VIEW-DEPENDENT LEVEL OF DETAIL FOR THE PARALLEL RENDERING OF COMPLEX MODELS

by

Krzysztof Niski

A dissertation submitted to the Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

March, 2007

Abstract

The ever increasing complexity of 3D models generated by 3D scanners and through modeling requires ever faster graphics cards to render them at interactive rates. Unfortunately, the rate at which the performance of graphics cards increases is outpaced by the advances in scanning technology, making today's 3D models too large to render interactively. Because of the sheer amount of geometry to render, datasets such as the USGS Earth model are also too large to fit into memory, making the rendering even more challenging.

In this dissertation we present a new multi-resolution, multi-grained level of detail hierarchy along with new algorithms used to traverse this hierarchy. The new hierarchy builds on previous work by permitting any element in the hierarchy two degrees of freedom as opposed to a single degree for previous methods, permitting a more flexible traversal of the hierarchy.

The adaptability of the system is especially useful in the current computing environment, with its ever-increasing variety of options in terms of number and power of CPUs and GPUs. Because any two computers can have any combination of both CPU and GPU, our method allows the hierarchy to be reused and adapted at runtime rather then requiring a time-consuming re-computation of the data.

The Level of Detail system presented in this dissertation also adds new capabilities to single-computer rendering in the form of support for both multi-CPU and multi-GPU architectures, achieving considerable performance improvements by utilizing the parallelism available in the latest generation of computers.

The design of the system also makes it capable of rendering to a tiled display wall using a rendering cluster. The adaptive hierarchy and regular grid layout used in our method allows the entire cluster to use the same hierarchy, regardless of the load on the machine or the machine configuration.

We demonstrate our approach on both large triangle meshes and terrains with up to billions of vertices.

Advisor: Professor Jonathan Cohen

Readers: Professor Jonathan Cohen

Professor Michael Kazhdan

Table of Contents

1.Introduction

1.1. Introduction

The field of computer graphics deals with representing physical and abstract models and images on a computer screen. The field is comprised of many sub-areas, such as medical imaging, surface reconstruction and visualization. This dissertation deals with the visualization of 3D models previously captured using 3D scanning, user modeling, or other input methods.

The ever increasing complexity of 3D models generated by 3D scanners and modelers requires ever faster *graphics processing units* (GPUs) to allow interactive rendering. Unfortunately, the rate at which the performance of graphics cards increases is outpaced by the advances in scanning technology, making 3D models generated today too large to be rendered directly using the latest GPUs. Even the fastest graphics processors available today are capable of rendering only around 400 million triangles per second under ideal circumstances. While impressive, this artificial number is insufficient considering that we wish to render at interactive rates of between 20-30 frames per second. At these frame-rates we can render at most 20-25 million triangles per frame when using the best-case scenario triangle throughput. Unfortunately this rendering throughput is highly optimistic, representing the rendering of a 2D mesh with no normal, color, or texture coordinate data. Once the required vertex attributes are added to a mesh the rendering

throughput drops significantly, to approximately 200 million triangles per second, or 6-10 million triangles per frame when maintaining an interactive frame rate.

Given the maximum frame rate of 200 million triangles per second and models with up to billions of triangles we can see that they can no longer be rendered interactively. The problem is further compounded by the storage requirements of these large models. For example, a mesh of 100 million vertices with connectivity information will require at least 6Gb of storage. This greatly exceeds the available GPU memory as well as the system RAM available of most desktop computers, forcing either a reduction in data size, or very expensive paging.

As a result, numerous methods have been created to render such data sets, or representations thereof, at interactive rates without sacrificing the visual quality of the original data.

The general solution to this problem is the use of Level of Detail (LOD) systems. The 3D model is represented by a hierarchical, multi-resolution tree with the set of nodes at a fixed depth partitioning the 3D surface into individual surface patches. The hierarchical structure is obtained by having child nodes partition the domain of their parent. The multi-resolution structure is obtained by associating a triangulation to each node such that the children in the tree refine the triangulation of their parent.

Using this structure, a representation of a surface can be obtained by defining a *cut* through the tree, and rendering the triangles associated to the leaf nodes in the clipped tree. The creation of the cut is guided by a variety of parameters, including a desired triangle count or geometric error of the final model. The *adapt* process uses these user-

specified parameters along with the current camera parameters to create a cut that satisfies the user constraints while optimizing the visible areas of the model.

The key limitation of this method is its dependance on a fixed-granularity hierarchy. By linking the resolution of the nodes to their depth, previous systems require the use of high node counts in order to render the model at high resolution. This can greatly reduce the performance of the hierarchy cut creation process, making it the bottleneck of the system.

In this work, we propose a novel LOD system that decouples the resolution of the node from its depth in the hierarchy, allowing the surface patch associated to a node to be rendered at a number of different resolutions. As a result, we obtain a more versatile level of detail representation, providing a basis for addressing a number of practical challenges in 3D rendering. These include:

- **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.

- **Loosely constrained hierarchy neighbors:** Compared to most 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.

- **Parallel Adapt:** The use of independent adapt threads allows our system to divide the hierarchy adapt workload, and adapt the model in parallel. This reduces the adapt time per frame, allowing more CPU time to be used for other tasks.

- **Multi-GPU Rendering:** The design of the system greatly simplifies the use of multiple GPUs to perform the rendering of large scale models, allowing a high-resolution model to be rendered on a single screen without the network traffic penalty.

- **Distributed Rendering:** By further generalizing the multi-GPU rendering our system can be used to perform distributed rendering. Because of the relatively coarse-grained parallelization scheme and the single-pass cut merging, our system requires only loose synchronization with little network synchronization 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.

1.2. System Overview

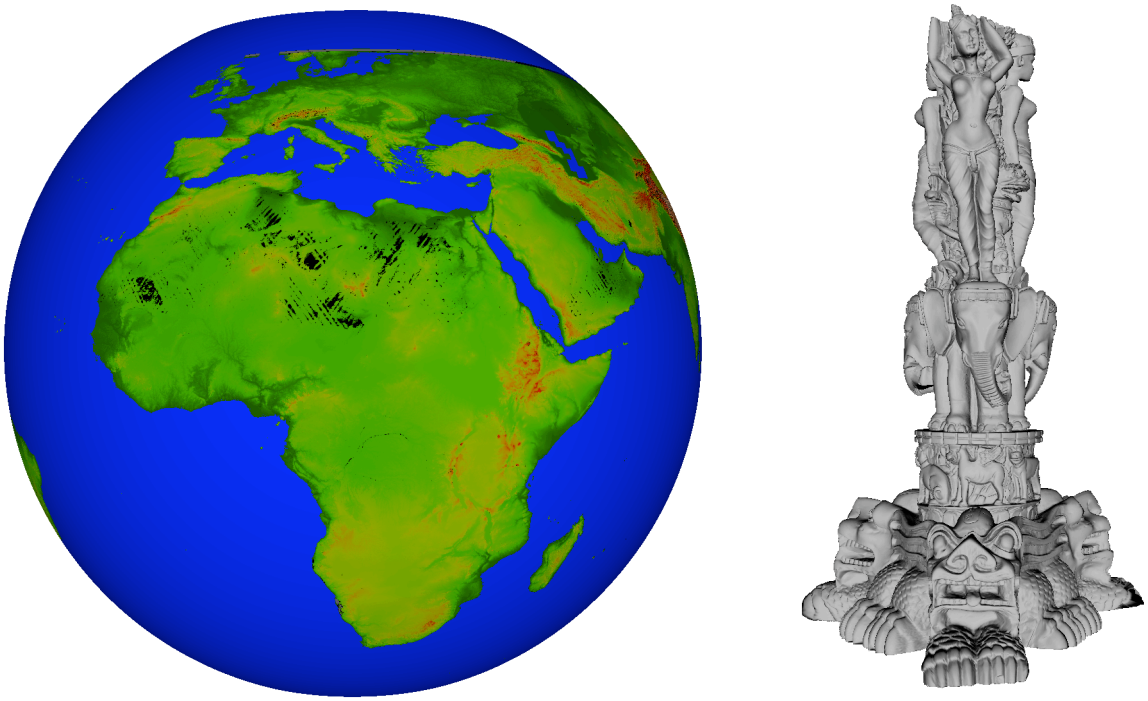

Figure 1.1: Large scale model rendering. On the left: USGS Earth Dataset, 44 Billion triangles; On the right: Thai Statue model, 100 Million triangles

Modern graphics cards are capable of rendering huge models at interactive rates. Models as large as 5-10M triangles can be rendered at rates of over 30 frames per second, with faster cards released every year. The on-board storage of high-end video cards is also increasing with every generation of hardware accelerators, matching the system RAM from computers of only a generation or two ago. As a result modern Level of Detail systems have to deal with data sets that outstrip the system memory capacity, and therefore must be capable of working out-of-core. Two examples of such models are dis-

played in Figure 1.1, showing the size of data that LOD systems have to operate on. The model on the left is particularly interesting, as it is built out of over 40 billion triangles, and easily exceeds available system and GPU memory. The magnitude of these models requires that the proposed system must be capable of not only interactive rendering, but also of managing large amounts of data. The system must be capable of adapting a large scale model quickly, loading the appropriate data elements in a non-disruptive fashion, and then rendering the data at interactive rates with little visual distortion.

The full system is comprised of a number of elements which combined form a Level of Detail system capable of rendering huge datasets. These parts can be broken down into four main elements: the LOD management system, the rendering system, the data management component and finally, the distributed aspect which handles the rendering of the object on a variety of parallel platforms.

1.2.1. Preprocessing

The preprocessing stage is the first step towards creating a LOD system. It prepares the data for the renderer, and creates the hierarchy used to control the system at runtime.

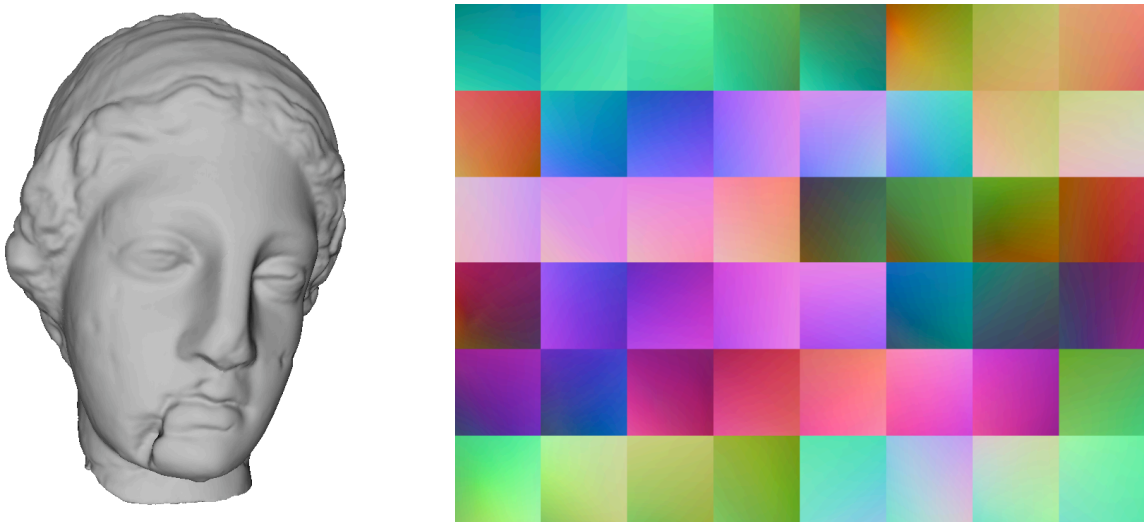

Figure 1.2: Parameterizing a model into a geometry atlas. The original 3D model is shown on the left along with its geometry atlas created through parameterization.

During the preprocessing stage several important processes take place:

- **Data Parameterization -** 3D models are converted into the 2D geometry image representation, generating an atlas which contains the models entire geometry. Height map data is divided into fixed-resolution sections.
- **Data Processing -** The created geometry images are written to a database file, facilitating data loading at runtime. The preprocessing stage also creates sub-sampled versions of the data to reduce the loading overhead of lower-resolution data.
- **Attribute Preservation -** The additional attributes such as normal or color data are created if necessary and stored in an atlas format along with the geometry

data. The additional attributes are stored in the most appropriate format, such as floats for normals and bytes for colors.

- **Hierarchy Creation -** The hierarchy is computed using the newly created data sets by processing each node, calculating its error values, and determining the neighbors. The resulting hierarchy can be used to adapt and render the model.

Upon the completion of the preprocessing stage all of the data is ready for rendering, including the hierarchy that is used to adapt the model during runtime.

1.2.2. Adaptation

The next stage in the rendering process is the *adaptation* of the hierarchy to the view parameters. During this stage the hierarchy is traversed and in order to locate geometry of appropriate resolution. This process terminates by creating a selection of hierarchy elements, called a *hierarchy cut*, which describe the rendering resolution for each part of the model. The cut will contain a selection of hierarchy internal and leaf nodes, each of which has a specific resolution associated with it.

The key difference between our method and previous work is the introduction of a 2-dimensional hierarchy, in which a hierarchy node can be refined not only through subdivision, but also through refining the node itself. As a result the adapt process becomes more difficult, as multiple aspects of the hierarchy need to be balanced and optimized.

We present a new algorithm that can be used to select a cut through the hierarchy that satisfies both the node threshold specified by the user, and the triangle budget re-

quested. This new algorithm is based on the dual-queue algorithm and, like its predecessor, it uses frame-to-frame coherence to accelerate the adapt process.

1.2.3. Visualization

The use of geometry images allows our system to optimize and reuse large parts of the rendering data. Instead of sending lists of triangles to the GPU we send the geometry images as textures and sample them on the GPU to recreate the original 3D model as shown in Figure 1.3. In this figure we see a single geometry image and the resampled 3D model. By rendering an atlas of geometry images an entire model can be reconstructed and connected without cracks using our system.

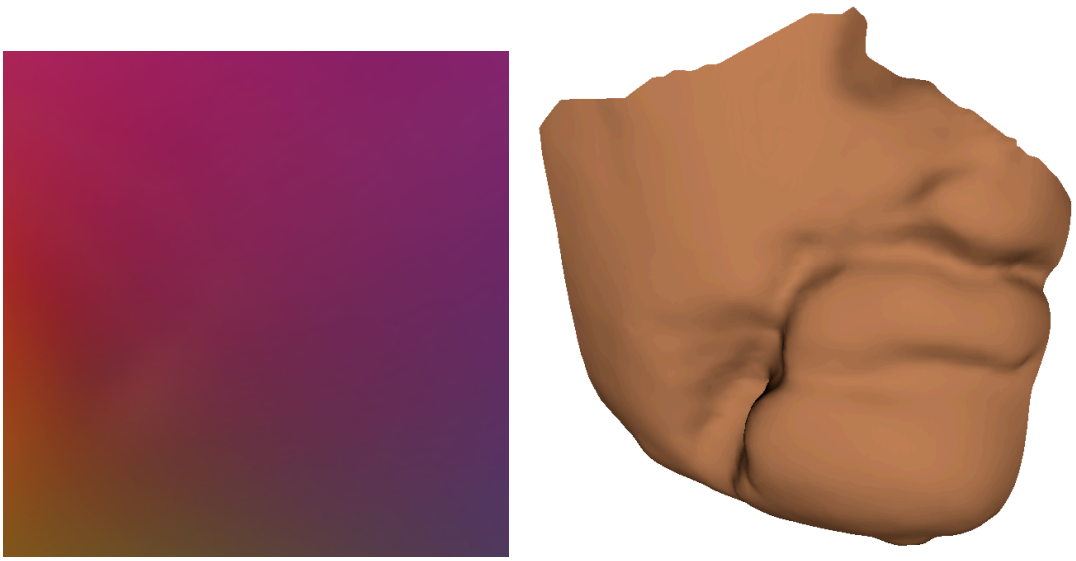

Figure 1.3: A single node from the geometry image atlas along with a piece of reconstructed geometry. The geometry is reconstructed through a process of sampling of the geometry image.

The geometry images are sent to the GPU as images, and they are sampled there using a triangulated 2D grid. This 2D mesh is used as both the *u,v* coordinates, and the connectivity information for the geometry, allowing a 2D image to be resampled into a solid 3D surface. By selecting the resolution of this 2D mesh we can control how many samples from the geometry image are selected during rendering, and therefore control the resolution of the final mesh as shown in Figure 1.4.

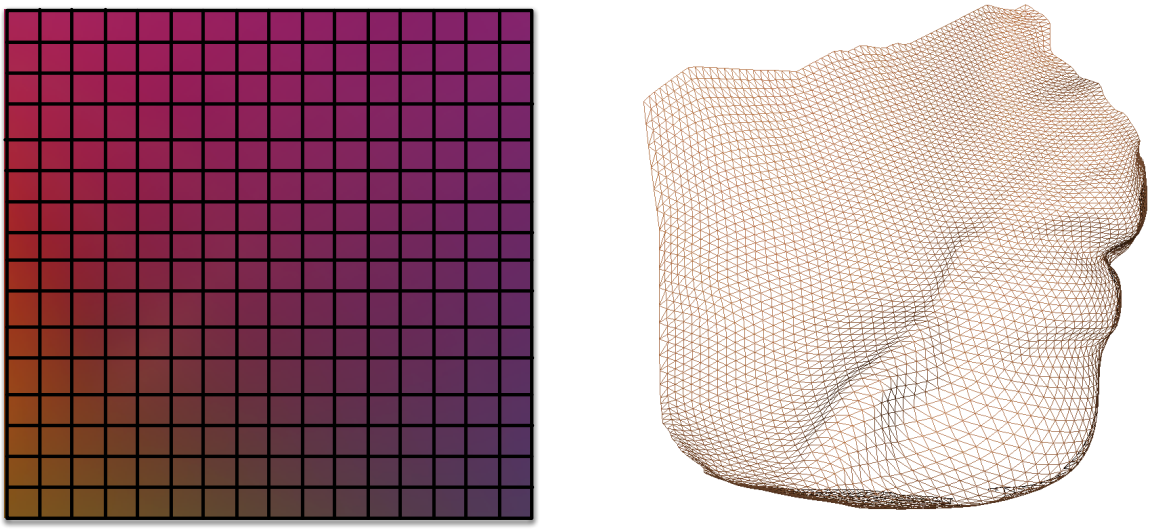

Figure 1.4: Resampling geometry from a geometry image. The image on the right is shown with an overlaid grid which represents the vertices and connectivity submitted to the vertex program as a 2D uniform grid mesh along with the geometry image to be sampled. By sampling the geometry image at the intersections of the grid lines and using the original connectivity of the 2D grid the mesh on the right will reconstructed.

While the geometry image representation has many benefits, we have also implemented a more traditional rendering method which processes the geometry images as streams of vertices with reusable connectivity data.

This process has numerous advantages, including:

- **Implicit Connectivity Information -** The 2D layout of geometry images allows a solid surface to be recreated by simply connecting neighboring samples from the image.
- **Reusable Topological Data -** The geometry images are sampled using a 2D mesh stored permanently on the GPU. Because we use uniform sampling to recreate the model, we can use the same mesh to sample every geometry image of the same resolution. This allows us to store a handful of 2D meshes and reuse them, instead of uploading separate index data for every geometry element.
- **Optimized geometry -** The 2D meshes used to sample the geometry images are trivial to optimize, and contain the bare minimum amount of data required to render a mesh. This greatly improves the rendering performance, allowing for more data to be rendered per frame.
- **Per-Fragment Data Mapping -** The use of geometry images creates parametrized meshes that can be easily textured. The additional data, such as normal maps or color maps can be applied in the fragment program, resulting in a higher-quality image, even when using a coarse underlying geometry. Furthermore, the resolution of all of the data is independent, allowing each of the resolutions to be adaptively selected
- **GPU-Based border merging -** By sampling the geometry images on the GPU we perform the border merging process in the vertex program. Through altering of the *u,v* coordinates used to sample the border vertices we can ensure that the

same samples are selected on both neighboring nodes, resulting in a seamless surface.

Due to the use of geometry images our method can take advantage of the implicit neighbor information present in this data layout, and reduce the amount of data sent per frame to the GPU.

1.2.4. Data Management

The data sets that this system is designed to work on will include models with billions of triangles, which can easily eclipse the available system memory as well as the available GPU memory. The system must therefore be capable of managing the vast amount of data that the model is composed of on the hard drive and in the system and GPU memory.

The first step in improving the loading system is the loading of data in separate threads, allowing the data to be loaded in the background while the main thread can render the model uninterrupted. This is vital to any out-of-core method as many standard I/O calls block the execution of the thread.

The use of geometry images in the form of 2D textures facilitates this process by making it easier to upload and remove large and coherent chunks of memory at a time. The geometry images also lend themselves very well to filtering, or mip-mapping, where a filtered lower resolution image is used instead of the original high resolution image when the surface is far away from the eye. By using this technique the size of textures can be easily reduced by a factor of 4 every time the texture is sub-sampled. This allows

the method to keep lower resolution texture in system memory, only loading the higher resolution ones when a node needs to be refined more then the current geometry image allows.

The geometry image representation provides additional benefits in the form of data reusability. The data stored in geometry images is in the form of a uniform grid which is sub-sampled and subdivided to cover sub-nodes and lower resolution nodes. This allows the same segment of data to be used for multiple nodes when the best data for that node is unavailable. This makes our data representation reusable, removing the possibility of a rendering stall caused by waiting for data to load.

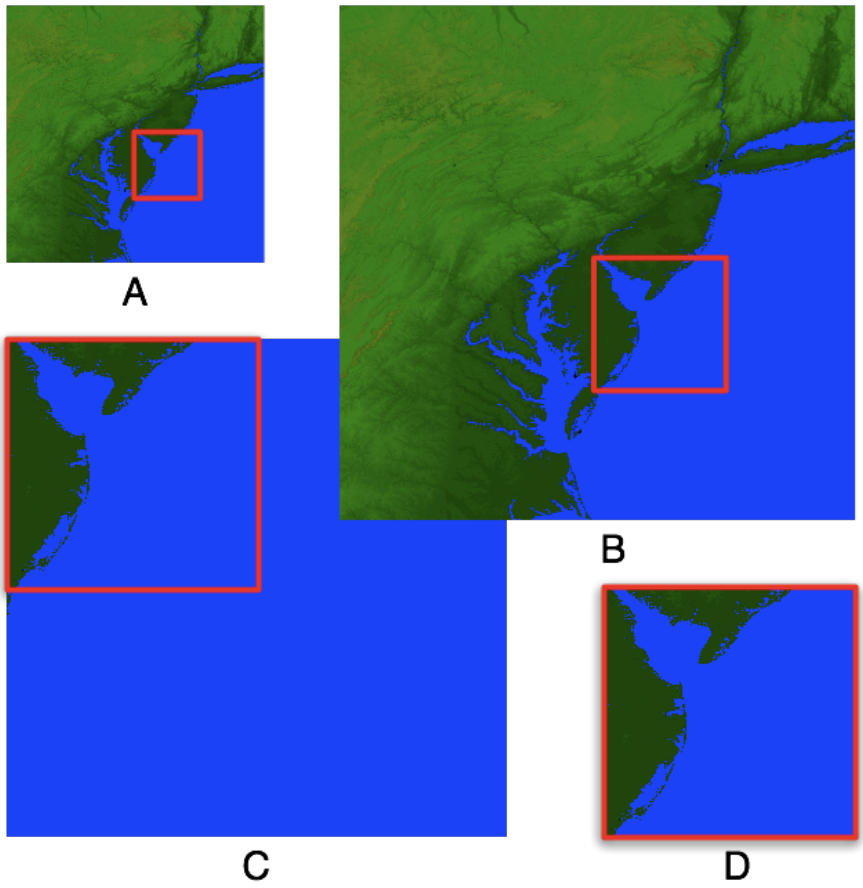

Figure 1.5: Data reusability example. This figure shows the data reusability process. Four sources of data are shown for a single node in the model. Source A is a low-resolution grandparent. Source B is a higher resolution grandparent. Source C is a high resolution parent, and D is the high resolution data for the node. Any of these can be used, and at least one is guaranteed to be in GPU

1.2.5. Parallel Adapt

The availability of multi-CPU and multi-core systems allows our method to parallelize the hierarchy adaptation process. By subdividing the view frustum into multiple tiles our system can adapt each tile independently, improving the overall performance of

the adapt subsystem. The parallel adapt is beneficial to all of the different modes of the system, including the distributed renderer, multi-GPU renderer and also the single GPU renderer that can now be operated using multiple adapt tiles.

1.2.6. Multi-GPU Rendering

The distributed algorithms investigated in this dissertation can also be applied to a computer utilizing a number of GPUs. The recent advances in video card and motherboard design have enabled the use of multiple GPUs in a single computer, allowing each GPU to render a different part of the scene. These segments can then be stitched and displayed on either a tiled display or on a single screen. This ability opens up new possibilities for distributed level of detail systems, since it replaces the network overhead present in the purely distributed methods with a smaller internal bus overhead.

The adaptation, management and rendering techniques employed in the multi-GPU methods are essentially identical to the ones presented for the distributed rendering systems, with the exception of the data loading system, which can take advantage of the shared system memory. This allows a smaller total amount of data to be loaded when compared to the distributed systems, which in turn helps to reduce the latency of the texture loads.

1.2.7. Distributed Rendering

The distributed rendering system presented in this paper uses the sort-first algorithm to distribute the rendering. The sort-first approach utilizes a cluster of computers,

each of which is connected to an output device. The output devices are arranged into a grid, forming a display wall in which each display is driven by a separate computer. This reduces the network overhead associated with single-output sort-first algorithms, in which each computer sends back its piece of the final image to a single computer that assembles the image on a single display.

The goal for the distributed system is maximum performance with minimal overhead. This goal has led us to the development of a distributed adapt and rendering system capable of performing its task with minimum inter-node synchronization. The loosely synchronized system allows each rendering node to create its own cut through the hierarchy which is subsequently unified with its neighbors and rendered with minimal network synchronization overhead.

2.Previous Work

The field of level of detail dates back to the beginnings of computer graphics. There have always been, and there always will be, models that are too large to render directly at interactive rates. As a result Level of Detail systems were created and used to simplify these models and render them more quickly. The first paper to propose this approach was described by Clark [Clark 1976]. In his paper Clark proposed the notion of replacing the original model by a simpler shape which will resemble it when the object is far away from the camera. For example a sphere can be represented by an octahedron when it is sufficiently far away, while a model of the human body can be represented by a rectangle if it covers only several pixels. The idea of using simple proxy models to represent more complex ones has evolved greatly from that original paper, and has been used ever since to visualize more complex data.

The additional work in this field can be divided into several major sections dealing with different aspects of the problem, starting with the creation of the proxy models through to full-fledged Level of Detail systems capable of processing and rendering of huge models.

2.1. Level of Detail

The first suggested method of generating these substitute models was to create them by hand by guessing the correct shape to use when an object is far away [Clark 1976]. The model is then substituted based on the distance from the camera and the area

occupied by the model on the screen. A similar approach is suggested in [Crow 1982], where more intricate proxy models are created to represent the original. In his paper Crow also suggest the creation of the proxy models automatically rather then by hand.

One of the first methods to automatically create a hierarchy of proxy models that can be used to represent the original is presented in [Floriani 1989]. In this paper the author suggests the use of the Delaunay triangulation method to create a hierarchy of proxy models increasing in complexity. The major disadvantage in this approach is its reliance on the Delaunay method which is relatively expensive to compute for general models. A similar solution is proposed in [Michael et al. 1991] where the densely triangulated model is simplified by growing polygons on the surface up to a specified error threshold. This method can suffer from initial placement problems and high computational cost because, unless all possible locations for polygon placement are investigated, only a local minimum can be reached. The method proposed in [Turk 1992] performs the simplification of the original model by re-triangulating the model to a user specified vertex count. In [Schroeder et al. 1992], Schroeder introduced a mesh simplification algorithm based on vertex removal and re-triangulation. This improves on [Turk 1992] by affecting only a local area rather then the entire model, allowing their method to simplify the model more quickly. The re-triangulation can, however, create less optimal shapes. This was further improved upon by Hoppe in [Hoppe 1996], where the model is simplified by removing one edge at a time from the mesh. The edge collapse method fuses two vertices together, leaving no hole in the resulting mesh. This method uses a priority queue to remove the triangles which will alter the shape the least, thereby optimizing the proxy

model. This method works well if minimizing the geometric error is the priority, and allows the simplification process to be fully reversible since only one edge is removed at a time.

2.2. Out of Core Simplification

While these approaches work well for small models that fit into the main system memory, they are less then optimal for large models that are too large to load into RAM. Because the majority of simplification algorithms have large memory requirements they would be unable to efficiently simplify large models due to limited system RAM and slow secondary storage. As a result, methods have been developed to simplify and adapt these larger models. One of the first of these methods is presented in [Lindstrom and Turk 2000] where the model is simplified by using a uniform grid and simplifying vertices in each cell down to a single point. This creates an approximation since it assumes a uniform sampling of the points, and the grid size has to be small enough to capture the finer features. This method is improved upon by using an octree, which improves the simplification near areas of high curvature [Lindstrom 2003]. By subdividing the model into cells using a tetrahedral mesh and simplifying each cell separately a more accurate representation of the original model can be reconstructed [Ganovelli et al. 2004]. The resulting system benefits from this adaptive subdivision by being able to adapt each cell independently, operating in a coarse-grained fashion. The latest work in this area breaks surfaces down into patches that are simplified separately, allowing each patch to be adapted in main memory independently[Borgeat et al. 2005]. By creating the patches based not only

on a uniform grid, but also on additional surface parameters, more optimal patches can be created, allowing larger and more coherent pieces of geometry to be adapted together.

2.3. Error Metrics

In order to accurately simplify a model, a metric has to be used in order to determine which vertices to remove during preprocessing and which proxy model to use during runtime. In the original LOD methods presented in [Clark 1976],[Crow 1982] the simplification metric was non-existent since the proxy models were created by hand. These models were substituted based on the number of pixels occupied by the object [Clark 1976] or the distance to the viewport [Crow 1982]. The method presented in [Michael 1991] uses a metric based on the distance from the original vertex to the polygon being grown on the surface. This approach forms the basis of the error calculated in [Hoppe 1996], which is based on bounding spheres. This error metric does a good job of creating uniformly spaced triangles since it will try to make the distance between vertices uniform, but this is not always the optimal solution. By using error quadrics presented in [Garland and Heckbert 1997] an error is calculated which is based on the distance of the vertex to a plane. By using this approach the proxy model will have more triangles in areas of high curvature and fewer triangles in flat areas.

All of the error calculations presented above only take the geometric error into account, since they deal only with the movement of the vertices. Numerous methods exist when additional data is present, such as textures or per-vertex colors. For example the error quadrics [Garland and Heckbert 1998] can be modified to take into account textures

and colors. Similarly in [Cohen et al. 1998] the presence of colors and textures is taken into account when the model is simplified, resulting in a more accurate surface. Another method of calculating the error introduced by the simplification process is to measure it directly from the rendered image[Lindstrom and Turk 2000]. By rendering the image from multiple viewpoints before and after removing a vertex the exact error can be calculated by comparing the two images. The advantage of this method is that it minimizes the visible error from the simplification. Unfortunately this error measurement is very expensive, requiring parts of the model to be constantly re-rendered for every proposed simplification operation.

2.4. View Dependent LOD

The previously mentioned methods create a hierarchy of proxy models that are static and do not adjust based on the location of the camera with respect to the model. This is where view dependent simplification becomes useful by adaptively refining the area of the proxy model facing the user and coarsening the area facing away from the camera in real time[Xia and Varshney 1996]. While this is a good approach in theory, in practice this method is very slow and memory intensive, as it requires the continuous refinement and coarsening of the model. This process can be performed over successive frames in order to amortize the cost of the adaptation over several frames [Hoppe 1997][El-Sana and Varshney 1999] or by using individual objects from CAD scenes to perform a more coarse grained adaptation [Luebke and Erikson 1997]. This coarse grained approach has become more popular due to its ability to balance the computational

cost with the ability to perform view dependent adaptation [Ganovelli et al. 2004], [Borgeat et al. 2005]. In these methods the block size of the adaptation process is much larger then single triangle which allows for large models to be adapted to a viewpoint in real time.

