LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JRNL-233845

Tile-based Level of Detail for the Parallel Age

Krzysztof Niski, Jonathan D. Cohen

August 21, 2007

IEEE Transactions on Visualization and Computer Graphics

Tile-based Level of Detail for the Parallel Age

Krzysztof Niski and Jonathan D. Cohen

Abstract— Today's PCs incorporate multiple CPUs and GPUs and are easily arranged in clusters for high-performance, interactive graphics. We present an approach based on hierarchical, screen-space tiles to parallelizing rendering with level of detail. Adapt tiles, render tiles, and machine tiles are associated with CPUs, GPUs, and PCs, respectively, to efficiently parallelize the workload with good resource utilization. Adaptive tile sizes provide load balancing while our level of detail system allows total and independent management of the load on CPUs and GPUs. We demonstrate our approach on parallel configurations consisting of both single PCs and a cluster of PCs.

Index Terms—Level of detail, out-of-core, distributed, parallel, geometry image.

1 Introduction

Parallel computing is quickly becoming mainstream. Physical constraints such as heat dissipation and power consumption have driven CPU manufacturers to deliver multiple CPU "cores" on a chip rather than a faster individual core. Dual-core and quad-core chips are currently available, with some motherboards supporting multiple such chips. As an indication of the future of this trend, Intel has built an 80-core prototype chip that may power the PC of 2011 [17].

Parallelism is also available at other levels. Many PCs today support multiple GPUs; dual-GPU machines are not uncommon, and quad-GPU machines are available as well. At the higher end, products such as NVIDIA's Quadro Plex allow up to eight GPUs to be attached to a single PC. Furthermore, it is easier today than ever before to build a cost-effective compute cluster out of many such commodity PCs.

Unfortunately, many tasks are not trivially parallelizable. Thus applications will often require explicit awareness of this parallelism as well as new, parallel-friendly algorithms to fully exploit the capabilities of the machines in this parallel age.

In this paper, we explore the use of level of detail (*LOD*) for interactive rendering in this parallel setting. Traditionally in parallel rendering, one builds a big-enough parallel machine to handle a given task, and *load balancing* techniques are used to maximize resource utilization and performance. By incorporating level of detail into the setting of parallel rendering, we move beyond the standard problem of load balancing to address the broader problem of *load management* that is necessary for parallel, interactive applications.

The system presented here takes advantage of both application-specific knowledge (e.g., bounding boxes of geometric data) and parallel architecture-specific knowledge (e.g., number and distribution of processors and their associated memories) to fully utilize the available system resources. This results in a more general and scalable approach than those currently provided, for example, within the graphics drivers themselves.

We model the mapping between parallel computing resources and LOD-based rendering tasks using a hierarchical arrangement of rectangular, screen-space *tiles*. Three classes of such tiles are used to model the exposed functional units of our parallel architectures: *machine tiles* (mapped to individual PCs), *render tiles* (mapped to individual GPUs), and *adapt tiles* (mapped to individual CPUs). These are nested within one another – adapt tiles within render tiles within machine tiles – and may be hierarchically arranged in a *kD*-tree structure within each class.

Our system has the following new and useful capabilities:

- **Parallel adapt:** Each adapt tile independently traverses our multi-resolution hierarchy, performing its own culling and LOD selection. This adaptation process is concluded by a fast, single-pass *cut unification* process to make a tile's geometry consistent with that of its neighbors. Adapt tiles are processed on separate CPUs and may share hierarchy data in configurations where they access the same memory pool.
- **Multi-GPU rendering:** Each render tile is assigned its own GPU and may be processed independently. These GPUs need not be a matched set, but may be heterogeneous in performance. Rendered pixels are transferred over the PCI-Express bus to a single frame buffer. Each GPU maintains an independent data cache in its video memory according to the recent needs of its associated render tile.
- **Distributed rendering:** Each machine tile is assigned its own PC, enabling the system to render to a tiled display wall. Because of the relatively coarse parallelization scheme and the single-pass cut merging, our system requires only *loose synchronization*, with little network overhead.
- **Parallel load management:** Our system dynamically resizes the adapt and render tiles in conjunction with level of detail control to enable total load management. We present algorithms not only for balancing the load across multiple CPUs and GPUs, but also for independently controlling the magnitude of the CPU and GPU loads.

Our system is applicable to a wide variety of parallel machine configurations. We demonstrate the adaptivity of our system for interactive rendering on several such configurations using scanned models with tens of millions of samples as well as the 22-billion-sample USGS Earth data set with normal and color texture maps, totaling over 180GB of data.

2 Related Work

2.1 Tile-based Parallel Rendering

Tile-based processing has a long history in parallel rendering, and has been used to categorize parallel rendering architectures [13]. A parallel rendering architecture is termed sort first [15], sort middle [7], or sort last [14], depending on where in the rendering pipeline primitives are mapped to screen tiles with respect to the transformation and rasterization stages.

Much of the recent work in the area of parallel rendering has focused on using networked clusters of commodity PCs. Such systems can generally drive a tiled display wall using a commodity local network as well. Chromium [9] is a general stream processing system which hijacks the calls to the OpenGL driver and can implement a variety of parallel architecture semantics on top of a PC cluster. Like our system, it can perform tile sorting to route primitives to appropriate machines. Although it acts as OpenGL, it also allows some useful hints such as bounding boxes for primitive blocks. It is possible that the multi-GPU portion of our research could leverage the widely-deployed Chromium system, though we have chosen not to pursue that route to implementation.

