APPROVAL SHEET

Title of Thesis: Rendering Massive Models

Name of Candidate: Mark A. Bolstad
Doctor of Philosopy,
2017

Thesis and Abstract Approved: _____

Dr. Marc Olano
Associate Professor
Department of
Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of Thesis: Rendering Massive Models
Mark A. Bolstad, Doctor of Philosophy, 2017

Thesis directed by: Dr. Marc Olano, Professor
Department of Computer Science and
Electrical Engineering

Whether in quest of increased visual realism in cinema, or in processing the latest scientific data sets, the sizes of models being rendered are ever increasing. Trends in recent films have models that exceed hundreds of millions of elements. Similarly, the datasets in scientific visualization are approaching an exabyte in size. As these trends continue, new methodologies will be required to render these sizes of datasets.

In order to reduce the geometric complexity that is processed, many renderers employ a number of tricks to ensure that the number of individual objects fits within a 32-bit integer. The total number of elements generated for a typical complex scene can easily exceed the capacity of a 32-bit integer, but approaches that allow for memory reuse keeps an upper bound on the number of objects processed by the renderer.

This dissertation presents methods for rendering of models and scenes that exceed the number of elements repesentable in an 32-bit integer and

the memory size of any single processor. These contributions include new parallel algorithms for rendering large models, and a new stochastic data structure to reduce node-to-node communication during rendering.

Rendering Massive Models

by

Mark A. Bolstad

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

I dedicate this dissertation to my lovely wife, who despite all the advances and set backs, insisted I graduate "soon", or suffer a most painful death.

ACKNOWLEDGMENTS

First, I would like to thank and express my most sincere gratitude to my advisor, Dr. Marc Olano, for his invaluable support, encouragement, patience, and guidance throughout my PhD. Over the years, I appreciate Dr. Olano's understanding of how life and work would interject itself in unexpected ways. Through his guidance, the hurdles were cleared and the research proceeded

I would also like to thank the companies and institutions that I have worked for during the development of my research. Raytheon, the Army Research Laboratory, and the Howard Hughes Medical Institute have all provided encouragement, moral and financial support. And, without the gracious granting of time on the large computational systems, some of the results of this dissertation would not have been realized.

Additionally, I'd like to thank Aspen Computer Systems, who, after having a casual discussion at Supercomputing, sent an eight core, dual CPU server blade to my house with no strings attached. If only they had sent sound proofing to go with it. I'd also like to thank Side Effects Software who have provided me with a license to their animation package, Houdini, for years. Using Houdini, I generated most of the test data used to drive development.

Most importantly, I would like to express my most sincere appreciation to my family, especially my wife Cheryl Bolstad and my children, Miranda and Parker for all their love, support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

As audiences become more sophisticated, they demand increasingly higher fidelity from the visual effects in movies and television to continue the illusion of reality. For example, in the movie *The Chronicles of Narnia*, the character Aslan the lion had over 20 million hairs in his mane. Each individual hair was a spline curve consisting of six control vertices. If you assume a minimum of five line segments per hair, Aslan's mane had a geometric complexity exceeding 100 million renderable objects. In a more recent film, the model of the USS Yorktown in *Star Trek Beyond* consisted of 832 buildings instanced across the model space. While the on-disk size of the building geometry was in the 10s to 100s of millions of polygons, when expanded by the renderer, the final geometry count was equivalent to 1.3 trillion polygons.

Similarly, datasets generated from simulations by supercomputers are generating petabytes of data. In scientific visualization, the largest data set openly available is the isosurface of the development of a Richtmeyr-Meshkov instability. The complexity of this isosurface is over 480 million polygons. These sophisticated simulations can involve millions to billions of computation cells e.g., a recent simulation of the Saudi Arabian peninsula oil fields exceeded a trillion computational cells. Traditionally, these simulation are visualized on a separate GPU cluster for interactive viewing, but many recent simulations

either exceeded the capabilities of the GPU nodes, or the amount of time to move the data was prohibitive. Therefore more simulations are being visualized in-situ using rendering algorithms typically reserved for photo-realistic rendering, such as ray-tracing. Additionally, there is a trend to try to generate realistic images from simulations using techniques developed for visual effects such as shadowing and advancing lighting, as these images tend to provide a quicker understanding of the problem to lay audiences.

Research into rendering these large models over the last couple of decades has focused on ways to reduce the complexity of the data before it is rendered. However these techniques do not address the parallel problem of the increasing density of our displays. In 2006, the highest desktop display systems were in the 1-2 Megapixel range. Only a limited number of movie theaters were able to project using the then newly-available 4k projections systems (3840 x 2160 pixels for a density of 8.3 megapixels). Supercomputer centers were building display systems by tiling either LCD panels or projectors into systems that had pixel counts approaching 100 megapixels. Currently, it is not uncommon for desktop displays to consists of multiple 4, 5 or 8k monitors. Most movie theaters use 4k or 8k displays, and large format display walls are approaching a billion pixels. What rendering techniques will work as these display sizes continue to increase?

1.1 Motivation

For many years, the standard rending technique for visual effects was the Reyes algorithm (sometimes written as REYES for Renders Everything You Ever Saw) [15] due to its ability to handle large amounts of geometry and textures and its ability to describe appearance through programmable shaders. Ray tracing [96], renowned for its ability to compute advanced lighting simulations (reflection, refraction, multi-bounce illumination), was generally thought to be too slow for visual effects work. Visualization research is primarily

focused on gaining insight into the data through interactive rendering techniques. However, a few groups such as the National Center for Supercomputing Applications (NCSA) [41] and the Australia National University [95], are exploring how higher fidelity rendering can be used to provide additional insight into scientific datasets.

The advantage that the Reyes algorithm had over other rendering techniques was its ability to limit memory growth by processing small parts of the scene at a time, only loading the parts that needed to be rendered for a small group of pixels. Figure 1.1 provides an overview of the Reyes algorithm. Initially, the objects in the scene are sorted into small, overlapping groups of pixels, called buckets. The algorithm then chooses a bucket, and for each object in that bucket, if its screen projected pixel footprint exceeds a threshold, splits the object into smaller objects. Each of these objects' footprints is tested against the current bucket. Objects that overlap the bucket are retained and the procedure repeated. Those that don't overlap are added to the first bucket that contains the object's upper left corner. Objects that pass the threshold test are passed down the pipeline to be lit, shaded, and textured before being finally output as a set of pixel values.

Research continued to make improvements to ray tracing based algorithms. In the 90's and early 2000's, researchers from the University of Utah [66, 67] and Saarland University [85, 89, 91] showed that under certain conditions and datasets, ray tracing could approach interactive frame rates. Over the next decade, ray tracing research continued to make improvements in performance and visual quality at a lower computational cost. These improvements led to a revolution in visual effects such that over the last several years, ray-tracing has supplanted Reyes as the dominant algorithm.

During the same time period, visualization researchers began applying the improved ray-tracing algorithms to datasets from large supercomputer simulations. Prior to these improvements, the typical workflow for visualizing data from large simulations was to run the simulation on a dedicated supercomputer and then transfer the data to a much smaller

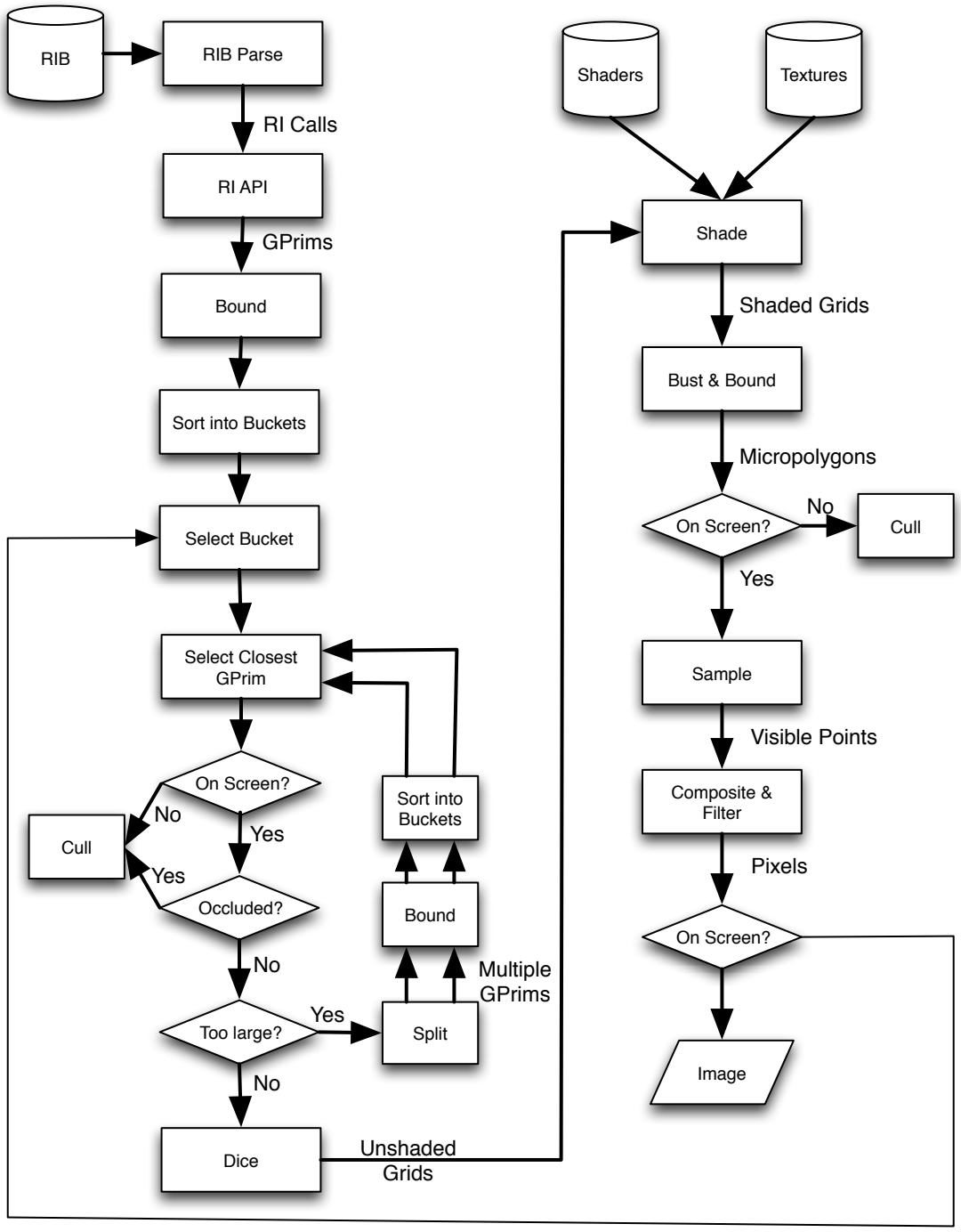

FIG. 1.1. Flow chart for the Reyes algorithm

Graphics Processing Cluster (GPU Cluster). If the datasets were large, then a reduction operation might be applied before transferring to reduce the size of the dataset. With the improvements to ray-tracing, the data could be visualized in-situ, directly on the nodes of the supercomputer. This freed up the GPU Cluster for other visualization work, or in some cases, allowed a center to reduce cost by removing the visualization cluster altogether.

There is, however, a problem with ray-tracing: to render all of the advanced lighting methodologies mentioned earlier, all of the data has to be resident in memory. In ray-tracing, rays are propagated from the camera though each pixel in the image plane and tested against the objects in the scene to determine which objects are intersected by the ray. If we only use this first set of rays, the algorithm is called ray-casting. This is the variant used in the in-situ visualization on supercomputers. These rays are all coherent, so a group of rays that access a neighborhood of pixels in the image plane will have little divergence from each other as they trace the scene. If we add reflection or shadow rays (secondary rays), these rays become incoherent, i.e., small changes in the initial direction can lead to large changes in the secondary rays. As shown in Figure 1.2, a small 16x16 bundle of rays shot from the front left corner to the upper right corner can access nearly every object in the model upon the second bounce.

In this dissertation, we present three significant improvements to existing rendering techniques for large models. We define a large model as either a single or composite set of objects whose total number of renderable elements exceeds four billion individual items. A renderable element can be a simple triangle or quadrilateral, a quadric surface such as a sphere or a cone, or a more complex object such as a parametric or subdivision surface. The key areas identified are:

1. A fully distributed, parallel implementation of the ray tracing algorithm

2. Deferred loading and communication of scene data through an innovative stochastic

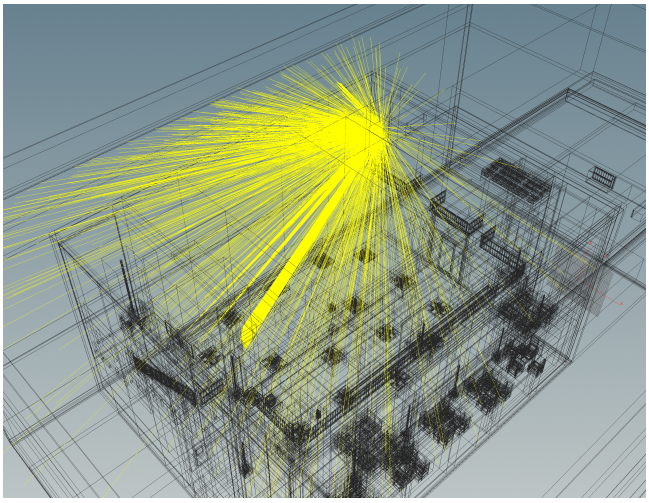

FIG. 1.2. A small coherent group of primary rays traced from the camera can spatially traverse nearly the entire scene volume on a second or higher bounce

map of hit probabilities

3. Handling large models everywhere in the rendering pipeline

We believe that research and innovation in these three areas is necessary to advance the rendering of large models. Advancements in each one of the areas will enable the rendering of larger models than current systems are capable of rendering today, simultaneous advancement in all the areas will enable the rendering of models beyond the capabilities of hardware and software systems predicted over the next five years.

1.2 Problem Statement

As audiences become more sophisticated, they demand more fidelity from the visual effects in movies and television to continue the illusion of reality. Similarly, datasets generated from simulations by supercomputers are generating petabytes of data. How do we render such large collections of data that may exceed the memory capacity of the computer or cluster?

To answer that question, several advances must occur simultaneously. First, a system must be developed that is designed specifically for rendering large models by optimizing for memory usage. Second, a method for deferring loading and unloading of objects through proxies is needed to reduce the memory footprint of the loaded scene. And finally, a method for fully distributing the entire rendering system, e.g., the full advanced lighting pipeline, not just primary rays from the camera, across a large heterogeneous cluster of nodes/systems.

1.3 Contributions

This dissertation contributes to the rendering of large models by advancing the research in several key areas: *Parallel Micropolygon Rendering, Parallel Rendering*, *Deferred Communication and Geometry Eviction*, and *Designing a Rendering Pipeline for Large Models*.

1.3.1 Parallel Micropolygon Rendering

The first main contribution of this dissertation is an examination of the methods for the parallel rendering of geometry for a micropolygon renderer, like Reyes. Micropolygon rendering is the process of tessellating input geometric primitives into polygons typically of less than one pixel in size. Micropolygon rendering was first described by Cook et al. [15] and has been used to render complex scenes for motion pictures for over 25 years. Over that period this algorithm has been shown to be stable and robust when rendering large, geometrically complex scenes. We investigate parallel extensions of a micropolygon renderer and show that none of the common work distribution methodologies are appropriate for all scene compositions. While the results are strictly applied to a micropolygon renderer, they are applicable to any rendering algorithm that processes the scene in small, limited parts,

such as a ray caster using a Hilbert curve for screen traversal.

1.3.2 Parallel Ray Tracing

Another contribution of this dissertation is a fully distributed, parallel version of the ray tracing algorithm. Since each ray can be traced through the scene independently of other rays, the algorithm is trivial to parallelize. This is under the assumption that a ray has access to all scene objects which is not the case when those objects have to be distributed across a group of nodes as the scene is too large to fit into the memory of a single system. We relax that assumption by representing all objects with proxies, and only when the proxy is intersected by a ray is the data loaded. The file I/O places a heavy burden on the system so we parallelize the processing, partitioning, and distribution of the scene description as it is being loaded from data store. Each server in the parallel system renders its portion of the scene, and when rays intersect objects outside of the server's domain the rays are queued and distributed directly to the server responsible for the object. We show that with this fully distributed architecture scaling increases with the number of servers.

1.3.3 Deferred Communication and Geometry Eviction

Communicating rays across the nodes of a distributed system places a heavy burden on the networking subsystem such that it is easy to overwhelm it. We spatially distribute the scene amongst the servers as our target scenes are larger in memory than a single server. However, as shown in Figure 1.2, a small group of rays can quickly force communication among all the servers. If we use an illumination algorithm that gathers light from multiple ray bounces, the amount of communication is exacerbated. We propose a unique stochastic map that reduces the number of rays transmitted. The map stores a probability of intersection with the actual geometry over incoming ray directions. Since all geometry is represented by a proxy object, in this case a bounding box, we associate the probablity

map with the proxy. During rendering, when a ray intersects the proxy object, the ray is tested against the map and if the value for that ray direction is above or below a bimodal threshold we retrieve a number of parameters for the geometry, update the ray, and return it. Otherwise the ray continues as described above with the map updated by the value returned with the ray. As the render progresses, objects that have been loaded will be removed from memory if they weren't recently accessed. This contribution reduces the memory footprint of the scene, such that the proposed systems can render models much larger than the available system memory. We show that with this modification the scalability of the system increases at the cost of additional variance in the result.

1.3.4 Designing a Rendering Pipeline for Large Models

Designing a rendering system is a complicated undertaking. Complex rendering algorithms, data representation and mathematical operations on a variety of geometric shapes, hierarchical representations of these shapes in the scene, the "look" (i.e., glass, fabric, etc.) of those objects, and other factors, all contribute to a significant effort of software engineering. Much of the research into rendering is focused on either performance or improving the quality of the final image. Very little effort has been focused on ensuring that rendering systems are engineered for extremely large models. Two examples:

1. Throughout the rendering pipeline, developers use indexing to access parts of the scene, such as an individual element of a collection, or a vertex, color, or normal, from a shared list. A majority of this indexing code will use an integer data type (or smaller) for the index, a type that is chosen for convenience rather than based on analysis.

2. For agglomerated objects like triangle and polygonal meshes, the representation of these objects will use lists for the components that describe the object. For meshes,

these components include points, normals, colors, edge connections, and other elements in quantities corresponding to the number of points, faces, or, one per object. In C++, these elements are often stored in a *vector* from the Standard Template Library. The size of *vector* is typically implemented as three 8-byte pointers + size of the element * capacity. For efficiency, the capacity of a vector can be up to twice the actual number of elements stored to minimize reallocations. If we have a scene with one billion elements, then a lower bound on storage due to *vector* overhead is an additional 24 gigabytes of memory for each component stored, e.g., points, normals, and colors.

We present a rendering system that is designed to scale efficiently with the size of the scene. We present the design decisions necessary to achieve this goal, and the analysis that we performed to support them.

1.4 Dissertation Roadmap

The rest of the dissertation is organized as follows:

- **Chapter 2: Previous Work**

  This chapter presents an overview of existing techniques that have been used to process large models and render them. It also discusses the general background on parallel rendering techniques.

- **Chapter 3: Parallel Methodologies for a Micropolygon Renderer**

  This chapter presents four algorithms for parallelizing an implementation of the Reyes micropolygon renderer. We show that none of the algorithms work for every scene type, but reasonable speedups occur for the appropriate scene type.

- **Chapter 3: An Algorithm for Distributed Ray Tracing**

This chapter presents an algorithm for parallelizing two variants of ray tracing. We show how we parallelize the loading of objects from disk and distribute them amongst the servers and the communication framework and patterns to connect the servers together.

- **Chapter 4: Deferred Communication and Geometry Eviction**

  This chapter demonstrates a novel technique for reducing the communication and data loading through maps of the probabilities of rays intersecting objects.

- **Chapter 5: Designing a Rendering System for Massive Models**

  This chapter demonstrates some of the design decisions and analysis required to engineer a system for rendering large models.

- **Chapter 6: Conclusion and Future Work**

  This chapter concludes our work, highlights our contribution, and lists a number of extensions and future work we plan to consider in the near future.

Chapter 2

BACKGROUND

Rendering is the process of taking a three-dimensional description of geometry, its material properties, and the environment and projecting them through a camera onto a two-dimensional plane. The material properties of the geometry can be described through a library of materials or parameters (e.g., glass, plastic, etc.) that provide the necessary information about the material's color, reflectance, and absorption to a renderer. Many physically-based renderers use this approach to provide control over the types of surfaces rendered. The other possibility is to describe a surface's properties algorithmically in a program called a shader. Offering greater flexibility than the library based approach, shaders are used for describing the appearance of the object and recently, with correctly simulating the physics of illumination. While shading is important for the final appearance of an image, a good design for a renderer separates the handling of geometry from its appearance. Therefore, the rest of this dissertation will focus on the geometry handling portion of the rendering pipeline and will discuss shading as appropriate in the context of management of large amounts of data.

There are number of approaches that are commonly taken when trying to render large models. These techniques fall into several categories: reduction in complexity, data reorganization, and brute force techniques. Each of these techniques will be discussed indivi-

sually in the sections below.

2.1 Simplification

One of the first methods developed by researchers to deal with the increasing size of models was simplification. As the name implies, simplification is a technique where an input model, usually polygonal, is reduced in complexity through reduction in the number of vertices, edges, and/or faces, to reach a specified complexity. Several different algorithms have been developed for simplifying surfaces. In vertex decimation or vertex removal, the algorithm iteratively selects a vertex for removal, removes the adjoining faces, and re-triangulates the resulting hole [75]. Turk [79] developed a method that reduces the amount of detail in a model while preserving its topology by distributing points on the surface of the original model, triangulating the model using both the old and generated vertices, and then removing the old vertices. This technique allows for the precise control in the number of vertices in the final model, and for control over the sampling density such that regions of high curvature are sampled more densely. Another technique is vertex clustering [72], in which the original model is placed within a grid, vertices within each grid cell are clustered into a single vertex, and the model faces are updated accordingly. Iterative edge contraction updates the model by selecting an edge and removing the adjacent triangles. These early vertex decimation techniques blindly removed vertices with out regard to the underlying topology occasionally resulting in deformed models.

