T.C. DOGUS UNIVERSITY
INSTITUTE OF SCIENCE AND TECHNOLOGY
COMPUTER AND INFORMATION SCIENCES MASTER'S DEGREE PROGRAM

REAL-TIME HYBRID PARALLEL RENDERING

Master of Science Thesis

M. Reha Cenani
200791003

Advisor:
Prof. Dr. Mithat Uysal

Istanbul, June 2009

Acknowledgements

I would like to express the deepest appreciation to my committee chair, Professor Mithat Uysal, who has the attitude and the substance of a patience: he continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank my committee members, Professor Selim Akyokus and Coskun Sonmez, whose work demonstrated to me that concern for informatics supported by an engagement in computer graphics.

Abstract

In computer graphics, rendering is described as the process of converting a description of a scene to an image. When the scene is complex and high quality images are required, the rendering process becomes computationally demanding. To provide the satisfactory performance, real-time computing techniques must be developed. Although parallelism has been extensively used in computer graphics for a long time, its initial use was primarily in specialized applications. Today, parallel computing is used in commodity personal computers, and various software-based rendering systems have been developed for general purpose real-time systems.

As the new GPUs released to the market, the available rendering performance increases constantly. Also more powerful multi-core CPUs that have enabled more flexible and faster software-based graphics, such as real-time ray tracing. Despite this tremendous hardware development progress in rendering power, there will always be some applications that require distributed configurations for rendering. In this thesis, I present a prototype solution consisting of a system that supports different rendering modules (e.g., rasterization, and ray tracing) and combine it with a distributed graphics processing.

This thesis provides a general introduction to the subject of real-time rendering, covering both hardware and software aspects. The main focus is on the underlying concepts and the issues which arise in the design of real-time rendering algorithms and systems. Different types of parallelism and how they can be applied in rendering applications are examined. Concepts from parallel computing, such as data decomposition, task granularity, scalability, and load balancing, are considered in relation to the rendering problem. Also concepts from computer graphics, such as coherence, culling, and level of detail which have a significant impact on the structure of parallel rendering algorithms are explored.

Özet

Bilgisayar grafikleri alanında *tarama (rendering)*, bir sahne tanımından görüntü oluşturulması süreci olarak tanımlanır. Sahne karışık ise ve yüksek kaliteli görüntüler isteniyorsa, *tarama* süreci uzun hesaplamalar gerektirebilir. Tatmin edici performansı elde etmek için, gerçek-zamanlı hesaplama yöntemleri geliştirilmelidir. Hernekadar bilgisayar grafiklerinde paralel işlem uzun süredir kapsamlı olarak kullanılsa da, temel kullanım alanı özel uygulamalar olmuştur. Bugün, paralel işlem kişisel bilgisayarlarda kullanılmaktadır ve genel amaçlı gerçek-zamanlı sistemler için çeşitli yazılım tabanlı *tarama* uygulamaları geliştirilmiştir.

Piyasaya yeni GPU'lar sürüldükçe, mevcut tarama preformansı sürekli artmaktadır. Aynı şekilde, daha güçlü çok çelirdekli işlemciler gerçek-zamanlı ışın izleme (ray tracing) gibi daha esnek ve daha hızlı yazılım tabanlı grafiklere imkan sağlıyorlar. Tarama gücünde artış sağlayan büyük donanım geliştirme ilerlemelerine rağmen, tarama için dağıtık konfigürayonlar gerektiren bazı uygulamalar herzaman olacaktır. Bu tezde, farklı tarama modülleri destekleyen bir sistem (rasterization, ışın izleme, v.s.) ve bunu dağıtık grafik işleme ile birleştiren bir prototip çözüm sunulmaktadır.

Bu tez, gerçek-zamanlı tarama konusuna hem yazılım hem de donanım tarafından genel bir giriş sunmaktadır. Ana odak, gerçek-zamanlı tarama algoritmaları ve sistemleri tasarlarken ortaya çıkan temel kavramlar ve konulardır. Farklı paralel işlem türlerini ve bunların tarama uygulamalarına nasıl uygulanabildiklerini incelenmiştir. Veri ayrıştırma (data decomposition), task granularity, ölçeklenebilirlik (scalability) ve yük dengeleme (load balancing) gibi paralel işlem kavramları, tarama problemi ile bağlantılı olarak değerlendirilmiştir. Eş fazlı olma (coherence), culling ve detay seviyesi (level of detail) gibi paralel tarama algoritmalarının yapısında önemli yere sahip bilgisayar grafiği kavramları da incelenmiştir.

Contents

List of Figures

Listings

Chapter 1

Related Work

There are several solutions which have been developed for the distribution of 3D graphics in a network. The WireGL (Humphreys et al., 2001) and Chromium (Humphreys et al., 2002) (Humphreys et al., 2008) graphics systems replace the OpenGL libraries of the host operating system, and send OpenGL commands to be rendered simultaneously on remote hosts across the network. While having the advantage of distributing applications transparently without modifications, the network bandwidth required for transmitting these OpenGL states and commands is very high (Eilemann, 2007).

In order to lower the required network bandwidth, the Equalizer framework (Eilemann and Pajarola, 2007) (Eilemann et al., 2008) the application is modified and higher-level commands are sent. While WireGL only supports a single sort-first architecture, Chromium provides arranging its stream filters to implement sort-first and sort-last alternatives. By allowing arbitrary distribution and providing a transparent definition of multi-display scenarios, Equalizer also extends these features.

All three mentioned frameworks are OpenGL based and cannot support other rendering techniques. The major drawback of these existing rendering frameworks is that they have fixed processing pipelines and do not allow to add special codecs or transport protocols which required for multi-view rendering. Since frameworks that allow distributed and parallel rendering like Equalizer and OpenRT (Dietrich et al., 2003) explicitly hide the distribution and they cannot support remote rendering or collaborative rendering.

On the other hand, the Network-Integrated Multimedia Middleware (NMM)(Lohse et al., 2008), provides separation between media processing and media transmission, and more transparent access to local and remote components. Media processing is specified by a flow graph where the nodes represent specific operations (e.g., rendering, or compressing images), and edges represent the transmission between nodes (e.g.,

pointer forwarding for local connections, or TCP for a network connection). Nodes can be connected to each other via their input streams and output streams; depending on the type of operation a node implements. Source nodes, for example, have no input streams, while sink nodes have no output streams. In the graph, media data flows from sources to sinks, being processed by each node in-between. Prerequisite for the successful connection of two nodes is a common format, which must be identical for the output stream of the preceding node and the input stream of the successive node to be connected.

The important aspect of NMM is, that nodes and edges are represented as first-class objects to the application, which allows to configure and control media processing and transmission transparently, for instance by choosing a certain transport protocol from the application layer (Repplinger et al., 2005). Even though this kind of distributed middleware solutions are especially designed for multimedia processing and do not explicitly consider rendering, their generic approach for distributed media processing is suitable for the requirement of flexibility the framework should provide. However, generic solutions like NMM have not yet been applied in other scenarios and might add significant overhead over specialized solutions.

Chapter 2

Stream Computing

Stream programming and streaming processors have recently become popular topics in computer architecture. The main motivation for stream processor development is that semiconductor technology is at a point where computation is cheap and bandwidth is expensive. Stream processors are designed to exploit this trend by exploiting both the parallelism and locality available in programs. The result is machines with higher performance per dollar (Khailany et al., 2000). To this end, stream processors provide hundreds of arithmetic processors to exploit parallelism, and a deep hierarchy of registers to exploit locality (Purcell, 2004).

The stream programming model constrains the way software is written such that locality and parallelism are explicit within a program. These constraints allow compilers to automatically optimize the code to take advantage of the underlying hardware. Of course, stream processors require sufficiently parallel computations to achieve this higher performance.

The stream programming model is based on kernels and streams. A kernel is a function that is going to be executed on over a large set of input records. A kernel loads an input record, performs computations on the values loaded, and then writes an output record. The more computation a kernel performs, the higher its arithmetic intensity or locality, and the better a stream processor will perform on it. Streams are the sets of input and output records operated on by kernels. Streams are what connect multiple kernels together.

The Imagine processor (Khailany et al., 2000) is a streaming processor made up of several arithmetic units connected to fast local registers and an on-chip memory called a stream register file. Imagine provides a bandwidth hierarchy with relatively small off-chip memory bandwidth, larger stream register file bandwidth, and very large local register file bandwidth. Programs written in the stream programming model can be

scheduled for the processor such that they mainly use internal bandwidth instead of external bandwidth. Imagine is programmed using StreamC and KernelC programming languages for streams and kernels that are a subset of C. These languages force programs to be written in a stream friendly manner, and are more general purpose than the StreaMIT language for the RAW processor. However, the underlying Imagine architecture is still exposed to the programmer when writing a stream program (Purcell, 2004).

Finally, there is the Merrimac streaming supercomputer (Houston, 2008). Mer- rimac is a large scale multi-chip streaming computer. Merrimac is programmed in a language called Brook (Buck, 2007). Brook is like StreamC and KernelC as it is an augmented subset of C designed for stream programming. However, one big difference between Brook and StreamC/KernelC is that Brook does not expose the details of the underlying architecture to the programmer. This means that programs written in Brook can be recompiled (instead of rewritten) for other stream machines.

Perhaps the most relevant target that Brook supports is GPUs. A BrookGPU program can compile to run on a standard Intel/AMD processor, or one of several different graphics processors (such as the NVIDIA GeForce FX or ATI Radeon GPUs) (Purcell, 2004). The ray tracing approach presented in this thesis was recently implemented in BrookGPU.

2.1 General Purpose Computing on Graphics Processing Units (GPGPU)

GPGPU stands for General-Purpose computation on GPUs. With the increasing programmability of commodity graphics processing units (GPUs), these chips are capable of performing more than the specific graphics computations for which they were designed. They are now capable coprocessors, and their high speed makes them useful for a variety of applications. The goal of this page is to catalog the current and historical use of GPUs for general-purpose computation.

General purpose computing on graphics processor units (GPGPU) becomes increasingly popular due to their remarkable computational power, memory access bandwidth and improved programmability. Current GPUs contain hundreds of compute cores and support thousands of light-weight threads, which hide memory latency and provide massive throughput for parallel computations. New programming models including CUDA from NVIDIA (Buck, 2007), Brook+ from AMD/ATI (Dimitrov et al., 2009),

and under-development OpenCL (Stone et al., 2009) facilitate programmers by allowing them to write GPU code in a familiar C/C++ environment, instead of forcing them to map general purpose computation to the graphics domain. In these programming models, the GPU is used as an accelerator to the CPU, from which memoryintensive and compute-intensive tasks are offloaded.

However, current GPUs do not provide hardware support for detecting soft or hard errors, which may occur in computation logic or memory storage. For instance, the off-chip storage of modern GPUs such as ATI Radeon HD series uses graphics double data rate (GDDR) type memories. As a result, any bit-flip in a memory cell may lead to silently corrupted results, i.e., erroneous results which are not detected. With soft-error rates predicted to grow exponentially (Harris, 2007) in future process generations and permanent failures/hard errors gaining importance, future GPUs are likely to be prone to hardware errors (Dimitrov et al., 2009). This has an adverse impact on GPGPU since many scientific, medical imaging and financial applications require strict correctness guarantees. Unfortunately, such reliability requirements are not likely to be answered in current or near future GPU generations. The reason is that even though GPGPU applications are gaining popularity, modern GPU design remains largely driven by the video games market, where totatly correct results are not strictly necessary.

2.2 Brook for GPU

Brook for GPU *(BrookGPU)* is a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor. It has a compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware (Buck et al., 2004).

BrookGPU is the Stanford University Graphics group's compiler and runtime implementation of the Brook stream programming language for using modern graphics hardware for non-graphical or general purpose computations. Use of Graphics Processing Unit (GPU) for doing non-graphical or general purpose calculations is also abbreviated as GPGPU (General Purpose Graphics Processing Unit). It can be used to program a graphics processing unit such as those found on ATI or NVIDIA graphics cards which are highly parallel in execution.

BrookGPU compiles programs written using Brook stream programming language, which is a variant of C. It can use OpenGL, DirectX or AMD Stream SDK for the computational backend and runs on Microsoft Windows, Linux and MacOS X. It can

also simulate a virtual graphics card by itself via a special CPU backend which is useful for debugging Brook kernels.

2.3 ATI Stream Computing

Using GPUs to perform computations holds a lot of potential for some applications because of the fundamental differences of GPU microarchitectures compared to CPUs. GPUs achieve much greater throughput (calculations per second) by executing many programs in parallel and restricting flow control (the ability of one program to execute instructions independently of another). Modern GPUs also have addressable on-die memory and extremely high performance multi-channel external memory.

ATI Stream technology is a set of advanced hardware and software technologies that enable AMD graphics processors (GPU), working in concert with the system's central processor (CPU), to accelerate many applications beyond just graphics (ATI, 2008). This enables better balanced platforms capable of running demanding computing tasks faster than ever.

Characteristics of GPU acceleration are enabling new applications on new architectures, solving parallel problems other than graphics that map well on GPU architecture, and making transition from fixed function to programmable pipelines.

The ATI Stream Computing Model includes a software stack and the ATI Stream processors. The ATI Stream Computing software stack provides end-users and developers with a complete, flexible suite of tools to leverage the processing power in ATI Stream processors. ATI software embraces open-systems, open-platform standards.

The software includes the following components (ATI, 2008):

1. Compilers - like the Brook+ compiler with extensions for ATI devices
2. Device Driver for stream processors - ATI Compute Abstraction Layer (CAL)
3. Performance Profiling Tools - Stream KernelAnalyzer
4. Performance Libraries - AMD Core Math Library (ACML) for optimized domain-specific algorithms

The latest generation of ATI Stream processors are programmed using the unified shader programming model. Programmable stream cores execute various user developed programs, called stream kernels (or simply: kernels) (Dimitrov et al., 2009).

These stream cores can execute non-graphics functions using a virtualized SIMD programming model operating on streams of data. In this programming model, known as stream computing, arrays of input data elements stored in memory are mapped onto a number of SIMD engines, which execute kernels to generate one or more outputs that are written back to output arrays in memory.

Each instance of a kernel running on a SIMD engine's thread processor is called a thread. A specified rectangular region of the output buffer to which threads are mapped is known as the domain of execution (Buck et al., 2004).

The stream processor schedules the array of threads onto a group of thread processors, until all threads have been processed. Subsequent kernels can then be executed, until the application completes.

2.4 NVIDIA CUDA

The advent of multi-core CPUs and muli-core GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop mainstream application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to multi-core GPUs with widely varying numbers of cores (Nickolls et al., 2008).

CUDA is a parallel computing architecture developed by NVIDIA . CUDA is the compute engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. C is used for CUDA, compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU (Ryoo et al., 2008). CUDA is architected to support various computational interfaces, including C and new open standards like OpenCL and DirectX Compute. Third party wrappers are also available for Python, Fortran and Java (Kirk, 2007).

The latest drivers all contain the necessary CUDA components. CUDA works with all NVIDIA GPUs from the G8X series onwards, including GeForce, Quadro and the Tesla line. NVIDIA states that programs developed for the GeForce 8 series will also work without modification on all future NVIDIA video cards, due to binary compatibility. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs effectively become open architectures like CPUs. Unlike CPUs however, GPUs have parallel multi-core architecture, each core capable of running thousands of threads simultaneously - if an application is suited to this kind of architecture, the GPU can offer

large performance benefits. In the computer gaming industry, in addition to graphics rendering, graphics cards are used in game physics calculations (physical effects like debris, smoke, fire, fluids), an example being PhysX and Bullet Physics. CUDA has also been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more (Buck, 2007).

According to conventional wisdom, parallel programming is difficult. Early experience with the CUDA scalable parallel programming model and C language, however, shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models (Buck, 2007). These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the new Tesla unified graphics and computing architecture run CUDA C programs and are widely available in laptops, PCs, workstations, and servers (Kirk, 2007). The CUDA model is also applicable to other shared-memory parallel processing architectures, including multi-core CPUs.

CUDA provides three key abstractions (a hierarchy of thread groups, shared memories, and barrier synchronization) that provide a clear parallel structure to conventional C code for one thread of the hierarchy (Nickolls et al., 2008).

Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions are used by the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel (Harris, 2007). The programming model may scale to large numbers of processor cores: a compiled CUDA program may execute on any number of processors, and only the run-time system needs to know the physical processor count.

CUDA provides both a low level API and a higher level API. NVIDIA has released versions of the CUDA API for Microsoft Windows, Linux and MacOS X.

Scattered reads (code can read to arbitrary addresses in memory), shared memory (CUDA exposes a fast shared memory region that can be shared amongst threads which can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups), faster downloads and read-backs to and from the GPU, and full support for integer and bitwise operations, including integer texture lookups are several advantages of CUDA over traditional general purpose computation on GPUs (GPGPU) using graphics APIs Che et al. (2008).

Some limitations of CUDA architecture can be summarizes as follows: CUDA uses a recursion-free, function-pointer-free subset of the C language, and some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments. Since CUDA does not support recursive functions, recursive code must be converted to loops. Also texture rendering is not supported (Harris, 2007). For double precision there are no deviations from the IEEE 754 standard. In single precision, Denormals and signaling NaNs are not supported; only two IEEE rounding modes are supported (chop and round-to-nearest even), and those are specified on a per-instruction basis rather than in a control word, and the precision of division/square root is slightly lower than single precision. In most cases the bus bandwidth and latency between the CPU and the GPU may be a bottleneck (Ryoo et al., 2008). Threads should be run in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task (e.g., traversing a ray tracing acceleration data structure). And finally, CUDA-enabled GPUs are only available from NVIDIA (GeForce 8 series and above, Quadro and Tesla) (Che et al., 2008).

2.5 OpenCL

OpenCL (Open Computing Language) is the first open standard for general-purpose parallel programming of heterogeneous systems. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance computing servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell Processor type architectures and other parallel processors such as DSPs (Dimitrov et al., 2009).

OpenCL supports a wide range of applications, from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient programming interface, OpenCL forms the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications (Stone et al., 2009).

OpenCL is being created by the Khronos Group with the participation of many industry leading companies and institutions.

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Facing technical challenges with higher clock speeds in a

fixed power envelope, Central Processing Units (CPUs) now improve performance by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into programmable parallel processors. As today's computer systems often include highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take full advantage of these heterogeneous processing platforms (Khr, 2009).

Creating applications for heterogeneous parallel processing platforms is challenging as traditional programming approaches for multi-core CPUs and GPUs are very different. CPU based parallel programming models are typically based on standards but usually assume a shared address space and do not encompass vector operations. General purpose GPU programming models address complex memory hierarchies and vector operations but are traditionally platform, vendor, or hardware specific. These limitations make it difficult for a developer to access the compute power of heterogeneous CPUs, GPUs and other types of processors from a single, multi-platform source code base. More than ever, there is a need to enable software developers to effectively take full advantage of heterogeneous processing platforms - from high performance compute servers, through desktop computer systems to handheld devices - that include a diverse mix of parallel CPUs, GPUs and other processors such as DSPs and the Cell Broadband Engine processor.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well- specified computation environment. The OpenCL standard supports both data and task-based parallel programming models, utilizes a subset of ISO C99 with extensions for parallelism, defines consistent numerical requirements based on IEEE 754, defines a configuration profile for handheld and embedded devices, and efficiently interoperates with OpenGL, OpenGL ES and other graphics APIs (Khr, 2009).

Chapter 3

Parallel Computing

Parallelism is familiar and frequently occurring concept in an everyday life (Lin and Snyder, 2009). An example for parallelism is building construction. Several workers simultaneously perform separate tasks such as plumbing, wiring, and furnace duct installation and so on. A call center, where many customer representatives serve customers at the same time, is an other example organization for parallelism. Also in manufacturing industry, most of the tasks are performed in parallel in the assembly line, in which many units of the product are under construction at once.

Although these tasks done in parallel, they differ in forms of parallelism. For example, the main difference between building construction and call center is that, calls are generally independent from each other and can be served in any order with little or no interaction among customer representatives. On the other hand, in building construction, some tasks can be done simultaneously -wiring and plumbing- while others must done in order -framing must precede wiring. The ordering restricts the amount of parallelism that can be done at once, limiting the speed at which a construction project can be done.

The ordering of the tasks also increases the degree of interaction among the workers. Assembly lines are different due to having strict ordering of tasks with the separate stages often being performed sequentially. In this case, parallelism arises from having many products in the assembly line at the same time.

In computer programs, the main purpose for executing program statements in parallel is to complete a task faster. But most of the today's existing programs are incapable of so much performance improvement through parallelism. Because these programs are written that statements would be executed sequentially, namely in order one at a time. Semantics of most programming languages enforce sequential execution. Still, there are some situations, such as the evaluation of the (a+b)*(c+d) expression (Lin

and Snyder, 2009). Assuming these are simple variables, sub-expressions (a+b) and (c+d) are independent of each other, so they can be calculated simultaneously. Such situations are examples of Instruction Level Parallelism (ILP).

3.1 Shared Memory Parallel Programming

The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI) (Basumallik and Eigenmann, 2005) (Krawezik, 2003).

OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library) (Nikolopoulos et al., 2000). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction (Mattson, 2003). These different associations may change the results of floating-point addition.

An OpenMP program begins as a single thread of execution, called the initial thread (Duran et al., 2005). The initial thread executes sequentially, as if enclosed in an

implicit task region, called the initial task region, that is defined by an implicit inactive parallel region surrounding the whole program.

When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the parallel construct (Smith and Bull, 2001). Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the parallel construct. Beyond the end of the parallel construct, only the master thread resumes execution, by resuming the task region that was suspended upon encountering the parallel construct (Mattson, 2003). Any number of parallel constructs can be specified in a single program.

Parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is created by a thread encountering a parallel construct inside a parallel region will consist only of the encountering thread (Jeun et al., 2008). However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread.

When any team encounters a worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. There is an optional barrier at the end of each worksharing construct. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a task construct, a new explicit task is generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later (Chapman, 2002). Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region (Duran et al., 2005). If the suspended task region is for an untied task, then any thread may resume its execution. In untied task regions, task scheduling points may occur at implementation defined points anywhere in the region. In tied task regions, task scheduling points may occur only in task, taskwait, explicit or implicit barrier constructs, and at the completion

point of the task. Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region (Smith and Bull, 2001). Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

Synchronization constructs and library routines are available in OpenMP to coordinate tasks and data access in parallel regions. In addition, library routines and environment variables are available to control or to query the runtime environment of OpenMP programs.

OpenMP makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output statements (or routines) using the provided synchronization constructs or library routines. For the case where each thread accesses a different file, no synchronization by the programmer is necessary (Müller, 2003).

3.2 Distributed Memory Parallel Programming

The evolution of parallel computer architectures has recently created new trends and challenges for both parallel application developers and end users. Systems comprised of tens of thousands of processors are available today; hundred-thousand processor systems are expected within the next few years. Monolithic high- performance computers are steadily being replaced by clusters of PCs and work- stations because of their more attractive price/performance ratio (Hale, 2004). However, such clusters provide a less integrated environment and therefore have different (and often inferior) I/O behavior than the previous architectures. Grid computing efforts yield a further increase in the number of processors available to parallel applications, as well as an increase in the physical distances between computational elements (Gabriel et al., 2004).

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today (Quinn, 2003).

The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly

run on shared memory computers (Karniadakis and Kirby, 2003). Designing programs around the MPI model (as opposed to explicit shared memory models) has advantages on NUMA architectures since MPI encourages memory locality.

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers of the reference model, with socket and TCP being used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e., an API) callable from Fortran, C, or C++ and from any language capable of interfacing with such routine libraries. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs) (Chapman, 2002).

MPI has Language Independent Specifications (LIS) for the function calls and language bindings. The first MPI standard specified ANSI C and Fortran-77 language bindings together with the LIS. The draft of this standard was presented at Supercomputing 1994 and finalized soon thereafter. About 128 functions constitute the MPI-1.2 standard as it is now defined.

There are two versions of the standard that are currently popular: version 1.2 (shortly called MPI-1), which emphasizes message passing and has a static runtime environment, and MPI-2.1 (MPI-2), which includes new features such as parallel I/O, dynamic process management and remote memory operations (Richard et al., 2006).MPI-2's LIS specifies over 500 functions and provides language bindings for ANSI C, ANSI Fortran (Fortran90), and ANSI C++. Interoperability of objects defined in MPI was also added to allow for easier mixed-language message passing programming (Bruck et al., 1995). A side effect of MPI-2 standardization (completed in 1996) was clarification of the MPI-1 standard, creating the MPI-1.2 level.

It is important to note that MPI-2 is mostly a superset of MPI-1, although some functions have been deprecated. Thus MPI-1.2 programs still work under MPI implementations compliant with the MPI-2 standard.

MPI is often compared with PVM, which is a popular distributed environment and message passing system developed in 1989, and which was one of the systems that motivated the need for standard parallel message passing systems (Spetka et al., 2008). Threaded shared memory programming models (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches, and can occasionally be seen used together in applications where this suits architecture, e.g. in servers with multiple large shared-memory nodes.

Chapter 4

Parallel Rendering Algorithms

4.1 Rasterisation

In many applications, particularly in the scientific visualization of large geometric data sets, we create images from data sets that might contain more than 500 million data points and generate more than 100 million polygons (Angel, 2008). This situation presents two immediate challenges. First, if we are to display this many polygons, how can we do so when even the best commodity displays contain only about two million pixels? Second, if we have multiple frames to display, either from new data or because of transformations of the original data set, we need to be able to render this large amount of geometry faster than can be achieved even with high-end systems.

One approach to both these problems is to use clusters of standard computers connected with a high-speed network (Humphreys et al., 2001) (Peng et al., 2006). Each computer might have a commodity graphics card. Note that such configurations are one aspect of a major revolution high-performance computing (Samanta et al., 2000). Formerly, supercomputers were composed of expensive fast processors that usually incorporated a high degree of parallelism in their designs (Crockett, 1997). These processors were custom designed and required special interfaces, peripheral systems, and environments that made them extremely expensive and thus affordable only by a few government laboratories and large corporations. Over the last few years, commodity processors have become extremely fast and inexpensive. The same technology has led to a variety of add-on graphics cards whose performance can be measured in millions of polygons per second and hundreds of millions of pixels per second. Computers assembled from such components can be connected standard networks that run at gigabit-persecond rates.

However, there are multiple ways we can distribute the work that must be done to render a scene among the processors. The simplest approach might be to execute the same application program on each processor but have each use a different window that corresponds to where the processor's display is located in the output array. For small applications, this approach might work; but for complex applications it is too slow because each processor is doing all the work and we are not taking advantage of having multiple processors. There are three other possibilities. In this taxonomy, the key difference is where in the rendering process we assign, or sort, primitives to the correct areas of the display.

Suppose that there is a large number of processors of two types: geometry processors and raster processors. This distinction corresponds to the two phases of the rendering pipeline. The geometry processors can handle front-end floating-point calculations, including transformations, clipping, and shading. The raster processors manipulate bits and handle operations such as scan conversion. Note that the present general-purpose processors and graphics processors can each do either of these tasks. Consequently, we can apply the following strategies to either the CPUs or the GPUs. Parallelism can be achieved among distinct nodes, within a processor chip through multiple cores, or within the GPU. The use of the sorting paradigm will help us organize the architectural possibilities.

Molnar et al. (1994) presented a classification scheme for distributed rendering. The authors subdivide techniques that distribute geometry according to screen-space tiles (sort-first), distribute geometry arbitrarily while doing a final z-compositing (sort-last), or distribute primitives arbitrarily, but do per-fragment processing in screen-space after sorting them during rasterization (sort-middle). This separation of techniques is based on rasterization, and where the rasterization pipeline distributes the workload across multiple processors.

4.1.1 Sort-Middle Rendering

Consider a group of geometry processors and raster processors are connected (Angel, 2008). Suppose that we have an application that generates a large number of geometric primitives. It can use multiple geometry processors in two obvious ways. It can run on a single processor and send different parts of the geometry generated by the application to different geometry processors. Alternatively, we can run the application on multiple processors each of which generates only part of the geometry. At this point, we need not worry about how the geometry gets to the geometry processors-as the best way is often application dependent-but on how to best employ the geometry processors that

are available.

Assume that we can send any primitive to any of the geometry processors, each of which acts independently. When we use multiple processors in parallel, a major concern is load balancing, that is, having each of the processors do about the same amount of work, so that none is sitting idle for a significant amount of time, thus wasting resources. One obvious approach would be to divide the object-coordinate space equally among the processors. Unfortunately, this approach often leads to poor load balancing because in many applications the geometry is not uniformly distributed in object space. An alternative approach is to distribute the geometry uniformly among the processors as objects are generated, independently of where the geometric objects are located. Thus, with n processors, we might send the first geo-metric entity to the first processor, the second to the second processor, the nth to the nth processor, the n + l-st to the first processor, and so on (Angel, 2008). Now consider the raster processors. We can assign each of these to a different region of the frame buffer or equivalently, assign each to a different region of the display. Thus, each raster processor renders a fixed part of screen space.

Now the problem is how to assign the outputs of the geometry processors to the raster processors. Note that each geometry processor can process objects that could go anywhere on the display. Thus, we must sort their outputs and assign primitives that merge from the geometry processors to the correct raster processors. Consequently, some sorting must be done before the raster stage. We refer to this architecture as sort-middle. This configuration was popular with high-end graphics workstations a few years ago, when special hardware was available for each task and there were fast internal buses to convey information through the sorting step. Recent GPUs contain multiple geometry processors and multiple fragment processors and so can be looked at as sort-middle processors. We tend to regard a particular commodity card with a single GPU as a combination of one geometry processor and one raster processor, thus aggregating the parallelism inside the GPU. Now the problem is how to use a group of commodity cards or GPUs. If we can use a GPU or a CPU as either a geometry processor or a raster processor and connect them with a standard network, the sorting step in sort-middle can be a bottleneck, and two other approaches have proved simpler.

4.1.2 Sort-Last Rendering

With sort-middle rendering, the number of geometry processors and the number of raster processors could be different. Now suppose that each geometry processor is connected to its own raster processor (Angel, 2008). This configuration would be what

we would have with a collection of standard PCs, each with its own graphics card, or on some of the most recent graphics cards that have multiple integrated vertex and fragment processors. Once again, let's not worry about how each processor gets the application data and instead focus on how this configuration process the geometry generated by the application.

Just as with sort-middle, we can load-balance the geometry processors by sending primitives to them in an order that ignores where on the display they might lie once they are rasterized. However, precisely because of this way of assigning geometry and lacking a sort in the middle, each raster processor must have a frame buffer that is the full size of the display. Because each geometry/raster pair contains a full pipeline, each pair produces a correct hidden-surface-removed image for part of the geometry.

Partial images can be combined with a compositing step (Angel, 2008). For the compositing calculations, we need not only the images in the color buffers of the geometry processors but also the depth information, because we must know for each pixel which of the raster processors contains the pixel corresponding to the closest point to the viewer 3. Fortunately, if we are using our standard OpenGL pipeline, the necessary information is in the z-buffer. For each pixel, we need only compare the depths in each of the z-buffers and write the color in the frame buffer of the processor with the closest depth. The difficulty is determining how to do this comparison efficiently when the information is stored on many processors.

Conceptually, the simplest approach, sometimes called binary-tree compositing, is to have pairs of processors combine their information. Consider that where there are four geometry/raster pipelines, numbered 0-3 (Angel, 2008). Processors 0 and 1 can combine their information to form a correct image for the geometry they have seen, while processors 2 and 3 do the same thing concurrently with their information. Let's assume that these new images are formed on processors 1 and 3. Thus, processors 0 and 2 have to send both their color buffers and their z buffers to their neighbors (processors 1 and 3, respectively). We then repeat the process between processors 1 and 3, with the final image being formed in the frame buffer of processor 3. Note that the required code is quite simple. The geometry/raster pairs each do an ordinary rendering. If implemented with OpenGL, the compositing step requires only the use of *glReadPixels* and some simple comparisons. However, in each successive step of the compositing process, only half the processors that were used in the previous step are still needed. In the end, the final image is prepared on a single processor.

There is another approach to the compositing step know as binary-swap compositing that avoids the idle processor problem. In this technique, each processor is responsible

for one part of the final image. Hence, for compositing to be correct, each processor must see all the data. If there are n processors involved in the compositing so they can be arranged in a round-robin fashion (Angel, 2008). The compositing takes n steps (rather than the log n steps required by tree compositing). On the first step, processor 0 sends portion 0 of its frame buffer to processor 1 and receives portion n from processor n. The other processors do a similar send and receive of the portion of the color and depth buffers of their neighbors. At this point, each processor can update one area of the display that will be correct for the data from a pair of processors. For processor 0 this will be region n. On the second round, processor 0 will receive from processor n the data from region n-1, which is correct for the data from processors n and n-1. Processor 0 will also send the data from region n, as will the other processors for part of their frame buffers. All the processors will now have a region that is correct for the data from three processors. Inductively, it should be clear that after n-1 steps, each processor has 1/n of the final image. Although more steps have taken, far less data has been transferred than with three compositing, and we have used all processors in each step.

4.1.3 Sort-First Rendering

One of the most appealing features of sort-last rendering is that we can pair geometric and raster processors and use standard computers with standard graphic cards (Correa et al., 2002). Suppose that we could decide first where each primitive lies on the final display. Then we could assign a separate portion of the display to each geometry/raster pair and avoid the necessity of a compositing network (Angel, 2008). Here we have included a processor at the front end to make the assignment as to which primitives go to which processors.

