Multi-grained Level of Detail Using a Hierarchical Seamless Texture Atlas

Krzysztof Niski* Johns Hopkins University
Budirijanto Purnomo† Johns Hopkins University
Jonathan Cohen‡ Lawrence Livermore National Laboratory

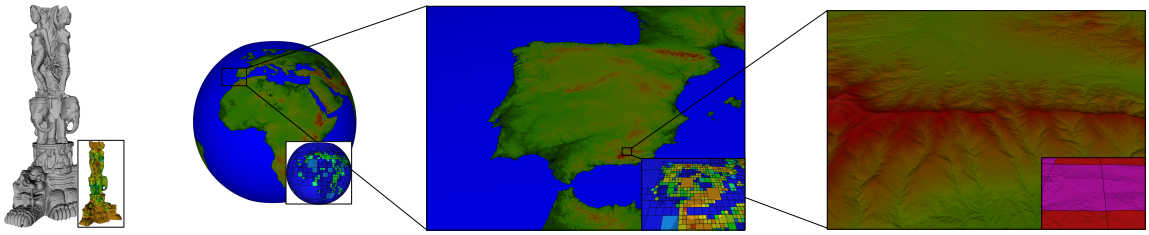

Figure 1: Renderings of the 36 million sample Thai statue model and the 22 billion-sample USGS Earth data set with node layouts and triangle densities. Tessellations range from sparse in blue through green, yellow and red to dense in purple. (Black regions represent missing samples in the input Earth data set.)

Abstract

Previous algorithms for view-dependent level of detail provide local mesh refinements either at the finest granularity or at a fixed, coarse granularity. The former provides triangle-level adaptation, often at the expense of heavy CPU usage and low triangle rendering throughput; the latter improves CPU usage and rendering throughput by operating on groups of triangles.

We present a new multiresolution hierarchy and associated algorithms that provide adaptive granularity. This multi-grained hierarchy allows independent control of the number of hierarchy nodes processed on the CPU and the number of triangles to be rendered on the GPU. We employ a seamless texture atlas style of geometry image as a GPU-friendly data organization, enabling efficient rendering and GPU-based stitching of patch borders. We demonstrate our approach on both large triangle meshes and terrains with up to billions of vertices.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, object hierarchies

Keywords: level of detail, texture atlas, parametrization, geometry image, out-of-core

1 Introduction

Since the early days of interactive 3D computer graphics, the need to represent complex 3D geometries at multiple levels of detail has been apparent [Clark 1976]. In the interim, CPU performance, main memory capacity, and triangle processing performance have all increased dramatically. However, despite these gains, the need for level of detail has increased in recent years.

The increased need stems from ever-improving 3D data acquisition methods. The scales of these modern 3D scanners, such as SIR (space-borne imaging radar), Lidar (light detection and ranging), standard laser range finding, photometric techniques, etc., range from meters down to millimeters down to tens of microns or finer, depending on the technology and the scale of the target model. As a result, large 3D models today contain hundreds of millions to billions of samples, which include not only the geometric position, but normals (measured or computed), colors, and other material properties.

Most recently, the highest performance systems have used a very coarse-grained approach, minimizing the CPU processing required while maximizing the triangle throughput of the graphics hardware. However, even this latest breed of algorithms has it limitations. In particular, the *granularity* (i.e., the number of triangles forming an atomic unit) at which the level of detail is adjusted at run-time is constant and fixed in advance, and in many cases a great deal of processing goes into computing the hierarchy for that one, particular granularity.

In this paper, we propose a mesh representation that is *multi-grained*. It is both spatially hierarchical and multi-resolution, and these two aspects are managed independently and dynamically rather than bound together during pre-processing. At run-time, it is possible to adjust the resolution of any individual hierarchy node, or to split or merge nodes. As we show in our analysis, this is beneficial for two reasons. First, the data structure is appropriate for any balance of CPU/GPU processing powers. This allows us to control both the CPU and the GPU usage, making our method adaptable to varying hardware configurations. Second, even for a given CPU/GPU combination, there may be no single, ideal granularity. The best granularity may depend on local characteristics of the surface and the viewing parameters (e.g., location of the view frustum). Thus, it is desirable to allow a spatially-adaptive granularity.

We develop our multi-grained hierarchy in the context of *hierarchical, seamless texture atlases*. For arbitrary-topology input meshes, these atlas domains are constructed out-of-core through a process of patchification and parametrization. Alternatively, for regular height field inputs, the domain is constructed through a simple partitioning process. Given the atlas domain, geometry may be stored as a three-channel geometry image or a single-channel height image as appropriate, and the same domain is used to store attribute textures such as color and normal maps.

Given this new hierarchy structure, we define the corresponding new problems for real-time geometry adaptation, allowing an application to specify not only an error threshold or a triangle budget, but also simultaneously a maximum number of nodes to be rendered. This last parameter directly impacts the time required for the CPU to perform the adaptation. We present algorithmic solutions to these

*e-mail: niski@cs.jhu.edu
†e-mail: bpurnomo@cs.jhu.edu
‡e-mail: jcohen@llnl.gov

adaptation problems, and also discuss the seamless rendering of the resulting geometry.