A version of the iterative edge collapse algorithm by Hoppe et al. [40], re-casts the simplification problem into the larger framework of optimization. Using the initial surface and a set of points generated randomly on that surface, the algorithm minimizes an energy equation first over vertex positions, then over simplical complexes that are homeomorphic to the original mesh. The first phase of the algorithm generates an optimal set of vertices

for a fixed simplical complex, and the second phase uses either an edge collapse, split, or swap to minimize the energy equation over a set of simplical complexes. By using the Gaussian curvature of the surface to control the initial distribution of points, the algorithm is able to preserve regions of high curvature.

The majority of simplification algorithms are designed for manifold surfaces, surfaces for which the neighborhood of every point is topologically equivalent to a disk. This restriction allows the algorithms to preserve the topology of the surface, but leaves it unable to simplify surfaces with disconnected regions. Garland and Heckbert [24] developed a simplification method for triangle based models that allows for the generation of non-manifold surfaces. A model is simplified by using a superset of the edge collapse operator, called pair contraction. Pair contraction allows for the algorithm to join previously unconnected regions of the model together. In order to select a pair for contraction during a given iteration, the authors derived a cost function from a quadratic equation that describes the squared distance from any point in space to the set of planes through each vertex of a triangle in the model. In the derivation of the quadric matrices, it was assumed that the matrices were well-conditioned and invertible. Typically, this is not the case in practice, especially in areas of a locally flat surface. Also, their technique was not explicitly designed to handle massive models and neither considered, nor preserved surface properties or non-geometric constraints (e.g., colors) when simplifing models.

All simplification methods use some sort of metric to determine the effectiveness of the algorithm from one iteration to the next. Typically, that involved some type of comparison against the original model. Lindstrom and Turk [60], however, developed a simplification technique that did not require a direct comparison against the original model. Their metric, a form of the quadric error metric proposed by Garland and Heckbert [24], uses constraints that minimize the change in volume (volume preservation), the change in area (boundary), the unsigned change in volume, and the change in unsigned area, and

optimized the shape of the resulting triangles. Their technique, while more memory efficient than previous algorithms, does not work for massive models. It assumed that the entire model resides in memory and requires a minimum of 160 bytes per vertex of memory to represent an n-vertex model, including the additional data structures [58] for edge prioritization.

As models have increased in size, simplification techniques moved away from in-core processing techniques to out-of-core methods [9, 10, 12, 47, 59]. One of the earlier out-of-core simplification methods was from Lindstrom [58], who used the quadric error method of Garland and Heckbert[24] to generate an out-of-core simplification of large models. His technique was memory sensitive in the output size of the model, memory insensitive with respect to the input model, and was as fast as most of the in-core simplification techniques published at the time. By using a singular value decomposition of the quadric error matrix, he was able to robustly produce the best candidate vertex for the current cluster, and additionally handle the cases when the error matrix was degenerate. That produces numerically robust results throughout the entire model. His method processes triangles one at a time, constructing an in-core representation of a simplified mesh centered on the current triangle. Simplification then proceeds using the vertex clustering algorithm of Rossignac et al. [72]. Like several of the other simplification methods listed in this section, it does not provide the ability to perform a simplification based on scalar value.

2.2 Multi-Resolution Modeling

As mesh sizes continue to increase, representing a mesh with only one level of simplification is not an efficient use of resources. Building off previous simplification work, multi-resolution modeling uses a hierarchy of simplified representations, with the application picking the appropriate resolution of a model based on any number of criteria, e.g.,

viewing parameters, hardware capability, or network bandwidth.

In his seminal paper, Hoppe [39] proposed progressive meshes, a hierarchy of meshes that encode a set of differences from the meshes that are lower in the hierarchy. By using the principle that edge collapse transformations are invertible, the progressive mesh consists of a simplified base mesh and a record of the vertex split operations required to recover the original mesh. This allows for several techniques to be implemented from this one representation:

1. Mesh Compaction: A progressive mesh has a very efficient space representation, using only $(\lceil \log_2(n) \rceil + 5)\, n$ bits per vertex as opposed to $6 \lceil \log_2(n) \rceil\, n$ bits per vertex for a standard mesh representation (each face requires references to its three vertices, and there are $2n$ faces).

2. Mesh Simplification: At its heart, progressive meshes use a standard edge collapse technique to reduce the mesh complexity. Given the representation, a user can request any desired resolution between the base model and the original model.

3. Progressive Transmission: With a progressive mesh, a mesh representation can be streamed across a network efficiently by sending the base mesh and the required number of vertex split operations to reach a desired resolution.

Using the simplification algorithm first proposed by Hoppe et al. [40], progressive meshes extend the energy metric used to control refinement to include terms for preserving the accuracy of scalar attributes such as color, and discontinuity curves such as boundaries. The sheer number of refinement operations and the requirement for the model to be resident in memory makes progressive mesh representation unsuitable for massive models.

While much of the multi-resolution literature has focused on terrain rendering [11, 13, 19, 27], two papers [12, 28] describe techniques that allow for the interactive rendering of

large static data sets. In Cignoni et al. [12], the data is pre-processed offline into a hierarchy of tetrahedra each containing a specified number of triangles. If a tetrahedron exceeds the maximum triangle count, it is bisected, and the triangle insertion continues recursively. When rendering, the computed hierarchy is traversed in a top-down manner combining back-face culling and view-frustum culling. Traversal stops only when a user-specified screen space error is achieved.

Gobetti and Marton [28] build their multi-resolution hierarchy by using a deep Binary Space Partitioning (BSP) tree. They build the final node representation by traversing the hierarchy in order of increasing Morton code of each node's center point to optimize memory coherency. Leaf nodes are rendered using triangle strips, while interior nodes of the hierarchy are rendered into "impostors" using ray-casting from multiple directions. To render, they created three shader types: two flat shaders (differing in how normals are computed), and one smooth shader. The hierarchy is built in a parallel, offline construction, while the rendering is single-threaded. Both of these techniques work only with static datasets, and they do not consider scalar fields, transparency, or specular reflection in the construction phase.

2.3 Randomization and Statistical Methods

Wand et al. [93] presented an algorithm for interactive rendering of large scenes, the Randomized Z-Buffer, in which candidate points are randomly selected based on an estimate projected area of a "batch" or group of triangles. In order to achieve the speed-ups in the paper, scenes were preprocessed using an $O(n \ln n)$ time algorithm. The running time of the algorithm is proportional to the size of the triangle's projected area, so large triangles in screen space are rendered using a conventional z-buffer. By using splatting, the algorithm can achieve even greater speed-ups over exactly reconstructing each pixel.

Additional auxiliary data structures are used to increase performance (e.g., octree, caching of sample points).

Boubekeur and Alexa [8] describe an algorithm for mesh simplification that uses a stochastic estimator to select vertices from areas of high curvature, but also ensures sampling in flat areas. The triangles connecting the selected vertices are computed by assigning all vertices in the original mesh to the topologically closest selected vertex and then identifying triangles that are incident upon three different selected vertices and then simplifying them. Their method is fast, preserves topology by simplifying in topological space, but is unable to distinguish noise from features.

Many renderers use simplification techniques for high-quality surface-based rendering based on element detail (i.e., detail due to the complexity of individual elements). Cook et al. [16] describe a stochastic technique for simplification based on aggregate detail (i.e., detail due to the large number of elements). Scenes are rendered by randomly selecting a subset of the geometric elements and altering those elements statistically to preserve the overall appearance of the scene.

2.4 Reyes

The original Reyes algorithm by Cook et. al. [15] processed each primitive in the geometric database one at a time independent of all other primitives. Each primitive that survived the culling phase is split into finer pieces and eventually diced into grids of sub-pixel sized quadrilaterals called micropolygons. The grids are then shaded, broken apart ("busted") into individual micropolygons, and then sampled. Because primitives are processed one at a time, all of the samples from the micropolygons are retained until the final primitive is processed, otherwise, the visibility could not be properly determined. As a result, the memory consumption of the original Reyes algorithm was linear in the number

of micropolygons that survived to the final stage of the process, and the number of micropolygons generated were several orders of magnitude larger than the original geometric database. An improved Reyes algorithm [5] was designed that removed the requirement for the retention of the visible point lists. The main change was that rather than process each primitive separately, the entire geometric database is read, bound, and sorted into buckets. Then, each of the buckets is processed as described in the original algorithm. When a primitive is split, it falls into one of three conditions: inside, outside, or straddles the current bucket. If the split primitive falls outside of the current bucket, the primitive is forwarded to the first bucket that overlaps its bounds and further processing of the primitive is delayed until its new bucket is processed. The primitives remaining in the current bucket then complete the entire dice-shade-bust-hide algorithm described above. When the entire bucket is processed, all of the memory used by the overlapping primitives, the visible point lists, and the micropolygons can be reclaimed, and utilized by the next bucket. The trade-off made to achieve the lower memory footprint is the assumption that the entire geometric database can be resident in memory. Several extensions to the RenderMan Interface Bytestream (RIB) alleviated some of that assumption by delaying the reading of a primitive until the renderer began the processing of a bucket that overlapped its bounds.

While the Reyes algorithm has been parallelized before (NetRenderMan) [5], its primary function is to speed up rendering through the distributed processing of the individual buckets by using a replicated database.

In recent years work has focused on enabling the Reyes algorithm for real-time applications using the GPU [22, 23, 68, 94, 99]. While exciting results are coming out of this line of research, for the most part it is not applicable to us as we are investigating scene sizes that far exceed the capabilities of modern GPUs except through streaming extensions. Relevant parts of the research have been investigated and where pertinent, implemented, e.g., DiagSplit by Fisher et. al [23].

2.5 Ray Tracing

Ray tracing is an image generation technique that simulates the traversal of light through a scene and records how much of that light arrives at a camera. Whitted [96] brought ray tracing to the attention of graphics researchers as a method for simulating realistic lighting effects. Over the next two decades, researchers worked to improve many aspects of the ray tracing algorithm including acceleration structures to speed up ray-object intersection by only testing those objects likely to be intersected by the ray [3, 18, 29, 33, 37], improving the capabilities of the algorithm through more complex lighting effects [25, 48, 49, 52, 80, 81], and improvements in the overall image quality by reducing the noise and variance within the generated images [14, 51, 53, 82].

While most research in ray tracing has been focused on photo-realistic rendering, recent research has been exploring the use of ray tracing for rendering large or massive models. Pharr et al. [69] described a system that preprocesses triangles into cache friendly bundles and reorders rays to test against geometry that is already in memory. This allows them to render models larger than would fit into the rendering hardware's memory. In the early 2000's, Wald and Parker [66, 91] independently developed systems that showed the viability of interactive ray tracing using either clusters of computers or large shared memory systems. These two advances led to an explosion of research in the exploration of real-time ray tracing for scientific visualization [17, 54, 64, 65, 87, 97] and the general exploration of large models using ray tracing [2, 35, 55, 56, 76, 77, 83, 88, 89, 92].

2.6 Parallel Rendering

As with the multi-resolution techniques, the parallel rendering approach to the large model problem is to reduce the number of primitives processed by the renderer. The multi-resolution methods try to reduce the number of primitives rendered through the use of a

hierarchy of simplified representations that are visually indistinguishable from the full resolution model when viewed from a particular distance. The parallel rendering approaches typically use a brute force approach that tries to render the full resolution of the model by breaking the data into smaller chunks that can be processed efficiently on a large number of processors. More recent advances have seen the combination of parallel rendering methods with the multi-resolution techniques described in a previous section.

2.6.1 Parallel Rasterization

To begin the discussion of parallel rendering techniques, one has to understand that all parallel rendering algorithms require that the primitives to be rendered must be sorted somewhere in the rendering pipeline. The rendering algorithm can be classified into one of three categories depending on where in the pipeline the sorting occurs: sort-first, sort-middle, and sort-last [63].

Sort-first algorithms partition the screen-space into a set of non-overlapping tiles, each of which is rendered independently. All of the tiles are then composited into a final image. The final compositing step is simply a gluing of the various tiles together: the final pixels are not compared for depth. Unlike a sort-middle approach, the communication necessary to sort the primitives to the various tiles is relatively small. However, there is a high degree of overhead required to do the initial sort. Primitives must be transformed first to determine appropriate screen space bounds, primitive-tile overlap must be computed, and primitives must be rendered if they overlap multiple tiles. This re-rendering causes the largest constraint on the scalability of sort-first algorithms. As the number of processors increases, the amount of overlap, and thus the number of primitives that are rendered multiple times, increases significantly.

In a sort-middle approach, graphics primitives are partitioned equally among the processors, and after rasterization, the pixels are re-sorted into a set of screen space tiles that

have a fixed amount of overlap. This approach is best suited for tightly coupled systems with fast interconnects to allow for the redistribution of the primitives and pixels. In hardware, the primitives are distributed to the vertex and geometry processors, and the pixels are redistributed to a set of fragment processors. Currently, this approach does not work well for a software-based system due to the high bandwidth requirements necessary for efficient transmission of the data between the stages and the efficient redistribution of the data between frames.

Sort-last algorithms function by distributing the primitives evenly amongst the processors. There are many possible techniques to distribute the primitives including random allocation and round-robin. Once the primitives have been distributed the processors render their group. When complete, the images along with the depth buffer are composited together to create a final image. The compositing can occur on the master node, but higher performance is achieved when the images are composed pair-wise in hierarchially.

In 2000, Samanta et al. [74] proposed an algorithm that uses a combination of sort-first to decompose the tiles into N distinct groups and sort-last to resolve the depth on the overlap at the edges of the tiles. Their objectives were to balance the load across the processors and minimize the screen space overlaps. The data was distributed across all the nodes of the cluster, so the model size was limited by the machine with the smallest memory.

In the following year, Samanta et al. [73] described a method to achieve nearly the performance of full database replication by using partial replication. It uses the hybrid sort-first, sort-last architecture from their previous paper with partial replication of the model data. Even with these improvements, the model is still limited to the size of the smallest memory in the cluster.

2.6.2 Parallel Ray Tracing

Pure software renderers provide a finer grain of control over how the memory of a system is used for rendering. Green and Paddon [31] describe a set of methodologies to increase the performance of a multi-processor ray-tracing system by mimicking a virtual memory model. They show the effect of varying memory allocation between the resident set and the cache, and between the voxel hierarchy (octree) and the objects, has on performance. By rendering a low resolution image, they can determine a lower bound on the objects and memory required for the resident set.

Wald et al. [86] proposed a memory management scheme that allowed them complete control over when a page is read into, or evicted from, memory. In particular, they use the memory management scheme to control tile fetching and eviction for an out-of-core renderer. If a ray will access a tile that is not in memory, the ray is killed and replaced with a proxy. Proxies are computed for the interior nodes of the BSP tree and use an average of the material values visible from a particular direction to compute a representation. The proxies are computed using a lightfield approach. The result is an approximate image that can be refined by loading the actual geometry in-place of the proxies are subsequent renders.

2.7 Mesh Organization

Another recent trend in the processing of large models is to focus on the organization of the data provided to a renderer, thereby yielding more efficient access patterns to the data in the form of reduced cache misses or reduced disk reads.

Isenburg and Gumhold [47] described a multi-pass method for compressing large static meshes. They process the data by grouping primitives into clusters, then write each cluster separately to file. The heuristic for the number of clusters is based on the number

of vertices and the maximum memory footprint. One weakness of their technique is the assumption that the indices for the mesh/vertex elements can fit in a 32-bit word. They assume that the indices will never get too large since the indices are on a per-cluster, per-file basis. This is probably true for a massive model that has many parts that can be distributed, but may not be true for singular massive meshes. In order to get the data into the modified format, the algorithm requires six passes through the data to generate the data structures including a graph for cluster connectivity. Along with generating the clusters and other data structures, the algorithm compresses the data on the fly.

As the gap between CPU and memory performance grows, it becomes more important to ensure that data is available in the fastest memory for processing, and this implies minimization of cache misses. Yoon et al. [98] describe a device independent metric for computing estimated cache misses. Standard algorithms that are applied to a cache oblivious mesh layout show a speedup improvement over the standard data organization. Once the metric has been pre-processed, it is used to compute a layout that minimizes the expected number of cache misses for all cache parameters. The global optimization of the layout is NP-hard, so the authors used a heuristic based on multi-level minimization that computes a locally optimal solution.

Chapter 3

PARALLEL METHODOLOGIES FOR A MICROPOLYGON RENDERER

This chapter was originally published in the *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization* [6]. Micropolygon rendering is the process of tessellating input geometric primitives into polygons typically of less than one pixel in size. Micropolygon rendering was first described by Cook et al. [15] and has been used to render complex scenes for motion pictures for 25 years. Over that period this algorithm has been shown to be stable and robust when rendering large, geometrically complex scenes. In this chapter, we investigate parallel extensions of a micropolygon renderer and show that none of the common work distribution methodologies are appropriate for all scene compositions. While the results in this chapter are strictly applied to a micropolygon renderer, the results are applicable to any rendering algorithm that processes the scene in small, limited parts, such as a ray caster using a Hilbert curve for screen traversal.

3.1 Algorithms

In this chapter, we describe four different methodologies for parallelizing the Reyes algorithm. In all of these parallel variants, we are only looking at algorithms that are suitable for running on a shared memory architecture. This allows us to look at the efficiency

of the algorithms independent of any communication costs that would be associated with a message passing approach. We started by implementing the CASCADE algorithm. During testing and analysis, the performance was measured with several tools looking for bottlenecks in the code. The results from the analysis led to each of the following algorithms as a method to overcome one/several bottlenecks. The fourth algorithm was conceived by using the best aspects of the others. In the following sections, we describe each of the algorithms in detail.

3.1.1 CASCADE

One of the most straightforward approaches to parallelizing the Reyes algorithm is for each bucket to be rendered by one thread. Since we will nearly always have fewer threads than buckets, threads are reused on multiple buckets. In Molnar's parallel taxonomy, this approach is a modified version of the sort-first algorithm. The modifications are two-fold: first, threads process multiple tiles, and second, there is no replication of data amongst the tiles.

Figure 3.1 provides a visual overview of the algorithm. The split/dice loop proceeds as described in the improved Reyes algorithm [5] with the exception that the fragments that are output from the loop are forwarded as in the original method, albeit to a potentially different thread. In Figure 3.1a, the screen space size of the geometry is computed, and if it is "too large" (the polygon labeled 1[1]), it is split, bound, and forwarded to the buckets that overlap the upper-left corners of the split pieces (in this case, buckets (a) and (b)) (Figure 3.1b). The only geometry available to thread 2 (which is assigned to bucket (b)[2]), geometry (2), passes the size test and is diced into micropolygons approximately one pixel in size.

[1] From here on out we will refer to these types of elements by their label, i.e., geometry (1)

[2] Buckets hold geometry, and threads are assigned to buckets. When we refer to a bucket, it should be clear from the context as to whether we are referring to the storage entity, or the thread assigned to it.

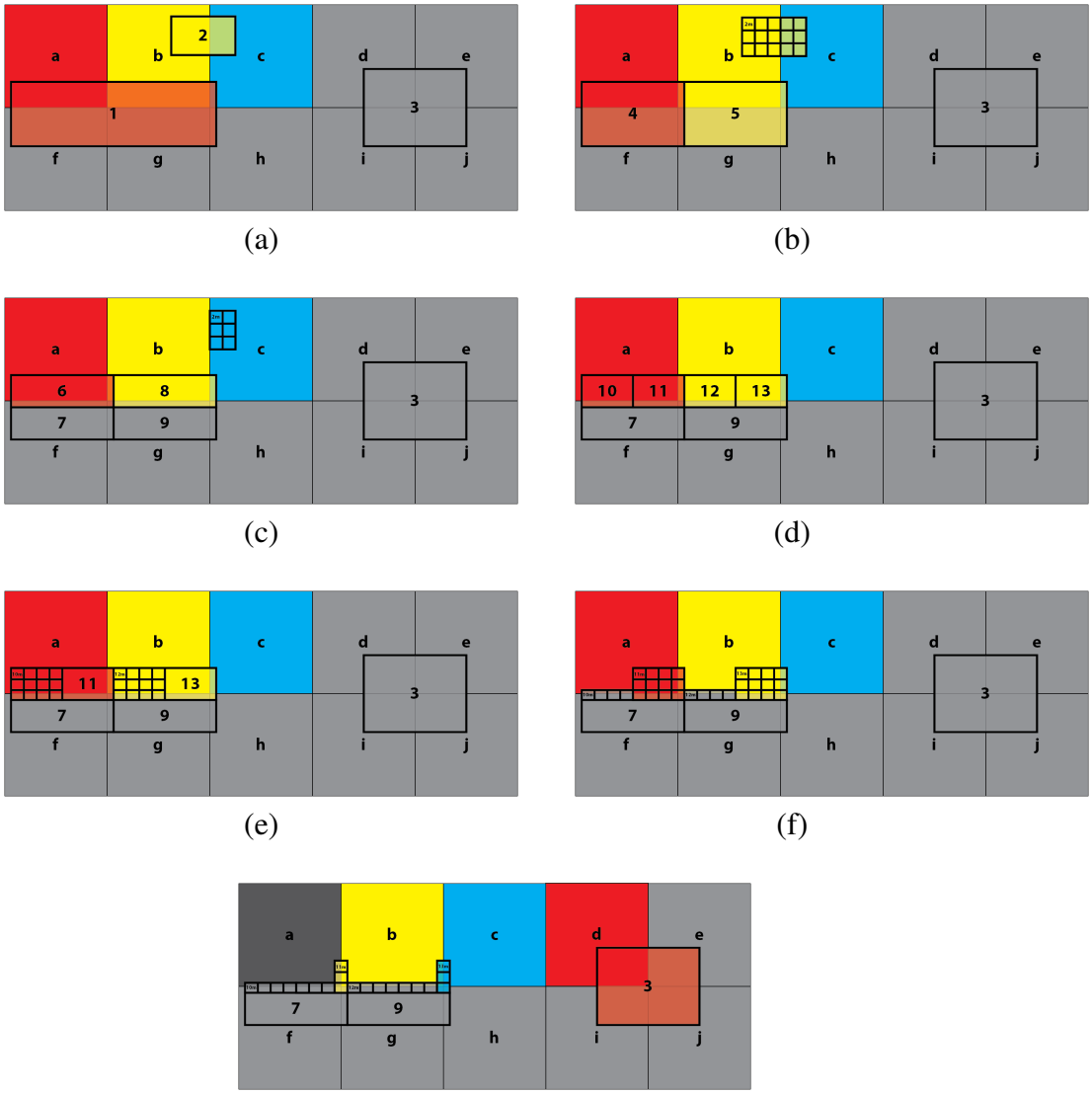

FIG. 3.1. In this Figure, geometry (represented by numbered axis-aligned bounding boxes) is assigned to the bucket (labeled with lower-case letters) that overlaps the upper left corner of its bounds. Both the geometry and the bucket are colored by the controlling thread. Detailed discussion is on page 26

In Figure 3.1c, the split remnants of the original geometry (1) (labeled (4) and (5)) are still too large, so they are split again along an alternating axis. The diced micropolygons from geometry (2) in Figure 3.1b are forwarded to their overlapping bucket (bucket (c)) and processed. In Figure 3.1d, the geometries (6) and (8) are split again by their controlling threads. The thread in bucket (c) waits as it can not proceed to the next bucket as geometry may be forwarded to it from any active bucket that is above it, or to its left. In Figures 3.1e and 3.1f, the geometries are now small enough to be diced, and bucket (c) continues to wait for forwarded geometry. In Figure 3.1g, bucket (a) has finished and forwarded its micropolygons both forward (to bucket (b)) and down (to buckets (f), (g), and (h)) and moved to the first inactive bucket ((d)) and begins processing geometry (3). Buckets (b) and (c) process the forwarded micropolygons. When finished, the controlling threads for these buckets will move to the next inactive buckets. Note, that requiring the buckets to wait for forwarded geometry forces a strict ordering to bucket processing, i.e., yellow always follows red, blue follows yellow, red follows blue, and so on.