This front-end sort is the key to making this scheme work. In one sense, it might seem impossible, since we are implying that we know the solution-where primitives appear in the display-before we have solved the problem for which we need the geometric pipeline. But things are not hopeless. Many problems are structured so that we may know this information in advance. We also can get the information back from the pipeline using *glGetFloatv* to find the mapping from object coordinates to screen coordinates. In addition, we need not always be correct. A primitive can be sent to multiple geometry processors if it straddles more than one region of the display. Even if we send a primitive to the wrong processor, that processor may be able to send it on to the correct processor. Because each geometry processor performs a clipping step, we are assured that the resulting image will be correct.

Sort-first rendering does not address the load-balancing issue, because if there are regions of the screen with very few primitives, the corresponding processors may not be very heavily loaded (Angel, 2008). However, sort-first rendering has one important advantage over sort-last rendering: It is ideally suited for generating high-resolution displays. Suppose that we want to display our output at a resolution much greater than we get with typical CRT or LCD displays that have a resolution in the range of 1-3 million pixels. Such displays are needed when we wish to examine high-resolution data that might contain more than 100 million geometric primitives.

One approach to this problem is to build a tiled display or power wall consisting of an array of standard displays (or tiles). The tiles can be CRTs, LCD panels, or the output of projectors. From the rendering perspective, we want to render an image whose resolution is the array of the entire display, which can exceed 4000 x 4000 pixels. Generally, these displays are driven by a cluster of PCs with commodity graphics cards. Hence, the candidate rendering strategies are sort-first and sort-last.

However, sort-last rendering cannot work in this setting because each geometry / rasterizer processor must have a frame buffer the size of the final image, and for the compositing step, extremely large amounts of data must be exchanged between processors. Sort-first renderers do not have this problem. Each geometry / processors pair need only be responsible for a small part of the final image, typically an image the size of a standard frame buffer.

4.2 Ray Tracing

Ray tracing (Whitted, 1980) is an extension of the same technique developed in scanline rendering and ray casting. Similar to them, it handles complicated objects well, and the objects may be described mathematically. Unlike scanline and ray casting, ray tracing is almost always a Monte Carlo method (Křivánek, 2008) that is one based on averaging a number of randomly generated samples from a model.

In real-time rendering, using a local lighting model is the norm. That is, only the surface data at the visible point is needed to compute the lighting. This is a strength of the hardware pipeline, that primitives can be generated, processed, and then be discarded (Akenine-Möller and Haines, 2002). Transparency, reflections, and shadows are examples of global illumination algorithms, in that they use information from other objects than the one being illuminated. One way to think of the problem of illumination is the paths the photons take. In the local lighting model, photons travel from the light to a surface (ignoring intervening objects), then to the eye. With reflection, the

photon goes from the light to some object, bounces off and travels to a shiny object, then reflects off it and travels to the eye. There are many possible paths light can take. The rendering equation (Kajiya, 1986), expresses this idea of summing up all possible paths to find the radiance for a given direction. A higher level of realism can be obtained by accounting for more of these sets of paths. Global illumination research focuses on methods for efficiently computing the effect of various sets of paths.

Ray tracing is a rendering method in which rays are used to determine the visibility of various elements. The basic mechanism is very simple, and in fact, functional ray tracers have been written that fit on the back of a business card (Heckbert, 1994). In classical ray tracing (Whitted, 1980), rays are shot from the eye through the pixel grid into the scene. For each ray, the closest object is found. This intersection point then can be determined to be in light or shadow by shooting a ray from it to each light and finding if anything blocks or attenuates the light.

Other rays can be spawned from an intersection point. If the surface is shiny, a ray is generated in the reflection direction. This ray picks up the color from any object in this direction by recursively repeating the process of checking for shadows and reflecting rays, until a diffuse surface is hit or some maximum depth is reached. Environment mapping can be thought about as a very simplified version of ray traced reflections; the ray reflects and the light coming from the reflection direction is retrieved. The difference is that, in ray tracing, nearby objects can be intersected by the reflection rays. Note that if these nearby objects are all missed, an environment map can be used to represent the rest of the environment. Rays can also be generated in the direction of refraction for transparent solid objects, again recursively evaluated. When the maximum number of reflections and refractions is reached, a ray tree has been built up. This tree is then evaluated from the deepest reflection and refraction rays on back to the root, so yielding a color for the sample. Ray tracing provides sharp reflection, refraction, and shadow effects. Because each sample on the image plane is essentially independent, any point sampling and filtering scheme desired can be used for antialiasing. Another advantage of ray tracing is that true curved surfaces and other untessellated objects can be intersected directly by rays (Akenine-Möller and Haines, 2002).

The main problem with ray tracing is simply speed. One reason graphics hardware (GPU) is so fast is that it uses coherence efficiently. Each triangle is sent through the pipeline and covers some number of pixels, and all these related computations can be shared when rendering a single triangle. Other sharing occurs at higher levels, such as when a vertex is used to form more than one triangle or a shader configuration is used for rendering more than one primitive (Hanrahan, 1989). In ray tracing, the ray

performs a search to find the closest object. Some caching and sharing of results can be done, but each ray potentially can hit a different object. Much research has been done on making the process of tracing rays as efficient as possible (Smits, 1998) (Arvo and Kirk, 1989) (Glassner, 1989) (Woop et al., 2006) (Parker et al., 2005).

There are a number of ways ray tracing can be used in a real-time context. One is for precomputing high-quality synthetic images to use for making environment maps, impostors, skyboxes, or other image-based parts of the scene. Ray tracing can also be used to generate and store other information, such as depths, normals, or transparency at each pixel of some distant object. By directly accessing this stored data in a pixel shader, it becomes possible to rapidly rerender the object when, say, lighting conditions change. Another use is that, during rendering itself, reflection or shadow rays can be generated for small parts of the scene (Wald et al., 2005). The resulting samples are blended into the Z-buffer image, and the process can be relatively inexpensive, though CPU intensive. Another way to integrate ray tracing is to fold it into the per vertex lighting computations. Tracing rays from only the vertices can significantly reduce the amount of computation, but suffers from typical Gouraud-shading artifacts. Sharp reflections will usually not be captured, though this could be considered an advantage, as the reflections will look blurry. Lindholm et al. (2001) give an example of a vertex shader performing ray tracing to reflect a nearby sphere in a curved surface.

In classical ray tracing (Whitted, 1980), rays are spawned in the most significant directions: toward the lights and for mirror reflections and refractions. Monte Carlo ray tracing takes the approach of having a single ray reflect or refract through the scene, with each surface's BRDF influencing the direction that the ray next travels. By shooting many rays for each pixel, a fuller sampling of each surface's incoming irradiance is formed. This technique is very expensive, with thousands or millions or more rays needed per pixel to converge to a precise solution. Given enough time, it fully solves Kajiya's (Kajiya, 1986) rendering equation. For more on the theory and practice of classical and Monte Carlo ray tracing (Křivánek, 2008), Shirley's book (Shirley and Morley, 2003) can be inspected.

Shooting rays through the entire scene and distributing them with respect to the BRDF in real time is well beyond even the fastest machines (Akenine-Möller and Haines, 2002). However, the idea of sampling the hemisphere with ray casting is a feasible preprocess. The idea is that vertices in cracks and crevices will tend to get less illumination. To approximate this effect of self-shadowing, shoot a set of rays outwards in a hemisphere from each vertex in a model. Weight the distribution by the cosine of the angle to the normal. Sum up the proportion of rays that do not intersect the model itself. This value is stored for each vertex and used during rendering to dim its illumination level.

The effect is to make objects have more definition and look more realistic (Zhukov et al., 1998). Another way to use hemisphere sampling is to precompute soft shadow textures for characters. An old technique is to put a fuzzy gray circle texture beneath a character. By using hemisphere ray casting at each texel's location and checking for intersection with the character, a more realistic all-purpose drop shadow texture can be created.

Interactive ray tracing has been possible on a limited basis for some time. For example, the demo scene (Scheib, 2001) has made real-time programs for years that have used ray tracing for some or all of the rendering. Because each ray is, by its nature, evaluated independently from the rest, ray tracing is "embarrassingly parallel" with more processors being thrown at the problem usually giving a nearly linear speedup. Ray tracing also has another interesting feature, that the time for finding the closest intersection for a ray is typically order $O(\log\ n)$ for n objects, when an efficiency structure is used. For example, bounding volume hierarchies typically have $O(\log\ n)$ search behavior. This compares well with the typical $O(n)$ performance of the basic Z-buffer, in which all polygons have to be sent down the pipeline. Techniques can be used to speed up the Z-buffer to give it a more $O(\log\ n)$ response, but with ray tracing, this performance comes with minimal user intervention.

One advantage of the Z-buffer is its use of coherence (Davis and Reinhard, 2002), sharing results to generate a set of fragments from a single triangle. As scene complexity rises, this factor loses importance. As Wald et al. (2001b) have shown, by carefully paying attention to the cache and other architectural features of the CPU, as well as taking advantage of CPU SIMD instructions, interactive and near-interactive rates can be achieved. While the results are impressive, Z-buffer graphics accelerators will be the mainstay for most real-time rendering work. Ray tracing also has its own limitations to work around. For example, the efficiency structure that reduces the number of ray/object tests needed is critical to performance. When an object moves, this structure needs to be updated rapidly to keep efficiency at a maximum, a task that can be difficult to do well. There are other issues as well, such as the cache-incoherent nature of reflection rays (Wald et al., 2001a). A summary of the state of the art in interactive ray tracing can be seen in See Wald and Slusallek's report (Wald and Slusallek, 2001). Since then, Purcell (2002), Purcell et al. (2002), and Purcell et al. (2005) have described how to use a graphics accelerator to accelerate ray tracing directly.

The object of parallel processing is to find a number of preferably independent tasks and to execute these tasks on different processors.

4.3 Radiosity

Radiosity, also known as *global illumination*, is a method which attempts to simulate the way in which directly illuminated surfaces act as indirect light sources that illuminate other surfaces. This produces more realistic shading and seems to better capture the 'ambience' of an indoor scene.

In advanced radiosity simulation (Hadwiger et al., 2008), recursive, finite-element algorithms *bounce* light back and forth between surfaces in the model, until some recursion limit is reached. The coloring of one surface in this way influences the coloring of a neighboring surface, and vice versa. The resulting values of illumination throughout the model (sometimes including for empty spaces) are stored and used as additional inputs when performing calculations in a ray casting or ray tracing model (Wald et al., 2003).

The optical basis of the simulation is that some diffused light from a given point on a given surface is reflected in a large spectrum of directions and illuminates the area around it. The simulation technique may vary in complexity. Many renderings have a very rough estimate of radiosity, simply illuminating an entire scene very slightly with a factor known as ambiance. However, when advanced radiosity estimation is coupled with a high quality ray tracing algorithm, images may exhibit convincing realism, particularly for indoor scenes (Reinhard, 2002).

If there is little rearrangement of radiosity objects in the scene, the same radiosity data may be reused for a number of frames, making radiosity an effective way to improve on the flatness of ray casting, without seriously impacting the overall rendering time-per-frame. Because of this, radiosity has become the leading real-time rendering method.

Due to the iterative/recursive nature of the technique, complex objects are particularly slow to emulate (Slusallek et al., 2005). Prior to the standardization of rapid radiosity calculation, some graphic artists used a technique referred to loosely as false radiosity by darkening areas of texture maps corresponding to corners, joints and recesses, and applying them via self-illumination or diffuse mapping for scanline rendering. Even now, advanced radiosity calculations may be reserved for calculating the ambiance of the room, from the light reflecting off walls, floor and ceiling, without examining the contribution that complex objects make to the radiosity or complex objects may be replaced in the radiosity calculation with simpler objects of similar size and texture (Křivánek, 2008).

The fixed-function pipeline allows point lights to have a constant illumination or fall off with distance or distance-squared (Akenine-Möller and Haines, 2002). Often local

light sources are not set to drop off with the square of the distance, as they would in the real world. One reason is that such lights are difficult to control. Such lights appear to drop off too quickly due to a lack of gamma correction. Another factor is that tone reproduction is difficult to perform in real time (Durand and Dorsey, 2000) (Lischinski et al., 2006). But an important reason that distance-squared lights look unrealistic is because most real-time systems do not properly account for indirect illumination. In reality, a significant amount of light in a scene comes from light reflecting from surfaces. At night, go into a room and close the blinds and drapes and turn a light on. The reason you can see anything not in line of sight of the light source is because the light bounces off objects in the room. This additional light is so significant that using distance-squared point lights without accounting for indirect illumination often means making errors in the opposite direction, with the overall lighting falling off too rapidly. Qualitatively, direct lighting from point sources gives a harsh look that indirect illumination will soften.

There are many different global illumination techniques for determining the amount of light reaching a surface and then travelling to the eye. Jensen's book (Jensen, 2001) begins with a good technical overview of the subject. While many of these techniques are not currently interactive, research shows a trend towards using the power of graphics accelerators to make them so. The hemicube method of creating form factors for radiosity algorithms naturally lends itself to hardware acceleration (Cohen et al., 1993) (Sillion and Puech, 1994). Stürzlinger and Bastos (1997) render photon-mapped surfaces by using textured sprites as splats. Stamminger et al. (2000) use projective textures to blend ray tracing samples to hardware accelerated renderings. Another example is Hakura and Snyder (2001), where they use a combination of minimal ray tracing for local objects and layered environment maps to produce reflections and refractions that closely match fully ray traced solutions. Atmospheric effects such as clouds are another area of research. For example, Harris and Lastra (2001) use an anisotropic multiple scattering approximation to generate cloud images, which are then displayed using impostors.

One technique that has found use within the real-time arena is radiosity, specifically meshed radiosity. There have been whole books written on this algorithm (Cohen et al., 1993) (Sillion and Puech, 1994) (Ashdown, 1995) (Dutre et al., 2006), but the basic idea is relatively simple. Light bounces around an environment; you turn a light on and the illumination quickly reaches a stable state. In this stable state, each surface can be considered as a light source in its own right. When light hits a surface, it can be absorbed, diffusely reflected, or reflected in some other fashion (specularly, anisotropically, etc). Basic radiosity algorithms first make the simplifying assumption

that all indirect light is from diffuse surfaces. This assumption fails for places with polished marble floors or large mirrors on the walls, but for most architectural settings, this is a reasonable approximation. The BRDF of a diffuse surface is a simple, uniform hemisphere, so the surface's radiance from any direction is proportional purely to the amount of incoming irradiance multiplied by the reflectance of the surface.

To begin the process, each surface is represented by a number of patches (i.e., polygons)(Akenine-Möller and Haines, 2002). The patches do not have to match one-for-one with the underlying polygons of the rendered surface. There can be fewer patches, as for a mildly curving spline surface, or more patches can be generated during processing, in order to capture features such as shadow edges.

To create a radiosity solution, the basic idea is to form a matrix of form factors among all the patches in a scene. Given some point or area on the surface (such as at a vertex or the patch itself), imagine a hemisphere above it (Akenine-Möller and Haines, 2002). Similar to environment mapping, the entire scene can be projected onto this hemisphere. The form factor is a purely geometric value denoting the proportion of how much light travels directly from one patch to the surface. A significant part of the radiosity algorithm is accurately determining the form factors between the receiving patch and each other patch in the scene. The area, distance, and orientations of both patches affect this value. Cohen and Wallace (Cohen et al., 1993), and Sillion and Puech (Sillion and Puech, 1994), cover a wide range of such formulae. As the viewed patch nears the horizon of the receiving patch's hemisphere its effect lessens, just the same as how a light's effect on a diffuse surface lessens under the same circumstances.

Another important factor is visibility between patches. If something else partially or fully blocks the tested patch from being seen by the receiver, the form factor is correspondingly reduced. Thinking back on the hemisphere, there is essentially only one surface visible in any given direction. Calculating the form factor of a patch for a receiving point is equivalent to finding the area of the patch visible on the hemisphere and then projecting the hemisphere onto the ground plane. The proportion of the circle on the ground plane beneath the hemisphere that the patch covers is the patch's form factor. Called the Nusselt analog (Ward, 2007), this projection effectively folds in the cosine term that affects the importance of the viewed patch to the receiving point.

Given the geometric relations among the various patches, some patches are designated as being emitters (i.e., lights). Energy travels through the system, reaching equilibrium. One way used to compute this equilibrium in fact, the first way discovered, using heat transfer methods) is to form a square matrix, with each row consisting of the form factors for a given patch times that patch's reflectivity.

It is worth mentioning that the radiosity solutions for independent lights can be solved for individually and combined later. For example, given a few light sources in a scene, a set of light maps capturing the radiosity solution for each light could be created. As lights are turned off and on, the various sets of light maps can be added together as needed. A significant amount of research has focused on simplifying the solution of this matrix. For example, in progressive radiosity, the idea is to shoot the light out from the light sources and collect it at each patch. The patch receiving the most light is then treated like an emitter, bouncing light back into the environment. The next brightest patch then shoots its light out, possibly starting to refill the first shot patch with new energy to shoot. This process continues until some level of diminishing returns is reached. This algorithm takes no less time to fully converge on a solution to the matrix, but has a number of advantages. Form factors are created for only one column of the radiosity equation for each patch shoot, an $O(n)$ process (Akenine-Möller and Haines, 2002). After any given shoot step, a preliminary radiosity solution can be output. This means a usable solution can be rapidly generated and displayed in seconds or less, with the ability to refine the solution over time with unused cycles.

A recent improvement on the progressive algorithm is eigenvector radiosity, proposed by Ashdown (2001), which is three orders of magnitude faster for moderately complex environments. The approach has reasonable memory requirements, and is good for simulating static environments under changing illumination conditions. Ashdown reports computing converged radiosity results in under a tenth of a second for 1000 element environments.

Typically, however, the radiosity process itself is usually performed offline. The resulting computed illumination is applied to the surfaces by either storing the amount of light reaching each vertex and interpolating, or by storing a light map for the surface.

Radiosity is an approach that gives a visual richness to an environment, while also precomputing all diffuse components, thereby allowing faster redisplay than computing these on the fly. There are a few drawbacks to the technique beyond the time cost of the algorithm itself and any visual artifacts caused by it. First, the solution is fixed in place for a given set of lights and object positions. Turning on and off a light is only a state change, so could be captured by storing two solution sets. However, as with any global illumination algorithm, moving an object will invalidate the rendering for other objects. Movement modifies some of the form factors, e.g., the object's angle and position to the lighting will change, the shadows cast by the object will change, and the light reflected by the object itself will change.

Specular highlights have to be handled carefully in a radiosity environment, as the

visibility of each light affects whether a highlight exists or not. Ignoring this detail means that objects in shadow will still shine. Also, if light reflects from a patch onto an object, this source of light will not cause a specular highlight to appear. Walter et al. address this problem by creating a more elaborate radiosity solution that also stores the directional specular component, fit point lights to best represent these highlights, then use these lights in conjunction with a displayed diffuse solution.

Radiosity theory has been used for terrain rendering in games (Akenine-Möller and Haines, 2002). Hoffman and Mitchell determine how much each vertex is directly illuminated by the sun by storing critical angles. They also compute the lighting effect of the sky and surrounding terrain by using horizon mapping. In horizon mapping, for each point on the terrain, the altitude angle of the horizon is determined for some set of azimuth directions (e.g., eight: north, northeast, east, southeast, etc.). By using this information and making some simplifying assumptions, they are able to get a reasonable approximation of the effect of surrounding terrain and the sky. Specifically, they use Stewart and Langer's result that for a scene under diffuse lighting conditions, the points near a given point have the same radiance. The result is the creation of a light map that is multiplied by the sky's color during run time.

Chapter 5

Acceleration Algorithms & Data Structures

5.1 Spatial Data Structures

Spatial data structures are used to organize geometry in some n-dimensional space (Chalmers et al., 2002). These data structures can be used to accelerate queries about if geometric entities overlap. Such queries are used in a wide variety of operations such as collision detection, culling algorithms, during intersection testing and ray tracing (Akenine-Möller and Haines, 2002).

Spatial data structures are usually hierarchically organized. This means that the top-most level encloses the level below it, which encloses the level below that level, and so on. Because of this, the structure is nested and has a recursive nature. The main reason for using such a hierarchy is that different types of queries get significantly faster. This improvement is usally from O(n) to O(log n). On the other hand, the contruction of most spatial data structures is an expensive task. So, it is done as a preprocess but incremental updates are done in real-time (Samet, 2005).

Bounding Volume Hierarchies (BVHs), variants of BSP trees, and octrees are some different types of spatial data structures. Octrees and BSP trees are based on space subdivision. This means that entire space of the scene is subdivided and encoded in the data structure. For example, the union of the space of all the leaf nodes is equal to the entire space of the scene. Both variants of BSP trees are irregular, namely the space can be subdivided more arbitrarly. On the other side, the octree is a regular structure so that the space is uniformly splitted. But this uniformity can often be a reason for efficiency (Samet, 2007).

A bounding volume hierarchy is not a space subdivision structure, but it encloses the regions of the space surrounding geometrical objects. Because of this, the BVH need not enclose all space (Samet, 2008). In addition to improving efficiency of queries, BVHs are also commonly used to describe model relationships and for control of hierarchical animation.

5.1.1 Bounding Volume Hierarchies (BVHs)

A bounding-volume hierarchy is a tree structure on a set of geometric data objects (Haverkort, 2004). Each object is stored in a leaf of the tree. Each internal node stores for each of its children v an additional geometric object *V(v)*, that encloses all data objects that are stored in descendants of v. In other words, *V(v)* is a bounding volume for the descendants of v.

Algorithm 5.1 $Intersected(Q, v)$

```
1: for all child x of v do
2:   if V(x) intersects Q then
3:     if x is a leaf then
4:       if M intersects Q then
5:         report M
6:       else
7:         Intersected(Q, v)
```

Bounding-volume hierarchies can be used to do many types of queries on the set of data objects. For example, the *Algorithm 5.1* finds all objects that intersect a query range Q and are stored in descendants of node v. To find all data input objects that intersect Q, start the algorithm with the root of the hierarchy as v. The query will then descend into the tree, visiting exactly those nodes whose bounding volumes intersect Q. The bounding-volume hierarchy can also be used for other types of queries, such as nearest-neighbour queries.

5.1.2 Binary Search Partitioning (BSP) Trees

When the original design of the algorithm for Binary Space Partitioning (BSP)-trees was formulated the idea was to use it to sort the polygons in the world. The reason for this was there did not exist hardware accelerated Z-buffers, and software Z-buffering was too slow. Today that area of usage is obsolete, since hardware accelerated Z-buffers exist. Instead the usage is to optimize a wide variety of areas, such as radiosity calculations, drawing of the world, collision detection and networking.

Binary Space Partioning (BSP)-trees were first described in 1969 by Schumacker et al. (1969), it was hardly meant to be an algorithm used to develop entertainment products, but BSP-trees have been used in the gaming industry to improve performance and make it possible to use more details in the maps.

A Binary Space Partitioning-tree is a structure that, as the name suggests, subdivides the space into smaller sets. These days, given hardware accelerated Z-buffers; the benefit of this is that one has a smaller amount of data to consider given a location in space. The main reason BSP-trees were being used was that they sorted the polygons in the scene so that you always drew back-to-front, meaning that the polygon with the lowest Z-value was drawn last. There are other ways to sort the polygons so that the closest polygon is drawn last, for example the Painter's algorithm (Hanrahan, 1989), but few are as cheap as BSP-trees, because the sorting of the polygons is done during the pre-processing of the map and not under run-time. The algorithm for generating a BSP-tree is actually an extension of Painter's algorithm. Just as the original design of the BSP algorithm, the Painter's algorithm works by drawing the polygons in the scene in back-to-front order. However, there are some major drawbacks with Painter's algorithm:

1. Polygons will not be drawn correctly if they pass through any other polygon
2. It is difficult and computationally expensive to calculate the order that the polygons should be drawn in for each frame
3. The algorithm cannot handle cases of cyclic overlap

The original idea for the creation of a BSP-tree is that you take a set of polygons that is part of a scene and divide them into smaller sets, where each subset is a convex set of polygons. That is, each polygon in this subset is in front of every other polygon in the same set. Polygon 1 is in front of polygon 2 if each vertex in polygon 1 is on the positive side of the plane polygon 2 defines or in that plane that. A cube made of inward facing polygons is a convex set, whilst a cube made of outwards facing polygons is not.

5.2 Culling

To cull means *to remove from a flock* and in the context of computer graphics, this is exactly what culling techniques do (Akenine-Möller and Haines, 2002). The flock is the whole scene that we want to render, and the removal is limited to those portions

of the scene that are not considered to contribute to the final image. The rest of the scene is sent through the rendering pipeline. Thus, the term visibility culling is also often used in the context of rendering. However, culling can also be done for other parts of a program. Examples include collision detection (by doing less accurate computations for invisible objects), physics computations, and AI. Examples of such techniques are backface culling, view frustum culling, and occlusion culling. Backface culling eliminates polygons facing away from the viewer. This is a simple technique that operates on only a single polygon at a time. View frustum culling eliminates groups of polygons outside the view frustum. As such, it is a little more complex. Occlusion culling eliminates objects hidden by groups of other objects. It is the most complex culling technique, as it requires an object or group of objects to gather and use information about other objects' locations.

The actual culling can theoretically take place at any stage of the rendering pipeline, and for some occlusion culling algorithms, it can even be precomputed. For culling algorithms that are implemented in hardware, we can sometimes only enable/disable or set some parameters for the culling function. For full control, the programmer can implement the algorithm in the application stage on the CPU. Assuming the bottleneck is not on the CPU, the fastest polygon to render is the one never sent down the accelerator's pipeline. Culling is often achieved by using geometric calculations but is in no way limited to these. For example, an algorithm may also use the contents of the frame buffer (Assarsson et al., 2003).

The ideal culling algorithm would send only the Exact Visible Set (EVS) of primitives through the pipeline. In this book, the EVS is defined as all primitives that are partially or fully visible. One such data structure, that allows for ideal culling, is the aspect graph, from which the EVS can be extracted given any point of view. Creating such data structures is possible in theory, but not really in practice, since worst-time complexity can be as bad as $O(n^9)$ (Cohen-Or et al., 2003). Instead, practical algorithms attempt to find a set, called the Potentially Visible Set (PVS), that is a prediction of the EVS. If the PVS fully includes the EVS, so that only invisible geometry is discarded, the PVS is said to be conservative. A PVS may also be approximate, in which the EVS is not fully included. This type of PVS may therefore generate incorrect images. The goal is to make these errors as small as possible. Since a conservative PVS always generates correct images, it is considered more useful. By overestimating or approximating the EVS, the idea is that the PVS can be computed much faster. The difficulty lies in how these estimations should be done to gain overall performance. For example, an algorithm may treat geometry at different granularities, i.e., polygons, whole objects, or groups of objects. When a PVS has been found, it is

rendered using the Z-buffer, which resolves the final visibility (Cohen-Or et al., 2003).

5.2.1 View Frustum Culling

As a basic principle in culling, only primitives that are totally or partially inside the view frustum need to be rendered. One way to speed up the rendering process is to compare the bounding volume (BV) of each object to the view frustum. If the BV is outside the frustum, then the geometry it encloses can be omitted from rendering. Since these computations are done within the CPU, this means that the geometry inside the BV does not need to go through the geometry and the rasterizer stages in the pipeline. If instead the BV is inside or intersecting the frustum, then the contents of that BV may be visible and must be sent through the rendering pipeline.

By using a spatial data structure, this kind of culling can be applied hierarchically. For a bounding volume hierarchy (BVH), a preorder traversal from the root does the job. Each node with a BV is tested against the frustum. If the BV of any type of node is outside the frustum, then that node is not processed further. The tree is pruned, since the BV's subtree is outside the view. If the BV intersects the frustum, then the traversal continues and its children are tested. When a leaf node is found to intersect, its contents (i.e., its geometry) is sent through the pipeline (Akenine-Möller and Haines, 2002). The primitives of the leaf are not guaranteed to be inside the view frustum. Clipping takes care of ensuring that only primitives inside the view frustum are being rendered. If the BV is fully inside the frustum, its contents must all be inside the frustum. Traversal continues, but no further frustum testing is needed for the rest of such a subtree.

View frustum culling operates in the application stage, which means that both the geometry and the rasterizer stages can benefit enormously. For large scenes or certain camera views, only a fraction of the scene might be visible, and it is only this fraction that needs to be sent through the rendering pipeline. In such cases a large gain in speed can be expected. View frustum culling techniques exploit the spatial coherence in a scene, since objects that are located near each other can be enclosed in a BV, and nearby BVs may be clustered hierarchically.

Other spatial data structures than the BVH can also be used for view frustum culling. This includes octrees and Binary Space Partitioning (BSP) trees. These methods are usually not flexible enough when it comes to rendering dynamic scenes. That is, it takes too long to update the corresponding data structures when an object stored in the structure moves. An exception for this situation is loose octrees. But for static scenes, these methods can perform better than BVHs.

Polygon-aligned BSP trees are simple to use for view frustum culling. If the box containing the scene is visible, then the root node's splitting plane is tested. If the plane intersects the frustum (i.e., if two corners on the frustum are found to be on opposite sides of the plane), then both branches of the BSP tree are traversed. If instead, the view frustum is fully on one side of the plane, then whatever is on the other side of the plane is culled. Axis-aligned BSP trees and octrees are also simple to use. Traverse the tree from the root, and test each box in the tree during traversal. If a box is outside the frustum, traversal for that branch is terminated.

For view frustum culling, there is a simple technique for exploiting frame-to-frame coherency (Akenine-Möller and Haines, 2002). If a BV is found to be outside a certain plane of the frustum in one frame, then (assuming that the viewer does not move too quickly) it will probably be outside that plane in the next frame too. So if a BV was outside a certain plane, then an index to this plane is stored (cached) with the BV. In the next frame in which this BV is encountered during traversal, the cached plane is tested first, and on average a speed-up can be expected.

If the viewer is constrained to only translation or rotation around one axis at a time from frame to frame, then this can also be exploited for faster frustum culling. When a BV is found to be outside a plane of the frustum, then the distance from that plane to the BV is stored with the BV. If the viewer only translates, then the distance to the BV can be updated quickly by knowing how much the viewer has translated. This can provide a generous speed-up in comparison to a naive view frustum culler.

5.2.2 Backface Culling

All backfacing polygons that are part of an opaque object can be culled away from further processing. A consistently oriented polygon is backfacing if the projected polygon is oriented in, say, a counterclockwise fashion in screen space. This test can be implemented by computing the normal of the projected polygon in two-dimensional screen space: $\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0)x(\mathbf{v}_2 - \mathbf{v}_0)$. This normal will either be (0,0,a) or (0,0,-a), where $a > 0$ (Akenine-Möller and Haines, 2002). If the negative z-axis is pointing into the screen, the first result indicates a frontfacing polygon. This test can also be formulated as a computation of the signed area of the polygon. Either culling test can be implemented immediately after the screen-mapping procedure has taken place (in the geometry stage). Backface culling decreases the load on the rasterizer since we do not have to scan convert the backfacing polygons. But the load on the geometry stage might increase because the backface computations are done there.

Another way to determine whether a polygon is backfacing is to create a vector from an arbitrary point on the plane in which the polygon lies (one of the vertices is the simplest choice) to the viewer's position. Compute the dot product of this vector and the polygon's normal. A negative dot product means that the angle between the two vectors is greater than $\pi/2$ radians, so the polygon is not facing the viewer. This test is equivalent to computing the signed distance from the viewer's position to the plane of the polygon. If the sign is positive, the polygon is frontfacing. Note that the distance is obtained only if the normal is normalized, but this is unimportant here, as only the sign is of interest (Akenine-Möller and Haines, 2002). This test can be performed after the model transform (into world space) or after the model and view transforms (into eye space), which is a bit earlier in the geometry stage than with the screen-space method.

Blinn (1996) points out that these two tests are geometrically the same. Both compute the dot product between the normal and the vector from a point on the polygon to the eye. In the test that is done in screen space, the eye has been transformed to $(0, 0, \infty)$, and the dot product is thus only the z-component of the polygon vector in screen-space. In theory, what differentiates these tests is the space where the tests are computed-nothing else. In practice, the screen space test is often safer, because edge-on polygons that appear to face slightly backward in eye space can become slightly forward in screen space. This happens because the eye-space coordinates get rounded off to screen-space integer pixel or subpixel coordinates.

Using an API such as OpenGL or DirectX, backface culling is normally controlled with a few functions that either enable backface or frontface culling or disable all culling. Note also that the objects need not be closed (solid) in order to take advantage of backface culling. It suffices to know that only one side of a polygon will be seen. This is often the case for buildings, where wall polygons are visible only from one side. Also, be aware that a mirroring transform (i.e., a negative scaling operation) turns backfacing polygons into frontfacing ones and vice versa (Blinn, 1996).

While backface culling is a simple technique for avoiding the rasterizing of many polygons without due cause, it would be even faster if the CPU could decide with a single test if a whole set of polygons should be sent through the entire pipeline or not. Such techniques are called clustered backface culling algorithms (Akenine-Möller and Haines, 2002). The basic concept that many such algorithms use is the normal cone (Shirman and Abi-Ezzi, 1993). Processing is made faster by dealing with a set of primitives. This can, for example, be a triangle mesh or region of a parametric surface. For each such set, a truncated cone is created that contains all the normal directions of the set, and all the points of the set. Shirman and Abi-Ezzi (1993) prove that if the viewer is

located in the frontfacing cone, then all faces in the cone are frontfacing, and similarly for the backfacing cone.

5.2.3 Detail Culling

Detail culling is a technique that sacrifices quality for speed. The rationale for detail culling is that small details in the scene contribute little or nothing to the rendered images when the viewer is in motion. When the viewer stops, detail culling is usually disabled. Consider an object with a bounding volume, and project this BV onto the projection plane. The area of the projection is then estimated in pixels, and if the number of pixels is below a user-defined threshold, the object is omitted from further processing. For this reason, detail culling is sometimes called screen-size culling. Detail culling can also be done hierarchically on a scene graph. The geometry and rasterizer stages both gain from this algorithm. Note that this could be implemented as a simplified LOD technique, where one LOD is the entire model, and the other LOD is an empty object.