2.5. LOD Systems

In order to tie in these level of detail creation and adaptation methods they need to be combined into a system that is capable of managing the various aspects of these methods. The first such system was described in [Clark 1976], where a hierarchical layout was created of the models in the scene, and depending on distance and visibility a cut was performed through this graph to determine which object to render and at what resolution. A more sophisticated approach was presented in [Funkhouser et al. 1992], where a comprehensive system is created that handles a walk-through through a large CAD model. This method computes the visibility of each object in the scene at every frame, and uses a lower resolution model if possible. Later methods also incorporate the management of data so that the data can be rendered even if the original data set does not fit in system memory [Chhugani et al. 2005], [Erikson and Manocha 2001]. These methods work well for CAD models, and require significant pre-computation in order to determine the visible set at each point in the scene. This is somewhat simplified when viewing large data sets such as terrains or scanned models which tend to be of simpler topology when compared to CAD models of buildings. Terrain models are especially convenient because they are usually based on height maps which can be easily processed by both a LOD sys-

tem and a visibility algorithm in order to determine the optimal visible set [Losasso and Hoppe 2004], [Rottger et al. 1998]. For large scanned models this is also somewhat easier as there are fewer disjoint objects in the scene. This allows the LOD system to subdivide the scene as necessary in order to balance block size and view dependent adaptability [Lindstrom 2003], [Ganovelli et al. 2004].

2.6. Parallel Level of Detail

Surprisingly little work has been done that integrates level of detail with parallel rendering. Level of detail has sometimes been listed as future work in papers about parallel rendering [Correa et al. 2002], and parallelization has been listed as future work in papers about level of detail [Luebke 1998].

The only published system we are aware of involving level of detail in a load-balanced, parallel rendering setting is by Samanta et al. [Samanta et al. 2001]. 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 [Correa et al. 2002], using a shared network disk for all distributed computers with cache in local RAM.

2.7. Distributed and Parallel Rendering

There has also been a significant amount of research done in the field of distributed rendering of 3D models. These generally divide into sort-first, sort-middle and sort-last methods, depending on the method used to select which object is sent to which rendering node[Molnar 1992], [Molnar et al. 1994], all of which have their unique advantages and disadvantages. Numerous systems have been created that utilize these sorting and compositing algorithms to render a scene on a cluster of computers. One of the first of these systems is presented in [Molnar et al. 1992], where a sort-last algorithm is used to composite pieces of an image rendered on separate nodes. One of the earlier works on sort-first algorithms is presented in [Mueller 1995]. It distributes the rendering process to a large number of nodes, thereby increasing the rendering throughput. Other methods improve upon this model by using different compositing methods [Bethel et al. 2003], [Samanta et al. 2000] and more efficient methods of transporting the data between the nodes [Heirich et al. 1999]. Other methods also assume that the data set may be too large to replicate on every node, and only replicate the necessary data[Samanta et al. 2001]. Recently new methods have been introduced which combine the benefits of both sort-first and sort-last methods by using an array of projectors instead of regular monitors. This allows a sort-last style algorithm without the necessity of reading back the data to a single

node [van der Schaaf et al. 2002]. By using the scalable tiles with projectors much network bandwidth is saved, allowing for a faster update rate of the screen.

One of the more general distributed rendering systems is Chromium [Humphreys et al. 2002]. The Chromium system replaces the OpenGL library present on the host system, allowing it to intercept all of the OpenGL calls issued. These calls can then be replayed to a variety of machine configurations, including render clusters. Chromium can also sort geometry into tiles, allowing it to render to a tiled-wall system without modifying the original application.

Some of the most recent work on parallel rendering has been seen in NVIDIA's SLI[Young 2005] and ATI's Crossfire[ATI 2005] methods that allow GPUs to operate in tandem using several parallelization modes. The Split Frame Rendering (SFR) mode creates tiles on the screen, one for each GPU, which are used to divide the work. While this mode greatly improves pixel throughput, the replication of vertex processing on both GPUs does not improve performance of vertex processing bound applications. The Alternate Frame Rendering (AFR) alternates the rendering between the two GPUs, improving both pixel and vertex throughputs at the expense of some input latency. Both of these SLI modes are very general, allowing the drivers to perform distributed parallel rendering without any modification of the original programs. 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[Bhaniramka and Eilemann 2005] 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.

2.8. Geometry Images

Geometry images are 2D images which contain the 3D geometry of the model they represent. The first mention of geometry images as a data representation was in [Gu et al. 2002], where a method to parametrize a model into a single rectangular geometry image. This method relies on cutting the model in order to accommodate non-genus zero geometry, which can create complicated continuity constraints. In [Losasso et al. 2003] the method was improved upon by restricting the geometry image creation to genus-zero models, allowing the entire model to be parameterized over a closed surface. A method of parameterizing arbitrary-genus models through subdivision is presented in [Sander 2003]. This method produces arbitrarily-shaped patches, making it difficult to resample the original model. Finally in [Purnomo 2004] a method is presented that can parameterize a 3D model of arbitrary genus into a seamless atlas of rectangular geometry images. It performs a similar model subdivision to [Sander 2003], but stretches and resamples the resulting geometry images into a seamless geometry atlas.

3.Preprocessing

3.1. Overview

The input data for this method includes both arbitrary-topology 3D meshes and 2D height-maps, ranging from small models to datasets of over 22 billion samples. The raw data entered into the system first needs to be converted into the representation used by the system. For 2.5D data this process is relatively simple, and involves the cutting of the data into more manageable blocks, as well as the creation of mip-maps of the data. When 3D data is used as the input the preprocessing stage is more involved, requiring the 3D data to be converted into the 2D geometry image representation.

The 2D geometry images are then processed and stored in data files, storing the textures in a more GPU-friendly format, so they can be read from the disk and loaded as quickly as possible.

Once the geometry images are created the hierarchy generation stage begins, during which the run-time data hierarchy is created. This hierarchy contains the information required to control the adaption of the model to the scene, including the bounding volume hierarchy and the geometric error of each level of detail for each patch in the object.

3.2. Data Representation

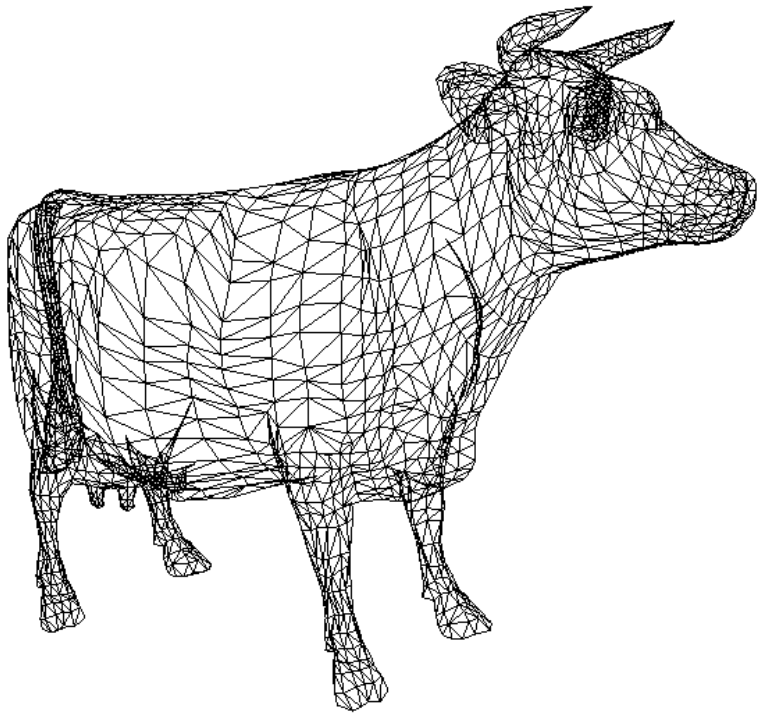

Figure 3.1: 3D model representation. A 3D model is composed of vertices connected into triangles, which drawn together create a model.

The traditional computer representation of 3D models is composed of vertices which are connected into triangles, forming a model with a solid surface as shown in Figure 3.1. This representation has dominated other representations for interactive computer graphics. While it has many shortcomings, such as the use of flat triangles which require oversampling in areas of high curvature, the triangular representation remains popular to this day. The graphics hardware designed and built today specializes in rendering triangular meshes, reaching extremely high rendering throughputs.

The triangular representation is not, however, perfect. It is difficult to perform stitching and cutting operations on the mesh due to its irregularity. This is especially evident in coarse-grained Level of Detail methods which attempt to break up a large triangular model into smaller sections, each of which has to be merged with its neighbors during rendering. In these methods this process is very difficult, and places many restrictions on such a system, such as LOD level difference between neighbors, or forcing the neighbor borders not to be simplified, wasting many triangles in the rendering process.

The method presented in this paper uses a different representation of the data, one that uses a 2D grid of data to represent the model. In this representation the geometric data is stored as a regular grid of 3D coordinates, shown in Figure 3.2, obtained by parameterizing the 3D mesh. This 2D data stores only the 3D coordinates, with the connectivity implied from the grid-like layout. In such a layout a vertex can be connected to each point in its one-ring neighborhood to form triangles, thus reconstructing the 3D mesh.

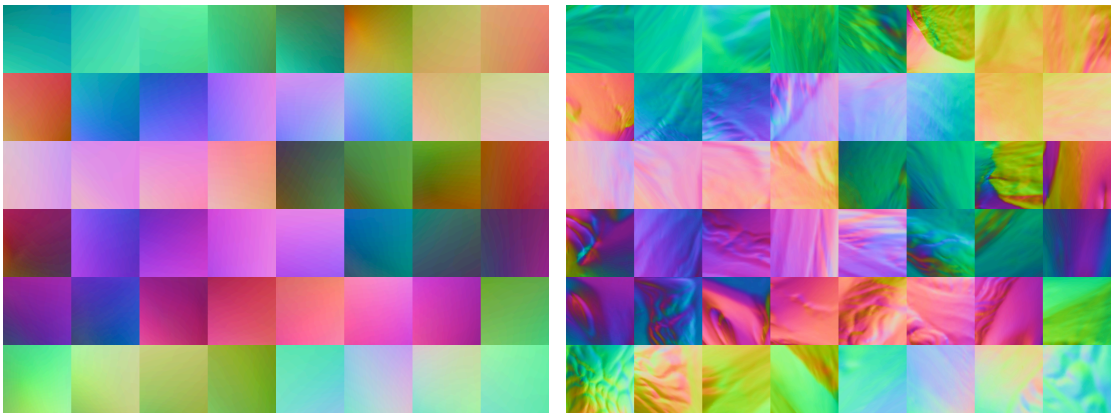

Figure 3.2: The geometry atlas representation. This is the data used to render a model using this system. The data is in the form of 2D images, which contain the 3D position data as shown on the left, and 3D normal data as shown on the right

This property of geometry images allows us to reconstruct a 3D mesh from the 2D geometry image through sampling and connecting of the vertices. By sampling the geometry image at every point and connecting neighbors the original model can be reconstructed at full resolution.

The regular grid structure of the geometry image data along with the implied connectivity greatly simplifies the process of subdividing and reconnecting of the 3D mesh. The very process of creating the geometry images causes the original model to be subdivided into multiple sub-regions, comparable to the subdivision process of traditional 3D models. The further subdivision of each geometry image is also trivial since each geometry image is a 2D rectangular array of data. The process of reconnecting neighboring patches is also simplified due to the regularity of the data. By assuring equal sampling at the border of both nodes along with a identical borders the nodes will sample the same vertices, and as a result will cause the edges to connect seamlessly.

3.3. Data Preprocessing

3.3.1. Data Acquisition

The data sources for this method are many, including 3D scanners, SIR (Space-borne Imaging Radar) shown in Figure 3.3, LIDAR (LIght Detection And Ranging), photometric techniques and regular laser range finding. All of these methods return data that is subsequently refined and joined together to create the final seamless processed output.

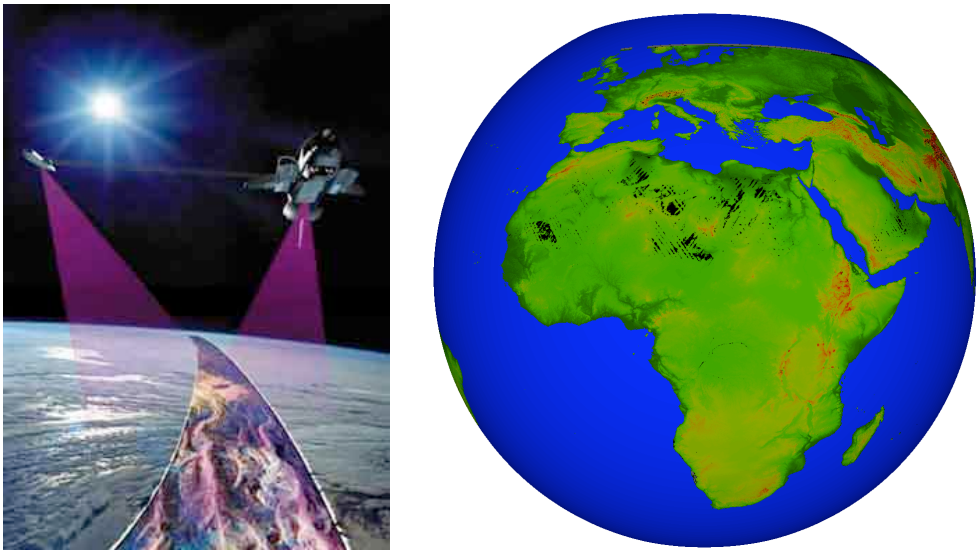

Figure 3.3: One form of scanning: The Shuttle Radar Topography Mission. The Space Shuttle is used to scan the Earth by bouncing a radar beam off the planets surface. The image on the right shows a rendering of the dataset acquired. Space shuttle image courtesy of NASA

The output of these methods falls into two basic categories: fully 3D data and 2.5D height-maps which contain height values for a 2D area. Both of these geometry

types can contain additional data such as normal and color information gathered during the scanning process. Because the 2.5D height-map data is already stored in a 2D format it requires little in the way of pre-processing. The data is merely split into more manageable chunks which allow it to be more easily stored and retrieved from the hard drive. The 2.5D data also requires far less storage then the 3D data since two of the dimensions are implied from the 2D grid storage structure. The only data that needs to be stored are the height values per pixel, requiring at most a third of the storage, depending on the initial scanning precision.

3.3.2. Data Preparation

The geometric data obtained during the scanning and modeling is converted into the geometry image representation through parameterization. The parameterization first subdivides the model, creating a number of surface patches. The surface patches are then converted into a planar representation and further partitioned into a number of square geometry images. The parameterization stage is described in full detail in [Purnomo 2004] and [Niski 2007].

The raw geometry images obtained during the parameterization or subdivision stage contain the geometry from the model, with a resolution that is a power of two. One of the first steps in the data preparation phase is the addition of duplicate borders to the geometry images. The borders are added such that the right border of a geometry image has the same column of pixels as the left border of its right neighbor, and similarly the bottom border contains the duplicate pixels from the top row of its bottom neighbor. This

raises the resolution of the geometry images by one row and one column, resulting in a final resolution of 2^n+1 by 2^n+1. The non-power-of-two texture resolution can cause problems with older GPUs that require power of two textures. Fortunately the latest graphics hardware is capable of utilizing such textures without forcing a large amount of space to be wasted as padding or falling back to software rendering mode.

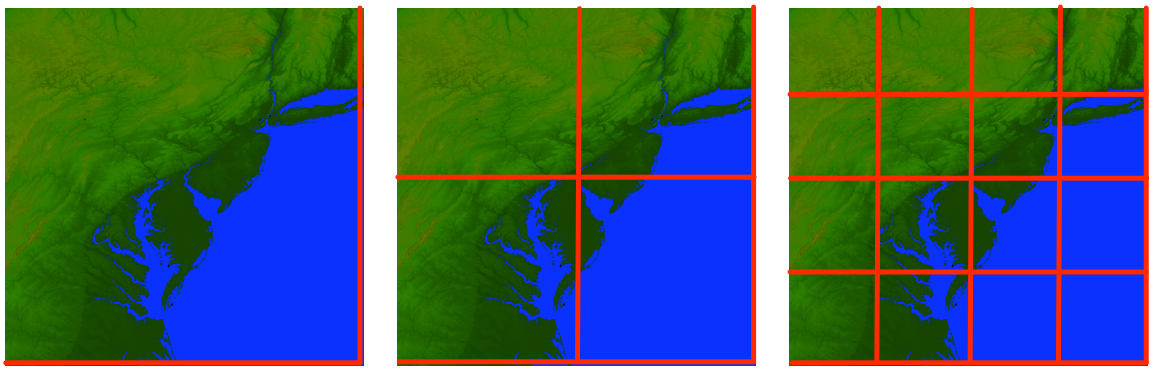

FIgure 3.4: Data reduction. The red lines represent the amount of data that would have to be discarded if regular power-of-two textures were used for one, two and three subdivision levels. Because every subdivision level creates more potential borders, more data would have to be discarded.

The benefit of increasing the resolution by one rather then trimming data is shown in the subdivision process. When a 2^n by 2^n texture is subdivided into four textures of resolution 2^{n-1} by 2^{n-1} the resulting textures will require two rows and two columns total to create seamless borders between themselves and their neighbors. Assuming that one row and one column of the original 2^n by 2^n texture has been used for the border matching, the new smaller textures would have to discard an additional row and column in order to fit the merge row and column into the 2^{n-1} by 2^{n-1} size texture. Thus for every sub-

division of the original node more and more data would have to be discarded, leading to either a loss in visual quality once the node has subdivided several levels, especially when low-resolution textures are used.

The 2^n+1 by 2^n+1 resolution textures on the other hand still create 2^n+1 resolution textures when they are subdivided. When a node is subdivided it is split in to two $2^{n-1}+1$ by 2^{n-1} textures, a 2^{n-1} by 2^{n-1} texture and a $2^{n-1}+1$ by $2^{n-1}+1$ texture. By adding an additional row or column to the non-$2^{n-1}+1$ textures the neighbor borders are preserved without a loss of data.

3.3.3. Texture Storage and Filtering

The geometry images are stored in a texture database which can be quickly traversed in order to locate the required data segment. Several of these database files are created, storing successive mip-map levels of the data. While this increases the storage by approximately 30%, it reduces the amount of data that has to be loaded when the full resolution data is not required. The storage of lower resolution data also allows us to store filtered versions of the data, reducing the high frequency noise introduced by a simple nearest neighbor sub-sampling that would be performed otherwise.

The stored mip-map data uses bilinear filtering to create each texture level, creating a better interpolated version of the texture then using nearest-neighbor sampling. The bilinear filtering is not used for the geometry data as it would interfere with the border matching process, and could also cause features to shift.

The resulting texture database files store the geometry images at uniform resolutions. This means that regardless of the mip-map level of the data, it is still stored at the maximum resolution supported by the GPU. Several textures from the previous mip-map level are combined into a single texture in the current mip-map level to create these larger textures. This creates higher resolution textures, which in turn can be used to cover a larger area of the original model, so that the final model can be rendered using fewer nodes. While storing and using multiple smaller textures does not prevent the use of larger, merged nodes, it does decrease the performance since multiple texture fetches would be required to load a single texture.

In the preprocessing stage we also compute normal maps if they weren't provided with the data. The normal computation is straightforward and works by computing the gradient from the geometry images. The normal map is also stored in a database-like format with its mip-maps which are computed using bilinear filtering.

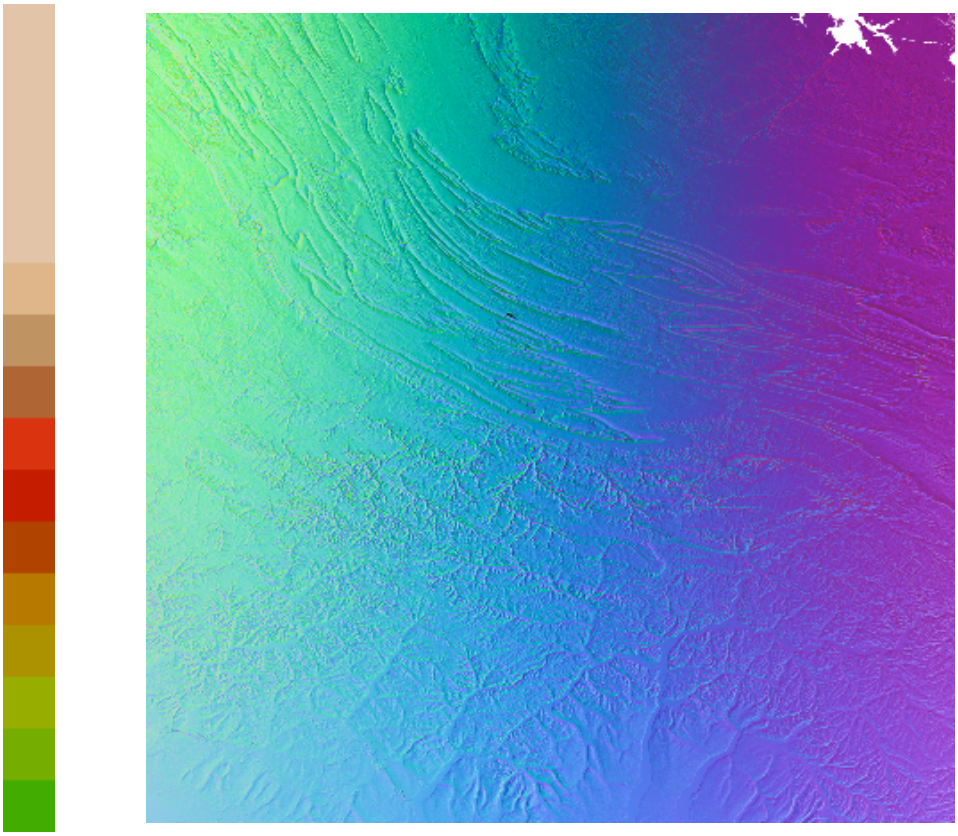

Figure 3.5: Color ramp and normal map examples. The color ramp is a 1D texture used to color the terrain models based on the elevation of the features. The 2D normal map is used to provide the shading information for the model.

Finally, we also compute color information for height-map models using a coloring scheme based on the height of a vertex above sea level. A color ramp, shown in Figure 3.5, is used in lieu of a high-resolution three channel image, which saves GPU memory. While this information can be computed at runtime in the fragment program, using a separate texture allows different resolution textures to be used for the color information and the geometry information, which saves GPU memory, and allows the separation of texture resolutions.

3.4. Hierarchy

3.4.1. Hierarchy Overview

The latest generation of Level of Detail systems have used partitioning techniques to create a coarse-grained LOD method. By partitioning the original model into smaller elements each piece can be adapted individually and rendered together to form the final object.

Since traditional LOD methods operate on 3D meshes, they are forced to use 3D subdivision methods. The most common subdivision methods are the uniform grid and feature based subdivision schemes. In the former the model is simply cut up into uniform cubes, each of which contains a portion of the model. In the latter, the model is subdivided based on features such as sharp edges, separate objects or color or texture boundaries. Each of these methods requires extra processing to avoid degenerate nodes, and to assure a roughly equal triangle count per node. These subdivision hierarchies can then be traversed in a single dimension, refining the geometry through splitting a node into its children.

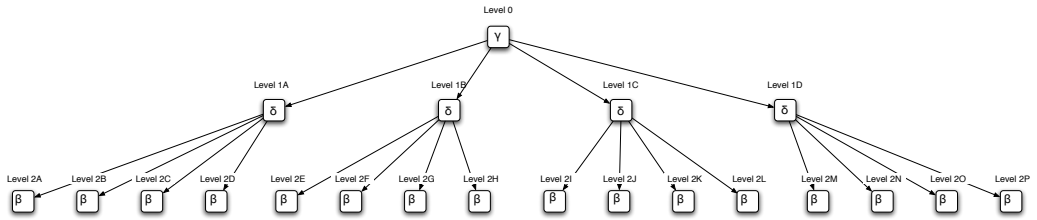

Figure 3.6: Traditional tree hierarchy. In this hierarchy a model can be only refined by splitting into its children, each of which has the same complexity as the parent. For example, to render the model at resolution β the hierarchy has to subdivide into 16 leaf nodes.

The key difference between previous hierarchies such as the one shown in Figure 3.6, and the one employed in our system is the two-dimensional nature of our hierarchy. While nodes can still be refined through subdivision, our method also permits a node to be refined by changing the resolution of the geometry of the node. As a result our method enables the geometry to be refined in multiple ways, creating a more flexible method of adapting a mesh.

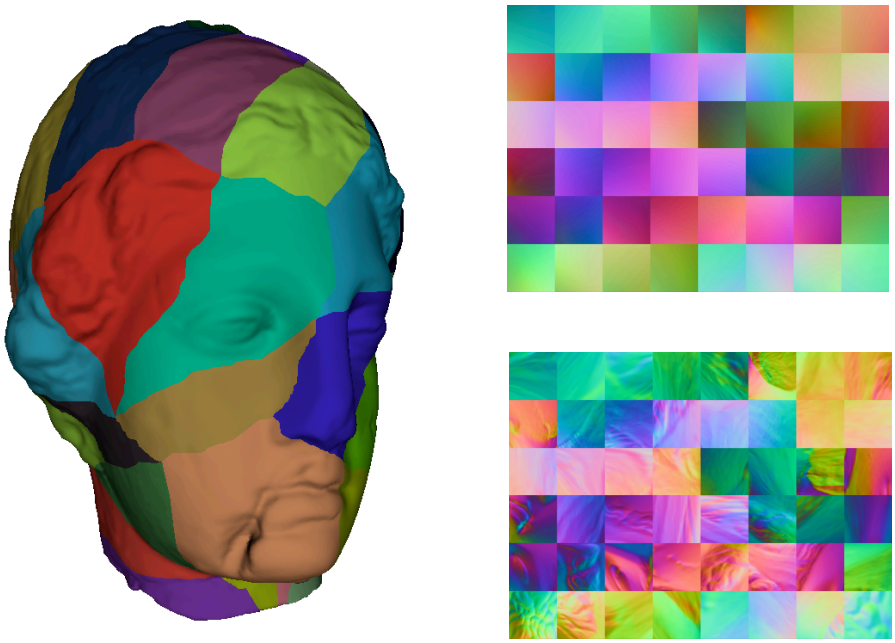

Figure 3.7: A model subdivided into quadrangular patches. The patches are flattened and stored in a geometry image atlas, along with a normal atlas used for normal mapping

The creation of such a hierarchy is made possible through the use of a 2D domain instead of the traditional 3D domain used in previous methods. When subdividing a 3D model each cell has to be self contained, ensuring that any triangles which cross the cell borders are truncated and added to both cells. The 2D domain of the geometry image simplifies the subdivision process, by allowing the subdivision scheme to operate on rectangular 2D data without worrying about maintaining connectivity information. This yields a subdivision that is guaranteed not to contain disjoint elements, simplifying the management and rendering of the hierarchy.

3.4.2. Hierarchy Benefits

The main advantage of the described hierarchy is the increased flexibility it provides. Previous LOD hierarchies have a limited, usually single, resolution at each hierarchy node, restricting the adapt process to a single dimension: the triangle count is directly tied to the number of nodes in the scene, forcing the refine process to increase both. This places a constraint on the number of nodes used to render a model: If a high triangle count is to be used, a large number of nodes has to be used as well. Since the additional nodes require additional CPU processing time, they increase the total time spent in the adapt algorithm. Furthermore because the granularity of the hierarchy is fixed during the preprocessing it is optimized for a single CPU/GPU combination, requiring a lengthy recompute should the computer configuration change. The fixing of detail level to the number of nodes used also prevents these hierarchies from reliably discarding nodes which are partially outside of the view frustum, since this would require the node to be subdivided, increasing the number of triangles rather then decreasing it (Figure 3.8).

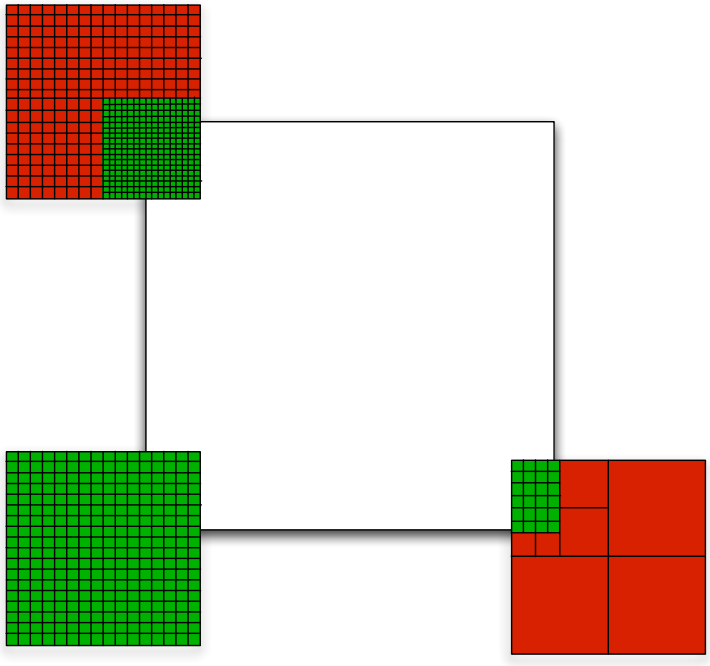

Figure 3.8: Hierarchy adaption example. The ability to both refine/coarsen and split/merge nodes allows our method to perform more efficient culling. In this example both of the geometry elements are rendered at the same resolution in the view frustum, with green regions drawn and red regions culled. On the lower-left triangles are wasted by being drawn off-screen since the node cannot be split. The top-right example attempts to discard some of the geometry by increasing resolution. The example on the right shows the benefit of our method. The areas off-screen are coarsened and culled, preventing triangle wastage without increasing the resolution.

The hierarchy created using the presented method is free from these restrictions. Each node in the hierarchy is capable of being rendered at multiple resolutions, ranging from coarse to fine. As a result, there are multiple ways to render the same model at the same resolution: It can be rendered using any number of nodes, the exact count of which can be selected by the user at runtime. Our hierarchy therefore allows the model to be

rendered not only with a user specified triangle budget, but also with a user specified node budget.

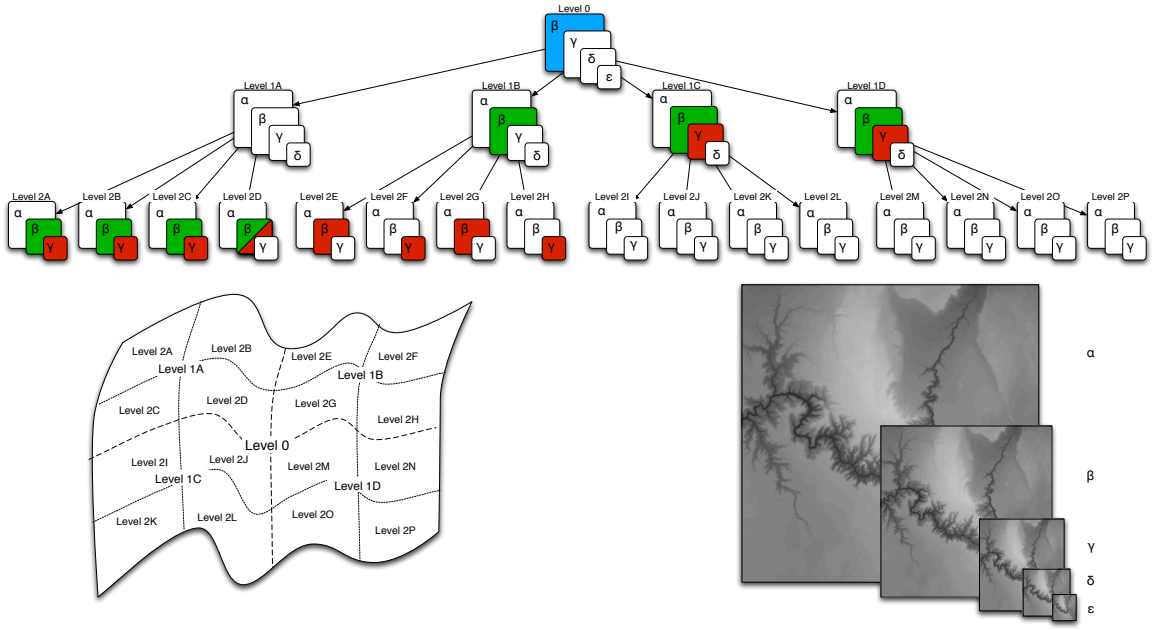

Figure 3.9: This image shows the quadtree hierarchy used in this paper. This example shows the hierarchy for a single top level node, which is subdivided as shown in the lower-left image. This hierarchy is capable of both subdividing the node domain, as well as refining and coarsening it.