Our new approach to level of detail has a number of desirable properties:

- **Load management:** The load on the CPU and GPU is managed independently by setting the maximum node count and maximum triangle count, respectively. This unique ability of our hierarchy provides an extra degree of freedom to our *quad-queue* adaptation algorithm over existing algorithms.
- **GPU-based border resolution:** Our implementation employs vertex textures to deliver geometry to the vertex processing unit, enabling the stitching of neighboring patch borders directly on the GPU.
- **Rendering-optimization-friendly:** Due to the regular grid structure of mesh patches the rendering primitives are relatively easy to optimize, producing triangle strips with excellent vertex cache coherence. Even in the presence of vertex texturing we have seen performance in excess of 100M triangles/second.
- **Reusable (implicit) topological data:** On the current hardware generation, we can store in texture memory reusable, regular grids of various resolutions, storing only (u,v) vertex coordinates and the corresponding index lists. This data is reusable across the entire model (and across all models). On future GPU architectures, it may well be possible to generate this underlying topology-driven data on the fly.
- **Loosely constrained hierarchy neighbors:** Compared to most quad-tree LOD hierarchies, we have few restrictions on the hierarchy level or resolution level of two neighboring surface patches. Seamless borders are achieved for neighboring patches even if they differ by several hierarchy levels and/or resolution levels.
- **Fragment-level attribute preservation:** Our general parametric approach allows preservation of mesh attributes in texture maps. Thus their resolution (and their corresponding footprint in texture memory) is determined independently of the load on the vertex processing unit.
- **Coherent data redundancy:** Given a child node that covers a subset of its ancestor's domain, we can temporarily use the ancestor's data to render the child, regardless of the required resolution. This enables our method to render any cut of the model without cracks or missing data while the correct data is loaded.

We demonstrate our approach on several large meshes and terrains with up to billions of vertices. We examine some of the benefits of the increased flexibility of our multi-grained hierarchy and look at rendering output and performance of our current prototype system.

2 Related Work

2.1 View-dependent Level of Detail

View-dependent level of detail algorithms allow localized changes in the resolution of a polygonal mesh according to the current viewing parameters. Early view-dependent algorithms [Xia and Varshney 1996; Floriani et al. 1997; Luebke and Erikson 1997; Hoppe 1997] used a tree or DAG structure to allow very fine-grained modifications to the mesh according to some error metric. This ability of view-dependent algorithms to operate at various locales across a mesh is especially important for rendering of terrains, which are typically vast in scale [Lindstrom et al. 1996; Duchaineau et al. 1997; Hoppe 1998].

For today's large data, algorithms must generally deal with issues of out-of-core operation. It is possible to apply fine-grained, view-dependent level of detail in an out-of-core setting [El-Sana and Chiang 2000; Lindstrom and Pasicco 2001; Lindstrom 2003]. However, the most recent algorithms generally apply changes in mesh resolution in a very coarse-grained fashion, seeking to minimize CPU usage while maximizing the triangle throughput of the GPU [Ganovelli et al. 2004; Borgeat et al. 2005; Cignoni et al. 2005; Hwa et al. 2005].

Our algorithm seeks a balance between the fine-grained and the coarse-grained representations by providing an adaptable granularity, and thereby providing the ability to balance CPU and GPU usage. As compared to [Borgeat et al. 2005] in particular, our algorithm performs patch border stitching on the GPU and allow multiple levels of resolution difference between neighbors as opposed to restricting to a single level difference.

Applications such as Google Earth [2005] perform a task similar to our system, with several key differences. The chief of these is their inability to opearate on general models, reducing their applicability to terrain datasets only. These systems also focus on tertiary data such as satellite imagery or street maps rather then the underlying geometry. As a result, they give no guarantees on error threshold or triangle count, replacing geometry with high-quality textures. While these systems are very effective at their task, the method presented in this paper strives to be more general and rigorous.

2.2 Geometry Images

Like the geometry clipmap approach to rendering large terrains [Losasso and Hoppe 2004; Asirvatham and Hoppe 2005], our hierarchical format stores geometric data in a form of geometry image. A geometry image [Gu et al. 2002] is essentially a two-dimensional array of (x,y,z) values. A mesh is defined by the implicit regular-grid structure of the array. A number of methods exist for constructing a geometry image by resampling an arbitrary-genus [Gu et al. 2002] or genus-0 [Praun and Hoppe 2003] polygonal mesh. It is also possible to construct geometry images of multiple charts – either regular [Purnomo et al. 2004] or irregular [Sander et al. 2003]. The simple topology of regular grids makes geometry images appealing for many forms of geometry processing, including compression and rendering.

Our data format takes its direction from the work of [Purnomo et al. 2004]. Each of our highest resolution geometry images is a single chart of their *seamless texture atlas*. However, we impose a hierarchical node structure on each of these charts. Multiple nodes may thus cover a chart, and each can select an appropriate resolution for rendering. Furthermore, we develop an out-of-core approach to constructing the texture atlas for application to large meshes. This choice of data format brings the rendering of arbitrary topology surfaces much closer to the domain of terrain rendering and makes it possible to produce seamless boundaries between adjacent nodes of different hierarchy levels and resolutions. Unlike the geometry clipmap approach, our algorithm provides error-guided adaptation, seamless patch boundaries, and the ability to trade CPU workload for rendering quality.

Another approach similar to ours on the surface is the work of [Ji et al. 2005]. Like us, they employ a form of seamless geometry image on the GPU using a quad-tree for level of detail. However, their approach is applied to models several orders of magnitude smaller than ours, restricts the quad-tree adaptation to a single level difference between neighbors, and is based on charts created through a manual process.