5.2.4 Portal Culling

For architectural models, there is a set of algorithms that goes under the name of portal culling. The first of these were introduced by Airey in 1990. Later, Teller and Séquin (1991) and Teller and Hanrahan (1993) constructed more efficient and more complex algorithms for portal culling. The rationale for all portal-culling algorithms is that walls often act as large occluders in indoor scenes. The idea is therefore to do view frustum culling through each portal (e.g., door or window). When traversing a portal, the frustum is diminished to fit closely around the portal. Therefore, this algorithm can be seen as an extension of view frustum culling. Portals that are outside the view frustum are discarded. Portal culling is a kind of occlusion culling algorithm, but is treated separately because of its importance.

Portal-culling methods preprocess the scene in some way, either automatically or by hand (Akenine-Möller and Haines, 2002). The scene is divided into cells that usually correspond to rooms and hallways in a building. The doors and windows that connect adjacent rooms are called portals. Every object in a cell and the walls of the cell are stored in a data structure that is associated with the cell. We also store information on adjacent cells and the portals that connect them in an adjacency graph. Teller presents algorithms for computing this graph (Teller, 1992). A commonly used alternative is to manually create it.

Luebke and Georges (1995) use a simple method that requires only a small amount of preprocessing. The only information that is needed is the data structure associated with each cell, as described above. An optimization that can well be worth implementing is to use the stencil buffer for more accurate culling. The stencil buffer can be used to mask away rasterization (e.g., texturing and depth test) outside that real portal.

There are many other uses for portals. Mirror reflections can be created by transforming the viewer when the contents of a cell seen through a portal are about to be rendered. That is, if the viewer looks at a portal, then the viewer's position and direction can be reflected in the plane of that portal. Other transformations can be used to create other effects, such as simple refractions. Portals can also be "one-way". For example, assume that you walk from cell A to cell B through a portal. If the portal is one-way, then we cannot go back from B to A-instead we may turn around and see another cell C. This is perfectly suited for creating a difficult maze.

5.2.5 Occlusion Culling

As we have seen, visibility may be solved via a hardware construction called the Z-buffer. Even though it may solve visibility correctly, the Z-buffer is not a very smart mechanism in all respects. For example, imagine that the viewer is looking along a line where 10 spheres are placed (Akenine-Möller and Haines, 2002). An image rendered from this viewpoint will show but one sphere, even though all 10 spheres will be scan-converted and compared to the Z-buffer, and then potentially written to the color buffer and Z-buffer. Depth complexity refers to how many times each pixel is overwritten. In the case of the 10 spheres, the depth complexity is 10 for the pixel in the middle as 10 spheres are rendered there (assuming backface culling was on), and this means that 9 writes to the pixel are unnecessary. This uninteresting scene is not likely to be found in reality, but it describes (from the given viewpoint) a densely populated model.

These sorts of configurations are found in real scenes such as those of a rain forest, an engine, a city, and the inside of a skyscraper. Given the examples in the previous paragraph, it seems plausible that an algorithmic approach to avoid this kind of inefficiency may pay off in terms of speed. Such approaches go under the name of occlusion culling algorithms, since they try to cull away (avoid drawing) objects that are occluded, that is, hidden by other objects in the scene. The optimal occlusion culling algorithm would select only the objects that are visible. In a sense, the Z-buffer selects and renders only those objects that are visible, but not without having to send all objects through most of the pipeline. The idea behind efficient occlusion culling algorithms is to perform some simple tests early on and so avoid sending data through much of the pipeline.

There are two major forms of occlusion culling algorithms, namely point-based and cell-based. Point-based visibility is just what is normally used in rendering, that is, what is seen from a single viewing location. Cell-based visibility, on the other hand, is done for a cell, which is a region of the space, normally a box or a sphere. An invisible object in cell-based visibility must be invisible from all points within the cell. The advantage of cell-based visibility is that once it is computed for a cell, it can usually be used for a few frames, as long as the viewer is inside the cell. However, it is usually more time consuming to compute than point-based visibility. Therefore, it is often done as a preprocessing step, and this is, in fact, the major reason this type of algorithm was developed. Note that point-based and cell-based visibility often are compared to point and area light sources. For an object to be invisible, it has to be in the umbra region, i.e., fully occluded.

One can also categorize occlusion culling algorithms into those that operate in image space, object space, or ray space (Akenine-Möller and Haines, 2002). Image space algorithms do visibility testing in two dimensions after some projection, while object space algorithms use the original three-dimensional objects. Ray space (Bittner and Prikryl, 2001) (Bittner et al., 2001) (Koltun et al., 2001) methods performs their tests in a dual space. Each point (often two-dimensional) of interest is converted to a ray in this dual space. The idea is that testing is simpler, more exact, or more efficient in this space.

Depending on the particular algorithm, O_R represents some kind of occlusion information. O_R is set to be empty at the beginning. After that, all objects (that pass the view frustum culling test) are processed. Consider a particular object. First, we test whether the object is occluded with respect to the occlusion representation O_R. If it is occluded, then it is not processed further, since we then know that it will not contribute to the image. If the object is determined not to be occluded, then that object has to be rendered, since it probably contributes to the image (at that point in the rendering). Then the object is added to P, and if the number of objects in P is large enough, then we can afford to merge the occluding power of these objects into O_R. Each object in P can thus be used as an occluder. Depending on the algorithm, the merging can be done at different frequencies. Also, all algorithms cannot afford to merge all objects in P. Therefore, one often estimates how good an occluder is and only merges the good ones. This last operation in the pseudocode is very important as it provides a mechanism to fuse occluders (Zhang et al., 1997). This means that several occluders together can occlude more than each occluder considered as a single entity. Occluder fusion is essential to get good cull rates. However, not all algorithms can fuse occluders.

Occluder fusion is even more important for cell-based than for point- based algorithms, as the occluded space for cell-based algorithms is smaller. For conservative visibility algorithms, the visibility test needs to overestimate the object to be tested for occlusion. This is often done with a bounding volume around the object. However, if the object is to be used as an occluder (inserted into the O_R), the occluding power of the object needs to be underestimated.

For some algorithms, it is expensive to update the occlusion representation, so this is only done once (before the actual rendering starts) with the objects that are believed to be good occluders. This set is then updated from frame to frame.

Note that for the majority of occlusion culling algorithms, the performance is dependent on the order in which objects are drawn (Akenine-Möller and Haines, 2002). As an example, consider a car with a motor inside it. If the hood of the car is drawn first, then the motor will (probably) be culled away. On the other hand, if the motor is drawn first, then the hood of the car will not be culled. Therefore, performance can be improved by techniques such as rough front-to-back sorting of the objects by their approximate distance from the viewer and rendering in this order. Also, it is worth noting that small objects potentially can be excellent occluders, since the distance to the occluder decides how much it can occlude. As an example, a match- box can occlude the Golden Gate Bridge if the viewer is sufficiently close to the matchbox.

5.3 Level of Detail

The basic idea of Levels of Detail (LODs) is to use simpler versions of an object as it makes less and less of a contribution to the rendered image (Akenine-Möller and Haines, 2002). For example, consider a detailed car that may consist of 10, 000 triangles. This representation can be used when the viewer is close to the car. When the object is farther away, say covering only 10 x 5 pixels, we do not need all 10,000 triangles. Instead, we can use a simplified model that has only, say, 100 triangles. Due to the distance, the simplified version looks approximately the same as the more detailed version. In this way, a significant speedup can be expected. Note also that fog is often used together with LODs. This allows us to completely skip the rendering of an object as it enters opaque fog. Also, the fogging mechanism can be used to implement time-critical rendering. By moving the far plane closer to the viewer, objects can be culled earlier and more rapid rendering achieved to keep the frame rate up.

Some objects, such as spheres, Bezier surfaces, and subdivision surfaces have levels of detail as part of their geometrical description. The underlying geometry is curved, and

a separate LOD control determines how it is tessellated into displayable polygons.

In general, LOD algorithms consist of three major parts, namely, generation, selection, and switching. LOD generation is the part where different representations of a model are generated with different detail. Simplification methods can be used to generate the desired number of LODs. Another approach is to make models with different levels of detail by hand. The selection mechanism chooses a level of detail model based on some criteria, such as estimated area on the screen. Finally, we need to change from one level of detail to another, and this process is termed LOD switching.

When switching from one LOD to another, an abrupt model substitution is often noticeable and distracting. This difference is called popping. Several different ways to perform this switching exists, and they all have different popping traits.

In the simplest type of LOD algorithm, the different representations are models of the same object containing different numbers of primitives. This algorithm is well-suited for modern graphics hardware (GPU) (Luebke et al., 2002), because the LODs can be turned into indexed triangle strips and pulled directly from DMA memory. A more detailed LOD has a higher number of primitives. The switching from one LOD to another just happens, that is, on the current frame a certain LOD is used. Then on the next frame, the selection mechanism selects another LOD, and immediately uses that for rendering. Popping is typically the worst for this type of LOD method.

Conceptually, a pretty obvious way to switch from one LOD to another is possible. Just do a linear blend between the two LODs over a short period of time. This is definitely going to make for a smoother switch. However, it is expensive to make such blends. Rendering two LODs for one object is naturally more expensive than just rendering one LOD, so this somewhat defeats the purpose of LODs. However, LOD switching takes place during only a short amount of time, and often not for all objects in a scene at the same time, so this may very well be profitable.

Since the results of a blending operation depend on the current contents of the frame buffer, care has to be taken to draw the models in an order that does not lead to additional artifacts due to the blend. Giegl and Wimmer (2007) propose a method that works well in practice. Assume a transition between two LODs-say LOD1 and LOD2-is desired, and that LOD1 is the current LOD being rendered. Now instead of drawing both LODs transparently, first draw LOD1 opaquely to the frame buffer (both color and Z). Then fade in LOD2 by increasing its alpha value from 0 to 1 and using the "over" blend mode. When LOD2 has faded so it is completely opaque, it is turned into the current LOD, and LOD1 will start to fade out. The LOD that is being faded (in or out) should be rendered with the z-test enabled and z-writes disabled. Note that

in the middle of the transition, both LODs are rendered opaquely, one on top of the other.

The advantage of the method is that it works on current graphics hardware and is simple to implement (Akenine-Möller and Haines, 2002). It also avoids the problems usually associated with alpha blending by always drawing one of the LODs opaquely. Sometimes the silhouettes of the different LODs might not match very well. In such cases, it is advisable to draw all opaque LODs first, and the LODs currently being faded afterwards. This makes sure a correct Z-buffer is established before transparent objects are drawn. The technique works best if the transition intervals are kept short, which also helps keeping the rendering overhead small.

Chapter 6

Hybrid Parallel Renderer (HPR)

6.1 What is HPR

Hybrid Parallel Renderer (HPR) is the real-time hybrid-parallel rendering software which is prototyped by implementing the research which is covered in this thesis. Main goals of the HPR are:

1. Complete computations in real-time
2. Utilize multi-core, multi-CPU, multi GPU, and multi-PC environment
3. Compute high quality images using ray tracing and radiosity
4. Handle complex and large scenes

Compared to conventional renderers which are based on direct illumination models, achieving these goals requires enormous amount of CPU, GPU, memory, and network resources. The solution to satisfy these requirements is to obtain the massive computational power from parallel processing using cost efficient PC cluster consisting of multi-core CPU, and multi-GPU machines interconnected by Gigabit Ethernet.

The benefits gained from the hardware developments apply the same way even if multiple computers are used. Especially, the developments in the networking environment will surely be the power to move forward the use of PC clustering.

Computer hardware technology is constantly evolving. Cheaper and faster components are released to the market everyday. However, the efficiency of image rendering using only one computer is limited mainly by the computer's performance. If more performance is required, lots of investment must be made on the high-end computers.

Because of this, the efficient methodology for the future is to combine the power of more than one cost-effective machines to satisfy the demand for computational power. This solution also brings forth the way of long-time use of computers.

The architecture of such rendering system is desired to extend reliably for years. Therefore, the flexibility to allow addition of various improvements is important. Furthermore, the renderer is intended to be the test environment for experimenting various rendering ideas.

HPR may render multiple frames while the scene data is resident in the memory. When HPR started, commands can be entered to modify the scene data before starting the rendering of the next frame. The detailed rendering control commands are sent to this console screen either directly or through socket communication. HPR waits for the next command until it is explicitly told to terminate.

Every part of the HPR such as ray tracing, shading, and network communication is designed to be processed in parallel. Shared memory parallel processing is implemented using OpenMP to maximize the performance on multi-core and multi-CPU systems.

All computations in HPR's distributed rendering processing are done in the message passing system using networked machines. Scene data are processed in sequence to allow the computation to continue over the network. HPR uses OpenMPI for the communication between multiple machines.

HPR's base rendering algorithm is Monte Carlo ray tracing which is simple but powerful, and perfectly suitable for parallel processing based on message passing. However, ray tracing requires all scene data to be loaded in the memory. This requirement becomes an obstacle in achieving HPR's goal to render large scenes. There are many proposed methods to overcome this problem. Paging the required data from disk is one the common solutions. In some cases, HPR distributes the large scene data to multiple machines.

Since implementation of photon map on a Monte Carlo ray tracer is easy and the algorithm reliably works, photon mapping is used for the global illumination computation.

6.2 System Design

If the scene to be rendered is simple and small enough to fit in the memory of a computer, HPR sends same copies of the entire scene data to all requesting computers and ray tracing is executed in parallel. In this case, the speed-up is proportional to

the number of computers. However, if the scene is large and cannot be stored in the memory of a computer, alternative ray tracing methodologies are considered.

Two rules for designing scene distribution methodology and rendering algorithms are:

1. Scene data should be loaded in memory
2. Ray data should passed between computers

Results of several experiments have shown that exchanging a part of the scene geometry data between computers results in too much communication overhead. This is because geometry data structure is complex and not suitable for partial transfers. On the other hand, ray data are independent from each other and are easier to be transferred between computers.

6.2.1 Processing Nodes

The HPR consists of some processing nodes. These are:

ManagerNode which distributes the workload of rendering image to the render nodes. The ManagerNode distributes the workload between the RenderNodes by dividing the image into many image tiles and assigning them to RenderNodes on demand.

SceneDataNode which stores and distributes the scene graph data to RenderNodes. Also SceneDataNode is in responsible for image and object space coherence.

RenderNode which performs the rendering of a scene data to an image. Key principle of the HPR is to render a single frame by distributing it to multiple render nodes. All RenderNodes have access to an identical copy of an the scene graph data.

AssemblyNode which receives frame tiles from all RenderNodes, and assembles them to a form a composite image buffer. As there is one dedicated tile assembly node for each display node, the nodes receive only those tiles of the rendered image stream that are relevant for the particular display node that they are responsible. The information about the given detail of rendered images is part of the communication between a RenderNode and AssemblyNode. Because of this a RenderNode knows the rendered tiles that have to be forwarded to a specific tile assembly node. By this way, there is no communication overhead occurs for parts of the image that would not be displayed later.

CodecNode has the functionality to compress image data in a desired format. These nodes take place between the AssemblyNode and DisplayNode. Encoding the image data in between tile assembly and display can enormously reduce the network traffic and

allows rendering across high-latency or low-bandwidth networks (e.g., the Internet). HPR may provide multiple configurations of encoder and decoder nodes.

DisplayNode which composes the chain of nodes, and presents any incoming image data synchronized according to its timestamp. HPR may allow to have more than one display nodes relate to the same rendering process, where a single display node represents one partial view of the rendered image, which can either be a part of the overall frame in a multi-display setup or a new view of the scene.

6.3 Implementation

The implementation of HPR's ray tracing algorithm which follows the above mentioned guidelines under the distributed environment is composed of two parts: scene distribution and distributed ray tracing.

6.3.1 Scene Distribution

Large scenes are distributed and stored in multiple computers. The size of data to be stored on each computer is determined by the physical memory it has. In most cases, the scene data is distributed to multiple machines.

HPR needs to exchange data across multiple cores, multiple CPUs, multiple GPUs and multiple machines. For these purposes OpenMP, Brook+ and OpenMPI are utilized.

Multi-thread implementation using OpenMP was used throughout the development of HPR. The number of main engine threads is depends on the number of CPUs in the computer where the process is running. That is, there will be two engine threads on dual core/CPU computers and one engine thread on single core/CPU computers. The main engine thread processes many tasks by scheduling and switching these tasks. Namely, ray tracing process, shading process, and other procedures within the lookup operation switch the execution continuously in the main engine thread.

This mechanism allows more accurate priority control and management of the execution components of HPR's pipeline. If each processing stage is implemented as an separate thread, the priority control must rely on OpenMP's scheduler. This complicates the precise priority adjustment. In HPR, the main engine thread exists as one OpenMP thread, and various stages making up the rendering pipeline are implemented as separate function calls. The stages of the pipeline are composed by local queues. An execution of a stage takes out an item from its input queue, process it, and write

out the result to the output queue. Because the stages are implemented as individual functions and they are connected by input and output queues, the main engine thread can freely change the execution order of the stages. This allows proper and precise execution priority control as needed. Because of these, minimizing the local queue size and latency in operations is possible.

6.3.2 Distributed Ray Tracing

A ray tracing computing in the distributed environment is executed by replicating exact copies of a ray data to all machines with distributed scene data. For each computer, ray tracing is executed according to the received ray data. Finally, the original machine receives results of ray tracing from all machines with distributed scene data.

If multiple sets of machines can be prepared for HPR to hold the entire scene data, ray tracing can be executed in parallel on each set independently. As the first step, the ray tracing is executed independently for each tile on the screen. The shading computations start when some of the screen rays will hit objects. The shading computation is processed in parallel with ray tracing. Depending on the shader used, more rays such as reflection, and shadow rays may be calculated. These rays are processed in parallel just like primary rays.

HPR process rays with higher generations first by entering the ray data in the input queue of the ray tracing engine. The engine takes the queued data in series to process them. If there are multiple sets of computers to hold the scene data, the computation workload is balanced by pulling the queued data out by first-come-first-served basis.

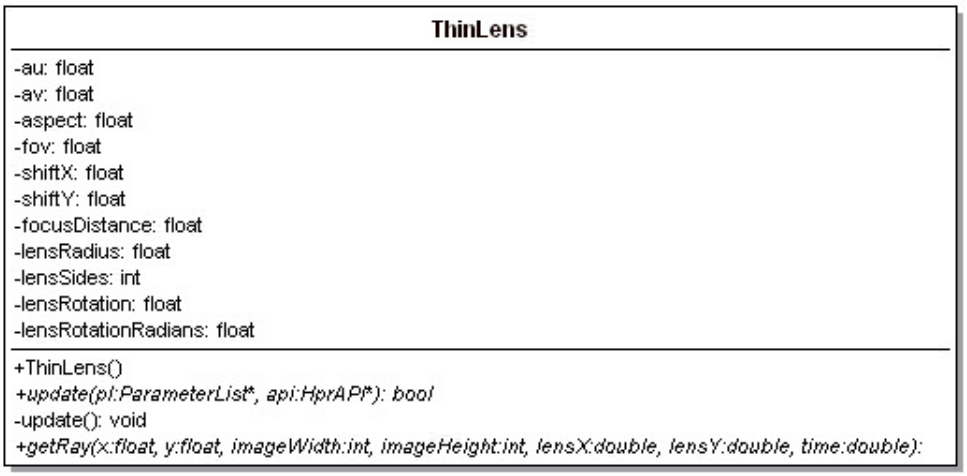
HPR's distributed ray tracing approach handles equally if the origin or the direction of the ray is different. In addition to classic ray tracing computation, HPR ray traces the photons which makes sorting of the ray data by their positions or directions quite difficult. Using the algorithm which provides consistent performance for all ray tracing requests in the global illumination rendering is important because of this requirement. This consistency provides simplicity in balancing the load of multiple CPUs.

Since all ray tracing calculations can be computed with the maximum of two data transmissions between machines, all ray tracing requests can be processed with almost equal latency. Depending on the ray tracing request, if the algorithm requires unpredictable number of data transmissions between computers, the variation in the computation time complicates the load balancing of the entire system. In order to keep the implementation simple, the consistency in latency of the ray tracing computation is important.

6.3.3 Structure of the Source Code

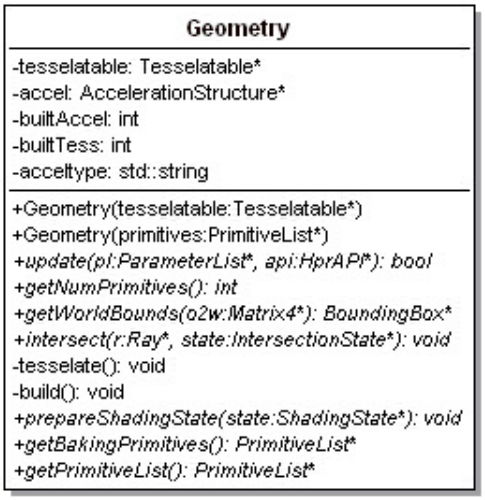
HPR source code is grouped into several namespaces according to their functionality. Camera, Display, Filter, Light, Parser, Primitive, Renderer, and Shader are some of them.

Figure 6.1: Camera Class Diagram



The Camera namespace (Figure 6.6) consists of projection lens related classes; FisheyeLens, PinholeLens, SphericalLens, and ThinLens (Figure 6.1). These classes determine the visual effects applied to the final image.

Figure 6.2: Geometry Class Diagram



FastDisplay, FileDisplay, FrameDisplay, and ImgPipedisplay classes are part of the Display namespace and they are responsible for on screen projection of respective properies.

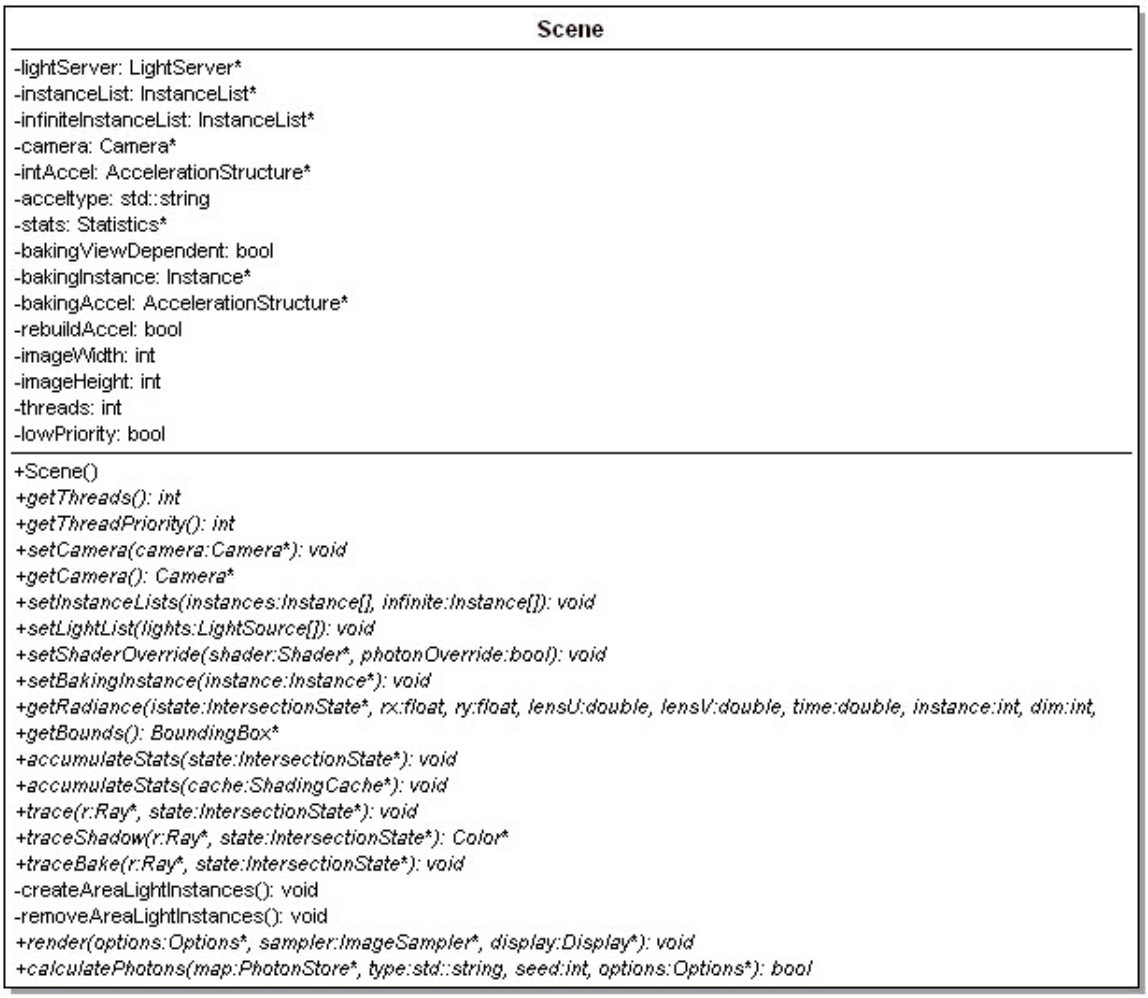
Classes in the Filter namespace are in charge of filtering the processed image in order to apply some visual effects. BoxFilter, CubicBSpline, GaussianFilter, SincFilter, and TriangleFilter are some of these classes.

DirectionalSpotlight, ImageBasedLight, PointLight, SphereLight, SunSkyLight, and TriangleMeshLight classes take place in the Light namespace (Figure 6.7). Their functionality is to provide light into the scene.

The Primitive namepace (Figure 6.8) contains classes like Background, Box, CubeGrid, Cylinder, ParticleSurface, Plane, QuadMesh, Sphere, Torus, and TriangleMesh. These are primitive base objects which are used to represent scene objects.

BucketRenderer, MultipassRenderer, ProgressiveRenderer, SimpleRenderer are some of the classes that take place in the Renderer namespace (Figure 6.9). Their functionallity is to render the scene data.

Figure 6.3: Scene Class Diagram

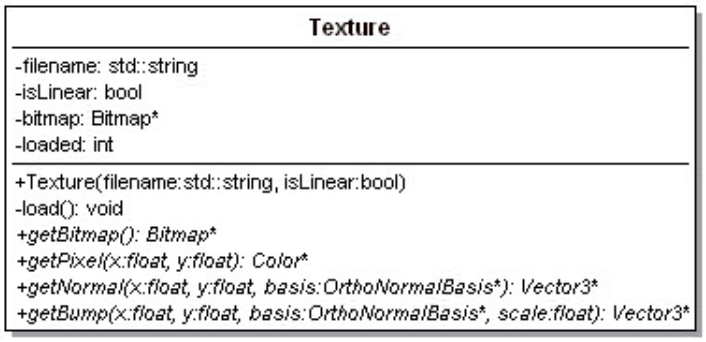


Classes from the Shader namespace (Figure 6.10) are in charge of shading the scene

Figure 6.4: Ray Class Diagram

data. AmbientOcclusionShader, ConstantShader, DiffuseShader, MirrorShader, NormalShader, PhongShader, ShinyDiffuseShader, SimpleShader, TexturedAmbientOcclusionShader, TexturedDiffuseShader, TexturedPhongShader, TexturedWardShader, UVShader, and WireframeShader are name to some.

Figure 6.5: Texture Class Diagram

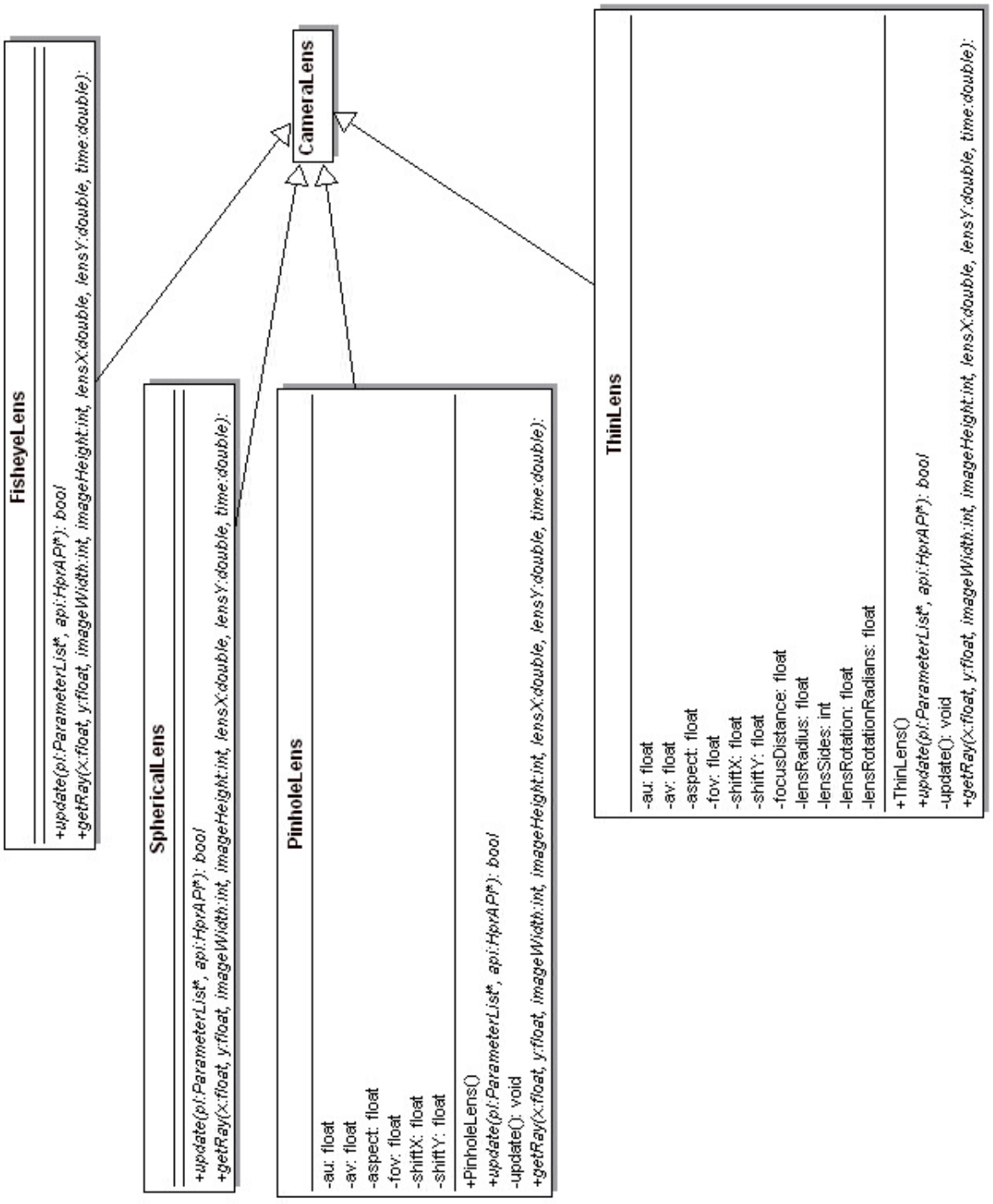


Classes like Geometry (Figure 6.2), Ray (Figure 6.4), Scene (Figure 6.3), and Texture (Figure 6.5) take place in the main namespace.

6.3.4 Performance Analysis

In order to find out the participation of components of the HPR to the overall rendering process, a series of performance anaylysis tests are executed. In these test, a sample

Figure 6.6: Camera Relation Diagram



scene consisting of two shiny monkey faces and a smooth background is used. This monkey face is known as 'Suzanne' the Blender monkey (Anonymous, undated), since it is one of the primitive objects of the Blender 3D modelling software.

As seen from the speedup data from hybrid OpenMPI-OpenMP-Brook+ comparison

(Table 6.1), performance gain from the increase of CPU cores and PCs (computers) are nearly proportional. The difference is caused by the data transfer latency between computer nodes. In the test configuration, Gigabit Ethernet (10000 Mbit/s, 1000BASE-T, IEEE 802.3ab) via Link Aggregation Control Protocol (IEEE 802.3ad) namely link bundling is used in order to increase PC-to-PC communication speed as much as possible.

On the other hand, gain from the GPU (stream) processing is less satisfactory. This may be due to insufficient optimization. Also, some parts of the ray tracing algorithms and calculations are not particularly suited to the GPU. So, some algorithms require extensive modifications. Nevertheless, the HPR code which is used in these tests needs much more optimization.

It is clear that better algorithm parellization and memory/cache utilization can lead to more improved results. Especially parallelizing algorithms by considering asymmetric processors (CPUs, and GPUs) which have different computing capabilities (SIMD vs. SISD methodology, different memory access characteristics, etc.), and complex parallelization configurations (CPU to CPU, CPU to GPU, GPU to GPU, core to core, computer to computer, etc.).