Given this new degree of freedom, there are multiple cuts through the tree and resolution choices that achieve the same error bound and triangle count as shown in Figure 3.9. For example, the blue cut contains one node, and the green cut contains seven nodes. However, both cuts render the chart entirely at resolution 6. 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 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.

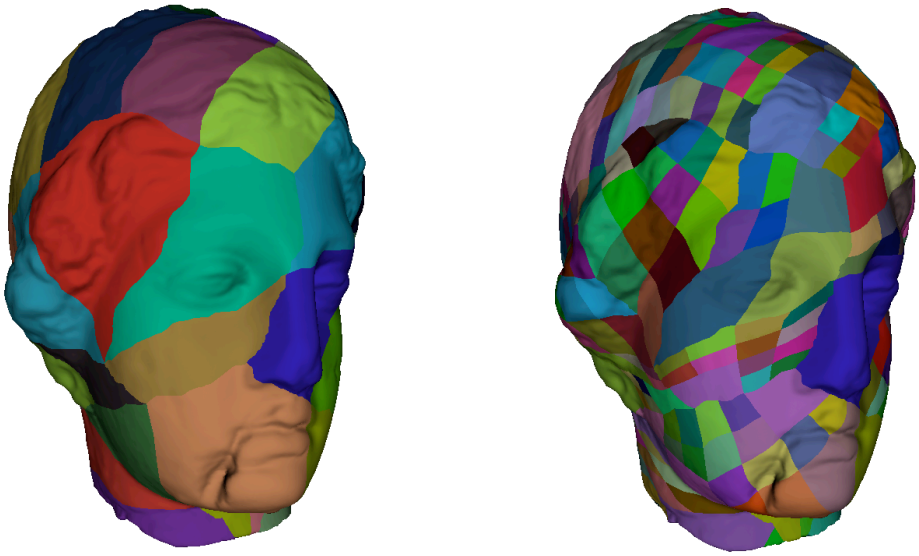

Figure 3.10: Different node layouts. The image on the left shows the model rendered using only the top level nodes. While this uses fewer nodes, it reduces the adaptivity of the system. The model on the right shows the benefit of using a larger number of nodes: By selecting the resolution on a per-node basis we can refine small areas as necessary, leaving other areas coarser while still maintaining the same error threshold.

This ability, combined with the ability to control the GPU load using the LOD system, allows our method to independently control and balance the GPU and CPU workloads. Through control of the node budget the user can adaptively select how much CPU time is spent performing the adapt process; By using a large number of nodes the user can reduce the maximum error in the scene through increased adaptability at the cost of a longer adapt time. Conversely, the user can use fewer nodes which may increase the maximum error, but will require less time to adapt. This added degree of flexibility al-

lows our method to adapt to a variety of host configurations without a lengthy re-processing step.

3.4.3. Partitioning Algorithm

The most common methods of 2D subdivision are the KD-tree and the quad-tree hierarchies. While both of these methods subdivide a 2D space into a tree of sub-spaces, they do so in different ways. First, the KD-tree is a binary tree, which subdivides each node into two children; the quad-tree, as its name implies, subdivides each node into four sub-nodes. The second, and more critical, difference is the predictability of the subdivision. While the quad-tree method performs the subdivision in a stable location, the geometric center of the 2D space, the KD-tree generally subdivides a region into two regions of equal density. Because the density can be arbitrarily defined, and the subdivision is performed along only one axis, the matching of node borders is far more difficult to achieve. Because of these inherent problems with the KD tree approach the subdivision algorithm chosen for this system was the quad-tree algorithm.

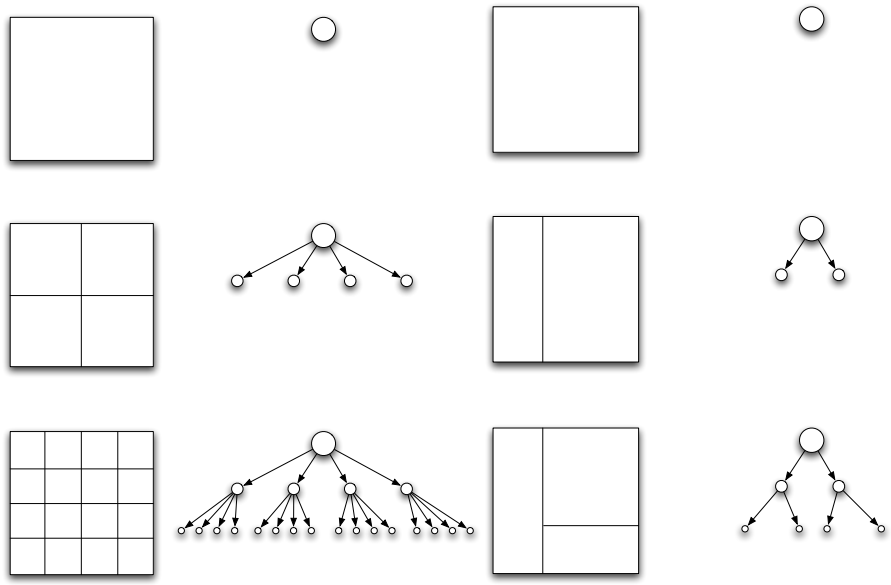

Figure 3.11: Quad-tree vs KD-tree example. The image on the left illustrates the layout of a quadtree. The root of the tree is a single node, which is subdivided into four nodes, each of which is further subdivided into four children. The binary KD-tree is shown on the right. It allows for a non-uniform subdivision of the domain, which is beneficial in some applications, but makes the border matching

3.4.4. Hierarchy Construction

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. Each of the children is then 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.

The hierarchy creation begins by associating an area of the 2D chart domain to the root of the tree, with each child node covering a portion of the original area. The tree is then iteratively subdivided down to the lowest level and for each leaf node several actions are performed:

- **Compute the bounding volume -** We compute the bounding box of the geometry contained in the section of the geometry image covered by this node.
- **Compute the geometric error for the node -** The geometric error for the leaf node is computed for every resolution of the geometry image that this node has access to.
- **Determine neighbor information -** We determine if the node is on the interior of the domain, or if one or more of its neighbors could come from another tree.

While performing the iterative subdivision of the nodes we also compute texture offsets to allow a sub-node to use a geometry image that covers a larger area then covered by the node. Each node holds an array of such texture coordinates so that it can map onto a texture from its parent, grandparent, etc. all the way to the root of the tree, as shown in Figure 3.12.

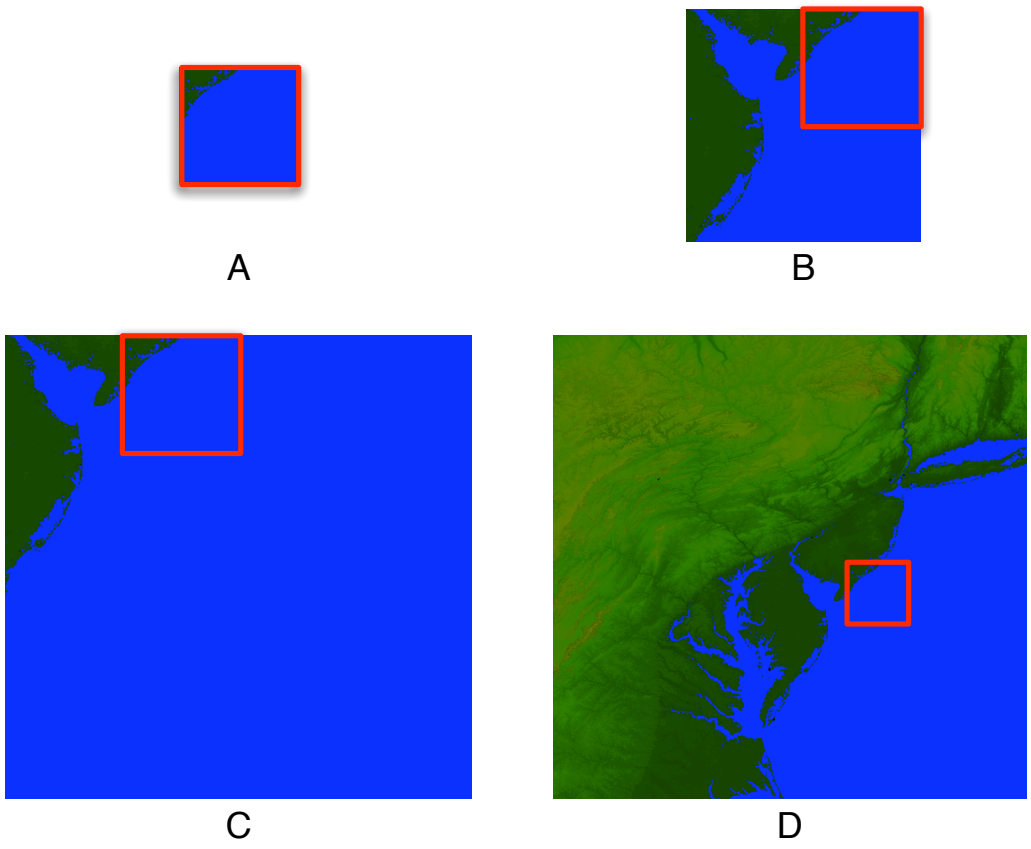

Figure 3.12: Mapping node data into ancestors. When child nodes are created they are given texture offsets which allow them to use data from their ancestors, providing data reusability while waiting for data to load.

The computation of the bounding volume is relatively straight forward, and essentially finds the smallest axis aligned bounding box that will contain every point in the geometry image. The geometry image is traversed, and every point is checked against the bounding box, and the box is expanded as necessary.

The error calculations require the knowledge of the sampling to be used for the given resolution. The error for a vertex is calculated by first finding the four nearest vertices that will be sampled during the rendering process, as shown in Figure 3.13. The high

resolution geometry image is sub-sampled at points P0, P1, P2 and P3, causing them to become neighbors. The point P4 is will be removed through the LOD process, inserting an error calculated by the distance from P4 to P5.

The point P5 is computed through bilinear interpolation of points P0-P3 at point P4, which represents the location of the original vertex after it has been removed. The distance between the two vertices is the geometric error introduced by removing that vertex;

By finding the maximum of all of the distances for a given resolution we calculate the maximum geometric error for the current leaf node.

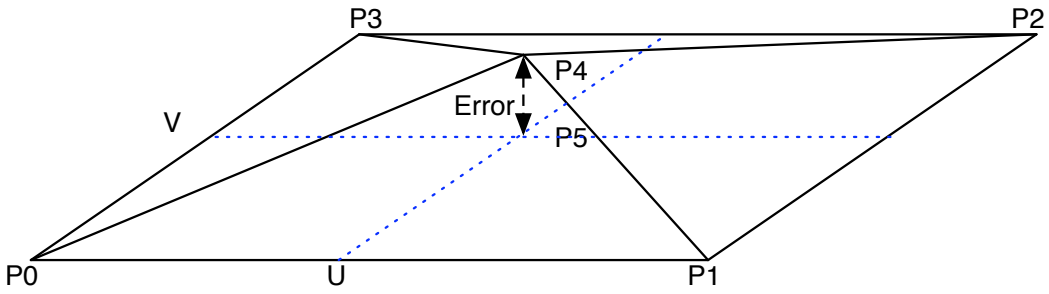

Figure 3.13: Calculating geometric error. The geometric error is calculated by measuring the distance between the original location, P4, and the new point P5 interpolated from neighbor points P0-P3

The error computation is performed for the leaf nodes, and then propagated through the rest of the tree by using the maximum error of the children as the error of their parent. A similar propagation process is used for the bounding volume hierarchy computation.

The hierarchy building stage is easily separable at many levels. First, each leaf node can be operated on individually since they do not share any information with their siblings. Second, since the final model is built out of multiple trees, each tree can be further separated into multiple threads. 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. The preprocessing could also be performed on a cluster of machines, as the per-tree data does not need to be shared until the processing is finished, and the final hierarchy is written out.

3.4.5. Results

One of the strengths of the presented method is its very fast pre-computation performance. The results given in Table 3.1 show the pre-computation times of the hierarchy creation stage. While they do not include the time required to parameterize the model, that stage of the pipeline is not directly linked to the preprocessing time since terrain models do not require it.

Model	Number of Samples	Top-level nodes	Hierarchy output size (MB)	Pre-computation time (m)
Earth (flat)	21.6B	70	69	120
Earth (round)	21.6B	70	69	718
US	5.5B	15	3.5	29
Puget Sound	268M	4	1	1.1
Thai Statue	130M	140	32	0.63
Thai Statue 2	130M	140	2	0.23
Tablet	8M	30	6.8	0.05

Table 3.1. Hierarchy pre-computation results. The pre-computation time for the models tested has remained low even for complex models such as the Earth with over 40 billion triangles. The output hierarchy size is similarly small, and depends on the number of top level samples and the subdivision depth. All of these times have been obtained using a Mac Pro with two dual-core Intel Xeon 2.66Ghz CPUs and 3Gb of RAM.

These results were obtained using a single processing node, a Mac Pro with dual Intel Xeon 2.66Ghz CPUs and 3Gb of ram running eight preprocessing threads. As these results show the preprocessing runs at over 2M triangles/second on a single machine, outperforming previous methods.

As the results also show, the output size of the hierarchy is very small, even for models with billions of samples. This is due to the minimal amount of data stored per node, as well as the use of quadtrees as the subdivision structure. Because the main vari-

able controlling the size of the hierarchy is easily adjustable the hierarchy size can be adjusted to fit the model being processed, as shown in the two Thai Statue models. The first Thai Statue model uses five levels of the quatree, which constructs approximately 140,000 subnodes, an unnecessarily large amount. The second Thai Statue model uses a subdivision depth of three, which constructs only 8k nodes, reducing the amount of storage required.

3.4.6. Summary

The hierarchy presented in this paper is an evolution of the hierarchies used in previous methods. The main contributions of our hierarchy are:

- **Two-dimensional hierarchy** - Our hierarchy can be adapted in two dimensions, allowing each node to be split/merged and/or refined/coarsened in order to adapt it to the error or triangle threshold.
- **Simplified border merging -** The combination of the quad-tree based hierarchy and geometry image based data representation greatly simplifies the process of removing cracks at node borders.
- **Data reusability -** The use of a quad-tree based hierarchy and 2D data allows child nodes to map into the data of their ancestors regardless of the resolution, creating a reusable representation in memory.

4.Adaptation

The hierarchy created during the preprocessing stage describes the model at every node with every combination of hierarchy depth and resolution. The goal of the adapt process is to select a cut through the hierarchy that satisfies a will satisfy one or more user-specified parameters while creating the most optimal cut through the hierarchy.

4.1. View-Dependent Adaption

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."

4.1.1. Error Bound Adaption

Consider the first problem with our modification. A simple top-down algorithm might work as follows. The scene is initialized with the minimum number of nodes (the root nodes), each of which is set to its coarsest level of detail. The resolution of each node is then increased until its current error is beneath the threshold specified by the user. This produces a model that satisfies the error budget requirement without taking into account the node budget.

In order to take the node budget into account at least one priority queue is required for the knapsack algorithm, the split queue. This queue will hold information for each node describing the cost of splitting one node into four sub-children. This splitting cost is calculated by taking the parent node and subdividing it into its sub-nodes, shown in Figure 4.1. Each sub-node is then adapted to the error budget, so that the error of the resulting sub-nodes does not exceed the error of the parent. The cost of splitting a node is then calculated as the number of triangles from the parent node, minus the number of triangles in the new sub-nodes. When this value is high, it indicates that by performing a split we can significantly reduce the number of triangles in the scene without increasing the error present.

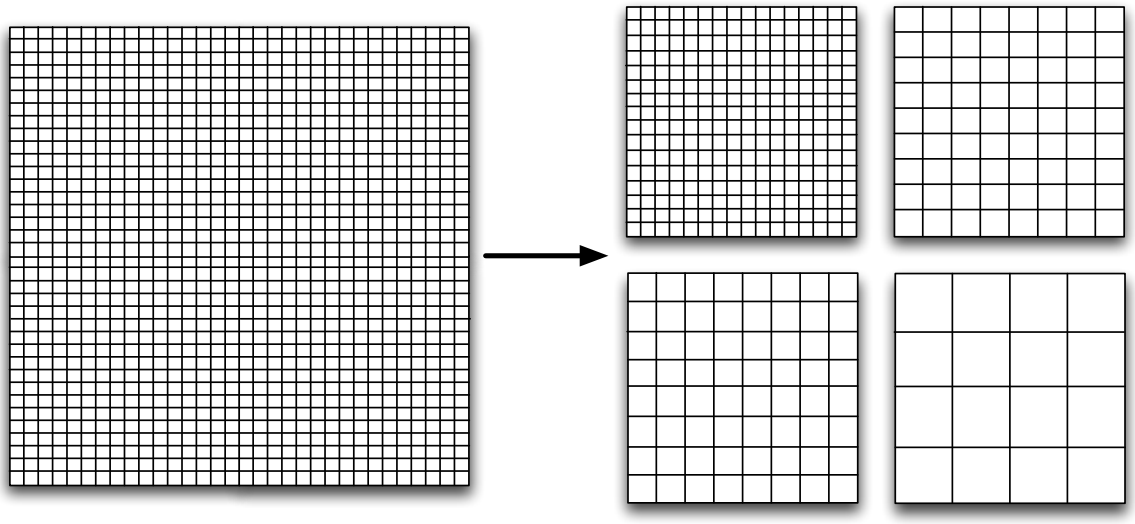

Figure 4.1: Calculating the split cost. The split cost is calculated by splitting a node into its four neighbors, and calculating the number of triangles required to maintain the same geometric error. This figure illustrates the benefit of this: If the geometric error is not distributed uniformly in the parent, the children nodes can be rendered using different resolutions, reducing the total triangle count.

The split queue is therefore used to determine the next node that should be split in order to reduce the triangle count. By performing this split operation, and adapting the resulting sub-nodes we can reduce the triangle count in the scene while maintaining the error threshold in the scene. When no more splits can be performed the scene adaptation is successfully completed, and the model can be rendered.

While this method will produce a model that is complete, and which will not go over the error threshold specified by the user, it is less then efficient. This algorithm requires the hierarchy to be re-computed from scratch every frame, resulting in a large computational overhead when a large number of nodes is present. The complexity of the

knapsack algorithm is on the order of *O(n)* where *n* is the number of nodes used to render the scene. Since this algorithm has to start anew every frame this complexity is always reached, and this algorithm is fairly inefficient.

A more efficient, frame to frame coherent algorithm for this problem is a variant of the well-known dual-queue algorithms [Luebke 1997]. The two queues are used to store the split and merge information for the cut, which is initialized using the data from the previous frame. The merge data is computed using a process similar to the split data, except the algorithm determines the number of triangles required to merge four child nodes into their parent without increasing the error in the scene. The larger the number, the more triangles need to be added to the model when the nodes are merged, and the less we want to merge it.

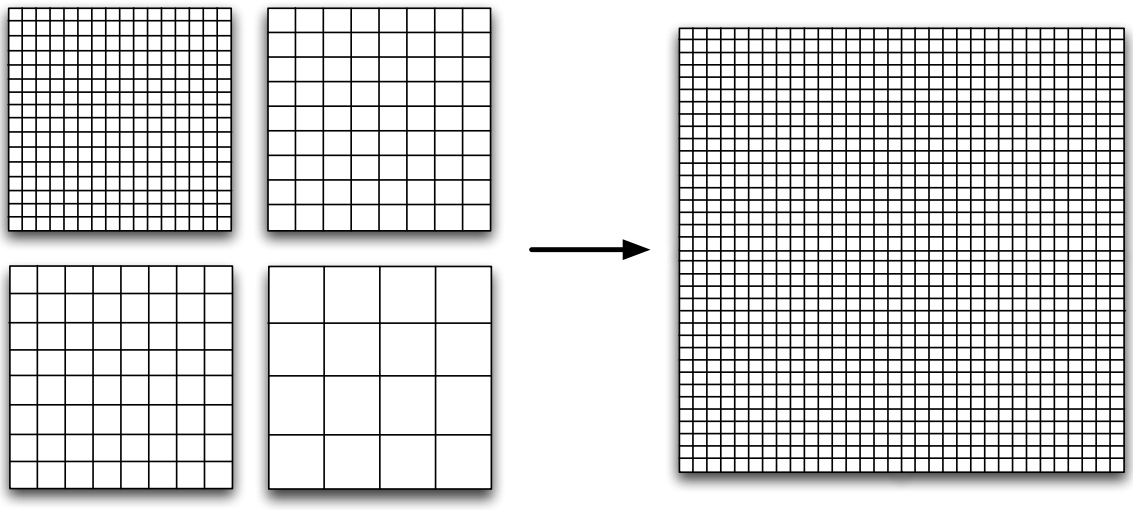

Figure 4.2: Calculating the merge cost. The merge cost is the number of triangles required to merge four children nodes into their parent without increasing the geometric error in the scene.

By using these two queues we can construct a error-bound adaptation algorithm that will use temporal coherence to adapt the model in a view-dependent fashion. Starting with the cut from the previous frame we can coarsen and refine patches in order to make the error of the current nodes fit into the budget. Once the current nodes are adapted, the merge and refine queues are checked in order to determine whether any nodes should be merged and split in order to reduce the number of triangles in the scene. The new nodes are once again adapted to the error threshold so the scene once again respects the limits set by the user. These operations are performed until the split and merge queues are balanced. This condition is dependent on the number of triangles required to perform the next merge operation and the number of triangles saved by performing the next split. If the number of triangles required for the merge is greater then the number of triangles saved by a split, the queues are considered balanced.

The complexity of the dual-queue algorithm is similar to the complexity of the knapsack algorithm, and it is *O(n)* in the worst case situation. However, since this algorithm takes the cut from the previous frame as its starting point, it will almost always operate at a much lower complexity of *O(log n)*, making this method much more efficient.

4.1.2. Triangle Budget Adaption

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 while maintaining a triangle budget. 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. The dual-queue method uses two queues, the refine and coarsen queues, to sort the current nodes based on the error of their next coarsen and refine operations.

The dual-queue algorithm operates using these queues and the current triangle count. The first stage is the refinement phase in which nodes are refined based on the order of the refine queue until no node can be refined without going over the triangle budget. The nodes are selected based on their current error in the scene, so that nodes with the greatest error will be refined first. Once the refinement stage is completed the coarsen queue is used to coarsen nodes if they are over the budget, or if the coarsen and refine queues are unbalanced. The balance of these queues is defined by the error of the next refine and of the next coarsen. If the next node to be coarsened has a lower error then the next node to be refined the queues are unbalanced, since the coarsen/refine operation will further decrease the error in the scene.

The adapt algorithm iterates between these two stages until no more nodes can be refined, the scene is under budget, and the queues are balanced. When this occurs, the scene is considered to be successfully adapted, and the scene is ready to be rendered.

The ability to adapt just the current nodes is insufficient in our case. Since our hierarchy permits nodes to both refine/coarsen and split/merge, we need an additional degree of control to adapt in both of those directions. We therefore 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.

We start by updating each node from the current cut in all four of the queues in order to refresh the queue states. We subsequently iterate over two phases: the refine/coarsen phase and the split/merge phase.

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. This process adapts the current selection of nodes to fit into the triangle budget specified by the user, without splitting or merging any of the nodes.

In the second phase, we propose a set of splits and merges of nodes, performed by simulating 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. The nodes are then split until no more splits can be performed. If the split/merge queues are unbalanced once no more nodes can be split, the split/merge operations will iterate until they are.

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 the error, but this would require many more cycles through the refine/coarsen queues.

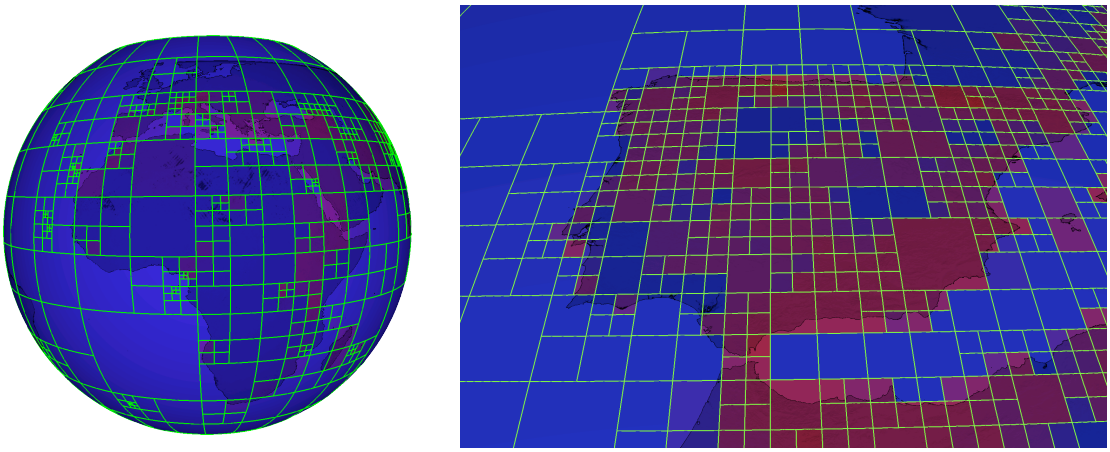

Figure 4.3: The Earth dataset adapted to a fixed triangle budget. The two images demonstrate the ability of the system to split and refine nodes with the highest geometric error. The triangle density of the nodes ranges from low colored in blue to high colored in red.

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

4.2. Results

The performance of the adapt algorithms used in this method is shown in Figure 4.4. The adapt performance of the knapsack algorithm is poor, due to its re-computation of the entire cut every frame. While the performance of the knapsack algorithm is roughly comparable when using a small number of nodes, it begins to lag considerably when more nodes are added as the adapt process requires more and more iterations to complete.

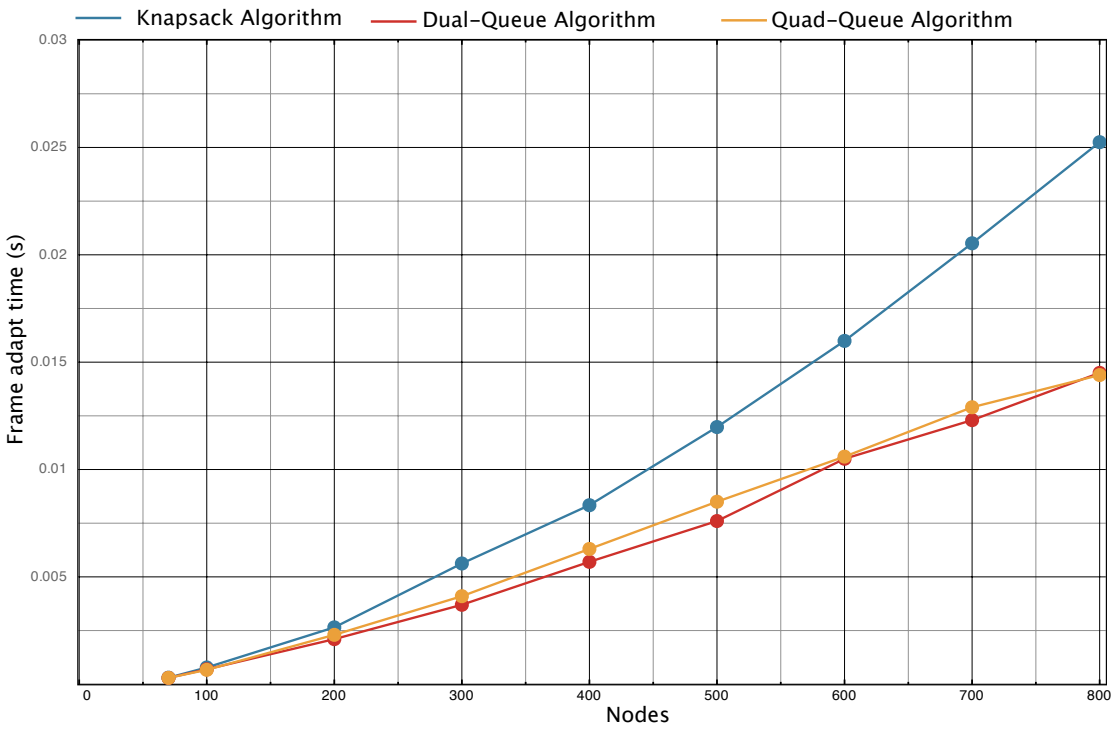

Figure 4.4: Algorithm adapt times. The time to adapt a single frame using the discussed adapt algorithms. As expected, the knapsack algorithm performs the worst because of its lack of frame-to-frame coherence. The dual and quad-queue methods adapt to error and triangle thresholds respectively, and utilize the cut from the previous frames. As a result, they are much faster, particularly when using a large number of nodes.

The dual-queue algorithm timings shown in Figure 4.4 show the benefit of the temporal coherence offered in this method. Because the dual-queue algorithm starts with the state of the previous frame as its input it can adapt much more quickly than the knapsack algorithm by only performing an update of the cut, rather then a complete recalculation. This not only saves time when iterating through the various queues, but also helps to avoid the updating of all error values for a given node. The quad-queue algorithm, which

adapts the scene to a user-specified triangle budget, is the most complex of the adapt algorithms. By using temporal coherence the quad-queue algorithm operates at rates very similar to the dual-queue method.

One of the key features of our hierarchy is the ability to render a model with a variable number of nodes, allowing our method to control the granularity of the adaptation.

Figure 4.5 demonstrates the use of increasing the node budget. The increasing granularity of the adapt process allows it to reduce the maximum screenspace error. This in turn permits the model to be rendered in a more adaptive manner, with areas refined and coarsened as necessary. This ability is especially useful when using the triangle budget mode, as it allows the dataset to be rendered at lower geometric error without increasing the triangle budget. The decrease in error, however, comes at a cost. By increasing the number of active nodes the total adapt time increases, requiring more time to be spent adapting the cut.

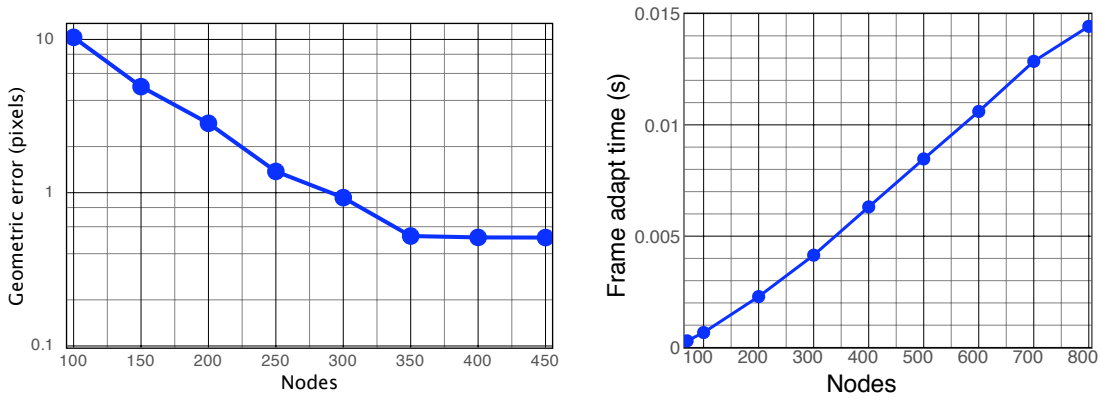

FIgure 4.5: Maximum error over path. The left graph shows the benefit of using a larger number of nodes in a fixed triangle budget scene. By adding more nodes we allow the error to be more evenly distributed in the scene, reducing the error without increasing the triangle count. The right plot shows the cost of using a large number of nodes: increasing the node budget also increases the adapt time, slowing down the adapt process.

Depending on the adapt mode used the increase in node budget can also improve rendering performance. When rendering to a fixed error threshold adding more nodes will decrease the total number of triangles in the scene, allowing for a higher framerate when rendering the same number of triangles per second as shown in Figure 4.6.