Mueller's sort-first emulator [15] uses *kD*-trees as in our system to provide load balance to a sort first architecture. Samanta's PC cluster-based system [20] targets tiled displays as in our system, but allows

- *Krzysztof Niski performed this work at Johns Hopkins University, and is now at NVIDIA Corporation, E-mail: niski@cs.jhu.edu.*
- *Jonathan D. Cohen is at Lawrence Livermore National Laboratory, E-mail: jcohen@llnl.gov*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 27 October 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

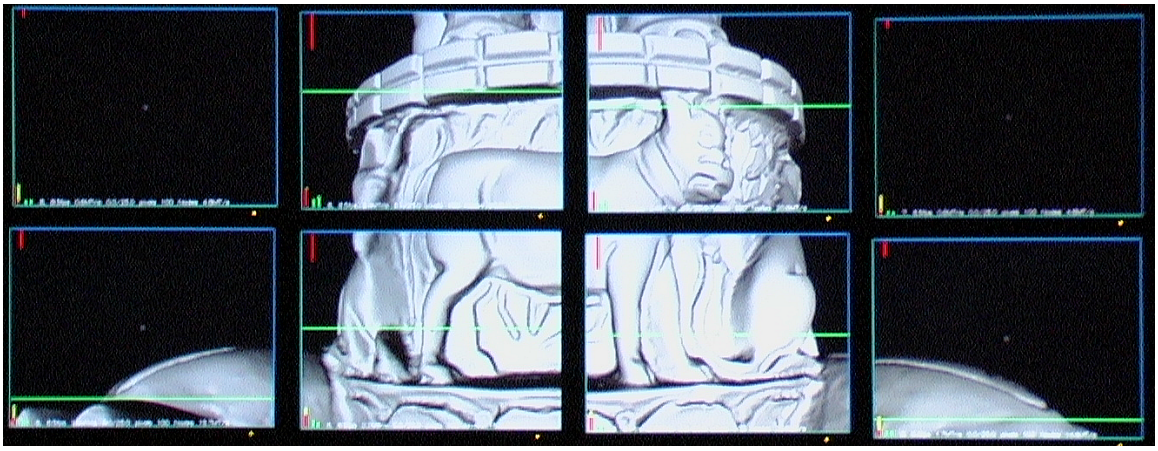

Fig. 1. Photograph of a tiled LOD system running on a PC cluster using eight machine tiles, each with one render tile and two adapt tiles (the images are seamless on a projected display, but LCD frames appear around each image here).

load balance and pixel transport among machines. In general, their algorithm attempts to render most pixels on the machine local to the display, but allows remote rendering when it should benefit performance. Remote regions to be rendered in this system need not be assigned to neighboring tiles of the local region.

NVIDIA's SLI [22] and ATI's Crossfire [2] provide in-driver support for rendering on multi-GPU PCs. In SLI, pixel traffic is moved among the cards using a dedicated SLI connection pathway, bypassing the PCI-Express bus and the CPU entirely. The Alternate Frame Rendering (AFR) mode renders successive frames on different GPUs, moving the resulting frame buffers as necessary at scanout time. The Split Frame Rendering (SFR) mode uses dynamic tiles to partition the workload according to performance feedback. It apparently replicates the vertex processing on each GPU, so the speedup in this case is just for fragment processing. This approach is very general from the driver's perspective, because it does not need to worry about arbitrary transformations that could occur in a user-defined vertex program, for example, and it does not burden the CPU with tile sorting. We compare our approach to this in-driver approach in Section 9. Crossfire is similar to SLI, but it allows GPU combinations that are not perfectly matched, and also supports a Super Tiling mode which generates a fixed grid of tiles which are distributed among the GPUs (where each GPU operates on multiple tiles). Like SLI's screen partitioning mode, it performs all transformation on both GPUs.

OpenGL Multipipe SDK [4] is a highly configurable toolkit supporting both screen-space and temporal workload distributions as well as database distribution with image composition. It takes more of an application aware approach than SLI, Crossfire, or Chromium, creating more opportunities for optimization. It runs on SGI's multipipe rendering hardware platform.

2.2 Parallel Level of Detail

Less work has been done on integrating level of detail with parallel rendering. Level of detail has sometimes been listed as future work in papers about parallel rendering [5], and parallelization has been listed as future work in papers about level of detail [11].

A recent system for rendering large terrains to a tiled display employed the ROAM-2 [10] view-dependent simplification on each PC of a cluster [6]. In this system, a triangle budget was used to control performance on each PC an error threshold was used for all primitives crossing the tile boundaries, ensuring a crack-free mesh without requiring any network synchronizations during the frame.[1] There is no opportunity for load balancing in this system beyond the LOD control.

One of the few level of detail load-balanced parallel rendering systems was presented by Samanta et al. [19]. This hybrid sort-first/sort-last system decomposes a large mesh using a *kD*-tree, then maps this tree structure into a scene graph, with reduced resolution mesh pieces in the interior nodes. Geometry chunks are pre-distributed across computers with k-way replication. A per-frame adapt process traverses the scene graph, selecting processor assignments to balance the load on the processors while refining the LOD to an appropriate level. This is the first system to allow performance-driven, load-balanced parallel rendering using level of detail. However, with respect to the level of detail algorithm employed, it is somewhat simplistic, allowing arbitrary cracks between the geometry chunks, which are essentially treated as discrete levels of detail. Another major difference from our system is that we opt for an out-of-core approach to the data distribution problem [5], using a shared network disk for all distributed computers with cache in local RAM.

Parallel rendering systems utilizing level of detail have also been used to visualize volumetric data [3]. In this method the volumetric data is distributed from a single server node at run time and visualized on a display wall. Each worker node is used to render a section of the final framebuffer on its attached display. This sort-first method is similar to the one used in this paper, except for its use of level of detail. In [3] level of detail is used only to reduce the network bandwidth between the host and render nodes, without any effective per-tile load management.

3 LOD System

The method presented in this paper requires a level of detail (LOD) system to manage the complexity of interactively rendering large mesh data. While many such LOD systems exist, we employ the hierarchical, seamless texture atlas (HSTA) method presented in [16] due to its unique ability to adjust CPU and GPU loads relatively independently.

The geometry and attribute data in this approach is resampled into a form of multichart geometry image [21] called a seamless texture atlas [18] (algorithms are presented in [16] for converting large meshes to this format). Each square chart image is stored as an image pyramid. Logically overlaid on each chart's image pyramid is a quadtree structure used for LOD manipulation. Each *node* of the quadtree covers a particular region of the chart's 2D domain, and can access the geometry and attribute data for that region at any desired resolution (with some restrictions at the extremes).

The process of adapting the mesh LOD using HSTA produces a *cut* of nodes which forms a partition of the atlas domain. Each node in the cut comes from some particular level in the quadtree hierarchy (which dictates its size in the 2D domain) and has a particular resolution selected for the geometry (resolution of other attributes, often used for shading in the fragment unit, may be selected independent of the geometry resolution).

[1] These unpublished details about the parallel LOD setup were provided by personal communication with Mark Duchaineau.