Table 6.1: Speedup data from hybrid OpenMPI/OpenMP/Brook+

# of CPU Cores (w/ OpenMP)	# of GPUs (w/ Brook+)	# of PCs (w/ OpenMPI)	Render Time (seconds)	Achived Speedup
1	1	1	7:16	-
2	1	1	3:39	1.99x
3			2:29	2.93x
4			1:41	4.32x
1	2	1	5:06	1.42x
	3		3:29	2.09x
	4		2:22	3.07x
1	1	2	4:01	1.81x
		3	2:45	2.64x
		4	1:55	3.79x
2	1	2	2:04	3.52x
		3	1:23	5.25x
		4	1:03	6.92x
3	1	2	1:23	5.25x
		3	0:56	7.79x
		4	0:40	10.09x
4	1	2	1:01	7.15x
		3	0:41	10.63x
		4	0:29	15.03x

Figure 6.7: Light Relation Diagram

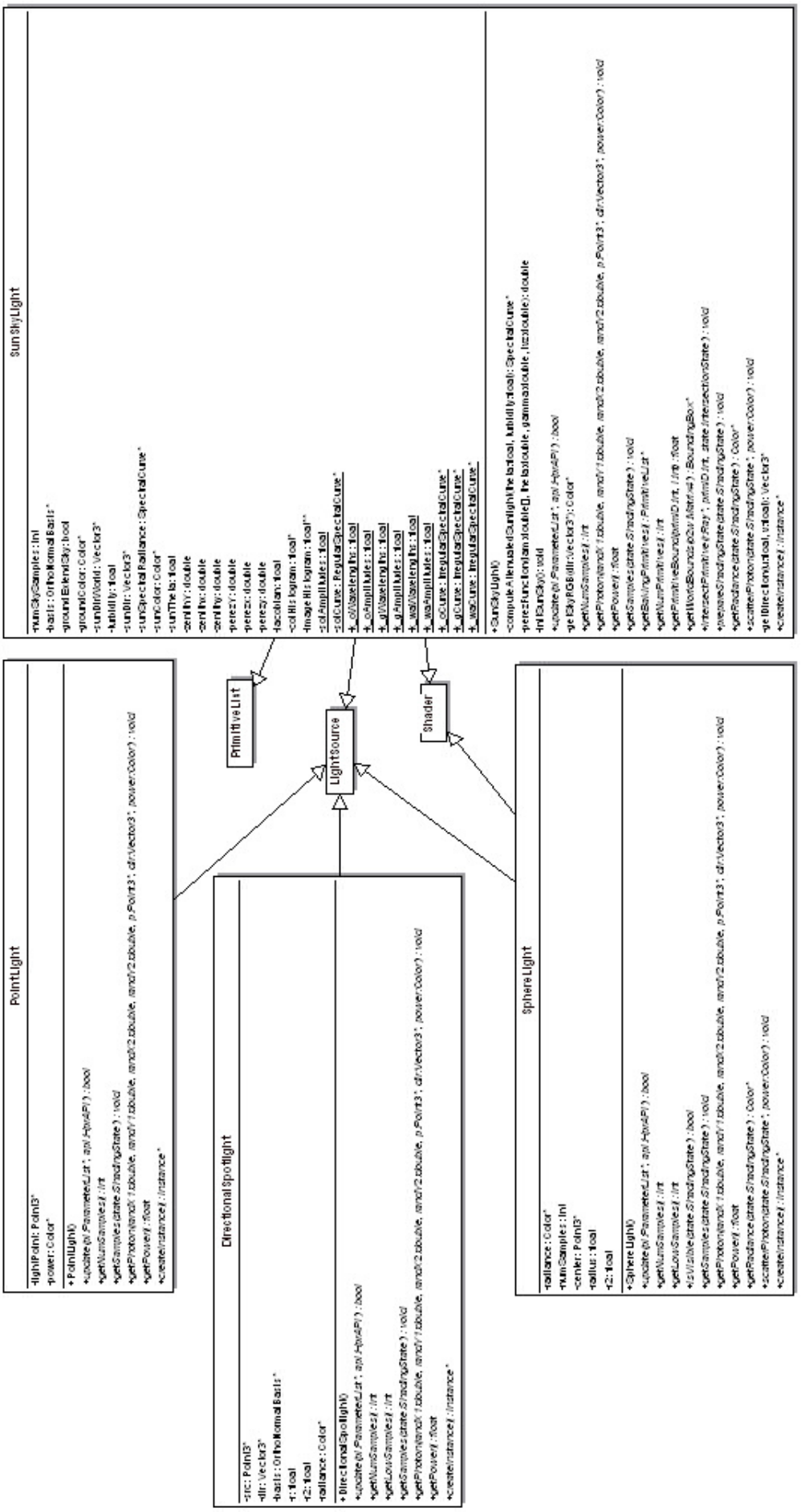


Figure 6.8: Primitive Relation Diagram

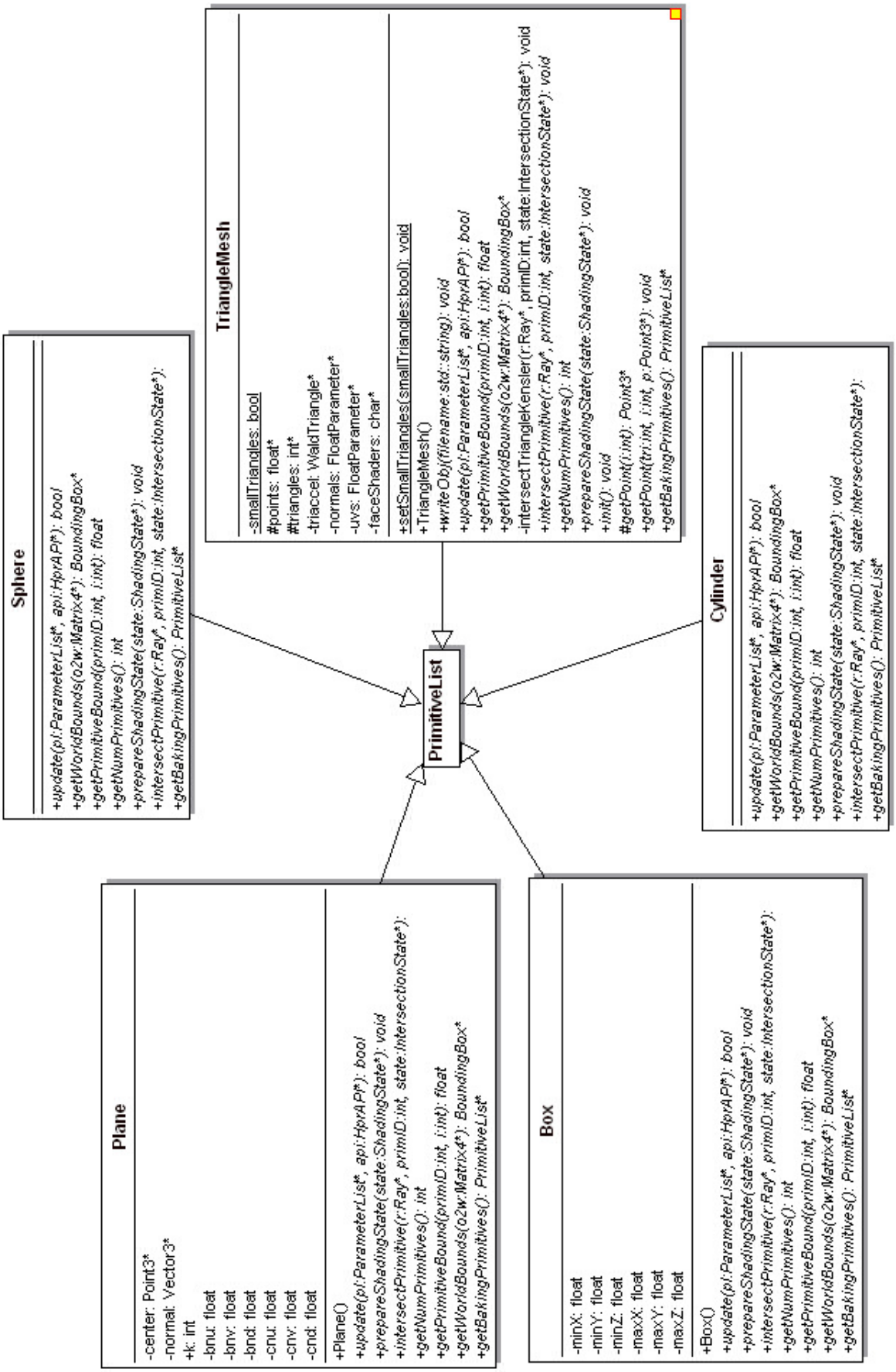


Figure 6.9: Renderer Relation Diagram

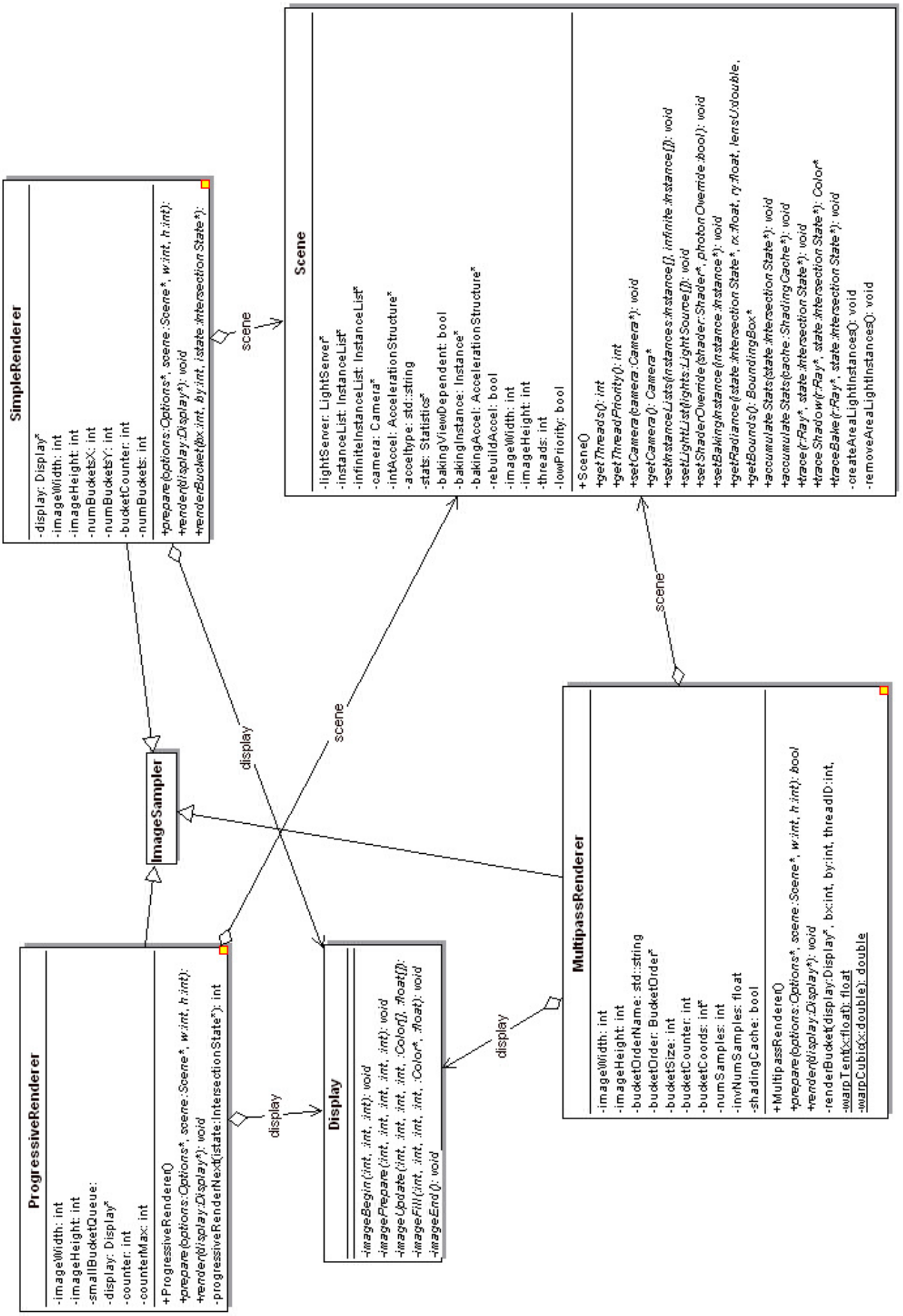


Figure 6.10: Shader Relation Diagram

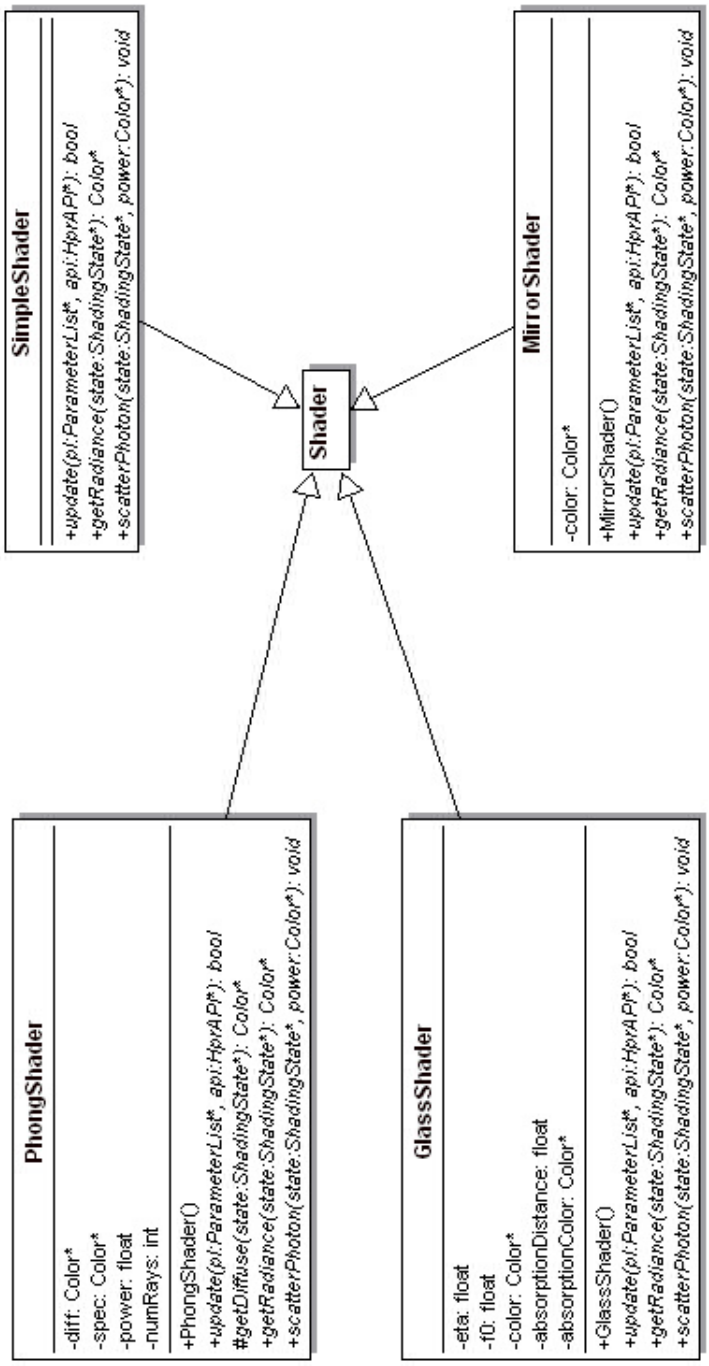
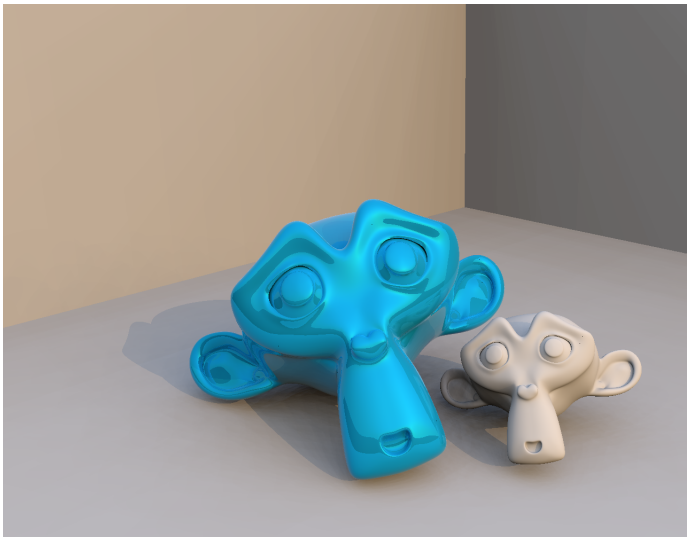


Figure 6.11: Shiny Monkeys ('Suzanne', The Blender monkey) (1280x1024 resolution)



Chapter 7

Conclusions

The HPR is prototyped as a fully scalable parallel and distributed system achieving exceptional computational power by using an affordable multi-core, multi-CPU, multi-GPU PC cluster. Depending on the demanded speed and scene size, the number of machines to be used can be flexibly determined.

The use of distributed computing for ray tracing and data-flow oriented shading engine resulted in excellent load balancing to keep all CPUs constantly busy, and a general solution to model a shading process in a parallel computing environment. The final version is capable of global illumination rendering in the sample scenes.

Various new rendering algorithms can be tested by developing simple plug-in's. Also, the methods on how to parallelize algorithms can also be tested easily. These indicate that HPR may succeed in providing a test environment for different rendering and parallel processing algorithms.

Further speed-up, optimization, and feature improvements are possible. A lot of room for the improvements are still exist, especially for speeding up the rendering process. However, the original two goals to render global illumination and processing large scenes are mostly achieved using practical algorithms.

Hybrid Parallel Rendering is executed on asymmetric processors. During the implementation of the HPR's parallel rendering algorithms, it's seen that most of the existing research in parallel algorithms is based on symmetric processing. However, considering hybrid parallel processing requirements, even asymmetric processing considerations are not enough asymmetric.

Asymmetric processors (CPUs, and GPUs) have different computing capabilities (SIMD vs. SISD methodology, different memory access characteristics, etc.), and complex parallelization configurations (computer to computer, core to core, CPU to CPU, CPU

to GPU, GPU to GPU, etc.). Therefore parallelizing algorithms by simply dividing them into parallel executable parts which require equivalent processing capabilities is not very efficient in hybrid parallel processing. Algorithms must be parallelized by considering computaional capabilities and diversities of the participating processors.

Biography

Reha Cenani holds a Bachelor of Science (B.S.) degree in Computer Engineering from Marmara University of Istanbul, Turkey.

His primary research interests include distributed parallel computing, computer graphics, and e-commerce. Reha's work experience includes training, teaching, retail product management and software development.

Prior to joining the masters program in Computer Engineering at the Dogus University of Istanbul, Reha worked for 10 years as a product manager and software architect at a leading retail chain.

Currently, he works as a part-time lecturer in the Dogus Univeristy.

Appendix A

Program Source Code

Listing A.1: Texture.h

```
#pragma once
#include "Bitmap.h"
#include "FileUtils.h"
#include "UI.h"
#include "PluginRegistry.h"
#include "BitmapBlack.h"
#include "Color.h"
#include "MathUtils.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
#include <string>

namespace hpr.core {

  using hpr::PluginRegistry;
  using hpr::image::Bitmap;
  using hpr::image::BitmapReader;
  using hpr::image::Color;
  using hpr::image::BitmapReader::BitmapFormatException;
  using hpr::image::formats::BitmapBlack;
  using hpr::math::MathUtils;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Vector3;
  using hpr::system::FileUtils;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  // Represents a 2D texture, typically used by shaders
  class Texture {
```

```
    private:
    std::string filename;
    bool isLinear;
    Bitmap *bitmap;
    int loaded;
    void load();

    public:
    Texture(std::string filename, bool isLinear);
    virtual Bitmap *getBitmap();
    virtual Color *getPixel(float x, float y);
    virtual Vector3 *getNormal(float x, float y, OrthoNormalBasis *basis);
    virtual Vector3 *getBump(float x, float y, OrthoNormalBasis *basis, float scale);
  };
}
```

Listing A.2: Texture.cpp

```
#include "Texture.h"

using hpr::PluginRegistry;
using hpr::image::Bitmap;
using hpr::image::BitmapReader;
using hpr::image::Color;
using hpr::image::BitmapReader::BitmapFormatException;
using hpr::image::formats::BitmapBlack;
using hpr::math::MathUtils;
using hpr::math::OrthoNormalBasis;
using hpr::math::Vector3;
using hpr::system::FileUtils;
using hpr::system::UI;
using hpr::system::UI::Module;

Texture::Texture(std::string filename, bool isLinear) {
  this->filename = filename;
  this->isLinear = isLinear;
  loaded = 0;
}

void Texture::load() {
  if (loaded != 0) {
    return;
  }
  std::string extension = FileUtils::getExtension(filename);
  try {
    UI::printInfo(Module::TEX, "Reading texture bitmap from: \"%s\" ...", filename);
```

```
    BitmapReader *reader = PluginRegistry::bitmapReaderPlugins->createObject(extension);
    if (reader != 0) {
      bitmap = reader->load(filename, isLinear);
      if (bitmap->getWidth() == 0 || bitmap->getHeight() == 0) {
        delete bitmap;
      }
    }
    if (bitmap == 0) {
      UI::printError(Module::TEX, "Bitmap reading failed");
      bitmap = new BitmapBlack();
    } else {
      UI::printDetailed(Module::TEX, "Texture bitmap reading complete: %dx%d pixels found",
          bitmap->getWidth(), bitmap->getHeight());
    }
  } catch (IOException *e) {
    UI::printError(Module::TEX, "%s", e->getMessage());
  } catch (BitmapFormatException *e) {
    UI::printError(Module::TEX, "%s format error: %s", extension, e->getMessage());
  }
  loaded = 1;
}

Bitmap *Texture::getBitmap() {
  if (loaded == 0) {
    load();
  }
  return bitmap;
}

Color *Texture::getPixel(float x, float y) {
  Bitmap *bitmap = getBitmap();
  x = MathUtils::frac(x);
  y = MathUtils::frac(y);
  float dx = x * (bitmap->getWidth() - 1);
  float dy = y * (bitmap->getHeight() - 1);
  int ix0 = static_cast<int> (dx);
  int iy0 = static_cast<int> (dy);
  int ix1 = (ix0 + 1) % bitmap->getWidth();
  int iy1 = (iy0 + 1) % bitmap->getHeight();
  float u = dx - ix0;
  float v = dy - iy0;
  u = u * u * (3.0f - (2.0f * u));
  v = v * v * (3.0f - (2.0f * v));
  float k00 = (1.0f - u) * (1.0f - v);
  Color *c00 = bitmap->readColor(ix0, iy0);
```

```
  float k01 = (1.0f − u) ∗ v;
  Color ∗c01 = bitmap−>readColor(ix0, iy1);
  float k10 = u ∗ (1.0f − v);
  Color ∗c10 = bitmap−>readColor(ix1, iy0);
  float k11 = u ∗ v;
  Color ∗c11 = bitmap−>readColor(ix1, iy1);
  Color ∗c = Color::mul(k00, c00);
  c−>madd(k01, c01);
  c−>madd(k10, c10);
  c−>madd(k11, c11);
  return c;
}

Vector3 ∗Texture::getNormal(float x, float y, OrthoNormalBasis ∗basis) {
  float ∗rgb = getPixel(x, y)−>getRGB();
  return basis−>transform(new Vector3(2 ∗ rgb[0] − 1, 2 ∗ rgb[1] − 1, 2 ∗ rgb[2] − 1))−>
      normalize();
}

Vector3 ∗Texture::getBump(float x, float y, OrthoNormalBasis ∗basis, float scale) {
  Bitmap ∗bitmap = getBitmap();
  float dx = 1.0f / bitmap−>getWidth();
  float dy = 1.0f / bitmap−>getHeight();
  float b0 = getPixel(x, y)−>getLuminance();
  float bx = getPixel(x + dx, y)−>getLuminance();
  float by = getPixel(x, y + dy)−>getLuminance();
  return basis−>transform(new Vector3(scale ∗ (b0 − bx), scale ∗ (b0 − by), 1))−>normalize();
}
}
```

Listing A.3: Scene.h

```
#pragma once
#include "AccelerationStructure.h"
#include "PrimitiveList.h"
#include "LightSource.h"
#include "Shader.h"
#include "Point3.h"
#include "Vector3.h"
#include "BoundingBox.h"
#include "Color.h"
#include "Display.h"
#include "ImageSampler.h"
#include "FrameDisplay.h"
#include "UI.h"
#include "MathUtils.h"
```

```
#include "PhotonStore.h"
#include <string>
#include <vector>

namespace hpr.core {

  using hpr::core::display::FrameDisplay;
  using hpr::image::Color;
  using hpr::math::BoundingBox;
  using hpr::math::MathUtils;
  using hpr::math::Point3;
  using hpr::math::Vector3;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  // Represents a entire scene, defined as a collection of instances viewed by a camera.
  class Scene {
    private:
    LightServer *lightServer;
    InstanceList *instanceList;
    InstanceList *infiniteInstanceList;
    Camera *camera;
    AccelerationStructure *intAccel;
    std::string acceltype;
    Statistics *stats;
    bool bakingViewDependent;
    Instance *bakingInstance;
    PrimitiveList *bakingPrimitives;
    AccelerationStructure *bakingAccel;
    bool rebuildAccel;
    int imageWidth;
    int imageHeight;
    int threads;
    bool lowPriority;
    void createAreaLightInstances();
    void removeAreaLightInstances();

    public:
    Scene(); // Creates an empty scene.
    virtual int getThreads(); // Get number of allowed threads for multi-threaded operations.
    virtual int getThreadPriority(); // Get the priority level to assign to multi-threaded operations
        .
    virtual void setCamera(Camera *camera); // Sets the current camera (no support for multiple
        cameras yet).
    virtual Camera *getCamera();
```

```
    virtual void setInstanceLists(Instance instances[], Instance infinite[]); // Update the instance
        lists for this scene.
    virtual void setLightList(LightSource lights[]); // Update the light list for this scene.
    virtual void setShaderOverride(Shader *shader, bool photonOverride); // Enables shader
        overiding (set null to disable). The specified shader will be used to shade all surfaces.
    virtual void setBakingInstance(Instance *instance); // The provided instance will be considered
         for lightmap baking.
    virtual ShadingState *getRadiance(IntersectionState *istate, float rx, float ry, double lensU,
        double lensV, double time, int instance, int dim, ShadingCache *cache); // Get the
        radiance seen through a particular pixel.
    virtual BoundingBox *getBounds(); // Get scene world space bounding box.
    virtual void accumulateStats(IntersectionState *state);
    virtual void accumulateStats(ShadingCache *cache);
    virtual void trace(Ray *r, IntersectionState *state);
    virtual Color *traceShadow(Ray *r, IntersectionState *state);
    virtual void traceBake(Ray *r, IntersectionState *state);
    virtual void render(Options *options, ImageSampler *sampler, Display *display); // Render the
         scene using the specified options, image sampler and display.
    virtual bool calculatePhotons(PhotonStore *map, std::string type, int seed, Options *options);
        // Create a photon map as prescribed by the given PhotonStore.
  };
}
```

Listing A.4: Scene.cpp

```
#include "Scene.h"

using hpr::core::display::FrameDisplay;
using hpr::image::Color;
using hpr::math::BoundingBox;
using hpr::math::MathUtils;
using hpr::math::Point3;
using hpr::math::Vector3;
using hpr::system::UI;
using hpr::system::UI::Module;

Scene::Scene() {
  lightServer = new LightServer(this);
  instanceList = new InstanceList();
  infiniteInstanceList = new InstanceList();
  acceltype = "auto";
  stats = new Statistics();

  bakingViewDependent = false;
  delete bakingInstance;
  delete bakingPrimitives;
```

```
  delete bakingAccel;
  delete camera;
  imageWidth = 640;
  imageHeight = 480;
  threads = 0;
  lowPriority = true;

  rebuildAccel = true;
}

int Scene::getThreads() {
  return threads <= 0 ? Runtime::getRuntime()->availableProcessors() : threads;
}

int Scene::getThreadPriority() {
  return lowPriority ? Thread::MIN_PRIORITY : Thread::NORM_PRIORITY;
}

void Scene::setCamera(Camera *camera) {
  this->camera = camera;
}

Camera *Scene::getCamera() {
  return camera;
}

void Scene::setInstanceLists(Instance instances[], Instance infinite[]) {
  infiniteInstanceList = new InstanceList(infinite);
  instanceList = new InstanceList(instances);
  rebuildAccel = true;
}

void Scene::setLightList(LightSource lights[]) {
  lightServer->setLights(lights);
}

void Scene::setShaderOverride(Shader *shader, bool photonOverride) {
  lightServer->setShaderOverride(shader, photonOverride);
}

void Scene::setBakingInstance(Instance *instance) {
  bakingInstance = instance;
}

ShadingState *Scene::getRadiance(IntersectionState *istate, float rx, float ry,
```

```
    double lensU, double lensV, double time, int instance, int dim,
    ShadingCache *cache) {
  istate->numEyeRays++;
  float sceneTime = camera->getTime(static_cast<float> (time));
  if (bakingPrimitives == 0) {
    Ray *r = camera->getRay(rx, ry, imageWidth, imageHeight, lensU, lensV, sceneTime);
    return r != 0 ? lightServer->getRadiance(rx, ry, sceneTime, instance, dim, r, istate, cache) : 0;
  } else {
    Ray *r = new Ray(rx / imageWidth, ry / imageHeight, -1, 0, 0, 1);
    traceBake(r, istate);
    if (!istate->hit()) {
      return 0;
    }
    ShadingState *state = ShadingState::createState(istate, rx, ry, sceneTime, r, instance, dim,
        lightServer);
    bakingPrimitives->prepareShadingState(state);
    if (bakingViewDependent) {
      state->setRay(camera->getRay(state->getPoint(), sceneTime));
    } else {
      Point3 *p = state->getPoint();
      Vector3 *n = state->getNormal();
      // create a ray coming from directly above the point being shaded
      Ray *incoming = new Ray(p->x + n->x, p->y + n->y, p->z + n->z, -n->x, -n->
          y, -n->z);
      incoming->setMax(1);
      state->setRay(incoming);
    }
    lightServer->shadeBakeResult(state);
    return state;
  }
}

BoundingBox *Scene::getBounds() {
  return instanceList->getWorldBounds(0);
}

void Scene::accumulateStats(IntersectionState *state) {
  stats->accumulate(state);
}

void Scene::accumulateStats(ShadingCache *cache) {
  stats->accumulate(cache);
}

void Scene::trace(Ray *r, IntersectionState *state) {
```

```
  state->numRays++;
  state->instance = 0;
  state->current = 0;
  for (int i = 0; i < infiniteInstanceList->getNumPrimitives(); i++) {
    infiniteInstanceList->intersectPrimitive(r, i, state);
  }
  state->current = 0;
  intAccel->intersect(r, state);
}

Color *Scene::traceShadow(Ray *r, IntersectionState *state) {
  state->numShadowRays++;
  trace(r, state);
  return state->hit() ? Color::WHITE : Color::BLACK;
}

void Scene::traceBake(Ray *r, IntersectionState *state) {
  state->current = bakingInstance;
  state->instance = 0;
  bakingAccel->intersect(r, state);
}

void Scene::createAreaLightInstances() {
  std::vector<Instance*> infiniteAreaLights = 0;
  std::vector<Instance*> areaLights = 0;
  // create an area light instance from each light source if possible
  for (LightSource[]::const_iterator l = lightServer->lights->begin(); l != lightServer->lights->
      end(); l++) {
    Instance *lightInstance = *l->createInstance();
    if (lightInstance != 0) {
      if (lightInstance->getBounds() == 0) {
        if (infiniteAreaLights == 0) {
          infiniteAreaLights = std::vector<Instance*>();
        }
        infiniteAreaLights.push_back(lightInstance);
      } else {
        if (areaLights == 0) {
          areaLights = std::vector<Instance*>();
        }
        areaLights.push_back(lightInstance);
      }
    }
  }
  // add area light sources to the list of instances if they exist
  if (infiniteAreaLights != 0 && infiniteAreaLights.size() > 0) {
```

```
    infiniteInstanceList->addLightSourceInstances(
        infiniteAreaLights.toArray(new Instance[infiniteAreaLights.size()]));
  } else {
    infiniteInstanceList->clearLightSources();
  }
  if (areaLights != 0 && areaLights.size() > 0) {
    instanceList->addLightSourceInstances(areaLights.toArray(new Instance[areaLights.size()])
        );
  } else {
    instanceList->clearLightSources();
  }
  // TODO: this _could_ be done incrementally to avoid top-level rebuilds each frame
  rebuildAccel = true;
}

void Scene::removeAreaLightInstances() {
  infiniteInstanceList->clearLightSources();
  instanceList->clearLightSources();
}

void Scene::render(Options *options, ImageSampler *sampler, Display *display) {
  stats->reset();
  if (display == 0) {
    display = new FrameDisplay();
  }

  if (bakingInstance != 0) {
    UI::printDetailed(Module::SCENE, "Creating primitives for lightmapping ...");
    bakingPrimitives = bakingInstance->getBakingPrimitives();
    if (bakingPrimitives == 0) {
      UI::printError(Module::SCENE,"Lightmap baking is not supported for the given instance
          .");
      return;
    }
    int n = bakingPrimitives->getNumPrimitives();
    UI::printInfo(Module::SCENE, "Building acceleration structure for lightmapping (%d num
        primitives) ...", n);
    bakingAccel = AccelerationStructureFactory::create("auto", n, true);
    bakingAccel->build(bakingPrimitives);
  } else {
    delete bakingPrimitives;
    delete bakingAccel;
  }
  bakingViewDependent = options->getBoolean("baking.viewdep", bakingViewDependent);
```

```
if ((bakingInstance != 0 && bakingViewDependent && camera == 0) || (bakingInstance ==
    0 && camera == 0)) {
  UI::printError(Module::SCENE, "No camera found");
  return;
}

// read from options
threads = options->getInt("threads", 0);
lowPriority = options->getBoolean("threads.lowPriority", true);
imageWidth = options->getInt("resolutionX", 640);
imageHeight = options->getInt("resolutionY", 480);
// limit resolution to 16k
imageWidth = MathUtils::clamp(imageWidth, 1, 1 << 14);
imageHeight = MathUtils::clamp(imageHeight, 1, 1 << 14);

// prepare lights
createAreaLightInstances();

// get acceleration structure info
// count scene primitives
long long numPrimitives = 0;
for (int i = 0; i < instanceList->getNumPrimitives(); i++) {
  numPrimitives += instanceList->getNumPrimitives(i);
}
UI::printInfo(Module::SCENE, "Scene stats:");
UI::printInfo(Module::SCENE, " * Infinite instances: %d", infiniteInstanceList->
    getNumPrimitives());
UI::printInfo(Module::SCENE, " * Instances: %d", instanceList->getNumPrimitives());
UI::printInfo(Module::SCENE, " * Primitives: %d", numPrimitives);
std::string accelName = options->getString("accel", 0);
if (accelName != 0) {
  rebuildAccel = rebuildAccel || !acceltype.equals(accelName);
  acceltype = accelName;
}
UI::printInfo(Module::SCENE, " * Instance accel: %s", acceltype);
if (rebuildAccel) {
  intAccel = AccelerationStructureFactory::create(acceltype, instanceList->
      getNumPrimitives(), false);
  intAccel->build(instanceList);
  rebuildAccel = false;
}
UI::printInfo(Module::SCENE, " * Scene bounds: %s", getBounds());
UI::printInfo(Module::SCENE, " * Scene center: %s", getBounds()->getCenter());
UI::printInfo(Module::SCENE, " * Scene diameter: %.2f", getBounds()->getExtents()->
    length());
```

```
      UI::printInfo(Module::SCENE, " * Lightmap bake: %s", bakingInstance != 0 ? (
          bakingViewDependent ? "view" : "ortho") : "off");
      if (sampler == 0) {
        return;
      }
      if (!lightServer->build(options)) {
        return;
      }
      UI::printInfo(Module::SCENE, "Rendering ...");
      stats->setResolution(imageWidth, imageHeight);
      sampler->prepare(options, this, imageWidth, imageHeight);
      sampler->render(display);
      stats->displayStats();
      lightServer->showStats();
      removeAreaLightInstances();
      delete bakingPrimitives;
      delete bakingAccel;
      UI::printInfo(Module::SCENE, "Done.");
    }

    bool Scene::calculatePhotons(PhotonStore *map, std::string type,
        int seed, Options *options) {
      return lightServer->calculatePhotons(map, type, seed, options);
    }
  }
```