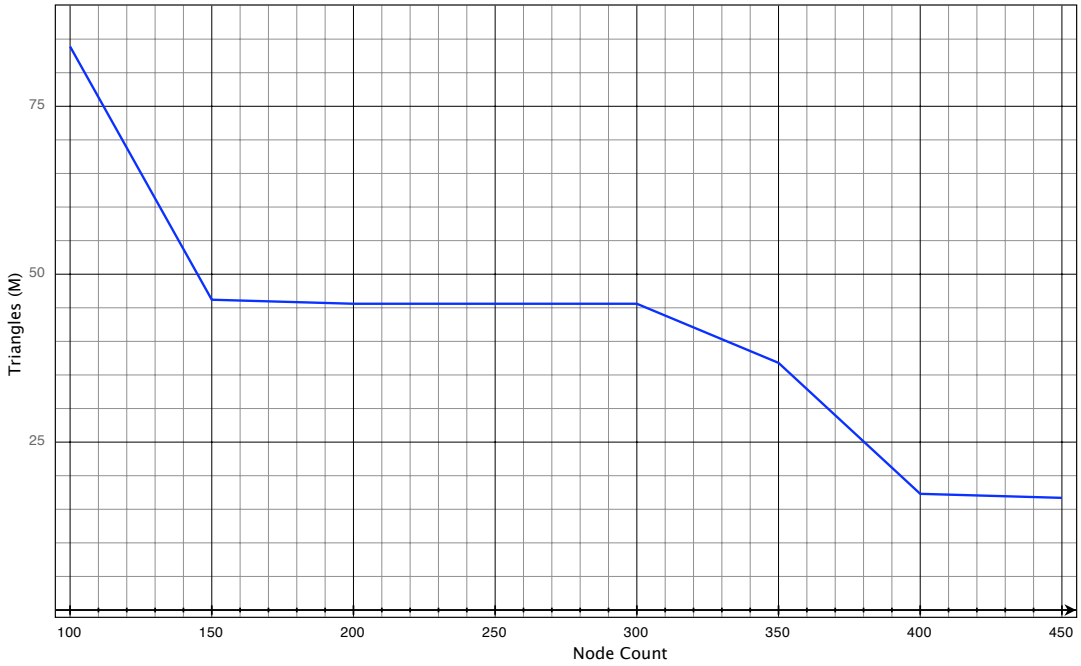

FIgure 4.6: Node to triangle ratio: By increasing the number of nodes used to render the model the total number of triangles decreases. This decrease in triangle count occurs without increasing the geometric error, and therefore increases the overall framerate.

The ability to select the resolution of each node and to control the number of nodes gives our method another unique ability: the ability to independently control the workload on the CPU and the GPU. As shown in Figure 4.7, our system can use fewer nodes to render a scene, which in turn lowers the amount of time spent adapting the scene by the CPU. We were able to simulate this by artificially reducing the performance of the CPU through placing the system in a power-saving mode, reducing the core frequency of the CPUs. By reducing the number of nodes used to render the scene we can compensate for the reduced CPU performance, resulting in roughly equal adapt time for the path.

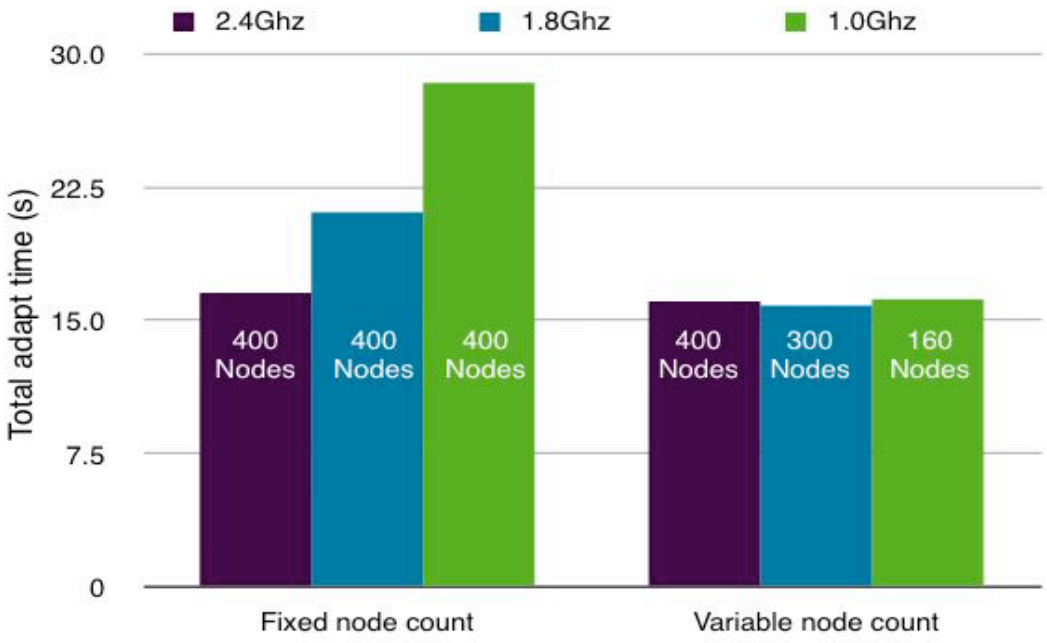

Figure 4.7: CPU workload control. By adjusting the number of nodes used to adapt the scene our method is capable of performing the adapt process at a constant rate when different CPUs are used. Note that the triangle budget and rendering rate remain unchanged.

To compare the performance of our hierarchy against previous methods we have simulated the adaptation limitations of the previous systems in our method. We did this by forcing the refine/coarsen algorithms to split/merge the nodes, thereby simulating the fixed-resolution models. This causes the adapt algorithm to only traverse the hierarchy tree, without allowing any refinement/coarsening of the nodes themselves, making it equivalent to the previous methods. The results from this comparison are shown in Figure 4.8.

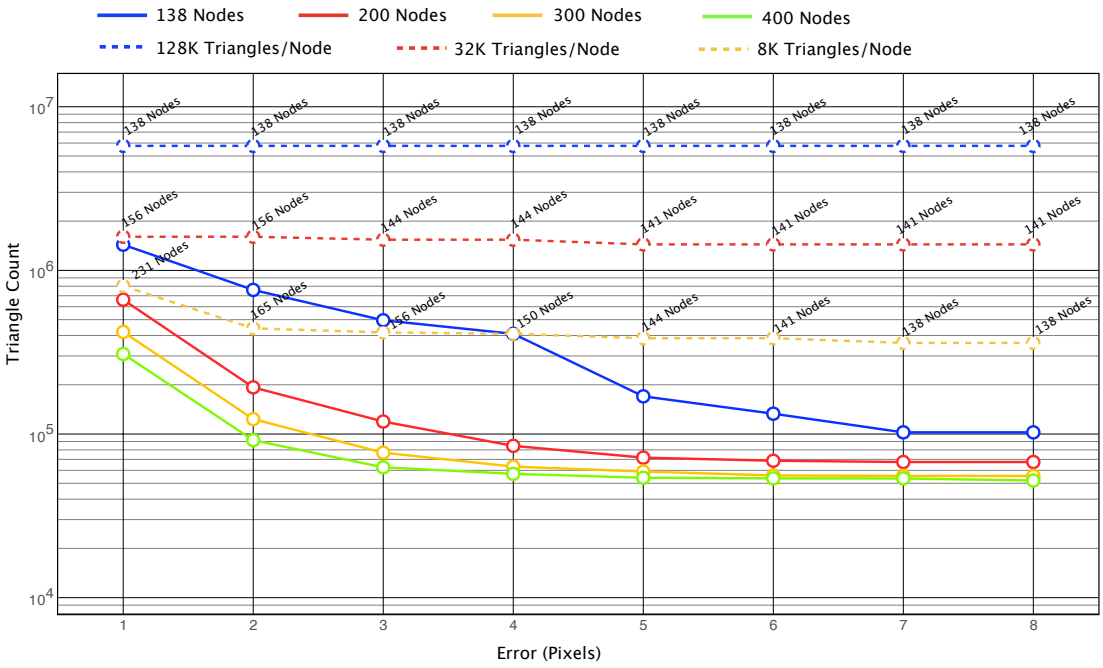

Figure 4.8: Adapt performance comparison. The adaptation performance between our method, in solid, and previous methods drawn using broken lines is shown in this graph. The ability to adjust the resolution of individual nodes allows our method to adapt with a lower triangle to error ratio by using more triangles only where needed.

As Figure 4.8 shows the method presented in this paper produces a better adaptation of the model when compared to the previous systems. Our system can achieve a lower triangle count for a given error threshold then the previous system by adaptively selecting the resolution of each node. Thus, by using more nodes the triangle count can be further reduced through a more tightly bound adapt process. Previous methods with fixed node resolutions are generally unable to reduce the triangle count by increasing the node count. In these systems the geometric error is tied directly to the number of nodes in the scene. This forces the adapt algorithms to use a large number of nodes to refine the model

to the desired error threshold, without allowing individual nodes to coarsen and reduce the triangle count.

The benefit of the multi-resolution hierarchy is shown in Figure 4.9. Where the previous methods utilize roughly constant triangle counts for each of the children nodes, our method allows each child to select its resolution independently, resulting in increased quality using fewer triangles. Thus when a geometry node is subdivided each child node can be adapted independently as shown in the right image of Figure 4.9, allowing our method to use low-resolution geometry where appropriate. The left image of the figure shows the previous methods, restricted to a single resolution per node. When the nodes are subdivided even empty nodes, such as water, are refined forcing a greater triangle usage then using our method.

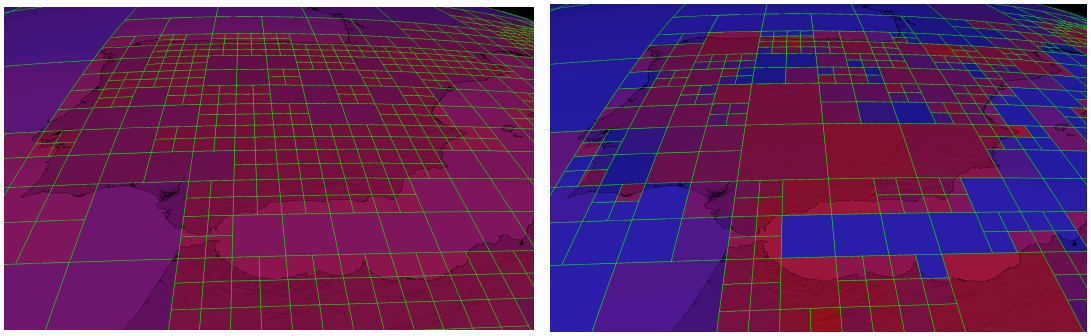

Figure 4.9: Comparing fixed resolution and multi-grain, multi-resolution adapt performance. This figured demonstrates the ability of our system to adaptively reduce resolution of nodes as necessary, allowing triangles to be allocated where needed instead of being distributed uniformly. Both methods use the same number of nodes and triangles. Tessellations range from sparse in blue to dense in red.

4.3. Discussion

The screen-space adapt algorithms used in our method are modified versions of algorithms used in the most recent work in the LOD area. While these algorithms have many advantages, namely the ability to adapt to a specific budget in a relatively accurate fashion, they also have several disadvantages.

The main disadvantage of the queue based methods is the non-exhaustive search for the best node layout and detail level. Because the nodes are selected using a queue based on local metrics it is possible to arrive at a local minimum, which is not guaranteed to be the global minimum. As a result, it is possible that with more work a better set of nodes could be selected that would give a tighter bound on the error in the scene. This is especially evident when operating the split/merge section of the algorithm. It is very possible that a single split of a node will yield very little improvement in the number of triangles required due to the location of the detail in the node. If the same node were to be subdivided again, however, it is possible that the savings would be quite substantial. Unfortunately, because we cannot do an exhaustive search we will not try to perform the first split since the benefit of that operation is negligible. Although this information could be incorporated into the split/merge cost metrics, it would bias the subdivision method, potentially wasting a number of nodes which could be used more effectively in a different location.

Another disadvantage of the queue-based algorithms is their reliance on accurate metrics for the stability of the scene. If the metrics are inaccurate, or fluctuate, the result-

ing scene will likewise flicker as nodes are refined and coarsened or split and merged in consecutive frames. This is exacerbated by the 2D adaptive process which allows multiple dimensions of adapting the model, where a tiny change in screenspace error can cause a drastic change in the hierarchy cut. When more constrained tree methods are used, such as the tree traversal methods presented in [Ganovelli 2004] or [Borgeat 2005] the LOD selection becomes much simpler, and involves traversing the tree until each branch is under the error threshold. While this approach will create the same cut every time given a error threshold, it has the unfortunate downside of requiring an large number of nodes to perform the adaptation. The downside of using the simpler methods is their limitation to a single dimension of adaptation. Our method strives to improve the flexibility of adaptation to a scene by allowing both splitting/merging as well as refining/coarsening of nodes, which would be impossible when doing a simple tree traversal.

A similar problem occurs in the triangle budget mode, which is more involved since it has to balance four queues rather then two. As a result, the final model is again more unstable, with nodes possibly changing LOD level between consecutive frames. This is minimized by using thresholds below which splits and merges are not performed. These anti-hysteresis features help to overcome the inherent instability of adapt mechanisms, and also lower the adapt time.

4.4. Summary

The algorithms described in this paper offer a new way to control the adaptation of the quad-tree based hierarchy, and similar hierarchies in which multiple elements in-

fluence the final selection. The main improvement over previous methods is the incorporation of multiple parameters into the adapt algorithm. In particular, the quad-queue algorithm described in this chapter is a new hierarchy adaptation algorithm capable of balancing two variables when creating the cut. As a results our quad-queue algorithm is capable of adapting both the node layout of the cut and the resolution of each of the nodes simultaneously.

The new adapt algorithms also retain the advantages of the previous algorithms, namely the temporal coherency present in the dual-queue algorithm. This allows our new algorithm to operate at rates similar to the dual-queue algorithm, without the performance penalties of the knapsack algorithm.

5.Visualization

The leaf nodes of the trimmed tree created during the adapt process describe the geometry elements that will form the rendered model. The visualization stage is responsible for converting this geometric data into the final output seen on the computer screen. During this process the raw geometric data stored in the geometry images is converted into a 3D mesh, and rendered to the screen. The rendering stage also performs additional processing on the geometry, including the merging of borders between neighboring nodes.

5.1. Rendering

The rendering approach used in this method differs from the traditional triangle rendering method used in previous LOD systems. The traditional method of rendering 3D meshes is to send triples of vertices which define a triangle in 3D space. The coordinates sent are then projected onto the screen, and the resulting 2D triangle is rasterized on the display.

The rendering method presented in this paper relies on the ability of the GPU to obtain the 3D coordinates inside the GPU. To render the actual model we send in triples of vertices which represent a 2D planar triangle. Using a vertex program on the GPU we sample the actual 3D coordinates from the geometry image, which replace the input coordinates. We then project the resulting 3D triangle onto the 2D screen, and rasterize the final model.

5.1.1. Geometry Image 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. Because the input geometry is replaced in the vertex program we are free to use the texture coordinates as the 2D vertex coordinates for the grid, which reduces the amount of data required. As a result we only need to send 8 bytes per vertex into the GPU, since all of the other data is sampled directly in the vertex and fragment programs.

To further accelerate the rendering of these planes we use several optimizations which result from the grid-like layout of the vertices.

The most important of the optimizations performed on the *u,v* grid is the organization of mesh rendering into triangle strips of adjacent columns shown in Figure 5.1. 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, instead of processing the same vertex six times on average. The ability to maximize the number of cache hits is very important to this method, especially when combined with the performance hit of the geometry image rendering. If a cache hit occurs, the current vertex does not need to be recomputed, since the results are already known by the GPU. As a result this vertex is processed for free, improving the rendering throughput.

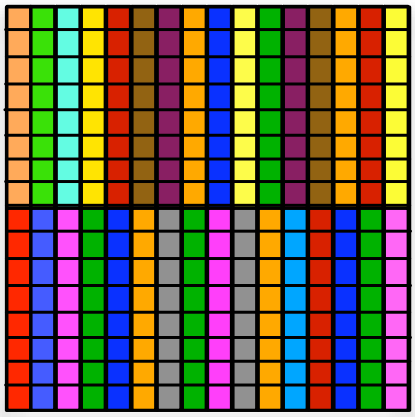

Figure 5.1: Strip-based rendering. The 2D u,v plane is broken down into a series of strips in order to maximize GPU cache hits. This allows the expensive vertex fetch instructions to be reused. The strips also reduce the amount of memory required to store the index lists for the geometry.

Other optimizations include storing the minimal amount of data possible, the *x,y* coordinates. This results in less data going through the rendering pipeline, and therefore less data to process and cache, resulting in more cache hits. The vertex data can be stored in any format, including 1, 2 and 4-byte representations. Our system uses the 4-byte float representation due to its direct mapping with the GPU internal data format, reducing processing overhead of the CPU to GPU data copy.

The generated mesh is stored on the GPU using a Vertex Buffer Object (VBO) which is currently the fastest way of rendering static data. By storing all of the data in video memory, including the indices, we can minimize the amount of data being sent to the GPU, and free up the bandwidth for texture uploads.

5.1.2. Mesh-Based Rendering

The vertex texture renderer relies on very recent changes to the GPU architecture. As a result, it depends on parts of the GPU that have not been thoroughly optimized yet. As a result we have decided to also implement a more traditional renderer, one that uses the triangular mesh data representation. The renderer uses *Vertex Buffer Objects* (VBO) to render the triangular mesh, currently the fastest path through the rendering pipeline.

The main difference between the two rendering methods is the addressing of the data on the GPU. In the vertex texture renderer the data is stored as a 2D array, allowing the GPU to access any part of the data at runtime. The mesh-based method stores the data as a 1D stream, with each vertex stored independently of its neighbors. While this improves performance, it limits the amount of work that can be offloaded to the GPU.

5.1.3. GPU Processing

A large amount of the work performed in this method is done on the graphics processing unit. This is possible due to the ever-increasing capabilities of modern GPUs, which allow the vertex and fragment units to be programmed. The programability of GPU hardware can be accessed in two main ways: either by using assembly-style code, or by using a high-level programming language which compiles down to the assembly code.

In this method we have used the Cg programming language developed by NVidia to write the GPU programs. The Cg framework allows the code to be written in a C-style syntax which is subsequently compiled into a assembly representation compatible with

the graphics hardware. The advantage of this approach is the use of much cleaner and more understandable code, as well as the optimizing compiler. Because the Cg framework is created by NVidia, who also manufacture the GPU used in this method, the compiler can take advantage of special features such as instruction scheduling. The compiler can for example move instructions around in order to reduce the use of conditionals or to minimize the latency of time consuming operations, such as vertex texture lookups.

The actual GPU processing falls into two categories: vertex and fragment processing. The vertex processing determines the geometric information of the output model by computing the location of the vertices on the screen. The fragment program performs additional shading computations on a per-pixel level, allowing for additional data to be incorporated into the final on-screen rendering of the model.

5.1.4. Vertex Processing

The vertex processing relies on the input vertex data passed into the program from the GL layer. Additional data can also be sent to the processor as long as it is invariant for every vertex in the form of constant values. The input data is either a *u,v* for the vertex texture renderer or the *x,y,z* for the mesh-based renderer, along with several constants used for border merging and reusable texture mapping.

The most basic operation of the vertex program is the retrieval of geometric coordinates from the geometry image, shown in Figures 5.2 and 5.3. Upon 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. The vertex program then performs the necessary

transformations, and sends the results down the graphics pipeline to the fragment processor.

```
INPUTS: vec2 inCoord, texture gImage;
vec3 outCoord = tex2DSample(gImage, inCoord);
outCoord = outCoord*projMatrix;
return outCoord;
```

Figure 5.2: This code sample retrieves the vertex coordinate from the geometry image and transforms it into screen coordinates. The results are then returned to the fragment unit.

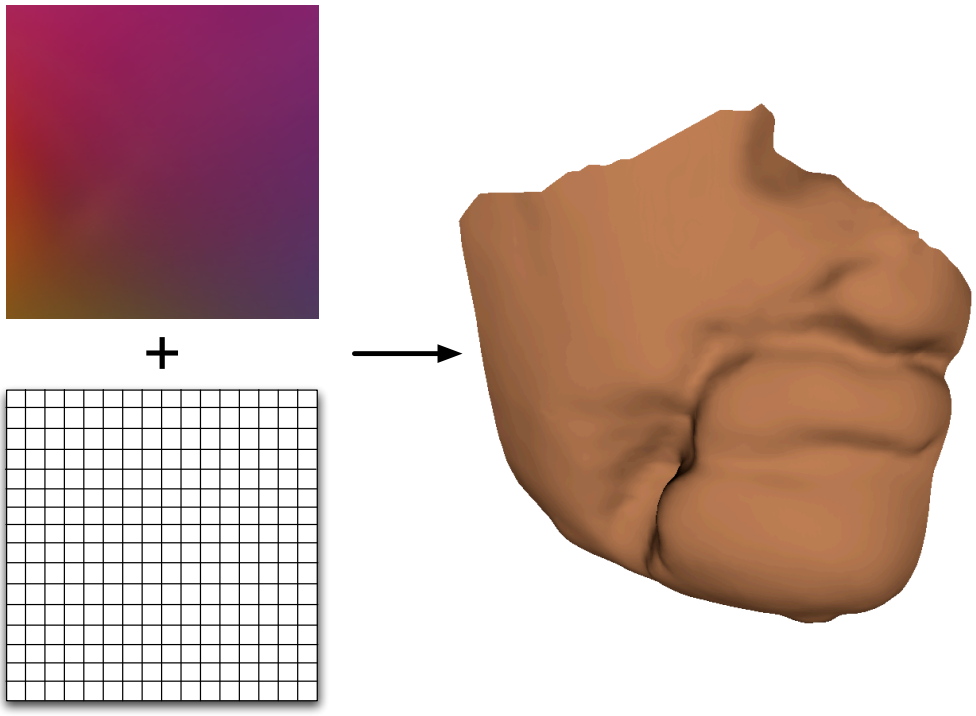

Figure 5.3: Sampling geometry images. The geometry image in the upper left corner is sampled using the vertices of the uniform grid in the lower left corner. The new 3D coordinates combined with connectivity from the grid create a surface shown on the left

When height map geometry images are used the vertex program can be used to perform additional transformation operations, turning a 2.5D heightmap into a 3D mesh. These operations are again based on the input *u,v* coordinates, and additional parameters passed into the vertex program such as the diameter of the resulting sphere.

The calculation is based on the location of the node in the model, which requires global *u,v* coordinates, which extend from the origin to the full size of the model, allowing each vertex to know its exact location in the height map. These global *u,v* coordinates are obtained through adding a node offset to the local *u,v* coordinates that are passed in to the vertex program. The resulting mesh ultimately has several sets of *u,v* coordinates, al-

lowing each vertex to know its position in reference to both its node and the global geometry.

The global *u,v* coordinates are subsequently used as the angles for the spherification process. By multiplying the texture coordinates by appropriate values the *u,v* coordinates are converted into angles α and β, used to transform the 2D uniform grid into a 3D spherical model.

The final stage in the spherification process is the application of features from the height-map to the spherical representation. Since the height-map features are always vertical, they can be applied as features that push the surface of the sphere out along the normal vector of the corresponding point on the sphere using the vertex program shown in Figure 5.4.

The resulting model is converted from a plane with height values into a spherical model with the features from the height map.

```
INPUTS: vec2 inCoord, texture gImage, vec2 globalOff-
set;
float alpha = inCoord.x* globalOffset.x;
float beta = inCoord.y* globalOffset.y;
vec3 sphericalCoord;
sphericalCoord.x = sin(alpha)*sin(beta);
sphericalCoord.y = cos(alpha)*sin(beta);
sphericalCoord.z = cos(beta);
float displacement = tex2DSample(gImage, inCoord);
vec3 outCoord = sphericalCoord*displacement;
outCoord = outCoord*projMatrix;
return outCoord;
```

Figure 5.4: This version of the vertex program performs spherical mapping of the height map data. It receives the scale multipliers from the main program, and subsequently uses them to convert the planar mesh into a sphere. The height-map is then sampled, and the resulting height is added into the spherified coordinate.

The regular grid layout of the processed data greatly simplifies these computational operations in the vertex program by allowing access to any part of the geometry data, allowing more complex operations to be performed, such as parametric deformations or animations. Figure 5.5 shows the result of one such mesh operation, allowing a planar height-map to be converted into a spherical representation. The benefit of performing this spherification on the GPU is the reduction of storage requirements for the data.

By keeping the data on the CPU and GPU as a single height value the storage requirements are reduced by a factor of 3, allowing more data to be stored in the limited GPU memory.

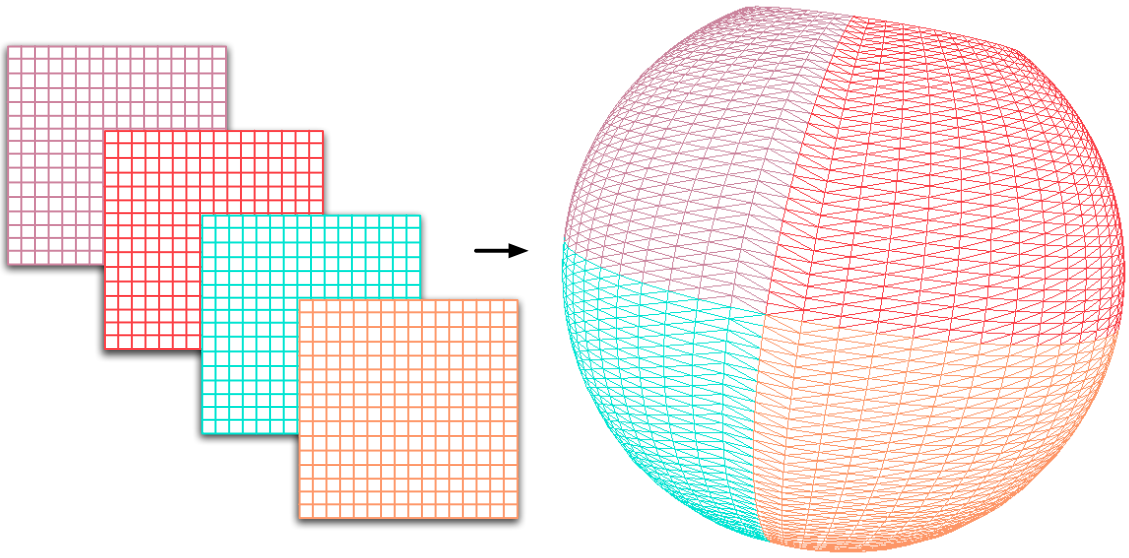

FIgure 5.5: Converting a planar mesh into a spherical representation. The vertex program can be used to take in the vertices from the 2D grids, and transform them into a spherical version of the model. This is useful for heightmaps, which can be converted into a 3D mesh representing the curvature of the Earth.

5.1.5. Fragment Processing

In the fragment processor the color and normal maps can be applied to further enhance the visual quality of the resulting image. The color and normal values for a given pixel are obtained from their respective texture maps through texturing. The normals are

then used to shade the pixel based on the color value and the lighting parameters obtained from OpenGL state.

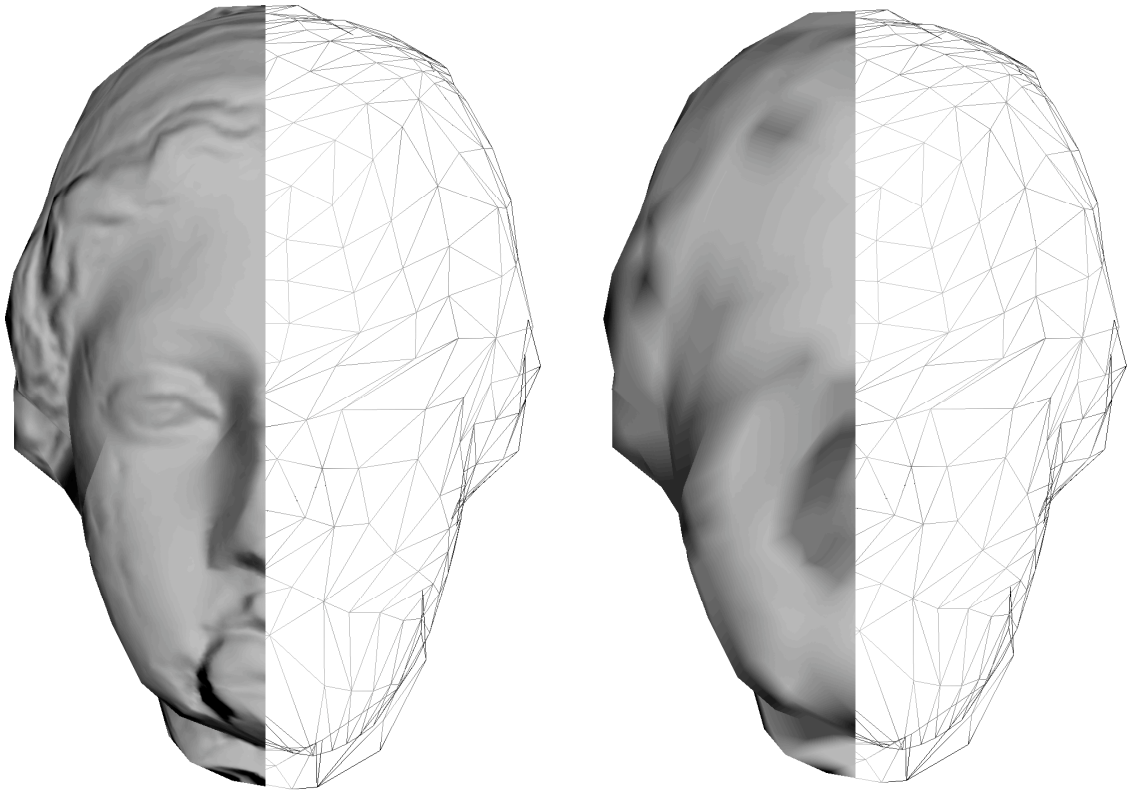

Figure 5.6: Comparison of per-fragment and per-vertex shading. The image on the left shows a per-fragment shaded model, which utilizes a low-resolution mesh and a higher resolution normal map to increase the visual fidelity of the final rendering. On the right the same resolution geometry is used with a per-vertex resolution normal map, resulting in a far lower quality output

The normal mapping process allows a low-resolution mesh to be rendered with much higher resolution normals, since the normal data resolution is no longer tied to the number of vertices used to render the model, as shown in Figure 5.6.

During the fragment processing step we can also perform other operations, such as the coloring of height data using a 1D texture map, or thresholding values in order to fill areas below sea level with a blue color representing water. We can also discard fragments which originate from missing data in order to avoid degenerate triangles.

The fragment processor can also perform additional operations when converting planer height map into a spherical mesh. Because the normals stored in the texture maps were created for a 2.5D height map, the spherical model will have incorrect normals if the original normal map was to be mapped onto a spherical object. The normals are therefore converted to a spherical version in the fragment program as necessary. This process is performed by taking the original normal from the normal map and rotating it around the origin by the spherical projection at the current fragment.

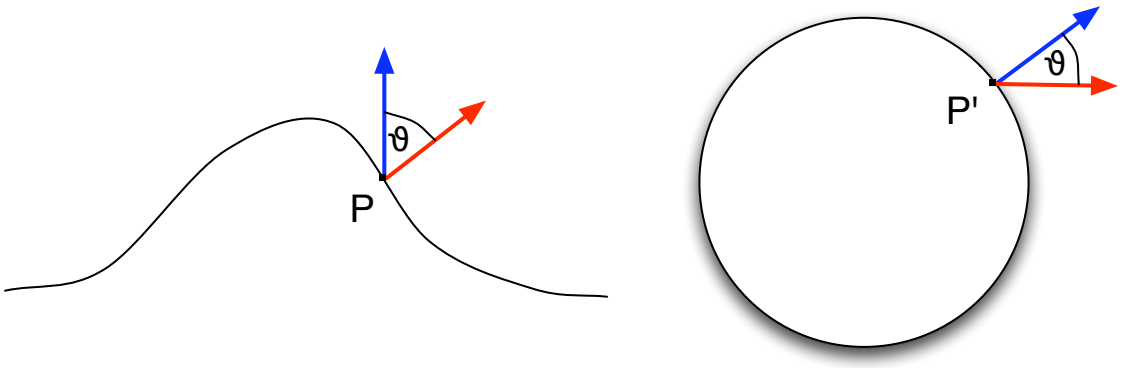

Figure 5.7: Rotating normals. The normal is converted from a planar normal to a spherical normal by calculating the angle ϑ, which is the angle of deviation from the vertical axis. This angle is then used to rotate the normal against the axis of the sphere at the point of the normal as shown on the right.