3 Hierarchical Seamless Texture Atlas

Our multi-grained level of detail hierarchy consists of a forest of quad-trees, each of which is built on the domain of a single square chart of a seamless texture atlas. A one-tree hierarchy is illustrated in Figure 2. The texture data α (a height map in this case) is filtered to resolutions β, γ, δ, and ε. The chart domain is hierarchically

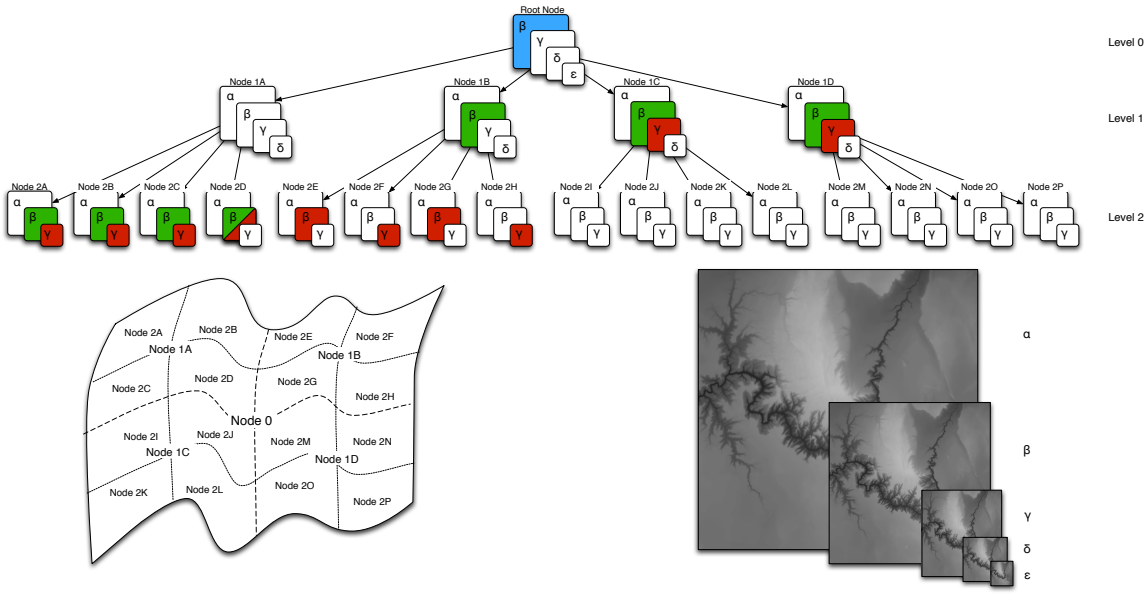

Figure 2: One chart of a *hierarchical seamless texture atlas.* Each node of the quadtree spatial hierarchy has access to multiple resolutions of the geometric data, allowing separate, dynamic adjustment of node spatial extent and resolution. Geometric data for each chart is stored separately in a image pyramid. The blue and green cuts produce the same number of triangles and the same maximum error. The red cut maintains the maximum error bound, but uses fewer triangles, as not all areas have the same geometric error.

subdivided using a quad-tree structure. Each node of the quad-tree covers a particular region of the chart domain and has access to several image resolutions for that region.

Ideally, every node could access every resolution, but there are a few practical limitations to this. For example, nodes at level 1 cannot access texture resolution ε because that resolution has too few samples to be split amongst the four nodes of level 1 (because ε has the smallest allowable node texture resolution). Similarly, nodes at level 2 cannot access texture resolutions ε or δ, because the δ resolution is too small to split across the nodes of level 2. Access to a resolution may also be restricted because it is too large. For example, the root node (level 0) cannot access resolution α. This is due to the fact that our implementation assumes a single texture state and draw call per node, and the hardware only supports textures up to some fixed, maximum resolution. Given these two restrictions, a node has access to

$$r = \min(\log_2(M) - l, \log_2(R)) \quad (1)$$

resolutions, where l is the level of the node, M is the maximum resolution of this chart's texture, and R is the maximum texture resolution supported by the hardware.

This hierarchy structure is a departure from the traditional tree structure employed in LOD systems. In a more traditional LOD tree, each node represents one particular level of detail. One can subdivide a node into its sub-nodes to refine the object or merge nodes into their parent to coarsen the object. In our hierarchy, a node can be refined not only by subdividing it into its sub-nodes, but also by increasing its choice of resolution for its region of coverage.

Given this new degree of freedom, notice that there are actually multiple cuts through the tree and resolution choices that achieve the same error bound and triangle count. For example, the blue cut contains one node, and the green cut contains 7 nodes. However, both cuts render the chart entirely at resolution β. In general, the blue cut would be considered superior because it achieves the same result with fewer nodes, and our algorithm would ultimately merge the nodes of the green cut up to the single blue node if that resolution was really appropriate everywhere. The more typical red cut exposes the benefit of our method: by subdividing a node we can often use one or more lower-resolution sub nodes while maintaining the same geometric error bound.

As a result there are many ways to reconstruct a model with a given error threshold or triangle budget by varying the number of nodes used. This permits the balancing of CPU and GPU workloads by adjusting the number of nodes parameter independent of the error or triangle parameter.

It is worth noting here that at first glance, this ability to store multiple levels of detail at each node may resemble the structure employed by the HLOD algorithm [Erikson et al. 2001]. However, in that work, the tree represents a scene graph, and merging children nodes into the parent implies merging the representations of multiple distinct objects. Each node in that structure does store multiple levels of detail, but there is still only single cut that can achieve any particular scene triangulation. Thus it is not possible in that system to use the desired number of nodes to control the CPU load of the adaptation algorithm independent of the triangle count.

4 Seamless Texture Atlas Construction

In previous work [Purnomo et al. 2004], we introduced the seamless texture atlas as a parametric domain for texturing arbitrary-topology meshes with guaranteed $C0$ continuity in the presence of texture mip-mapping and geometric level of detail. It consists of a collection of quadrilateral charts which cover the surface. We showed that by sampling surface attributes on the charts of such an atlas using a 1-pixel overlap on all the boundaries, it is straightforward to maintain continuity. That process has the following steps:

1. **Cluster:** Using a combined metric incorporating planarity and compactness, iteratively merge triangles into clusters using a greedy heuristic. The result is a collection of polygonal patches.
2. **Quadrangulate:** Partition each n-sided patch into n quadrilateral patches. This process connects a central vertex of each patch to a central vertex on each of the patch boundaries, reminiscent of the first level of Catmull-Clark subdivision.