Listing A.5: Ray.h

```
#pragma once
#include "Vector3.h"
#include "Point3.h"
#include "Matrix4.h"
#include <cmath>

namespace hpr.core {

  using hpr::math::Matrix4;
  using hpr::math::Point3;
  using hpr::math::Vector3;

  // Represents a ray as a oriented half line segment.
  class Ray  {
    private:
    float tMin;
    float tMax;
    static const float EPSILON; // 0.01f;
```

```
    Ray();

    public:
    float ox, oy, oz;
    float dx, dy, dz;
    Ray(float ox, float oy, float oz, float dx, float dy, float dz); // Creates a new ray that points
        from the given origin to the given direction.
    Ray(Point3 *o, Vector3 *d); // Creates a new ray that points from the given origin to the given
        direction.
    Ray(Point3 *a, Point3 *b); // Creates a new ray that points from point a to point b.
    Ray *transform(Matrix4 *m); // Create a new ray by transforming the supplied one by the
        given matrix.
    void normalize(); // Normalize the direction component of the ray.
    float getMin(); // Gets the minimum distance along the ray - usually 0.
    float getMax(); // Gets the maximum distance along the ray. May be infinite.
    Vector3 *getDirection(); // Creates a vector to represent the direction of the ray.
    bool isInside(float t); // Checks to see if the specified distance falls within the valid range on
        this ray.
    Point3 *getPoint(Point3 *dest); // Gets the end point of the ray.
    float dot(Vector3 *v); // Computes the dot product of an arbitrary vector with the direction of
        the ray.
    float dot(float vx, float vy, float vz); // Computes the dot product of an arbitrary vector with
        the direction of the ray.
    void setMax(float t); // Updates the maximum to the specified distance if and only if the new
        distance is smaller than the current one.
  };
}
```

Listing A.6: Ray.cpp

```
#include "Ray.h"

using hpr::math::Matrix4;
using hpr::math::Point3;
using hpr::math::Vector3;

Ray::Ray() {
}

Ray::Ray(float ox, float oy, float oz, float dx, float dy, float dz) {
  this->ox = ox;
  this->oy = oy;
  this->oz = oz;
  this->dx = dx;
  this->dy = dy;
  this->dz = dz;
```

```
  float _in = 1.0f / static_cast<float> (sqrt(dx * dx + dy * dy + dz * dz));
  this->dx *= _in;
  this->dy *= _in;
  this->dz *= _in;
  tMin = EPSILON;
  tMax = 1.0f / 0.0f;
}

Ray::Ray(Point3 *o, Vector3 *d) {
  ox = o->x;
  oy = o->y;
  oz = o->z;
  dx = d->x;
  dy = d->y;
  dz = d->z;
  float _in = 1.0f / static_cast<float> (sqrt(dx * dx + dy * dy + dz * dz));
  dx *= _in;
  dy *= _in;
  dz *= _in;
  tMin = EPSILON;
  tMax = 1.0f / 0.0f;
}

Ray::Ray(Point3 *a, Point3 *b) {
  ox = a->x;
  oy = a->y;
  oz = a->z;
  dx = b->x - ox;
  dy = b->y - oy;
  dz = b->z - oz;
  tMin = EPSILON;
  float n = static_cast<float> (sqrt(dx * dx + dy * dy + dz * dz));
  float _in = 1.0f / n;
  dx *= _in;
  dy *= _in;
  dz *= _in;
  tMax = n - EPSILON;
}

Ray *Ray::transform(Matrix4 *m) {
  if (m == 0) {
    return this;
  }
  Ray *r = new Ray();
  r->ox = m->transformPX(ox, oy, oz);
```

```
  r->oy = m->transformPY(ox, oy, oz);
  r->oz = m->transformPZ(ox, oy, oz);
  r->dx = m->transformVX(dx, dy, dz);
  r->dy = m->transformVY(dx, dy, dz);
  r->dz = m->transformVZ(dx, dy, dz);
  r->tMin = tMin;
  r->tMax = tMax;
  return r;
}

void Ray::normalize() {
  float _in = 1.0f / static_cast<float> (sqrt(dx * dx + dy * dy + dz * dz));
  dx *= _in;
  dy *= _in;
  dz *= _in;
}

float Ray::getMin() {
  return tMin;
}

float Ray::getMax() {
  return tMax;
}

Vector3 *Ray::getDirection() {
  return new Vector3(dx, dy, dz);
}

bool Ray::isInside(float t) {
  return (tMin < t) && (t < tMax);
}

Point3 *Ray::getPoint(Point3 *dest) {
  dest->x = ox + (tMax * dx);
  dest->y = oy + (tMax * dy);
  dest->z = oz + (tMax * dz);
  return dest;
}

float Ray::dot(Vector3 *v) {
  return dx * v->x + dy * v->y + dz * v->z;
}

float Ray::dot(float vx, float vy, float vz) {
```

```
  return dx * vx + dy * vy + dz * vz;
}

void Ray::setMax(float t) {
  tMax = t;
}
}
```

Listing A.7: Cameara.h

```
#pragma once
#include "RenderObject.h"
#include "CameraLens.h"
#include "MovingMatrix4.h"
#include "HprAPI.h"
#include "UI.h"
#include "Point3.h"
#include "Matrix4.h"
#include <cmath>

namespace hpr.core {

  using hpr::HprAPI;
  using hpr::math::Matrix4;
  using hpr::math::MovingMatrix4;
  using hpr::math::Point3;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  // Represents a camera to the renderer. It handles the mapping of camera space to world space.
  class Camera : RenderObject {
    private:
    const CameraLens *lens;
    float shutterOpen;
    float shutterClose;
    MovingMatrix4 *c2w;
    MovingMatrix4 *w2c;

    public:
    Camera(CameraLens *lens);
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual float getTime(float time); // Computes actual time from a time sample in the interval
        [0,1).
    virtual Ray *getRay(float x, float y, int imageWidth, int imageHeight, double lensX, double
        lensY, float time); // Generate a ray passing though the specified point on the image plane
        .
```

```
    virtual Ray *getRay(Point3 *p, float time); // Generate a ray from the origin of camera space
        toward the specified point.
    virtual Matrix4 *getCameraToWorld(float time); // Returns a transformation matrix mapping
        camera space to world space.
    virtual Matrix4 *getWorldToCamera(float time); // Returns a transformation matrix mapping
        world space to camera space.
  };
}
```

Listing A.8: Cameara.cpp

```
#include "Camera.h"

using hpr::HprAPI;
using hpr::math::Matrix4;
using hpr::math::MovingMatrix4;
using hpr::math::Point3;
using hpr::system::UI;
using hpr::system::UI::Module;

Camera::Camera(CameraLens *lens) {
  this->lens = lens;
  c2w = new MovingMatrix4(0);
  w2c = new MovingMatrix4(0);
  shutterOpen = shutterClose = 0;
}

bool Camera::update(ParameterList *pl, HprAPI *api) {
  shutterOpen = pl->getFloat("shutter.open", shutterOpen);
  shutterClose = pl->getFloat("shutter.close", shutterClose);
  c2w = pl->getMovingMatrix("transform", c2w);
  w2c = c2w->inverse();
  if (w2c == 0) {
    UI::printWarning(Module::CAM, "Unable to compute camera's inverse transform");
    return false;
  }
  return lens->update(pl, api);
}

float Camera::getTime(float time) {
  if (shutterOpen >= shutterClose) {
    return shutterOpen;
  }
  // warp the time sample by a tent filter
  if (time < 0.5) {
    time = -1 + static_cast<float> (sqrt(2 * time));
```

```
  } else {
    time = 1 − static_cast<float> (sqrt(2 − 2 ∗ time));
  }
  time = 0.5f ∗ (time + 1);
  return (1 − time) ∗ shutterOpen + time ∗ shutterClose;
}

Ray ∗Camera::getRay(float x, float y, int imageWidth, int imageHeight, double lensX, double lensY,
      float time) {
  Ray ∗r = lens−>getRay(x, y, imageWidth, imageHeight, lensX, lensY, time);
  if (r != 0) {
    // transform from camera space to world space
    r = r−>transform(c2w−>sample(time));
    // renormalize to account for scale factors embedded in the transform
    r−>normalize();
  }
  return r;
}

Ray ∗Camera::getRay(Point3 ∗p, float time) {
  return new Ray(c2w == 0 ? new Point3(0, 0, 0) : c2w−>sample(time).transformP(new Point3(0,
       0, 0)), p);
}

Matrix4 ∗Camera::getCameraToWorld(float time) {
  return c2w == 0 ? Matrix4::IDENTITY : c2w−>sample(time);
}

Matrix4 ∗Camera::getWorldToCamera(float time) {
  return w2c == 0 ? Matrix4::IDENTITY : w2c−>sample(time);
}
}
```

Listing A.9: Display.h

```
#pragma once

namespace hpr.core {

  using hpr::image::Color;

  // Represents an image output device.
  class Display {
    virtual void imageBegin(int, int, int) = 0; // Called before an image is rendered to indicate how
         large the rendered image will be.
    virtual void imagePrepare(int, int, int, int, int) = 0; // Prepare the specified area to be
```

```
        rendered.
    virtual void imageUpdate(int, int, int, int, Color[] , float[] ) = 0; // Update the current image
        with a bucket of data.
    virtual void imageFill(int, int, int, int, Color*, float) = 0; // Update the current image with a
        region of flat color.
    virtual void imageEnd() = 0; // This call is made after the image has been rendered.
  };
}
```

Listing A.10: Display.cpp

```
#include "Display.h"

using hpr::image::Color;
}
```

Listing A.11: Geometry.h

```
#pragma once
#include "RenderObject.h"
#include "Tesselatable.h"
#include "PrimitiveList.h"
#include "AccelerationStructure.h"
#include "HprAPI.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include "UI.h"
#include "NullAccelerator.h"
#include <string>

namespace hpr.core {

  using hpr::HprAPI;
  using hpr::core::accel::NullAccelerator;
  using hpr::math::BoundingBox;
  using hpr::math::Matrix4;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  // Represents a geometric object in its native object space.
  class Geometry : RenderObject {
    private:
    Tesselatable *tesselatable;
    PrimitiveList *primitives;
    AccelerationStructure *accel;
    int builtAccel;
    int builtTess;
```

```
    std::string acceltype;
    void tesselate();
    void build();

    public:
    Geometry(Tesselatable *tesselatable); // Create a geometry from the specified tesselatable
        object.
    Geometry(PrimitiveList *primitives); // Create a geometry from the specified primitive
        aggregate.
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual int getNumPrimitives();
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual void intersect(Ray *r, IntersectionState *state);
    virtual void prepareShadingState(ShadingState *state);
    virtual PrimitiveList *getBakingPrimitives();
    virtual PrimitiveList *getPrimitiveList();
  };
}
```

Listing A.12: Geometry.cpp

```
#include "Geometry.h"

using hpr::HprAPI;
using hpr::core::accel::NullAccelerator;
using hpr::math::BoundingBox;
using hpr::math::Matrix4;
using hpr::system::UI;
using hpr::system::UI::Module;

Geometry::Geometry(Tesselatable *tesselatable) {
  this->tesselatable = tesselatable;
  delete primitives;
  delete accel;
  builtAccel = 0;
  builtTess = 0;
  acceltype = "";
}

Geometry::Geometry(PrimitiveList *primitives) {
  delete tesselatable;
  this->primitives = primitives;
  delete accel;
  builtAccel = 0;
  builtTess = 1; // already tesselated
}
```

```
bool Geometry::update(ParameterList *pl, HprAPI *api) {
  acceltype = pl->getString("accel", acceltype);
  // clear up old tesselation if it exists
  if (tesselatable != 0) {
    delete primitives;
    builtTess = 0;
  }
  // clear acceleration structure so it will be rebuilt
  delete accel;
  builtAccel = 0;
  if (tesselatable != 0) {
    return tesselatable->update(pl, api);
  }
  // update primitives
  return primitives->update(pl, api);
}

int Geometry::getNumPrimitives() {
  return primitives == 0 ? 0 : primitives->getNumPrimitives();
}

BoundingBox *Geometry::getWorldBounds(Matrix4 *o2w) {
  if (primitives == 0) {
    BoundingBox *b = tesselatable->getWorldBounds(o2w);
    if (b != 0) {
      return b;
    }
    if (builtTess == 0) {
      tesselate();
    }
    if (primitives == 0) {
      return 0; // failed tesselation, return infinite bounding box
    }
  }
  return primitives->getWorldBounds(o2w);
}

void Geometry::intersect(Ray *r, IntersectionState *state) {
  if (builtTess == 0) {
    tesselate();
  }
  if (builtAccel == 0) {
    build();
  }
```

```
  accel->intersect(r, state);
}

void Geometry::tesselate() {
  // double check flag
  if (builtTess != 0) {
    return;
  }
  if (tesselatable != 0 && primitives == 0) {
    UI::printInfo(Module::GEOM, "Tesselating geometry ...");
    primitives = tesselatable->tesselate();
    if (primitives == 0) {
      UI::printError(Module::GEOM, "Tesselation failed - geometry will be discarded");
    } else {
      UI::printDetailed(Module::GEOM, "Tesselation produced %d primitives", primitives->
          getNumPrimitives());
    }
  }
  builtTess = 1;
}

void Geometry::build() {
  // double check flag
  if (builtAccel != 0) {
    return;
  }
  if (primitives != 0) {
    int n = primitives->getNumPrimitives();
    if (n >= 1000) {
      UI::printInfo(Module::GEOM, "Building acceleration structure for %d primitives ...", n);
    }
    accel = AccelerationStructureFactory::create(acceltype, n, true);
    accel->build(primitives);
  } else {
    // create an empty accelerator to avoid having to check for null pointers in the intersect
        method
    accel = new NullAccelerator();
  }
  builtAccel = 1;
}

void Geometry::prepareShadingState(ShadingState *state) {
  primitives->prepareShadingState(state);
}
```

```
PrimitiveList *Geometry::getBakingPrimitives() {
  if (builtTess == 0) {
    tesselate();
  }
  if (primitives == 0) {
    return 0;
  }
  return primitives->getBakingPrimitives();
}

PrimitiveList *Geometry::getPrimitiveList() {
  return primitives;
}
}
```

Listing A.13: RenderObject.h

```
#pragma once

namespace hpr.core {

  using hpr::HprAPI;

  // This is the base interface for all public rendering object interfaces.
  class RenderObject {
    virtual bool update(ParameterList*, HprAPI*) = 0;
  };
}
```

Listing A.14: RenderObject.cpp

```
#include "RenderObject.h"

using hpr::HprAPI;
}
```

Listing A.15: SimpleRenderer.h

```
#pragma once
#include "Scene.h"
#include "Options.h"
#include "Timer.h"
#include "UI.h"
#include "IntersectionState.h"
#include "Color.h"
#include "ShadingState.h"
#include <cmath>
```

```
#include "mpi.h"
#include <omp.h>
#include <brook.h>

namespace hpr.core.renderer {

  using hpr::core::Display;
  using hpr::core::ImageSampler;
  using hpr::core::IntersectionState;
  using hpr::core::Options;
  using hpr::core::Scene;
  using hpr::core::ShadingState;
  using hpr::image::Color;
  using hpr::system::Timer;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  class SimpleRenderer : ImageSampler {
    private:
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;
    Scene *scene;
    Display *display;
    int imageWidth, imageHeight;
    int numBucketsX, numBucketsY;
    int bucketCounter, numBuckets;
    class BucketThread : Thread {
      private:
      const IntersectionState *istate = new IntersectionState();
      public:
      virtual void run();
      virtual void updateStats();
    };

    public:
    virtual bool prepare(Options *options, Scene *scene, int w, int h);
    virtual void render(Display *display);
    virtual void renderBucket(int bx, int by, IntersectionState *istate);
  };
}
```

Listing A.16: SimpleRenderer.cpp

```
#include "SimpleRenderer.h"
```

```
using hpr::core::Display;
using hpr::core::ImageSampler;
using hpr::core::IntersectionState;
using hpr::core::Options;
using hpr::core::Scene;
using hpr::core::ShadingState;
using hpr::image::Color;
using hpr::system::Timer;
using hpr::system::UI;
using hpr::system::UI::Module;

bool SimpleRenderer::prepare(Options *options, Scene *scene, int w, int h) {
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  this->scene = scene;
  imageWidth = w;
  imageHeight = h;
  numBucketsX = (imageWidth + 31) >>> 5;
  numBucketsY = (imageHeight + 31) >>> 5;
  numBuckets = numBucketsX * numBucketsY;
  return true;
}

void SimpleRenderer::render(Display *display) {
  this->display = display;
  display->imageBegin(imageWidth, imageHeight, 32);
  // set members variables
  bucketCounter = 0;
  // start task
  Timer *timer = new Timer();
  timer->start();
  BucketThread renderThreads[scene->getThreads()];
  for (int i = 0; i < sizeof(renderThreads) / sizeof(renderThreads[0]); i++) {
    renderThreads[i] = new BucketThread();
    renderThreads[i]->start();
  }
  for (int i = 0; i < sizeof(renderThreads) / sizeof(renderThreads[0]); i++) {
    try {
      renderThreads[i]->join();
    } catch (InterruptedException *e) {
      UI::printError(Module::BCKT,
          "Bucket processing thread %d of %d was interrupted", i + 1,
```

```
            sizeof(renderThreads) / sizeof(renderThreads[0]));
      } finally {
        renderThreads[i]->updateStats();
      }
    }
    timer->end();
    UI::printInfo(Module::BCKT, "Render time: %s", timer);
    display->imageEnd();
}

void SimpleRenderer::BucketThread::run() {
    while (true) {
      int bx, by;
      #pragma omp parallel default(shared) private(iam, np)
      {
        if (bucketCounter >= numBuckets) {
          return;
        }
        by = bucketCounter / numBucketsX;
        bx = bucketCounter % numBucketsX;
        bucketCounter++;
      }
      MPI_Finalize();
      renderBucket(bx, by, istate);
    }
}

void SimpleRenderer::BucketThread::updateStats() {
    scene::accumulateStats( istate);
}

void SimpleRenderer::renderBucket(int bx, int by, IntersectionState *istate) {
    // pixel sized extents
    int x0 = bx * 32;
    int y0 = by * 32;
    int bw = __min(32, imageWidth - x0);
    int bh = __min(32, imageHeight - y0);

    Color bucketRGB[bw * bh];
    float bucketAlpha[bw * bh];

    for (int y = 0, i = 0; y < bh; y++) {
      for (int x = 0; x < bw; x++, i++) {
        ShadingState *state = scene->getRadiance(istate, x0 + x, imageHeight - 1 - (y0 + y), 0.0,
            0.0, 0.0, 0, 0, 0);
```

```
      bucketRGB[i] = (state != 0) ? state->getResult() : Color::BLACK;
      bucketAlpha[i] = (state != 0) ? 1 : 0;
    }
  }
  // update pixels
  display->imageUpdate(x0, y0, bw, bh, bucketRGB, bucketAlpha);
}
}
```

Listing A.17: MultipassRenderer.h

```
#pragma once
#include "Scene.h"
#include "Options.h"
#include "MathUtils.h"
#include "BucketOrderFactory.h"
#include "UI.h"
#include "Timer.h"
#include "IntersectionState.h"
#include "ShadingCache.h"
#include "Color.h"
#include "QMC.h"
#include "ShadingState.h"
#include <string>
#include <cmath>

namespace hpr.core.renderer {

  using hpr::core::BucketOrder;
  using hpr::core::Display;
  using hpr::core::ImageSampler;
  using hpr::core::IntersectionState;
  using hpr::core::Options;
  using hpr::core::Scene;
  using hpr::core::ShadingCache;
  using hpr::core::ShadingState;
  using hpr::core::bucket::BucketOrderFactory;
  using hpr::image::Color;
  using hpr::math::MathUtils;
  using hpr::math::QMC;
  using hpr::system::Timer;
  using hpr::system::UI;
  using hpr::system::UI::Module;

  class MultipassRenderer : ImageSampler  {
    private:
```

```
    Scene *scene;
    Display *display;
    int imageWidth;
    int imageHeight;
    std::string bucketOrderName;
    BucketOrder *bucketOrder;
    int bucketSize;
    int bucketCounter;
    int *bucketCoords;
    int numSamples;
    float invNumSamples;
    bool shadingCache;
    void renderBucket(Display *display, int bx, int by, int threadID, IntersectionState *istate,
        ShadingCache *cache);
    class BucketThread : Thread {
      private:
      const int threadID;
      const IntersectionState *istate;
      const ShadingCache *cache;

      public:
      BucketThread(int threadID);
      virtual void run();
      virtual void updateStats();
    };

    public:
    MultipassRenderer();
    virtual bool prepare(Options *options, Scene *scene, int w, int h);
    virtual void render(Display *display);
    static float warpTent(float x);
    static double warpCubic(double x);
    static double qpow(double x);
    static double distb1(double x);
  };
}
```

Listing A.18: MultipassRenderer.cpp

```
#include "MultipassRenderer.h"

using hpr::core::BucketOrder;
using hpr::core::Display;
using hpr::core::ImageSampler;
using hpr::core::IntersectionState;
using hpr::core::Options;
```

```
using hpr::core::Scene;
using hpr::core::ShadingCache;
using hpr::core::ShadingState;
using hpr::core::bucket::BucketOrderFactory;
using hpr::image::Color;
using hpr::math::MathUtils;
using hpr::math::QMC;
using hpr::system::Timer;
using hpr::system::UI;
using hpr::system::UI::Module;

MultipassRenderer::MultipassRenderer() {
  bucketSize = 32;
  bucketOrderName = "hilbert";
  numSamples = 16;
  shadingCache = false;
}

bool MultipassRenderer::prepare(Options *options, Scene *scene, int w, int h) {
  this->scene = scene;
  imageWidth = w;
  imageHeight = h;

  // fetch options
  bucketSize = options->getInt("bucket.size", bucketSize);
  bucketOrderName = options->getString("bucket.order", bucketOrderName);
  numSamples = options->getInt("aa.samples", numSamples);
  shadingCache = options->getBoolean("aa.cache", shadingCache);

  // limit bucket size and compute number of buckets in each direction
  bucketSize = MathUtils::clamp(bucketSize, 16, 512);
  int numBucketsX = (imageWidth + bucketSize - 1) / bucketSize;
  int numBucketsY = (imageHeight + bucketSize - 1) / bucketSize;
  bucketOrder = BucketOrderFactory::create(bucketOrderName);
  bucketCoords = bucketOrder->getBucketSequence(numBucketsX, numBucketsY);
  // validate AA options
  numSamples = __max(1, numSamples);
  invNumSamples = 1.0f / numSamples;
  // prepare QMC sampling
  UI::printInfo(Module::BCKT, "Multipass renderer settings:");
  UI::printInfo(Module::BCKT, " * Resolution: %dx%d", imageWidth, imageHeight);
  UI::printInfo(Module::BCKT, " * Bucket size: %d", bucketSize);
  UI::printInfo(Module::BCKT, " * Number of buckets: %dx%d", numBucketsX, numBucketsY);
  UI::printInfo(Module::BCKT, " * Samples / pixel: %d", numSamples);
  UI::printInfo(Module::BCKT, " * Shading cache: %s", shadingCache ? "enabled" : "disabled");
```

```
  return true;
}

void MultipassRenderer::render(Display *display) {
  this->display = display;
  display->imageBegin(imageWidth, imageHeight, bucketSize);
  // set members variables
  bucketCounter = 0;
  // start task
  Timer *timer = new Timer();
  timer->start();
  UI::taskStart("Rendering", 0, sizeof(bucketCoords) / sizeof(bucketCoords[0]));
  BucketThread renderThreads[scene->getThreads()];
  for (int i = 0; i < sizeof(renderThreads) / sizeof(renderThreads[0]); i++) {
    renderThreads[i] = new BucketThread(i);
    renderThreads[i]->setPriority(scene->getThreadPriority());
    renderThreads[i]->start();
  }
  for (int i = 0; i < sizeof(renderThreads) / sizeof(renderThreads[0]); i++) {
    try {
      renderThreads[i]->join();
    } catch (InterruptedException *e) {
      UI::printError(Module::BCKT, "Bucket processing thread %d of %d was interrupted", i + 1,
          sizeof(renderThreads) / sizeof(renderThreads[0]));
    } finally {
      renderThreads[i]->updateStats();
    }
  }
  UI::taskStop();
  timer->end();

  UI::printInfo(Module::BCKT, "Render time: %s", timer);
  display->imageEnd();
}

MultipassRenderer::BucketThread::BucketThread(int threadID) {
  this->threadID = threadID;
  istate = new IntersectionState();
  cache = shadingCache ? new ShadingCache() : 0;
}

void MultipassRenderer::BucketThread::run() {
  while (true) {
    int bx, by;
    THREAD_SYNCRONIZED(MultipassRenderer::this) {
```

```
        if (bucketCounter >= bucketCoords::length) {
          return;
        }
        UI::taskUpdate(bucketCounter);
        bx = bucketCoords[bucketCounter + 0];
        by = bucketCoords[bucketCounter + 1];
        bucketCounter += 2;
      }
      renderBucket(display, bx, by, threadID, istate, cache);
    }
}

void MultipassRenderer::BucketThread::updateStats() {
    scene::accumulateStats( istate);
    if (shadingCache) {
      scene::accumulateStats( cache);
    }
}

void MultipassRenderer::renderBucket(Display *display, int bx, int by, int threadID,
       IntersectionState *istate, ShadingCache *cache) {
    // pixel sized extents
    int x0 = bx * bucketSize;
    int y0 = by * bucketSize;
    int bw = __min(bucketSize, imageWidth - x0);
    int bh = __min(bucketSize, imageHeight - y0);

    // prepare bucket
    display->imagePrepare(x0, y0, bw, bh, threadID);

    Color bucketRGB[bw * bh];
    float bucketAlpha[bw * bh];

    for (int y = 0, i = 0, cy = imageHeight - 1 - y0; y < bh; y++, cy--) {
      for (int x = 0, cx = x0; x < bw; x++, i++, cx++) {
        // sample pixel
        Color *c = Color::black();
        float a = 0;
        int instance = ((cx & ((1 << QMC::MAX_SIGMA_ORDER) - 1))
            << QMC::MAX_SIGMA_ORDER) + QMC::sigma(cy & ((1
            << QMC::MAX_SIGMA_ORDER) - 1), QMC::MAX_SIGMA_ORDER);
        double jitterX = QMC::halton(0, instance);
        double jitterY = QMC::halton(1, instance);
        double jitterT = QMC::halton(2, instance);
        double jitterU = QMC::halton(3, instance);
```

```
      double jitterV = QMC::halton(4, instance);
      for (int s = 0; s < numSamples; s++) {
        float rx = cx + 0.5f + static_cast<float> (warpCubic(QMC::mod1(jitterX + s *
            invNumSamples)));
        float ry = cy + 0.5f + static_cast<float> (warpCubic(QMC::mod1(jitterY + QMC::halton
            (0, s))));
        double time = QMC::mod1(jitterT + QMC::halton(1, s));
        double lensU = QMC::mod1(jitterU + QMC::halton(2, s));
        double lensV = QMC::mod1(jitterV + QMC::halton(3, s));
        ShadingState *state = scene->getRadiance(istate, rx, ry, lensU, lensV, time, instance + s,
            5, cache);
        if (state != 0) {
          c->add(state->getResult());
          a++;
        }
      }
      bucketRGB[i] = c->mul(invNumSamples);
      bucketAlpha[i] = a * invNumSamples;
      if (cache != 0) {
        cache->reset();
      }
    }
  }
  // update pixels
  display->imageUpdate(x0, y0, bw, bh, bucketRGB, bucketAlpha);
}

float MultipassRenderer::warpTent(float x) {
  if (x < 0.5f) {
    return -1 + static_cast<float> (sqrt(2 * x));
  } else {
    return +1 - static_cast<float> (sqrt(2 - 2 * x));
  }
}

double MultipassRenderer::warpCubic(double x) {
  if (x < (1.0 / 24)) {
    return qpow(24 * x) - 2;
  }
  if (x < 0.5f) {
    return distb1((24.0 / 11.0) * (x - (1.0 / 24.0))) - 1;
  }
  if (x < (23.0f / 24)) {
    return 1 - distb1((24.0 / 11.0) * ((23.0 / 24.0) - x));
  }
```

```
  return 2 − qpow(24 ∗ (1 − x));
}

double MultipassRenderer::qpow(double x) {
  return sqrt(sqrt(x));
}

double MultipassRenderer::distb1(double x) {
  double u = x;
  for (int i = 0; i < 5; i++) {
    u = (11 ∗ x + u ∗ u ∗ (6 + u ∗ (8 − 9 ∗ u))) / (4 + 12 ∗ u ∗ (1 + u ∗ (1 − u)));
  }
  return u;
}
}
```

Listing A.19: Box.h

```
#pragma once
#include "HprAPI.h"
#include "ParameterList.h"
#include "BoundingBox.h"
#include "ShadingState.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
#include "Ray.h"
#include "IntersectionState.h"
#include "Matrix4.h"

namespace hpr.core.primitive {

  using hpr::HprAPI;
  using hpr::core::IntersectionState;
  using hpr::core::ParameterList;
  using hpr::core::PrimitiveList;
  using hpr::core::Ray;
  using hpr::core::ShadingState;
  using hpr::core::ParameterList::FloatParameter;
  using hpr::math::BoundingBox;
  using hpr::math::Matrix4;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Vector3;

  class Box : PrimitiveList {
    private:
    float minX, minY, minZ;
```

```
    float maxX, maxY, maxZ;

    public:
    Box();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual void prepareShadingState(ShadingState *state);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual int getNumPrimitives();
    virtual float getPrimitiveBound(int primID, int i);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual PrimitiveList *getBakingPrimitives();
  };
}
```