As shown in Figure 5.7, the first step in spherifying the normal is calculating the angle ϑ between the normal drawn in red at the point P and the vertical axis drawn in blue. This angle is the angle of deviation between the normal of the base surface, and the normal at the point. By taking this angle, and rotating the normal of the sphere at the point *P'* we obtain the new normal, *N'*. This new normal is correct for the spherical representation of the model, allowing it to be shaded properly.

5.1.6. Pipelining

The rendering of the geometry can be separated from the adapt process, allowing the rendering thread to render the previously adapted cut while the adapt thread is adapting the next frame as shown in Figure 5.8.

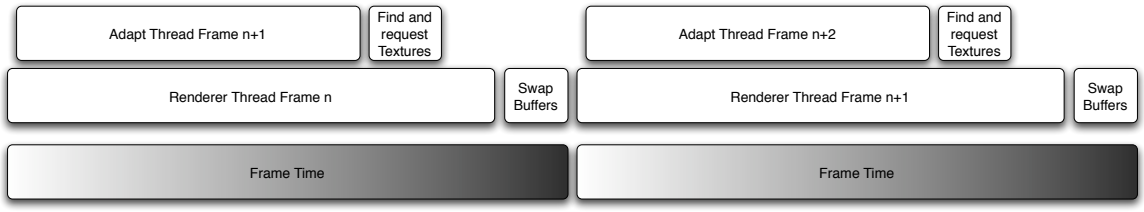

Figure 5.8: Pipelining Diagram: The thread pipelining method used in our system is shown in this diagram. By performing the adapt for the next frame while rendering the current frame our method can reduce the total frame time by removing the adapt time from the sum.

At the start of frame *n*, the pipelining begins by saving the hierarchy cut from frame *n-1*, which represents the cut from the previous frame. The adapt thread then launches the adapt process with the camera parameters for frame *n*, creating the cut for frame *n*. While the adapt thread is adapting the hierarchy for the new camera parameters, the rendering thread begins rendering frame *n-1*.

At the end of the frame the rendering thread will have finished rendering the cut from frame *n-1*, and the cut for frame *n* will be ready to render, allowing the renderer to always stay a frame behind the adapt thread. The adapt thread can also perform additional operations while adapting the geometry, such as finding and requesting the best textures for the geometry appearing in the next frame. This gives the loader threads additional time to load the requested data before the next frame begins rendering, reducing the wait time for textures to load.

The main advantage of such a layout is the reduction in the total time required to adapt and render a frame since the adapt time is hidden by the rendering time, as shown in Figure 5.8. The render and adapt times added together take more time then the actual time used to render the frame, showing the benefit of the pipelined adapt method.

To test the effectiveness of the pipelining method we compared the measured frame time to the sum of the rendering time and the adapt time, shown in Figure 5.9. Because the measured frame time is equal to the render time, and lower then the sum of the render and adapt time, we can see that the pipelining is functioning correctly. The adapt process does indeed occur in the background, allowing the renderer to take up the entirety of the frame time.

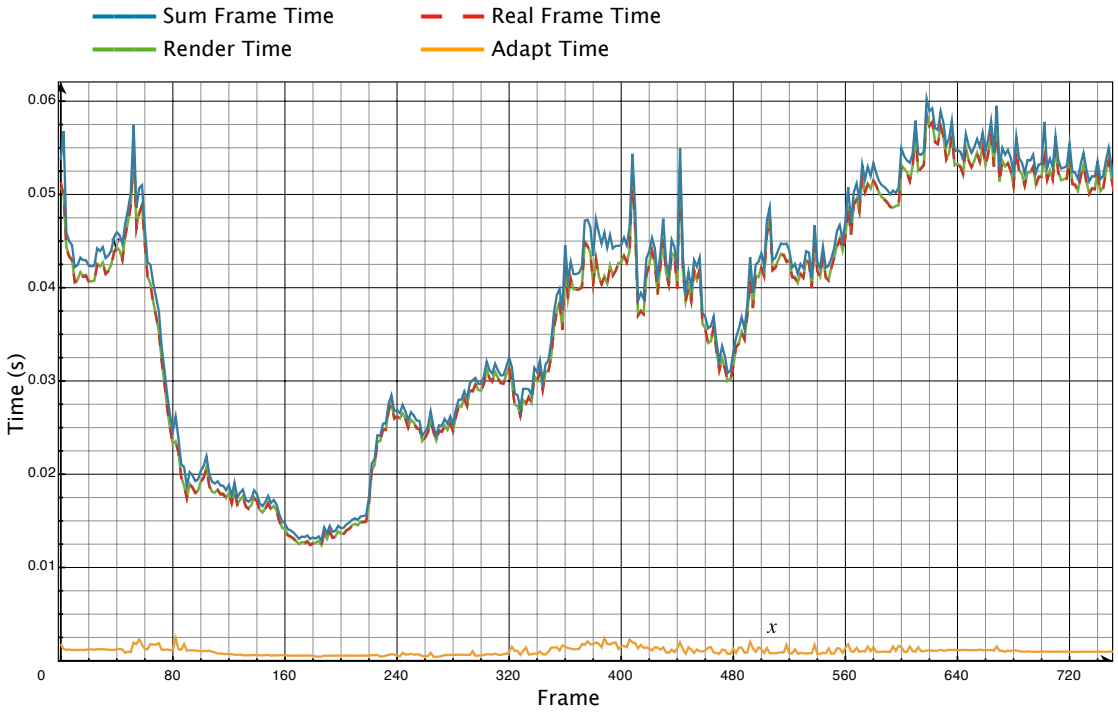

Figure 5.9: Pipelining Results: This graph shows the effectiveness of the pipelined design. As shown in the graph, the sum of the render and adapt time is greater then the actual frame measured by our system. This shows that the pipelining is working, allowing the adapt process to be performed in the background. Times measured over a 750 frame path of the Thai Statue adapting to a 1 pixel error threshold.

5.2. Border Merging

5.2.1. Geometry Image Border Merging

The use of uniformly-tessellated 2D planes allows us to reconstruct the original model without cracks even in the presence of neighbors of different LOD and hierarchy

levels. By adaptively collapsing triangles on node borders we can force them to seamlessly match their lower resolution neighbors.

The first step in this process is performed during the preprocessing stage, where we duplicate one row and column from a geometry image to its bottom and right neighbors. This forces every node to share an identical border with its immediate neighbor when the nodes are at the same resolution.

At runtime we begin by determining if the current vertex is located on one of the borders of the current node by checking the texture coordinate of the current vertex. If either u or v is zero or one, the current vertex is on a border and requires further processing. We then obtain the tessellation of the corresponding neighbor node and use it to decide where the current vertex should be moved. If the tessellation of the neighbor node is greater then or equal to the tessellation of the current node no action is performed as the other node will adapt to the lower tessellation. If the tessellation of the current node is greater then its neighbors the current vertex has to be relocated in order to avoid cracks. This is performed using a multiply and a mod operation, which ensures that the current texture coordinate corresponds to a texture coordinate of the neighbor node, so that the sampled geometry will be identical on the border as shown in Figure 5.10.

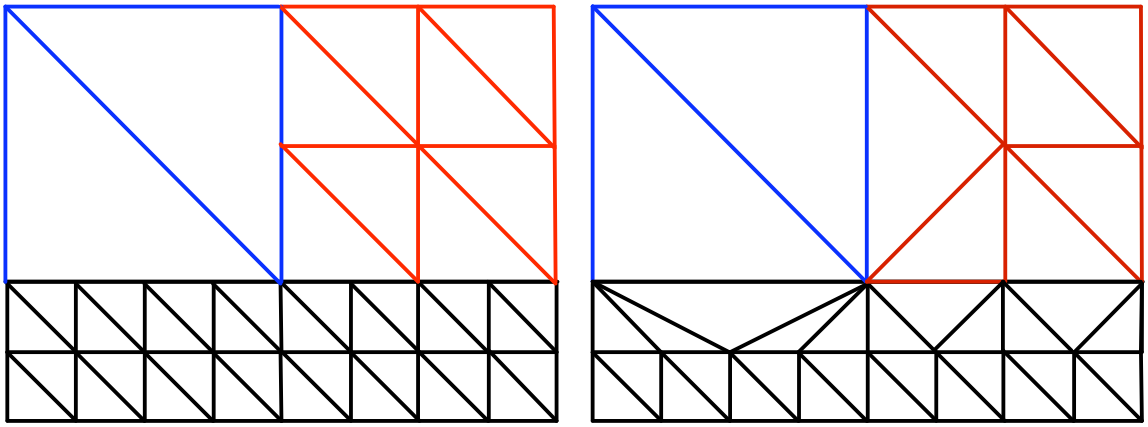

Figure 5.10: The border merging process. The image on the left shows three nodes, each of different resolution, sharing a border. In order to merge the borders the higher resolution node needs to coarsen its border to match its neighbors, as shown on the right.

Traditionally this process would be performed on the CPU due to lack of random access to the geometry during GPU processing. This forces additional data to be created and uploaded to the GPU per frame, introducing additional CPU and GPU overhead. However, since the geometry data used in our method is stored as textures, the vertex program can access any element of it based on the texture coordinates used. Thus we are able to perform the border matching on the GPU by simply modifying the u,v coordinates of border vertices, utilizing the SIMD capabilities of the GPU and reducing the CPU overhead of the process.

We see a more complex situation in Figure 5.11. 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 resolu-

tion 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 send an array of neighbor node resolutions for each of the four borders to the GPU when rendering the current node. Because each border can have different resolution values in the array, we determine the correct array element to sample using a multiply and mod operation. This allows us to have multiple neighbor nodes per border, which allows our method to be more flexible. Because this data is constant per patch, it does not need to be re-sent for every vertex. Thus even though our geometry is still very compact, we can match borders flexibly and reliably. The overhead of the border matching process is reasonable, currently requiring 18 ARB vertex program instructions to perform.

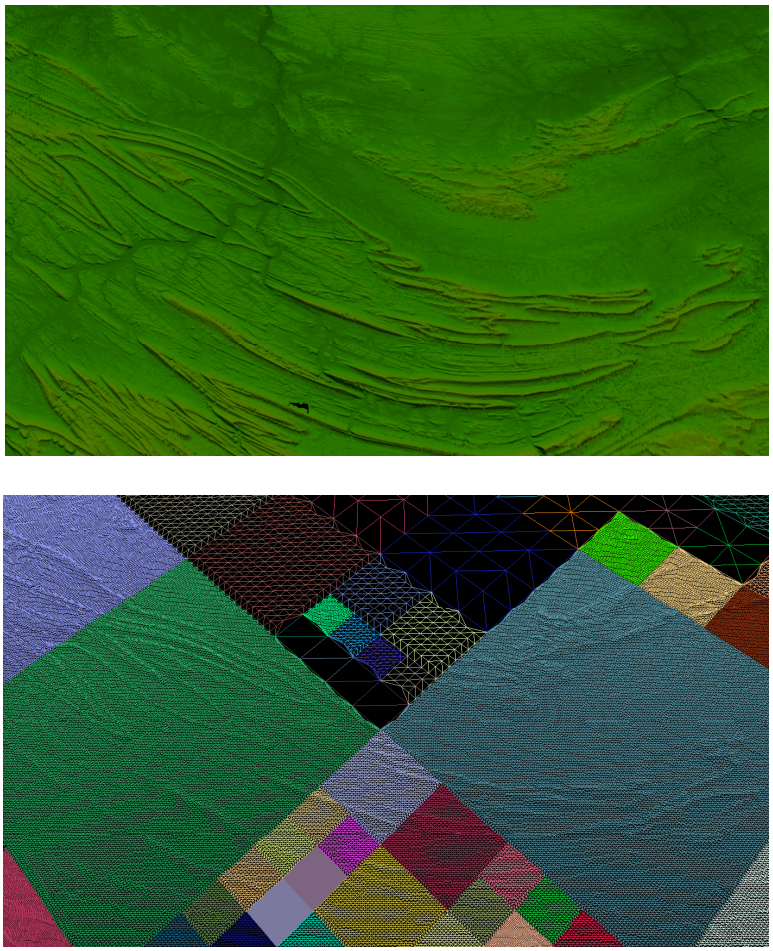

Figure 5.11: Complicated neighbor layout. This figure demonstrates a more complicated neighbor layout that occurs when rendering regular datasets. Because some of the areas are flat, the hierarchy is adapted such that flat areas are rendered using few triangles, while other areas are more densely tessellated.

Although node corners may seem challenging at first, they don't require any additional processing to match borders. Since each texture covers a square area it must maintain corners in the same position in order to fully cover its domain. As a result the corners are stationary on each node, and cannot be moved or removed and will always map to the same texture coordinate.

The merging 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 at least that amount of error along its boundary. As a result the maximum error has already been determined by the lower resolution neighbor, and the reduction in resolution of the higher resolution border will not increase the maximum error in the scene.

5.2.2. Mesh-Based Border Merging

The border merging process used in the mesh-based rendering method is relatively similar to the vertex texturing method, as it also reduces the resolution of the node borders to match neighboring nodes. The main difference between the two methods is the use of index lists to perform the border rendering. These index lists tell the GPU which vertices to place in the strip, and they allow a reduced amount of data to be sent to the GPU in order to render primitives. By creating appropriate index lists our system can create borders that match any resolution neighbors, once more creating a seamless model. To reduce the size of index lists created every frame our method separates the interior of the node from the borders, allowing each border to be created and uploaded separately.

The new borders are created for every node leading to significant processing overhead, especially when a large number of nodes is used to render the scene. To reduce this load our method utilizes caching, allowing index lists to be reused as long as the underlying geometry remains constant and the neighbor nodes do not change resolution. To further optimize the caching system our method will replace only the data that has

changed, so for example if the resolution of the interior is unchanged, and only one neighbor node has changed resolutions, our method will reconstruct only the border touching the new neighbor. As shown in Figure 5.12 the caching process greatly decreases the border creation time. Table 5.1 shows that even for larger node counts the caching is very effective at decreasing the amount of time required to construct node borders.

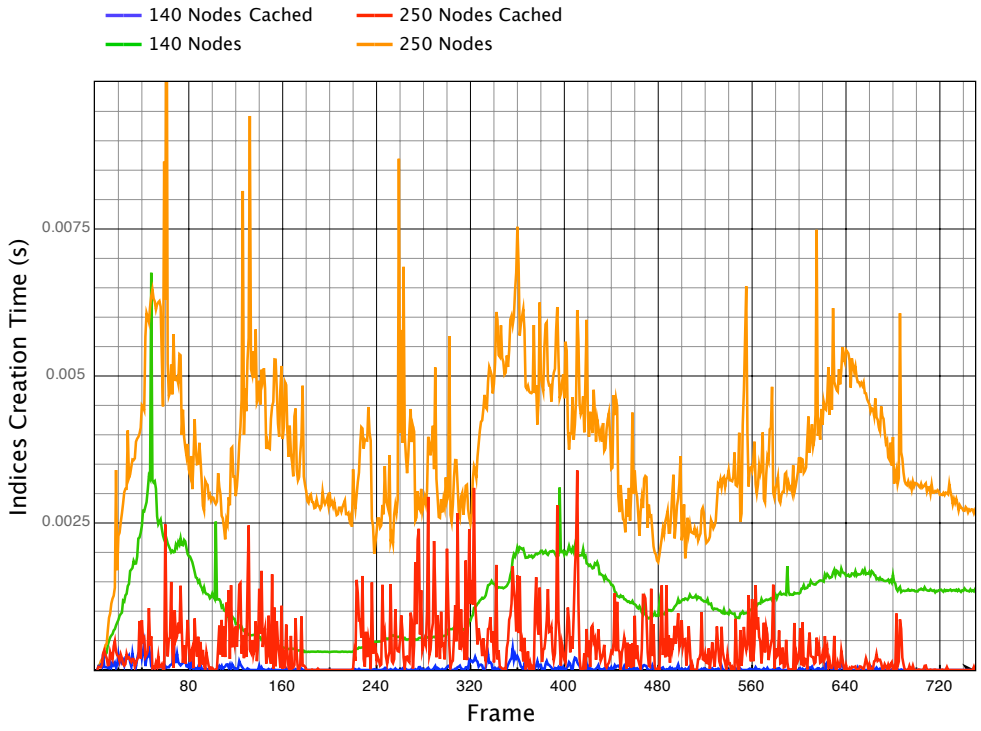

Figure 5.12: Index list creation time: The amount of time required to create border indices is shown in this graph for both the cached and non-cached modes. While the amount of time required for the non-cached mode remains largely steady, the caching method spikes only when coherency is lost and some indices have to be recomputed.

Nodes	Min (ms)	Max (ms)	Avg (ms)	StDev (ms)
140	0.54	6.80	1.2	0.630
140 Cached	0	0.57	0.035	0.064
250	0.57	13.3	3.7	1.2
250 Cached	0	3.4	0.39	0.49
500	1.2	17.5	7	3.2
500 Cached	0	7.3	0.5	0.7

Table 5.1: Cached index lists: The timings for the creation of index lists for the geometry are shown in this table. The timings for the cached version of the index generation show a significant increase in performance over the non-cached version.

Although the majority of the border operations are now performed on the CPU, some of the work can still be handled by the GPU. One of the main tasks still performed on the GPU is the modification of texture coordinates of the vertices when mapping into ancestor or lower-resolution data.

5.3. Results

The first set of results compares the rendering throughputs of the vertex texturing method to the mesh-based based renderer. As shown in Figure 5.13, the throughput of the vertex texturing system is significantly lower then the mesh-based renderer, by as much as a factor of four.

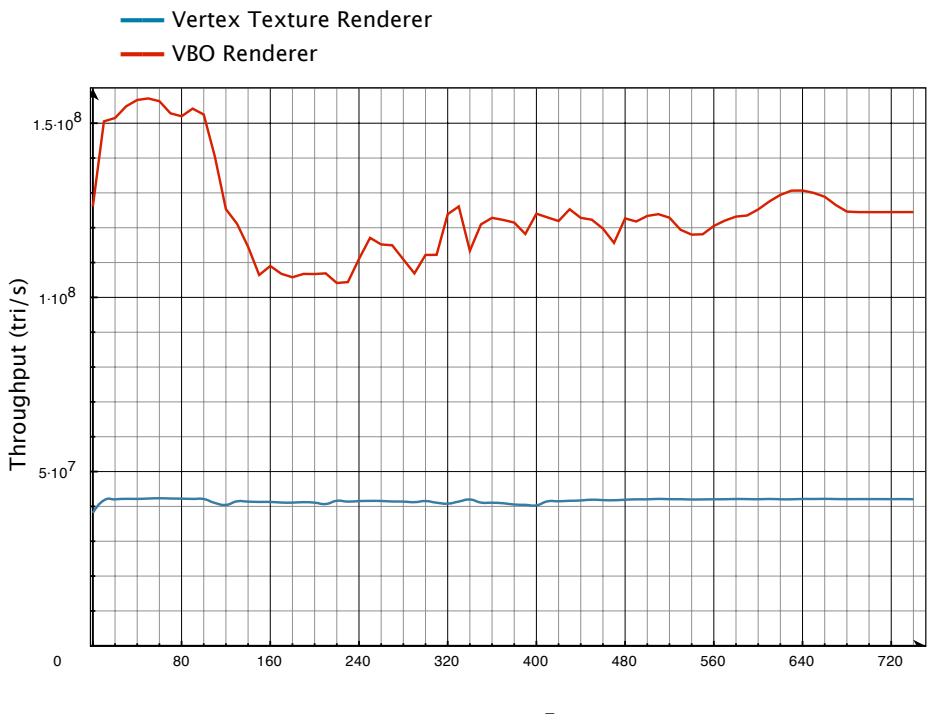

Figure 5.13: Mesh-Based and Vertex Texture Rendering: The mesh-based renderer created in this system outperforms the Vertex Texture renderer by a wide margin. The performance of the mesh-based renderer is up to four times that of the vertex texturing method. The results were recorded over a 750 frame path over the Thai Statue model on a quad-core 2.66Ghz Pentium Xeon system with an NVIDIA 7800GT video card running SUSE 10.2.

We have also compared the performance of our system when using an increasing number of nodes to render the final model. While increasing the number of nodes has an effect on the throughput of the system, that effect is relatively small compared to the benefit obtained from the increase in node count. As Figure 5.14 shows, the rendering throughput for a path over the thai statue model is comparable for all node configurations

tested. While the different rendering rates are not identical, Table 5.2 shows that there is a relatively small difference in the overall performance of the system.

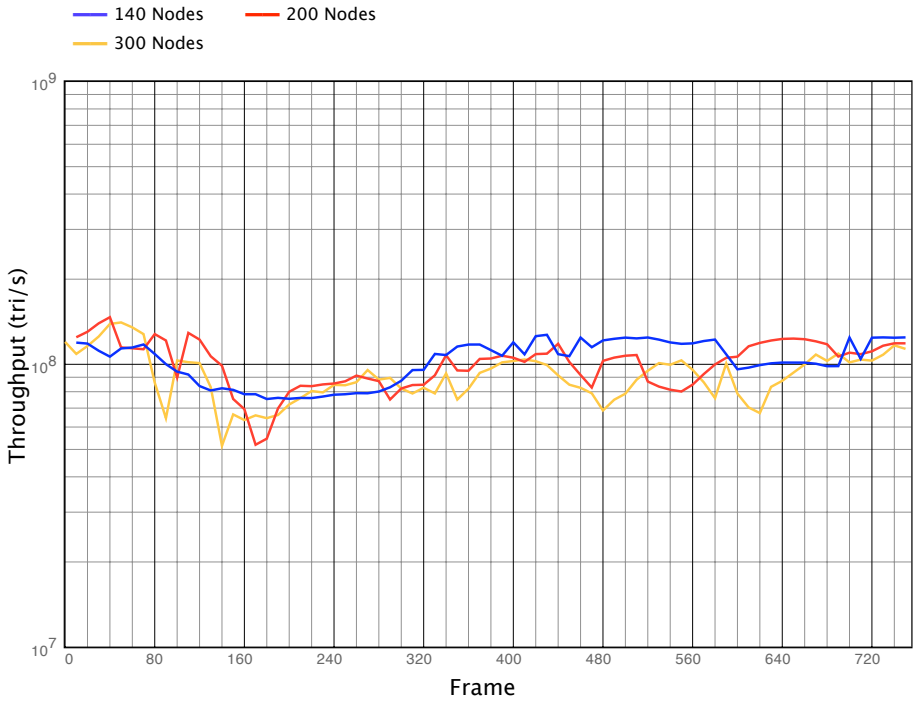

Figure 5.14: Throughput with different node configurations: The overall performance of the system does not decrease substantially when using a large number of nodes.

Node Count	Minimum	Maximum	Average	Standard Deviation
140	100%	100%	100%	0
200	66%	146%	99%	18%
300	57%	131%	89%	17%

Table 5.2: Node throughput: Increasing the number of nodes used to render the model slightly reduces the overall throughput of the system by issuing more small geometry patches. 4 core Mac Pro running SUSE 10.2 with nVidia 7800GT video card.

The stable performance of the system when rendering geometry using a different number of nodes allows our system to render a model at the same geometric error using more nodes. The additional nodes help the system to subdivide the nodes with an uneven error distribution, allowing each child node to adapt independently. As a result, the overall framerate of the system increases as shown in Figure 5.15.

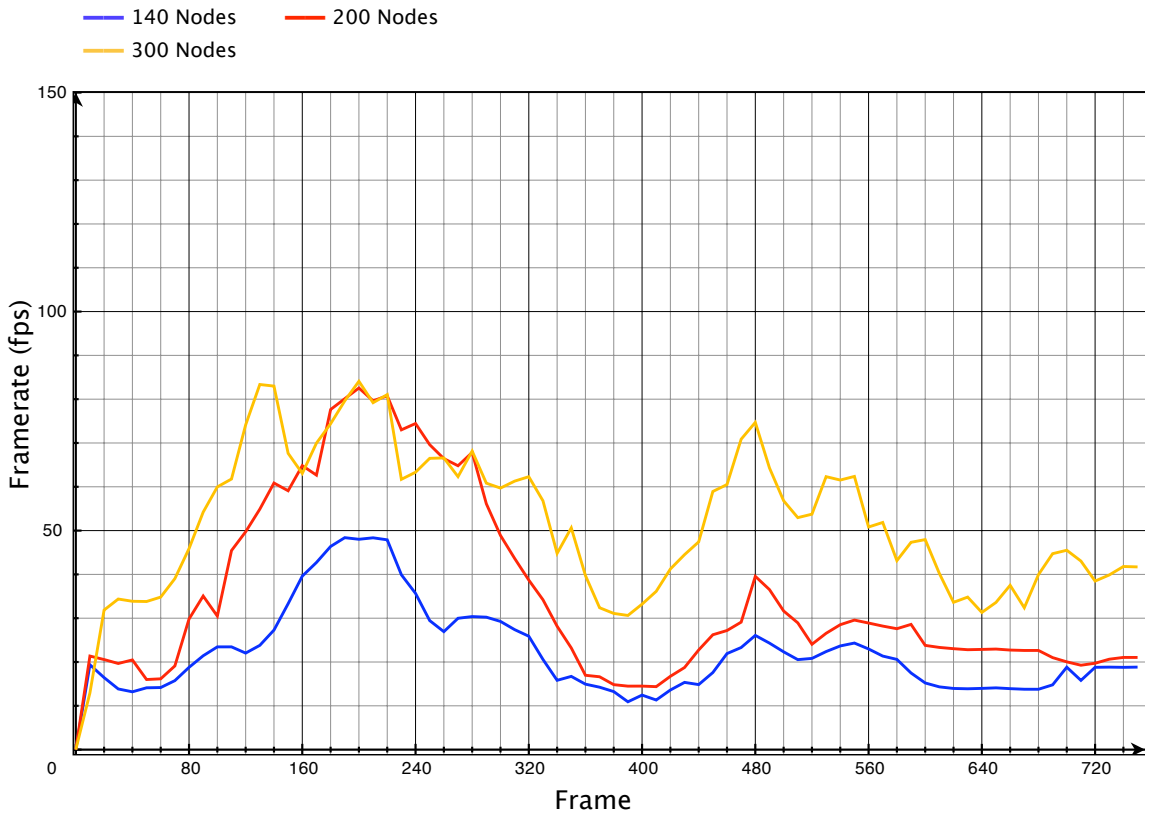

Figure 5.15: Framerate with different node configurations: By using a larger number of nodes our method helps to reduce the total number of triangles in the scene by subdividing nodes and adapting their children independently. Results obtained from a path over the Thai Statue model using a 1 pixel error threshold

Node Count	Minimum	Maximum	Average	Standard Deviation
140	100%	100%	100%	0
200	105%	265%	157%	42%
300	67%	350%	241%	50%

Table 5.3: Node Framerate: Increasing the number of nodes used to render the model greatly increases the overall framerate of the system, even with the increased border creation time. 4 core Mac Pro running SUSE 10.2 with nVidia 7800GT video card.

The ability to control the number of triangles in the scene gives us control over the GPU workload. By reducing the frequency of the GPU processor we can reduce its performance, simulating GPUs of various performance capabilities. To compensate for this we can reduce the maximum number of triangles in the scene, resulting in a stable framerate between different GPU configurations, shown in Figure 5.16.

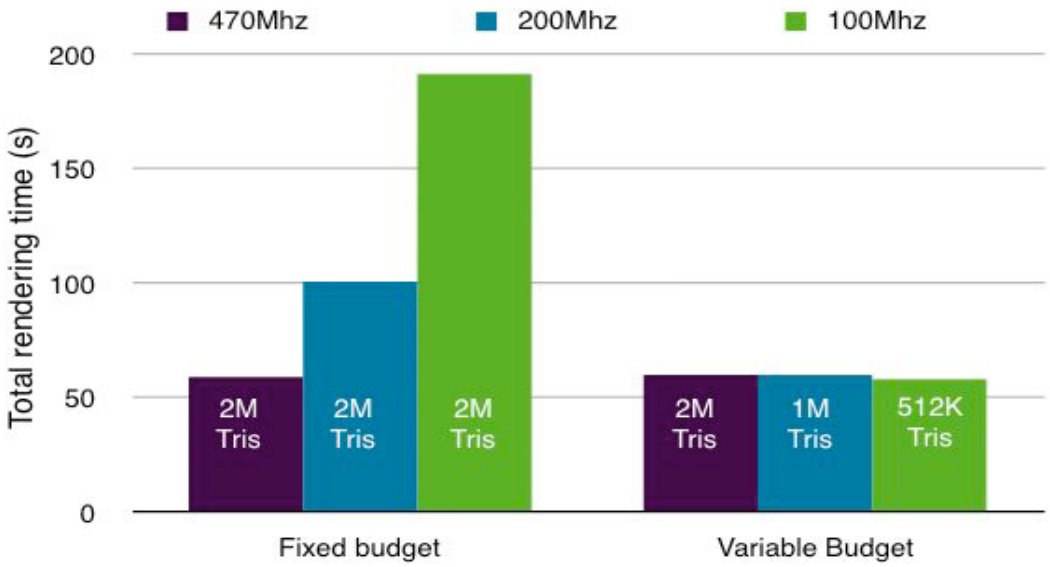

Figure 5.16: GPU workload control. By adjusting the triangle budget our method can compensate for a slower GPU. The number of nodes in the scene remains constant since the CPU and GPU workloads are independent.

Combining the control over the node budget with the control over the triangle budget and allowing these processes to be performed independently, our method can optimize the performance when using various hardware configurations. Previous methods that tie the resolution of a node to its level in the hierarchy would suffer when using a slower CPU or GPU due to inability to adjust the per-node resolution without an expensive recalculation of the hierarchy.

5.4. Analysis

The rendering model used in this method differs significantly from previous approaches. As described above the actual geometric data is stored in textures and fetched in the vertex program. While this approach gives us more flexibility to store and render the data, it also imposes a significant performance penalty. The rendering throughput for the vertex texturing method drops to a maximum of 120 million triangles/second compared to 450 million triangles/second for the mesh-based renderer. Although this performance difference diminishes when running the full system due to the overhead of other aspects of the program, such as adaptation performance or the loading of textures, it nevertheless still exists.

As mentioned before the performance of the rendering is dependent on many factors, such as amount of data sent to the GPU and the format of that data. Because our meshes are 2D we can send much less to the GPU then traditional methods, which require a minimum of 20 bytes per vertex, or 2.5 times the data we send. This 20 bytes of data per vertex provides only the position and texture coordinate, with the remaining data being sampled in the fragment program. In order to send a full mesh with all of the information approximately 44 bytes are required.

While the vertex texturing approach is preferable for the operation of our system, the huge performance decrease of that method led us to the mesh-based approach. The use of the Vertex Buffer Object data has greatly improved the rendering throughput of our system, allowing more geometry to be rendered per second as shown in the results. While

it has many advantages, the mesh-based rendering method also has several disadvantages, the main of which is the increase in amount of data sent to the GPU per frame. By creating the border strips we are forced to continuously create and upload new versions of the border geometry, even when caching the borders. This leads to greater GPU bandwidth consumption, potentially reducing the overall performance of the system. As a result there is a tradeoff between the number of nodes used to render the geometry and rendering performance. While the performance decrease is minimal for several hundred nodes, the use of larger node budgets will incur a significant performance penalty.

The use of static index lists for all combinations of border conditions has been investigated in order to reduce the computational overhead of mesh-based border merging. Unfortunately, there are far too many possible combinations of border resolutions to store all of the index lists in memory, since each border can have multiple neighbors, each at a different hierarchy depth and resolution. Models such as the Earth Dataset would require hundreds of megabytes of memory to store the index data, especially when trying to use a single render call per border.

5.5. Summary

The rendering method presented in this chapter utilizes some of the fastest paths through the OpenGL pipeline in order to maximize the rendering throughput of the system. The main contributions presented in this chapter include:

- **Optimized Rendering -** Limited per-vertex data combined with vertex reordering and GPU processing greatly increases the rendering performance of our method

- **GPU-Based Border Merging -** The vertex texturing method uses the GPU to perform node border merging, matching node resolutions to remove cracks. Even the mesh-based method offloads some of the border merging work to the GPU.

- **Optimized Data Storage -** Because of the use of geometry images we can store terrain data in a reduced form, storing only the height value and still producing a spherical model of the earth.

- **GPU Vertex Processing -** The use of geometry images permits the vertex data to be more easily modified on the GPU due to the use of parameterized data along with random access to neighbor vertex data.

- **Per-Fragment Data Mapping -** Throughout the rendering process we operate on parameterized data, which allows us to use apply additional data such as normal or color information from texture maps in the fragment program, resulting in higher quality output.