3. **Parameterize:** Parameterize each quadrilateral patch onto the unit square domain using an efficient, sparse, linear-least-squares solution to a uniform spring system followed by an iterative algorithm optimizing an area-preserving texture stretch metric.
4. **Resample:** Capture attributes, such as position, color, or normal by uniform sampling in the square domain of each chart. Align the samples with the domain boundaries to ensure 1 ring of replicated texels around the patch.

We have adapted that original process for out-of-core operation to enable processing of larger meshes. It is primarily the initial clustering phase that requires modification. We perform a two-phase clustering as follows:

1. **Gridify:** Use a uniform 3D grid to partition a large, unindexed collection of triangles into multiple files.
2. **Per-cell Cluster** For each grid cell, perform geometric vertex hashing followed by in-core clustering. The goal is for the union of clusters of all grid cells to fit in core. Each cluster stores only aggregate information used to compute the combined error metric: quadric error matrix, surface area, and per-boundary-edge length. In addition, triangles that cross cell boundaries are stored with their current cluster for use in the next step.
3. **Global Cluster:** Load coarsest level patches from all cells together. Perform geometric vertex hashing among boundary-crossing triangles to compute shared patch boundaries and their lengths. Cluster patches until desired number (or cost metric threshold) is reached.

Following the second clustering pass, perform quadrangulation, parametrization, and resampling on a per-patch basis. Note that during these steps, as well as the per-cell clustering, the computation is trivially parallelizable and may be performed on a large compute cluster if necessary.

For regular height map inputs, the preceding parametrization algorithm is unnecessary. We simply partition the height map into charts of the desired maximum node resolution, maintaining the expected 1-pixel overlap between adjacent charts.

5 Hierarchy Creation

For each chart in our texture atlas, we create a quadtree hierarchy, starting with a root node. Each root node is subdivided into four children nodes in the texture domain, and each these is further subdivided, and so on, down to a pre-specified subdivision depth. For atlases with multiple charts, the root nodes are initialized with a pointer for each of their four boundary edges to the root node of the adjacent tree. As we subdivide, this information provides the foundation for computing all node neighbor relationships during the view-dependent adaptation algorithm.

We associate with each node in the hierarchy a region of the chart domain that is one quarter of its parent's region. The associated textures are created to be of resolution $(2^n + 1)$x$(2^n + 1)$, where n is the LOD level of the data. These "power of two plus one"-resolution textures have the desirable property that when one splits them into quadrants with a one-pixel shared interior boundary, their children's resolutions are the next smaller power of two plus one. This is convenient for crack elimination between adjacent patches on the surface.

The hierarchy generation process also computes bounding boxes for each node and error values for each level of detail. We measure the error for a given level of detail by considering the distance from each of the original samples in the node's region of coverage from the corresponding point in parameter space on the simplified mesh. The error is calculated by interpolating a corresponding vertex position from the four nearest vertices from the simplified mesh, and calculating the distance to the original vertex, thus giving a geometric deviation for that point. Our current implementation uses the maximum operator to combine the sample errors for each level of detail, but the average operator is an equally valid choice, depending on the needs of the application.

The hierarchy building stage is easily separable, as the computations for each node are independent of neighboring nodes. As a result the preprocessing stage is easily multi-threaded, allowing for a significant improvement in preprocessing performance, especially on multiprocessor (or multi-core) machines.

6 Interactive Rendering

Given the complete multi-grained hierarchical data structure, the major components necessary for interactive rendering are algorithms for view-dependent adaptation, a scheme for data management, methods of rendering the selected patches, and an approach to stitching together the boundaries of adjacent patches at different resolutions and hierarchy levels.

6.1 View-dependent Adaptation

Traditional view-dependent LOD methods have only one degree of freedom – they can only increase the detail of a node by subdividing the node into its more densely tessellated children. Thus, the number of triangles and the number of nodes are typically tied together by a roughly constant number of triangles per node.

The method presented in this paper is free from these restrictions, allowing the application to select both the desired amount of detail in terms of error thresholds or maximum triangle count, as well as the number of nodes used to render the object. While this allows for more flexibility in the system, it also requires a new method for adapting the mesh to the specified detail thresholds based on the current viewing parameters. In the standard LOD formulations, two common problems statements are: (1) "Given a maximum error threshold (object space, screen space, etc.), compute a mesh which minimizes the number of triangles without exceeding the error threshold," and (2) "Given a maximum triangle budget, compute a mesh which minimizes the error without using more triangles than the budget allows." For each of these problems, our new degree of freedom adds to the problem formulation: "...given a maximum number of allowable nodes."

Consider the first problem with our new amendment. A simple top-down algorithm might work as follows. Start with the minimum number of nodes (the root nodes) on the active cut, each at its lowest level of detail. Refine each node to the first level with an error beneath the error threshold. If there are more nodes available in the budget, place the current nodes on a priority queue for splitting. The split priority is set to the number of triangles that would be saved if the node was split and each of its children adapted to the error threshold. Iteratively remove a node from the queue, split it, adapt its children and place them on the split queue until the node budget is exhausted.

A more efficient, coherent algorithm for this problem is a variant of the well-known dual-queue algorithms [Duchaineau et al. 1997; Luebke and Erikson 1997]. Two queues, the split and merge queues, hold every node that is currently on the cut. The split priority is computed as above. The merge priority is the number of triangles that would be added as a result of merging siblings up to their parent. We perform a set of merge, split and adapt operations until no more benefit is to be gained.

Now consider the second problem statement. This one requires the balancing of detail for individual nodes so that the best possible choice is made for the entire mesh. In the traditional hierarchy, where refining and splitting a node are synonymous, as are coarsening and merging, a dual-queue approach can solve this using a greedy heuristic.