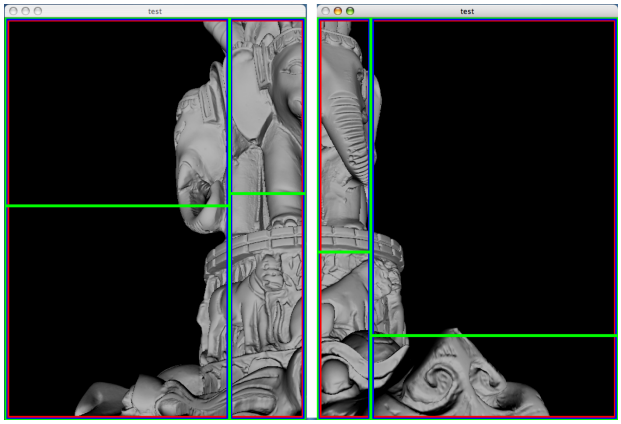

Fig. 2. The *kD*-tree structure used for load balancing is apparent in this two-PC system with two render tiles per machine tile and two adapt tiles per render tile.

The LOD system adapts the rendering according to either a desired error threshold, minimizing the number of triangles, or to a maximum triangle count, minimizing the error. In either case, unlike other systems, HSTA solves these problems with the additional constraint that the cut contain no more than a specified number of nodes. The number of nodes on the cut correlates with the CPU work required to adjust the cut every frame, and the number of triangles produced correlates with the GPU work required to render the triangles. Using more nodes increases the CPU load, but decreases the triangle count for a given error, and improves view frustum culling (which can be important for a tile-based parallel system).

Unlike most quadtree-based systems, the adapt process places no constraints on the cut due to neighbor node relationships. Neighboring nodes may have widely different quadtree depths and/or image pyramid resolutions. Discontinuities between nodes are handled by resolution matching along the seams. This resolution matching can be performed in the vertex unit using vertex texture lookups, or by dynamically creating and issuing seam-stitching strips on the CPU. Although the former is more elegant, it can suffer from some performance problems due to the vertex texturing. Thus we use the latter method for the results reported in this paper.

HSTA keeps the quadtree data structure resident in main memory, with the image pyramid data stored on disk and cached locally in main memory and GPU memory. The caching mechanisms used in this system utilize out-of-core management to asynchronously load appropriate resolution data from the hard-drive into system and video memory as needed. The caching and loading algorithms are described in full detail in [16].

The LOD method used in this system was chosen due to its quadtree structure that greatly simplifies screen tile unification. The design of the parallel system is not, however, limited to the HSTA method; other LOD systems such as TetraPuzzles or GoLD could be used in its place. The main change required of these methods is the ability to independently adapt screen tiles and unify the separate cuts through the hierarchy.

4 Tiled Approach

Tiled-based approaches to parallelization are commonly used in both sort-first and sort-middle rendering architectures. In our sort-first approach, tiles effectively distribute both the computation and the memory usage.

In our setting, the important questions are how to assign the tiles, how to manage the load, and how to maintain consistent LOD across tile boundaries.

4.1 Arrangement

Three classes of nested, screen-space tiles in our system – machine tiles, render tiles, and adapt tiles – correspond to the PCs, GPUs, and CPUs of a particular parallel configuration (see Figure 2. Each machine tile is responsible for delivering the pixels from all the render tiles in that machine to the final display buffer. For simplicity and efficiency, we currently restrict our distributed systems to those with distributed frame buffers (typically used to support a tiled display wall). This avoids the complexity of trying to efficiently transport pixel data over a network (which may involve fast compression/decompression, etc.). Thus the arrangement of machine tiles is fixed to match the tiled display wall.

Within each machine tile is nested one or more render tiles, each generally associated with a unique GPU on that machine. The arrangement of render nodes follows an alternating-dimension *kD*-tree layout (i.e., the tile is recursively split by lines in the *x* and then *y* dimensions until enough rectangular regions are formed).

Similarly, adapt tiles associated with the CPUs on that machine are nested within that machine's render tiles. So a machine with two GPUs and four CPUs would have two adapt tiles in each render tile. In the case of a machine with more GPUs than CPUs, multiple adapt tiles are assigned to a single CPU. As above, multiple adapt tiles within a render tile are arranged according to an alternating-dimension *kD*-tree.

These tiling arrangements are sufficiently general to describe a wide variety of parallel machines: multi-CPU/single-GPU computers, single-CPU/multi-GPU computers, multi-CPU/multi-GPU computers, and heterogeneous clusters of such computers.

4.2 Load Management

Load management on a parallel system requires controlling both the balance and the magnitude of the load. Our system employs a reactive approach to overall load management, using performance measurements of recent frames to judge how to adjust the load on each hardware unit in the frames to come. Although one could possibly design a more predictive approach [8] to improve response time to load changes, the reactive approach has the benefit of relying primarily on current real performance times, making it simple, practical, and general enough to operate on a variety of systems without complex performance modeling.

One approach to load balancing in a tile-based parallel setting is to create many more tiles than processors, then dynamically assign tiles to processors as the load changes [7]. However, this would significantly increase the overhead for tiles due to the increased number of nodes that would cross tile boundaries and require unification.

We have opted instead to match tiles to processors and dynamically resize the tiles in screen-space to control the load balance. As in some other sort-first systems [15], we adjust the tile sizes by modifying the position of each splitting line in the *kD*-tree (see Figure 2). For a given splitting line the new position is computed as follows:

$$x_{new} = \frac{x_{old} * time_{right}}{time_{right} * x_{old} + time_{left} * (1 - x_{old})}, \quad x_{old}, x_{new} \in (0,1) \qquad (1)$$

We take a similar approach to controlling the loads on the CPUs and GPUs by dynamically adjusting parameters of the underlying LOD system. To adjust the load on an adapt tile, we adjust the number of nodes it is assigned as follows:

$$nodes_{new} = \sqrt{\frac{time_{target}}{time_{old}}} * nodes_{old}, \qquad (2)$$

where $time_{target}$ is the desired adapt time and $time_{old}$ is the previously measured adapt time for the tile in question. The square root promotes hysteresis and temporal coherence.