6.Data Management

The data management subsystem is responsible for locating the appropriate data segment for every node in the cut. The data is requested using a queue based mechanism which allows the data loading to be prioritized, optimizing the request process. The data requests are then checked against resident data, and only loaded from the hard-drive when necessary. The loading process itself is performed asynchronously to the rendering and adapt threads, allowing the data to be loaded in the background.

6.1. Data Management

Our system maintains most-recently-used caches of data in both video memory and main memory, with non-resident data being fetched asynchronously in a separate thread based on a priority queue mechanism.

The asynchronous loading is achieved through the use of secondary threads that fetch the requested data from the hard-drive and load it into system memory. The reason for this is twofold: the OpenGL pipeline can lock CPU processing until it completes, and more importantly the hard-drive I/O instructions will lock the main thread and prevent the frame from being rendered.

Our solution to this problem is the use of additional loader threads, each of which is responsible for the loading of a specific texture type, such as the geometry image or the normal data. Each of these is compartmentalized, and handled by a texture manager class, which is responsible for the starting of these loader threads. Each texture manager

launches one loader thread which is kept alive and idle for the majority of the time. This is done to avoid the overhead of spawning a new process, which would increase the latency between receiving a request and the loading of textures.

Each texture manager also employs several priority based queues which keep track of the requested textures as well as the least recently used textures stored in both the GPU and the system memory. By properly refreshing these queues we can easily find the next texture that should be unloaded, giving us the ability to control the memory usage of the system.

6.1.1. Texture Request System

Once the model is adapted to the scene and is ready to be rendered our system attempts to locate the appropriate textures for each node that is going to be rendered. Each texture request contains a number of fields describing the resolution of the texture and the area the texture is meant to cover. Each request also contains a priority value and a time stamp which allows requests to be processed at various rates, or discarded if it is no longer necessary.

The data locator function first checks for any textures that are of at least the required resolution and which cover at least the required area. If a texture that exactly matches the requested texture is available, it is assigned as the current texture for that node. If the texture covers an area larger then what the node covers, the node passes appropriate offsets and multipliers into the vertex program so that it addresses the appropriate portion of the texture while still issuing a load request for the optimal-sized data. If no

suitable texture is found at the nodes resolution, the search resolution is lowered by half and the process is repeated. If a texture of lower resolution is found, it is set as the current texture for that node, and a load request is added to the texture load stack with a priority value equal to the number of resolution levels that the current texture is below the required texture. Textures of lower resolution that cover an area larger then the requested area are also considered, and follow the same steps.

If no texture is found that satisfies the texture request, a load request is added with a maximum priority value that will cause the program to wait until the texture is loaded.

If a request for the same texture already exists, the request is ignored and the existing request is stamped with the new request time to keep it active. If a request is received for a texture that has already been requested, only the highest resolution request is kept, since the higher resolution texture can be used for the lower resolution requests.

When a texture request is added to the request queue we also take note of the storage size of the requested texture in order to make sure that once the texture is ready to load there will be sufficient memory to load it. This is necessary to prevent the loader from loading too many textures, which may cause the system and the GPU to begin swapping.

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 selection of the appropriate resolution of the texture is based on the resolution of the display window, and the bounding box associated with the node. As the node approaches the camera its bounding

box will occupy an increasing portion of the window, requiring a higher resolution texture in order to maintain high visual fidelity. Our method attempts to achieve a 1-to-1 mapping between the texel from the texture and pixel from the display. As a result, the textures will be progressively loaded as the object approaches the camera.

6.1.2. Data Loading

The data loading process begins by removing a texture request from the load queue. The request is then checked to determine if it is recent in order to avoid the loading of textures that are no longer required. If the texture has not been requested recently it is ignored.

The next step is to check for sufficient memory on both the GPU and in the system RAM for the texture to be loaded. If the amount of available memory is insufficient textures will be removed from the GPU and RAM as necessary to provide storage for the new textures.

The texture loading process then starts by preparing to read the data from the harddrive. As with most of the loading functions, this is performed in a loader thread so the I/O does not block the renderer as shown in Figure 6.1. To load a texture we first determine what mip-map resolution stored on the harddrive matches the user-requested texture most closely. The loader then loads the texture, or a subset of it, using a single read call. If the requested texture is a subset of the texture stored on the disk, a horizontal strip is read whose height is the size of the requested texture. This allows us to still perform

the read using a single I/O command, and to perform some copy operations in system RAM where the penalty for such operations is much lower.

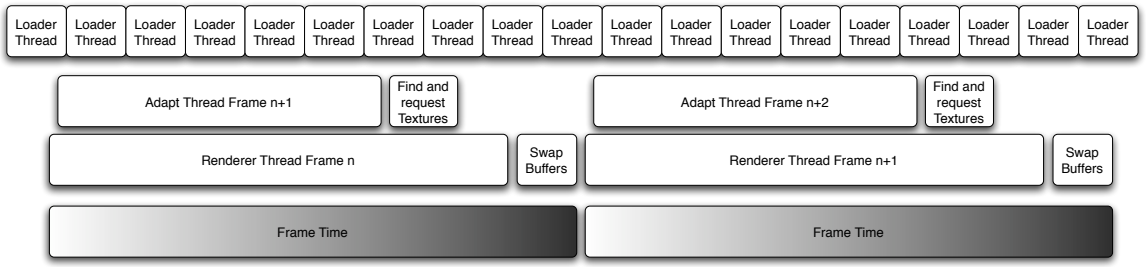

Figure 6.1: Data Load Pipelining: The loader thread operates independently of the render and adapt threads, loading data in the background as it becomes requested.

Once the block of data is loaded, some processing needs to be performed on it, including removing the excess data and reducing the resolution of the texture as necessary. While this could be performed by OpenGL when the texture is loaded, the larger texture would still have to be stored in RAM unnecessarily increasing the storage size, or read back from the GPU which is still a relatively slow operation.

The final stage of the loading process is to add the texture to the storage queues and to notify the main rendering thread that the texture is ready to be loaded onto the GPU. The loading cannot be performed in the loader thread because the OpenGL context generally cannot be shared between multiple threads or processes since the GPU performs its own memory management. As a result the main thread is still responsible for the actual loading of the texture onto the GPU which incurs a small performance penalty as the data is transfered to the graphics card. Fortunately GPUs have been designed for this type of upload action, and can transfer vast amounts of data to the graphics card, in ex-

cess of 4GB/s. We also throttle the number of textures loaded per frame to reduce the possibility of too many texture loads pausing the rendering for a noticeable period of time.

6.1.3. Runtime Management

The loaded textures remain stored on both the GPU and in system RAM for as long as they are in active use. Every frame the texture is used, it becomes stamped with that frames time stamp to notify the manager that it is still necessary.

Because the storage of all the textures on the GPU and in the system RAM would easily exceed the capacities of both, they need to occasionally be pruned. This is performed by maintaining an accurate count of the total amount of memory used by the textures in both the system and GPU memory. Once the amount of available memory becomes low we will start to remove textures as necessary. A texture can be removed from the GPU or from both the GPU and the RAM.

The removal of a texture from the GPU does not force the removal of the texture from the system RAM, allowing the texture to be still quickly loaded onto the GPU should it again become necessary. Thus the system RAM becomes a cache for the GPU, allowing more textures to be stored there and quickly loaded onto the GPU as necessary. To delete the texture from OpenGL we send a delete request to the main render thread, which in turn removes the texture from the GPU.

Once a texture is no longer loaded on the GPU, and has not been requested by the rendering thread for a preset period of time, or if more space is required, it will be re-

moved from the system RAM as well. The texture manager will remove the texture and all of its associated information, and the data is once more only available from the much slower hard-drive.

6.2. Data Reusability

One of the strengths of our system is its ability to reuse the currently available data in place of the requested data. 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. While this has some associated temporary effect on the triangle count and geometric error, it prevents our method from stalling while waiting for data. 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. In this data representation, 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 reusable data is inexpensive storage-wise (on disk, the data are stored in blocks with each resolution level only represented once), and the least-recently-used cache replacement policy ensures that textures are replaced in a timely fashion after their replacements arrive. As an exception to the LRU policy, we also find it convenient to lock the lowest resolution texture for the level-zero nodes in memory to ensure there is always some fast-rendering representation available for the entire model (this low-

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

To map a node into ancestor or lower resolution data some additional processing has to occur. For the vertex texturing method all of these operations are minimal, and involve the alteration of the texture coordinates on the GPU. By passing texture coordinate offsets into the vertex program along with resolution difference information the vertex program can adjust all of the texture coordinates such that the correct subset of the data is used. The addressing of the data in the mesh-based renderer is a little bit more expensive performance-wise, as it requires a new index list to be computed whenever data of different resolution or size is used. By using the caching functionality of the VBO border strips the recreation of the index lists is kept to a minimum, with cached indices used whenever possible.

6.3. Results

To test the performance of the loading mechanism we have measured the amount of time between a texture request and the time the texture becomes available to the renderer. We have played a path on the Earth Dataset that moves in close to the surface and moves around the surface to simulate typical user activity. The results for this experiment are shown in Figure 6.2. As expected, when the user performs more sudden movements such as zooming or fast panning, the loader needs to request more textures to keep up with the motion. However, since our system uses pre-existing data in place of unavailable data the system will never stall while waiting for data.

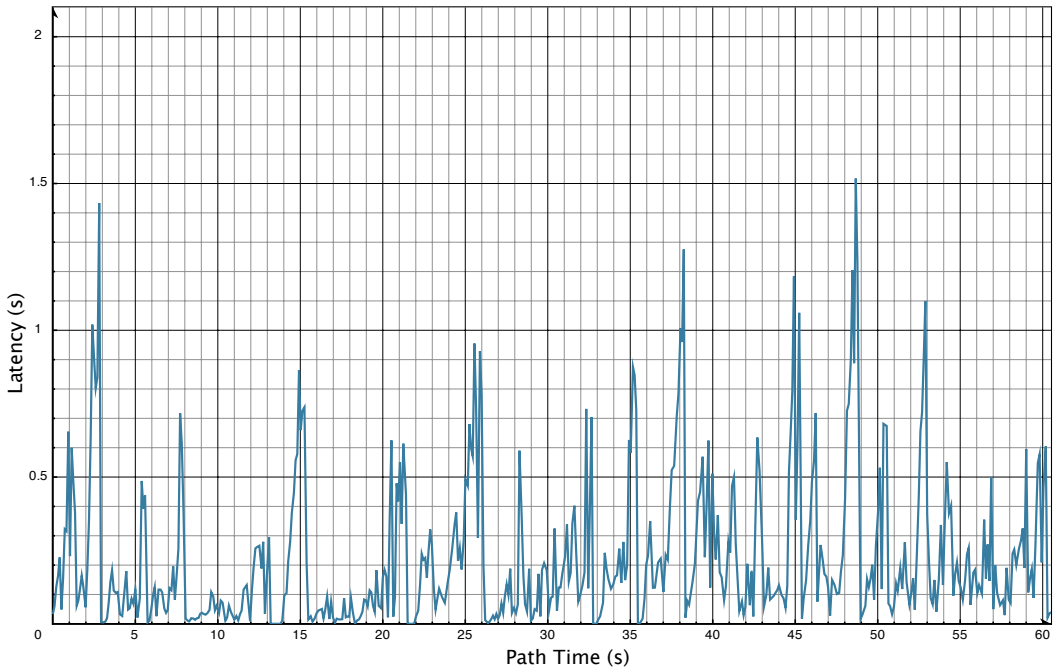

Figure 6.2: Loading Latency. By simulating typical user activity of zooming in and out and panning around the Earth Dataset we have tested the loader latency. As shown in the graph, our system handles the loading of large amounts of data well, with an average latency of less then ¼ of a second.

We have also tested the throughput of the loader by measuring the amount of data loaded in the path, as shown in Figure 6.3. On average, our system is capable of loading 10-12MB/s, enough to fulfill the texture requests placed by the system.

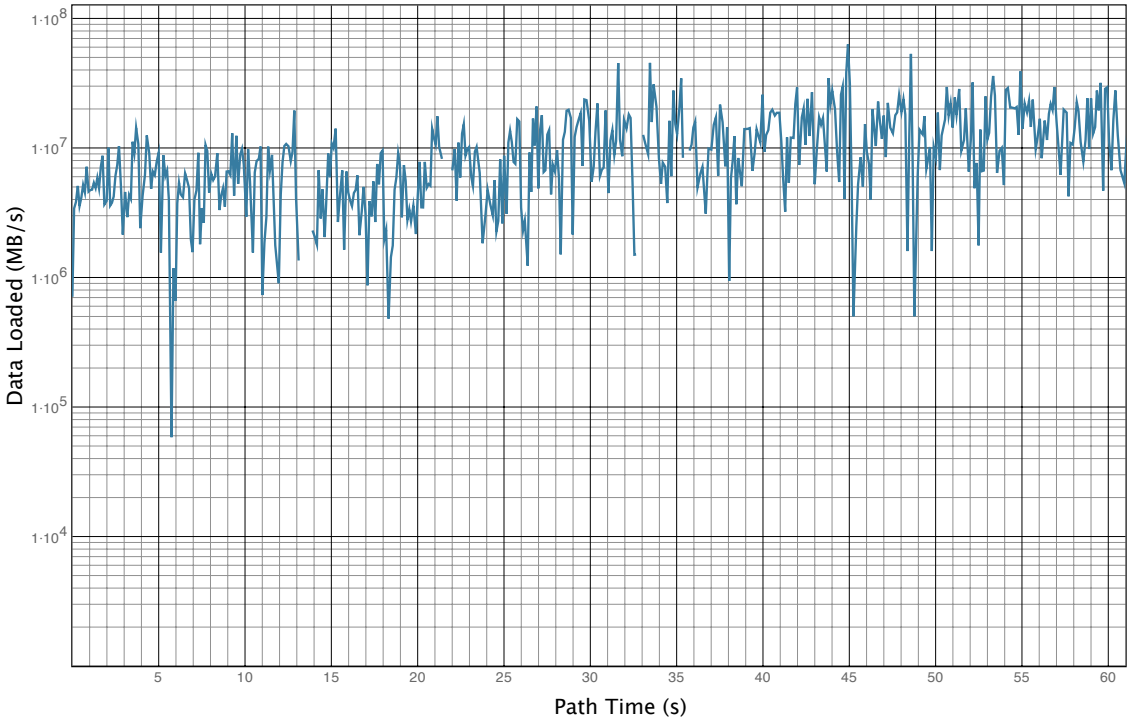

Figure 6.3: Loading Performance. The loading performance of the data management subsystem is shown in this graph. The loader performs well, with the majority of the spikes caused by requests for smaller textures.

6.4. Analysis

The data loading algorithm presented in this chapter is relatively similar to what most comparable methods employ. By caching the most recently used data we can retrieve it from RAM and load it into the GPU more quickly, thus reducing the data latency caused by the slow read speeds of the hard drive.

The main departure from traditional data management is the use of the reusable data. The reusability provided by data cached on the GPU and in RAM, allowing our method to never stall while waiting for data, since both lower and higher resolution data can be used to fill in missing data while it is being loaded from the hard drive. Although minimizing data read latency is still important, it is no longer a bottleneck. After the first frame sufficient data to render any portion of the model is always stored on the GPU. As a result we do not have to have load all of the data requested in a frame, nor do we have to maintain data outside the view frustum at high resolution. Because the low resolution data can be used in place of higher resolution data, nodes outside the view frustum can be stored with minimal data, and still rendered when they come into view, regardless of their hierarchy and LOD level.

This capability combined with the very flexible node border merging allows us to render areas that come quickly into view, without stalling while waiting for data. More traditional methods would find it difficult to render nodes which quickly come into the view frustum without using multiple view frusta, since the detailed nodes coming into the view would have to be rendered at high resolution. Without the appropriate data loaded, or with only low resolution data, they would be incapable of rendering the complete model due to the node neighbor restrictions. Previous methods have used nested frusta to load data that is within a specific radius of the main view frustum. This allows data near the view frustum to be fetched in the background, creating a buffer of data already in memory. This however depends on the user not navigating quickly, as only a limited amount of data can be loaded in this fashion.

The rendering of VBO geometry while using non-optimal data requires new sets of indices to be computed. While it is possible to pre-compute all of the possible permutations of index lists, this would require vast amounts of storage due to the sheer number of possible layouts. The use of high-resolution original data, such as the Earth Dataset combined with a deep hierarchy would further increase the memory requirements by allowing thousands of possible index lists. The use of caching is therefore a more viable solution, even if it requires the index lists to be occasionally recomputed.

7.Parallel Processing

In this paper we use a tile-based layout to distribute the workload amongst multiple CPUs, GPUs and computers. The tile-based layout simplifies the subdivision of the workload and reduces the amount of inter-node communication required to achieve a seamless final image. Furthermore the tiles can be dynamically resized in order to balance the work performed by each tile, allowing for a heterogeneous system with a non-uniform distribution of geometry in the view frustum. The use of tiles also allows the system to scale more easily by simply further subdividing the view frustum, allowing the system to be used with a variety of multi-GPU, multi-core and multi-node systems.

7.1. Distributed Pre-processing

The preprocessing stage presented in this method operates on a node basis, in which each top-level node is processed independently, and subsequently merged into an object. The independence of these computations allows the preprocessing process to be performed in a distributed setting, for example using multiple threads or multiple computers.

7.1.1. Pre-Processing Nodes

The distributed preprocessing version of the preprocessing system is performed in two stages. The first stage of the process performs a single-node pre-computation as described in Section 3.4.4. The pre-computation for each root node of the hierarchy is inde-

pendent, and can be performed on a separate computer in a cluster. Each computer can also parallelize the pre-processing when multiple CPUs are available by processing one node per CPU.

The data for each root node is then saved to a intermediate file, which contains sufficient information for the root nodes to be merged into a single object in the second stage of the pre-processor.

7.1.2. Merging Node Data

The second stage of the preprocessor combines the information from the individual top-level nodes into a complete object. The main task in this stage is to take the data from the individual top-level nodes, and find neighbors for each tile created. Because this task requires the access to all of the nodes, it must be performed on a single machine. This task is very fast, as all of the data required to locate the neighbor nodes has been pre-computed in the first stage of the preprocessing. All that remains is the search through other top-level nodes to find one with a matching border. This process is unnecessary when dealing with height maps, due to their longitude and latitude partitioning, creating implicit neighbor information.

7.1.3. Results

To test the performance of the distributed preprocessing system we have compared the performance of a single preprocessing node to the performance of a cluster, with the results shown in Table 7.1. As expected the performance improvement is very

significant when using a large number of compute nodes to perform the preprocessing. The tests were run on a compute cluster composed of dual Pentium Xeon 2.8Ghz CPUs and 2GB of ram, sharing the dataset on a parallel file system.

Dataset	Processing Nodes	Preprocessing Time	Speedup
Thai Statue	1	440s	1x
Thai Statue	35	14s	31x
Earth Dataset (Spherical)	1	5529m	1x
Earth Dataset (Spherical)	35	347m	16x
Earth Dataset (Flat)	1	3251m	1x
Earth Dataset (Flat)	35	130m	25x

Table 7.1: Distributed Preprocessing: By distributing the preprocessing workload amongst a cluster of computers the time required to process a model is drastically reduced. All of the times are computed using a cluster of computers with dual Intel Xeon 2.8Ghz CPUs and 2Gb of RAM using a shared Lustre file system [Schwan 2003].

The only factor limiting the distributed performance is the shared storage used to store the geometry data. While the filesystem used is efficient, using an array of parallel storage nodes, there is still an overhead to accessing the data over a network and from multiple computers simultaneously.

7.2. View-Frustum Tiling

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.

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 7.1). 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 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.

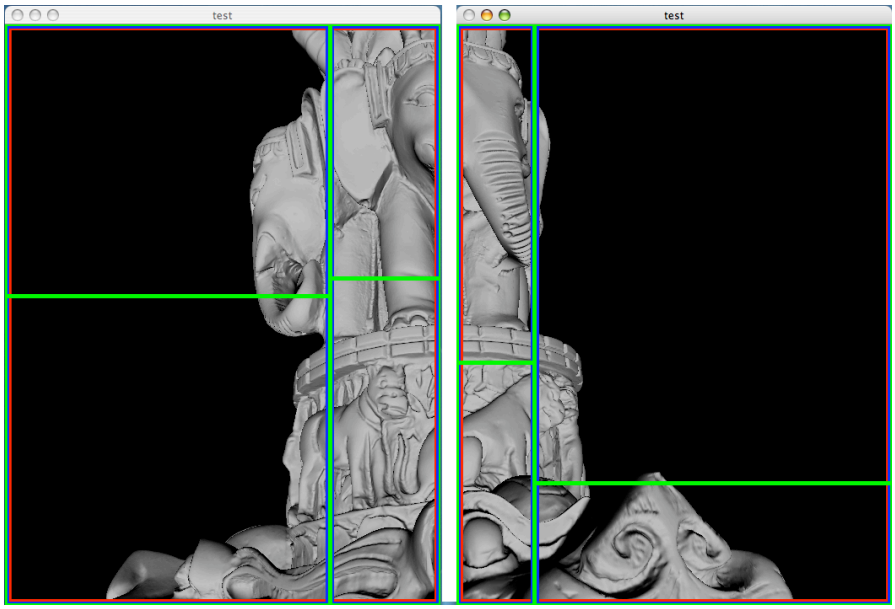

Figure 7.1: kD-tree view frustum partitioning. The kD-tree structure used for load balancing is apparent in this two-PC system with two render tiles per machine tile and two adapt tiles per render tile.

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.

The merging of borders between screen tiles is required to prevent cracks and seams between neighboring screen regions caused by rendering a piece of geometry at two different resolutions on neighboring screen regions as shown in Figure 7.1. By merging two hierarchy cuts such that the overlapping geometry is rendered at the same resolution we prevent these artifacts from occurring, resulting in a higher quality image.

7.3. Load Balancing

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 design a more predictive approach [Funkhouser and Sequin 1993] 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 [Fuchs et al. 1989]. 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 [Mueller 1995], we adjust the tile sizes by modifying the position of each splitting line in the view frustm. For a given splitting line the new position is computed as follows:

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

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 tiles, we adjust the number of nodes it is assigned as follows:

$$nodes_{new} = \sqrt{\frac{time_{target}}{time_{old}}} * nodes_{old},$$

where $\text{time}_{\text{target}}$ 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:

$$error_{new} = \sqrt{\frac{time_{old}}{time_{target}}} * error_{old},$$

where $\text{time}_{\text{target}}$ 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.

7.4. 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 the 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 7.2). 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.

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 it's 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 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 united.

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 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 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(B*log*N). Once cut unification is completed the tiles can be rendered seamlessly on the screen as shown in Figure 7.2.

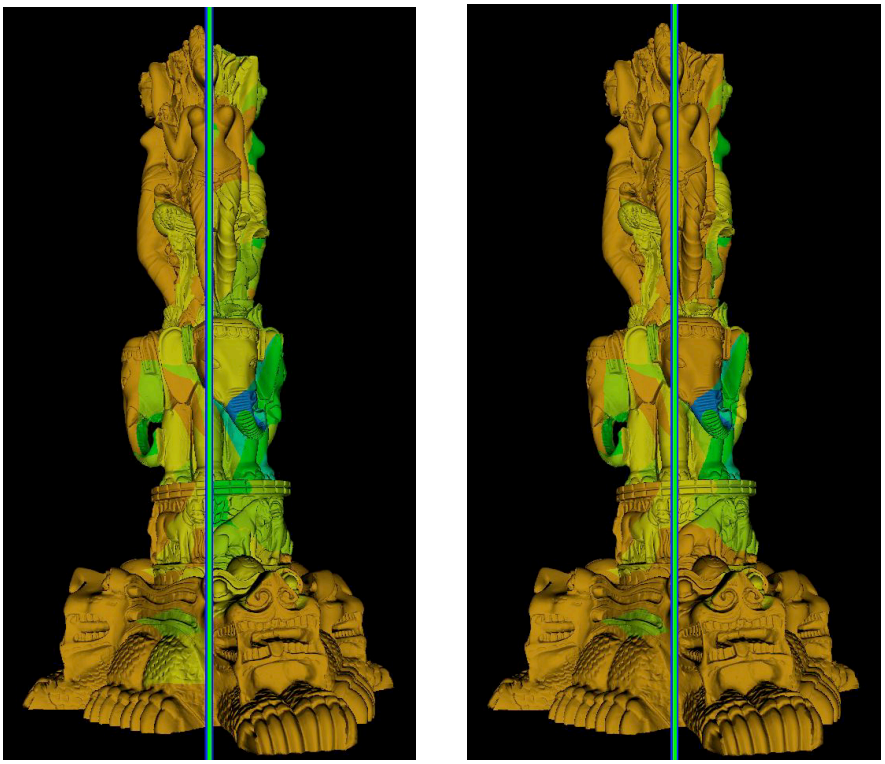

Figure 7.2: Cut Unification: The left image shows the two hierarchy cuts disagree on the resolution of nodes overlapping the tile boundaries. The right image shows the model after the cut unification process. The nodes are colored according to their resolution, from low-resolution in blue to high-resolution in orange.

To test the cut unification process we have measured the time required to unify several cuts through the Earth Dataset hierarchy as shown in Figure 7.3. As expected, the amount of time required to unify the hierarchy depends on the number of nodes used and

the number of screen tiles. However, even when unifying four adapt tiles with 6000 tiles each our method requires only 1.4ms which becomes pipelined with the rendering process as shown in Figure 7.3.

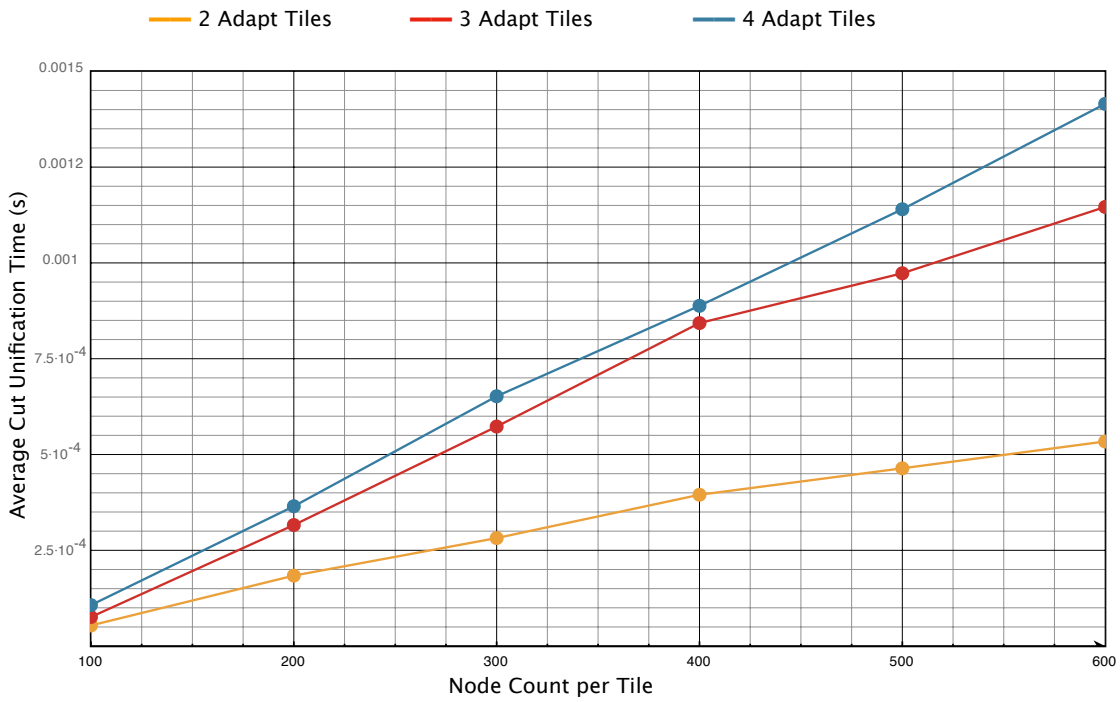

Figure 7.3: Cut Unification Results: To test the performance of the cut unification process we have tested a variety of adapt tile configurations and node configurations. While the unification time increases as a factor of the node budget and number of tiles, the time required to unify the cuts remains low

7.5. Parallel Adaptation

7.5.1. Introduction

The ability to perform work in parallel is of ever increasing importance with the availability of multi-core, multi-CPU workstations and personal computers. Today the majority of computers sold contain a multi-core CPU, or multiple multi-core CPUs, multiplying the possible computing power of a single workstation. The availability of this processing power places higher demands on programs utilizing it, as it is much more difficult to harness the power of these multi-core systems due to the difficulty of subdividing and parallelizing the workload.

The first stage in the parallelized renderer is to parallelize the adapt process. The advantage of such a system is that even though the aggregate time of all the adapt processes might exceed the adapt time of a single adapt node, the maximum CPU time will be lower, allowing the adapt process to complete more quickly when using a parallel method.

7.5.2. Parallel Adaptation

The distributed adapt process starts by using the previously described adapt algorithm to adapt the geometry in each adapt tile to the user specified adapt parameters. This process is independent for each screen region, allowing each adapt tile to be processed separately.

The parallel adapt process is also pipelined with the renderer, allowing the adapt threads to perform the adaptation while the rendering thread is drawing geometry to the screen as shown in Figure 7.4.

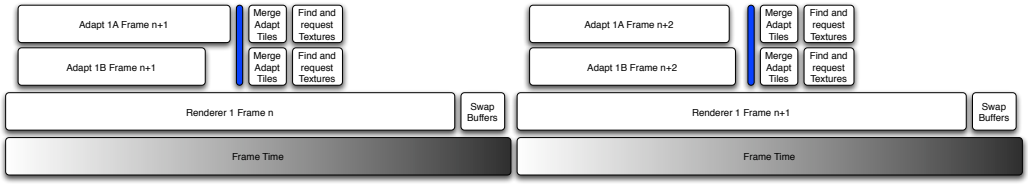

Figure 7.4: Multi-CPU Adapt: The parallel adapt process is pipelined with the renderer, allowing the adapt time to be removed from the frame time. The parallel adapt process requires a synchronization at the end of the adapt process, shown in blue, required to unify hierarchy cuts.

The parallel adapt process requires a cut unification described in Section 7.4. Upon the completion of the adapt process, the adapt threads synchronize as shown in blue in Figure 7.4. The synchronization allows the adapt tiles to exchange the cut unification data so that the individual hierarchy cuts can be unified.

7.5.3. Data Management

The use of multiple adapt tiles requires an additional copy of some of the hierarchy data to be present in order to accommodate the selection of multiple cuts in parallel. These additional copies of the cut are not, however, expensive in terms of RAM usage, as they duplicate the bare minimum of data, referencing the original data wherever possible.

The additional adaptation tiles also change the data management process, as multiple hierarchy cuts can now request the data to be loaded onto the GPU from different processing threads. In order to prevent multiple copies of a texture from being loaded for each adapt tile a single set of loading threads is used. Each adapt tile sends its requests to

a single texture manager where the textures are loaded into a single cache that can be used by any of the adapt tiles.

7.5.4. Results

The use of multiple adapt tiles allows our system to distribute the work involved in selecting a cut from a hierarchy. The parallel adapt process therefore decreases the maximum amount of time required to adapt the hierarchy, even if the aggregate of all the adapt nodes is greater. As a result the adapt process can be completed in less time when using multiple adapt nodes as shown in Table 7.2.

In this example a constant error threshold and node budget was used, and this budget was evenly distributed to each of the adapt nodes. As a result each of the adapt tiles in the four tile graph had a quarter of the nodes when compared to the single adapt tile. As shown in Table 7.2, our method achieves a significant speedup when using multiple adapt tiles, with average boost of 250% for a four-tile adapt setup.

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

Table 7.2: Adapt performance of multi-tile systems: This table shows the percentage improvement in adapt times when using a variety of node configurations. While the system may slow down at times, the overall performance increase is significant.