In our case however, we need to control refining and coarsening resolutions as well as splitting and merging nodes. We propose a new *quad-queue* algorithm to perform this optimization. The queues are organized as two dual-queue pairs: the split/merge queues and the refine/coarsen queues.

Each of the nodes of the current cut appears on all four queues, divided into two phases. In the first phase, as in the standard dual-queue algorithm, we refine/coarsen the current nodes so that (a) they are within the triangle budget, (b) no node can be refined without going over the budget, and (c) the refine and coarsen queues are balanced (i.e., they minimize error).

In the second phase we propose a set of splits and merges of nodes, the order of which is determined using the merge and split cost heuristics. We then split and merge the nodes until (a) the node count is below the maximum allowable node count, and (b) the next (least bad) merge will require more triangles then can be saved by performing the next (best) split.

We then proceed back to the refine/coarsen process where we adapt the new hierarchy node cut so that it fits within the user-specified limits. At the end of each refine/coarsen iteration we store the maximum geometric error in the cut, which is then used to evaluate the effectiveness of the split/merge iteration. If the new error is lower then the error before the split/merge process, we know that a better node layout has been selected, and we perform another iteration of the split/merge process to further improve the cut. If, on the other hand, the geometric error has increased, we know that we have made the node layout worse, and we roll back the last set of split/merge operations and terminate the adapt process. Although our current split/merge heuristic does not guarantee improvement at every iteration, a more thorough optimization would be prohibitively expensive as it might have to search the whole space of possible split/merge and refine/coarsen operations to find the best choice.

Given the final set of nodes and their associated geometric resolutions generated by the adaptation process, we also compute an appropriate resolution for any additional attribute textures, such as normal maps or color textures, using the projected screen-space size of the nodes' bounding volumes and a desired pixel-to-texel size ratio. These textures are then placed on the request stack for loading and management.

6.2 Data Management

Similar to many large rendering systems, our system maintains least-recently-used caches of data in both video memory and main memory, with non-resident data being fetched asynchronously in a separate thread according to a priority queue.

However, our system has some unique capabilities in terms of data redundancy and reuse. While a node awaits some particular resolution of data for rendering, it may be temporarily rendered using any other resolution of the node itself or of one of its ancestors, all of which cover its entire domain (and this has some associated temporary effect on the triangle count and the visual error). This is made possible by the use of the quad-tree structure, which allows for the trivial mapping of a child node into its parent. Furthermore, if the mesh has a complete representation in frame i, we are assured of having a complete, usable representation for frame $i+1$, regardless of which data updates arrive on time.

The use of this redundant data is inexpensive storage-wise (on disk, the data are stored in blocks with each resolution level only represented once), and the least-recently-used cache replacement policy ensures that textures are replaced in a timely fashion after their replacements arrive. As an exception to the LRU policy, we also find it convenient to lock the lowest resolution texture for the level-zero nodes in memory to ensure there is always some fast-rendering representation available for the entire model (this low-resolution data occupies less then one tenth of one percent of GPU RAM for the 22 billion sample Earth data set).

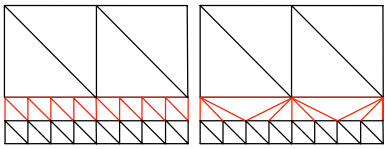

Figure 3: Border stitching is performed when nodes of different tessellations share a border. In this example the nodes are two LOD levels apart, forcing vertices in the more densely tessellated node to be collapsed in order to match the lower resolution node.

6.3 Patch Rendering

Our patch-rendering approach pre-computes a set of uniform (u,v)-grids, each a triangulated plane in 2D, to feed to the vertex processing unit. Each grid is a power of two plus one resolution to match the resolution of the geometry images containing the actual (x,y,z)-coordinates. These grids are stored in video memory and are reused for all rendered patches.

On receiving a (u,v)-grid vertex, the vertex processing unit looks up the (x,y,z)-coordinates from the geometry image using a vertex texture lookup, performs the necessary transformations, and sends the results down the graphics pipeline. In the fragment unit the color and normal maps can be applied to further enhance the visual quality of the resulting image. The geometry and attribute image map resolutions are managed independently, allowing higher-resolution color and normal data to be used where necessary, while still retaining lower-resolution geometry data.

The use of regular grids of vertices allows us to perform some very simple but effective optimizations. The most important of the optimizations performed on the (u,v) grid is the organization of mesh rendering into triangle strips of adjacent columns. Adjusting the strip length to roughly half the vertex cache size of the hardware minimizes vertex cache misses, achieving close to the optimal condition of executing the vertex program only once per vertex (or 0.5 vertex program executions per triangle). A second important optimization is the use of minimal per-vertex data sent to the GPU. By using the (u,v) grids we can minimize the data to two floating point values per vertex, achieving close to maximum performance from the GPU.

The use of the 2D input data also allows us to perform additional mesh operations on the GPU, such as mesh spherification for terrains, which allow us to send height maps to the GPU and render a spherical earth. Our method thus allows the data to be stored in a more compact representation, requiring $\frac{1}{6}$ the storage when using unsigned short height maps.

Although the vertex texturing method provides a much cleaner solution to rendering, we have also implemented a VBO version, which uses the CPU to perform border matching and vertex arrays for the renderer instead of vertex texture lookups. The VBO method has the advantage of using the fastest rendering path in current generation GPUs, resulting in improved rendering throughput at the cost of a somewhat higher CPU workload.

6.4 Border Stitching

The use of uniformly-tessellated 2D planes allows us to reconstruct the original model without cracks even in the presence of different LOD neighbors. To this end we duplicate one row and column in each geometry image so that any two neighboring nodes share an identical border for the same image resolution.