We control the load on a render tiles by adjusting it's error threshold used for LOD selection as follows:

$$error_{new} = \sqrt{\frac{time_{old}}{time_{target}}} * error_{old}, \qquad (3)$$

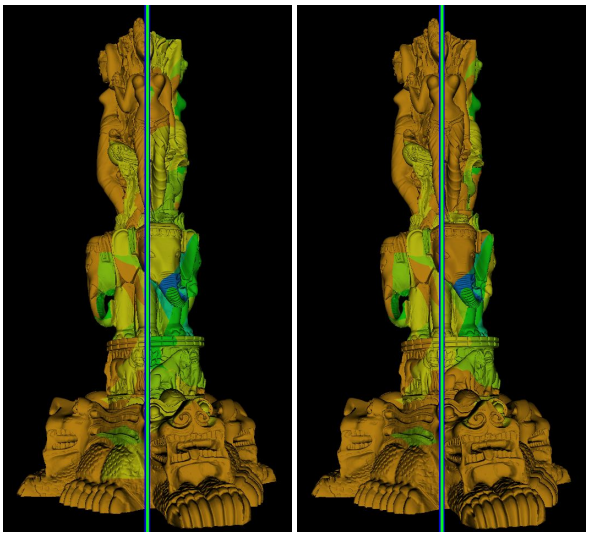

Fig. 3. These images show two render tiles before and after the cut unification process. The image on the left shows multiple pieces of geometry that do not agree on resolution between the two render tiles. The image on the right shows the seamless model after the unification process.

where $time_{target}$ and $time_{old}$ now refer to the render time for the tile being adjusted.

To achieve total load management, we alternate adjusting the load balance and the load magnitude in successive frames. In practice, we average all timing measurements used for $time_{old}$ over a temporal window of several frames and also wait several frames between adjustments to promote coherence and stability in the management system.

Our system can run in *quality mode* and *performance mode*. In quality mode, we perform load balancing among adapt tiles and among render tiles, but the error threshold used by the render nodes is a user-controlled parameter. The system can still adjust the number of nodes used in the adapt tiles to improve performance. To achieve the best performance for a given error threshold, the adapt time should be roughly equal to the render time, since these stages are pipelined in our system (we render frame i while adapting frame $i+1$). Increasing the number of nodes used up to this limit actually improves performance by reducing the number of triangles issued to the GPUs.

In performance mode, we not only load balance among both the adapt and render tiles, but control the load on both the adapt and render tiles to target an overall desired frame rate. By default, the system again tries to match adapt times to render times to maximize performance. However, it is also possible in our system to specify desired render times and adapt times separately. This is useful if an application wishes to reserve a certain percentage of the CPU time for other tasks.

4.3 Cut Unification

During the adapt process, each adapt tile creates its own cut for its subset of the viewing frustum. Each cut has a set of hierarchy nodes with associated rendering resolutions. Within a tile, the nodes in the cut have the property that no node is an ancestor of any other node.

However, when we consider the set of all nodes from the cuts of all adapt tiles, we may have inconsistencies. The same node may appear on two or more cuts at different resolutions, and there may be nodes with ancestor-descendant relationships in this set. Due to the ability of the underlying LOD system to adjust number of nodes and number of triangles independently, these inconsistencies could occur even if all adapt tiles were set to adapt to the same error threshold.

Our solution is to unify the cuts of neighboring tiles (see Figure 3). Each adapt tile adjusts the node resolutions such that each set of overlapping nodes (those with ancestor-descendant relationships) uses the same resolution. The adjustment operator depends on the management mode. In quality mode, we set all resolutions in the overlap set to the maximum of all members; in performance mode, we use the minimum. Note that the stitching of any resulting resolution discontinuities along the node boundaries is already handled by the underlying LOD system. The process works as follows.

During the adapt process, the adapt tile tracks which nodes may cross the tile boundaries (this information is produced naturally by the view-frustum culling process). When the adapt is complete, this set of boundary nodes is shared with all other adapt tiles (this is more robust than just sharing with neighbors, in case a node is large enough to completely cross a neighbor tile).

After sending its set of boundary nodes, the adapt tile begins construction of an *overlap tree*, which will be used to compute the sets of overlapping boundary nodes. The overlap tree follows the structure of the quadtree LOD hierarchy. Each *o-node* in the overlap tree corresponds to either a boundary node or one of its ancestors, and contains a list all currently known boundary nodes that are its descendants (e.g., the root o-node will have a list of all currently known boundary nodes). Each leaf o-node represents an actual boundary node, and the node list stored at the leaf contains a minimal set of overlapping nodes to be unified.

To insert a boundary node into the overlap tree, we do a top-down search to find whether or not its associated o-node is already in the tree. If the o-node is not yet present, we create that o-node and all the necessary o-nodes on the path down to the one we wish to insert. If the o-node is already present, but is not a leaf, we prune out all the descendant o-nodes, making that node a leaf. If the o-node is already a leaf, then we do not create or remove any o-nodes. In all cases, we add the boundary node to the node lists of every o-node on the path from the root down to a leaf o-node.

As sets of boundary nodes are received from other tiles, we add their boundary nodes to the overlap tree. When all boundary nodes have been received from all tiles, we unify the resolution of all the nodes listed in each leaf o-node of the overlap tree. If the total number of nodes in the LOD hierarchy is N and the total number of boundary nodes for the current frame is B, a conservative asymptotic time for cut unification is $O(BlogN)$.

5 Parallel Adapt

Each adapt tile in our system is associated with its own execution thread and operates in parallel. After receiving its error threshold and maximum number of nodes, it uses the error budget adaptation method of the underlying LOD system to traverse the mesh hierarchy, producing a view dependent cut of nodes. Although the adapt tiles each produce their own cut, tiles on the same PC can share the same copy of the LOD quadtree hierarchy in RAM as well as recently used geometry cached from disk. Thus there is not a significant memory burden for using multiple adapt tiles.

During the hierarchy traversal, each adapt tile produces its list of boundary nodes as part of the regular view-frustum culling process. It is worth noting that the LOD system we use is particularly good at view frustum culling compared to typical view-dependent hierarchies. It can use smaller nodes at the view frustum to achieve better culling while maintaining the desired resolution. This is especially beneficial in a tiled setting because we have several smaller view frusta rather than a single large one.

The adapt threads are synchronized once after computing the view-dependent LOD to exchange the border node lists, then they each apply the cut merging process to adjust their node resolutions for consistency. At that point, the cut data are handed off to the render tiles for rendering.