To reduce the amount of synchronization involved with the forwarding of primitives, a per-thread queue is added to each bucket. Each thread writes to only a single queue per bucket thereby eliminating any write conflicts between threads. When the split/dice loop is ready to process the next primitive, the thread queues are scanned for any forwarded primitives which are then sorted into the main queue. The primary point of synchronization is due to the unpredictable arrival of forwarded primitives. Since these primitives can arrive at any point during the rendering of a bucket, no bucket can complete until all buckets previous to it have completed. Buckets are statically assigned to threads modulo the thread count.

3.1.2 ROUND_ROBIN

Using a sort-last methodology, we implemented the initial distribution of primitives through round robin allocation. Once the primitives have been allocated, each thread proceeds with a single-threaded variant of the CASCADE algorithm with the exception that the visible point lists for a completed bucket are placed in a thread specific queue of the master process for compositing. The master is responsible for freeing the visible point lists once it has received the lists from all active threads and compositing has completed.

3.1.3 NO_FORWARDING

The third algorithm is also based on a sort-first methodology. Figure 3.2 shows an overview of the algorithm. In the Reyes algorithm, as geometry is bound it is placed into the first bucket that overlaps the upper left corner of the bounding box. NO_FORWARD follows a traditional sort-first distribution by placing a reference to the geometry into each bucket that overlaps the geometry's projected bounds. See Figure 3.2a. The algorithm proceeds as in Figure 3.1, however, the split and dice operations for each bucket now operate independently (Figure 3.2b). When geometry is split or diced, only the geometries that overlap the current bucket are retained. Items that projects outside of the current bucket are discarded. Otherwise, the algorithm proceeds similar to CASCADE. In Figures 3.2c, 3.2d, and 3.2e, the buckets continue to split and dice as necessary, sometimes duplicating geometry. In Figure 3.2f, the thread in bucket (a) finishes first and moves on to the next available bucket (d). Unlike CASCADE, there is no strict ordering to the processing, so a thread advances to the next available bucket as soon as it finishes the current one.

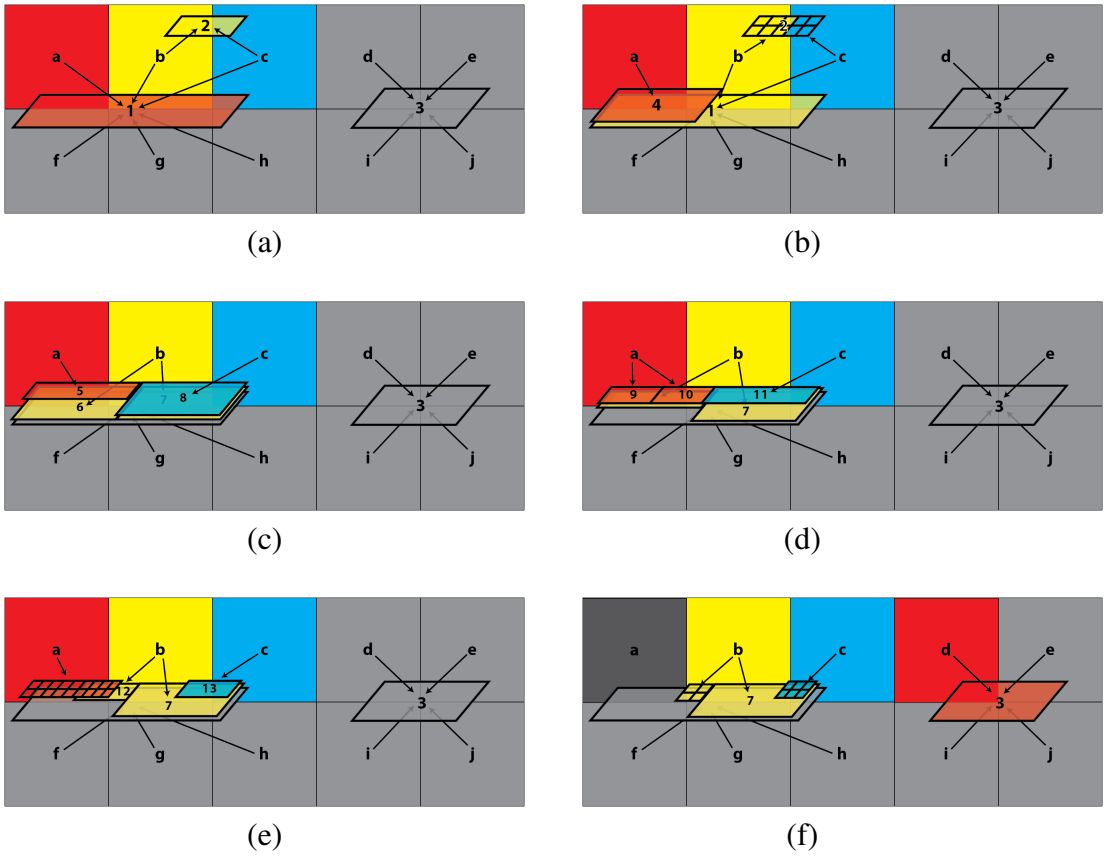

FIG. 3.2. In this Figure, geometry (represented by numbered stacks of bounding boxes) is assigned to all buckets (labeled with lower-case letters) that overlap the object's bounds. Both the geometry and the bucket are colored by the controlling thread. In this algorithm the splitting and dicing of a single piece of geometry occurs in parallel, therefore, we present the same geometry as in Figure 3.1 with as oblique view to enable its visualization. Arrows show buckets referencing the geometry. Detailed discussion is on page 29

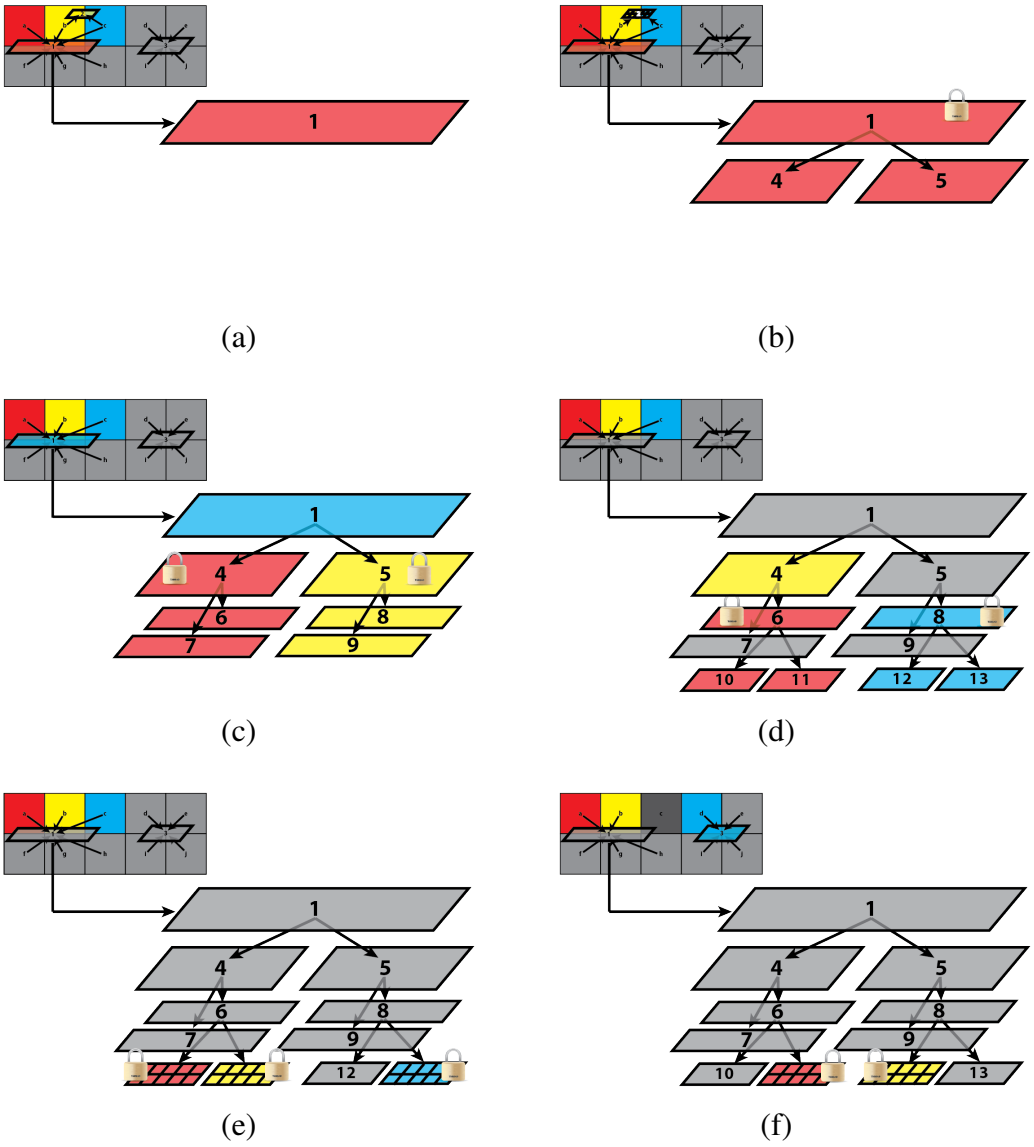

FIG. 3.3. This sequence of images focuses on how this algorithm operates on a single piece of geometry. As in NO FORWARDING, geometry (represented by numbered bounding boxes) is assigned to all buckets (labeled with lower-case letters) that overlap its bounds. Again, both the geometry and the bucket are colored by the controlling thread. In this algorithm geometry is split by the first thread to access the geometry, but the split pieces are stored hierarchically with the parent. Detailed discussion is on page 32.

3.1.4 MODIFIED_NO_FORWARDING

After analyzing the previous methods, this algorithm was derived by combining CASCADE and NO_FORWARDING such that we minimize the bottlenecks from either algorithm (discussed late in the Results section on page 34). From NO_FORWARDING we retain the bucket/thread allocation and the geometry distribution to the buckets. One issue in NO_FORWARDING is that geometry can be split and diced multiple times, one for each bucket the geometry overlaps. Since the geometry is split and diced in the first bucket it overlaps, CASCADE does not suffer from this issue. Figure 3.3 provides an overview of the processing of a single piece of geometry. Figure 3.4 shows the changes made to the original Reyes algorithm. NO_FORWARDING was modified such that the geometry is split by the first thread to access it. In Figure 3.3b, the first thread (bucket (a)) to access geometry (1), locks it, and proceeds to split. Once split or diced, the new items are added back as children of the original node (shown as geometries (4) and (5)). If another thread is unable to lock the geometry, it moves that geometry to a deferred list and then it moves to the next available geometry. In the current Figure, buckets (b) and (c) are unable to process geometry (1), and move to geometry (2). On each iteration through the rendering loop, the algorithm checks if geometry on the deferred list is available and closer than the geometry on the main list. If the deferred primitive has been split, we add the children back to the main geometry list. If the geometry was diced, we shade and bust as normal. In Figure 3.3c, all of the buckets are free to access the top-level geometry and traverse the hierarchy. Bucket (a) (red) locks the left branch (geometry (4)) and splits. Bucket (b) (yellow) would preferentially operate on geometry (4), but as it has been locked by bucket (a), it moves to the right child and splits geometry (5). Bucket (c) (blue) stalls as it has no other geometry to operate upon. In Figure 3.3d, buckets (a) and (b) traverse the left hierarchy with bucket (b) locked from accessing the lowest level. Bucket (c) traverses the right hierarchy and

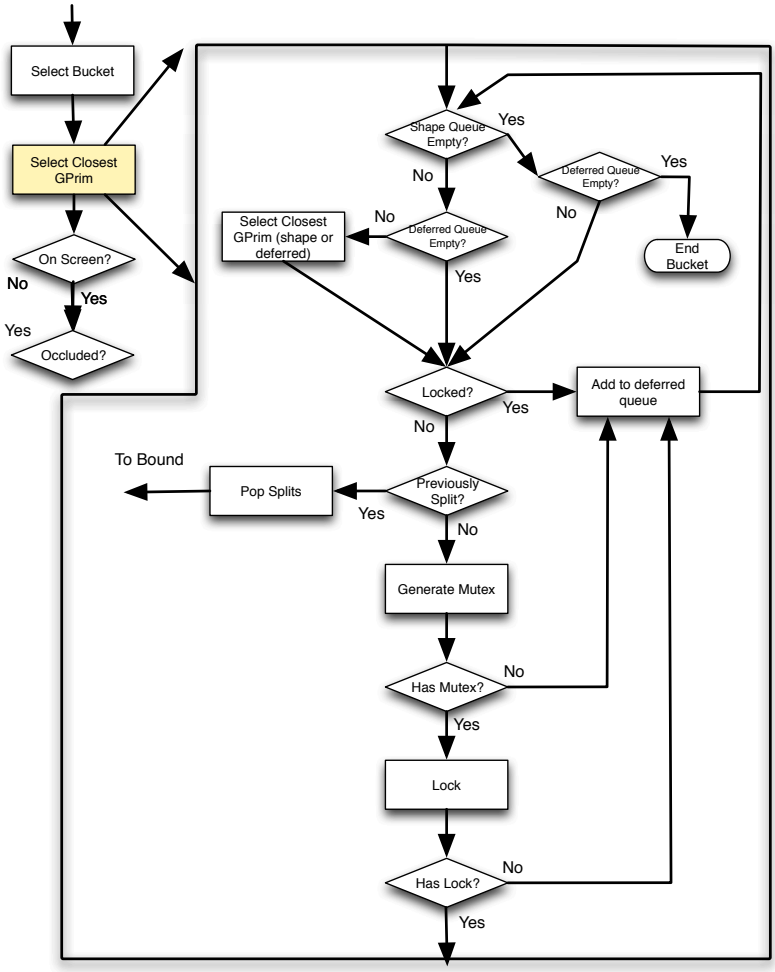

FIG. 3.4. Modified Reyes algorithm

locks geometry (8) which it splits. This causes bucket (b) to stall as it is unable to access any geometry. In Figure 3.3e, all of the threads/buckets are able to dice their preferential geometry. In Figure 3.3f, bucket (c) has finished and moves to the next available bucket. Buckets (a) and (b) process the micropolygons generated by other threads in Figure 3.3e.

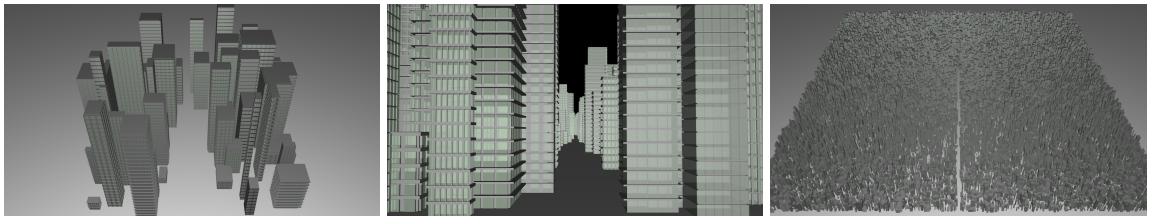

FIG. 3.5. Aerial and street canyon views of cityscapes with 0.1, 6, and 46 million polygons

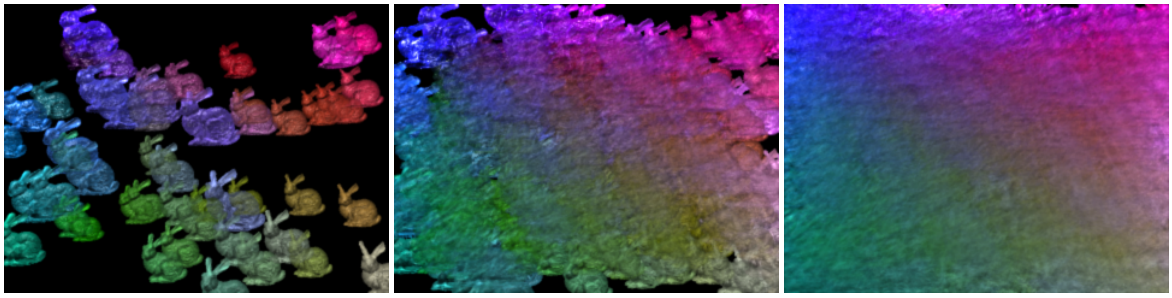

FIG. 3.6. Image result for 50, 400, and 1600 transparent Stanford Bunnies

3.2 Results

The results in this section were obtained on a single Linux system with two 2.67 GHz Intel "Nehalem" quad-core processors and 24 GB of RAM with hyper-threading disabled. The renderer is run with N threads for rendering and a master that handles the parsing of the scene description, converting the visible point lists into final pixel color, and for ROUND_ROBIN, compositing the point lists.

We used synthetic and procedurally generated scenes consisting of polygons (triangles and quadrilaterals) to enable easy comparison of results across the different algorithms. One of our procedural datasets is cityscapes generated using the Houdini modeling and animation software. We vary a large number of parameters for each building including width, depth, the number of floors, windows, and window and floor inset and thickness. Buildings are randomly placed on a grid, one per cell, with the grid ranging in size from 10x10 cells to 160x160 cells, resulting in polygon counts between 147,000 and 50 million.

Two different camera views (see Figure 3.5) were used for testing the effect of minimal and heavy occlusion culling. The synthetic scenes consist of the Stanford bunny randomly replicated 10 to 1600 times in two sets, one opaque, the other with an alpha value of 0.02 (see Figure 3.6). In addition, the number of rendering threads used was varied between one and eight (two to nine actual threads including the master). For all tests, parsing the input data was a single threaded process and the timing for parsing was consistent across primitive counts and algorithms, and so was excluded from the timing results.

Figure 3.7 shows the results for different thread counts versus problem size for the city scene for each of the two views. In the heavily occluded street canyon view, MODIFIED_NO_FORWARDING outperforms all the other algorithms for any combination of threads and scene size (see Table 3.1. Depending on the problem size, MODIFIED_NO_FORWARDING is between 16-40% faster than the other methods for 1 thread, and between 8-90% faster for 8 threads. In the second test scene, the view is modified to be looking at the entire city. Occlusion plays a role in this scene, but the viewpoint was chosen to minimize its impact. As a consequence, MODIFIED_NO_FORWARDING is the fastest algorithm in nearly all combinations of thread count and problem size, except for the largest scene with the two highest thread counts. Two items to note in these results: First, the algorithm that is second fastest changes depending on the scene size and thread count, and second, the differences between these algorithms (excluding ROUND_ROBIN) is typically small, on the order of 3-10% for the larger scenes.

In both scenes, ROUND_ROBIN consistently performs the worst except for the single threaded cases. In the three other algorithms, the master thread's only function post-parsing is to write the final image pixels to file. In ROUND_ROBIN, compositing of the pixels from the rendering threads is performed on the master thread, which gives it a slight advantage over some of the other algorithms. However, in all of the other cases, the performance gain due to occlusion culling is lost as the primitives are assigned to the buckets in random

order.

For the second set of test cases involving the Stanford bunny, the results are significantly more varied (see Figure 3.8). In most of the cases, the original Reyes algorithm, CASCADE, is the fastest with MODIFIED_NO_FORWARDING surpassing it only in the large scenes with low thread counts. But, like the low occlusion city scene, the differences between the algorithms is minimal (excluding ROUND_ROBIN in most of the cases). For the transparent cases, again the results are significantly different in that ROUND_ROBIN is the fastest in nearly all cases. The reason is that with the very low alpha value, occlusion never comes into play except in the largest scene. The offloading of the compositing of the pixels to the main thread allows the worker threads to return to rendering faster than the other algorithms. Varying the opacity value produces results in between these two extremes, depending on how quickly occlusion culling comes back into play (e.g., for an opacity value of 0.1, it takes approximately 45 samples before we reach fully opaque).

# Polygons	# Rendering Threads	Round Robin	Cascade	Modified No Forwarding	No Forwarding Original
147428	1	49.65	41.73	34.67	62.08
	2	49.81	28.13	19.73	31.96
	4	52.82	17.10	10.98	16.65
	7	58.15	12.51	7.15	10.20
	8	81.65	11.26	6.50	9.07
701374	1	46.951	39.10	32.053	59.71
	2	53.79	29.10	17.96	30.84
	4	52.91	17.99	9.93	15.86
	7	60.33	12.91	6.38	9.78
	8	87.69	11.76	5.80	8.70
1568796	1	48.89	41.50	34.28	61.10
	2	51.00	27.14	19.81	31.20
	4	53.88	16.35	11.17	16.25
	7	61.28	11.92	7.57	9.99
	8	89.87	10.87	7.06	8.89
2836538	1	47.83	40.77	33.23	60.79
	2	53.48	30.25	18.89	31.57
	4	55.85	18.62	10.72	16.44
	7	62.22	13.57	7.50	10.14
	8	93.48	12.38	7.02	9.05
6374458	1	54.20	48.16	39.78	69.12
	2	54.54	32.81	22.99	35.77
	4	58.05	20.52	13.00	18.75
	7	67.27	14.58	8.63	11.61
	8	101.03	13.27	8.02	10.38
17600502	1	60.60	57.42	46.78	76.43
	2	61.72	38.19	27.46	40.23
	4	64.59	24.46	16.26	21.87
	7	73.63	17.81	11.68	14.44
	8	102.52	16.71	11.00	13.21
46481006	1	80.08	82.83	66.22	99.57
	2	73.78	56.25	42.43	55.87
	4	73.69	35.06	27.13	32.47
	7	88.50	26.84	21.22	23.13
	8	103.94	25.51	20.36	22.10

Table 3.1. Timing values for the Building scene. This table show the time in seconds for the four techniques versus the number of polygons for the street canyon scene Figure3.5. Note, that in most of the cases, there is very little improvement between seven and eight threads. The eight thread case results in oversubscription of the system as the master thread increases the thread count to a total of nine (on an 8-core system).