Listing A.20: Box.cpp

```
#include "Box.h"

using hpr::HprAPI;
using hpr::core::IntersectionState;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::core::ParameterList::FloatParameter;
using hpr::math::BoundingBox;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Vector3;

Box::Box() {
  minX = minY = minZ = -1;
  maxX = maxY = maxZ = +1;
}

bool Box::update(ParameterList *pl, HprAPI *api) {
  FloatParameter *pts = pl->getPointArray("points");
  if (pts != 0) {
    BoundingBox *bounds = new BoundingBox();
    for (int i = 0; i < pts->data->length; i += 3) {
      bounds->include(pts->data[i], pts->data[i + 1], pts->data[i + 2]);
    }
    // cube extents
    minX = bounds->getMinimum()->x;
    minY = bounds->getMinimum()->y;
    minZ = bounds->getMinimum()->z;
```

```
    maxX = bounds->getMaximum()->x;
    maxY = bounds->getMaximum()->y;
    maxZ = bounds->getMaximum()->z;
  }
  return true;
}

void Box::prepareShadingState(ShadingState *state) {
  state->init();
  state->getRay()->getPoint(state->getPoint());
  int n = state->getPrimitiveID();
  switch (n) {
    case 0:
      state->getNormal()->set(new Vector3(1, 0, 0));
      break;
    case 1:
      state->getNormal()->set(new Vector3(-1, 0, 0));
      break;
    case 2:
      state->getNormal()->set(new Vector3(0, 1, 0));
      break;
    case 3:
      state->getNormal()->set(new Vector3(0, -1, 0));
      break;
    case 4:
      state->getNormal()->set(new Vector3(0, 0, 1));
      break;
    case 5:
      state->getNormal()->set(new Vector3(0, 0, -1));
      break;
    default:
      state->getNormal()->set(new Vector3(0, 0, 0));
      break;
  }
  state->getGeoNormal()->set(state->getNormal());
  state->setBasis(OrthoNormalBasis::makeFromW(state->getNormal()));
  state->setShader(state->getInstance()->getShader(0));
  state->setModifier(state->getInstance()->getModifier(0));
}

void Box::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  float intervalMin = -1.0f / 0.0f;
  float intervalMax = 1.0f / 0.0f;
  float orgX = r->ox;
  float invDirX = 1 / r->dx;
```

```
float t1, t2;
t1 = (minX − orgX) ∗ invDirX;
t2 = (maxX − orgX) ∗ invDirX;
int sideIn = −1, sideOut = −1;
if (invDirX > 0) {
  if (t1 > intervalMin) {
    intervalMin = t1;
    sideIn = 0;
  }
  if (t2 < intervalMax) {
    intervalMax = t2;
    sideOut = 1;
  }
} else {
  if (t2 > intervalMin) {
    intervalMin = t2;
    sideIn = 1;
  }
  if (t1 < intervalMax) {
    intervalMax = t1;
    sideOut = 0;
  }
}
if (intervalMin > intervalMax) {
  return;
}
float orgY = r−>oy;
float invDirY = 1 / r−>dy;
t1 = (minY − orgY) ∗ invDirY;
t2 = (maxY − orgY) ∗ invDirY;
if (invDirY > 0) {
  if (t1 > intervalMin) {
    intervalMin = t1;
    sideIn = 2;
  }
  if (t2 < intervalMax) {
    intervalMax = t2;
    sideOut = 3;
  }
} else {
  if (t2 > intervalMin) {
    intervalMin = t2;
    sideIn = 3;
  }
  if (t1 < intervalMax) {
```

```
      intervalMax = t1;
      sideOut = 2;
    }
  }
  if (intervalMin > intervalMax) {
    return;
  }
  float orgZ = r->oz;
  float invDirZ = 1 / r->dz;
  t1 = (minZ - orgZ) * invDirZ;
  t2 = (maxZ - orgZ) * invDirZ;
  if (invDirZ > 0) {
    if (t1 > intervalMin) {
      intervalMin = t1;
      sideIn = 4;
    }
    if (t2 < intervalMax) {
      intervalMax = t2;
      sideOut = 5;
    }
  } else {
    if (t2 > intervalMin) {
      intervalMin = t2;
      sideIn = 5;
    }
    if (t1 < intervalMax) {
      intervalMax = t1;
      sideOut = 4;
    }
  }
  if (intervalMin > intervalMax) {
    return;
  }
  if (r->isInside(intervalMin)) {
    r->setMax(intervalMin);
    state->setIntersection(sideIn);
  } else if (r->isInside(intervalMax)) {
    r->setMax(intervalMax);
    state->setIntersection(sideOut);
  }
}

int Box::getNumPrimitives() {
  return 1;
}
```

```
float Box::getPrimitiveBound(int primID, int i) {
  switch (i) {
    case 0:
      return minX;
    case 1:
      return maxX;
    case 2:
      return minY;
    case 3:
      return maxY;
    case 4:
      return minZ;
    case 5:
      return maxZ;
    default:
      return 0;
  }
}

BoundingBox *Box::getWorldBounds(Matrix4 *o2w) {
  BoundingBox *bounds = new BoundingBox(minX, minY, minZ);
  bounds->include(maxX, maxY, maxZ);
  if (o2w == 0) {
    return bounds;
  }
  return o2w->transform(bounds);
}

PrimitiveList *Box::getBakingPrimitives() {
  return 0;
}
}
```

Listing A.21: Cylinder.h

```
#pragma once
#include "HprAPI.h"
#include "ParameterList.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include "ShadingState.h"
#include "Instance.h"
#include "Point3.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
```

```
#include "Ray.h"
#include "IntersectionState.h"
#include "Solvers.h"
#include <cmath>

namespace hpr.core.primitive {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::IntersectionState;
  using hpr::core::ParameterList;
  using hpr::core::PrimitiveList;
  using hpr::core::Ray;
  using hpr::core::ShadingState;
  using hpr::math::BoundingBox;
  using hpr::math::Matrix4;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Point3;
  using hpr::math::Solvers;
  using hpr::math::Vector3;

  class Cylinder : PrimitiveList  {
    public:
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual float getPrimitiveBound(int primID, int i);
    virtual int getNumPrimitives();
    virtual void prepareShadingState(ShadingState *state);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual PrimitiveList *getBakingPrimitives();
  };
}
```

Listing A.22: Cylinder.cpp

```
#include "Cylinder.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::IntersectionState;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::math::BoundingBox;
using hpr::math::Matrix4;
```

```
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Solvers;
using hpr::math::Vector3;

bool Cylinder::update(ParameterList *pl, HprAPI *api) {
  return true;
}

BoundingBox *Cylinder::getWorldBounds(Matrix4 *o2w) {
  BoundingBox *bounds = new BoundingBox(1);
  if (o2w != 0) {
    bounds = o2w->transform(bounds);
  }
  return bounds;
}

float Cylinder::getPrimitiveBound(int primID, int i) {
  return (i & 1) == 0 ? -1 : 1;
}

int Cylinder::getNumPrimitives() {
  return 1;
}

void Cylinder::prepareShadingState(ShadingState *state) {
  state->init();
  state->getRay()->getPoint(state->getPoint());
  Instance *parent = state->getInstance();
  Point3 *localPoint = state->transformWorldToObject(state->getPoint());
  state->getNormal()->set(localPoint->x, localPoint->y, 0);
  state->getNormal()->normalize();

  float phi = static_cast<float> (atan2(state->getNormal()->y, state->getNormal()->x));
  if (phi < 0) {
    phi += 2 * 3.1415;
  }
  state->getUV()->x = phi / static_cast<float> (2 * 3.1415);
  state->getUV()->y = (localPoint->z + 1) * 0.5f;
  state->setShader(parent->getShader(0));
  state->setModifier(parent->getModifier(0));
  // into world space
  Vector3 *worldNormal = state->transformNormalObjectToWorld(state->getNormal());
  Vector3 *v = state->transformVectorObjectToWorld(new Vector3(0, 0, 1));
  state->getNormal()->set(worldNormal);
```

```
  state->getNormal()->normalize();
  state->getGeoNormal()->set(state->getNormal());
  // compute basis in world space
  state->setBasis(OrthoNormalBasis::makeFromWV(state->getNormal(), v));
}

void Cylinder::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  // intersect in local space
  float qa = r->dx * r->dx + r->dy * r->dy;
  float qb = 2 * ((r->dx * r->ox) + (r->dy * r->oy));
  float qc = ((r->ox * r->ox) + (r->oy * r->oy)) - 1;
  double *t = Solvers::solveQuadric(qa, qb, qc);
  if (t != 0) {
    // early rejection
    if (t[0] >= r->getMax() || t[1] <= r->getMin()) {
      return;
    }
    if (t[0] > r->getMin()) {
      float z = r->oz + static_cast<float> (t[0]) * r->dz;
      if (z >= -1 && z <= 1) {
        r->setMax(static_cast<float> (t[0]));
        state->setIntersection(0);
        return;
      }
    }
    if (t[1] < r->getMax()) {
      float z = r->oz + static_cast<float> (t[1]) * r->dz;
      if (z >= -1 && z <= 1) {
        r->setMax(static_cast<float> (t[1]));
        state->setIntersection(0);
      }
    }
  }
}

PrimitiveList *Cylinder::getBakingPrimitives() {
  return 0;
}
}
```

Listing A.23: Plane.h

```
#pragma once
#include "Point3.h"
#include "Vector3.h"
#include "HprAPI.h"
```

```
#include "ParameterList.h"
#include "ShadingState.h"
#include "Instance.h"
#include "OrthoNormalBasis.h"
#include "Ray.h"
#include "IntersectionState.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include <cmath>

namespace hpr.core.primitive {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::IntersectionState;
  using hpr::core::ParameterList;
  using hpr::core::PrimitiveList;
  using hpr::core::Ray;
  using hpr::core::ShadingState;
  using hpr::math::BoundingBox;
  using hpr::math::Matrix4;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Point3;
  using hpr::math::Vector3;

  class Plane : PrimitiveList {
    private:
    Point3 *center;
    Vector3 *normal;
    public:
    int k;
    private:
    float bnu, bnv, bnd;
    float cnu, cnv, cnd;

    public:
    Plane();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual void prepareShadingState(ShadingState *state);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual int getNumPrimitives();
    virtual float getPrimitiveBound(int primID, int i);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual PrimitiveList *getBakingPrimitives();
  };
```

```
}
```

Listing A.24: Plane.cpp

```
#include "Plane.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::IntersectionState;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::math::BoundingBox;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Vector3;

Plane::Plane() {
  center = new Point3(0, 0, 0);
  normal = new Vector3(0, 1, 0);
  k = 3;
  bnu = bnv = bnd = 0;
  cnu = cnv = cnd = 0;
}

bool Plane::update(ParameterList *pl, HprAPI *api) {
  center = pl->getPoint("center", center);
  Point3 *b = pl->getPoint("point1", 0);
  Point3 *c = pl->getPoint("point2", 0);
  if (b != 0 && c != 0) {
    Point3 *v0 = center;
    Point3 *v1 = b;
    Point3 *v2 = c;
    Vector3 *ng = normal = Vector3::cross(Point3::sub(v1, v0, new Vector3()), Point3::sub(v2, v0,
        new Vector3()), new Vector3())->normalize();
    if (abs(ng->x) > abs(ng->y) && abs(ng->x) > abs(ng->z)) {
      k = 0;
    } else if (abs(ng->y) > abs(ng->z)) {
      k = 1;
    } else {
      k = 2;
    }
    float ax, ay, bx, by, cx, cy;
    switch (k) {
```

```
        case 0: {
          ax = v0->y;
          ay = v0->z;
          bx = v2->y - ax;
          by = v2->z - ay;
          cx = v1->y - ax;
          cy = v1->z - ay;
          break;
        }
        case 1: {
          ax = v0->z;
          ay = v0->x;
          bx = v2->z - ax;
          by = v2->x - ay;
          cx = v1->z - ax;
          cy = v1->x - ay;
          break;
        }
        case 2:
        default: {
          ax = v0->x;
          ay = v0->y;
          bx = v2->x - ax;
          by = v2->y - ay;
          cx = v1->x - ax;
          cy = v1->y - ay;
        }
      }
      float det = bx * cy - by * cx;
      bnu = -by / det;
      bnv = bx / det;
      bnd = (by * ax - bx * ay) / det;
      cnu = cy / det;
      cnv = -cx / det;
      cnd = (cx * ay - cy * ax) / det;
    } else {
      normal = pl->getVector("normal", normal);
      k = 3;
      bnu = bnv = bnd = 0;
      cnu = cnv = cnd = 0;
    }
    return true;
}

void Plane::prepareShadingState(ShadingState *state) {
```

```
  state->init();
  state->getRay()->getPoint(state->getPoint());
  Instance *parent = state->getInstance();
  Vector3 *worldNormal = state->transformNormalObjectToWorld(normal);
  state->getNormal()->set(worldNormal);
  state->getGeoNormal()->set(worldNormal);
  state->setShader(parent->getShader(0));
  state->setModifier(parent->getModifier(0));
  Point3 *p = state->transformWorldToObject(state->getPoint());
  float hu, hv;
  switch (k) {
    case 0: {
      hu = p->y;
      hv = p->z;
      break;
    }
    case 1: {
      hu = p->z;
      hv = p->x;
      break;
    }
    case 2: {
      hu = p->x;
      hv = p->y;
      break;
    }
    default:
      hu = hv = 0;
  }
  state->getUV()->x = hu * bnu + hv * bnv + bnd;
  state->getUV()->y = hu * cnu + hv * cnv + cnd;
  state->setBasis(OrthoNormalBasis::makeFromW(normal));
}

void Plane::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  float dn = normal->x * r->dx + normal->y * r->dy + normal->z * r->dz;
  if (dn == 0.0) {
    return;
  }
  float t = (((center->x - r->ox) * normal->x) + ((center->y - r->oy) * normal->y) + ((
      center->z - r->oz) * normal->z)) / dn;
  if (r->isInside(t)) {
    r->setMax(t);
    state->setIntersection(0);
  }
```

```
}

int Plane::getNumPrimitives() {
  return 1;
}

float Plane::getPrimitiveBound(int primID, int i) {
  return 0;
}

BoundingBox *Plane::getWorldBounds(Matrix4 *o2w) {
  return 0;
}

PrimitiveList *Plane::getBakingPrimitives() {
  return 0;
}
}
```

Listing A.25: Sphere.h

```
#pragma once
#include "HprAPI.h"
#include "ParameterList.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include "ShadingState.h"
#include "Instance.h"
#include "Point3.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
#include "Ray.h"
#include "IntersectionState.h"
#include "Solvers.h"
#include <cmath>

namespace hpr.core.primitive {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::IntersectionState;
  using hpr::core::ParameterList;
  using hpr::core::PrimitiveList;
  using hpr::core::Ray;
  using hpr::core::ShadingState;
  using hpr::math::BoundingBox;
```

```
  using hpr::math::Matrix4;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Point3;
  using hpr::math::Solvers;
  using hpr::math::Vector3;

  class Sphere : PrimitiveList {
    public:
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual float getPrimitiveBound(int primID, int i);
    virtual int getNumPrimitives();
    virtual void prepareShadingState(ShadingState *state);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual PrimitiveList *getBakingPrimitives();
  };
}
```

Listing A.26: Sphere.cpp

```
#include "Sphere.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::IntersectionState;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::math::BoundingBox;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Solvers;
using hpr::math::Vector3;

bool Sphere::update(ParameterList *pl, HprAPI *api) {
  return true;
}

BoundingBox *Sphere::getWorldBounds(Matrix4 *o2w) {
  BoundingBox *bounds = new BoundingBox(1);
  if (o2w != 0) {
    bounds = o2w->transform(bounds);
  }
  return bounds;
```

```
}

float Sphere::getPrimitiveBound(int primID, int i) {
  return (i & 1) == 0 ? -1 : 1;
}

int Sphere::getNumPrimitives() {
  return 1;
}

void Sphere::prepareShadingState(ShadingState *state) {
  state->init();
  state->getRay()->getPoint(state->getPoint());
  Instance *parent = state->getInstance();
  Point3 *localPoint = state->transformWorldToObject(state->getPoint());
  state->getNormal()->set(localPoint->x, localPoint->y, localPoint->z);
  state->getNormal()->normalize();

  float phi = static_cast<float> (atan2(state->getNormal()->y, state->getNormal()->x));
  if (phi < 0) {
    phi += 2 * 3.1415;
  }
  float theta = static_cast<float> (acos(state->getNormal()->z));
  state->getUV()->y = theta / static_cast<float> (3.1415);
  state->getUV()->x = phi / static_cast<float> (2 * 3.1415);
  Vector3 *v = new Vector3();
  v->x = -2 * static_cast<float> (3.1415) * state->getNormal()->y;
  v->y = 2 * static_cast<float> (3.1415) * state->getNormal()->x;
  v->z = 0;
  state->setShader(parent->getShader(0));
  state->setModifier(parent->getModifier(0));
  // into world space
  Vector3 *worldNormal = state->transformNormalObjectToWorld(state->getNormal());
  v = state->transformVectorObjectToWorld(v);
  state->getNormal()->set(worldNormal);
  state->getNormal()->normalize();
  state->getGeoNormal()->set(state->getNormal());
  // compute basis in world space
  state->setBasis(OrthoNormalBasis::makeFromWV(state->getNormal(), v));
}

void Sphere::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  // intersect in local space
  float qa = r->dx * r->dx + r->dy * r->dy + r->dz * r->dz;
  float qb = 2 * ((r->dx * r->ox) + (r->dy * r->oy) + (r->dz * r->oz));
```

```
  float qc = ((r->ox * r->ox) + (r->oy * r->oy) + (r->oz * r->oz)) - 1;
  double *t = Solvers::solveQuadric(qa, qb, qc);
  if (t != 0) {
    if (t[0] >= r->getMax() || t[1] <= r->getMin()) {
      return;
    }
    if (t[0] > r->getMin()) {
      r->setMax(static_cast<float> (t[0]));
    } else {
      r->setMax(static_cast<float> (t[1]));
    }
    state->setIntersection(0);
  }
}

PrimitiveList *Sphere::getBakingPrimitives() {
  return 0;
}
}
```

Listing A.27: TriangleMesh.h

```
#pragma once
#include "ParameterList.h"
#include "UI.h"
#include "HprAPI.h"
#include "MathUtils.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include "Ray.h"
#include "IntersectionState.h"
#include "ShadingState.h"
#include "Instance.h"
#include "Point3.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
#include <string>
#include <cmath>

namespace hpr.core.primitive {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::IntersectionState;
  using hpr::core::ParameterList;
  using hpr::core::PrimitiveList;
```

```
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::core::ParameterList::FloatParameter;
using hpr::core::ParameterList::InterpolationType;
using hpr::math::BoundingBox;
using hpr::math::MathUtils;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Vector3;
using hpr::system::UI;
using hpr::system::UI::Module;

class TriangleMesh : PrimitiveList  {
  private:
  static bool smallTriangles;
  WaldTriangle *triaccel;
  FloatParameter *normals;
  FloatParameter *uvs;
  char *faceShaders;
  void intersectTriangleKensler(Ray *r, int primID, IntersectionState *state);
  class WaldTriangle {
    // private data for fast triangle intersection testing
    private:
    int k;
    float nu, nv, nd;
    float bnu, bnv, bnd;
    float cnu, cnv, cnd;
    WaldTriangle(TriangleMesh *mesh, int tri);

    public:
    void intersect(Ray *r, int primID, IntersectionState *state);
  };
  class BakingSurface : PrimitiveList {
    public:
    virtual PrimitiveList *getBakingPrimitives();
    virtual int getNumPrimitives();
    virtual float getPrimitiveBound(int primID, int i);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual void prepareShadingState(ShadingState *state);
    virtual bool update(ParameterList *pl, HprAPI *api);
  };

  protected:
```

```
    float *points;
    int *triangles;
    virtual Point3 *getPoint(int i);

    public:
    static void setSmallTriangles(bool smallTriangles);
    TriangleMesh();
    virtual void writeObj(std::string filename);
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual float getPrimitiveBound(int primID, int i);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual int getNumPrimitives();
    virtual void prepareShadingState(ShadingState *state);
    virtual void init();
    virtual void getPoint(int tri, int i, Point3 *p);
    virtual PrimitiveList *getBakingPrimitives();
  };
}
```

Listing A.28: TriangleMesh.cpp

```
#include "TriangleMesh.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::IntersectionState;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::core::ParameterList::FloatParameter;
using hpr::core::ParameterList::InterpolationType;
using hpr::math::BoundingBox;
using hpr::math::MathUtils;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Vector3;
using hpr::system::UI;
using hpr::system::UI::Module;

void TriangleMesh::setSmallTriangles(bool smallTriangles) {
  if (smallTriangles) {
    UI::printInfo(Module::GEOM, "Small trimesh mode: enabled");
  } else {
```

```
    UI::printInfo(Module::GEOM, "Small trimesh mode: disabled");
  }
  TriangleMesh::smallTriangles = smallTriangles;
}

TriangleMesh::TriangleMesh() {
  delete[] triangles;
  delete[] points;
  normals = uvs = new FloatParameter();
  delete[] faceShaders;
}

void TriangleMesh::writeObj(std::string filename) {
  try {
    FileWriter *file = new FileWriter(filename);
    file->write(std::string::format("o object\n"));
    for (int i = 0; i < sizeof(points) / sizeof(points[0]); i += 3) {
      file->write(std::string::format("v %g %g %g\n", points[i], points[i + 1], points[i + 2]));
    }
    file->write("s off\n");
    for (int i = 0; i < sizeof(triangles) / sizeof(triangles[0]); i += 3) {
      file->write(std::string::format("f %d %d %d\n", triangles[i] + 1, triangles[i + 1] + 1,
          triangles[i + 2] + 1));
    }
    file->close();
  } catch (IOException *e) {
    e->printStackTrace();
  }
}

bool TriangleMesh::update(ParameterList *pl, HprAPI *api) {
  bool updatedTopology = false;
  int *triangles = pl->getIntArray("triangles");
  if (triangles != 0) {
    this->triangles = triangles;
    updatedTopology = true;
  }
  if (triangles == 0) {
    UI::printError(Module::GEOM, "Unable to update mesh - triangle indices are missing");
    return false;
  }
  if (sizeof(triangles) / sizeof(triangles[0]) % 3 != 0) {
    UI::printWarning(Module::GEOM, "Triangle index data is not a multiple of 3 - triangles may
        be missing");
  }
```

```
  pl->setFaceCount(sizeof(triangles) / sizeof(triangles[0]) / 3);
  FloatParameter *pointsP = pl->getPointArray("points");
  if (pointsP != 0) {
    if (pointsP->interp != InterpolationType::VERTEX) {
      UI::printError(Module::GEOM, "Point interpolation type must be set to \"vertex\" - was
          \"%s\"", pointsP->interp::name()->toLowerCase(Locale::ENGLISH));
    } else {
      points = pointsP->data;
      updatedTopology = true;
    }
  }
  if (points == 0) {
    UI::printError(Module::GEOM, "Unable to update mesh - vertices are missing");
    return false;
  }
  pl->setVertexCount(sizeof(points) / sizeof(points[0]) / 3);
  pl->setFaceVertexCount(3 * (sizeof(triangles) / sizeof(triangles[0]) / 3));
  FloatParameter *normals = pl->getVectorArray("normals");
  if (normals != 0) {
    this->normals = normals;
  }
  FloatParameter *uvs = pl->getTexCoordArray("uvs");
  if (uvs != 0) {
    this->uvs = uvs;
  }
  int *faceShaders = pl->getIntArray("faceshaders");
  if (faceShaders != 0 && sizeof(faceShaders) / sizeof(faceShaders[0]) == sizeof(triangles) / sizeof(
      triangles[0]) / 3) {
    this->faceShaders = new char[sizeof(faceShaders) / sizeof(faceShaders[0])];
    for (int i = 0; i < sizeof(faceShaders) / sizeof(faceShaders[0]); i++) {
      int v = faceShaders[i];
      if (v > 255) {
        UI::printWarning(Module::GEOM, "Shader index too large on triangle %d", i);
      }
      this->faceShaders[i] = static_cast<char> (v & 0xFF);
    }
  }
  if (updatedTopology) {
    // create triangle acceleration structure
    init();
  }
  return true;
}

float TriangleMesh::getPrimitiveBound(int primID, int i) {
```

```
  int tri = 3 * primID;
  int a = 3 * triangles[tri + 0];
  int b = 3 * triangles[tri + 1];
  int c = 3 * triangles[tri + 2];
  int axis = static_cast<unsigned int> (i) >> 1;
  if ((i & 1) == 0) {
    return MathUtils::min(points[a + axis], points[b + axis], points[c + axis]);
  } else {
    return MathUtils::max(points[a + axis], points[b + axis], points[c + axis]);
  }
}

BoundingBox *TriangleMesh::getWorldBounds(Matrix4 *o2w) {
  BoundingBox *bounds = new BoundingBox();
  if (o2w == 0) {
    for (int i = 0; i < sizeof(points) / sizeof(points[0]); i += 3) {
      bounds->include(points[i], points[i + 1], points[i + 2]);
    }
  } else {
    // transform vertices first
    for (int i = 0; i < sizeof(points) / sizeof(points[0]); i += 3) {
      float x = points[i];
      float y = points[i + 1];
      float z = points[i + 2];
      float wx = o2w->transformPX(x, y, z);
      float wy = o2w->transformPY(x, y, z);
      float wz = o2w->transformPZ(x, y, z);
      bounds->include(wx, wy, wz);
    }
  }
  return bounds;
}

void TriangleMesh::intersectTriangleKensler(Ray *r, int primID, IntersectionState *state) {
  int tri = 3 * primID;
  int a = 3 * triangles[tri + 0];
  int b = 3 * triangles[tri + 1];
  int c = 3 * triangles[tri + 2];
  float edge0x = points[b + 0] - points[a + 0];
  float edge0y = points[b + 1] - points[a + 1];
  float edge0z = points[b + 2] - points[a + 2];
  float edge1x = points[a + 0] - points[c + 0];
  float edge1y = points[a + 1] - points[c + 1];
  float edge1z = points[a + 2] - points[c + 2];
  float nx = edge0y * edge1z - edge0z * edge1y;
```

```
  float ny = edge0z * edge1x - edge0x * edge1z;
  float nz = edge0x * edge1y - edge0y * edge1x;
  float v = r->dot(nx, ny, nz);
  float iv = 1 / v;
  float edge2x = points[a + 0] - r->ox;
  float edge2y = points[a + 1] - r->oy;
  float edge2z = points[a + 2] - r->oz;
  float va = nx * edge2x + ny * edge2y + nz * edge2z;
  float t = iv * va;
  if (!r->isInside(t)) {
    return;
  }
  float ix = edge2y * r->dz - edge2z * r->dy;
  float iy = edge2z * r->dx - edge2x * r->dz;
  float iz = edge2x * r->dy - edge2y * r->dx;
  float v1 = ix * edge1x + iy * edge1y + iz * edge1z;
  float beta = iv * v1;
  if (beta < 0) {
    return;
  }
  float v2 = ix * edge0x + iy * edge0y + iz * edge0z;
  if ((v1 + v2) * v > v * v) {
    return;
  }
  float gamma = iv * v2;
  if (gamma < 0) {
    return;
  }
  r->setMax(t);
  state->setIntersection(primID, beta, gamma);
}

void TriangleMesh::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  // alternative test -- disabled for now
  // intersectPrimitiveRobust(r, primID, state);
  if (triaccel != 0) {
    // optional fast intersection method
    triaccel[primID]->intersect(r, primID, state);
    return;
  }
  intersectTriangleKensler(r, primID, state);
}

int TriangleMesh::getNumPrimitives() {
  return sizeof(triangles) / sizeof(triangles[0]) / 3;
```

```
}

void TriangleMesh::prepareShadingState(ShadingState *state) {
  state->init();
  Instance *parent = state->getInstance();
  int primID = state->getPrimitiveID();
  float u = state->getU();
  float v = state->getV();
  float w = 1 - u - v;
  state->getRay()->getPoint(state->getPoint());
  int tri = 3 * primID;
  int index0 = triangles[tri + 0];
  int index1 = triangles[tri + 1];
  int index2 = triangles[tri + 2];
  Point3 *v0p = getPoint(index0);
  Point3 *v1p = getPoint(index1);
  Point3 *v2p = getPoint(index2);
  Vector3 *ng = Point3::normal(v0p, v1p, v2p);
  ng = state->transformNormalObjectToWorld(ng);
  ng->normalize();
  state->getGeoNormal()->set(ng);
  switch (normals->interp) {
    case NONE:
    case FACE: {
      state->getNormal()->set(ng);
      break;
    }
    case VERTEX: {
      int i30 = 3 * index0;
      int i31 = 3 * index1;
      int i32 = 3 * index2;
      float *normals = this->normals->data;
      state->getNormal()->x = w * normals[i30 + 0] + u * normals[i31 + 0] + v * normals[i32 +
          0];
      state->getNormal()->y = w * normals[i30 + 1] + u * normals[i31 + 1] + v * normals[i32 +
          1];
      state->getNormal()->z = w * normals[i30 + 2] + u * normals[i31 + 2] + v * normals[i32 +
          2];
      state->getNormal()->set(state->transformNormalObjectToWorld(state->getNormal()));
      state->getNormal()->normalize();
      break;
    }
    case FACEVARYING: {
      int idx = 3 * tri;
      float *normals = this->normals->data;
```

```
      state->getNormal()->x = w * normals[idx + 0] + u * normals[idx + 3] + v * normals[idx
          + 6];
      state->getNormal()->y = w * normals[idx + 1] + u * normals[idx + 4] + v * normals[idx
          + 7];
      state->getNormal()->z = w * normals[idx + 2] + u * normals[idx + 5] + v * normals[idx
          + 8];
      state->getNormal()->set(state->transformNormalObjectToWorld(state->getNormal()));
      state->getNormal()->normalize();
      break;
    }
  }
  float uv00 = 0, uv01 = 0, uv10 = 0, uv11 = 0, uv20 = 0, uv21 = 0;
  switch (uvs->interp) {
    case NONE:
    case FACE: {
      state->getUV()->x = 0;
      state->getUV()->y = 0;
      break;
    }
    case VERTEX: {
      int i20 = 2 * index0;
      int i21 = 2 * index1;
      int i22 = 2 * index2;
      float *uvs = this->uvs->data;
      uv00 = uvs[i20 + 0];
      uv01 = uvs[i20 + 1];
      uv10 = uvs[i21 + 0];
      uv11 = uvs[i21 + 1];
      uv20 = uvs[i22 + 0];
      uv21 = uvs[i22 + 1];
      break;
    }
    case FACEVARYING: {
      int idx = tri << 1;
      float *uvs = this->uvs->data;
      uv00 = uvs[idx + 0];
      uv01 = uvs[idx + 1];
      uv10 = uvs[idx + 2];
      uv11 = uvs[idx + 3];
      uv20 = uvs[idx + 4];
      uv21 = uvs[idx + 5];
      break;
    }
  }
  if (uvs->interp != InterpolationType::NONE) {
```

```
    // get exact uv coords and compute tangent vectors
    state->getUV()->x = w * uv00 + u * uv10 + v * uv20;
    state->getUV()->y = w * uv01 + u * uv11 + v * uv21;
    float du1 = uv00 - uv20;
    float du2 = uv10 - uv20;
    float dv1 = uv01 - uv21;
    float dv2 = uv11 - uv21;
    Vector3 *dp1 = Point3::sub(v0p, v2p, new Vector3()), *dp2 = Point3::sub(v1p, v2p, new
        Vector3());
    float determinant = du1 * dv2 - dv1 * du2;
    if (determinant == 0.0f) {
      // create basis in world space
      state->setBasis(OrthoNormalBasis::makeFromW(state->getNormal()));
    } else {
      float invdet = 1.f / determinant;
      Vector3 *dpdv = new Vector3();
      dpdv->x = (-du2 * dp1->x + du1 * dp2->x) * invdet;
      dpdv->y = (-du2 * dp1->y + du1 * dp2->y) * invdet;
      dpdv->z = (-du2 * dp1->z + du1 * dp2->z) * invdet;
      dpdv = state->transformVectorObjectToWorld(dpdv);
      // create basis in world space
      state->setBasis(OrthoNormalBasis::makeFromWV(state->getNormal(), dpdv));
    }
  } else {
    state->setBasis(OrthoNormalBasis::makeFromW(state->getNormal()));
  }
  int shaderIndex = faceShaders == 0 ? 0 : (faceShaders[primID] & 0xFF);
  state->setShader(parent->getShader(shaderIndex));
  state->setModifier(parent->getModifier(shaderIndex));
}

void TriangleMesh::init() {
  delete[] triaccel;
  int nt = getNumPrimitives();
  if (!smallTriangles) {
    if (nt > 2000000) {
      UI::printWarning(Module::GEOM, "TRI - Too many triangles -- triaccel generation
          skipped");
      return;
    }
    triaccel = new WaldTriangle[nt];
    for (int i = 0; i < nt; i++) {
      triaccel[i] = new WaldTriangle(this, i);
    }
  }
```

```
}

Point3 *TriangleMesh::getPoint(int i) {
  i *= 3;
  return new Point3(points[i], points[i + 1], points[i + 2]);
}

void TriangleMesh::getPoint(int tri, int i, Point3 *p) {
  int index = 3 * triangles[3 * tri + i];
  p->set(points[index], points[index + 1], points[index + 2]);
}

TriangleMesh::WaldTriangle::WaldTriangle(TriangleMesh *mesh, int tri) {
  k = 0;
  tri *= 3;
  int index0 = mesh->triangles[tri + 0];
  int index1 = mesh->triangles[tri + 1];
  int index2 = mesh->triangles[tri + 2];
  Point3 *v0p = mesh->getPoint(index0);
  Point3 *v1p = mesh->getPoint(index1);
  Point3 *v2p = mesh->getPoint(index2);
  Vector3 *ng = Point3::normal(v0p, v1p, v2p);
  if (abs(ng->x) > abs(ng->y) && abs(ng->x) > abs(ng->z)) {
    k = 0;
  } else if (abs(ng->y) > abs(ng->z)) {
    k = 1;
  } else {
    k = 2;
  }
  float ax, ay, bx, by, cx, cy;
  switch (k) {
    case 0: {
      nu = ng->y / ng->x;
      nv = ng->z / ng->x;
      nd = v0p->x + (nu * v0p->y) + (nv * v0p->z);
      ax = v0p->y;
      ay = v0p->z;
      bx = v2p->y - ax;
      by = v2p->z - ay;
      cx = v1p->y - ax;
      cy = v1p->z - ay;
      break;
    }
    case 1: {
      nu = ng->z / ng->y;
```

```
      nv = ng->x / ng->y;
      nd = (nv * v0p->x) + v0p->y + (nu * v0p->z);
      ax = v0p->z;
      ay = v0p->x;
      bx = v2p->z - ax;
      by = v2p->x - ay;
      cx = v1p->z - ax;
      cy = v1p->x - ay;
      break;
    }
    case 2:
    default: {
      nu = ng->x / ng->z;
      nv = ng->y / ng->z;
      nd = (nu * v0p->x) + (nv * v0p->y) + v0p->z;
      ax = v0p->x;
      ay = v0p->y;
      bx = v2p->x - ax;
      by = v2p->y - ay;
      cx = v1p->x - ax;
      cy = v1p->y - ay;
    }
  }
  float det = bx * cy - by * cx;
  bnu = -by / det;
  bnv = bx / det;
  bnd = (by * ax - bx * ay) / det;
  cnu = cy / det;
  cnv = -cx / det;
  cnd = (cx * ay - cy * ax) / det;
}

void TriangleMesh::WaldTriangle::intersect(Ray *r, int primID, IntersectionState *state) {
  switch (k) {
    case 0: {
      float det = 1.0f / (r->dx + nu * r->dy + nv * r->dz);
      float t = (nd - r->ox - nu * r->oy - nv * r->oz) * det;
      if (!r->isInside(t)) {
        return;
      }
      float hu = r->oy + t * r->dy;
      float hv = r->oz + t * r->dz;
      float u = hu * bnu + hv * bnv + bnd;
      if (u < 0.0f) {
        return;
```

```
    }
    float v = hu * cnu + hv * cnv + cnd;
    if (v < 0.0f) {
      return;
    }
    if (u + v > 1.0f) {
      return;
    }
    r->setMax(t);
    state->setIntersection(primID, u, v);
    return;
  }
  case 1: {
    float det = 1.0f / (r->dy + nu * r->dz + nv * r->dx);
    float t = (nd - r->oy - nu * r->oz - nv * r->ox) * det;
    if (!r->isInside(t)) {
      return;
    }
    float hu = r->oz + t * r->dz;
    float hv = r->ox + t * r->dx;
    float u = hu * bnu + hv * bnv + bnd;
    if (u < 0.0f) {
      return;
    }
    float v = hu * cnu + hv * cnv + cnd;
    if (v < 0.0f) {
      return;
    }
    if (u + v > 1.0f) {
      return;
    }
    r->setMax(t);
    state->setIntersection(primID, u, v);
    return;
  }
  case 2: {
    float det = 1.0f / (r->dz + nu * r->dx + nv * r->dy);
    float t = (nd - r->oz - nu * r->ox - nv * r->oy) * det;
    if (!r->isInside(t)) {
      return;
    }
    float hu = r->ox + t * r->dx;
    float hv = r->oy + t * r->dy;
    float u = hu * bnu + hv * bnv + bnd;
    if (u < 0.0f) {
```

```
        return;
      }
      float v = hu * cnu + hv * cnv + cnd;
      if (v < 0.0f) {
        return;
      }
      if (u + v > 1.0f) {
        return;
      }
      r->setMax(t);
      state->setIntersection(primID, u, v);
      return;
    }
  }
}

PrimitiveList *TriangleMesh::getBakingPrimitives() {
  switch (uvs->interp) {
    case NONE:
    case FACE:
      UI::printError(Module::GEOM, "Cannot generate baking surface without texture coordinate
          data");
      return 0;
    default:
      return new BakingSurface();
  }
}

PrimitiveList *TriangleMesh::BakingSurface::getBakingPrimitives() {
  return 0;
}

int TriangleMesh::BakingSurface::getNumPrimitives() {
  return TriangleMesh::getNumPrimitives();
}

float TriangleMesh::BakingSurface::getPrimitiveBound(int primID, int i) {
  if (i > 3) {
    return 0;
  }
  switch (uvs::interp) {
    case NONE:
    case FACE:
    default: {
      return 0;
```

```
    }
    case VERTEX: {
      int tri = 3 * primID;
      int index0 = triangles[tri + 0];
      int index1 = triangles[tri + 1];
      int index2 = triangles[tri + 2];
      int i20 = 2 * index0;
      int i21 = 2 * index1;
      int i22 = 2 * index2;
      float *uvs = TriangleMesh::uvs::data;
      switch (i) {
        case 0:
          return MathUtils::min(uvs[i20 + 0], uvs[i21 + 0], uvs[i22 + 0]);
        case 1:
          return MathUtils::max(uvs[i20 + 0], uvs[i21 + 0], uvs[i22 + 0]);
        case 2:
          return MathUtils::min(uvs[i20 + 1], uvs[i21 + 1], uvs[i22 + 1]);
        case 3:
          return MathUtils::max(uvs[i20 + 1], uvs[i21 + 1], uvs[i22 + 1]);
        default:
          return 0;
      }
    }
    case FACEVARYING: {
      int idx = 6 * primID;
      float *uvs = TriangleMesh::uvs::data;
      switch (i) {
        case 0:
          return MathUtils::min(uvs[idx + 0], uvs[idx + 2], uvs[idx + 4]);
        case 1:
          return MathUtils::max(uvs[idx + 0], uvs[idx + 2], uvs[idx + 4]);
        case 2:
          return MathUtils::min(uvs[idx + 1], uvs[idx + 3], uvs[idx + 5]);
        case 3:
          return MathUtils::max(uvs[idx + 1], uvs[idx + 3], uvs[idx + 5]);
        default:
          return 0;
      }
    }
  }
}

BoundingBox *TriangleMesh::BakingSurface::getWorldBounds(Matrix4 *o2w) {
  BoundingBox *bounds = new BoundingBox();
  if (o2w == 0) {
```

```
    for (int i = 0; i < uvs::data::length; i += 2) {
      bounds->include(uvs::data[i], uvs::data[i + 1], 0);
    }
  } else {
    // transform vertices first
    for (int i = 0; i < uvs::data::length; i += 2) {
      float x = uvs::data[i];
      float y = uvs::data[i + 1];
      float wx = o2w->transformPX(x, y, 0);
      float wy = o2w->transformPY(x, y, 0);
      float wz = o2w->transformPZ(x, y, 0);
      bounds->include(wx, wy, wz);
    }
  }
  return bounds;
}

void TriangleMesh::BakingSurface::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  float uv00 = 0, uv01 = 0, uv10 = 0, uv11 = 0, uv20 = 0, uv21 = 0;
  switch (uvs::interp) {
    case NONE:
    case FACE:
    default:
      return;
    case VERTEX: {
      int tri = 3 * primID;
      int index0 = triangles[tri + 0];
      int index1 = triangles[tri + 1];
      int index2 = triangles[tri + 2];
      int i20 = 2 * index0;
      int i21 = 2 * index1;
      int i22 = 2 * index2;
      float *uvs = TriangleMesh::uvs::data;
      uv00 = uvs[i20 + 0];
      uv01 = uvs[i20 + 1];
      uv10 = uvs[i21 + 0];
      uv11 = uvs[i21 + 1];
      uv20 = uvs[i22 + 0];
      uv21 = uvs[i22 + 1];
      break;

    }
    case FACEVARYING: {
      int idx = (3 * primID) << 1;
      float *uvs = TriangleMesh::uvs::data;
```

```
      uv00 = uvs[idx + 0];
      uv01 = uvs[idx + 1];
      uv10 = uvs[idx + 2];
      uv11 = uvs[idx + 3];
      uv20 = uvs[idx + 4];
      uv21 = uvs[idx + 5];
      break;
    }
  }

  double edge1x = uv10 - uv00;
  double edge1y = uv11 - uv01;
  double edge2x = uv20 - uv00;
  double edge2y = uv21 - uv01;
  double pvecx = r->dy * 0 - r->dz * edge2y;
  double pvecy = r->dz * edge2x - r->dx * 0;
  double pvecz = r->dx * edge2y - r->dy * edge2x;
  double qvecx, qvecy, qvecz;
  double u, v;
  double det = edge1x * pvecx + edge1y * pvecy + 0 * pvecz;
  if (det > 0) {
    double tvecx = r->ox - uv00;
    double tvecy = r->oy - uv01;
    double tvecz = r->oz;
    u = (tvecx * pvecx + tvecy * pvecy + tvecz * pvecz);
    if (u < 0.0 || u > det) {
      return;
    }
    qvecx = tvecy * 0 - tvecz * edge1y;
    qvecy = tvecz * edge1x - tvecx * 0;
    qvecz = tvecx * edge1y - tvecy * edge1x;
    v = (r->dx * qvecx + r->dy * qvecy + r->dz * qvecz);
    if (v < 0.0 || u + v > det) {
      return;
    }
  } else if (det < 0) {
    double tvecx = r->ox - uv00;
    double tvecy = r->oy - uv01;
    double tvecz = r->oz;
    u = (tvecx * pvecx + tvecy * pvecy + tvecz * pvecz);
    if (u > 0.0 || u < det) {
      return;
    }
    qvecx = tvecy * 0 - tvecz * edge1y;
    qvecy = tvecz * edge1x - tvecx * 0;
```

```
    qvecz = tvecx * edge1y - tvecy * edge1x;
    v = (r->dx * qvecx + r->dy * qvecy + r->dz * qvecz);
    if (v > 0.0 || u + v < det) {
      return;
    }
  } else {
    return;
  }
  double inv_det = 1.0 / det;
  float t = static_cast<float> ((edge2x * qvecx + edge2y * qvecy + 0 * qvecz) * inv_det);
  if (r->isInside(t)) {
    r->setMax(t);
    state->setIntersection(primID, static_cast<float> (u * inv_det), static_cast<float> (v *
        inv_det));
  }
}

void TriangleMesh::BakingSurface::prepareShadingState(ShadingState *state) {
  state->init();
  Instance *parent = state->getInstance();
  int primID = state->getPrimitiveID();
  float u = state->getU();
  float v = state->getV();
  float w = 1 - u - v;
  // state.getRay().getPoint(state.getPoint());
  int tri = 3 * primID;
  int index0 = triangles[tri + 0];
  int index1 = triangles[tri + 1];
  int index2 = triangles[tri + 2];
  Point3 *v0p = getPoint(index0);
  Point3 *v1p = getPoint(index1);
  Point3 *v2p = getPoint(index2);

  // get object space point from barycentric coordinates
  state->getPoint()->x = w * v0p->x + u * v1p->x + v * v2p->x;
  state->getPoint()->y = w * v0p->y + u * v1p->y + v * v2p->y;
  state->getPoint()->z = w * v0p->z + u * v1p->z + v * v2p->z;
  // move into world space
  state->getPoint()->set(state->transformObjectToWorld(state->getPoint()));

  Vector3 *ng = Point3::normal(v0p, v1p, v2p);
  if (parent != 0) {
    ng = state->transformNormalObjectToWorld(ng);
  }
  ng->normalize();
```

```
state->getGeoNormal()->set(ng);
switch (normals::interp) {
  case NONE:
  case FACE: {
    state->getNormal()->set(ng);
    break;
  }
  case VERTEX: {
    int i30 = 3 * index0;
    int i31 = 3 * index1;
    int i32 = 3 * index2;
    float *normals = TriangleMesh::normals::data;
    state->getNormal()->x = w * normals[i30 + 0] + u * normals[i31 + 0] + v * normals[i32 +
        0];
    state->getNormal()->y = w * normals[i30 + 1] + u * normals[i31 + 1] + v * normals[i32 +
        1];
    state->getNormal()->z = w * normals[i30 + 2] + u * normals[i31 + 2] + v * normals[i32 +
        2];
    if (parent != 0) {
      state->getNormal()->set(state->transformNormalObjectToWorld(state->getNormal()))
          ;
    }
    state->getNormal()->normalize();
    break;
  }
  case FACEVARYING: {
    int idx = 3 * tri;
    float *normals = TriangleMesh::normals::data;
    state->getNormal()->x = w * normals[idx + 0] + u * normals[idx + 3] + v * normals[idx
        + 6];
    state->getNormal()->y = w * normals[idx + 1] + u * normals[idx + 4] + v * normals[idx
        + 7];
    state->getNormal()->z = w * normals[idx + 2] + u * normals[idx + 5] + v * normals[idx
        + 8];
    if (parent != 0) {
      state->getNormal()->set(state->transformNormalObjectToWorld(state->getNormal()))
          ;
    }
    state->getNormal()->normalize();
    break;
  }
}
float uv00 = 0, uv01 = 0, uv10 = 0, uv11 = 0, uv20 = 0, uv21 = 0;
switch (uvs::interp) {
  case NONE:
```

```
    case FACE: {
      state->getUV()->x = 0;
      state->getUV()->y = 0;
      break;
    }
    case VERTEX: {
      int i20 = 2 * index0;
      int i21 = 2 * index1;
      int i22 = 2 * index2;
      float *uvs = TriangleMesh::uvs::data;
      uv00 = uvs[i20 + 0];
      uv01 = uvs[i20 + 1];
      uv10 = uvs[i21 + 0];
      uv11 = uvs[i21 + 1];
      uv20 = uvs[i22 + 0];
      uv21 = uvs[i22 + 1];
      break;
    }
    case FACEVARYING: {
      int idx = tri << 1;
      float *uvs = TriangleMesh::uvs::data;
      uv00 = uvs[idx + 0];
      uv01 = uvs[idx + 1];
      uv10 = uvs[idx + 2];
      uv11 = uvs[idx + 3];
      uv20 = uvs[idx + 4];
      uv21 = uvs[idx + 5];
      break;
    }
  }
  if (uvs::interp != InterpolationType::NONE) {
    // get exact uv coords and compute tangent vectors
    state->getUV()->x = w * uv00 + u * uv10 + v * uv20;
    state->getUV()->y = w * uv01 + u * uv11 + v * uv21;
    float du1 = uv00 - uv20;
    float du2 = uv10 - uv20;
    float dv1 = uv01 - uv21;
    float dv2 = uv11 - uv21;
    Vector3 *dp1 = Point3::sub(v0p, v2p, new Vector3()), *dp2 = Point3::sub(v1p, v2p, new
        Vector3());
    float determinant = du1 * dv2 - dv1 * du2;
    if (determinant == 0.0f) {
      // create basis in world space
      state->setBasis(OrthoNormalBasis::makeFromW(state->getNormal()));
    } else {
```

```
        float invdet = 1.f / determinant;
        // Vector3 dpdu = new Vector3();
        // dpdu.x = (dv2 * dp1.x - dv1 * dp2.x) * invdet;
        // dpdu.y = (dv2 * dp1.y - dv1 * dp2.y) * invdet;
        // dpdu.z = (dv2 * dp1.z - dv1 * dp2.z) * invdet;
        Vector3 *dpdv = new Vector3();
        dpdv->x = (-du2 * dp1->x + du1 * dp2->x) * invdet;
        dpdv->y = (-du2 * dp1->y + du1 * dp2->y) * invdet;
        dpdv->z = (-du2 * dp1->z + du1 * dp2->z) * invdet;
        if (parent != 0) {
          dpdv = state->transformVectorObjectToWorld(dpdv);
        }
        // create basis in world space
        state->setBasis(OrthoNormalBasis::makeFromWV(state->getNormal(), dpdv));
      }
    } else {
      state->setBasis(OrthoNormalBasis::makeFromW(state->getNormal()));
    }
    int shaderIndex = faceShaders == 0 ? 0 : (faceShaders[primID] & 0xFF);
    state->setShader(parent->getShader(shaderIndex));
}

bool TriangleMesh::BakingSurface::update(ParameterList *pl, HprAPI *api) {
  return true;
}
}
```