6 Multi-GPU Rendering

With the advent of multi-GPU computers, we can take a similar approach to parallelizing the rendering itself. Each render tile operates its own execution thread on the CPU and is attached to its own GPU. The *primary* render tile has an OpenGL context attached to a GPU on the real display with a real window. Each *secondary* render tile creates

Fig. 4. Combining framebuffers. The raw read/write timings for a variety of resolutions is shown using two GeForce 7300 GPUs. In the full system the readback time is pipelined through tile balancing such that the readback from secondary GPUs completes as the primary GPU finishes rendering.

its own OpenGL context tied to a different GPU and renders into an OpenGL Frame Buffer Object.

After all adapt tiles have produced their cuts for rendering, the render tiles begin issuing the appropriate drawing commands to their respective GPUs (meanwhile, the adapt tiles begin processing the next frame). When each secondary render tile completes its rendering, it reads back the rendered pixels from the frame buffer object on the GPU. The readback time is taken into account when balancing the tiles such that when the primary GPU finishes rendering, the framebuffers from the secondary GPUs are already in system memory. When the primary render tile completes its rendering, it writes the data from each of the secondary tiles to the window-attached frame buffer. When all data has been written to the frame buffers, the primary render tile swaps the front and back buffers and prepares to begin the next frame.

The time required to read and write the framebuffer fragments is shown in Figure 4. By load balancing the tiles, our system is only limited by the much faster framebuffer write operation.

Unlike the CPUs on the same PC, each GPU has its own local memory (we are assuming GPUs on separate cards connected by PCI-Express, as opposed to a GPU residing directly on the motherboard). Our system manages each GPU's memory independently, maintaining a cache of recently used geometry and texture data for each. This makes better use of the memory than mirroring them, and allows the system to handle GPUs which are heterogeneous in both processing power and memory size.

7 Distributed Rendering

Our system assigns a machine tile to each PC of a cluster to operate in a distributed rendering setting, enabling it to drive high-resolution display walls. In fact, we do not currently support transporting pixel data between PCs, so we assume that the PCs drive a display with one or more tiles of the physical display attached to each PC. This simplifies the system implementation, but does have some ramifications for the load management system. Because there is no way to adjust the distribution of work among the PCs, the LOD system becomes the only way to reduce the workload on highly burdened machines. Our algorithm could support pixel data transport in principle, but the transport time would naturally be a more dominant component of the rendering time than it is for merely moving data over the local PCI-Express bus.

In both the single-machine and distributed settings, our parallel rendering pipeline requires only two synchronization points: at the start/end of the frame (to synchronize the swapping of the frame buffers and transmit camera and other user parameters) and at the completion of the LOD adaptation (to communicate border node information). Of course these synchronization points now incur greater latency because they occur over the network rather than among local threads. We perform all inter-PC communication over the network using the Spread Toolkit [1], which supports efficient group communication using multicast.

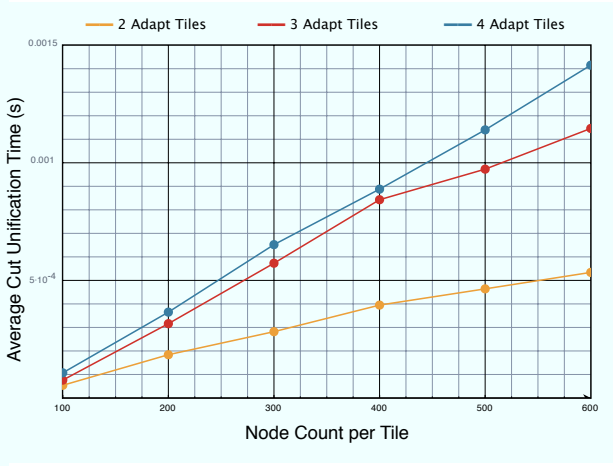

Fig. 5. Cut unification timings. Our system can perform the cut unification in very limited time. The cut unification process is pipelined with the renderer, removing its overhead from the frame time.

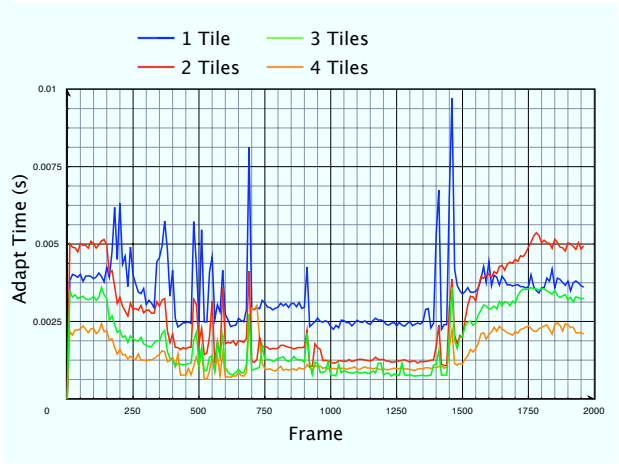

Fig. 6. Parallel adapt. By using a number adapt tiles our method is capable of distributing the adapt work, completing it in a shorter period of time. Results obtained from a path over the Earth data set using a 8M triangle budget and 1000 node budget.

Each PC keeps a copy of the LOD hierarchy structure in its local RAM and has access to the full geometry and texture hierarchy on either shared or local disk.

8 Results

The parallel and distributed modes presented in this system rely on the cut unification method to create a seam-free final image. The timings from the cut unification algorithm are shown in Figure 5 using a path over the 36M sample Thai statue model. As shown in the graph the cut unification time is low, and combined with pipelining, is hidden in the rendering time.

To test the parallel adapt performance of our method we have compared the maximum frame adapt time for our method using one, two and four adapt tiles, as shown in Figure 6.

As shown in Table 1 the use of multiple adapt tiles in our method generally outperforms the single adapt tile version due to the use of multiple CPUs. The average adapt time decreases as more CPUs are used to adapt the hierarchy. Due to a different distribution of geometry amongst the adapt tiles it is possible for the maximum performance increase to become super-linear. The average performance increase, however, indicates the true benefit of the parallel adapt algorithm.