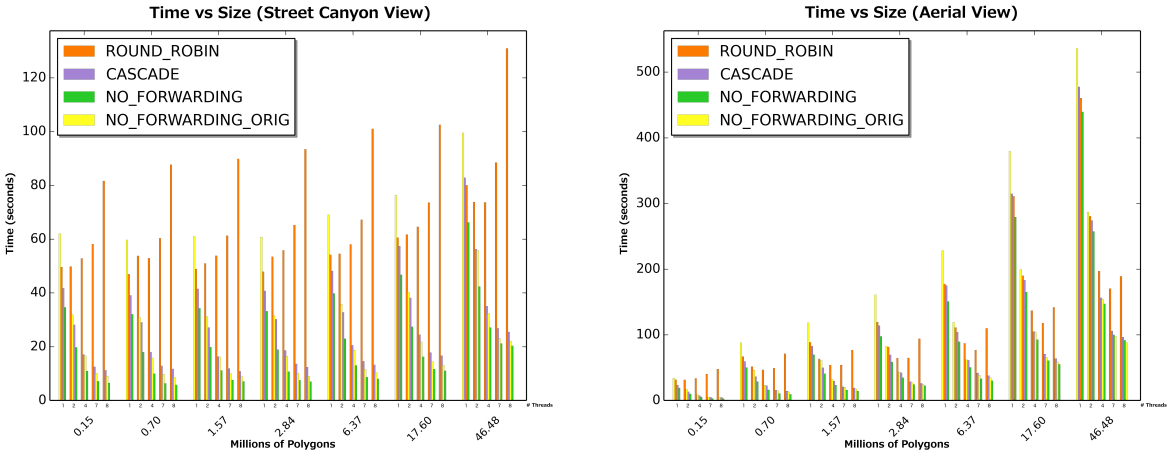

FIG. 3.7. Timing results for the two different city views. Note that the two plots have different maximums for the Y-axis.

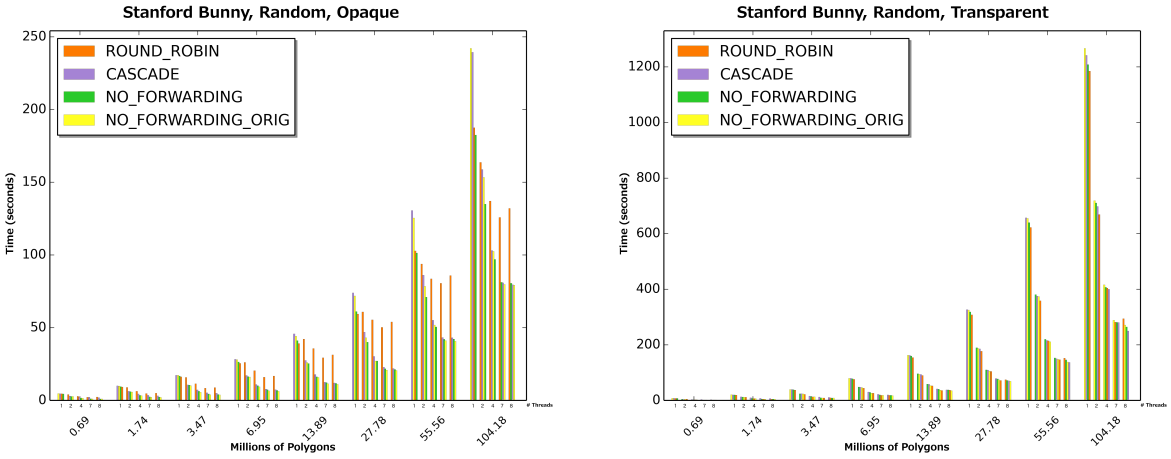

FIG. 3.8. Timing results for the two different Stanford bunny test scenes

Chapter 4

AN ALGORITHM FOR DISTRIBUTED RAY TRACING

As shown in the previous chapter, Reyes is one type of algorithm for rendering. There are a number of other algorithms used in rendering (ray tracing, rasterization, and many others), and each has its own strengths and weaknesses. For photorealistic rendering used in visual effects, ray tracing and Reyes are clearly superior. The Reyes algorithm has a long history in rendering for visual effects, but it has recently been displaced by ray tracing as the main rendering algorithm. Using the lessons learned from the previous chapter, we have updated the rendering system to be based on the ray tracing algorithm.

4.1 Ray Tracing: A Primer

Ray tracing is an image generation technique that simulates the traversal of light through a scene and records how much of that light arrives at a camera. This is accomplished by following photons as they move throughout the scene. Rays are used as a representation of the photon's path as they traverse the scene. A ray tree is the set of paths taken by a photon from emission at a light source to either absorption on a surface or escaping the bounds of the scene.

Forward ray tracing generates rays from every light source either by sampling the hemisphere around the light or by sampling the light's power distribution. Those rays

move through the scene, interacting with the geometric objects and any volumetric media (air, water, smoke,...) until either the rays/photons are absorbed by a surface or media, leave the scene, or, eventually, reach the camera. Many types of complex lighting effects are handled by forward ray tracing, e.g., caustics due to a curved glass or water surface, or color bleeding when diffuse light is bounced from one surface to another. These effects come with an extremely high computational cost. For all of the complicated lighting effects that forward ray tracing can do, it is exceptionally inefficient. Only a tiny fraction of the rays/photons generated from the light sources ever reach the camera.

In order to improve efficiency, the *backward ray tracing*[1] algorithm was created. Instead of tracing a ray from the light to the camera (where a photon goes), backward ray tracing traces a ray from the camera to a light (where a photon originated). Given a scene consisting of various geometric objects, a set of light sources, and a camera, the backward ray tracing algorithm generates a ray with an origin at the camera and a direction through a pixel on the camera's image plane and propagates that ray through the scene testing interactions with geometric elements. These rays coming from the camera are called *primary rays.*

First and foremost, ray tracing is a visibility algorithm. For each geometric object intersected by the ray, sorting the intersection distances provides a front-to-back ordering of the objects in relation to the ray's origin. If we are only concerned with which objects are visible from the camera, then the algorithm is called *ray casting.* Ray casting is one of the main algorithms in scientific visualization for volume rendering. But for photo-realistic rendering, we are interested in simulating the complete illumination in a scene and ray casting is not sufficient.

For each point on an object in the scene, we are able to compute the incoming photons

[1] from this point forward, any time ray tracing is mentioned, it is referring to the *backward* version of the algorithm.

and determine how they were reflected such that they arrive at our eye/camera. We cannot simulate every incoming photon, so we attempt to pick a representative sample of incoming photons and generate *secondary rays* to trace paths to the origins of these photons. One special case is when photons are coming directly from a light source. To handle these photons, we generate rays from an intersection point on an object to each light source, testing if the light is visible along the ray. These are called *shadow rays.* If the geometric object has a mirror-like appearance, we may want to know what other geometric objects are visible from the intersection point. New rays will be generated along a direction by reflecting the incident ray about the normal of the object. These rays are called *reflection rays.* If the object is transparent, we will want to know what is visible through the object, so rays will be generated that are reflected through the object using the object's index of refraction. These are called *transmission rays.* Collectively, the primary and secondary rays are part of the *recursive ray tracing* algorithm.

Recursive ray tracing can account for many modes of light transport, but not all. Heckbert [38] introduced a notation for classifying light paths through the use of regular expressions. In Heckbert's notation, L, a light, is the first vertex in a path; E, an eye or camera, is the last vertex; and the remaining vertices are labeled D or S, which represent whether the light was reflected through a diffuse or a specular bounce from the surface of an object. Additionally, a '?' following a letter means either once or none (D? is interpreted as at most one diffuse bounce); '*' means zero or more; and '+' means one or more. Then, in this notation, recursive ray tracing traces the light paths

LD?S*E.

We can interpret this path in several ways:

- A direct connection from the light to the eye (LE)
- A path from the light to the eye through a single diffuse bounce (LDE)

- A path from the light to the eye through one or more specular bounces (LS+E)
- Or, a path from the light to the eye through a single diffuse bounce followed by one or more specular bounces (LDS+E)

What is missing from this interpretation are the two examples that were listed for forward ray tracing:

- Caustics, given in light path notation as LS+DS+E which we interpret as one or more specular bounces onto a diffuse surface followed by one or more specular bounces
- Color bleeding, which is LD+E in light path notation.

Over the years, several different algorithms [26, 48, 52, 81] have been proposed to account for these additional light paths, but for this dissertation we will use *bidirectional path tracing (BDPT).* BDPT works by generating two light paths, one from the camera (backward ray tracing) and one from the light (forward ray tracing), and then connecting them. The camera subpath starts as in backward ray tracing, but on a surface hit, we sample a new ray direction[2] at the hit point, and continue tracing. Each one of these hit points forms a vertex in the camera subpath. The process is repeated starting from a light, and generating a series of vertices to form the light subpath. Then, for each vertex, c_i, in the camera subpath, we generate a connection to each vertex, l_j in the light subpath, and compute the weighted contribution of light along the path

$$c_0, c_1, ..., c_{i-1}, c_i, l_j, l_{j-1}, ..., l_1, l_0$$

Each of these paths represents one specific photon path from the light to the camera. The more subpaths that are generated, the better we can approximate all of the possible paths

[2]The new ray direction is chosen by sampling from the Bidirectional Scattering Function (BSDF), but how we do that is irrelevant for this dissertation.

for a photon to leave the light source and arrive at the camera. For the rest of this chapter, we will examine the parallelization of the standard ray tracing algorithm and BDPT.

4.2 Target Architecture

Before the discussion of the parallelization of the ray tracing algorithm, we need to discuss what hardware architecture our design targets. The system consists of a networked collection of computational nodes, each with one or more central processing units (CPUs) that consists of a small number of individual cores connected by fast shared memory. Relative to the core to core communication speed, the inter-node network connection is several orders of magnitude slower. Typically these types of systems are found in dedicated compute clusters or in a set of networked desktop workstations. While the system described below will have its highest performance on a system with tightly networked nodes, there is nothing in the design that prohibits it from running on a geographically distributed, loosely connected set of nodes.

4.3 Parallelization of the ray tracing algorithm

Much of the complexity in parallel algorithms comes from the coordination between the parallel tasks. Ray tracing falls into a category of algorithms that are labeled as "embarrassingly parallel" or "perfectly parallel." Embarrassingly parallel algorithms are defined as those where there is little to no coordination between the parallel tasks. In either ray tracing or BDPT, the point of ray generation from the camera is where the parallel tasking begins. On a system where all of the scene objects are directly accessible by the ray and its child rays, this algorithm will exhibit linear or nearly linear scaling, However, it experiences performance degradation when the scene is loaded into distributed memory, such as in a cluster or a distributed set of individual computers.

Many researchers have worked to increase the performance of ray tracing by using distributed systems with mixed results [36, 71, 76, 90]. In this dissertation, we approach the distributed data problem from a different viewpoint: How do we *efficiently* render a scene that *must* be distributed because the entire scene exceeds the memory of a single system/node, or, if the scene exceeds the distributed memory of the entire rendering system?

4.4 A Distributed Ray Tracing Algorithm

In a distributed ray tracing system, three ways to parallel render are by transmitting geometric or other scene data between nodes, transmitting rays, or sort-last compositing [63]. Of these, sort-last has seen wide use in distributed volume rendering of large datasets. For volume rendering, the dataset is typically partitioned onto a grid, with each block or subset of blocks distributed to a processing element. One algorithm for volume rendering uses ray casting, in which a primary ray is sent to the processing elements whose grid element is intersected by the ray. Because we are ray casting, the strict front-to-back ordering of the intersections with the grid elements allows the resulting color and depth values to be sorted and composited either on the master node or hierarchically. However, when ray tracing, the secondary rays break the strict depth ordering, and sort-last compositing is no longer viable. In this section, we propose a system for distributed rendering based on an efficient method for sending rays.

For the rest of this section, our rendering system consists of a single data server node and N rendering server nodes that have been fully bootstrapped. For each server node *i,* we assume there are M_i cores available for usage. How the system is bootstrapped is unimportant for this chapter, but is described in Chapter 6. We assume the data server is fully connected to the rendering servers, and the rendering servers are aware of the other

servers in the system, but are only connected to the data server initially[3].

All of the information about the scene is stored in the RenderMan Interface Bytestream (RIB) [78] file format. RIB was chosen as it is widely supported amongst commercial and open source modeling tools, has wide spread support in the visual effects industry, has a binary format for reduced file size, and has a number of features that work well with our distributed renderer. The format is less used for visualization, but converting to the format is fairly easy using various modeling tools. The first part of a RIB file is the Options section. It contains one-time initialization information such as search paths for referenced files and all the information necessary to create the viewing camera (transforms, image resolution, image depth, etc) In a RIB file, the tag *WorldBegin* delineates the end of the Options section, and the beginning of the scene description that contains all of the attributes and geometry.

4.4.1 Partitioning

Once the viewing system has been completely defined, we create a camera and the image object for storing pixels. We initially create a static partition of the image plane into rectangular groups of pixels based on the number of render servers. Since we put no restrictions on the number of render servers, the partitioning scheme used is slightly more involved than a system that restricts the node count to a power of two. The goal is to find the two multiplicative factors of the number of render servers that best approximates the aspect ratio of the output image with extra servers being assigned to a reserve pool where they are reallocated at a later time. The partitioning scheme is as follows:

- We remove one server and assign it all the space behind the near clipping plane
- If the remaining number of render servers is odd, we take one additional server and

[3]We make this assumption for clarity when discussing the algorithm. For practical purposes, a forwarding layer of nodes is created if the data server is unable to fully connect to the rendering servers due to lack of availability of system resources, e.g., to fully connect exceeds the maximum number of sockets.

also assign it the space behind the clipping plane, now dividing the region in half.

- We compute all of the factors for the remaining number of servers, and sort the values into a list. Starting from the middle of the list, two pairs of factors are chosen, the aspect ratio is computed for each pair, and the values are compared to the image aspect ratio until two pairs are found that straddle the value[4]. The pair that is closest to the image aspect ratio is chosen.

By reducing the server count by two and iterating the final step, the system can find a pair of factors that closely matches the image aspect ratio. Four iterations were used for the discovery of the best pair of factors as it limits the number of servers moved to the reserve pool to at most eight and preserves the majority of the servers for the tracing of primary rays. We leave a more thorough evaluation of finding the optimal number of iterations for future work.

Not all of the scene objects lie within the bounds of the viewing frustum, and the partitioning must be adjusted to account for the scene objects that lie outside of this bound. The bounds of the grid cells that lie on the image boundary are extended to infinity. To account for the additional objects held by the boundary grid cells, we decrease the size of the edge grid cells by one half (this automatically reduces the corners to one quarter of the original area) and adjust the remaining grid cells to compensate as shown in Figure 4.1. Servers from the reserve pool are statically assigned to edge cells that have the largest amount of geometry. These regions are then spatially subdivided and their geometry reassigned.

Since the system is statically load balanced for the distribution of scene objects and not for workload, the decision to reduce the edges by a fixed amount worked well for our test scenes. However, dynamically adjusting the edges is an area for future work that would fall

[4]For an aspect ratio of 1.77778 (16:9) and 80 render servers, the factors are: 1, 2, 4, 5, 8, 10, 16, 20, 40, and 80. The middle two pairs are 8 and 10, and 5 and 16 with aspect ratios of 1.25 and 3.2. respectively.

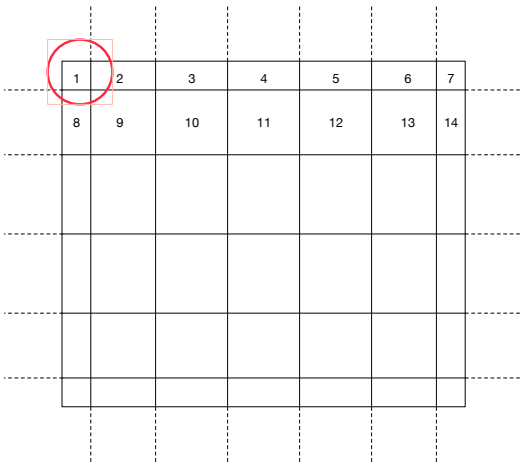

FIG. 4.1. Image space allocation of processors. Grid cells on the boundary of the image plane are extended to infinity (dotted lines). Since the boundary cells covered more physical space, their screen space allocation is reduced. The red circle is the screen space projection of an object which will be attached to partitions 1,2,8, and 9.

into the larger research question of finding a balance between statically and dynamically load balancing for both distribution and performance.

4.4.2 Parallel Loading of Scene Objects

A significant bottleneck for rendering massive models is the sheer amount of data that must be read from disk. In order to reduce the time spent by the renderer in I/O, we need to parallelize the loading of scene objects. Most file formats, and with RIB files in particular, are not conducive to reading from multiple threads. Scene description files (a scene file is one that includes both graphics state and geometry) are typically structured in a hierarchical manner such that the position and appearance are declared prior to the geometry declaration. Some formats (such as RIB) allow for the scene description to be split amongst multiple files, each containing stand-along graphics state and geometry. While these files may be read from a separate thread, the main file which contains the entire graphics state remains single-threaded. Additionally, we want to perform some inline preprocessing such

as spatially partitioning large compound geometry objects or separating disconnected geometry that has been grouped into a single entity. These changes reduce the I/O load at later stages of rendering and increase the probability that data might not be loaded due to occlusion. In general we are assuming that we have a *write once, read many* model of data I/O such that any preprocessing that is done on the first read will be of benefit to any other read (render) in the future. Performing these steps on first read allows us to parallelize the processing of the geometry. Since we defer loading of the actual geometry until it is necessary to be rendered, these steps allow us to parallelize subsequent file I/O.

On a server with M cores, the data server runs a single "read" thread, and M-1 processing threads. The scene data is parsed by the read thread and proceeds through the scene description gathering graphics state, e.g., object to world transforms, texture coordinates, color, opacity, etc. When the parser identifies a geometry declaration, the declaration is buffered into memory and placed into a queue along with the current active state. The read thread continues in this manner until the entire scene description is processed. The next available processing thread pops the topmost geometry declaration from the queue and parses the geometry type. Depending on the type of geometry, several different types of processing can occur.

- If the processed geometry is of a singleton type, e.g., quadrics (spheres, cones, ...), bilinear patch, or a single polygon, we compute the bounding box and the starting and ending byte offsets into the file.

- For compound objects such as polygon or patch meshes, we load the vertices and edges into a graph data structure and run a connected components algorithm [7]. We take special care in preparing the data for the connected components algorithm to ensure that duplicate vertices are removed and that any attributes (colors, normals, etc.) and their mappings (constant, face, vertex) are preserved. Then, for each connected

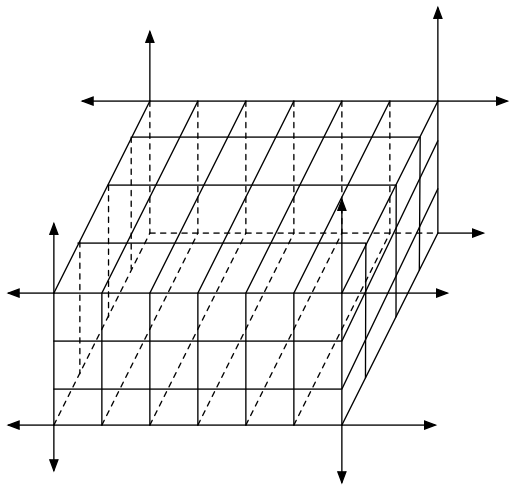

FIG. 4.2. Grid for loading massive models.

component, we export a new file of the geometry and its attributes. The bounding box and byte offsets are then computed as above. When all of the components have been written, we replace the geometry definition in the original file with references to the new files containing the partitioned geometry.

- For a single massive compound mesh, i.e., a volumetric dataset, we first stochastically read approximately 5% of the vertices to estimate the bounds of the geometry. Once we've estimated the bounds, the mesh is subdivided into a grid. The grid's cell size is calculated such that for a uniform distribution of vertices, the number in each grid cell is below a user-defined threshold. The space exterior of the grid is divided into 26 cells as defined by infinite planes on each of the grid's faces as shown in Figure 4.2. The geometry is then read again, and the vertices, faces, and attributes are streamed into separate files corresponding to the geometry's overlapping grid cell. If a grid cell exceeds the size threshold, this process is repeated for the offending grid cells. In this iteration, though we have complete knowledge of the bounds and the

number of items in the cell, and those values can guide the second partitioning. The original dataset is replaced with references to the new partitioned data, and we then run the process for compound objects listed above for each of the new files.

Regardless of the method by which the geometric objects are processed, in the end we have a *Shape_Proxy* object. A Shape_Proxy contains all the state information necessary to load the object on demand.

4.4.3 Distribution of Scene Objects

After a proxy object is instantiated, its screen space footprint is computed. For each grid cell partition of the image plane that it overlaps, the partition identifiers are added to the proxy object (Figure 4.1). For those objects whose projection falls outside of the image plane, their bounds are mapped to the closest edge and placed into the appropriate boundary cell partition. The object is then serialized and broadcast to the render servers.

The render servers deserialize the objects and add the object into a Bounding Volume Hierarchy (BVH). A BVH is an acceleration structure that reduces the cost of ray object intersection testing by first testing the rays against an object that is cheaper to test and only then testing the full object if the ray intersects the simpler object first. There are many different constructors for BVHs [3, 37, 83, 84], for this project we use a modified version of the Approximate Agglomerative Clustering (AAC) builder [33]. The AAC builder is fast, tunable, and has ray intersection performance that is comparable to the current state of the art. The modifications that were made to the builder were to drastically reduce the size of the intermediary data structures used by the algorithm. As the render servers deserialize the proxy objects, they are added into the AAC builder.