To stitch the border between adjacent nodes with different resolutions we calculate the texel selected by the lower-resolution neighbor, and force the current vertex to select the same texel. This guarantees that both vertices will pick the same sample from

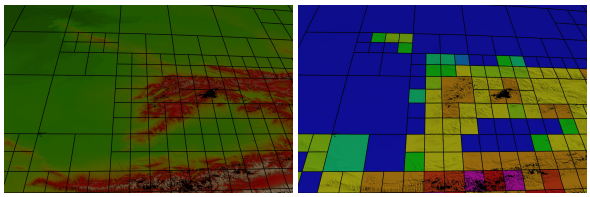

Figure 4: The triangle density is represented by the color of each node, ranging from blue for very low density to purple for highly-tessellated nodes. Our system is capable of adaptively selecting the resolution of each node, permitting large LOD differences between adjacent nodes.

the geometry image, matching the higher-resolution to the lower-resolution border and eliminating cracks as shown in Figure 3. Traditionally this process would be performed on the CPU because the GPU cannot access the vertex coordinates of its neighbors. However, because the vertex coordinates are stored as geometry images and sampled on the GPU, we can access any sample by changing the texture coordinates of the lookup, allowing a node's sampling to match a lower resolution neighbor.

We see a more complex situation in Figure 4. In this example from the Earth data set we see a very coarse flat area surrounded by densely tessellated mountains, as well as two levels of hierarchy depth difference on a node border. Because we do not restrict the hierarchy depth differences between neighboring nodes, a node may have multiple neighbors along a single border edge. To handle this general case we send an array containing the resolutions of the neighboring nodes along all four node boundaries to the node being rendered. This enables the node to match the higher-resolution segments of its border to its lower-resolution neighbor. This data is constant per node and is sent to the GPU using constant registers, leaving vertex data unaffected.

While this process changes the geometric error of the node, it does not change the overall geometric error of the model. Although the error is increased along the boundary of the higher-resolution node, the maximum error in the scene remains unaffected since the new error along the border cannot exceed the error level of the lower-resolution neighbor.

However, the border stitching does impose one new restriction on the selection of node resolutions: a node's resolution may not be so low that it could have more neighbors on a single edge than it has boundary vertices for them to match.

7 Experimental Results

We have applied our algorithms to several terrain and mesh data sets, including the USGS Earth (21.6 billion samples), USGS North America (5.5 billion samples), Puget Sound (16k x 16k), Grand Canyon (2k x 4k), Thai statue (10M polygons), and cuneiform tablet (1M polygons).

7.1 Atlas Construction

The cuneiform tablet model contains 1M triangles. Running on a AMD dual-Opteron 2.4GHz, the model took 16 seconds for gridify, 30 seconds for in-core clustering, 30 seconds for global clustering, 15 seconds for quadrangulation, 1 hour for parametrization, and 2 minutes to resample the 30 resulting charts at 512x512 samples per chart (coordinates and normals).

The Thai statue model contains 10M triangles. Running on a AMD dual-Opteron 2.4GHz, the model took 2 minutes to gridify, 5 minutes for in-core clustering, 2 minutes for global clustering, 3 minutes for quadrangulation, 6 hours for parametrization, and 5 minutes to resample the 138 resulting charts at 512x512 samples per chart (coordinates and normals).

Model	Samples	Trees	Size (MB)	Time (min)
Earth-flat	21.6 B	70	17	120
Earth-sphere	21.6 B	70	17	718
N. America	5.5 B	15	3.5	29
Puget Sound	268 M	4	1	1.1
Thai Statue	130 M	140	32	1.2
Tablet	8 M	30	6.8	.05

Figure 5: Hierarchy creation information. Samples are scalars for the height fields and 3D positions for the general models. Output size refers to the hierarchy node data and not the actual sample data. All of the quadtrees have a depth of 5.

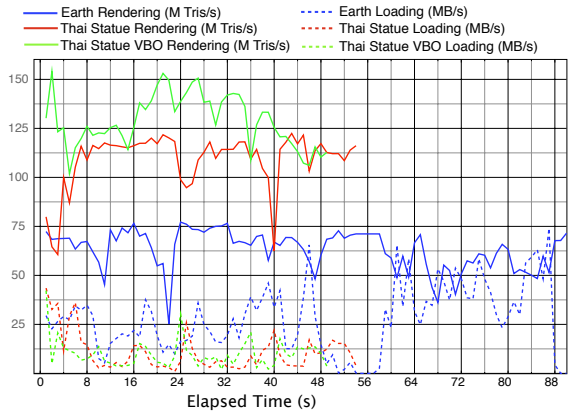

Figure 6: Rendering throughput over 2,000-frame paths, using a budget of 4M triangles and 350 nodes. The rendering rate exceeds 50M triangles/s for the earth data set (with GPU spherification) and 100M triangles/s for the Thai statue model, even in the presence of large data transfers to the GPU. The average frame rate is approximately 15fps for the earth data set and 20fps for the Thai statute model.

In practice, we achieved a nearly 40x speedup by performing the parametrization stages on a 20-node Intel dual-Xeon 3.2GHz cluster. Thus, the total wall-clock time for the cuneiform tablet was roughly 4 minutes and the total time for the Thai statue was roughly 30 minutes.

7.2 Hierarchy Creation

The hierarchy creation stage of our method was performed on a quad-core 2.66GHz Mac Pro system running OSX using 8 simultaneous threads using less than 500MB of memory. The models processed are shown in Figure 5. For the spherical version of the Earth model, the additional time was required to convert from rectilinear coordinates to spherical coordinates for the purpose of bounding sphere and error computations. Note that for the spherical earth, we have chosen to minimize storage size by maintaining the data as a height (radius) field and spherifying the data on the GPU.