The next test shows the benefit of the dynamic tiling utilized in

Tiles	Min	Max	Average	Std. Dev
1	100%	100%	100%	0
2	64%	312%	150%	54%
3	97%	481%	218%	80%
4	97%	670%	255%	81%

Table 1. Parallel adapt performance: The performance of the parallel adapt is calculated as the percentage ratio of single-tile adapt to multi-tile adapt time. Overall the use of multiple adapt tiles greatly improves the adapt performance of the system, and helps to utilize more of the available resources. The results were obtained from a path over the Earth Data set with each tile using a equal part of a 8M triangle budget and 1000 node budget.

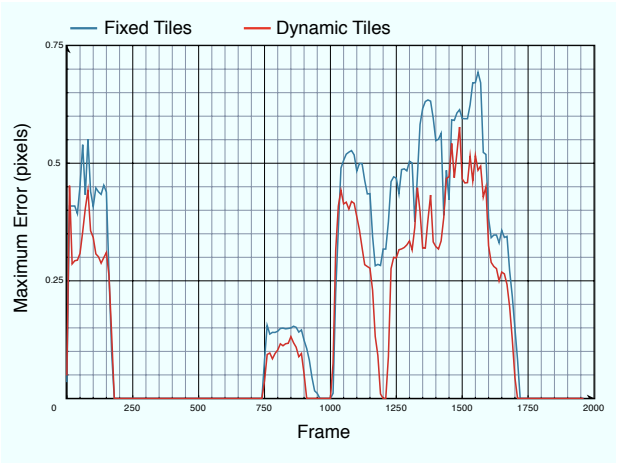

Fig. 7. A uniform subdivision of the view frustum can create unbalanced tiles. By dynamically adjusting the tile sizes our method ensures that the workload is uniformly distributed over a path of the Thai statue model. Each of the two adapt tiles used a 2M triangle budget and a 450 node budget.

our system (see Figure 7). The maximum error of a path along the Thai Statue model with dynamic tiling disabled is compared to the same path when dynamic tiling is turned on. As expected, the use of dynamic tiling reduces the maximum error by shifting the borders of the adapt tiles such that neither of the tiles is empty and the geometric error is approximately equal in both tiles.

The rendering throughput of our system was tested using an AMD Opteron system with two GPUs, a NVIDIA Quadro 4500 and a GeForce 7300. These two cards have a large imbalance of performance capabilities, with the Quadro being able to render approximately twice as much geometry as the GeForce 7300. To test the multi-GPU rendering abilities we played a path along the Thai statue with and without dynamic tiling enabled, with the results shown in Figure 8. As demonstrated in this test our method does not limit the multi-GPU capabilities to homogeneous hardware, something not possible in the

GPUs	Min	Max	Average	Std. Dev
1	100%	100%	100%	0
2 Static	70%	128%	132%	29%
2 Dynamic	128%	192%	164%	18%

Table 2. Multi-GPU rendering. This table shows the advantage of using multiple GPUs to render the final frame. By distributing the rendering workload our system can increase the rendering throughput. Note that the two GPUs are not equal; the second GPU is a lower performance model. The results are computed using a percentage ratio of the single-GPU results to the multi-GPU results.

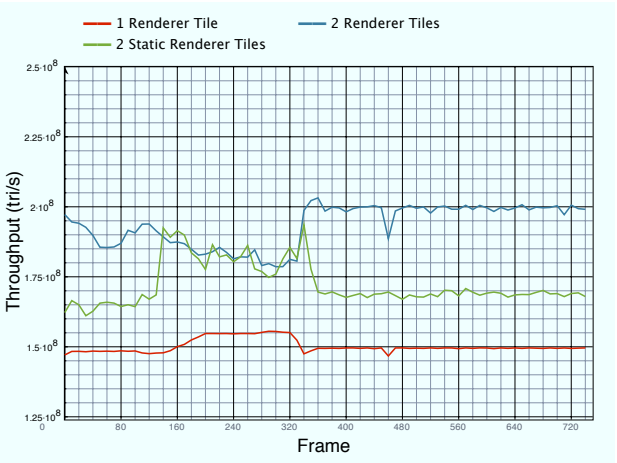

Fig. 8. Multi-GPU rendering. Our method is capable of distributing the rendering workload amongst multiple GPUs in a single computer. Through this process our method can achieve a considerable boost in rendering performance (the vertical axis here is triangles/sec). This acceleration is further enhanced through workload balancing which adaptively distributes the rendering workload. The path over the Thai Statue model is rendered using a NVIDIA Quadro 4500 and GeForce 7300GT. The Quadro was used for the single-GPU results.

driver-provided SLI methods. As shown in Table 2, the benefit of using multiple GPUs to perform the rendering is considerable, especially given the performance disparity of the two GPUs.

In addition to requiring homogeneous hardware, previous SLI methods forced all of the data in GPU memory to be mirrored across all of the GPUs. As shown in Figure 9 our method allows each GPU to maintain a separate data cache, duplicating some of the textures while storing additional textures not present on other GPUs. In general the majority of the duplicate textures are stored in cache, with few duplicate textures in use. As a result the unused duplicate textures can be swapped out on a per-GPU basis, allowing each GPU to maintain an independent data cache. This allows more of the memory to be used independently, increasing the total GPU memory usable for texture storage.

The workload balancing methods used in the parallel adapt and multi-GPU rendering sections of the results dealt with adjusting the size of the corresponding tile. By adjusting the error threshold and the node budget the performance mode can further optimize the performance in order to achieve a user-specified frame time. As shown in Figure 10 our system can balance the two variables to adjust the performance, while also adjusting the adapt and GPU tiles. The feedback-based algorithm allows the performance mode to compensate for a variety of factors, including the use of too many nodes to render the geometry when using multiple GPUs. By minimizing the error in the scene the dual-GPU configuration is generally capable of reducing the error threshold when compared to the single GPU renderer. It is, however, possible for the dual-GPU error to occasionally exceed the error of the single-GPU renderer due to the feedback-based balancing algorithm which may lag behind sudden model movement.

The results from the distributed system are shown in Figures 11, 12 and 13. All of the results were collected using a homogeneous cluster of dual-CPU Pentium 4 Xeon computers, each with a GeForce 6800GT, 2GB of RAM using a distributed Lustre filesystem.

