Multi-grained Level of Detail for Rendering Complex Meshes Using a Hierarchical Seamless Texture Atlas

Krzysztof Niski Budirijanto Purnomo Jonathan Cohen*

Johns Hopkins University

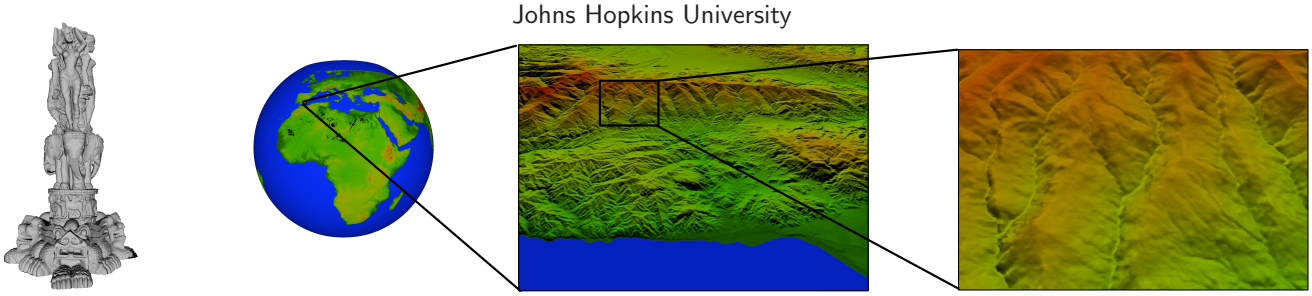

Figure 1: Example renderings from the 36 million-sample Thai statue and the 22 billion-sample USGS Earth data sets.

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 minimizes the triangles to error ratio, often at the expense of heavy CPU usage and low triangle rendering throughput; the latter improves CPU usage and rendering throughput at the expense of the triangles to error ratio.

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. 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, parameterization, 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 [4]. 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 not decreased, but has in fact increased.

The increased need stems from ever-improving 3D data acquisition methods. The resolutions 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. Thus, large 3D models today may contain hundreds of millions of samples, which may include not only the geometric position, but normals (measured or computed), colors, and other material properties.

Most recently, the highest performance systems use 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* at which the level of detail is adjusted at run-time (i.e., the number of triangles refined together as an atomic unit) is fixed in advance, and in many cases a great deal of processing goes into computing the hierarchy for that one, particular granularity.

Main contribution: In this paper, we propose a mesh representation which is *multi-grained*. It is both hierarchical and multi-resolution, and these two properties are managed independently 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 with respect to 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 parameterization. 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 color and normal maps.

Given this new hierarchy structure, we define the corresponding

*e-mail: {niski,bpurnomo,cohen}@cs.jhu.edu

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 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 are 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 stitching of neighboring patch borders directly on the GPU.
- **Rendering-optimization-friendly:** Due to the regular grid structure of mesh patches, rendering is relatively easy to optimize. Producing triangle strips with excellent vertex cache coherence is straightforward. 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:** Hierarchy nodes in our system contain a superset of their ancestor's data but cover a subset of their domain. Ancestor nodes are easily used temporarily to render any of their descendants without introducing cracks while the most appropriate data resolutions are being 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 [23, 8, 19, 11] 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 [16, 5, 12].

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 [6, 17, 15]. 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 [9, 2, 3, 13].

Our algorithm seeks a balance between the most fine-grained and the most coarse-grained representations. We provide an adaptable granularity, and thereby provide the ability to balance CPU and GPU usage. As compared to [2] 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.

2.2 Geometry Images

Like the geometry clipmap approach to rendering large terrains [18, 1], our hierarchical format stores geometric data in a form of geometry image. A geometry image [10] is essentially a two-dimensional array of (x,y,z) values. A mesh is defined by according to the implicit regular-grid structure of the array. A number of methods exist for constructing a geometry image by resampling an arbitrary-genus [10] or genus-0 [20] polygonal mesh. It is also possible to construct geometry images of multiple charts – either regular [21] or irregular [22]. 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 [21]. 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 selecting 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 [14]. 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 Overview

Our algorithm for large mesh rendering has three phases: seamless texture atlas construction, hierarchy creation, and interactive rendering. For general meshes, we perform an out-of-core parameterization of the mesh to generate atlas charts and resample the mesh to