If the collective size of the proxy objects exceeds 50% of the available memory on a render server, the server temporarily stops accepting broadcasts and runs the AAC builder

to generate an intermediate BVH. We then traverse the BVH looking for nodes in the hierarchy that exist wholly on a distant[5] render server or servers. At this point, we prune the hierarchy, generating a new proxy object with the current bounds and partition identifiers. This node is added into the AAC builder while removing all of the pruned objects. We then re-enable receiving broadcasts and continue operating as before.

4.4.4 Communication Layer

We use the socket library ZeroMQ [4] as the basis for the communication layer. Each render server consists of a single asynchronous receive socket (in ZeroMQ terminology, a ROUTER socket) and an asynchronous send socket (a DEALER socket) per connected peer. As stated earlier, at the start of rendering, every render server has only a single connection with the data server node.

Each message has two parts, an envelope and a payload. The envelope consists of the identifier of the sending server, a message type, and the size of the payload. We use the size field to allocate a buffer large enough to hold the payload for deserialization. When a render server receives a message on the ROUTER socket, it parses the envelope and creates the DEALER peer socket if it currently does not exist. The size field is compared to the current receive buffer and the buffer is reallocated if it is too small. The payload is then received and handled based on the message type where the appropriate deserialization can occur. The system has the ability to remove inactive sockets based on either a timeout or on a fixed-size queue of peers. We use the queue if we need to set a hard limit on the maximum number of open sockets on a rendering server, but with a small queue size, the system will aggressively close sockets such that a significant overhead is incurred in closing and re-opening sockets. Figure 4.3 shows the performance drop off as a function of the size of the

[5] As each render server has all the information to compute the partitions of every other server, we compute the distance as the farthest servers in reference to the screen space partition.

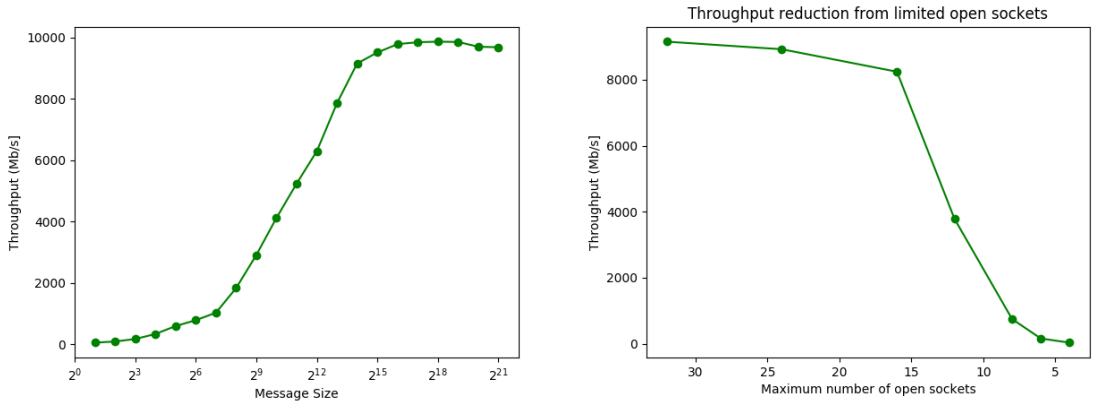

FIG. 4.3. The left image shows ZeroMQ scaling with message size between two 10GigE connected nodes. The right image shows performance thrashing as a function of the maximum number of connected peers. We broadcast 16k message from 32 nodes round robin.

queue. As shown in the figure, as the queue size gets smaller the system begins to thrash on opening and closing sockets leading to a large drop in total message bandwidth.

4.4.5 Rendering

For a render server with M cores, we launch M-1 rendering threads, with the main thread handling all communication to the data server and the peers. Each of the rendering threads is given a small bucket of pixels (typically 16 x 16, but under user control) to render. Each thread traces a primary ray[6] through a pixel and into the scene.

4.4.5.1 Recursive Ray Tracing For the recursive ray tracing algorithm, all of the primary rays begin and end within the domain of the render server. The distributed nature of the algorithm begins when the secondary rays are generated. As the ray traverses the BVH hierarchy, several different behaviors can occur. See Section 4.1 for an overview of

[6]Without loss of generality we can say "ray", typically we are tracing multiple rays per pixel to reduce variance.

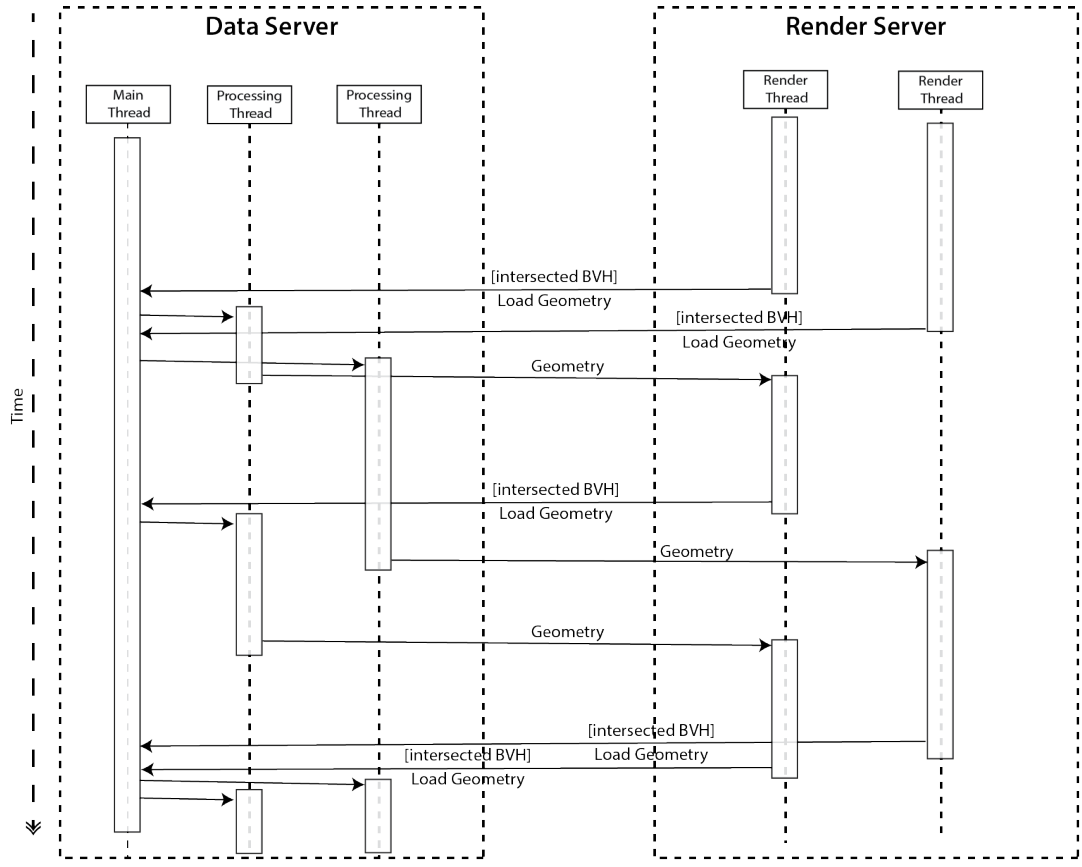

FIG. 4.4. Geometry Request Sequencing. As each rendering thread intersects a local BVH, a message is sent to the data server requesting the full geometry. The request is handed to a processing thread on the data server that loads the geometry and sends it back to the requesting server.

the algorithm. First, if the intersected node is an intermediate one, we recursively depth-first trace into the children, starting with the child closest to the ray origin. If the node is a leaf, the behavior depends on whether the responsibility for that BVH node is with this render server. If the BVH node resides on this server, we transmit the information from the proxy object back to the data server and wait for the response. If the node is a compound object, the response will be a second-level BVH hierarchy that gets stored with the proxy object. See Figure 4.4 for a representation of the load request sequencing. We then continue recursively tracing into the second level hierarchy. If the data server response is a simple primitive, we test the ray against the primitive, and if it is a hit, we compute the shading information, update the color and opacity for the ray, and generate the shadow rays and any other secondary rays required by the shading material.

If the leaf node resides on a different render server, than the ray is added to a queue for that server. Unlike the Hyperion renderer [21], which bins all rays into one of six cardinal directions and then processes them in groups, we directly transmit groups of rays to the server where the data resides. The algorithm continues in this manner until all of the primary rays have been generated and traced. There is one aspect of the algorithm that has not been discussed yet, and that is the treatment of shadow rays. Light sources have the potential to be a massive ray sink, because, for each surface intersection point we trace shadow rays to the light source to test for visibility. See Figure 4.5 for a visualization of shadow rays. For this reason, we treat light sources differently from other geometry sources and distribute the lights and any associated geometry to all of the render servers. When tracing a shadow ray, we first try directly loading any intervening geometry on the current render server, even if the geometry is the responsibility of another server. If the intervening geometry is too heavy to load on the current server, we defer the evaluation of shadow rays and handle them with the other secondary rays. Depending on the number of lights, the majority of light sources will be outside the domain of a render server.

Algorithm 4.1 Algorithm for distributed recursive ray tracing.

```
For each thread:
  For each pixel in bucket:
    Generate primary ray through the pixel
    Trace ray into BVH
    While ray intersects BVH nodes:
      If BVH node is allocated to this render server:
        If node is a leaf:
          Send proxy information to data server
          If node is a compound primitive:
            Receive 2nd level BVH hierarchy
            Trace into children
          If node is a simple primitive
            Receive primitive information
            Test ray against primitive
            If ray hits primitive:
              Shade
              Generate shadow and secondary rays
              Continue tracing with generated rays
        If node is intermediary node:
          Test ray against each of the children
      If BVH node is not allocated to this render server:
        Trace ray until node is a leaf
        Add to queue for distant render node
```

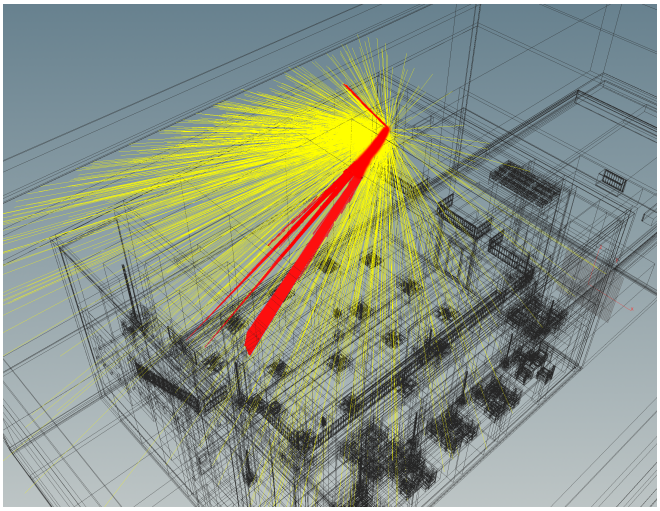

FIG. 4.5. Light sources as a ray sink. The four bundles of rays highlighted in red are shadow rays to different light sources.

Once a render thread has exhausted all of the primary rays for its current bucket, it pulls ray packets from remote servers and processes them based on the packet type. If the ray packet contains secondary rays to trace, then those rays are extracted from the packet and traced using the algorithm described above. If a ray terminates within a thread's domain, the color values for the original ray are updated with the color value for this ray, and then placed in a queue for transmission back to the originating server. If the packet contains rays that have terminated and are being returned to the server, then we update the color and opacity of the originating ray. Once all the spawned secondary rays have returned for the originating ray, if that ray began on another server, the ray is added to a queue to be returned. If the ray originated on the current server, then once all of the secondary rays have terminated and returned, the final color from the ray is used to update the appropriate pixels in the image plane. See Figure 4.6 for an example of the message passing and sequencing for the algorithm.

If the renderer were to wait for the remote requests to finish before generating the

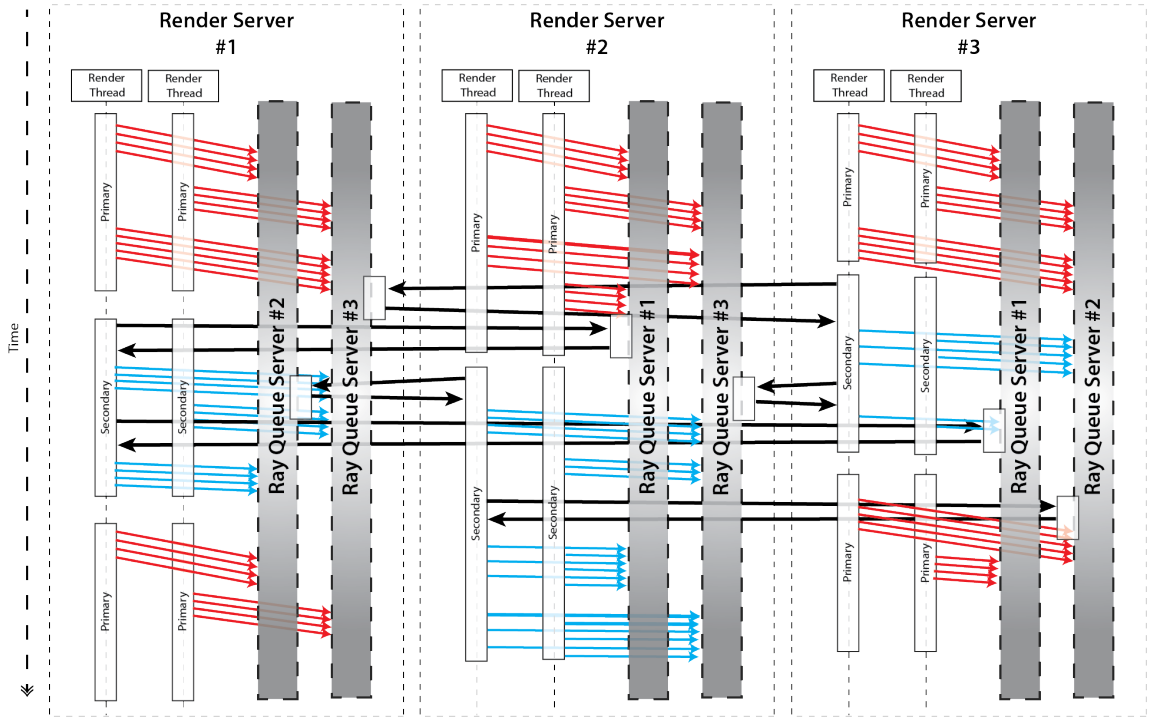

FIG. 4.6. Parallel Rendering Sequencing. This figure depicts the sequencing for a three server system. Threads in each server trace primary rays (red), and upon an intersection with a remote BVH, the ray is placed into the queue for that server. Once the primary rays are exhausted, the server pulls ray requests from a remote server queue (heavy black arrows), and begins tracing secondary rays (blue). This is repeated for each remote server. The server continues to pull rays from the remote servers round-robin until the number of in-flight rays falls below a threshold, then primary ray tracing is restarted.

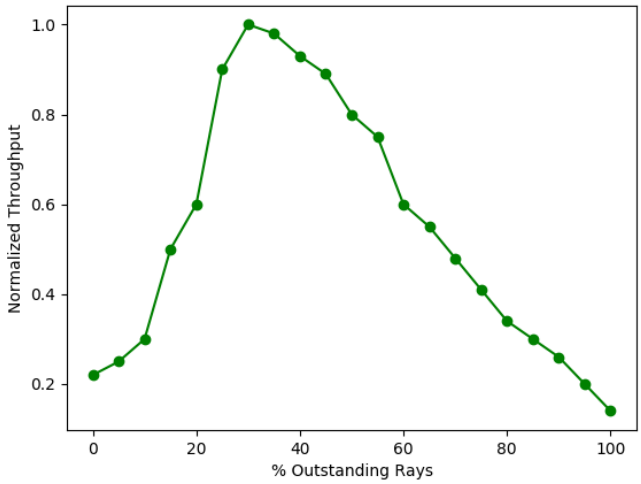

FIG. 4.7. Performance versus percentage of outstanding primary rays. Under heavy load, the renderer will generate a new set of primary rays when the number of pending primary rays falls below a threshold. In the image, 0% means the renderer waits until all of the primary rays have returned, and 100% means the renderer generates all of the primary rays for its domain.

next buckets worth of primary rays, the system would experience starvation. To minimize starvation, each render server continuously polls for incoming messages, and when the queue is empty, the renderer initiates processing of the primary rays for the next set of buckets. As the system progresses beyond the initial set of buckets, the probability of an empty message queue approaches zero. At that point, a second strategy is employed. When the number of active primary rays for the renderer falls below a certain threshold, the renderer initiates primary ray generation for the next bucket. After a series of small experiments (see Figure 4.7), it was determined that a threshold of 30% provided the best tradeoff between performance and memory consumption.

4.4.5.2 BDPT The distributed version of the bidirectional path tracing algorithm is implemented similarly to 4.1 but with a few key differences. First, recall that the BDPT

algorithm traces individual rays from the camera and from each light through the scene, bouncing through surface interactions until they terminate. Then the vertices from the camera path are connected to the vertices from a light path, and the color contribution is accumulated. There are two main differences as to how the rays are propagated through the system. First, when a ray intersects a surface, instead of generating the shadow rays and all of the secondary rays, the renderer generates a single secondary ray based on the scattering properties of the surface. The second change is that the renderer does not accumulate the color from the rays, but instead accumulates the vertices generated by the ray's interactions with the scene objects. The algorithm then proceeds through three different phases:

1. The renderer generates all of the initial camera subpaths for a bucket, traces them through the server's domain following single scattering events, and places any ray that leaves the domain into an outgoing queue.

2. The renderer generates the initial rays for the light subpaths. Because the number of lights is variable per scene, as is the number of samples used per light, the renderer will generate a fixed minimum number of rays rather than a complete set for all of the primary rays, a number which could easily overwhelm the system. For the minimum, the renderer will generate at least as many rays as there were primary rays. However, there will generally be several times that number. Since we have only minimal duplication of geometry across servers, and the initial partition is optimized for tracing of primary rays, there is a large communication cost for tracing the light subpaths. In the San Miguel dataset, the render server responsible for the domain labeled one (see Figure 4.8) begins tracing light rays from the light source in domain seven, each path from the light that intersects the geometry in the alcove requires communication. We trace those rays as in step one above.

3. As we receive back the initial camera and light subpaths, the renderer generates the

FIG. 4.8. Overview of communication cost in BDPT. When the server for region 1 needs to generate light paths for BDPT, the first intersection point will almost be guaranteed to intersect geometry owned by a different server and therefore require communication

rays that connect the vertices between the subpaths and traces those rays as above.

4. When the rays that connect the camera and light subpaths are returned, we accumulate the color value along the connected subpaths.

5. Steps 2-4 are repeated until all of the light subpaths have been generated for all of the camera subpaths.

In between each of the steps 1-4 above, the renderer receives and traces the subpaths requested from the other render servers as described in 4.4.5.1.

4.5 Results

When evaluating a rendering algorithm, there are two main characteristics that the user wants to know about: what is its feature set, and how quickly can it render a scene? For this dissertation, we wanted to evaluate our renderer on two additional criteria: what is its capability, and what is its scalability? For capability, we want to know if it can handle

massive datasets as designed, and for scalability we want to know how our performance scales as more render servers are added to the system.

The hardware used for testing is part of the Howard Hughes Medical Institute at Janelia Farm's compute cluster consisting of servers interconnected with 10Gbit Ethernet. Each node in the system consists of dual 2.7 GHz Intel Sandy Bridge (E5-2680) with eight cores per CPU and 32 GB of memory per node.

For testing we used three models, San Miguel, the UNC Power Plant, and a model of the Boeing 777. By modern standards, neither San Miguel or the Power Plant are considered massive, but they allow us to rapidly test for scalability and communication costs using both recursive ray tracing and BDPT. Due to the size of the Boeing model, we were unable to load it into any modeling program to properly set up interior lighting for a BDPT test. Table 4.1 shows model statistics pre and post partitioning including the memory footprint for the loaded models.

Model		Polygons (M)	Objects	Memory (GB)	Proxies Created/Loaded	Partitioned Objects	Proxies Created/Loaded	Memory (GB)
San Miguel		6.5	5372	1.7	5356/5339	216765	211405/191565	1.8
UNC Power Plant	Exterior	12.7	29	2.93	22/21	87175	102471/91871	2.87
	Interior						92807/18803	1.11
Boeing 777		420	13672	28.3	13665/12250	706382	706345/119954	18.2

Table 4.1. Model statistics.

We first compare threaded and distributed scaling using the San Miguel and the Power Plant models. Since we can fit both of these models on a single node of our target architecture, we can compare the effect communication has on scaling with the same number of cores.