The first set of results from the distributed renderer are the rendering and network overhead timings shown in Figure 11. The time spent synchronizing the renderers is a relatively small percentage of the overall frame time. The network synchronization time could be further reduced by using local storage instead of the shared network drives, which compete with the system for network bandwidth.

The use of a rendering cluster enables our system to render more geometry per frame to a higher-resolution screen. As shown in Figure 12

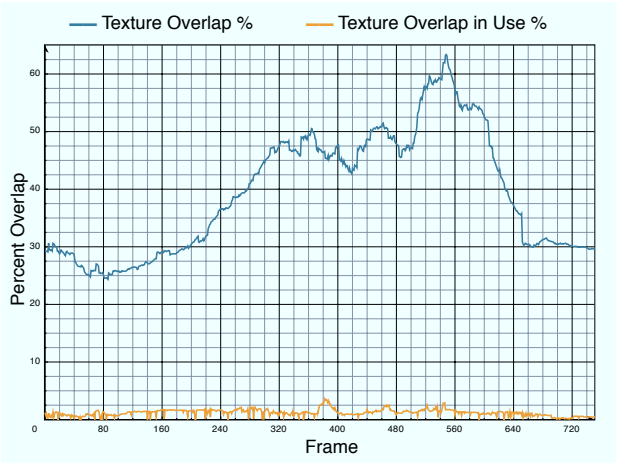

Fig. 9. Texture overlap. The percentage of textures mirrored on both GPUs (texture overlap) and the percentage of these textures in use (texture overlap in use) is shown in this graph. The mirrored textures represent cached data independent per-GPU, with each GPU able to remove its copy of the data independently. The in-use overlapping textures form a small subset of the total overlapping textures, allowing each GPU to optimize its texture cache.

the throughput of our system increases up to five times compared to a single renderer. The resulting throughput increase allows our system to render the Earth data set at sub-pixel error on a 5120x4096 display. The resulting rendering is shown in Figure 13 using a 4x2 rear-projection display wall. Figure 1 shows a section of the Thai Statue model rendered using the display cluster on a 4x2 LCD display wall.

9 Discussion

Our system provides a unique solution to managing load for LOD rendering on parallel PC platforms. We adjust the number of LOD nodes to control CPU load and the number of triangles (via the error threshold) to control GPU load. We also distribute work among CPUs and among GPUs by adjusting tile sizes dynamically.

It is worth comparing our approach to other possible approaches. For example, one could run an existing LOD system on a multi-GPU NVIDIA platform using the SLI mode of the driver to transparently run on multiple GPUs (though this would not solve the multi-CPU problem). The in-driver approach to multi-GPU rendering has the advantage of simplicity for the application developer. However, both the Split Frame Rendering (SFR) and Alternate Frame Rendering (AFR) modes require mirroring the memory across all GPUs. This can limit the scalability of the approach in practice. SFR mode is not really applicable to big model rendering, because the vertex processing is replicated on all GPUs, so it is only useful for speeding up fragment processing. AFR mode introduces a frame of latency for every GPU, which also limits scalability somewhat for interactive rendering. However, AFR mode has the benefit that it can potentially achieve excellent combined GPU performance. NVIDIA reports up to 1.9x performance over a single GPU by using 2 GPUs, which is much more than we can hope for moving data over current-generation PCI-Express (note that in tests of the SLI AFR mode using a simple VBO-based geometry benchmark, we have so far only achieved about 1.3x speedup). However, PCI-Express performance is expected to double with PCI-Express 2.0, and this can further improve our results.

It is also worth comparing our performance mode with systems that use asynchronous adaptation, such as VDS [12]. Asynchronous adaptation is an interruptible process that allows the rendering pipeline stage to begin at any time with whatever is on the current cut. This is useful, because it divorces the frame rate from the adaptation process entirely. However, the rendering quality is lower when the adaptation process falls behind and has to catch up. If the camera speed is too fast for the chosen granularity of the simplification operations, the adaptation may in fact never catch up. In our performance mode, the system will use fewer nodes in the adapt tiles if there is not enough coherence. This reduces the CPU load while keeping the quality relatively high and correctly adapted to the current view point.

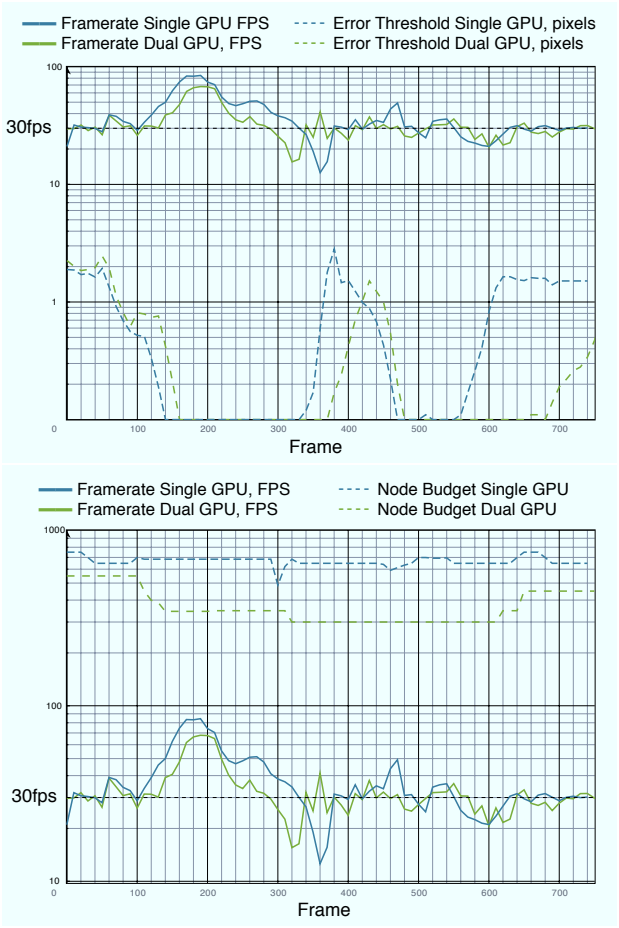

Fig. 10. Performance mode. The performance mode adjusts the node budget and error threshold along with adapt and GPU tile sizes to achieve a user-specified frame time while balancing CPU and GPU workloads. The data is recorded using a path over the Thai Statue data set with two renderer tiles and two adapt tiles.