Listing A.29: PointLight.h

```
#pragma once
#include "Point3.h"
#include "Color.h"
#include "HprAPI.h"
#include "ParameterList.h"
#include "ShadingState.h"
#include "Vector3.h"
#include "LightSample.h"
#include "Ray.h"
#include "Instance.h"
#include <cmath>

namespace hpr.core.light {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::LightSample;
```

```
  using hpr::core::LightSource;
  using hpr::core::ParameterList;
  using hpr::core::Ray;
  using hpr::core::ShadingState;
  using hpr::image::Color;
  using hpr::math::Point3;
  using hpr::math::Vector3;

  class PointLight : LightSource {
    private:
    Point3 *lightPoint;
    Color *power;

    public:
    PointLight();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual int getNumSamples();
    virtual void getSamples(ShadingState *state);
    virtual void getPhoton(double randX1, double randY1, double randX2, double randY2, Point3
        *p, Vector3 *dir, Color *power);
    virtual float getPower();
    virtual Instance *createInstance();
  };
}
```

Listing A.30: PointLight.cpp

```
#include "PointLight.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::LightSample;
using hpr::core::LightSource;
using hpr::core::ParameterList;
using hpr::core::Ray;
using hpr::core::ShadingState;
using hpr::image::Color;
using hpr::math::Point3;
using hpr::math::Vector3;

PointLight::PointLight() {
  lightPoint = new Point3(0, 0, 0);
  power = Color::WHITE;
}

bool PointLight::update(ParameterList *pl, HprAPI *api) {
```

```
  lightPoint = pl->getPoint("center", lightPoint);
  power = pl->getColor("power", power);
  return true;
}

int PointLight::getNumSamples() {
  return 1;
}

void PointLight::getSamples(ShadingState *state) {
  Vector3 *d = Point3::sub(lightPoint, state->getPoint(), new Vector3());
  if (Vector3::dot(d, state->getNormal()) > 0 && Vector3::dot(d, state->getGeoNormal()) > 0) {
    LightSample *dest = new LightSample();
    // prepare shadow ray
    dest->setShadowRay(new Ray(state->getPoint(), lightPoint));
    float scale = 1.0f / static_cast<float> (4 * 3.1415 * lightPoint->distanceToSquared(state->
        getPoint()));
    dest->setRadiance(power, power);
    dest->getDiffuseRadiance()->mul(scale);
    dest->getSpecularRadiance()->mul(scale);
    dest->traceShadow(state);
    state->addSample(dest);
  }
}

void PointLight::getPhoton(double randX1, double randY1, double randX2, double randY2, Point3
    *p, Vector3 *dir, Color *power) {
  p->set(lightPoint);
  float phi = static_cast<float> (2 * 3.1415 * randX1);
  float s = static_cast<float> (sqrt(randY1 * (1.0f - randY1)));
  dir->x = static_cast<float> (cos(phi)) * s;
  dir->y = static_cast<float> (sin(phi)) * s;
  dir->z = static_cast<float> (1 - 2 * randY1);
  power->set(this->power);
}

float PointLight::getPower() {
  return power.getLuminance();
}

Instance *PointLight::createInstance() {
  return 0;
}
}
```

Listing A.31: SunSkyLight.h

```
#pragma once
#include "OrthoNormalBasis.h"
#include "Color.h"
#include "Vector3.h"
#include "SpectralCurve.h"
#include "RegularSpectralCurve.h"
#include "IrregularSpectralCurve.h"
#include "MathUtils.h"
#include "RGBSpace.h"
#include "ConstantSpectralCurve.h"
#include "HprAPI.h"
#include "ParameterList.h"
#include "ChromaticitySpectrum.h"
#include "XYZColor.h"
#include "Point3.h"
#include "ShadingState.h"
#include "LightSample.h"
#include "Ray.h"
#include "BoundingBox.h"
#include "Matrix4.h"
#include "IntersectionState.h"
#include "Instance.h"
#include <cmath>

namespace hpr.core.light {

   using hpr::HprAPI;
   using hpr::core::Instance;
   using hpr::core::IntersectionState;
   using hpr::core::LightSample;
   using hpr::core::LightSource;
   using hpr::core::ParameterList;
   using hpr::core::PrimitiveList;
   using hpr::core::Ray;
   using hpr::core::Shader;
   using hpr::core::ShadingState;
   using hpr::image::ChromaticitySpectrum;
   using hpr::image::Color;
   using hpr::image::ConstantSpectralCurve;
   using hpr::image::IrregularSpectralCurve;
   using hpr::image::RGBSpace;
   using hpr::image::RegularSpectralCurve;
   using hpr::image::SpectralCurve;
   using hpr::image::XYZColor;
```

```
using hpr::math::BoundingBox;
using hpr::math::MathUtils;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Vector3;

class SunSkyLight : LightSource, PrimitiveList, Shader {
  private:
  int numSkySamples;
  OrthoNormalBasis *basis;
  bool groundExtendSky;
  Color *groundColor;
  Vector3 *sunDirWorld;
  float turbidity;
  Vector3 *sunDir;
  SpectralCurve *sunSpectralRadiance;
  Color *sunColor;
  float sunTheta;
  double zenithY, zenithx, zenithy;
  const double perezY[5];
  const double perezx[5];
  const double perezy[5];
  float jacobian;
  float *colHistogram;
  float **imageHistogram;
  static const float solAmplitudes[38] = {165.5f, 162.3f, 211.2f, 258.8f, 258.2f, 242.3f, 267.6f, 296.6f
      , 305.4f, 300.6f, 306.6f, 288.3f, 287.1f, 278.2f, 271.0f, 272.3f, 263.6f, 255.0f, 250.6f, 253.1f,
      253.5f, 251.3f, 246.3f, 241.7f, 236.8f, 232.1f, 228.2f, 223.4f, 219.7f, 215.3f, 211.0f, 207.3f,
      202.4f, 198.7f, 194.3f, 190.7f, 186.3f, 182.6f};
  static const RegularSpectralCurve *solCurve = new RegularSpectralCurve(solAmplitudes, 380,
      750);
  static const float k_oWavelengths[64] = {300, 305, 310, 315, 320, 325, 330, 335, 340, 345, 350,
      355, 445, 450, 455, 460, 465, 470, 475, 480, 485, 490, 495, 500, 505, 510, 515, 520, 525, 530,
      535, 540, 545, 550, 555, 560, 565, 570, 575, 580, 585, 590, 595, 600, 605, 610, 620, 630, 640,
      650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790};
  static const float k_oAmplitudes[64] = {10.0f, 4.8f, 2.7f, 1.35f,.8f,.380f,.160f,.075f,.04f,.019f,.007f
      ,.0f,.003f,.003f,.004f,.006f,.008f,.009f,.012f,.014f,.017f,.021f,.025f,.03f,.035f,.04f,.045f,.048f,.057
      f,.063f,.07f,.075f,.08f,.085f,.095f,.103f,.110f,.12f,.122f,.12f,.118f,.115f,.12f,.125f,.130f,.12f,.105f
      ,.09f,.079f,.067f,.057f,.048f,.036f,.028f,.023f,.018f,.014f,.011f,.010f,.009f,.007f,.004f,.0f,.0f};
  static const float k_gWavelengths[4] = {759, 760, 770, 771};
  static const float k_gAmplitudes[4] = {0, 3.0f, 0.210f, 0};
  static const float k_waWavelengths[13] = {689, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780,
       790, 800};
  static const float k_waAmplitudes[13] = {0, 0.160e-1f, 0.240e-1f, 0.125e-1f, 0.100e+1f, 0.870f,
```

```
        0.610e-1f, 0.100e-2f, 0.100e-4f, 0.100e-4f, 0.600e-3f, 0.175e-1f, 0.360e-1f};
    static const IrregularSpectralCurve *k_oCurve = new IrregularSpectralCurve(k_oWavelengths,
        k_oAmplitudes);
    static const IrregularSpectralCurve *k_gCurve = new IrregularSpectralCurve(k_gWavelengths,
        k_gAmplitudes);
    static const IrregularSpectralCurve *k_waCurve = new IrregularSpectralCurve(
      k_waWavelengths, k_waAmplitudes);
    SpectralCurve *computeAttenuatedSunlight(float theta, float turbidity);
    double perezFunction(double lam[], double theta, double gamma, double lvz);
    void initSunSky();
    Color *getSkyRGB(Vector3 *dir);
    Vector3 *getDirection(float u, float v);

    public:
    SunSkyLight();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual int getNumSamples();
    virtual void getPhoton(double randX1, double randY1, double randX2, double randY2, Point3
        *p, Vector3 *dir, Color *power);
    virtual float getPower();
    virtual void getSamples(ShadingState *state);
    virtual PrimitiveList *getBakingPrimitives();
    virtual int getNumPrimitives();
    virtual float getPrimitiveBound(int primID, int i);
    virtual BoundingBox *getWorldBounds(Matrix4 *o2w);
    virtual void intersectPrimitive(Ray *r, int primID, IntersectionState *state);
    virtual void prepareShadingState(ShadingState *state);
    virtual Color *getRadiance(ShadingState *state);
    virtual void scatterPhoton(ShadingState *state, Color *power);
    virtual Instance *createInstance();
  };
}

class RectangularArrays {
  public:
    static float** ReturnRectangularFloatArray(int Size1, int Size2) {
      float** Array = new float*[Size1];
      for (int Array1 = 0; Array1 < Size1; Array1++) {
        Array[Array1] = new float[Size2];
      }
      return Array;
    }
};
```

Listing A.32: SunSkyLight.cpp

```
#include "SunSkyLight.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::IntersectionState;
using hpr::core::LightSample;
using hpr::core::LightSource;
using hpr::core::ParameterList;
using hpr::core::PrimitiveList;
using hpr::core::Ray;
using hpr::core::Shader;
using hpr::core::ShadingState;
using hpr::image::ChromaticitySpectrum;
using hpr::image::Color;
using hpr::image::ConstantSpectralCurve;
using hpr::image::IrregularSpectralCurve;
using hpr::image::RGBSpace;
using hpr::image::RegularSpectralCurve;
using hpr::image::SpectralCurve;
using hpr::image::XYZColor;
using hpr::math::BoundingBox;
using hpr::math::MathUtils;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Vector3;

SunSkyLight::SunSkyLight() {
  numSkySamples = 64;
  sunDirWorld = new Vector3(1, 1, 1);
  turbidity = 6;
  basis = OrthoNormalBasis::makeFromWV(new Vector3(0, 0, 1), new Vector3(0, 1, 0));
  groundExtendSky = false;
  groundColor = Color::BLACK;
  initSunSky();
}

SpectralCurve *SunSkyLight::computeAttenuatedSunlight(float theta, float turbidity) {
  float data[91]; // holds the sunsky curve data
  const double alpha = 1.3;
  const double lozone = 0.35;
  const double w = 2.0;
  double beta = 0.04608365822050 * turbidity - 0.04586025928522;
  // Relative optical mass
```

```
  double m = 1.0 / (cos(theta) + 0.000940 * pow(1.6386 - theta, -1.253));
  for (int i = 0, lambda = 350; lambda <= 800; i++, lambda += 5) {
    // Rayleigh scattering
    double tauR = exp(-m * 0.008735 * pow(lambda / 1000.0, -4.08));
    // Aerosol (water + dust) attenuation
    double tauA = exp(-m * beta * pow(lambda / 1000.0, -alpha));
    // Attenuation due to ozone absorption
    double tauO = exp(-m * k_oCurve->sample(lambda) * lozone);
    // Attenuation due to mixed gases absorption
    double tauG = exp(-1.41 * k_gCurve->sample(lambda) * m / pow(1.0 + 118.93 * k_gCurve
        ->sample(lambda) * m, 0.45));
    // Attenuation due to water vapor absorption
    double tauWA = exp(-0.2385 * k_waCurve->sample(lambda) * w * m / pow(1.0 + 20.07 *
        k_waCurve->sample(lambda) * w * m, 0.45));
    // 100.0 comes from solAmplitudes begin in wrong units.
    double amp = solCurve->sample(lambda) * tauR * tauA * tauO * tauG * tauWA;
    data[i] = static_cast<float> (amp);
  }
  return new RegularSpectralCurve(data, 350, 800);
}

double SunSkyLight::perezFunction(double lam[], double theta, double gamma, double lvz) {
  double den = ((1.0 + lam[0] * exp(lam[1])) * (1.0 + lam[2] * exp(lam[3] * sunTheta) + lam[4] *
      cos(sunTheta) * cos(sunTheta)));
  double num = ((1.0 + lam[0] * exp(lam[1] / cos(theta))) * (1.0 + lam[2] * exp(lam[3] * gamma)
      + lam[4] * cos(gamma) * cos(gamma)));
  return lvz * num / den;
}

void SunSkyLight::initSunSky() {
  // perform all the required initialization of constants
  sunDirWorld->normalize();
  sunDir = basis->untransform(sunDirWorld, new Vector3());
  sunDir->normalize();
  sunTheta = static_cast<float> (acos(MathUtils::clamp(sunDir->z, -1, 1)));
  if (sunDir->z > 0) {
    sunSpectralRadiance = computeAttenuatedSunlight(sunTheta, turbidity);
    // produce color suitable for rendering
    sunColor = RGBSpace::SRGB->convertXYZtoRGB(sunSpectralRadiance->toXYZ()->mul(1
        e-4))->constrainRGB();
  } else {
    sunSpectralRadiance = new ConstantSpectralCurve(0);
  }
  float theta2 = sunTheta * sunTheta;
  float theta3 = sunTheta * theta2;
```

```
float T = turbidity;
float T2 = turbidity * turbidity;
double chi = (4.0 / 9.0 - T / 120.0) * (3.1415 - 2.0 * sunTheta);
zenithY = (4.0453 * T - 4.9710) * tan(chi) - 0.2155 * T + 2.4192;
zenithY *= 1000; // conversion from kcd/m^2 to cd/m^2
zenithx = (0.00165 * theta3 - 0.00374 * theta2 + 0.00208 * sunTheta + 0)
    * T2 + (-0.02902 * theta3 + 0.06377 * theta2 - 0.03202 * sunTheta
    + 0.00394) * T + (0.11693 * theta3 - 0.21196 * theta2 + 0.06052
    * sunTheta + 0.25885);
zenithy = (0.00275 * theta3 - 0.00610 * theta2 + 0.00316 * sunTheta + 0)
    * T2 + (-0.04212 * theta3 + 0.08970 * theta2 - 0.04153 * sunTheta
    + 0.00515) * T + (0.15346 * theta3 - 0.26756 * theta2 + 0.06669
    * sunTheta + 0.26688);

perezY[0] = 0.17872 * T - 1.46303;
perezY[1] = -0.35540 * T + 0.42749;
perezY[2] = -0.02266 * T + 5.32505;
perezY[3] = 0.12064 * T - 2.57705;
perezY[4] = -0.06696 * T + 0.37027;

perezx[0] = -0.01925 * T - 0.25922;
perezx[1] = -0.06651 * T + 0.00081;
perezx[2] = -0.00041 * T + 0.21247;
perezx[3] = -0.06409 * T - 0.89887;
perezx[4] = -0.00325 * T + 0.04517;

perezy[0] = -0.01669 * T - 0.26078;
perezy[1] = -0.09495 * T + 0.00921;
perezy[2] = -0.00792 * T + 0.21023;
perezy[3] = -0.04405 * T - 1.65369;
perezy[4] = -0.01092 * T + 0.05291;

int w = 32, h = 32;
imageHistogram = RectangularArrays::ReturnRectangularFloatArray(w, h);
colHistogram = new float[w];
float du = 1.0f / w;
float dv = 1.0f / h;
for (int x = 0; x < w; x++) {
  for (int y = 0; y < h; y++) {
    float u = (x + 0.5f) * du;
    float v = (y + 0.5f) * dv;
    Color *c = getSkyRGB(getDirection(u, v));
    imageHistogram[x][y] = c->getLuminance() * static_cast<float> (sin(3.1415 * v));
    if (y > 0) {
      imageHistogram[x][y] += imageHistogram[x][y - 1];
```

```
      }
    }
    colHistogram[x] = imageHistogram[x][h - 1];
    if (x > 0) {
      colHistogram[x] += colHistogram[x - 1];
    }
    for (int y = 0; y < h; y++) {
      imageHistogram[x][y] /= imageHistogram[x][h - 1];
    }
  }
  for (int x = 0; x < w; x++) {
    colHistogram[x] /= colHistogram[w - 1];
  }
  jacobian = static_cast<float> (2 * 3.1415 * 3.1415) / (w * h);
}

bool SunSkyLight::update(ParameterList *pl, HprAPI *api) {
  Vector3 *up = pl->getVector("up", 0);
  Vector3 *east = pl->getVector("east", 0);
  if (up != 0 && east != 0) {
    basis = OrthoNormalBasis::makeFromWV(up, east);
  } else if (up != 0) {
    basis = OrthoNormalBasis::makeFromW(up);
  }
  numSkySamples = pl->getInt("samples", numSkySamples);
  sunDirWorld = pl->getVector("sundir", sunDirWorld);
  turbidity = pl->getFloat("turbidity", turbidity);
  groundExtendSky = pl->getBoolean("ground.extendsky", groundExtendSky);
  groundColor = pl->getColor("ground.color", groundColor);
  // recompute model
  initSunSky();
  return true;
}

Color *SunSkyLight::getSkyRGB(Vector3 *dir) {
  if (dir->z < 0 && !groundExtendSky) {
    return groundColor;
  }
  if (dir->z < 0.001f) {
    dir->z = 0.001f;
  }
  dir->normalize();
  double theta = acos(MathUtils::clamp(dir->z, -1, 1));
  double gamma = acos(MathUtils::clamp(Vector3::dot(dir, sunDir), -1, 1));
  double x = perezFunction(perezx, theta, gamma, zenithx);
```

```
  double y = perezFunction(perezy, theta, gamma, zenithy);
  double Y = perezFunction(perezY, theta, gamma, zenithY) * 1e-4;
  XYZColor *c = ChromaticitySpectrum::get(static_cast<float> (x), static_cast<float> (y));
  float X = static_cast<float> (c->getX() * Y / c->getY());
  float Z = static_cast<float> (c->getZ() * Y / c->getY());
  return RGBSpace::SRGB->convertXYZtoRGB(X, static_cast<float> (Y), Z);
}

int SunSkyLight::getNumSamples() {
  return 1 + numSkySamples;
}

void SunSkyLight::getPhoton(double randX1, double randY1, double randX2, double randY2,
    Point3 *p, Vector3 *dir, Color *power) {
  // TODO: not implemented
}

float SunSkyLight::getPower() {
  return 0;
}

void SunSkyLight::getSamples(ShadingState *state) {
  if (Vector3::dot(sunDirWorld, state->getGeoNormal()) > 0 && Vector3::dot(sunDirWorld, state
      ->getNormal()) > 0) {
    LightSample *dest = new LightSample();
    dest->setShadowRay(new Ray(state->getPoint(), sunDirWorld));
    dest->getShadowRay()->setMax(0x1.fffffeP+127f);
    dest->setRadiance(sunColor, sunColor);
    dest->traceShadow(state);
    state->addSample(dest);
  }
  int n = state->getDiffuseDepth() > 0 ? 1 : numSkySamples;
  for (int i = 0; i < n; i++) {
    // random offset on unit square, we use the infinite version of getRandom because the light
        sampling is adaptive
    double randX = state->getRandom(i, 0, n);
    double randY = state->getRandom(i, 1, n);

    int x = 0;
    while (randX >= colHistogram[x] && x < sizeof(colHistogram) / sizeof(colHistogram[0]) - 1) {
      x++;
    }
    float *rowHistogram = imageHistogram[x];
    int y = 0;
    while (randY >= rowHistogram[y] && y < sizeof(rowHistogram) / sizeof(rowHistogram[0]) -
```

```
        1) {
      y++;
    }
    // sample from (x, y)
    float u = static_cast<float> ((x == 0) ? (randX / colHistogram[0]) : ((randX - colHistogram[
        x - 1]) / (colHistogram[x] - colHistogram[x - 1])));
    float v = static_cast<float> ((y == 0) ? (randY / rowHistogram[0]) : ((randY -
        rowHistogram[y - 1]) / (rowHistogram[y] - rowHistogram[y - 1])));

    float px = ((x == 0) ? colHistogram[0] : (colHistogram[x] - colHistogram[x - 1]));
    float py = ((y == 0) ? rowHistogram[0] : (rowHistogram[y] - rowHistogram[y - 1]));

    float su = (x + u) / sizeof(colHistogram) / sizeof(colHistogram[0]);
    float sv = (y + v) / sizeof(rowHistogram) / sizeof(rowHistogram[0]);
    float invP = static_cast<float> (sin(sv * 3.1415)) * jacobian / (n * px * py);
    Vector3 *localDir = getDirection(su, sv);
    Vector3 *dir = basis->transform(localDir, new Vector3());
    if (Vector3::dot(dir, state->getGeoNormal()) > 0 && Vector3::dot(dir, state->getNormal()) >
        0) {
      LightSample *dest = new LightSample();
      dest->setShadowRay(new Ray(state->getPoint(), dir));
      dest->getShadowRay()->setMax(0x1.fffffeP+127);
      Color *radiance = getSkyRGB(localDir);
      dest->setRadiance(radiance, radiance);
      dest->getDiffuseRadiance()->mul(invP);
      dest->getSpecularRadiance()->mul(invP);
      dest->traceShadow(state);
      state->addSample(dest);
    }
  }
}

PrimitiveList *SunSkyLight::getBakingPrimitives() {
  return 0;
}

int SunSkyLight::getNumPrimitives() {
  return 1;
}

float SunSkyLight::getPrimitiveBound(int primID, int i) {
  return 0;
}

BoundingBox *SunSkyLight::getWorldBounds(Matrix4 *o2w) {
```

```
  return 0;
}

void SunSkyLight::intersectPrimitive(Ray *r, int primID, IntersectionState *state) {
  if (r->getMax() == 1.0f / 0.0f) {
    state->setIntersection(0);
  }
}

void SunSkyLight::prepareShadingState(ShadingState *state) {
  if (state->includeLights()) {
    state->setShader(this);
  }
}

Color *SunSkyLight::getRadiance(ShadingState *state) {
  return getSkyRGB(basis->untransform(state->getRay()->getDirection()))->constrainRGB();
}

void SunSkyLight::scatterPhoton(ShadingState *state, Color *power) {
  // let photon escape
}

Vector3 *SunSkyLight::getDirection(float u, float v) {
  Vector3 *dest = new Vector3();
  double phi = 0, theta = 0;
  theta = u * 2 * 3.1415;
  phi = v * 3.1415;
  double sin_phi = sin(phi);
  dest->x = static_cast<float> (-sin_phi * cos(theta));
  dest->y = static_cast<float> (cos(phi));
  dest->z = static_cast<float> (sin_phi * sin(theta));
  return dest;
}

Instance *SunSkyLight::createInstance() {
  return Instance::createTemporary(this, 0, this);
}
}
```