7.3 Rendering

We have evaluated the rendering performance of our system as well as its ability to adapt to various hardware configurations using the same hierarchy. All of the tests were run on a system with dual 2.4GHz AMD Opteron CPUs and a NVIDIA Quadro 4500 with 512 MB VRAM.

Figure 6 shows the performance of the system while following a path over the 22 Billion Earth data set. From the graphs we can see that our method can reliably render at over 50M triangles/s even when loading large amounts of data due to both the out of core

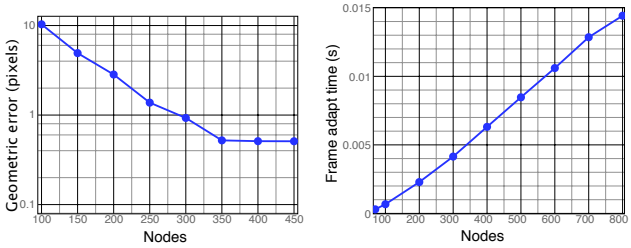

Figure 7: Effect of node count on screen-space error (left) and adapt time (right). Both graphs use the triangle budget mode with a 2M triangle budget to render the Earth data set.

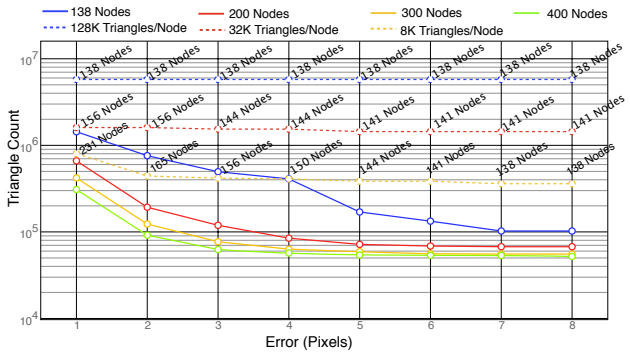

Figure 8: This graph compares our method, in solids, against fixed-granularity LOD methods drawn using broken lines. The graph shows the number of triangles required by each of the methods to render the model at a given screen-space error as well as the number of nodes required to render the model for the fixed resolution methods. As shown in the graph our method requires fewer triangles to render the scene, especially when additional nodes are used.

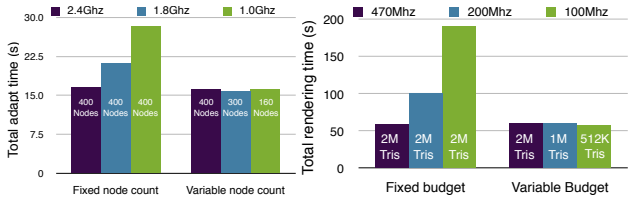

Figure 9: Throttling the CPU and GPU to simulate different machine configurations. **Left:** Using a 2M triangle budget, we adjust the node count to maintain a fixed adapt time in the presence of the throttled CPU. **Right:** Using a fixed node budget, we adjust the triangle budget to maintain a fixed rendering time. Thus the node and triangle budgets may be used to adapt the overall system to the particular CPU/GPU combination.

loading and redundant data layout. Note that for the spherical earth model, we have opted to store the model as a scalar (radius) field and compute the 3D coordinates on the fly in the vertex unit. The reduces storage and bandwidth requirements but has some significant impact on throughput. For standard height fields or geometry images, we regularly see throughput over 100M triangles/second with the vertex texture lookups and border stitching, and up to 150 M triangles/s when using a VBO implementation of our method (which does stitching on the CPU and no vertex texture lookups).

In Figure 7 we show the number of nodes versus the average adapt time and the number of nodes versus the average screen-space error in pixels for the Earth data set. Notice that there is a clear trade-off between the CPU time and the number of nodes as well as between the number of nodes and the error. Both of the experiments were run with a triangle budget of 2M triangles. Given a desired node (CPU) budget, our method allows an intelligent selection of nodes while maintaining either a maximum triangle count or maximum error bound.

A similar trade-off is performed when adapting to an error threshold, shown in Figure 8. Looking at just the solid lines at any particular error threshold, we see that using more nodes significantly reduces the number of triangles required to meet that error threshold. In practice, this means that increasing the number of nodes to an appropriate level increases the frame rate of the system (the adaptation stage is generally pipelined with the rendering stage). Note, however, that there is also some system-specific trade-off here due to the decreasing batch size of the rendering calls as we increase the number of nodes.

We have compared our multi-grained approach to the fixed-granularity methods used in previous systems by disabling our system's ability to refine and coarsen the nodes independently of the split/merge process. This forces our system to split a node to refine it, and to merge sibling nodes to a parent to coarsen them. The results for the adapt process, run in error threshold mode, are shown in Figure 8. The solid lines indicate our system run in its intended multi-grained fashion, with each line indicating a different node count setting. The dashed lines indicate the emulation of the fixed-granularity systems, with each line indicating a particular number of triangles per node. Note that in typical fixed-granularity system, each such granularity would require an entirely new hierarchy.

In general, we see that the availability of more nodes in the multi-grained system even as the error threshold is increased enables better adaptation, producing fewer triangles for a given error threshold than the fixed-granularity approach. As an example scenario, consider that when the error threshold is increased, our multi-grained system can coarsen individual patches as necessary without being forced to merge siblings (which may often increase the error prohibitively). Notice in the figure that at 1 pixel of error, the fixed-granularity approach with 8k triangles/node produces 2.6 times more triangles than our multi-grained approach with 400 nodes, and at 8 pixels of error, it produces 7 times more triangles.