10 Conclusion

In this paper we have described a new method for parallelized and distributed rendering of huge data sets. Our tile-based systems greatly improves utilization of hardware resources present in the system, allowing to utilize both multiple CPU cores and multiple GPUs simultaneously.

Our method presents a new means of utilizing multiple CPUs to perform much of the work in parallel, something not possible in the majority of previous systems. This ability enables our system to adapt the model more quickly utilizing independent adapt tiles, providing a scalable means of subdividing the workload.

Our method is also capable of operating in a distributed setting, a capability lacking from the majority of LOD systems. The combination of the tile based layout and minimal synchronization allows our method to utilize all of the available resources without incurring a high network overhead.

Acknowledgements

We would like to thank the USGS and Stanford University Graphics Laboratory for the models used in our experiments. This research was sponsored in part by NSF Medium ITR IIS-0205586, a DOE Early

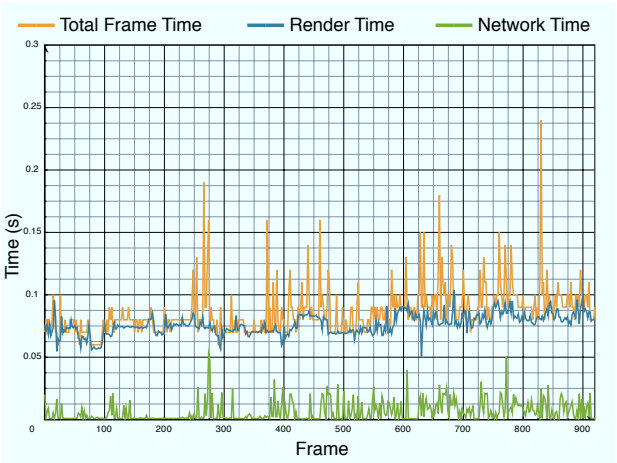

Fig. 11. Distributed timings. The frame time distribution between rendering, network overhead and other system functions such as GPU texture loads. The average frame rate over this path of the USGS Earth data set is 10fps when running on a cluster of eight computers, each with a NVIDIA GeForce 6800GT video card and a 2M triangle budget.

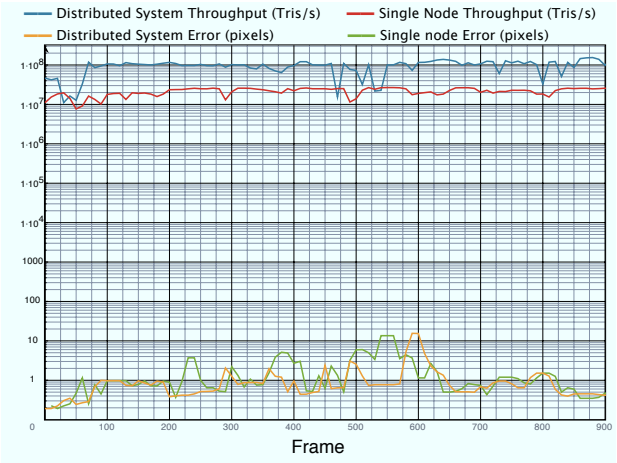

Fig. 12. Distributed throughput and error. Our system benefits from cluster-based rendering by greatly improving the rendering throughput. The overall error of the system remains comparable to a single-display renderer as the final display resolution increases as well.

Career Award, and an NVIDIA Fellowship (the views expressed in this work are not necessarily those of our sponsors).

References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins Univerisity Center for Networking and Distributed Systems, 2004.

[2] ATI. ATI Crossfire technology white paper. Technical report, ATI Technologies, 2005.

[3] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[4] P. Bhaniramka and P. C. R. . S. Eilemann. OpenGL multipipe SDK: A toolkit for scalable parallel rendering. In *IEEE Visualization 2005*, pages 119–126, 2005.

[5] W. T. Corréa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization*, 2002.

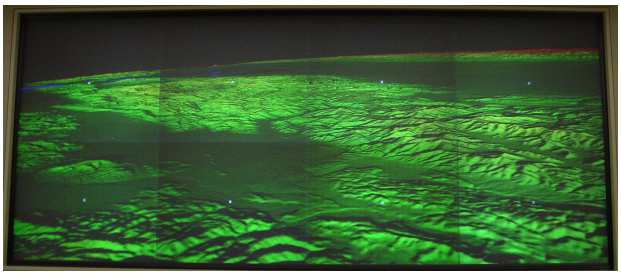

Fig. 13. Our system running the Earth data set on an eight-PC cluster with a tiled projector display

[6] A. Forsberg, Prabhat, G. Haley, A. Bragdon, J. Levy, C. I. Fassett, D. Shean, J. W. H. III, S. Milkovich, and M. Duchaineau. Adviser: Immersive field work for planetary geoscientists. *IEEE Compututer Graphics and Applications*, 26(4):46–54, 2006.

[7] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *SIGGRAPH 1989*, pages 79–88, 1989.

[8] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH 1993*, pages 247–254, 1993.

[9] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of SIGGRAPH 2002*, pages 693–702, 2002.

[10] L. M. Hwa, M. A. Duchaineau, and K. 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.

[11] D. Luebke. *View-Dependent Simplification of Arbitrary Polygonal Environments*. PhD thesis, 1998.

[12] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH 97*, pages 199–208, 1997.

[13] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[14] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: high-speed rendering using image composition. In *SIGGRAPH 1992*, pages 231–240, 1992.

[15] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 75–84 and 209, 1995.

[16] K. Niski, B. Purnomo, and J. Cohen. Multi-grained level of detail using a hierarchical seamless texture atlas. In *ACM Symposium on Interactive 3D Graphics and Games*, pages 153–160, 2007.

[17] P. Otellini. Keynote address from intel developer forum. Technical report, Intel Corporation, September 2006.

[18] B. Purnomo, J. D. Cohen, and S. Kumar. Seamless texture atlases. In *Symposium on Geometry Processing*, pages 65–74, 2004.

[19] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–84, 2001.

[20] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, 1999.

[21] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Symposium on Geometry Processing*, pages 146–155, 2003.

[22] P. Young. SLI best practices. Technical report, NVIDIA Corporation, July 2005.

This work was performed under the auspices of the U. S. DOE by UC, LLNL under Contract W-7405-Eng-48.