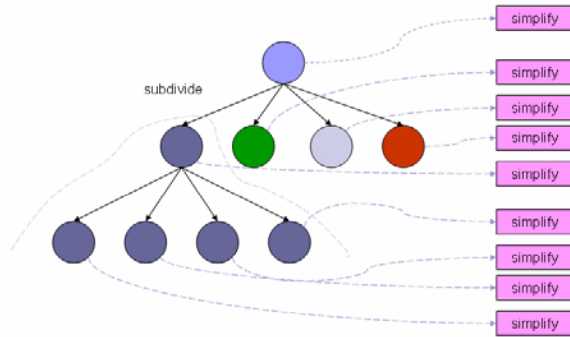


Figure 8 - Hybrid

## 6. Results

We implemented our algorithm in C++ using boost. All tests were performed on a hyper-threaded dual-core Xeon running at 2.66 GHz with 2GB of RAM. The city model was procedurally generated and provided by [Mueller06]. The model, shown in figure 5, is 5,646,041 triangles with 3,394,672 unique vertices and takes up 205,217 KB on disk in the wavefront format. All timings are in seconds.

Table 1 shows constants across all timings that have been factored out of the rest of the timings to isolate the speedup of the parallel algorithm.

Read model into memory	~8.55
Compute per vertex weight	~7.62

Table 1

Our first timing creates a tree of height 4 resulting in 2,603 nodes. Each node is written to disk in the same thread that was used for simplification. The timing breakdown for single threaded HLOD creation is shown in table 2.

Table 3 shows the percentage of time spent on each task when the constants

are factored out. Without the constants, the total time is 49.52

Read model into memory	8.45
Compute per vertex weight	8.01
Total spatial subdivision	13.45
Total polygonal simplification	4.48
Total node write disk IO	31.51
Total	65.98

Table 2

Total spatial subdivision	27%
Total polygonal simplification	9%
Total node write disk IO	64%

Table 3

Consider two observations from the first 3 tables. First, once per-vertex weights are computed, vertex clustering is incredibly fast. It is just 2 passes over the input vertices, 1 pass over the input indices and 1 pass over the output indices. Second, having each node write to disk generates too much disk IO. A production quality implementation should batch up writes to drastically improve performance.

Tables 4-7 show timings for various forms of the parallel algorithm. Since the disk IO takes 31.51 seconds in the single threaded implementation and some amount of single threaded work needs to occur before the parallelism can begin, reducing the time by 20% from 49.52 to 40.94 for using sub-tree parallelism is a worthwhile improvement.

Comparing tables 5 and 6 shows that sub-tree parallelism is best when only the root node starts threads, as opposed to every node adding a thread to the thread pool for each child. The thread management overhead is significant.

Comparing tables 4 and 6 shows that sub-tree parallelism outperforms simplification parallelism. Again, this is likely due to thread management overhead.

None	1 simplification worker	2 simplification workers	3 simplification workers
49.52	42.63	41.6	41.89

Table 4 – Simplification Workers

None	2 sub-tree workers	3 sub-tree workers	4 sub-tree workers
49.52	44.66	44.34	44.75

Table 5 - Sub-tree Workers

None	2 sub-tree workers	3 sub-tree workers	4 sub-tree workers
49.52	40.94	42.04	43.47

Table 6 - Sub-tree Workers, only root starts workers

None	1 simplification worker, 2 sub-tree workers (off root)
49.52	45.35

Table 7 - Hybrid of Simplification and Sub-tree Workers

Although table 7 suggests the hybrid algorithm doesn't perform as well as either of the other parallel algorithms, a modified test case argues otherwise. Since it is obvious that the disk IO dominates HLOD creation and significant improvement can be made to the disk access in our algorithm, we retested without writing nodes to disk. Furthermore, we increased the tree height to a more realistic height of 6 resulting in 95,479 nodes.

Table 8 shows the single thread timing breakdown while tables 9-11 show the total times for the parallel algorithms.

Total spatial subdivision	22.58
Total polygonal simplification	9.61
Total time	32.88

Table 8

None	1 simplification workers	2 simplification workers
32.882	23.65	24.15

Table 9 - Simplification Workers

None	2 sub-tree workers	3 sub-tree workers
32.882	20.46	21.01

Table 10 - Sub-tree Workers, only root starts workers

None	1 simplification worker 2 sub-tree workers (off root)	2 simplification workers 2 sub-tree workers (off root)
32.882	19.32	19.51

Table 11 - Hybrid of Simplification and Sub-tree Workers

With disk IO removed, all parallel algorithms significantly outperform the single threaded implementation. The hybrid algorithm achieves the best improvement, reducing HLOD creation time by 41%. This is due to the parallel simplification and subdivision at the root, where the most amount of geometry has to be processed. Given the isolation of each thread, we expect this algorithm to scale well with an increased number of cores.

## 7. Future Work

We've demonstrated a parallel algorithm for creation of HLODs. Our algorithm exploits both polygonal simplification and spatial subdivision parallelism and results in significant performance gains on multi-core processors.

There are many directions for future work. A higher fidelity simplification algorithm such as floating cells [Low97] or one based on quadric error metrics [Garland99] can replace vertex clustering. An adaptive subdivision scheme that finds a more optimal splitting plane than octree partitioning can be incorporated. More sophisticated handling of triangles that intersect multiple children without introduced cracks during rendering should also be considered. All of these improvements increase the CPU requirements and will benefit from our parallel algorithm.

For truly massive models, an out-of-core algorithm is called for. It may be possible to combine this work and the work of [Lindstrom00] to create a parallel out-of-core algorithm.

Finally, this algorithm builds HLODs top-down. It may be efficient to parallelize a bottom-up algorithm.

## 8. References

[Borgeat05] Borgeat, L., Godin, G., Blais, F., Massicotte, P., and Lahanier, C. "GoLD: Interactive Display of Huge Colored and Textured Models" Proceedings of ACM SIGGRAPH 2005.

[Erikson01] Erikson C., Manocha D., and Baxter, W. V. III. "HLODs for faster display of large static and dynamic environments." Proceedings of the 2001 symposium on Interactive 3D graphics, pages 111-120. ACM Press, 2001.

[Garland99] Garland, Michael. "Quadric-Based Polygonal Surface Simplification." PhD Thesis. Carnegie Mellon University, 1999.

[Lindstrom00] Lindstrom, P., "Out-of-Core Simplification of Large Polygonal Models." ACM SIGGRAPH 2000, pages 259 - 262

[low97] Low, K.-L. and Tan, T.-S. "Model Simplification using Vertex-Clustering".

Proceedings of 1997 Symposium on Interactive 3D Graphics, April 1997, pages 75 – 82.

[Luebke01] Luebke, D. "A Developer's Survey of Polygonal Simplification Algorithms" IEEE Computer Graphics and Applications. 2001.

[Mueller06] Mueller P., Wonka P., Haegler S., Ulmer A., and Van Gool, Luc. "Procedural Modeling of Buildings." Proceedings of ACM SIGGRAPH 2006.

[Rossignac93] Rossignac, J. and Borrel P. "Multi-Resolution 3D Approximations for Rendering Complex Scenes," Geometric Modeling in Computer Graphics, Springer-Verlag, Berlin, 1993, pages 455-465.

[Teschner03] Teschner, M., Heidelberger, B., Mueller, M., Pomeranets, D., and Gross, M. "Optimized spatial hashing for collision detection of deformable objects." Proceedings Vision, Modeling, Visualization. 2003. pages 47 - 54.