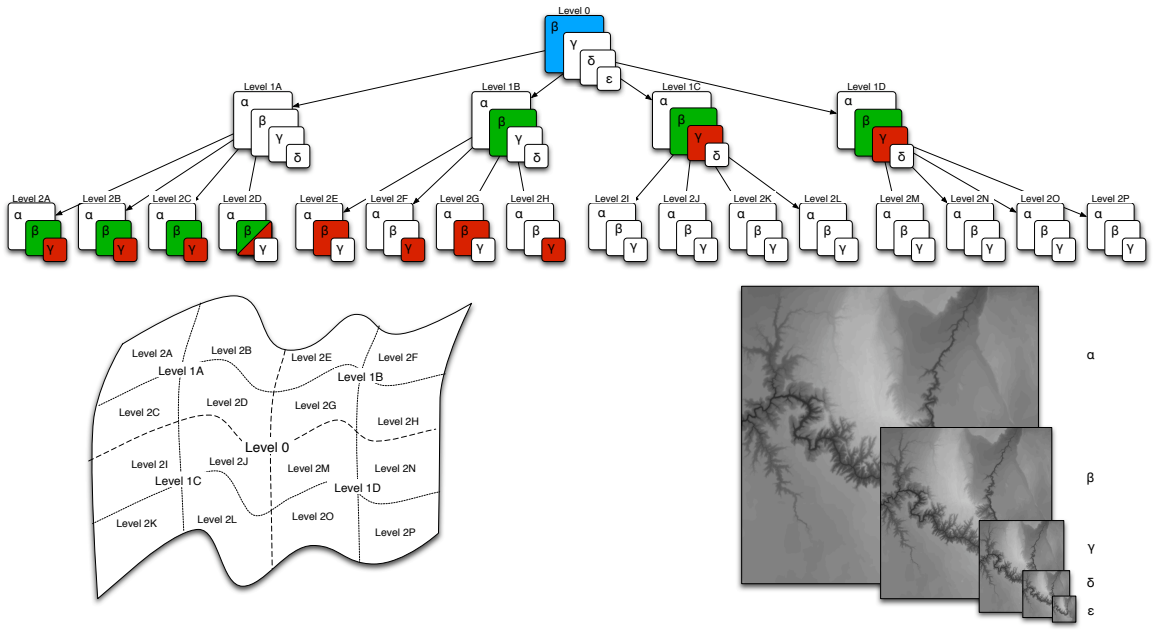

Figure 2: **Hierarchical seamless texture atlas.** The blue and green cuts produce the same number of triangles and the same maximum error by only varying the subdivision level. The red cut maintains the maximum error bound, but uses fewer triangles, as not all areas have the same geometric error.

form a geometry image for each chart. For height maps, we merely rearrange the input data into chart images for more convenient data management. For each chart, we also construct an image pyramid of the geometry data and form any additional attribute images, such as color and normal textures.

The hierarchy creation process builds a quad-tree hierarchy on top of each chart, resulting in a forest of quad-trees. Each quad-tree node covers a region of a particular chart. The process computes and stores with each node an error value for each available resolution of the node's region. In addition, each node stores a bounding volume, and each root node stores a pointer to the neighboring root node on each of its four sides.

The data generation and hierarchy construction phases are performed once for each mesh and the results are stored on disk for use during interactive rendering. At rendering time, our adaptation algorithm performs a local optimization every frame to select: a set of nodes (a *cut* through the trees) that covers the model, resolutions for rendering each node's geometry using vertex textures, and resolutions for applying any auxiliary attribute textures. The optimization is guided by parameters such as the maximum number of nodes and either a maximum number of triangles or a maximum screen-space error. Given the results of the adaptation, the rendering system checks for the availability of the required geometry and auxiliary data and loads it in a separate thread as necessary. The nodes are then rendered using the currently available textures by feeding a uniform (u,v)-grid to the vertex processing unit, performing vertex texture lookups to fetch the (x,y,z) geometry, and shading on the fragment processing units. During vertex processing, a border stitching process guarantees seamless geometry between neighboring nodes with different geometry resolutions.

We next present a look at the overall multi-grained hierarchy structure, followed by details of the three phases.

4 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 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. Notice that every node cannot access every resolution. 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

$$r = \min(\log_2(M) - l, \log_2(R)) \tag{1}$$

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, 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 or triangle threshold by varying the number of nodes used. This permits the balancing of GPU and CPU load by changing the number of nodes used.

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 [7]. 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.

5 Seamless Texture Atlas Construction

The seamless texture atlas was originally proposed by [21] 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. They show 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. Their 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 their 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 error threshold) is reached.