In the same figure, we see that our system even outperforms the fixed-granularity approach when comparing particular adaptations using the same number of nodes (and the fixed-granularity approach in practice has little control over the number of nodes used). At 8-pixels of error, the 8k triangles/node adaptation uses 138 nodes, but produces 3.5 times more triangles than our multi-grained adaptation with 138 nodes. Similarly, we see that the multi-grained adaptation with 138 nodes produces the same number of triangles at 4 pixels of error as that 8k triangles/node adaptation does at 8 pixels of error. So the new approach produces an adaptation with half the error using the same number of nodes and triangles.

To test the flexibility of our system we simulated varying machine configurations by throttling both the CPU and the GPU performance of the computer. In the Figure 9 we throttle the 2.4GHz AMD Opteron CPU to three different settings, 2.4GHz, 1.8GHz and 1.0GHz, by using available power-saving modes. The data shows that it is possible to achieve a desired performance on a various CPUs by adjusting the number of nodes parameter to the adapt routine.

Next we test the GPU load balancing capabilities of our system by throttling the GPU to 470MHz, 200MHz and 100MHz using GPU overclocking tools. By adjusting the triangle budget only we can control the rendering rate of our system in order to adjust for various levels of GPU performance. Our system as a whole is thus capable of balancing the GPU workload independently of the CPU workload, allowing it to take advantage of a larger number of nodes

while maintaining a steady frame rate.

8 Conclusions

We have presented a series of data structures and algorithms, as well as a prototype system, that explore a number of ideas and trends for high performance rendering on evolving graphics hardware. Our approach incorporates ideas from seamless geometry atlas parametrization, geometry image resampling, and quad-tree-based hierarchical rendering to expand on the state-of-the-art in level of detail systems. The resulting multi-grained, multi-hierarchy system builds on the previous work by expanding the adaptation options for every node, allowing them to not only split/merge, but also to refine/coarsen. This added flexibility allows not only for better adapt performance, reducing the triangle count for a given error threshold, but also for the balancing of the CPU and GPU workloads. As shown in our results, through adjusting the node and triangle budgets the user can control the amount of work performed by the CPU and the GPU, respectively. The flexibility of the system allows it to be more adaptable to the hardware configuration of the host and to reduce the triangle to error ratio by adaptively selecting node resolutions. By adaptively selecting resolutions our system can produce more detailed meshes using fewer triangles then previously possible on a fixed node budget.

Our system also uses the latest abilities of GPUs to take advantage of their ever-increasing performance, and the increasing flexibility of the vertex and fragment processors. The use of features such as geometry images and vertex texturing provides an elegant solution to the border stitching problem and allows our method to be easily extended in the future.

Finally, a number of additional features common to high-performance rendering systems could be incorporated with our proof-of-concept implementation in the future to produce a system that is both very flexible and has even better performance. These include features such as accurate data pre-fetching, data compression, back-patch culling, geomorphing and occlusion culling. The resulting system would be even more capable of fully harnessing the power of the latest GPUs. In our opinion the proposed approach represents a promising visualization technique due to its flexible adapt algorithm, GPU-friendly data representation and easy extensibility.

9 Acknowledgments

We would like to thank the USGS, Stanford University Graphics Laboratory, and National Research Council of Canada for the models used in our experiments. This research was sponsored in part by NSF Medium ITR IIS-0205586, a DOE Early Career Award, and an NVIDIA Fellowship (the views expressed in this work are not necessarily those of our sponsors).

References

ASIRVATHAM, A., AND HOPPE, H. 2005. *Terrain rendering using GPU-based geometry clipmaps*. GPU Gems 2. ch. 2, 27–46.

BORGEAT, L., GODIN, G., BLAIS, F., MASSICOTTE, P., AND LAHANIER, C. 2005. GoLD: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3, 869–877.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2005. Batched multi-triangulation. In *IEEE Visualization '97*, 27–35.

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10, 547–554.

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. Roaming terrain: real-time optimally adapting meshes. In *IEEE Visualization '97*, 81–88.

EL-SANA, J., AND CHIANG, Y.-J. 2000. External memory view-dependent simplification. *Comput. Graph. Forum 19*, 3, 139–150.

ERIKSON, C., MANOCHA, D., AND BAXTER, W. V. 2001. HLODs for faster display of large static and dynamic environments. In *ACM Symposium on Interactive 3D Graphics*, 111–120.

FLORIANI, L. D., MAGILLO, P., AND PUPPO, E. 1997. Building and traversing a surface at variable resolution. In *IEEE Visualization '97*, 103–110.

GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH 2004*, 796–803.

GOOGLE, 2005. Google earth. http://earth.google.com.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. In *SIGGRAPH 2002*, 355–361.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH 97*, 189–198.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98*, 35–42.

HWA, L. M., DUCHAINEAU, M. A., AND JOY, K. I. 2005. Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Trans. Vis. Comput. Graph 11*, 4, 355–368.

JI, J., WU, E., LI, S., AND LIU, X. 2005. Dynamic LOD on GPU. In *Computer Graphics International 2005*, 108–114.

LINDSTROM, P., AND PASICCO, V. 2001. Visualization of large terrains made easy. In *IEEE Visualization 2001*, 363–370.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. A. 1996. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH 96*, 109–118.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *ACM Symposium on Interactive 3D Graphics*, 93–102.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. 23*, 3, 769–776.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97*, 199–208.

PRAUN, E., AND HOPPE, H. 2003. Spherical parametrization and remeshing. *ACM Transactions on Graphics 22*, 3 (July), 340–349.

PURNOMO, B., COHEN, J. D., AND KUMAR, S. 2004. Seamless texture atlases. In *ACM/Eurographics Symposium on Geometry Processing*, 65–74.

SANDER, P. V., WOOD, Z. J., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *ACM/Eurographics Symposium on Geometry Processing*, 146–155.

XIA, J. C., AND VARSHNEY, A. 1996. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96*, 327 – 334.