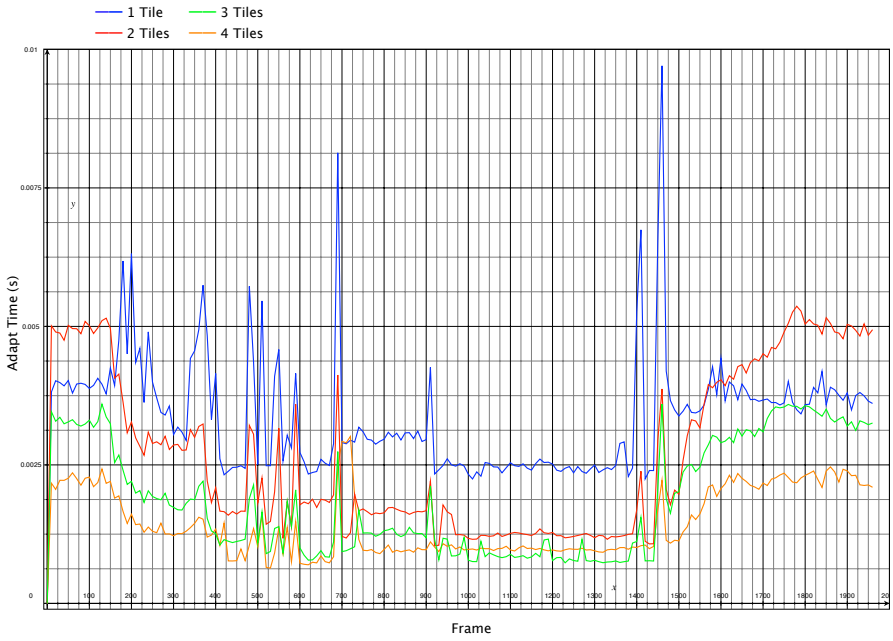

Figure 7.5: Adapt times when using multiple adapt tiles. As shown in the graph, by using multiple adapt times the overall adapt time decreases. Results gathered on a Mac Pro running OSX 10.4.8 with 2 dual-core Xeon 2.66Ghz CPUs adapting the Earth dataset to a triangle budget of 8 million triangles.

We have also tested the workload-balancing capabilities of the parallel adapt system. To test this we compared the maximum errors over a path when using 2 adapt tiles with fixed and dynamic borders on the USGS Earth dataset. The results of this test are shown in Figure 7.6.

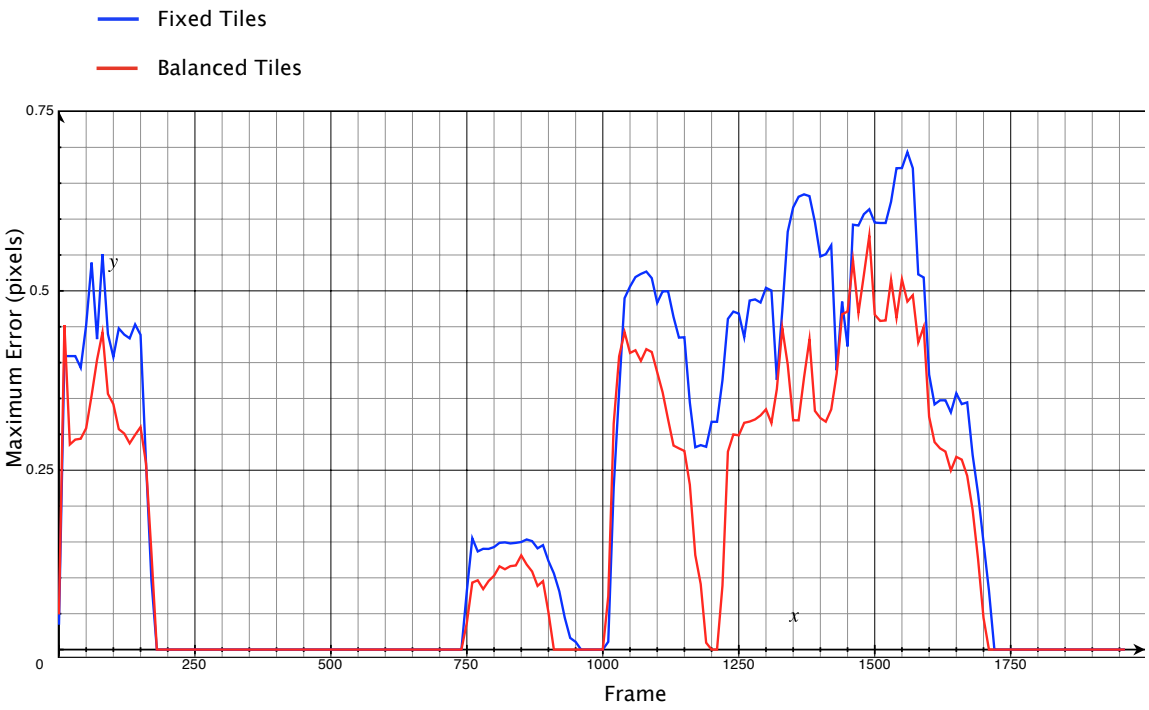

Figure 7.6: Workload balancing example. By allowing the tile borders to shift our method can reduce the geometric error or triangle count when using multiple adapt tiles. Through balancing the amount of geometry or error in each tile we distribute the geometry more evenly, ensuring that no empty adapt tiles exist. Results from a path over the USGS Earth dataset using a triangle budget of 8 million triangles.

7.5.5. Discussion

The parallel adapt capabilities of our system allow multiple adapt tiles to assemble the final hierarchy cut used to render the model. This allows the system to take advantage of the latest trends in multi-core and multi-CPU processing. By breaking down the

adapt problem into several smaller pieces the adaptation time is generally reduced, allowing the system to spend less time adapting the model and more time rendering it.

While this is generally true, there are several cases when a single adapt thread can adapt a model with the same performance as a parallel implementation. The main cause of this is the increase in view-frustum culling present in the parallel adapt process. The appearance of a new node in the view-frustum requires additional computation to refine it, forcing more time to be spent adapting. By splitting the view frustum more nodes enter and leave each adapt tile, increasing the processing overhead.

7.6. Multi-GPU Rendering

7.6.1. Introduction

Distributed rendering systems generally refer to systems which operate on a cluster of computers tied together using a high-performance network. Each node in such a system performs its share of the rendering workload synchronized using a central node.

Recently, however, a new type of parallel rendering system has become possible. The availability of multiple GPUs in a single computer has created a parallel renderer with capabilities similar to distributed renderers, but using a system bus instead of a network connection.

7.6.2. Multi-GPU Setup

The multi-GPU rendering of a hierarchy is relatively similar to the single-GPU method with some added set-up and data management involved.

The first step in creating a multi-GPU renderer is the creation of separate thread with a OpenGL context on the secondary GPU, allowing the system to communicate with the video card. This is accomplished by first making the operating system aware of a secondary video card by either configuring the X server in Linux, or by attaching a monitor to the secondary GPU in OSX. Once the OS is aware of multiple GPUs, OpenGL contexts on the secondary GPUs can be created using the appropriate CGL or GLX calls. The context is then attached to the renderer thread, creating one thread per context so that no OpenGL context switches are required when running the renderer.

The use of multiple rendering and adapt threads requires some synchronization in order to correctly display the frame. As a result, the pipelining requires some additional synchronization stages which allow the two GPUs to communicate, as shown in Figure 7.7.

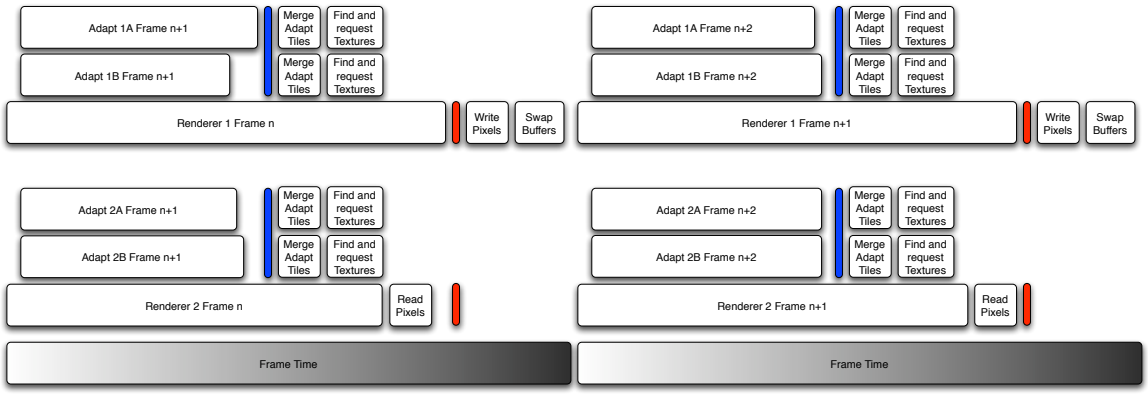

Figure 7.7: Multi-GPU pipelining: The multi-GPU renderer requires some synchronization to take place between the GPUs in order to write a complete framebuffer fragment to the final display device. The GPU synchronization location is shown in red.

The use of multiple GPUs requires the readback of the secondary GPUs in order to display the output on a single monitor. As a result, our system will allow each GPU to render its hierarchy cut independently, synchronizing when the secondary GPUs have finished reading back their framebuffers, shown in red in Figure 7.7. At this point the primary renderer will finish its section of the framebuffer and write the secondary framebuffers to the display, completing the rendering of the frame.

7.6.3. Multi-GPU Adaptation

The multi-GPU adapt process is very similar to the parallel adapt approach presented above. The screen is divided into several sub-regions, one for each physical GPU present in the system, each of which is adapted independently. Each of the renderer tiles can then be subdivided into any number of adapt tiles, each of which will operate inde-

pendently. At the end of each adapt phase each of the adapt tiles will exchange cut unification data as described above in order to unify all of the hierarchy cuts present in the view frustum.

The hierarchy cuts for each GPU are sent to the individual video cards, and rendered in parallel. Once the rendering is completed, the resulting framebuffer is retrieved from the video cards, and temporarily stored in system memory. The amount of data read back is limited to the portion of the image rendered on the GPU, which helps to improve the readback performance of the card. The framebuffer fragment is subsequently written to the master video card, where the final image is formed and displayed on the screen.

7.6.4. Multi-GPU Data Management

The single-computer, multi-GPU greatly simplifies the data management process. A truly distributed system requires that each node have access to the data, and it's own cached copy of parts of the data in system memory. By using a single-computer parallel renderer the GPUs can share the system RAM cache, reducing the hard-drive loading overhead.

The data management is therefore very similar to the parallel adapt version of the system, where a single set of texture managers is used to load the texture from the hard drive into the system RAM. The texture requests are issued from each renderer into a single queue which is used to load textures from the harddrive. The texture managers then proceed to load the texture into system memory, at which point the texture is loaded into the GPU that originally requested the texture. This allows both GPUs to share the system

RAM cache while using separate GPU memory caches. Each GPU can therefore store only the data that it needs for its part of the screen, allowing a larger GPU cache to be used for each GPU.

7.6.5. Results

To test the dual-GPU renderer we rendered a test model using a computer with two video cards. The video cards used were the nVidia 7800GT and the nVidia 7300GT, unbalanced in their rendering performance in a approximately 2:1 ratio. The results show that using the two GPUs in unison gives our method a marked performance increase, as shown in Figure 7.8.

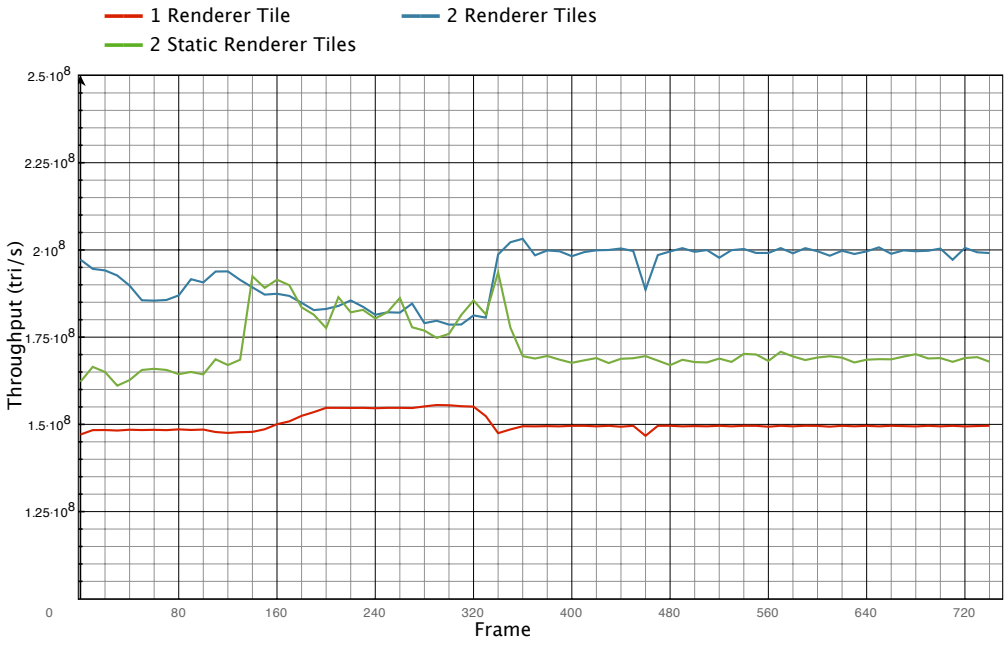

Figure 7.8: Dual-GPU rendering. This graph shows the performance increase when using a dual-GPU setup. The use of two GPUs with dynamic balancing gives our method a performance increase of up to 50M triangles/s. Even without balancing we see a 20M triangle/s improvement over the single GPU. Results gathered using a Mac Pro running SUSE 10.2 with two dual-core 2.66Ghz Intel Xeon CPUs, a nVidia GeForce 7800GT and a nVidia GeForce 7300GT and the Thai Statue model with a 1 pixel error threshold using 300 nodes per tile.

As shown in the figure, our method achieves a performance boost even without balancing the workload in this path. The workload balancing does, however, improve performance further, and ensures that no renderer tile is left empty.

We have also investigated the benefit of managing the memory of each GPU separately. To test this we have measured the percentage of textures shared by the two GPUs, and the percentage of the shared textures that are actively in use, shown in Figure 7.9.

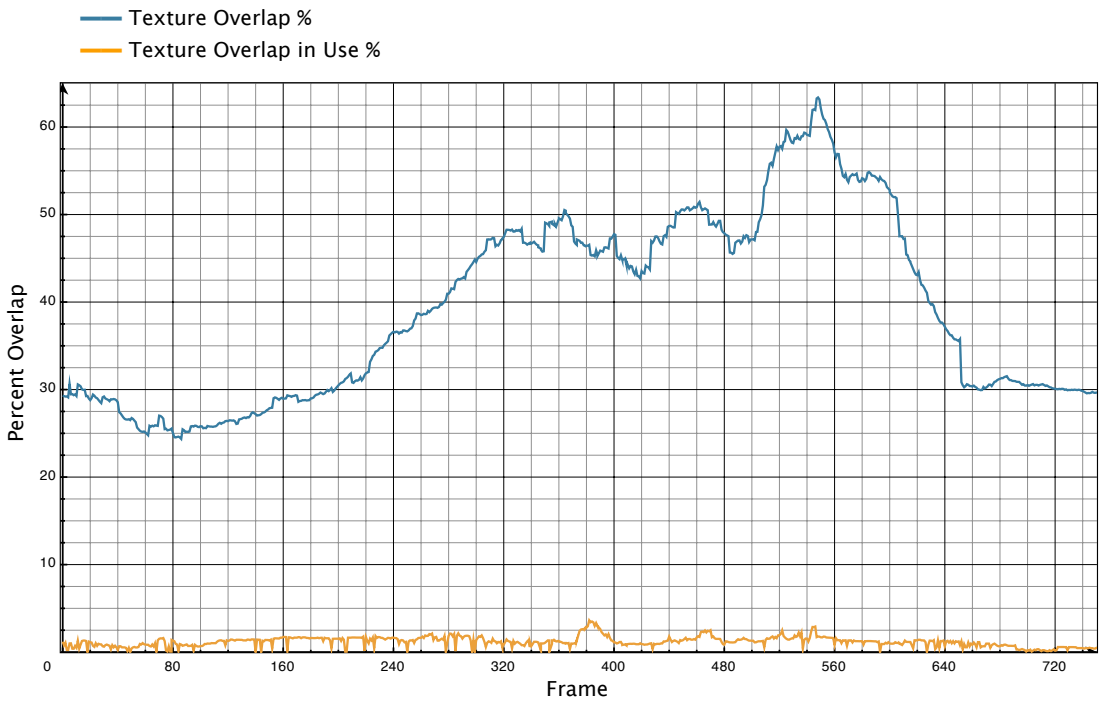

Figure 7.9: Shared textures: This graph shows the percentage of textures in GPU memory which exist on both of the GPUs, as well as the percentage of shared textures in use. While both GPUs share a large percentage of textures, approximately 40% on average, the majority of those textures are cached for future use, and could be paged out. Results obtained using the Thai Statue model using 300 nodes per tile and a 1 pixel error threshold

As Figure 7.9 shows the average shared memory usage is approximately 40% of the total textures stored on the GPU, and only a few of these textures are actually in use. As a result our method can use more of the GPU memory to store textures when compared to the SLI methods which mirror textures on both GPUs.

7.6.6. Discussion

The ability to use multiple GPUs has become increasingly important in recent years with even the video card vendors utilizing it in their high-end video products. Features such as SLI are now available in products from both ATI and nVidia, allowing games and other software to harness the power of multiple video cards. The SLI systems are transparent to the user, and can be used with any OpenGL or Direct3D capable application seamlessly. The main advantage of the SLI setup is its ease of use and the performance boost when rendering. Depending on the mode used, the SLI setup can increase both the vertex and fragment performance when using Alternate Frame Rendering (AFR). The AFR approach to multiple-GPU rendering provides a great performance increase at the cost of control latency and GPU memory overhead. The control latency is caused by the issuing of rendering commands to alternating GPUs before the input is processed. Although this may not be a large problem in a dual-GPU layout, when a larger number of GPUs is present the control delay might become unacceptable. In a SLI/AFR configuration the texture and geometry data is also mirrored to all of the GPU's in a SLI/AFR setup, the additional memory is lost. Furthermore the SLI/AFR approach requires that identical GPUs be used, reducing the upgradability of such a system.

The benefit of the multi-GPU method presented in this paper is the scalability and adaptability that it allows. By allowing for dynamically sized tiles our method can use any number of GPUs to render the scene, even in the presence of a heterogeneous configuration of video cards. The separation of GPUs also allows our method to take advantage of the additional memory of each GPU, allowing each video card to store its own

subset of the geometry and additional data without forcing a replication of the data. As a result each GPU can store the data it needs without wasting memory on data that is outside of its view frustum.

The main disadvantage of the multi-GPU renderer is the system overhead required to process calls to multiple GPUs. While rendering a small number of nodes with large numbers of triangles yields large performance increases, the use of a large number of nodes greatly reduces performance. This cost can be attributed mostly to the X-server which is not designed for parallel communication with video cards, and requires OpenGL calls to be serialized. This serialization of OpenGL calls reduces overall performance, causing the X-Server and the OpenGL drivers to slow down parallel rendering.

7.7. Performance Mode

While geometry-driven adapt modes are useful, it is frequently important to visualize geometry at a user-specified frame-rate, allowing as much of the detail to be visualized while maintaining an interactive frame-rate. Our system utilizes a performance mode, built on top of the error threshold adapt algorithm, to adjust the workload on the renderer.

7.7.1. Performance Mode Algorithm

The performance mode adjusts two key controls of the adapt and rendering system, the error threshold and the node budget. By monitoring the time required to render a

frame and adjusting the controls accordingly, the performance mode can maintain a user-specified frame rate.

The two factors adjusted by the performance mode have similar effects on the frame rate of the renderer. By increasing the error threshold the adapt mechanism will reduce the number of triangles in the scene, increasing the frame rate. Similarly, increasing the node budget will reduce the triangle count in the scene, further increasing rendering performance.

The performance mode algorithm is divided into two stages: error bound adjustment and node budget adjustment. The performance mode itself iterates through these two modes, adjusting the performance based on the time to render a window of frames using a feedback mechanism.

The first of the two stages, the error bound adapt stage, adjusts the error threshold to bring the frame time close to the user specified time. The error threshold is adjusted based on the ratio of the current frame time to the goal frame time. By performing this process iteratively, adjusting the estimated error threshold, the performance mode will achieve the user-specified frame time while minimizing the geometric error in the scene.

When the user-specified frame rate is achieved the performance mode will attempt to adjust the node budget to further improve the performance. By adjusting the node count the system will attempt to further increase performance, allowing the first stage to further lower the geometric error in the scene. The node adjustment algorithm will increase the node budget by a fixed node count and compare the performance before and after the modification. If the new node layout improved rendering performance the

node adjust system will keep increasing the node count until the performance stops improving. If the new node layout decreased performance, the adapt system will revert to the previous node layout and attempt to improve performance by decreasing the node count. The performance adjustment compares the rendering performance of both increasing and decreasing the node count due to the potential rendering performance decrease when too many rendering primitives are issued.

The two processes will iterate, switching after every successful modification of the node budget. This minimizes the error threshold in the scene while meeting the user-specified frame time.

7.7.2. Performance Mode Results

Using the performance mode our system is capable of achieving a target frame rate, while optimizing the use of hierarchy nodes. As shown in Figure 7.10, our method can adjust the performance to match the user-specified frame rate while minimizing the geometric error in the scene. The results were taken while rendering a path of the Thai Statue model using a quad-core 2.66Ghz Intel Xeon system with a NVIDIA 7800GT GPU.

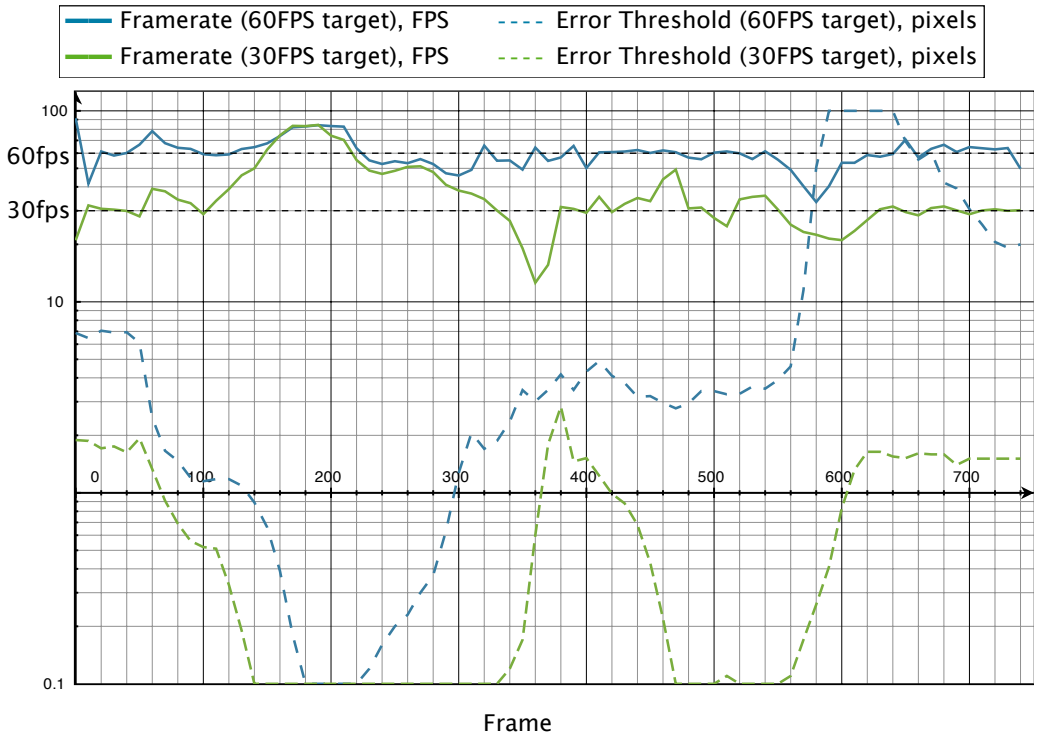

Figure 7.10: Performance Mode Framerate: By adjusting the error threshold for the hierarchy our method can achieve user-specified frame-rates while minimizing the error in the scene.

The performance mode also adjusts the node budget, increasing and decreasing it as necessary to optimize performance as shown in Figure 7.11. By increasing and decreasing the node count the performance mode can optimize performance, adding nodes to reduce triangle count or removing unnecessary nodes.

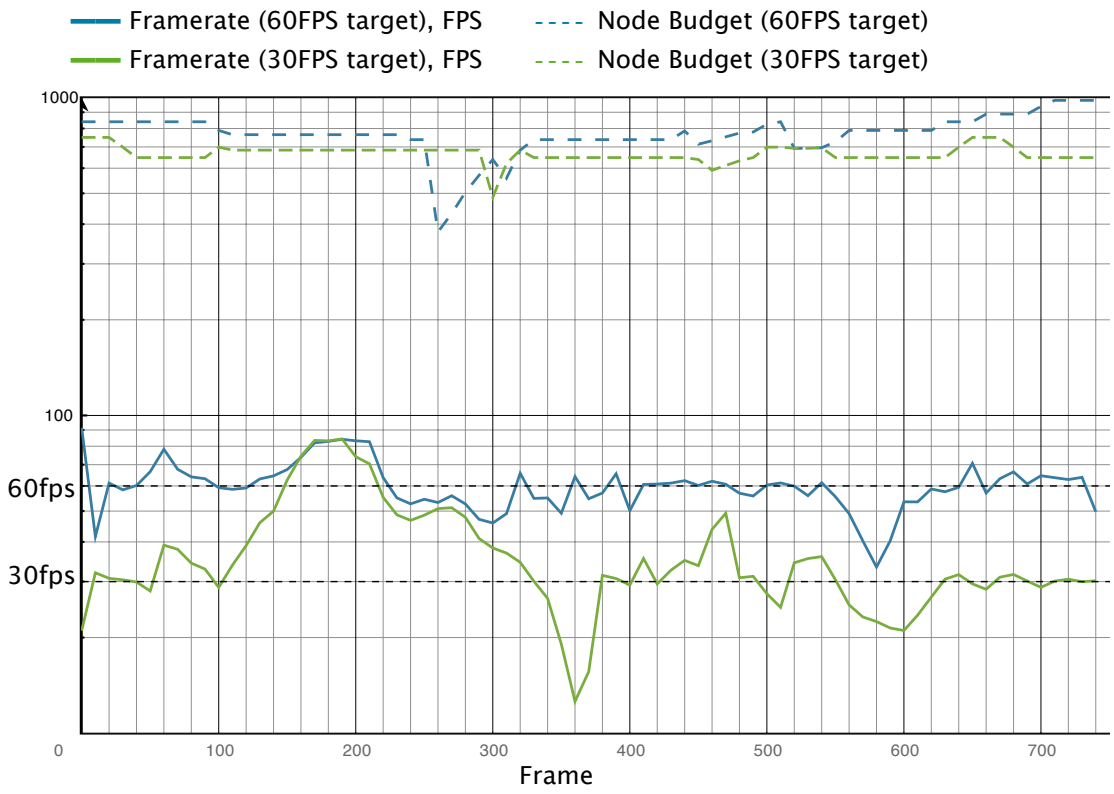

Figure 7.11: Performance Mode Node Budget: By adjusting the node budget our method uses the appropriate number of nodes to reduce triangle count without adversely impacting performance.

We have also tested the performance mode when rendering the same path over the Thai Statue model using a dual-GPU renderer on a dual-GPU computer using dual dual-core Intel Xeon 2.66Ghz CPUs with 3Gb of RAM and a NVIDIA 7800GT and 7300GT video cards. The results, shown in Figure 7.12, show that our method is capable of balancing the performance in this setting as well.

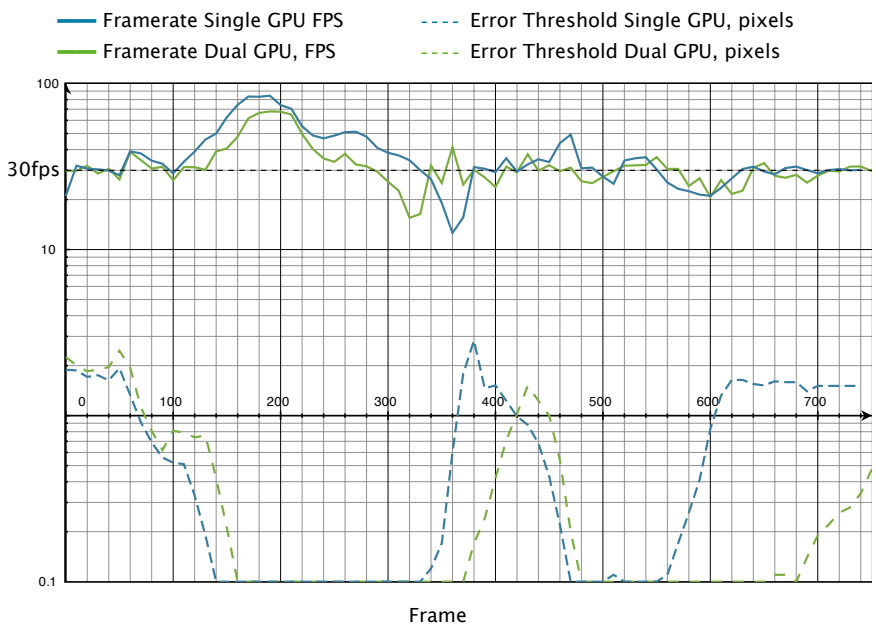

Figure 7.12: Performance Mode Dual GPU framerate: By adjusting the error threshold of both of the renderer tiles our method can adjust the performance of the rendering system to the user specified frame rate of 30Hz.

The performance mode also adjusted the node count for the dual-GPU method, reducing the number of nodes used per render tile in order to improve performance as shown in Figure 7.13. The reduction in the number of nodes helps to reduce the overhead of the X Server, allowing more geometry to be rendered.

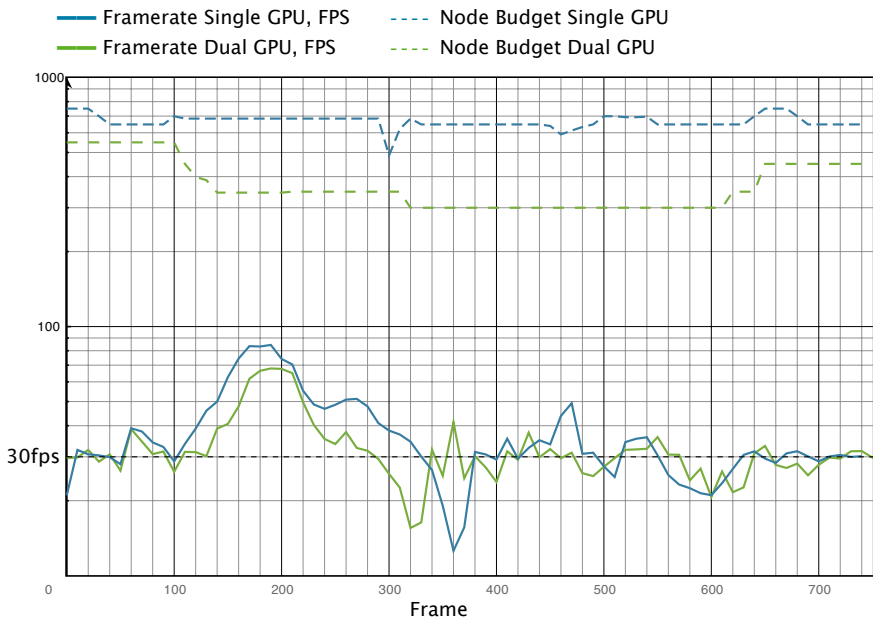

Figure 7.13: Performance Mode Dual GPU node layout: The dual-GPU version of the renderer prefers to use fewer nodes to render the hierarchy due to the increased overhead from the X Server when issuing many small GL calls from multiple threads. Results obtained from a path over the Thai statue model with node per-GPU node budgets.

The overall performance of the performance mode is summarized in Table 7.3. As the table shows our system is capable of rendering at or close to the user specified frame rate. While it is difficult to remain at a stable frame rate when rendering a path, our feedback-based method remains close to the target, especially at higher frame rate targets.

Configuration	Average FPS	Standard Deviation
Single GPU 30fps goal	35.65	13.96
Single GPU 60fps goal	60.31	10.36
Dual GPU 30fps goal	32.18	9.53

Table 7.3: Performance Mode Summary: The performance mode presented in this section is capable of maintaining a stable framerate, even when using multiple GPUs for the rendering.

7.7.3. Analysis

The goal of the performance algorithm presented in this section is to achieve a user-specified frame time. Because it operates using a feedback mechanism, its actions will always be delayed by at least one frame. As a result, a feedback based approach will always be close to the optimal framerate, but may need to continuously adapt the error threshold and node budget to stay on target.