Following the second clustering pass, perform quadrangulation, parameterization, 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 parameterization 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.

6 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 $(2^n + 1)\text{x}(2^n + 1)$, 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.

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.

7 INTERACTIVE RENDERING

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

7.1 View-dependent Adaptation

Traditional 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, and place its children 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 [5, 19]. 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 merge followed by a split 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 appear on all four queues. We iterate over 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.

Then in the second phase, we propose a set of splits and merges of nodes, performed by pretending to do an error threshold adaptation with the current error of each node as a local error threshold. First the nodes are merged 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. Then nodes are split until no more splits can be performed.

We then proceed back to the refine/coarsen process. After each round of the refine/coarsen process, we compare the maximum error to what it was before the previous split/merge operations. If the new error is greater then the previous error, the split/merge operations are undone, and the adapt process terminates successfully. This undo-based termination is employed because selection of merges and splits is a heuristic choice based on triangle counts, and the true cost or benefit of the split/merge is not known until the refine/coarsen process adjusts the resulting node resolutions. We could improve the predictive power of the split/merge process by finding the true cost of splits and merges in terms of reducing error, but the would require many more cycles through the refine/coarsen queues (which would each be undone in much the same way we would be avoiding on the split/merge queues).

Given the final set of nodes and their associated geometric resolutions generated by the adaptation process, we also estimate 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.

7.2 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 possible because of the simple quad-tree structure, the ability to use geometry image textures to look up data from an ancestor nodes by transforming texture coordinates, and the ability to seamlessly render adjacent patches of different resolutions. 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

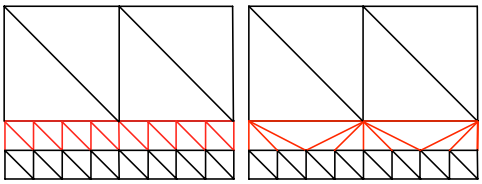

Figure 3: Border stitching is performed when nodes of different tessellations share a border. In this example the nodes are two LOD levels apart, forcing vertices in the more densely tessellated node to be collapsed in order to match the lower resolution node.

occupies less then one tenth of one percent of GPU RAM for the 22 billion sample Earth dataset).

7.3 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 an appropriately 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).

Since texture maps are used to render the 3D mesh, we can easily apply additional textures to the recreated model, such as color or normal maps, which greatly improve the visual quality of the model. These textures are not dependent on each other, and can therefore be of different resolutions. This allows for higher resolution color and normal data to be used where necessary, while still retaining low resolution geometry data, allowing the best-fit of data to be used for each area of the model.

7.4 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 together the border between adjacent nodes with different resolutions, we calculate in the vertex program the texel selected by the lower-resolution neighbor, and force the current vertex to select the same texel, thereby matching the higher-resolution border to the lower-resolution border and eliminating cracks, as shown in Figure 3. Traditionally this process would be performed on the CPU because that is where the resolutions of adjacent patches are determined. However, because the original vertex coordinates cached as geometry images on the GPU, it is convenient to perform this matching on the GPU as well.

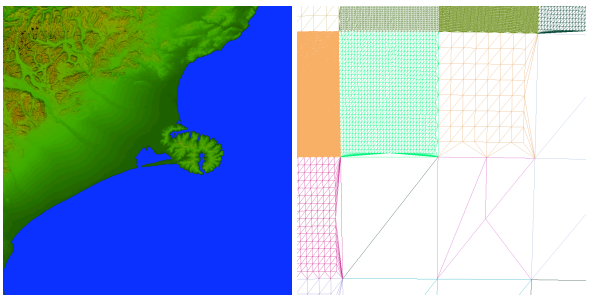

Figure 4: An example from the Earth dataset. Notice that the triangle budget mode rendering benefits from neighboring patches with levels differing by 3 or more.

We see a more complex situation in Figure 4. In this example from the Earth dataset we see a mountainous area tessellated much more densely then an adjacent area representing water or other flat feature. Adjacent nodes differ not only by multiple resolution levels, but also by multiple levels in the hierarchy. Thus, a node may have multiple smaller neighbor nodes along a single one of its border edges. To handle this general case, we pass to each vertex an array of all the resolutions of the neighboring nodes along all four node boundaries. This enables matching of all high-resolution boundaries to all low-resolution boundaries. This data is constant per patch, so it may be set in the constant registers during the rendering of a patch. The overhead of the border matching process is reasonable, currently requiring 18 ARB vertex program instructions to perform.