Figure 4.9 compares single node versus distributed performance. As expected, there is a significant drop in performance as we add communication to the rendering process. As

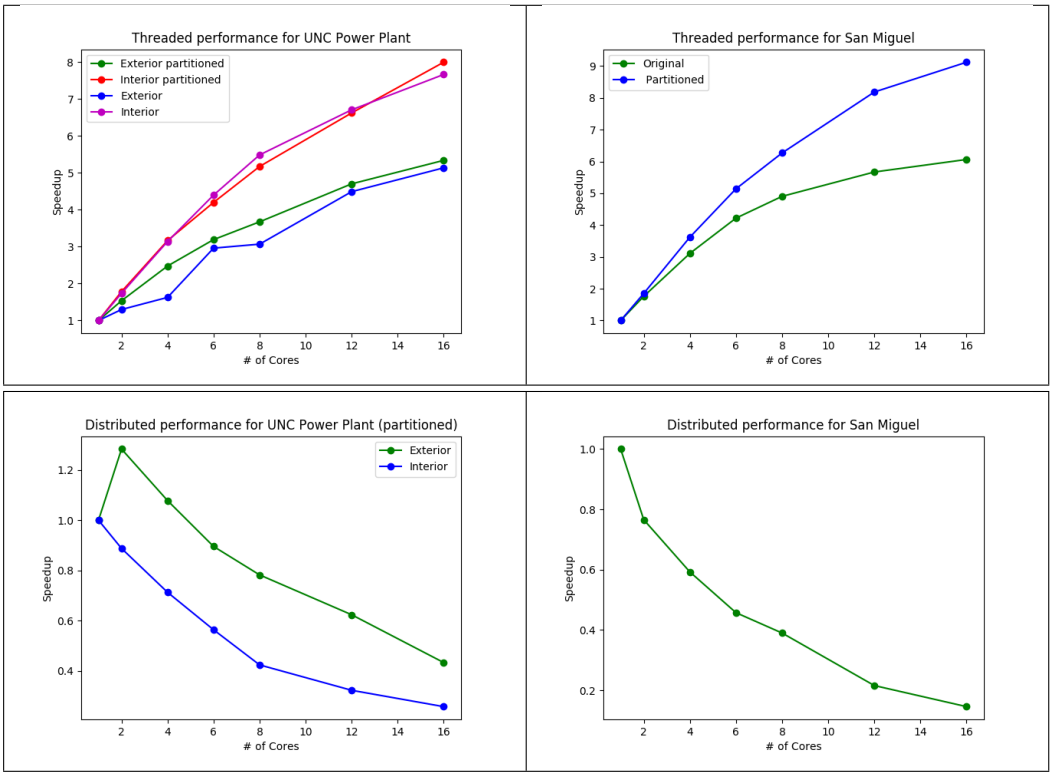

FIG. 4.9. Comparison of threaded versus distributed rendering for the same number of cores at four samples per pixel. In the threaded cases, we do not achieve linear speedup due to dynamic loading of geometry. While the speedup curves for the Power Plant are nearly identical for the original versus partitioned, the partitioned version was ~20% faster.

we increase the number of servers while holding the total core count constant, performance degrades due to the additional communication. In comparing the scaling of the San Miguel and Power Plant models, even though the Power Plant model is more complex (12 million versus four million polygons), the heavier occlusion reduces the amount of communication necessary to load the parts of the model from the server.

Figure 4.10 shows the scaling for our three models as we increase the number of servers. In order to ensure the servers had work to fully measure scaling, we increased the samples per pixel to 256. As the number of servers increase, in this particular hardware configuration the number of cores also doubles. As shown in Figure 4.9, our single node

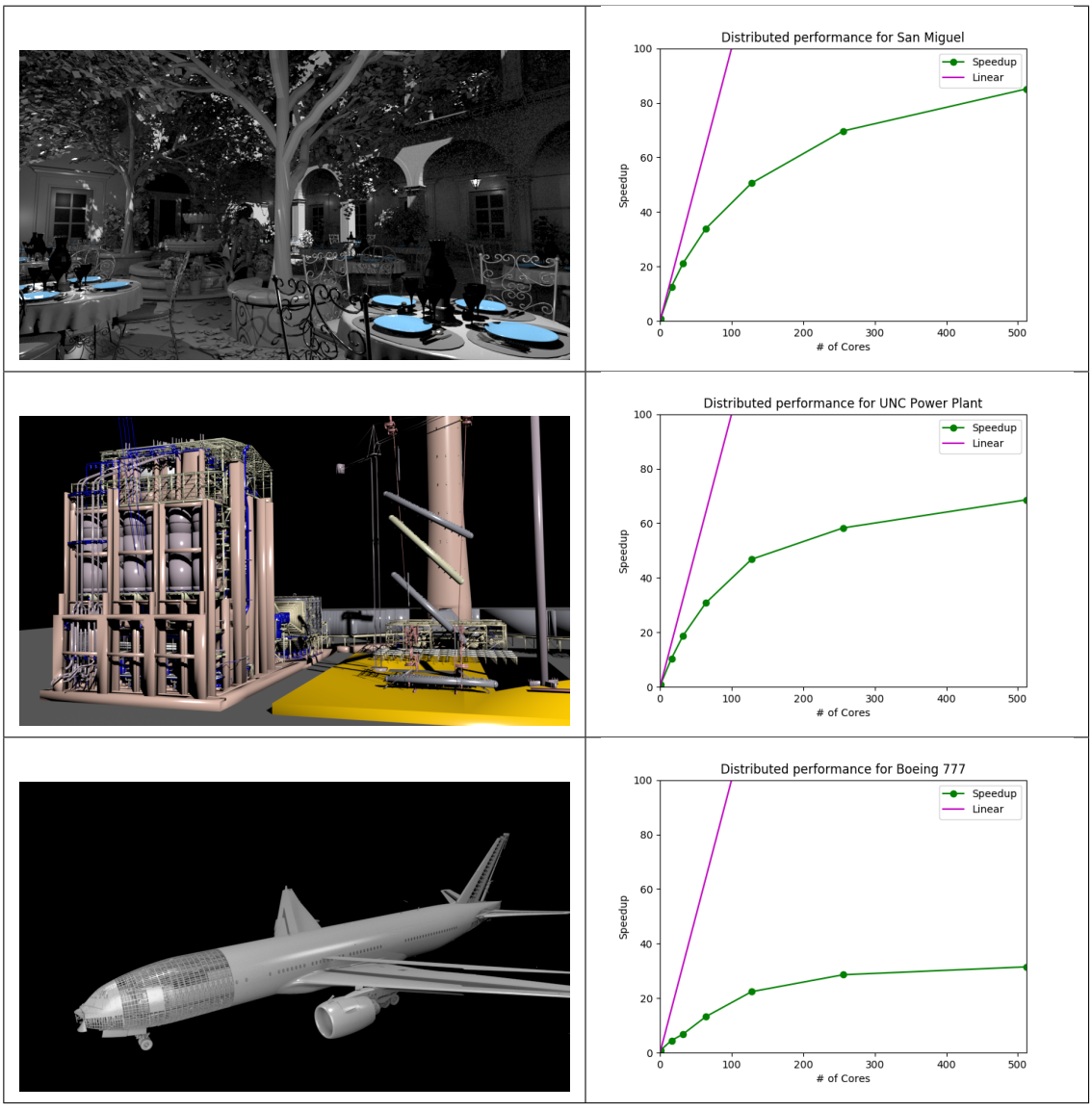

FIG. 4.10. Scaling for San Miguel (6.5 million polygons), the UNC Power Plant (12.7 million polygons), and the Boeing 777 (420 million polygons).

results do not exhibit linear scaling and performance decreases with the number of nodes. The plots in Figure 4.10 show that as more servers are added, our scaling increases sub-linearly but continuously through the maximum number of servers. Theoretically, for some large number of servers the amount of communication between nodes will dominate the entire rendering process and scaling will halt, but the trends amongst the data sets and number of nodes tested do not indicate where that would occur. In the Boeing model, the speedup is dominated by I/O. With so many individual objects that require a read from disk, the master server that handles all the file reads becomes the bottleneck.

With both the Power Plant and Boeing models, the static partitioning of the data leads to an imbalance of work amongst the servers. For the Boeing image, the server(s) assigned to the upper left of the image have only a small amount of or no geometry to process and thus remain idle for the majority of rendering. With the Power Plant image, the opposite is true: the servers assigned the area over the main power plant building handle the majority of the rendering. This imbalance could be remedied by allowing the servers to dynamically update their region of influence since all servers know about the distribution of objects.

Figure 4.11 shows scaling when using BDPT. As shown in Figure 4.8, the light paths generated for a particular server are nearly guaranteed to be communicated to a another server that handles the geometry surrounding a particular light source. As seen in the performance plots, the extra communication has a dramatic impact on scaling. For the San Miguel model, the increase in scaling from two servers to 16 is approximately two. This is due to the large number of light sources that require communication to nearly all of the servers for the light paths. The Power Plant model fares slightly better as there are only three light sources in this scene (above the camera, above the stairs on the left, and halfway down the corridor.

The final dataset, Figure 4.12, shows performance scaling for a test case using 36 replicated models of the Boeing 777. The geometry files for the single Boeing model are

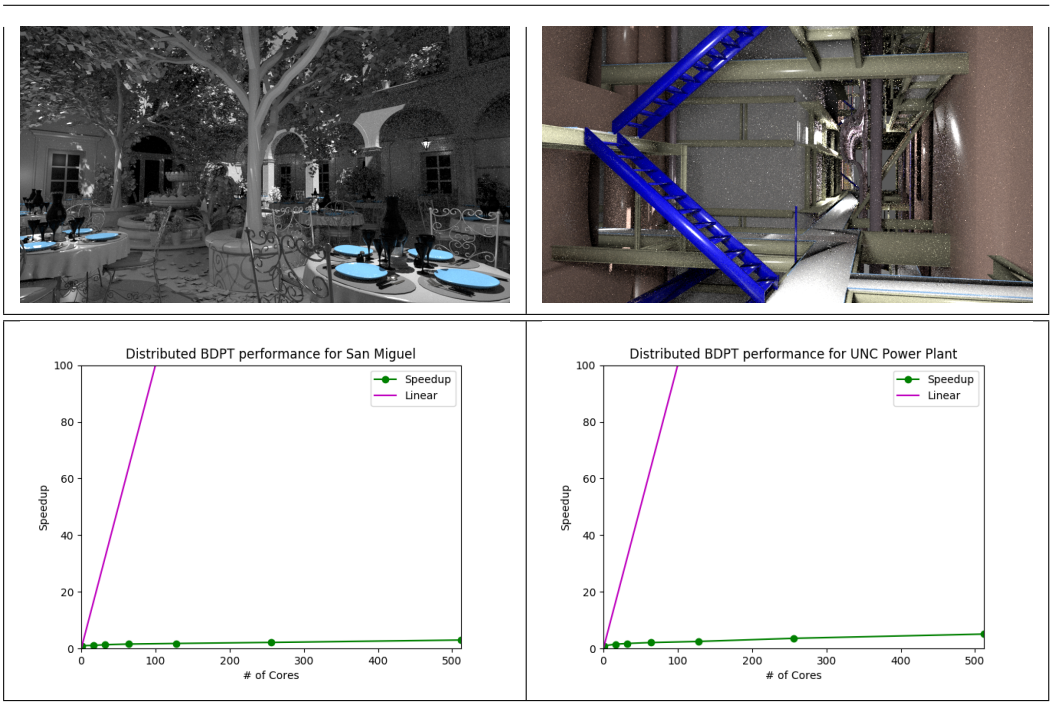

FIG. 4.11. Scaling for San Miguel and the UNC Power Plant using BDPT.

exceptionally large and require nearly 50GB of disk space. To limit the total impact on the file system, we modified the code so that we could use references in the RIB scene file, but the code treats each instantiation as if it were a separate set of data files. As shown in the performance plot, we achieve a continuous speedup out through 32 nodes.

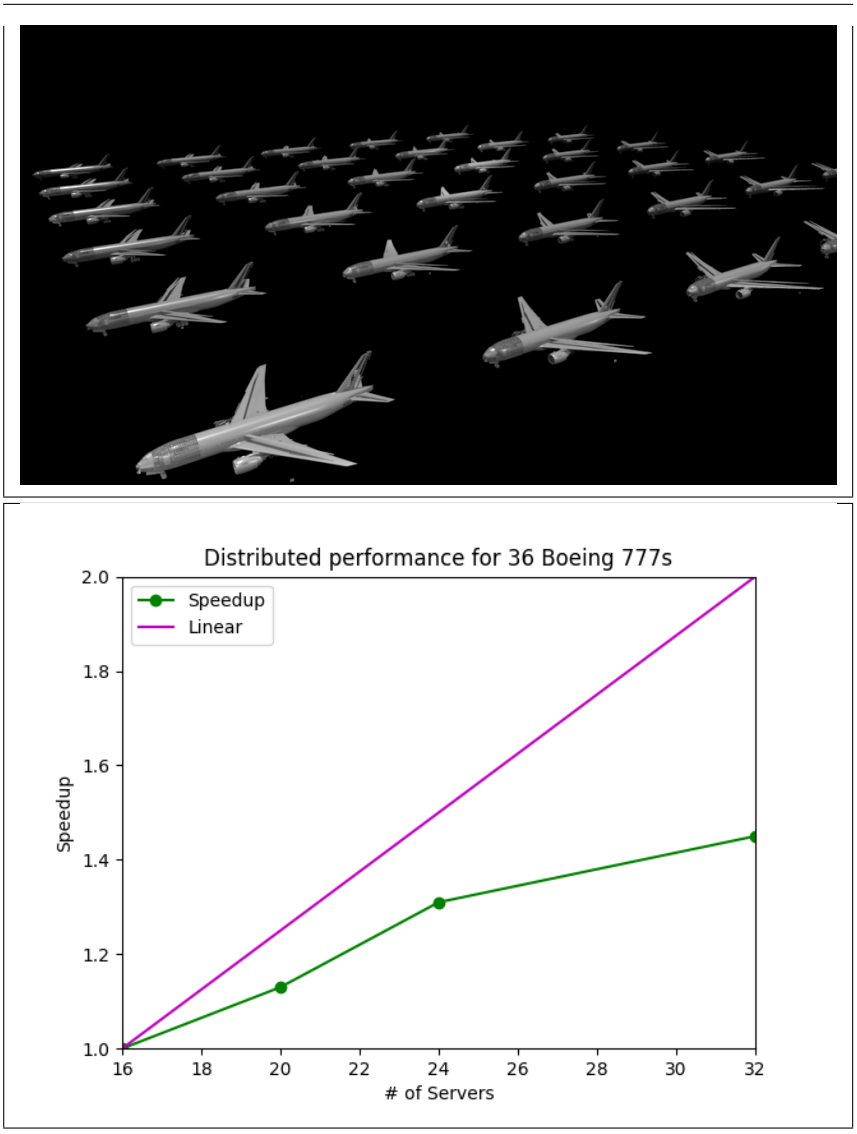

FIG. 4.12. Scaling for 36 replicated Boeing 777 models. The total polygon count exceeds 15 billion.

Chapter 5

DEFERRED COMMUNICATION AND GEOMETRY EVICTION

When rendering models of any significant size, out-of-core techniques are necessary for several reasons. First, the scene or model is too large to be loaded into the memory (monolithic or distributed) of the rendering system. Second, the depth complexity of the scene is deep. We define the depth complexity as the number of primitives hit by a single ray if it were to traverse the bounds of the entire scene. When a scene has high depth complexity as shown in Figure 5.1, only those objects closest to the camera would need to

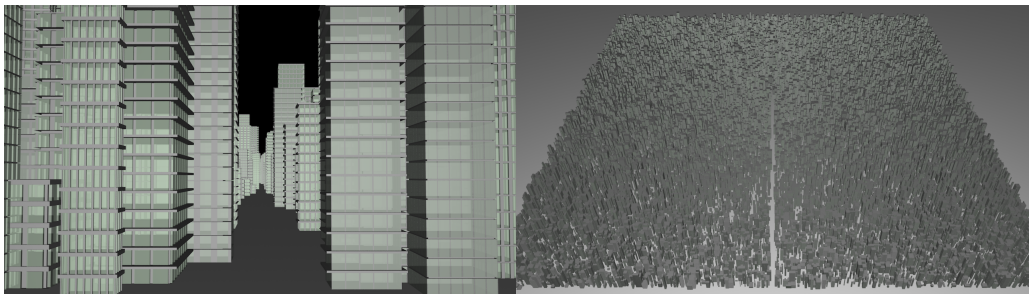

FIG. 5.1. Rendering the Figure on the left only requires a tiny fraction of the total geometry shown on the right to be loaded into memory. The point of view for the left image is from the bottom center.

be loaded. If the windows in the buildings of Figure 5.1 were to be made transparent, or

even completely removed, the rendered complexity of the scene would increase, but a large number of the buildings farther from the eye would still be occluded and therefore not need to be loaded into memory.

In the previous chapter, we presented an algorithm for rendering massive scenes distributed across a collection of servers. The scene to be rendered uses spatial subdivision to distribute the geometric elements among the servers. Up to the limit of a server's memory, each server contains bounding box proxies for all groups of geometric elements in the scene. When rendering begins, a server uses these proxies to decide whether to forward a ray to a remote server responsible for the actual geometry based on the intersection results with the local proxy element. As seen in Section 4.5, one issue with this approach is that the amount of communication between the servers limits the scalability of the algorithm.

In this chapter, we present an innovative data structure designed to minimize the inter-server communication by reusing previous intersection results from similar rays. This data structure consists of ray intersection probabilities that we call a *probability hit map (PHM)*. We store the PHMs with the geometric proxies (Shape_Proxy class) in the scene. When a ray intersects the proxy object, we look up the hit probability in the PHM based on the intersection point and a discretization of the incoming ray direction over a hemisphere at that point. If the hit probability is below the *miss probability threshold,* we reject the ray and return a miss. If the ray is above the *hit probability threshold* we perform a second lookup and return the average of the shading information computed from previous intersections. If the probability lies in-between the two thresholds, we proceed with the algorithm described in Section 4.4. The addition of this data structure increases the memory size of the proxy object, but is smaller than the majority of geometric objects represented by the proxy. In combination with evicting stale (least recently used) geometry, the PHMs reduce the total communications load and, thus, allows for the rendering of even larger scenes.

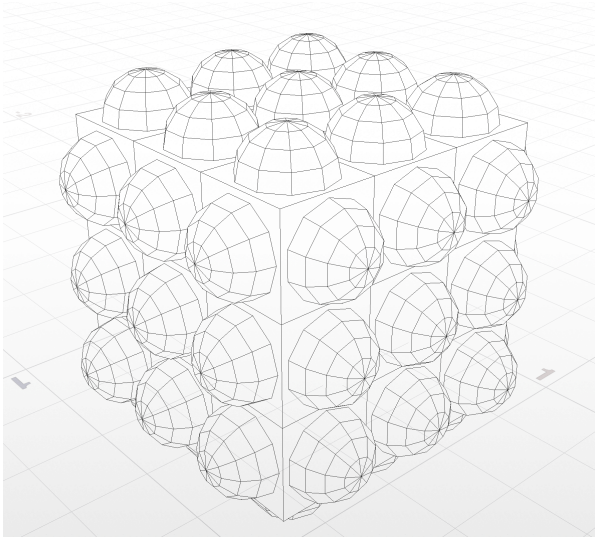

FIG. 5.2. The structure of the probability hit map

5.1 Probability Hit Map

The probability hit map is similar in structure to a cube map [32]. In cube mapping, a cube is placed around an object, and the scene is projected onto the six faces of the cube. At render time, the object generates a ray (usually an incoming eye ray reflected about the surface normal), computes the intersection point on a face, and uses those coordinates to look up the value stored in the texture element (texel). Additionally, the PHM has similarities to light field rendering and the Lumigraph [30, 57]. The main difference in the structure of the PHM and the light field work is that the light field authors discretize ray directions over the entire face of the cube while the PHM discretizes them over each texel. The probability hit map is structured as a low-resolution cube map, but each texel stores a discretization of incoming ray direction over a local hemisphere. Figure 5.2 shows a visualization of the PHM.

The PHM is associated with the proxy objects that represent geometry in the scene. When a ray traverses the scene and intersects a proxy, we determine the face that was hit

and discretize the ray's direction over the hemisphere normal to the face. The hit probability is accessed and compared against a bimodal threshold distribution. If the returned probability is lower than the miss probability threshold the ray is considered to have missed the geometry represented by the proxy. Similarly, if the returned value is higher than the hit probability threshold, the ray is considered to have hit the underlying geometry. If the returned value is in-between the two values then the algorithm proceeds as before with one difference. When the ray returns from intersection testing the underlying geometry, the PHM is updated and the probability for that ray direction-texel pair is recomputed. Upon a hit, the shading information for the ray direction-texel pair is updated by averaging the results returned with the ray. In the initial implementation, the shading information was replaced by the value from the last ray, but that resulted in significant variance, particularly when the underlying model had large variation in color mapping and normal directions.

We use the PHM in two different modes, but before we can describe those modes we first need to cover some background in Monte Carlo integration.

In ray tracing, we are trying to estimate the illumination at a point over the space of all light transport paths [49]. We do this by applying Monte Carlo techniques to estimate the following integral (following the derivation in [34]):

$$\mathbf{I} = \int_{\Phi} f(\mathbf{x}) d\mathbf{x}, \tag{5.1}$$

where $\mathbf{x} \in \Phi$ and $f : \Phi \to \mathbb{R}$ is a scalar function. For ray tracing, $\mathbf{I}$ is the pixel intensity to be computed, $f(\mathbf{x})$ is the energy carried by the photons along $\mathbf{x}$, and Φ is the space of light paths. Monte Carlo integration [61] is a numerical integration method that can provide an approximate solution to such an integral. The Monte Carlo method states that given a random variable $\frac{f(\mathbf{x})}{p(\mathbf{x})}$ with probability density function $p(\mathbf{x})$ and a set of independent samples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbf{X}$, the expected value of the random variable is:

$$\mathrm{E}[f(\mathbf{X})] = \int_{\Phi} f(\mathbf{x})\, p(\mathbf{x})\, d\mathbf{x}, \tag{5.2}$$

By reformulating our original integral, we can compute its estimate as follows:

$$\mathrm{E}\left[\frac{f(\mathbf{X})}{p(\mathbf{X})}\right] = \int_{\Phi} \frac{f(\mathbf{x})}{p(\mathbf{x})} p(\mathbf{x})\, d\mathbf{x} = \int f(\mathbf{x}) d\mathbf{x} = \mathbf{I} \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} \tag{5.3}$$

A Monte Carlo estimator with N samples, $f_N(\mathbf{X})$, is *consistent* if the estimate converges to the correct solution with an infinite number of samples. Therefore, a *consistent* estimator satisfies:

$$\lim_{N\to\infty} \Pr\left[\int_{\Phi} f(\mathbf{x}) d\mathbf{x} - f_N(\mathbf{X}) = 0\right] = 1, \tag{5.4}$$

where $\Pr[F]$ is the probability that F is true. An *inconsistent* estimator is one that does not satisfy equation 5.4.

A Monte Carlo estimator $f_N(\mathbf{X})$ is *unbiased* if the expected value of the estimate is equal to the correct value:

$$\mathrm{E}[\mathrm{f_N}(\mathbf{X})] - \mathrm{I} = 0 \tag{5.5}$$

A biased estimator is one that never satisfies equation 5.5. It is generally preferred to have an unbiased estimator in Monte Carlo rendering, it is not uncommon to use a biased estimator to improve speed or reduce variance.