While the use of a predictive method that estimates the rendering time of a frame before it is rendered might give a tighter bound on the framerate, it would be very difficult to adapt the hierarchy to a rendering time due to the many factors affecting rendering performance.

The adjustment of the node budget is an important part of the performance mode, allowing the geometry to be rendered at a lower geometric error while maintaining a sta-

ble frame rate. The main difficulty with adjusting the number of nodes used to render the model is its impact on the rendering time. While increasing the node budget generally lowers the triangle count, it can also decrease the frame rate due to the use of more OpenGL calls, and the issuing of smaller primitives. Thus while it might initially seem beneficial to use as many nodes as possible without making the adapt time longer then the rendering time, in a real system the increased node count is detrimental to overall performance. As a result, the performance mode will at times reduce the number of nodes used to render the geometry, as long as the reduction improves the framerate.

7.8. Distributed Rendering

The most common type of distributed rendering systems is the sort-first system which utilizes an array of displays, frequently called a tiled display wall. In this distributed rendering paradigm each rendering computer is connected to one or more displays and renders the geometry that falls in its view frustum.

The main benefit of such a cluster is the ability to render the model at very high resolution, allowing either more detail or more area of the model to be viewed at one time.

7.8.1. Distributed Setup

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. We do not currently support transporting pixel data between PCs, 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 be a more dominant component of the rendering time than it is for merely moving data over the local PCI-Express bus.

In this configuration each computer in the rendering cluster is responsible for a distributed tile, performing both the hierarchy adaptation and rendering for that part of the view frustum.

7.8.2. Distributed Communication

Our parallel rendering pipeline aims to minimize the number of synchronizations among parallel units. In both the single-machine and distributed settings, it requires only two synchronization points, one at the start/end of the frame (to synchronize the swapping of the frame buffers and transmit camera and other user parameters) and one 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 [Amir et al. 2004], which supports efficient group communication using multicast.

The network communication used in this system is used to exchange camera parameters, cut unification data and synchronization information. Our method limits the network communication to two exchanges, one at the beginning of the frame and one at the end. The first exchange sends out camera parameters to be used when adapting the next frame, ensuring that all of the renderers use the same parameters when adapting their view frustum tile. This packet is sent by the master computer that receives the user input

The second network exchange occurs after the geometry has been rendered to the back buffer. It synchronizes the framebuffer swap, forcing all of the displays to be refreshed simultaneously, reducing the flickering on the display wall. The second network exchange is also used to exchange cut unification data for the machine tiles. These cut unification data serves as the actual synchronization data, telling each computer to swap buffers upon receipt of cut unification data from every computer.

The rendering and adapt processes used in the distributed renderer are otherwise identical to their single-computer counterparts described in Sections 4 and 5.

7.8.3. Results

The system has been tested on a 4x2 tiled display wall using 8 rendering nodes and a single control node. The control node was used only for the mouse and keyboard input, with all of the adapt and rendering work being performed on the rendering cluster. The rendering cluster was composed of 8 nodes, each with a dual 2.8Ghz Xeon CPU, 2GB of RAM and a NVidia QUADRO 3300 video card with 256MB of onboard video memory. Because these video cards were incapable of rendering using vertex texturing,

the mesh-based rendering method was used to render the models. The tile layout is shown in Figure 7.14, and operates at an aggregate resolution of 5120x2048, with each tile at 1280x1024 pixels.

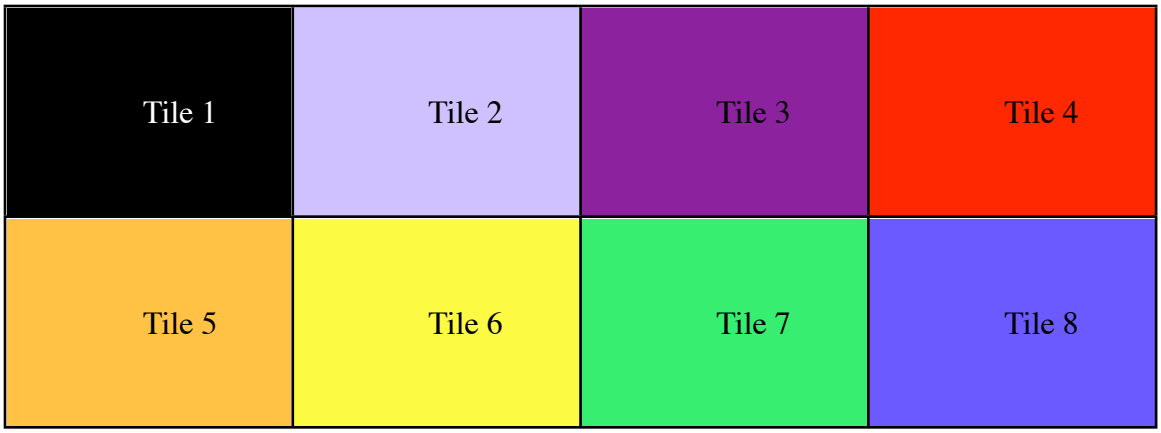

Figure 7.14: Tile layout in the tiled-wall display used in this paper. The tiled wall is powered using a homogenous rendering cluster, each driven by a dual-CPU 2.8Ghz Pentium Xeon system with GeForce Quadro 3000 GPUs

The result of rendering to the tiled display layout shown in Figure 7.14 is shown in Figure 7.15. The model being rendered is the Thai Statue model, rendered using an eight node rendering cluster running Linux. Each node is a dual-CPU 2.8Ghz Pentium Xeon based system with a NVIDIA Quadro 3000 video card.

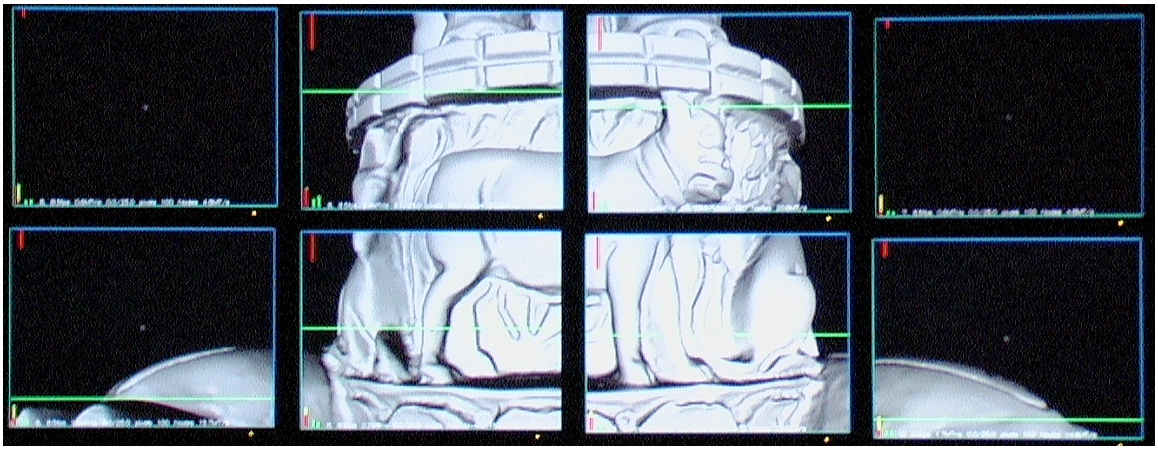

Figure 7.15: Distributed renderer. A snapshot of the distributed renderer rendering a path of the Thai statue.The tiled wall is powered using a homogenous rendering cluster, each driven by a dual-CPU 2.8Ghz Pentium Xeon system with GeForce Quadro 3000 GPUs

To test the performance of the distributed renderer we have tested it using the earth dataset along a 950 frame path, shown in Figure 7.16. The timings taken from the path ran on a homogenous rendering cluster built out of 8 rendering nodes, each with dual 2.8Ghz Pentium Xeon CPUs and a NVIDIA GeForce 6800GT. The total time and the rendering time are obtained by finding the maximum frame time for the 8 renderers. The total time includes both the rendering and the network times, as well as GPU data loading, cut unification and other timings. The network time is calculated as the minimum time from the 8 renderers. The minimum is used as some of the renderers can have little or no geometry to render, forcing them to continuously check network messages, resulting in a overestimate for the network time. The minimum function is therefore used to

estimate the time between the last rendering computer finishing rendering and the it receiving all of the synchronization messages.

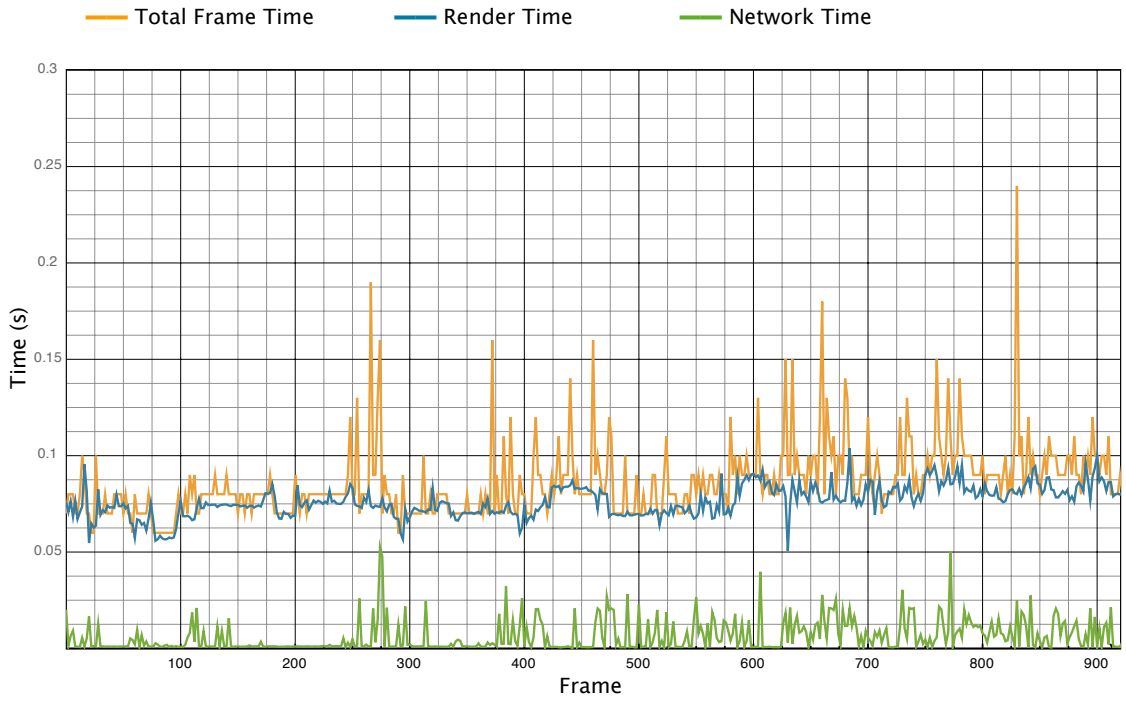

Figure 7.16: Distributed rendering timings. The timings shown in this graph are captured from a rendering cluster playing a path over the Earth Dataset. Each node in the cluster adapts its section of the display to a 2M triangle budget using 450 hierarchy nodes. As shown in this graph, the whole cluster renders at approximately 10fps on average.

The timings shown in Figure 7.16 demonstrate the relatively low overhead of our system: in general, the rendering time constitutes over 91% of the frame time. The synchronization overhead for this path was approximately 7% of the frame time, leaving only 2% of the frame time for other per-frame operations.

To compare the rendering performance of the cluster to the rendering performance of a single node we ran the same path over the Earth Dataset using a single node from that cluster, with the results shown in Figure 7.17. As shown in those results the maximum rendering time of the distributed system is very similar to the frame time of the single node renderer.

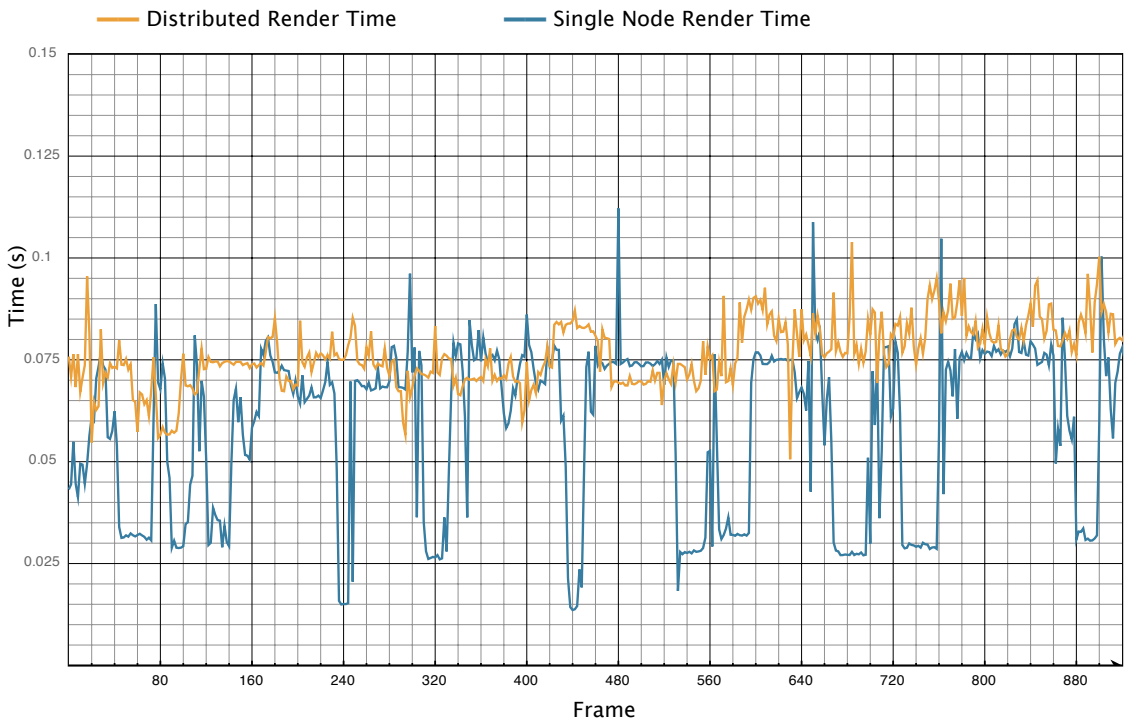

Figure 7.17: Distributed vs single node frame time. By comparing the maximum frame rendering time of the distributed system versus a single node, we can see that the rendering performance of the two renderers is roughly equal. Results obtained from a path over the Earth dataset using a 2M triangle budget.

We have also tested the impact of using different layouts of the distributed renderer on the synchronization overhead. We have tested the distributed renderer on a variety of configurations, using two, four and eight rendering nodes, with the results shown

in Figure 7.18. While the network times frequently spike due to various network congestion, the results shown in Table 7.4 show that the synchronization overhead can be reduced by decreasing the number of nodes in the system. Although the synchronization overhead exists, it is relatively small at approximately 6ms for a 4x2 tiled display wall. The resulting synchronization overhead reduces the performance of the overall system, with the exact reduction dependent on the target framerate. For example, if the system is rendering at 10 frames per second, the synchronization overhead reduces the performance by only 6%. As the framerate increases, so does the penalty: at 30 frames per second the synchronization would require 18% of the frame time.

The synchronization overhead shown in Figure 7.18 and Table 7.4 shows a large amount of noise. One of the main causes of the non-uniform network timings is the use of network-attached storage to store the dataset. This causes the network synchronization to compete with the data management for the access to the network, reducing the overall performance of the synchronization.

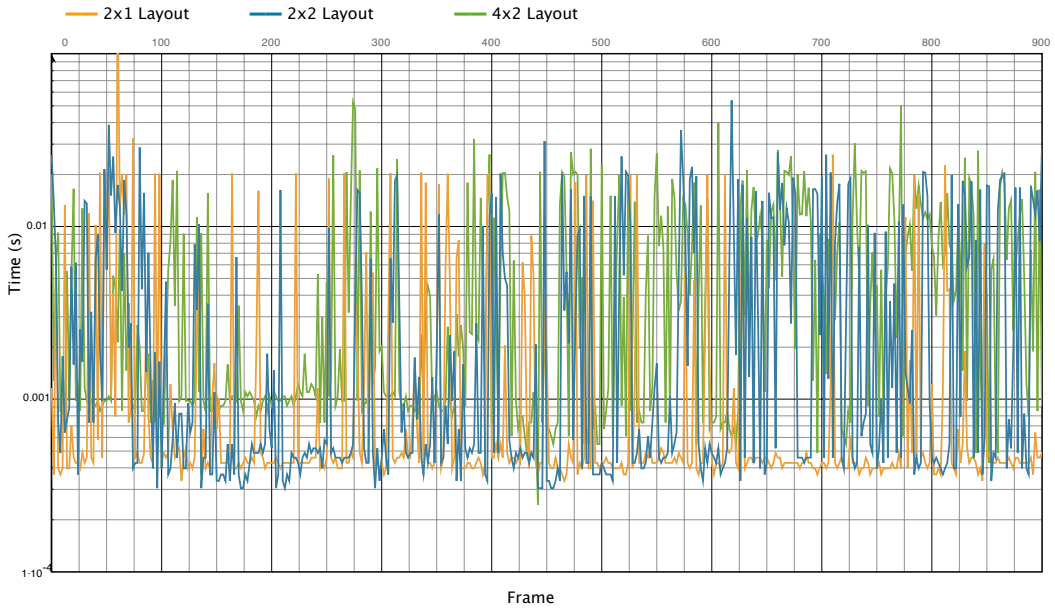

Figure 7.18: Synchronization Overhead. The synchronization overhead experienced when rendering using the rendering cluster is shown in this graph. While the synchronization overhead varies significantly in-between frames, it steadily increases as more rendering nodes are added to the distributed system.

Layout	Average Synchronization Overhead (ms)	Standard Deviation (ms)
4x2	6.3	8.1
2x2	4.9	6.7
2x1	2.7	10.1

Table 7.4: Synchronization Overhead Summary: The synchronization overhead of the distributed renderer increases as more renderer nodes are added to the system. This is expected, as more synchronization and cut unification packets are sent per frame, requiring more time to deliver.

To test the overall performance of the system we have compared the overall throughput and error level of the final rendering when rendering using the distributed system and a single node. The results for a path over the Earth dataset are shown in Figure 7.19. While the single node is generally faster at rendering the geometry then a individual computer in the cluster, the distributed system uses 8 rendering nodes in parallel to create the final output. As a result, the overall throughput of the distributed system increases from an average of 20M triangles/second for the single renderer to an average of 100M triangles/second for the distributed renderer. The five-fold increase in throughput helps to overcome the eight-fold increase in screen area, allowing the distributed system to render a 5120x2048 buffer at the same error as the single node renderer.

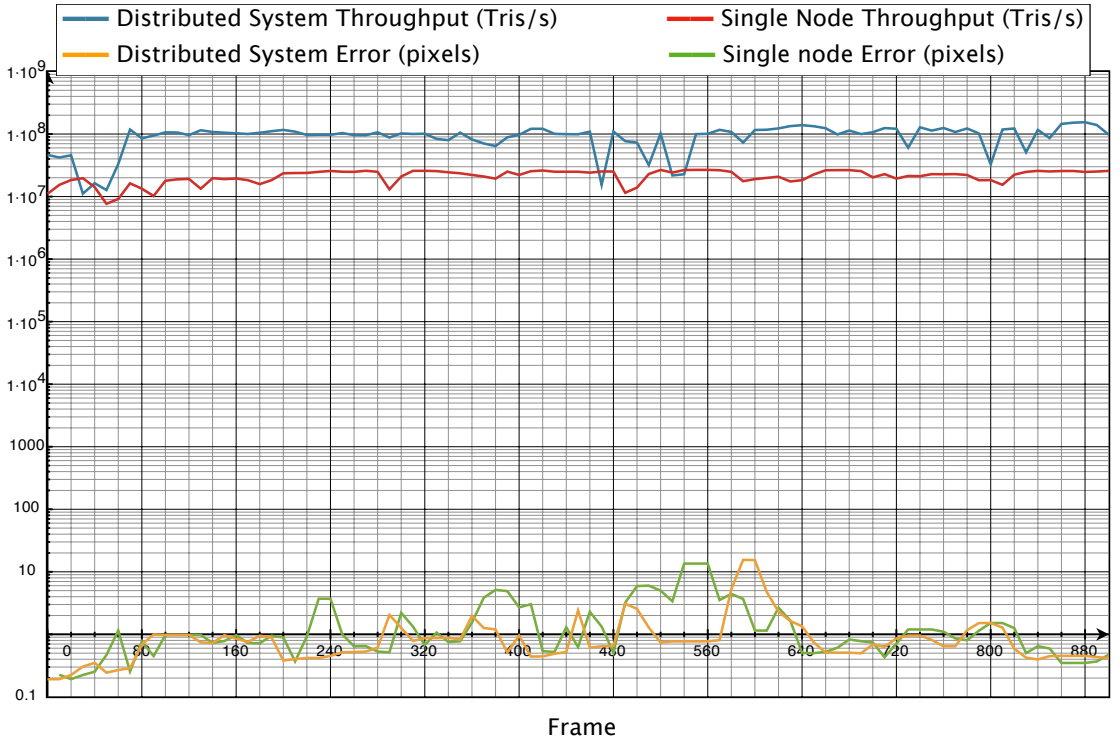

Figure 7.19: Distributed vs single node throughput: The use of a rendering cluster greatly improves the aggregate throughput of the system, allowing the distributed renderer to render a 5120x2048 image at sub-pixel error for most of this path over the Earth dataset. All of the results were obtained on a rendering cluster with dual 2.8Ghz Pentium Xeon CPUs and NVIDIA GeForce 6800GT video cards.

7.8.4. Analysis

During the design of the distributed rendering system several important design choices were made in order to improve the performance of the system. The key design

choice made was the minimization of network communication between the nodes during the adapt and rendering process.

The first, and most important, choice was the use of a multi-display sort-first renderer instead of a single-screen sort-first approach. This greatly reduces the network communication overhead, as no data is sent back from the render nodes back to the control node. The use of the multi-display sort-first method does, however, have disadvantages, such the lack of load balancing abilities between the rendering nodes. Since the rendering regions are fixed for the rendering nodes, they cannot be resized to balance out the workload of each renderer. The LOD system therefore becomes the only means of balancing the workload on the distributed system.

The reduction of network communication overhead is also accomplished through the use of a single interchange of information during the adapt process, reducing the latency of the rendering. This is a huge benefit to a rendering system, as any network communication during the rendering process greatly reduces the framerate of the system because of the network latency and limited network throughput.

While this increases the overall performance of the system, it forces a more conservative merge process, which in turn causes the overall geometric error in the scene to increase when using the triangle budget adapt method. The screenspace error method remains largely unaffected by this problem since all of the nodes should start with the same geometric error, and the only difference is the subdivision of geometry nodes in the various screen regions.

This rise in the geometric error is caused by the forced coarsening of geometry nodes which overlap multiple screen regions in order to match the resolution of the other rendering nodes. Although any consistent algorithm for the merging process could be used, such as refining to match neighbors, we have decided to coarsen to match in order to always stay under the triangle budget. Furthermore, refining the border nodes would still raise the overall geometric error, since other nodes would have to be coarsened in order to bring the screen region under the triangle budget set by the user.

7.9. Discussion

The use of the tiled-based subdivision of the view frustum presents several benefits to the distributed system. The main advantage of such a system is the ability to combine the various distributed elements of the presented method in single system. As a result, the system can use a multi-core CPU to drive a multi-GPU rendering node belonging to a multi-renderer distributed rendering system. The flexibility afforded by the tiled-based system allows any subset of such a system to be created, allowing each individual node to perform workload balancing so that the work is divided equally amongst each element of the system.

The main disadvantage of the tiling approach is the increase in duplicate processing stemming from overlapping geometry. As the number of tiles increases so does the amount of duplicate work, reducing the overall efficiency of the system. While our adapt mechanisms attempt to minimize the amount of overlap by subdividing geometry near boundaries, the overlap can never be eliminated.

7.10.Summary

The three distributed methods presented in this chapter decompose the final frame-buffer into tiles, each level of which is independent of its parent. For example, the adapt tiles subdivide a rendering tile, which in turn subdivides a distributed tile. As a result, the tiles can be resized independently of each other without affecting the workload of their parents.

The tiling system is also capable of operating with any combination of adapt, rendering and distributing tiles. This in turn allows for a wide variety of machines to be used, including multi-core, multi-GPU distributed systems, or any subset thereof. The presented system is therefore capable of adjusting to a variety of systems, and is capable of utilizing the various advantages of these machines.

The tiling algorithm used also allows for easy expandability due to the use of a KD-style screen subdivision algorithm. This greatly simplifies the addition of additional GPUs or CPUs without increasing the complexity of the rendering system, or introducing additional overhead or latency which is present in the alternate frame rendering systems.

8.Conclusion

This level of detail system introduces two new major contributions to the Level of Detail field: The ability to increase and decrease the node size and the number of nodes without altering the triangle count to balance the work of the CPU and GPU, and the ability to perform the LOD operations in a distributed setting, where multiple computers are responsible for rendering of the final frame.

We have presented a series of data structures and algorithms, as well as a prototype system, that explore a number of ideas and trends for high performance rendering on evolving graphics hardware. Our approach incorporates ideas from seamless geometry atlas parameterization, geometry image resampling, and quad-tree-based hierarchical rendering to expand on the state-of-the-art in level of detail systems. The resulting multi-grain, multi-hierarchy system builds on the previous work by introducing a multi-resolution element to the adapt process expanding the adaptation options for every node, allowing them to not only split/merge, but also to refine/coarsen.

This added flexibility allows for not only better adapt performance, reducing the triangle count for a given error threshold, but also for the balancing of the CPU and GPU workloads. As shown in our results, through adjusting the node and triangle budgets the user can control the amount of work performed by the CPU and the GPU, respectively. The flexibility of the system allows it to be more adaptable to the hardware configuration of the host and to reduce the triangle to error ratio by adaptively selecting node resolutions. By adaptively selecting resolutions our system can produce more detailed meshes using fewer triangles then previously possible on a fixed node budget.

The flexibility of the adapt algorithm and the border merging process also enables our method to operate in a distributed setting. By allowing arbitrary node resolution and hierarchy depth neighbors our method removes many of the obstacles for a distributed renderer. Furthermore, the predictability of our method allows it to perform the adapt process with a single stage of communication, greatly reducing its latency, and maximizing rendering throughput. As a result our method is uniquely capable of rendering huge datasets in distributed rendering environments, providing increased performance to the user.

Our system also 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 easily 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.

The system presented in this paper also expands the abilities of modern Level of Detail systems to operate in parallelized settings, both local and distributed. By being aware of and utilizing these new resources our system remains remarkably future-proof, with the ability to expand to a variety of multi-CPU, multi-GPU and distributed configurations.

Finally, a number of additional features common to high-performance rendering systems could be incorporated with our proof-of-concept implementation in the future to

produce a system which is both very flexible and has even better performance. These include features such as accurate data pre-fetching, data compression, back-patch culling, geomorphing and occlusion culling. The resulting system would be capable of fully harnessing the power of the latest GPUs, allowing arbitrarily large models to be rendered at interactive rates. In our opinion this approach represents a very promising visualization technique due to its very flexible adapt algorithm, very fast preprocessing, GPU-friendly data representation and easy extensibility.

9.Bibliography

Amir, Yair, 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.

ATI. ATI Crossfire technology white paper. Technical report, ATI Technologies, 2005.

Bethel, E. W., 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.

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

Borgeat, L., G. Godin, F. Blais, P. Massicotte, and C. Lahanier. GoLD: interactive display of huge colored and textured models. *ACM Trans. Graph.*, 24(3):869–877, 2005.

Chhugani, Jatin, Budirijanto Purnomo, Shankar Krishnan, Jonathan Cohen, Suresh Venkatasubramanian, David S. Johnson and Subodh Kumar. vlod: High-fidelity walkthrough of large virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 11(1):35–47, 2005. Member-Subodh Kumar.

Cignoni, P., F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive Tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH*, 2004.

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

Cohen, Jonathan, Mark Olano and Dinesh Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM Press.

Corréa, W. T., 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.

Crow, F. C. A more flexible image generation environment. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 9–18, New York, NY, USA, 1982. ACM Press.

El-Sana, Jihad and Amitabh Varshney. Parallel processing for view-dependent polygonal virtual environments. In *Proc. SPIE Vol. 3643, p. 62-70, Visual Data Exploration and Analysis VI, Robert F. Erbacher; Philip C. Chen; Craig M. Wittenbrink; Eds.*, pages 62–70, Mar. 1999.

Erikson, C., D. Manocha, and W. V. B. III. Hlods for faster display of large static and dynamic environments. In *ACM Symposium on Interactive 3D Graphics*, pages 111–120, 2001.

Floriani, L. D. A pyramidal data structure for triangle-based surface description. *IEEE Comput. Graph. Appl.*, 9(2):67–78, 1989.

Fuchs, H., 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.

Funkhouser, Tom, C. H. Séquin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 11–20, New York, NY, USA, 1992. ACM Press.

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

Garland, Michael and , Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

Garland, Michael and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 263–269, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

Gu, X., S. J. Gortler, and H. 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.

Heirich, A., and L. Moll. Scalable distributed visualization using off-the-shelf components. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 55–59, New York, NY, USA, 1999. ACM Press.

Hoppe, Hugues. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23th annual conference on Computer graphics and interactive techniques*, pages 99–108, 1996.

Hoppe, Hugues. 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.

Humphreys, G., 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.

Lindstrom, Peter and Greg Turk. Image-driven simplification. *ACM Trans. Graph.*, 19(3):204–241, 2000.

Lindstrom, Peter. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

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

Losasso, F., H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 138–145, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

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

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

Luebke, David. *View-Dependent Simplification of Arbitrary Polygonal Environments*. PhD thesis, 1998.

Michael, J., J. DeHaemer and J.M. Zyda. Simplification of objects rendered by polygonal approximations. In *Computer & Graphics*, volume 15, pages 175–184, 1991.

Molnar, S. *Image-composition architectures for real-time image generation*. PhD thesis, Chapel Hill, NC, USA, 1992.

Molnar, S., J. Eyles, and J. Poulton. Pixelflow: high-speed rendering using image composition. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240, New York, NY, USA, 1992. ACM Press.

Molnar, S., M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

Mueller, C. The sort-first rendering architecture for high-performance graphics. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75–ff., New York, NY, USA, 1995. ACM Press.

Niski, Krzysztof, Budirijanto Purnomo and Jonathan D. Cohen. Multi-grained Level of Detail Using a Hierarchical Seamless Texture Atlas. In *SGP '04: Proceedings of the 2007 symposium on Interactive 3D graphics*, Seattle, WA, USA, 2007. ACM Press.

Purnomo, Budirijanto, 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.

Rottger, S., W. Heidrich, P. Slusallek, and H.-P. Seidel. Real-time generation of continuous levels of detail for height fields. In *International Conference in Central Europe on Computer Graphics and Visualization*, pages 315–322, 1998.

Samanta, R., T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, New York, NY, USA, 2000. ACM Press.

Samanta, R., T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 75–84, Piscataway, NJ, USA, 2001. IEEE Press.

Sander, P. V., Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

van der Schaaf, T., L. Renambot, D. Germans, H. Spoelder, and H. Bal. Retained mode parallel rendering for scalable tiled displays. In *7th annual Immersive Projection Technology (IPT) Symposium*, 2002.

Schroeder, W. J., J. A. Zarge and W. E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 65–70, New York, NY, USA, 1992. ACM Press.

Schwan, Philip. Lustre: Building a file system for 1000-node clusters, 2003.

Turk, Greg. Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 55–64, New York, NY, USA, 1992. ACM Press.

Xia, J. C. and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Visualization*, pages 327 – 334, 1996.

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

Vita

Krzysztof Niski was born in Warsaw, Poland on November 6th, 1980. He did his early schooling in Island School, Hong Kong, International School of Kuala Lumpur, Malaysia, and finally finished at the International School of Geneva, Switzerland, obtaining an International Baccalaureate in 1999. He obtained his Bachelors Degree in Science in 2002 from Shepherd College in West Virginia. From 2002 until 2007 he was a graduate student at the Johns Hopkins University where he obtained his Masters Degree in 2005 followed by his PhD in 2007. His research interests include realtime visualization, graphics hardware processing and digital image processing.