It is worth noting that we do not have a problem avoiding cracks at the corners, because our current implementation employs simple sub-sampling to reduce the resolution of the geometry images. This issue will require more careful treatment when a better downsampling filter is employed.

Also, this process does not change the overall geometry error of the model. Although the error is increased along the boundary of the higher-resolution node, the neighboring node has already accepted that amount of error along its identical boundary, so it is acceptable for both nodes to have that same error there.

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 small neighbors than it has boundary vertices for them to match.

8 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).

8.1 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 parameterization, 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 parameterization, and 5 minutes to resample the 138 resulting charts at 512x512 samples per chart (coordinates and normals).

In practice, we achieved a nearly 40x speedup by performing the parameterization 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.

8.2 Hierarchy Creation

The hierarchy creation stage of our method was performed on an AMD 2.4GHz dual-Opteron system running Linux using 4 simultaneous threads and using less than 500MB of memory. The models processed are shown in Table 1. 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.

Model	Samples	Trees	Size	Time
Earth (flat)	21.6B	70	17MB	310
Earth (spherical)	21.6B	70	17MB	1115
US	5.5B	15	3.5Mb	59
Puget Sound	268M	4	1MB	2.3
Thai statue	130M	140	32MB	2.4
Tablet	8M	30	6.8MB	.08

Table 1: Hierarchy creation information for the datasets. Samples are scalars for the height fields and 3D vectors for the general models. Output size refers to the hierarchy node data and not the actual sample data. Time is in minutes. All the quadtrees are build to contain 5 levels.

8.3 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.

Figure 5 shows the performance of the system while following a path over the 22 Billion Earth dataset. From the graphs we can see that the rendering performance is unaffected by the loading of data, and is capable of consistently rendering over 75 million triangles/second. The second graph shows that the use of fewer rendering nodes allows our method to maintain a low error threshold while spending only a fraction of frame time adapting the frame. Note that for the spherical earth model, we have opted to store the model

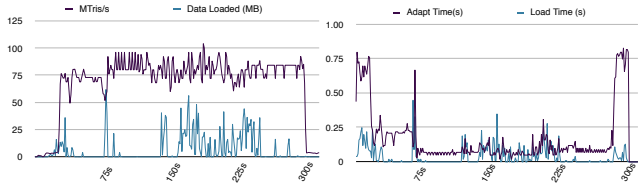

Figure 5: The graph on the left shows the throughput of the system while rendering a path along the 22B sample Earth dataset. The rendering rate stays over 75M triangles/s, even when large amounts of data are loaded. The graph on the right shows adapt and load times from the same pass, along with the screen-space error which has been capped at 4 pixels by the system.

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.

In Figure 6 we show the number of nodes versus the average adapt time over a path and the number of nodes versus the average screen-space error in pixels over the same path for the North America 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 paths were run with a triangle budget of 4M triangles.

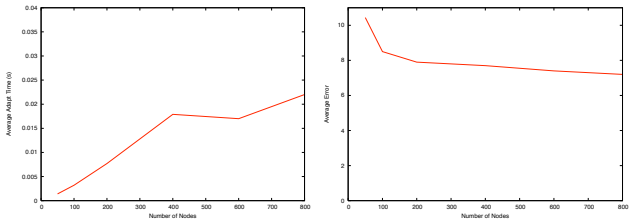

Figure 6: The plot on the left shows the advantage of using fewer nodes to render a scene. As more nodes are used the adapt time increases, forcing more time to be spent adapting the scene. The plot on the right demonstrates the benefit of using more nodes, as it allows the error to be more evenly distributed across the scene

The plots shown in Figure 6 demonstrate the need for our method. While we want to use more nodes to better distribute the error, we also want to use fewer nodes to minimize the computation time spent adapting the scene. Our method permits exactly this type of balancing, since nodes have a selection of LOD levels, allowing a fixed number of nodes to be used. For our method we could, therefore, pick the optimal number of nodes to render an optimal number of triangles, removing the CPU and GPU bottlenecks.

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 7 we throttle the 2.4GHz AMD Opteron CPU to three different settings, 2.4GHz, 1.8GHz and 1.0GHz and show that the ability to choose the number of nodes allows our method to perform well on various configurations.