With this background we can now describe two modes for the PHM, consistent and inconsistent. Because of the nature of the PHM both of these modes are biased estimators of the actual lighting solution. Kirk and Arvo [50] described a method for unbiased sampling by using one set of samples to compute the value of the integral we are estimating, and

a second set of samples to update the estimate or compute the variance. In that vein, we used a biased strategy for the probabilistic continuation of ray paths when the value in a cell of the PHM has moved into the miss/hit threshold ranges. At that point, for every ray that passes through the thresholded cell, we generate a random number and test against a threshold. If the value exceeds the threshold, we generate a random ray through the cell, run the intersection test, and update the PHM. In the second mode, once the value in the cell has triggered one of the miss/hit thresholds, we use the returned PHM value for all subsequent rays. When a majority of the cells have triggered the threshold, we evict the geometry.

Each method has its own set of advantages. The consistent method will eventually converge to the correct average value for a PHM cell. But more importantly, if a PHM cell contains a thin emitter that has been missed by the initial samples, with enough samples the consistent method will eventually discover the emitter, and force the PHM to use the methods from the previous chapter. The inconsistent method will never discover geometry missed in the initial sampling. However, the inconsistent method allows us to evict geometry once a majority of the cells have exceeded the threshold value allowing the ability to render larger models.

These new additions to the proxy objects increases its memory footprint dramatically. For the smallest configuration of a PHM, nine texels per face (a three-by-three layout) and 12 hemispherical cells for directions, each proxy will require at least an additional 10,000 bytes. And while the PHM is memory intensive, it is fairly simple computationally. If we compare the PHM with other intersection reduction techniques such as a k-DOP (k-sided Discrete Oriented Polytope. A bounding box is a 3-DOP), the PHM has a number of advantages:

1. A k-DOP is computationally expensive. A k-DOP first tests against the proxy bounding box, and upon a hit then checks the ray against each of the k slabs. The PHM

simply needs to retrieve the value associated with the direction of the ray and the hit point.

2. Since a k-DOP is simply a convex hull, an object with holes, i.e. an open cylinder or a toroidal shape, will still have a need for a full intersection testing, while the PHM can capture ray misses due to the holes.

3. A k-DOP has limited scalability. There is a point with a k-DOP where additional planes provide no additional useful information, whereas a PHM can scale with an increase in hemisphere and hit point sampling.

4. If the k-DOP is intersected, we still have to communicate the ray to the remote server while the PHM will only probabilistically communicate the ray, with the majority of the time returning the averaged result.

5.2 Initialization

While most of the components of our target models are still heavier in memory than a PHM when fully instantiated, the memory footprint of the PHM places a burden when rendering many small or simple objects. To address this problem, all proxy objects respond to probability queries, but only proxy objects with large geometry will have a PHM. For those proxies without a PHM, they will return the value 0.5 for all probability queries as this forces the renderer to use the original algorithm. The renderer will not allow setting of the miss or hit probability values above or below 0.5 respectively, guaranteeing that the original proxy will behave as in Section 4.4. As geometry is read from disk, simple objects such as quadric shapes or single polygons and patches are instanced without a PHM, and aggregate objects are instantiated with a PHM.

The initial PHM for an object is seeded with a weighted probability of 0.5. Using a

weighted value reduces the variance in the probability of a hit/miss in the early phase of rendering. The initial weight chosen was 100 samples as that requires a minimum of an additional 800 samples of either pure hit or pure miss before the hit probability was above 0.9 or the miss probability below 0.1. Using a smaller number of initial samples resulted in too much variance in the final image, and higher values only converged near the end of a render.

5.3 Dynamic Updating

As rays are traced through the scene, the PHM for a particular proxy is updated in several places: the true proxy that contains the geometry, and its representatives on the other servers. As stated earlier, when a ray is forced to intersect the underlying geometry of a proxy, the PHM is updated with the return hit or miss status. For the proxy that lies outside a servers domain of influence, its PHM is updated from rays originating on this server, and secondary or shadow rays that are passed to the server. As a consequence, the representative proxies' PHM will drift in value from the proxy that controls the geometry (master proxy). When the master proxy finishes a bucket, it will send an updated PHM back with any rays that intersected the proxy. By attaching the updated PHM to the return rays, communication is limited to only those servers that are accessing the PHM.

5.4 Results

To test the validity of the probability hit map, we used two data sets, San Miguel and the two Boeing 777 scene from the previous chapter. Since the PHM is designed to reduce communication of rays between servers, we made the PHM tunable for ray type and invisible to primary rays. In the current design, primary rays are traced only on the originating server, therefore, using the PHM for primary rays does not reduce the communication, and

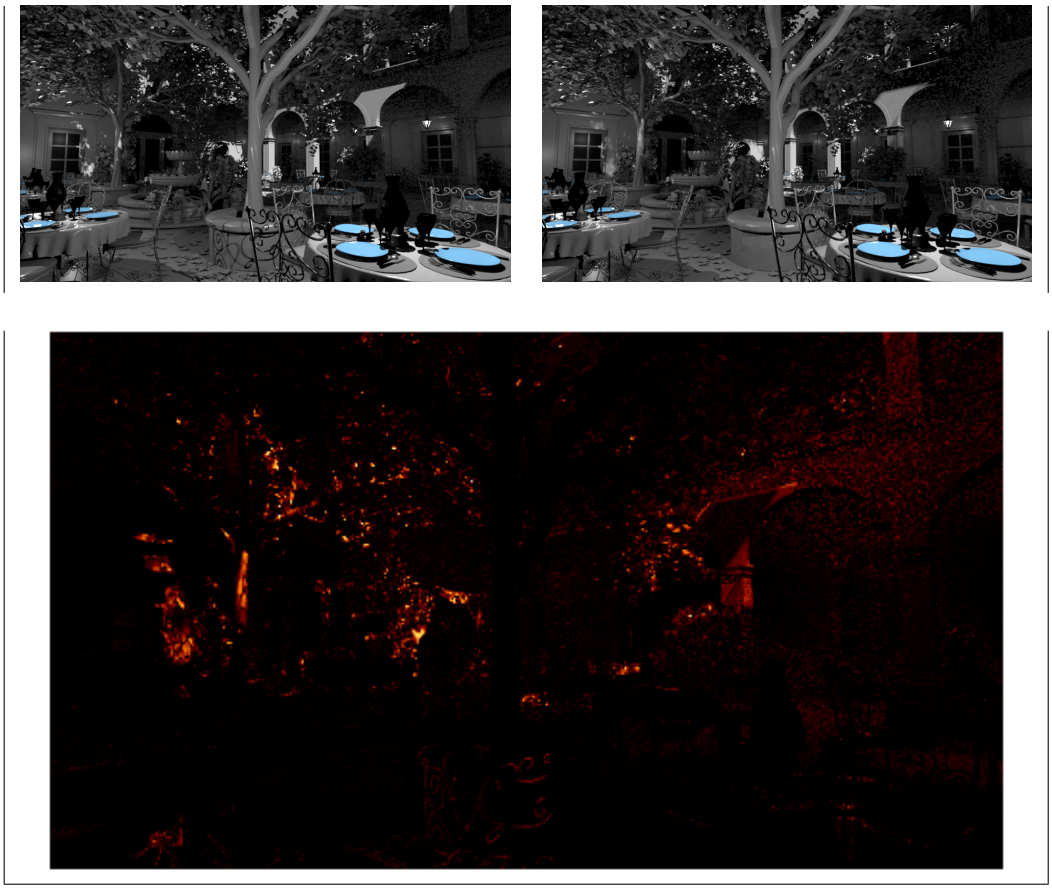

FIG. 5.3. Comparison of San Miguel rendered with and without the PHM. The leftmost figure is the original image from Chapter 3. The right image is rendered using the PHM, and the bottom image plots the difference..

simply reduces the visual quality of the rendered image. For the following results, we used 16 primary rays per pixel, one shadow ray for distant light sources, and between 10 and 100 rays for area lights.

Figure 5.3 compares the San Miguel scene rendered with and without the PHM. One interesting side effect of the separation of disconnected objects discussed in Section 4.4.2 is that, after running the connected components algorithm, more than 90% of the geometry falls below the large object threshold that was set for attaching a PHM to the proxy. For this test, we reduced the threshold so that, in addition to the building being PHM enabled, the

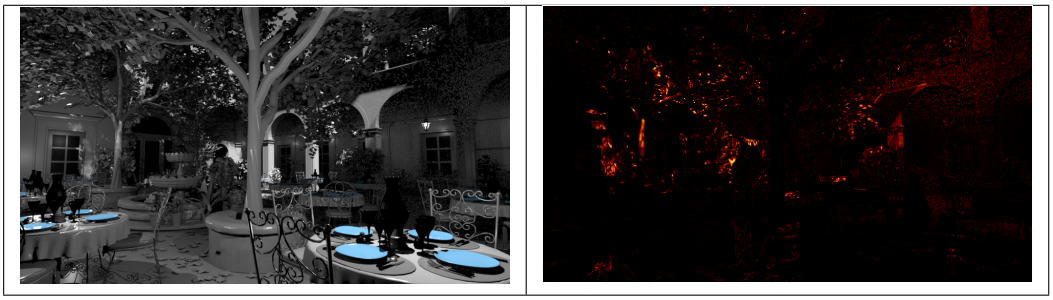

FIG. 5.4. Comparison of PHM enabled San Miguel. The inconsistent algorithm is on the left and the difference compared to Figure 5.3 is on the right.

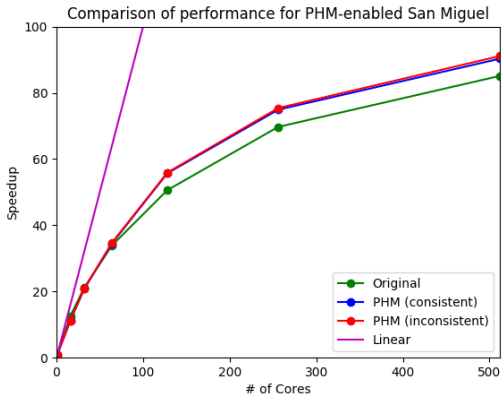

FIG. 5.5. Comparison of PHM performance for San Miguel versus the standard algorithm from Chapter 3.

trees, tables, and chairs are also enabled, but not the dinnerware on the tables. As shown in the figure, the main visual impact of the PHM is in the sharpness of the shadows. Since there is virtually no reflection in the scene, the PHM only has an effect upon the shadow rays.

Figure 5.4 compares the results of using the two different algorithms, consistent and inconsistent. The main difference between the images is the popping of highlights and shadows in the inconsistent algorithm, most notably on the small shadow glitch on the central planter and in the noisy lighting on the upper walls visible through the trees.

Finally, Figure 5.5 compares the performance for the PHM-enabled San Miguel scene

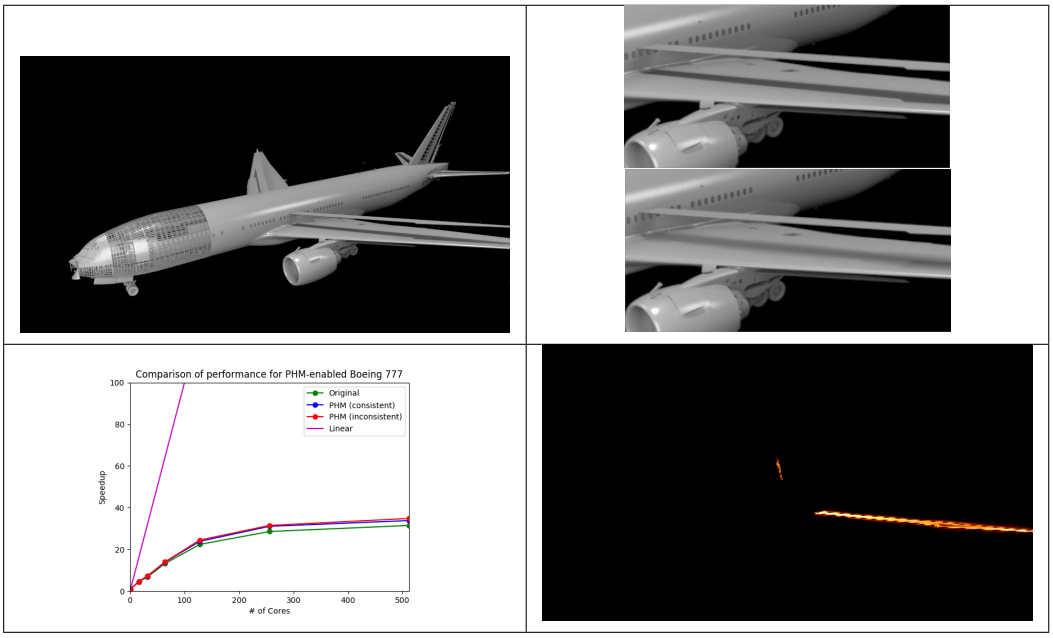

FIG. 5.6. The single 777 scene using PHM. The image on the right shows a zoomed in view of the original (top) and the PHM enabled version (bottom). The bottom left image plots performance and the bottom right show the difference.

versus the scaling results from Chapter 3. As expected, the performance gains from the PHM-enabled scene become apparent as the number of servers increase. Additionally, the PHM-enabled scene had an approximately 5% decrease in its memory footprint overall, and a 7.3% reduction in the total number of rays transmitted for the largest server configuration. The memory reduction is due to the PHM occluding geometry along the ray that would have been traversed and loaded without the PHM. We see additional performance gains due to a reduction in the total number of intersection tests once the PHM is active. For the inconsistent version of the algorithm, there is only a 1.6% reduction in memory once geometry starts to be evicted. The number is low due to the current implementation only evicting geometry once all the cells have triggered the miss/hit thresholds.

For the two Boeing 777 scenarios, after partitioning, approximately 30% of the objects are PHM enabled. For the single model scene (see Figure 5.6) with the consistent

algorithm, the memory footprint decreased by 3.8%, and for the 32 server configuration there was a 9.4% reduction in rays transmitted. With the inconsistent version, less than one percent of the geometry is evicted with a 12.3% reduction in total ray traffic. For the 36 model scene using the consistent algorithm, the memory footprint decreased by 8.4% with a total reduction in ray traffic of 15.5%. For the inconsistent version, approximately one percent of the geometry was evicted with a total reduction in communication of 18.2%.

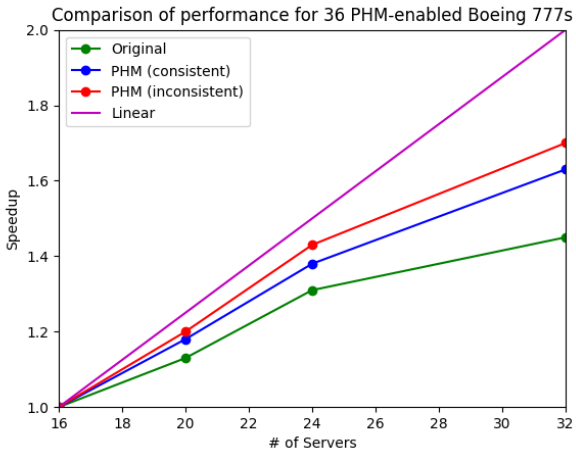

FIG. 5.7. Scaling for 36 Boeing models with PHM. We did not include a difference image as the results are indistinguishable from the original at the rendered resolution. See Figure 5.6 to see the difference between the PHM and original models.

Chapter 6

DESIGNING A RENDERING SYSTEM FOR MASSIVE MODELS

Early in development, a software engineer needs to make a decision about the primary focus of their renderer, do we design for capability (feature set), or do we design for performance? These decisions are not mutually exclusive, no one designs for feature set and being slow, but they do affect the decisions made throughout the development of the renderer. The commercial rendering systems Arnold [44], mental ray [46], and RenderMan [43] all were initially designed for capability with subsequent releases adding new features and improving the performance of existing ones. Other rendering systems such as OptiX [42] and Embree [92], were designed to provide the highest possible performance from a limited feature set. Later releases slowly added new capabilities, but these releases focused on improving performance through algorithm improvements, or exploiting improved capabilities of newly released hardware, e.g. improved CPU performance, an increase in GPU capability, or new exotic hardware such as the Xeon Phi [45].

In the rendering system proposed for this dissertation, a different approach was taken: design the renderer for massive models while retaining an advanced feature set akin to those found in commercial renderers. This choice affected every design decision and implementation detail throughout the system. What is chosen to store, what to re-compute, and what

to communicate? How do we index into data structures whose element count can exceed the size of an integer? Would the choice of a programming language and its surrounding ecosystem make a difference in the ability to handle large models?

Throughout the development of this rendering system, the questions above and many others were asked, analyzed, and answered in someway to facilitate the rendering of massive models. A set of the more important decisions made and their consequences are discussed in the sections below.

6.1 Choice of a programming language

The original design and implementation of a large model rendering system was in the programming language Eiffel [62]. Eiffel was designed around the idea of provable program correctness. While automatic provability is not currently feasible, Eiffel took many steps in its design to give software engineers language features that codify their assumptions about a class, a feature, and data elements. Collectively, these concepts are called *Design By Contract.* Figure 6.1 shows a brief example of the structure of an Eiffel program. In Eiffel, **require** is a pre-condition for the execution of the feature/routine. It is a contract to the caller of the routine that states that if these conditions hold, i.e., n is greater than 0, then the feature guarantees that the post-conditions, the **ensure** clause, will also be true. An **invariant** is a statement that must be true before a feature is called, and remains true on exit. These features together have the effect of pushing the discovery of errors very early into the design cycle, requiring less debugging further on.

Eiffel is also a garbage collected language like Java, thus freeing the programmer from tracking the lifetime of objects. One of the first problems encounterd was that the garbage collector had a large negative impact on performance as the number of threads increased. The garbage collector uses a mark and sweep algorithm: starting from global

pointers in the heap, the algorithm follows all pointer references marking the objects as valid. The second phase identifies all unmarked objects in the heap as garbage and collects them for reallocation. In our algorithm, since objects could be shared across threads, the thread that invoked the garbage collector locked all the other threads until the collector finished. And, the collector may be invoked by each thread. We were able to remove much of the penalty introduced by the collector by explicitly marking the sections of code where it could run, thus defeating the purpose of not having to track object lifetimes. The main decision point on whether to continue developing in Eiffel came when testing the performance of loading ten million objects from disk and found a 20 times speedup when using a file reader written in C++.

The code base was converted to C++11 [1] over a period of eight weeks. Through custom macros and helper functions the system retained most of the pre- and post-conditions from the original Eiffel code base. The system now has the benefits that come from a high-performance language such as C++, but with a higher cost in development time and debugging. As in

```
class
    STACK [G]
feature -- create
    make (n: INTEGER_32)
        -- Allocate list with `n' items.
        -- (`n' may be zero for empty list.)
        require
            valid_number_of_items: n >= 0
        ensure
            correct_position: after
            is_empty: is_empty
feature -- Access
    has (v: like item): BOOLEAN
        -- Does current include `v'?
        ensure
            not_found_in_empty: Result implies not is_empty

    item: G
        -- Current item
        require
            not_off: not off
            readable: readable

feature -- Measurement
    count: INTEGER_32
        -- Number of items.
        ensure
            count_non_negative: Result >= 0
...

invariant
    -- from DISPENSER
    readable_definition: readable = not is_empty
    writable_definition: writable = not is_empty

    -- from ACTIVE
    writable_constraint: writable implies readable
    empty_constraint: is_empty implies (not readable) and (
         not writable)
```

FIG. 6.1. Eiffel-based implementation of a stack data structure

all C++ code, we now must explicitly track
the lifetime of objects to prevent memory leaking, but we minimize the impact using C++ smart pointers.

6.2 Smart Pointers

Smart pointers are one potential implementation of reference counting. A smart pointer stores a counter that is incremented and decremented as the object is accessed and released. When the last reference to the object is removed, the counter goes to zero and the object is deleted. The C++ std::shared_ptr is one variant of a non-intrusive reference counter, where the counter is associated with the accessor and not with the object itself. The standard implementation of a shared pointer is 16 bytes, eight bytes for the actual pointer and eight bytes for a pointer to a control block that stores the counter among other elements. The other type of reference counting, an intrusive pointer, stores the counter with the class and the intrusive pointer itself has no overhead, it is the same size as a normal pointer.

For this system, we chose to use an intrusive pointer for any class that was to be shared. This design decision was driven by one simple issue, the memory overhead of the non-intrusive smart pointer. The system described in this dissertation relies heavily on smart pointers to reduce the overhead on the programmer to track object lifetimes. As a consequence, there are significantly more smart pointers than actual objects. For example, as described in Section 4.4.3, geometry objects are sorted into grid cells based on their screen space footprint, and a reference to the object is stored in each grid cell it overlaps. When a one billion element model is sorted into grid cells and if it has an overlap of two cells per element, then we need an additional eight gigabytes[1] of memory to store the reference to the counter. In practical terms, the overlap of the grid cells is dependent on the

[1]Throughout this section when referring to a gigabyte we are using the IEC definition of 1 gigabyte as 10^9 or 1,000,000,000 bytes.

projected size of the geometry elements and their distribution, so the ratio of pointers to objects may not be two, but is always greater than one. Testing over a number of different scenes and camera positions, on average there was a 25% - 35% reduction in total allocated memory by switching to an intrusive pointer. In Appendix A.1 is the definition of the C++ mixin class that is inherited by every class that is tracked through a smart pointer.

6.3 64-bit Indexing

One consequence of designing for large models is that eventually the number of elements that need to be addressed exceeds the capacity of a 32-bit word. This fact ripples through the code base in that every variable that indexes into the model needs to be 64-bit. A 32-bit integer can address approximately two billion elements, an unsigned 32-bit integer can address four billion elements, and the model of the USS Yorktown in the recent movie, Star Trek Beyond, exceeded 1.3 trillion polygons [20]. So, 64-bit integers are necessary for indexing into large models. However, blindly converting all indices to 64-bit integers can lead to memory explosion. In our renderer, we implement polygons in terms of the *Bilinear_Patch* class. All of the polygons of a model defined in the input file as an archive[2] will share the lists of points, colors, and normals. So, our *Bilinear_Patch* must use 64-bit integers to index into the lists. Triangles, and quads with a degenerate point, are both represented by bilinear patches. When the parser encounters either a triangle or a degenerate quad, it reorders the vertices to define the missing or degenerate vertex as the last element. Since the offset to the starting reordered vertex is no greater than three (zero-based indexing of four elements), we use the smallest type available to us, uint8_t or a single byte. For our hypothetical billion element model, if we assume the elements are polygons, this optimization saved three gigabytes of memory.