We have also throttled the GPU to 470MHz, 200MHz and 100MHz as shown in Figure 7. This test shows that adjusting the triangle count of a node allows for a more optimal matching to the CPU and GPU performance. Because we can do this independently of the number of rendering nodes we can adapt a slow GPU while still using a large number of nodes, giving a better bound on the error in

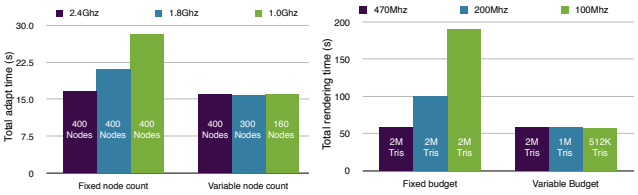

Figure 7: In the graphs on the left the number of nodes is maintained while the CPU is throttled. By using fewer nodes we can maintain a stable adapt time while still maintaining the 2M triangle budget. The graphs on the right show rendering performance of the throttled GPU. By allowing our method to adaptively select the number of triangles per node we benefit from the larger number of nodes while still staying within the triangle budget. All of the graphs show the total time to render a 1917 frame path using 400 nodes and a 2M triangle budget unless specified otherwise.

the scene.

9 Conclusions

The data structures, algorithms, and prototype system we have presented here explore a number of useful ideas and trends for high performance rendering on evolving graphics hardware. Our approach incorporates ideas from seamless geometry atlas parameterization, geometry image resampling, and quad-tree-based hierarchical rendering.

We have presented a new step forward in the trends of hierarchical rendering – a multi-grained level of detail system rendering from geometry image and height map data. This system is capable of replicating the previous LOD approaches such as discrete, view-dependent and coarse-grained LOD. These methods effectively become a subset of our method, giving our system far more flexibility. This permits the balancing of the CPU and GPU workloads, allowing for the adapt and rendering work to be managed independently, something not previously possible with other LOD systems.

The flexibility of our method also permits the balancing of both the CPU and GPU workload across various machines, allowing our system to use a common hierarchy on a variety of hardware with varying capabilities.

Our system uses the latest abilities of GPU's in order 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 allows our method to be extended in the future. We believe that the use of such geometry images is a trend that will increase as the texturing pathways through the graphics hardware become more complete, and that such methods provide an elegant solution to problems like boundary stitching on the GPU.

Finally, it seems that a number of additional features common to high-performance rendering systems could be incorporated with our approach in the future to produce a system which is both very flexible and has even better performance. These include data prefetching, compression, back-patch culling, occlusion culling.

References

[1] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps, March 2005.

[2] Louis Borgeat, Guy Godin, Francois Blais, Philippe Massicotte, and Christian Lahanier. Gold: interactive display of huge colored and textured models. *ACM Trans. Graph.*, 24(3):869–877, 2005.

[3] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Batched multi triangulation. In *IEEE Visualization*, page 27, 2005.

[4] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[5] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[6] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification. *Comput. Graph. Forum*, 19(3), 2000.

[7] Carl Erikson, Dinesh Manocha, and William V. Baxter III. Hlods for faster display of large static and dynamic environments. In *ACM Symposium on Interactive 3D Graphics*, pages 111–120, 2001.

[8] Leila De Floriani, Paola Magillo, and Enrico Puppo. Building and traversing a surface at variable resolution. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 103–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[9] Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federic Ponchio, and Roberto Scopigno. Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH*, 2004.

[10] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM Press.

[11] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, 1997.

[12] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[13] Lok M. Hwa, Mark A. Duchaineau, and Kenneth I. Joy. Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Trans. Vis. Comput. Graph*, 11(4):355–368, 2005.

[14] Junfeng Ji, Enhua Wu, Sheng Li, and Xuehui Liu. Dynamic lod on gpu. In *Computer Graphics International 2005*, pages 108–114, 2005.

[15] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Symposium on Interactive 3D Graphics*, pages 93–102, 2003.

[16] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM Press.

[17] Peter Lindstrom and V Pasicco. Visualization of large terrains made easy. In *Visualization*, pages 363–370, 2001.

[18] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[19] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH*, pages 199–208, 1997.

[20] Emil Praun and Hugues Hoppe. Spherical parametrization and remeshing. *j-TOG*, 22(3):340–349, July 2003.

[21] Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. Seamless texture atlases. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 65–74, New York, NY, USA, 2004. ACM Press.

[22] Pedro V. Sander, Zoë J. Wood, Steven J. Gortler, John Snyder, and Hugues Hoppe. Multi-chart geometry images. In *Symposium on Geometry Processing*, pages 146–155, 2003.

[23] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Visualization*, pages 327 – 334, 1996.