[2] In RenderMan vernacular, these can either be Archives, or Procedural2 “DelayedReadArchive”

6.4 Data Structures

It is critical when designing a rendering system for large models that every consideration is taken to optimize for memory consumption. In a typical ray tracer that is optimized for performance, choices of computation versus memory are nearly always resolved by storing a pre-computed value. For a large model renderer, the opposite is true; memory is our most critical resource. A classic technique in ray tracing to reduce the number of intersection tests is mail boxing. When an object spans multiple cells of a acceleration structure, mail boxing is used to prevent the same ray from testing that object multiple times. Each ray is assigned a unique identifier (typically an incrementing integer) that is stored in an object's mailbox. For each ray-object intersection test, the ray's identifier is compared against the value stored in the mailbox and the test is rejected if the two values are the same. In a multi-threaded environment, where an object can be tested by multiple rays simultaneously, one mailbox is required per thread. For our one billion element model rendered with eight threads, the amount of memory used for the mailbox alone is 64 gigabytes. Our system uses an acceleration structure that minimizes the need for multiple intersections testing, but where necessary it simply retests the ray.

In C++, the first place a programmer looks for data structures is the Standard Template Library (STL) [70]. The STL is a set of classes for C++ that provide many of the fundamental programming elements used in modern software, i.e., lists, queues, maps, and the algorithms that operate on them. After conversion from Eiffel, the initial updates of the C++ code used std::vector as the base type for lists of points and other attributes, e.g., colors, normals, and texture coordinates. Initial testing proved that the STL data structures were indeed robust and high performance. With large model testing, similar problems to those with smart pointers became apparent. A typical implementation of std::vector stores three values, a pointer to the actual data, the allocated size of the data (capacity), and the

number of elements (count), for a total of 24 bytes per vector. For our one billion element model, if stored as a single archive, the rendering system allocated four vectors (points and attributes) for the model, and pointers to those vectors for each element. If our model was defined as one billion individual elements, for example, points in a particle system, the system would allocate the four vectors per element along with the pointers to those elements. The vectors were eventually replaced in the renderer with a custom vector-like class that embeds the count and capacity in the class, and enables the use of intrusive pointers. In the large model, single archive case, these is no savings in memory for the custom vector versus using the standard vector, with the exception that we now have the convenience of a smart pointer at no extra memory. In the one billion individual element case, we have a savings of eight gigabytes since we no longer have to allocate the extra memory from the vector overhead.

The last memory optimization applied to the data structures was to reorder elements to improve compactness. Storage for basic C/C++ datatypes on an x86 processor have alignment requirements: chars can start on any byte address, but everything else must be *self-aligned,* two-byte types such as shorts must start on an even address, four-byte types must start on an address divisible by four, and eight-byte types (longs, doubles, and pointers) must start on an address divisible by eight. The reason is that this allows the compiler to generate single instruction memory fetches for faster access. Some compilers allow the use of a pragma, *pack,* to align the datatypes of a class/struct to a specified boundary, typically one. On older processors, and under the right conditions on an x86, unaligned datatypes generate an illegal instruction. It is illegal to pass unaligned structures to modern vector instructions such as SSE or AVX.

When the compiler encounters elements in a class with different self-alignment requirements, it will introduce padding between the elements to satisfy the alignment. By rearranging the elements in a class, we can minimize the amount of padding between the

elements. Most compilers have a flag similar to clang's -Wpadded which will issue a warning whenever a class/struct is required to have padding, but it can be cumbersome to identify the offending element when C++ inheritance is involved. Instead, we used two different tools, *pahole* on Linux, and a python script, *struct_layout,* on OS X. These tools not only identify where padding occurs in a class (including from inheritance), but can also make suggestions on how to minimize the padding. Figure 6.2 shows the before and after layout using struct_layout on one descendant of the Geometry class. Overall, these tools reduced padding in the descendants of the Geometry class by 1-24 bytes, resulting in a large memory reduction in a large model renders.

```
1  struct ::Bilinear_Patch [200 Bytes]
2       : [Geometry : 128] <base-class>
3       :   [Shape : 64] <base-class>
4       :     [Primitive : 40] <base-class>
5       0:       [<fun_ptr>** : 8] _vptr$Primitive         -- {cache-line 0}
6       :        [Referenced<Primitive> : 4] <base-class>
7       :          [atomic<int> : 4] _counter
8       :            [base_atomic<int, int> : 4] <base-class>
9       :              [aligned : 4] m_storage
10      8:                [int : 4] value
11     --- 4 Bytes padding ---
12      :        [intrusive_ptr<Transform> : 8] _object_to_world
13     16:          [Transform* : 8] px
14     24:        [long unsigned int : 8] _id
15      :        [intrusive_ptr<Light> : 8] _area_light
16     32:          [Light* : 8] px
17      :      [Memory_Traits : 8] <base-class>
18     40:        [<fun_ptr>** : 8] _vptr$Memory_Traits
19      :      [Bit_Field : 1] _bool_flags
20     48:        [unsigned char : 1] a
21     48:        [unsigned char : 1] b
22     48:        [unsigned char : 1] c
23     48:        [unsigned char : 1] d
24     48:        [unsigned char : 1] e
25     48:        [unsigned char : 1] f
26     48:        [unsigned char : 1] g
27     48:        [unsigned char : 1] h
28     --- 1 Bytes padding ---
29     50:      [unsigned short : 2] _bucket_id
30     52:      [int : 4] _last_bucket_id
31      :      [intrusive_ptr<Attribute_Wrapper> : 8] _attributes
32     56:        [Attribute_Wrapper* : 8] px
33
34 ...
35
36      :    [intrusive_ptr<Material> : 8] _material
37    120:      [Material* : 8] px
38    128: [unsigned char : 1] _offset                     -- {cache-line 2}
39     --- 7 Bytes padding ---
40      : [intrusive_ptr<Real_Array> : 8] _coordinates
41    136:   [Real_Array* : 8] px
42    144: [long unsigned int : 8] _index
43    152: [long unsigned int[4] : 32] _indices
44      : [Vector : 12] _p_error
45    184:   [float : 4] x
46    188:   [float : 4] y
47    192:   [float : 4] z                                 -- {cache-line 3}
48     --- 4 Bytes padding ---
```

FIG. 6.2. Structure Alignment. This is the output from running struct_layout on the Bilinear_Patch class. Lines 11, 28, 56, and 65 show a total of 16 bytes of padding introduced by type misalignment. By moving the variable *_offset* at line 56 to after line 64 we remove the seven bytes of padding and reduce the final four padding bytes by one, for a savings of eight bytes for each instance of this class.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this dissertation, we demonstrated two methods that advance the rendering of large models, and the design choices made to implement them. Working under the assumption that the models are large enough that moving across file systems is impractical, we demonstrated an inline method of partitioning the models that reduces the in memory footprint without increasing disk consumption. While the partitioning algorithm was designed to improve the memory footprint of the models, it provided modest performance gains in rendering as well.

We created a Reyes-style parallel micropolygon renderer and tested three different algorithms for distributing the workload across threads. After analyzing the results of the three algorithms, we created a forth algorithm that combined the best aspects of the original three. Even though none of the algorithms worked well for all the scene types and geometry configurations, the hybrid algorithm had the best performance under most conditions.

We presented a fully distributed, parallel implementation of a ray tracing system. One bottleneck in this type of system is reading the scene description for a massive model. The proposed system parallelized the processing, partitioning, and distribution of the scene description as it is being loaded from data store. All objects in the scene are represented

by a proxy object, and, only when the proxy is intersected by a ray, is the data loaded. Each server in the parallel system renders its portion of the scene, and, when rays intersect objects outside of the server's domain, the rays are queued and distributed directly to the server responsible for the object. We showed that with this fully distributed architecture the scaling increased with the number of servers, even for a 15 billion polygon model.

Next, with the high communication cost in the initial design, we needed to reduce the total amount of data transmitted to and between the servers. We presented a unique stochastic map that reduces the number of rays transmitted and scene objects loaded. The map stores a probability of intersection over incoming ray directions at the location of intersection on the proxy object. As rendering progresses, when a ray intersects the proxy object, it is tested against the map and returns the average shading information with the ray if the value for that ray direction is above or below a bimodal threshold. Otherwise, the ray continues as previously described, and the map is updated by the returned value from the ray. We showed that with this modification the scalability of the system increased at a cost of additional variance in the result.

The final contribution is a discussion of the design decisions required to develop a rendering system for large models. In particular, we showed how the choice of a programming language can impact the overall performance of the system. Analysis of every aspect of memory usage within such a system is crucial to handling very large models.

7.2 Future Work

While this dissertation presented a unique set of solutions to the problem of rendering large models, there are still a number of interesting questions that we would like to address in future work. Listed in no particular order of importance:

- What happens when the number of lights exceed the capacity of a render server?

We make the assumption that the lights can be replicated on every server to try to minimize the light sink issue, but this methodology would break down with a scene consisting of millions of lights, e.g., a particle system where every particle glows or, if the light sources were attached to heavy geometry that consisted of millions of polygons.

- How would tracing ray packets for primary and secondary distributed rays change performance? Modern ray tracers [21, 92] bundle rays together to exploit coherence between rays shot in a similar direction, called ray packeting. The issue is that secondary rays have almost no coherence. In the system proposed in this dissertation, rays are queued for a particular server, implying that we could reorder the rays as they arrive in the queue to be able to trace ray packets on the server containing the geometry.

- A better method for updating PHM shading info. The current methodology for updating the PHM is to average the results over the intersection information gathered from ray that intersected a particular PHM cell. We believe that a better method is to implement the ray interpolation methodologies from the light field papers [30, 57].

- We currently initialize the PHM with a fixed probability value. The value could be improved by projecting the geometry onto the PHM during the initial parsing and loading. By initializing the PHM in this manner, we may be able to reduce communication further by deferring loading of the geometry.

Appendix A

SUPPORTING MATERIAL

A.1 Intrusive Pointer

Intrusive pointers are a version of a smart pointer where the reference count is included in the object itself, rather than in the pointer (a C++ smart pointer). In our rendering system, the class listed in A.1 is inherited by every class that should have a shared reference.

```
#ifndef _REFERENCED_H_
#define _REFERENCED_H_

#include <boost/atomic.hpp>
#include <boost/checked_delete.hpp>
#include <boost/intrusive_ptr.hpp>
#include <boost/interprocess/detail/atomic.hpp>

/**
 * Referenced.
 **/

template< class T >
class Referenced
{
public:

    // Initialization
    Referenced(): _counter(0) {}

    // Duplication
    Referenced( Referenced const& c ): _counter(0) {}
    Referenced& operator=( Referenced const& c ) { return *this; }

    // Destruction
    ~Referenced() {}

    // Access
    boost::uint32_t ref_count() const { return _counter; }

    boost::intrusive_ptr<T> self()
       { return boost::intrusive_ptr<T>((T*)this); }

    boost::intrusive_ptr<const T> self() const
    { return boost::intrusive_ptr<const T>((T const*)this); }


    // Inapplicable
    friend void intrusive_ptr_add_ref( Referenced< T > const* s )
    {
        assert( s != 0 );
        s->_counter.fetch_add( 1, boost::memory_order_relaxed );
    }

    friend void intrusive_ptr_release( Referenced< T > const* s )
    {
        assert( s != 0 );
        if ( s->_counter.fetch_sub( 1, boost::memory_order_release) == 1 )
        {
            boost::atomic_thread_fence( boost::memory_order_acquire );
            boost::checked_delete(static_cast< T const* >( s ));
        }
    }

private:
    ///should be modifiable even from const intrusive_ptr objects
    mutable boost::atomic< boost::uint32_t > _counter;

};  // end of class Referenced


#endif // _REFERENCED_H_
```

FIG. A.1. Mixin class for enabling intrusive pointers. Every class that is to be referenced from a boost intrusive pointer inherits from this class.

REFERENCES

[1] I. J. S. 22. Information technology – Programming languages – C++. Technical Report 4, ISO, Dec. 2014.

[2] A. T. Afra. Interactive Ray Tracing of Large Models Using Voxel Hierarchies. *Computer Graphics Forum*, 31(1):75–88, 2012.

[3] T. Aila, T. Karras, and S. Laine. On Quality Metrics of Bounding Volume Hierarchies. *ACM Trans. Graph.*, 32(4), 2013.

[4] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.

[5] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., 1999.

[6] M. A. Bolstad. Parallel methodologies for a micropolygon renderer. In *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization*, PGV '14, pages 17–24, Aire-la-Ville, Switzerland, Switzerland, 2014. Eurographics Association.

[7] Boost. Boost C++ Libraries, 2016.

[8] T. Boubekeur and M. Alexa. Technical Section: Mesh Simplification by Stochastic Sampling and Topological Clustering. *Comput. Graph.*, 33(3):241–249, June 2009.

[9] Y. Chiang, R. Farias, C. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphic*, pages 59–66. IEEE Press, 2001.

[10] Y. Chiang, C. Silva, and W. Schroeder. Interactive out-of-core isosurface extraction. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 167–174. IEEE Computer Society Press, 1998.

[11] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 1–20. IEEE Computer Society, 2003.

[12] P. Cignoni, 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. *ACM Trans. Graph.*, 23(3):796–803, 2004.

[13] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[14] R. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, 1986.

[15] R. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniq*, pages 95–102, New York, New York, USA, July 1987. ACM Press.

[16] R. L. Cook, J. Halstead, M. Planck, and D. Ryu. Stochastic simplification of aggregate detail. *ACM Trans. Graph.*, 26(99):79–8, July 2007.

[17] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009*

symposium on Interactive 3D graphics and games, page 15, New York, New York, USA, Feb. 2009. ACM Request Permissions.

[18] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *EGSR '08: Proceedings of the Nineteenth Eurographics conference on Rendering*, pages 1225–1233. Eurographics Association, June 2008.

[19] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88. IEEE Computer Society Press, 1997.

[20] G. Edwards. *Star Trek Beyond*. Number 148. Cinefex, 2016.

[21] C. Eisenacher, G. Nichols, and A. Selle. Sorted deferred shading for production path tracing. *Computer Graphics*, 2013.

[22] K. Fatahalian, E. Luong, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, page 59, 2009.

[23] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. DiagSplit: Parallel, Crack-free, Adaptive Tesselation for Micropolygon Rendering. *ACM Transactions on Graphics*, 28(5):1, Dec. 2009.

[24] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniq*, volume pp, pages 209–216, New York, New York, USA, Aug. 1997. ACM Press/Addison-Wesley Publishing Co.

[25] I. Georgiev, J. Krivánek, T. Davidovic, and P. Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):1, 2012.

[26] I. Georgiev, J. Křivánek, and P. Slusallek. Bidirectional light transport with vertex merging. *SIGGRAPH Asia 2011 Sketches*, page 27, 2011.

[27] E. Gobbetti and E. Bouvier. Time-critical multiresolution scene rendering. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 123–130. IEEE Computer Society Press, 1999.

[28] E. Gobbetti and F. Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885, 2005.

[29] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and . . .*, 1987.

[30] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniq*, pages 43–54. ACM Press, 1996.

[31] S. Green and D. Paddon. Exploiting coherence for multiprocessor ray tracing. In *Comput. Graph. Forum*, volume 22, pages 12–26, 2003.

[32] N. Greene. Environment Mapping and Other Applications of World Projections. *IEEE Comput. Graph. Appl.*, 6(11):21–29, Nov. 1986.

[33] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch. Efficient BVH construction via approximate agglomerative clustering. In *HPG '13: Proceedings of the 5th High-Performance Graphics Conference*, page 81, New York, New York, USA, July 2013. ACM Request Permissions.

[34] T. Hachisuka. Five common misconceptions about bias in light transport simulation, 2013.

[35] J. Hanika, A. Keller, and H. P. A. Lensch. Two-level ray tracing with reordering for highly complex scenes. In *GI '10: Proceedings of Graphics Interface 2010*, pages 145–152. Canadian Information Processing Society, May 2010.

[36] C. D. Hansen, T. Ize, and C. Brownlee. Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In *Proceedings of the 2011 Eurographics Symposium on Parallel Graphics and Visualization*, 2011.

[37] V. Havran. *Heuristic Ray Shooting Algorithms*. phdthesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov. 2000.

[38] P. S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *ACM SIGGRAPH Computer Graphics*, 24(4):145–154, Sept. 1990.

[39] H. Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniq*, pages 99–108. ACM Press, 1996.

[40] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, volume d, pages 19–26. ACM Press, 1993.

[41] http://avl.ncsa.illinois.edu/wp-content/uploads/2010/09/NCSA_F3_Tornado_H264_864.mov. Visualization of an F3 Tornado, Sept. 2010.

[42] https://developer.nvidia.com/optix. Optix Ray Tracing Engine.

[43] https://renderman.pixar.com/. RenderMan.

[44] https://www.solidangle.com/. Arnold Renderer.

[45] http://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi processors.html. Intel Xeon Phi.

[46] http://www.nvidia.com/object/nvidia-mental ray.html. mental ray.

[47] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graph.*, 22(3):935–942, 2003.

[48] H. W. Jensen. Global Illumination Using Photon Maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[49] J. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniq*, pages 143–150. ACM Press, 1986.

[50] D. Kirk, J. Arvo, D. Kirk, and J. Arvo. *Unbiased sampling techniques for image synthesis*, volume 25. ACM, July 1991.

[51] T. Kollig and A. Keller. Efficient Multidimensional Sampling. *Computer Graphics Forum*, 21(3):557–563, Sept. 2002.

[52] E. P. Lafortune and Y. D. Willems. A Theoretical Framework for Physically Based Rendering. *Computer Graphics Forum*, 13(2):97–107, May 1994.

[53] E. P. Lafortune and Y. D. Willems. The Ambient Term as a Variance Reducing Technique for Monte Carlo Ray Tracing. In *Photorealistic Rendering Techniques*, pages 168–176. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 1995.

[54] M. Larsen, J. S. Meredith, P. A. N. x00E1, til, and H. Childs. Ray tracing within a data parallel framework. In *2015 IEEE Pacific Visualization Symposium (PacificVis*, pages 279–286. IEEE, 2015.

[55] D. Laur, J. Fong, W. Wooten, and D. Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum*, 22(3):543–552, 2003.

[56] C. Lauterbach, S.-e. Yoon, M. Tang, and D. Manocha. ReduceM: interactive and memory efficient ray tracing of large models. In *EGSR '08: Proceedings of the Nineteenth Eurographics conference on Rendering*, pages 1313–1321. Eurographics Association, June 2008.

[57] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniq*, pages 31–42. ACM Press, 1996.

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

[59] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 93–102. ACM Press, 2003.

[60] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 279–286. IEEE Computer Society Press, 1998.

[61] N. Metropolis and S. M. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, Sept. 1949.

[62] B. Meyer. *Eiffel, The Language*. Prentice Hall, 1992.

[63] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[64] P. Navratil, H. Childs, D. Fussell, and C. Lin. Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing. *IEEE Trans. Visual. Comput. Graphics*, (99):1, 2013.

[65] P. A. Navratil. *Memory-efficient, scalable ray tracing*. PhD thesis, 2010.

[66] S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126. ACM Press, 1999.

[67] S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hansen, and P. Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):238–250, 1999.

[68] A. Patney and J. D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics*, 27(5):1, Dec. 2008.

[69] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, New York, USA, Aug. 1997. ACM Press/Addison-Wesley Publishing Co. Request Permissions.

[70] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[71] E. Reinhard and F. W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7):873–885, 1997.

[72] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, June 1993.

[73] R. Samanta, 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 graphic*, pages 75–84. IEEE Press, 2001.

[74] R. Samanta, T. Funkhouser, K. Li, and J. 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. ACM Press, 2000.

[75] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniq*, pages 65–70. ACM Press, 1992.

[76] R. E. Somers. FlexRender: A distributed rendering architecture for ray tracing huge scenes on commodity hardware. 2012.

[77] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. G. Parker. An Application of Scalable Massive Model Interaction using Shared-Memory Systems. *EGPGV*, 2006.

[78] P. A. Studios. *The RenderMan Interface Specification V3.2.1*, 2005.

[79] G. Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniq*, pages 55–64. ACM Press, 1992.

[80] E. Veach. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, 1997.

[81] E. Veach and L. Guibas. Bidirectional Estimators for Light Transport. In *Photorealistic Rendering Techniques*, pages 145–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[82] E. Veach and L. J. Guibas. *Optimally combining sampling techniques for Monte Carlo rendering*. ACM, New York, New York, USA, Sept. 1995.

[83] M. Vinkler, J. Bittner, and V. Havran. Massively Parallel Hierarchical Scene Processing with Applications in Rendering. *Computer Graphics*, 2013.

[84] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs -. *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 49–57, 2008.

[85] I. Wald, C. Benthin, and P. Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphic*, pages 1–11. IEEE Computer Society, 2003.

[86] I. Wald, A. Dietrich, and P. Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In Alexander Keller and Henrik Wann Jensen, editor, *Proceedings of the 15th Eurographics Workshop on Rendering Techniques, 2004*, pages 81–92. , 6 2004.

[87] I. Wald, G. P. Johnson, and J. Amstutz. OSPRay-A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Trans. Visual. Comput. Graphics*, 23(1):931–940, 2017.

[88] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka. CPU ray tracing large particle data with balanced P-k-d trees. *2015 IEEE Scientific Visualization Conference (SciVis)*, pages 57–64, 2015.

[89] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001*, pages 277–288. Springer, Vienna, Vienna, 2001.

[90] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001*, pages 277–288. Springer Vienna, Vienna, 2001.

[91] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–165, Sept. 2001.

[92] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree. *ACM Trans. Graph.*, 33(4):1–8, July 2014.

[93] M. Wand, M. Fischer, I. Peter, F. Heide, and W. Straï¿œer. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniq*, pages 361–370. ACM Press, 2001.

[94] D. Wexler, L. Gritz, E. Enderton, and J. Rice. GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPH-*

ICS conference on Graphics hardware, pages 7–14, New York, New York, USA, July 2005. ACM Press.

[95] D. Whitehouse. Visualization at the Australian National University. *SIGGRAPH Computer Graphics*, Nov.2000.

[96] T. Whitted. *An improved illumination model for shaded display*, volume 13. ACM, Aug. 1979.

[97] B. Wilson, K. Ma, and P. McCormick. A hardware-assisted hybrid rendering technique for interactive volume visualization. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 123–130. IEEE Press, 2002.

[98] S. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Trans. Graph.*, 24(3):886–893, 2005.

[99] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, and B. Guo. RenderAnts: interactive Reyes rendering on GPUs. *ACM Transactions on Graphics*, 28(5):155, 2009.