Listing A.33: SphereLight.h

```
#pragma once
#include "Color.h"
#include "Point3.h"
#include "HprAPI.h"
```

```
#include "ParameterList.h"
#include "ShadingState.h"
#include "Vector3.h"
#include "OrthoNormalBasis.h"
#include "Solvers.h"
#include "LightSample.h"
#include "Ray.h"
#include "Instance.h"
#include "Sphere.h"
#include "Matrix4.h"
#include <cmath>

namespace hpr.core.light {

  using hpr::HprAPI;
  using hpr::core::Instance;
  using hpr::core::LightSample;
  using hpr::core::LightSource;
  using hpr::core::ParameterList;
  using hpr::core::Ray;
  using hpr::core::Shader;
  using hpr::core::ShadingState;
  using hpr::core::primitive::Sphere;
  using hpr::image::Color;
  using hpr::math::Matrix4;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Point3;
  using hpr::math::Solvers;
  using hpr::math::Vector3;

  class SphereLight : LightSource, Shader {
    private:
    Color *radiance;
    int numSamples;
    Point3 *center;
    float radius;
    float r2;

    public:
    SphereLight();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual int getNumSamples();
    virtual int getLowSamples();
    virtual bool isVisible(ShadingState *state);
    virtual void getSamples(ShadingState *state);
```

```
    virtual void getPhoton(double randX1, double randY1, double randX2, double randY2, Point3
        *p, Vector3 *dir, Color *power);
    virtual float getPower();
    virtual Color *getRadiance(ShadingState *state);
    virtual void scatterPhoton(ShadingState *state, Color *power);
    virtual Instance *createInstance();
  };
}
```

Listing A.34: SphereLight.cpp

```
#include "SphereLight.h"

using hpr::HprAPI;
using hpr::core::Instance;
using hpr::core::LightSample;
using hpr::core::LightSource;
using hpr::core::ParameterList;
using hpr::core::Ray;
using hpr::core::Shader;
using hpr::core::ShadingState;
using hpr::core::primitive::Sphere;
using hpr::image::Color;
using hpr::math::Matrix4;
using hpr::math::OrthoNormalBasis;
using hpr::math::Point3;
using hpr::math::Solvers;
using hpr::math::Vector3;

SphereLight::SphereLight() {
  radiance = Color::WHITE;
  numSamples = 4;
  center = new Point3();
  radius = r2 = 1;
}

bool SphereLight::update(ParameterList *pl, HprAPI *api) {
  radiance = pl->getColor("radiance", radiance);
  numSamples = pl->getInt("samples", numSamples);
  radius = pl->getFloat("radius", radius);
  r2 = radius * radius;
  center = pl->getPoint("center", center);
  return true;
}

int SphereLight::getNumSamples() {
```

```
  return numSamples;
}

int SphereLight::getLowSamples() {
  return 1;
}

bool SphereLight::isVisible(ShadingState *state) {
  return state->getPoint()->distanceToSquared(center) > r2;
}

void SphereLight::getSamples(ShadingState *state) {
  if (getNumSamples() <= 0) {
    return;
  }
  Vector3 *wc = Point3::sub(center, state->getPoint(), new Vector3());
  float l2 = wc->lengthSquared();
  if (l2 <= r2) {
    return; // inside the sphere?
  }
  // top of the sphere as viewed from the current shading point
  float topX = wc->x + state->getNormal()->x * radius;
  float topY = wc->y + state->getNormal()->y * radius;
  float topZ = wc->z + state->getNormal()->z * radius;
  if (state->getNormal()->dot(topX, topY, topZ) <= 0) {
    return; // top of the sphere is below the horizon
  }
  float cosThetaMax = static_cast<float> (sqrt(__max(0, 1 - r2 / Vector3::dot(wc, wc))));
  OrthoNormalBasis *basis = OrthoNormalBasis::makeFromW(wc);
  int samples = state->getDiffuseDepth() > 0 ? 1 : getNumSamples();
  float scale = static_cast<float> (2 * 3.1415 * (1 - cosThetaMax));
  Color *c = Color::mul(scale / samples, radiance);
  for (int i = 0; i < samples; i++) {
    // random offset on unit square
    double randX = state->getRandom(i, 0, samples);
    double randY = state->getRandom(i, 1, samples);

    // cone sampling
    double cosTheta = (1 - randX) * cosThetaMax + randX;
    double sinTheta = sqrt(1 - cosTheta * cosTheta);
    double phi = randY * 2 * 3.1415; //Math.PI;
    Vector3 *dir = new Vector3(static_cast<float> (cos(phi) * sinTheta), static_cast<float> (sin(
        phi) * sinTheta), static_cast<float> (cosTheta));
    basis->transform(dir);
```

```
    // check that the direction of the sample is the same as the normal
    float cosNx = Vector3::dot(dir, state->getNormal());
    if (cosNx <= 0) {
      continue;
    }

    float ocx = state->getPoint()->x - center->x;
    float ocy = state->getPoint()->y - center->y;
    float ocz = state->getPoint()->z - center->z;
    float qa = Vector3::dot(dir, dir);
    float qb = 2 * ((dir->x * ocx) + (dir->y * ocy) + (dir->z * ocz));
    float qc = ((ocx * ocx) + (ocy * ocy) + (ocz * ocz)) - r2;
    double *t = Solvers::solveQuadric(qa, qb, qc);
    if (t == 0) {
      continue;
    }
    LightSample *dest = new LightSample();
    // compute shadow ray to the sampled point
    dest->setShadowRay(new Ray(state->getPoint(), dir));
    // TODO: arbitrary bias, should handle as in other places
    dest->getShadowRay()->setMax(static_cast<float> (t[0]) - 1e-3);
    // prepare sample
    dest->setRadiance(c, c);
    dest->traceShadow(state);
    state->addSample(dest);
  }
}

void SphereLight::getPhoton(double randX1, double randY1, double randX2, double randY2, Point3
    *p, Vector3 *dir, Color *power) {
  float z = static_cast<float> (1 - 2 * randX2);
  float r = static_cast<float> (sqrt(__max(0, 1 - z * z)));
  float phi = static_cast<float> (2 * 3.1415 * randY2);
  float x = r * static_cast<float> (cos(phi));
  float y = r * static_cast<float> (sin(phi));
  p->x = center->x + x * radius;
  p->y = center->y + y * radius;
  p->z = center->z + z * radius;
  OrthoNormalBasis *basis = OrthoNormalBasis::makeFromW(new Vector3(x, y, z));
  phi = static_cast<float> (2 * 3.1415 * randX1);
  float cosPhi = static_cast<float> (cos(phi));
  float sinPhi = static_cast<float> (sin(phi));
  float sinTheta = static_cast<float> (sqrt(randY1));
  float cosTheta = static_cast<float> (sqrt(1 - randY1));
  dir->x = cosPhi * sinTheta;
```

```
  dir->y = sinPhi * sinTheta;
  dir->z = cosTheta;
  basis->transform(dir);
  power->set(radiance);
  power->mul(static_cast<float> (3.1415 * 3.1415 * 4 * r2));
}

float SphereLight::getPower() {
  return radiance->copy()->mul(static_cast<float> (3.1415 * 3.1415 * 4 * r2))->getLuminance
      ();
}

Color *SphereLight::getRadiance(ShadingState *state) {
  if (!state->includeLights()) {
    return Color::BLACK;
  }
  state->faceforward();
  // emit constant radiance
  return state->isBehind() ? Color::BLACK : radiance;
}

void SphereLight::scatterPhoton(ShadingState *state, Color *power) {
  // do not scatter photons
}

Instance *SphereLight::createInstance() {
  return Instance::createTemporary(new Sphere(), Matrix4::translation(center->x, center->y,
      center->z)->multiply(Matrix4::scale(radius)), this);
}
}
```

Listing A.35: PinholeLens.h

```
#pragma once
#include "HprAPI.h"
#include "ParameterList.h"
#include "Ray.h"
#include <cmath>

namespace hpr.core.camera {

  using hpr::HprAPI;
  using hpr::core::CameraLens;
  using hpr::core::ParameterList;
  using hpr::core::Ray;
```

```
  class PinholeLens : CameraLens {
    private:
    float au, av;
    float aspect, fov;
    float shiftX, shiftY;
    void update();

    public:
    PinholeLens();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual Ray *getRay(float x, float y, int imageWidth, int imageHeight, double lensX, double
        lensY, double time);
  };
}
```

Listing A.36: PinholeLens.cpp

```
#include "PinholeLens.h"

using hpr::HprAPI;
using hpr::core::CameraLens;
using hpr::core::ParameterList;
using hpr::core::Ray;

PinholeLens::PinholeLens() {
  fov = 90;
  aspect = 1;
  shiftX = shiftY = 0;
  update();
}

bool PinholeLens::update(ParameterList *pl, HprAPI *api) {
  // get parameters
  fov = pl->getFloat("fov", fov);
  aspect = pl->getFloat("aspect", aspect);
  shiftX = pl->getFloat("shift.x", shiftX);
  shiftY = pl->getFloat("shift.y", shiftY);
  update();
  return true;
}

void PinholeLens::update() {
  au = static_cast<float> (tan(Math::toRadians(fov * 0.5f)));
  av = au / aspect;
}
```

```
Ray *PinholeLens::getRay(float x, float y, int imageWidth, int imageHeight, double lensX, double
    lensY, double time) {
  float du = shiftX - au + ((2.0f * au * x) / (imageWidth - 1.0f));
  float dv = shiftY - av + ((2.0f * av * y) / (imageHeight - 1.0f));
  return new Ray(0, 0, 0, du, dv, -1);
}
}
```

Listing A.37: PhongShader.h

```
#pragma once
#include "Color.h"
#include "HprAPI.h"
#include "ParameterList.h"
#include "ShadingState.h"
#include "OrthoNormalBasis.h"
#include "Vector3.h"
#include "Ray.h"
#include <cmath>

namespace hpr.core.shader {

  using hpr::HprAPI;
  using hpr::core::ParameterList;
  using hpr::core::Ray;
  using hpr::core::Shader;
  using hpr::core::ShadingState;
  using hpr::image::Color;
  using hpr::math::OrthoNormalBasis;
  using hpr::math::Vector3;

  class PhongShader : Shader {
    private:
    Color *diff;
    Color *spec;
    float power;
    int numRays;

    public:
    PhongShader();
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual Color *getRadiance(ShadingState *state);
    virtual void scatterPhoton(ShadingState *state, Color *power);

    protected:
    virtual Color *getDiffuse(ShadingState *state);
```

```
  };
}
```

Listing A.38: PhongShader.cpp

```
#include "PhongShader.h"

using hpr::HprAPI;
using hpr::core::ParameterList;
using hpr::core::Ray;
using hpr::core::Shader;
using hpr::core::ShadingState;
using hpr::image::Color;
using hpr::math::OrthoNormalBasis;
using hpr::math::Vector3;

PhongShader::PhongShader() {
  diff = Color::GRAY;
  spec = Color::GRAY;
  power = 20;
  numRays = 4;
}

bool PhongShader::update(ParameterList *pl, HprAPI *api) {
  diff = pl->getColor("diffuse", diff);
  spec = pl->getColor("specular", spec);
  power = pl->getFloat("power", power);
  numRays = pl->getInt("samples", numRays);
  return true;
}

Color *PhongShader::getDiffuse(ShadingState *state) {
  return diff;
}

Color *PhongShader::getRadiance(ShadingState *state) {
  // make sure we are on the right side of the material
  state->faceforward();
  // setup lighting
  state->initLightSamples();
  state->initCausticSamples();
  // execute shader
  return state->diffuse(getDiffuse(state))->add(state->specularPhong(spec, power, numRays));
}

void PhongShader::scatterPhoton(ShadingState *state, Color *power) {
```

```
    // make sure we are on the right side of the material
    state->faceforward();
    Color *d = getDiffuse(state);
    state->storePhoton(state->getRay()->getDirection(), power, d);
    float avgD = d->getAverage();
    float avgS = spec->getAverage();
    double rnd = state->getRandom(0, 0, 1);
    if (rnd < avgD) {
      // photon is scattered diffusely
      power->mul(d)->mul(1.0f / avgD);
      OrthoNormalBasis *onb = state->getBasis();
      double u = 2 * 3.1415 * rnd / avgD;
      double v = state->getRandom(0, 1, 1);
      float s = static_cast<float> (sqrt(v));
      float s1 = static_cast<float> (sqrt(1.0f - v));
      Vector3 *w = new Vector3(static_cast<float> (cos(u)) * s, static_cast<float> (sin(u)) * s, s1);
      w = onb->transform(w, new Vector3());
      state->traceDiffusePhoton(new Ray(state->getPoint(), w), power);
    } else if (rnd < avgD + avgS) {
      // photon is scattered specularly
      float dn = 2.0f * state->getCosND();
      // reflected direction
      Vector3 *refDir = new Vector3();
      refDir->x = (dn * state->getNormal()->x) + state->getRay()->dx;
      refDir->y = (dn * state->getNormal()->y) + state->getRay()->dy;
      refDir->z = (dn * state->getNormal()->z) + state->getRay()->dz;
      power->mul(spec)->mul(1.0f / avgS);
      OrthoNormalBasis *onb = state->getBasis();
      double u = 2 * 3.1415 * (rnd - avgD) / avgS;
      double v = state->getRandom(0, 1, 1);
      float s = static_cast<float> (pow(v, 1 / (this->power + 1)));
      float s1 = static_cast<float> (sqrt(1 - s * s));
      Vector3 *w = new Vector3(static_cast<float> (cos(u)) * s1, static_cast<float> (sin(u)) * s1, s
          );
      w = onb->transform(w, new Vector3());
      state->traceReflectionPhoton(new Ray(state->getPoint(), w), power);
    }
  }
}
```

Listing A.39: SimpleShader.h

```
#pragma once
#include "HprAPI.h"
#include "ParameterList.h"
#include "ShadingState.h"
```

```
#include "Color.h"
#include <cmath>

namespace hpr.core.shader {

  using hpr::HprAPI;
  using hpr::core::ParameterList;
  using hpr::core::Shader;
  using hpr::core::ShadingState;
  using hpr::image::Color;

  class SimpleShader : Shader {
    public:
    virtual bool update(ParameterList *pl, HprAPI *api);
    virtual Color *getRadiance(ShadingState *state);
    virtual void scatterPhoton(ShadingState *state, Color *power);
  };
}
```

Listing A.40: SimpleShader.cpp

```
#include "SimpleShader.h"

using hpr::HprAPI;
using hpr::core::ParameterList;
using hpr::core::Shader;
using hpr::core::ShadingState;
using hpr::image::Color;

bool SimpleShader::update(ParameterList *pl, HprAPI *api) {
  return true;
}

Color *SimpleShader::getRadiance(ShadingState *state) {
  return new Color(abs(state->getRay()->dot(state->getNormal())));
}

void SimpleShader::scatterPhoton(ShadingState *state, Color *power) {
}
}
```

Bibliography

Tomas Akenine-Möller and Eric Haines. A K Peters, Wellesley, MA, USA, 2nd edition, 2002. ISBN 9781568811826. (Cited on pages 21, 22, 23, 25, 27, 28, 29, 30, 32, 34, 35, 36, 37, 38, 39, 40, and 42.)

Edward Angel. *Interactive Computer Graphics: A Top-Down Approach using OpenGL (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2008. ISBN 978-0-321-54943-0. (Cited on pages 16, 17, 18, 19, 20, and 21.)

Anonymous. Suzanne (blender primitive), undated. URL http://en.wikipedia.org/wiki/Suzanne_(Blender_primitive). Accessed 16 May 2009. (Cited on page 51.)

James Arvo and David Kirk. *A Survey of Ray Tracing Acceleration Techniques*, pages 201–262. Academic Press Ltd., London, UK, UK, 1989. ISBN 0-12-286160-4. (Cited on page 23.)

Ian Ashdown. *Radiosity: a programmer's perspective*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-30488-3. (Cited on page 26.)

Ian Ashdown. Eigenvector radiosity. Master's thesis, Department of Computer Science, University of British Columbia, 2001. URL http://www.cs.ubc.ca/labs/imager/th/2001/Ashdown2001/Ashdown2001.pdf. (Cited on page 28.)

Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 33–40, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-739-7. (Cited on page 33.)

ATI. Ati stream computing technical overview. Technical report, ATI, 2008. URL http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf. (Cited on page 6.)

Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: http://doi.acm.org/10.1145/1088149.1088174. (Cited on page 12.)

Jiri Bittner and Jan Prikryl. Exact regional visibility using line space partitioning. Technical Report TR-186-2-01-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, March 2001. URL http://www.cg.tuwien.ac.at/research/publications/2001/Bittner-2001-ERV/. human contact: technical-report@cg.tuwien.ac.at. (Cited on page 39.)

Jiri Bittner, Peter Wonka, and Michael Wimmer. Visibility preprocessing for urban scenes using line space subdivision. In *PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, page 276, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1227-5. (Cited on page 39.)

Jim Blinn. *Jim Blinn's corner: a trip down the graphics pipeline*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1-55860-387-5. (Cited on page 36.)

Jehoshua Bruck, Danny Dolev, Ching Ho, Marcel Rosu, and Ray Strong. Efficient message passing interface (mpi) for parallel computing on clusters of workstations. Technical report, Ithaca, NY, USA, 1995. (Cited on page 15.)

Ian Buck. Gpu computing with nvidia cuda. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1281500.1281647. (Cited on pages 4 and 8.)

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1186562.1015800. (Cited on pages 5 and 7.)

Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. A. K. Peters, Ltd., Wellesley, MA, USA, 2002. ISBN 9-781568-811796. (Cited on page 30.)

Barbara M. Chapman. Parallel application development with the hybrid mpi+openmp programming model. In *Proceedings of the 9th European PVM/MPI Users' Group*

Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, page 13, London, UK, 2002. Springer-Verlag. ISBN 3-540-44296-0. (Cited on pages 13 and 15.)

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2008.05.014. (Cited on pages 8 and 9.)

Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993. ISBN 0-12-178270-0. (Cited on pages 26 and 27.)

Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003. ISSN 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2003.1207447. (Cited on pages 33 and 34.)

Wagner T. Correa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. ISBN 1-58113-579-3. (Cited on page 20.)

Thomas W. Crockett. An introduction to parallel rendering. *Parallel Comput.*, 23 (7):819–843, 1997. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/S0167-8191(97)00028-8. (Cited on page 16.)

Timothy Davis and Erik Reinhard. *Coherence in Ray Tracing*, pages 153–184. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1-56881-179-9. (Cited on page 24.)

Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The openrt application programming interface - towards a common api for interactive ray tracing. In *Proceedings of the 2003 OpenSG Symposium*, 2003. Available at http://www.openrt.de. (Cited on page 1.)

Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: http://doi.acm.org/10.1145/1513895.1513907. (Cited on pages 4, 5, 6, and 9.)

Alejandro Duran, Marc Gonzalez, and Julita Corbalán. Automatic thread distribution for nested parallelism in openmp. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 121–130, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: http://doi.acm.org/10.1145/1088149.1088166. (Cited on pages 12 and 13.)

Frédo Durand and Julie Dorsey. Interactive tone mapping. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 219–230, London, UK, 2000. Springer-Verlag. ISBN 3-211-83535-0. (Cited on page 26.)

Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006. ISBN 1568813074. (Cited on page 26.)

Stefan Eilemann. An analysis of parallel rendering systems, January 2007. URL http://www.equalizergraphics.com/documents/ParallelRenderingSystems.pdf. (Cited on page 1.)

Stefan Eilemann and Renato Pajarola. The equalizer parallel rendering framework. Technical Report IFI-2007.06, Department of Informatics, Unviersity of Zürich, 2007. (Cited on page 1.)

Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. Equalizer: a scalable parallel rendering framework. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–14, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1508044.1508088. (Cited on page 1.)

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. (Cited on page 14.)

Markus Giegl and Michael Wimmer. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum*, 26(1):46–49, March 2007. ISSN 0167-7055. URL http://www.cg.tuwien.ac.at/research/publications/2007/GIEGL-2007-UNP/. (Cited on page 41.)

Andrew S. Glassner, editor. Academic Press Ltd., London, UK, UK, 1989. ISBN 0-12-286160-4. (Cited on page 23.)

Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for gpu volume raycasting. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–166, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1508044.1508045. (Cited on page 25.)

Ziyad S. Hakura and John M. Snyder. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 289–300, London, UK, 2001. Springer-Verlag. ISBN 3-211-83709-4. (Cited on page 26.)

Jason Hale. Workshop: introduction to parallel programming in mpi and c. In *MSCCC '04: Proceedings of the 2nd annual conference on Mid-south college computing*, pages 2–2, Little Rock, Arkansas, United States, 2004. Mid-South College Computing Conference. (Cited on page 14.)

Pat Hanrahan. *A Survey of Ray-surface Intersection Algorithms*, pages 79–119. Academic Press Ltd., London, UK, UK, 1989. ISBN 0-12-286160-4. (Cited on pages 22 and 32.)

Mark Harris. Cuda: performance tips and tricks. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 9, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1281500.1281650. (Cited on pages 5, 8, and 9.)

Mark J Harris and Anselmo Lastra. Real-time cloud rendering. Technical report, Chapel Hill, NC, USA, 2001. (Cited on page 26.)

Herman J. Haverkort. *Results on Geometric Networks and Data Structures*. PhD thesis, Utrecht University, Netherlands, 2004. URL http://www.library.uu.nl/digiarchief/dip/diss/2004-0506-101707/c3.pdf. (Cited on page 31.)

Paul S. Heckbert. *A Minimal Ray Tracer*, pages 375–381. Academic Press Professional, Inc., San Diego, CA, USA, 1994. ISBN 0-12-336155-9. URL www.graphicsgems.org. (Cited on page 22.)

Mike Houston. Stream computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–37, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1401132.1401151. (Cited on page 4.)

Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: A scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive*

techniques, pages 129–140, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383272. (Cited on pages 1 and 16.)

Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566639. (Cited on page 1.)

Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–10, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1508044.1508087. (Cited on page 1.)

Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001. ISBN 1-56881-147-0. (Cited on page 26.)

Woo-Chul Jeun, Yang-Suk Kee, Soonhoi Ha, and Changdon Kee. Overcoming performance bottlenecks in using openmp on smp clusters. *Parallel Comput.*, 34(10): 570–592, 2008. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2008.06.002. (Cited on page 13.)

James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: http://doi.acm.org/10.1145/15922.15902. (Cited on pages 22 and 23.)

George M. Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0521817544. (Cited on page 15.)

Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, and Brian Towles. Imagine: Signal and image processing using streams. In *Hot Chips*, number 12, 2000. URL http://cva.stanford.edu/imagine. (Cited on page 3.)

The OpenCL Specification. Khronos Group, February 2009. URL http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf. (Cited on page 10.)

David Kirk. Nvidia cuda software and gpu parallel computing architecture. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 103–104, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: http://doi.acm.org/10.1145/1296907.1296909. (Cited on pages 7 and 8.)

Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Hardware-accelerated from-region visibility using a dual ray space. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 205–216, London, UK, 2001. Springer-Verlag. ISBN 3-211-83709-4. (Cited on page 39.)

Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127, New York, NY, USA, 2003. ACM. ISBN 1-58113-661-7. doi: http://doi.acm.org/10.1145/777412.777433. (Cited on page 12.)

Jaroslav Křivánek. Global illumination with monte carlo ray tracing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–25, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1401132.1401214. (Cited on pages 21, 23, and 25.)

Calvin Lin and Lawrence Snyder. Addison Wesley, Boston, MA, USA, 2009. ISBN 9780321549426. (Cited on page 11.)

Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383274. (Cited on page 23.)

Dani Lischinski, Zeev Farbman, Matt Uyttendaele, and Richard Szeliski. Interactive local adjustment of tonal values. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 646–653, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179352.1141936. (Cited on page 26.)

Marco Lohse, Florian Winter, Michael Repplinger, and Philipp Slusallek. Network-integrated multimedia middleware (nmm). In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 1081–1084, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-303-7. doi: http://doi.acm.org/10.1145/1459359.1459576. (Cited on page 1.)

David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive*

3D graphics, pages 105–ff., New York, NY, USA, 1995. ACM. ISBN 0-89791-736-7. doi: http://doi.acm.org/10.1145/199404.199422. (Cited on page 37.)

David Luebke, Jonathan Cohen, Rob Heubner, Martin Reddy, Amitabh Varshney, and Benjamin Watson. Advanced issues in level of detail. In *Course 14 notes at SIGGRAPH 2002*, 2002. (Cited on page 41.)

Timothy G. Mattson. How good is openmp. *Sci. Program.*, 11(2):81–93, 2003. ISSN 1058-9244. (Cited on pages 12 and 13.)

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994. ISSN 0272-1716. doi: http://dx.doi.org/10.1109/38.291528. (Cited on page 17.)

Matthias S. Müller. An openmp compiler benchmark. *Sci. Program.*, 11(2):125–131, 2003. ISSN 1058-9244. (Cited on page 14.)

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1401132.1401152. (Cited on pages 7 and 8.)

Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A transparent runtime data distribution engine for openmp. *Sci. Program.*, 8(3):143–162, 2000. ISSN 1058-9244. (Cited on page 12.)

Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 12, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1198555.1198751. (Cited on page 23.)

Haoyu Peng, Hua Xiong, and Jiaoying Shi. Parallel-sg: Research of parallel graphics rendering system on pc-cluster. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 27–33, New York, NY, USA, 2006. ACM. ISBN 1-59593-324-7. doi: http://doi.acm.org/10.1145/1128923.1128929. (Cited on page 16.)

Timothy J. Purcell. *Parallel Ray Tracing on a Chip*, pages 329–336. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1-56881-179-9. (Cited on page 24.)

Timothy J. Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Hanrahan,, Patrick M. (Cited on pages 3 and 4.)

Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566640. (Cited on page 24.)

Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 268, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1198555.1198798. (Cited on page 24.)

Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. ISBN 0071232656. (Cited on page 14.)

Erik Reinhard. *Parallel Global Illumination Algorithms*, pages 89–132. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1-56881-179-9. (Cited on page 25.)

Michael Repplinger, Florian Winter, Marco Lohse, and Philipp Slusallek. Parallel bindings in distributed multimedia systems. In *ICDCSW '05: Proceedings of the Seventh International Workshop on Multimedia Network Systems and Applications*, pages 714–720, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2328-5-07. doi: http://dx.doi.org/10.1109/ICDCSW.2005.107. (Cited on page 2.)

S. Richard, B. Miegemolle, J. M. Garcia, and T. Monteil. Performance of mpi parallel applications. In *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*, page 59, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2703-5. doi: http://dx.doi.org/10.1109/ICSEA.2006.58. (Cited on page 15.)

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345220. (Cited on pages 7 and 9.)

Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, New York, NY, USA, 2000. ACM. ISBN 1-58113-257-3. doi: http://doi.acm.org/10.1145/346876.348237. (Cited on page 16.)

Hanan Samet. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 0123694469. (Cited on page 30.)

Hanan Samet. Spatital data structures. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 1, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1281500.1281632. (Cited on page 30.)

Hanan Samet. Sorting in space: multidimensional, spatial, and metric data structures for computer graphics applications. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–106, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1401132.1401249. (Cited on page 31.)

Vincent Scheib. Introduction to demos & the demo scene, February 2001. URL http://www.gamasutra.com/features/20010216/scheib_01.htm. (Cited on page 24.)

Robert A. Schumacker, Brigitta Brand, Maurice G. Gilliland, and Werner H. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AD0700375, Air Force Human Resources Laboratory, Training Research Division, September 1969. URL http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0700375. (Cited on page 32.)

Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003. ISBN 1568811985. (Cited on page 23.)

Leon A. Shirman and Salim S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum, Eurographics*, 12(3):261–272, 1993. (Cited on page 36.)

Francois X. Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994. ISBN 1558602771. (Cited on pages 26 and 27.)

Philipp Slusallek, Peter Shirley, William R. Mark, Gordon Stoll, and Ingo Wald. Rendering massive models. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 16, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1198555.1198755. (Cited on page 25.)

Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Sci. Program.*, 9(2,3):83–98, 2001. ISSN 1058-9244. (Cited on pages 13 and 14.)

Brian Smits. Efficiency issues for ray tracing. *J. Graph. Tools*, 3(2):1–14, 1998. ISSN 1086-7651. (Cited on page 23.)

Scott Spetka, Haris Hadzimujic, Stephen Peek, and Christopher Flynn. High productivity languages for parallel programming compared to mpi. In *HPCMP-UGC '08: Proceedings of the 2008 DoD HPCMP Users Group Conference*, pages 413–417, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3515-9. doi: http://dx.doi.org/10.1109/DoD.HPCMP.UGC.2008.41. (Cited on page 15.)

Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 377–388, London, UK, 2000. Springer-Verlag. ISBN 3-211-83535-0. (Cited on page 26.)

John E. Stone, Jan Saam, David J. Hardy, Kirby L. Vandivort, Wen mei W. Hwu, and Klaus Schulten. High performance computation and interactive display of molecular orbitals on gpus and multi-core cpus. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 9–18, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: http://doi.acm.org/10.1145/1513895.1513897. (Cited on pages 5 and 9.)

Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 93–102, London, UK, 1997. Springer-Verlag. ISBN 3-211-83001-4. (Cited on page 26.)

Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Department of Computer Science, University of Berkeley, 1992. (Cited on page 37.)

Seth J. Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 239–246, New York, NY, USA, 1993. ACM. ISBN 0-89791-601-8. doi: http://doi.acm.org/10.1145/166117.166148. (Cited on page 37.)

Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 61–70, New York, NY, USA, 1991. ACM. ISBN 0-89791-436-8. doi: http://doi.acm.org/10.1145/122718.122725. (Cited on page 37.)

Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In D. Duke

and R. Scopigno, editors, *STAR Proceedings of Eurographics 2001*, Manchester, UK, September 2001. Eurographics Association. (Cited on page 24.)

Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Theresa-Marie Rhyne Alan Chalmers, editor, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001a. URL http://graphics.cs.uni-sb.de/Publications/2001/InteractiveRenderingWithCoherentRayTracing.pdf. (Cited on page 24.)

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 277–288, London, UK, 2001b. Springer-Verlag. ISBN 3-211-83709-4. URL http://graphics.cs.uni-sb.de/Publications/2001/egrw2001.pdf. (Cited on page 24.)

Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive global illumination in complex and highly occluded environments. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, pages 74–81, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 3-905673-03-7. (Cited on page 25.)

Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2005.79. Member-Slusallek,, Philipp. (Cited on page 23.)

Greg Ward. Irradiance caching algorithm. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 3, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1281500.1281619. (Cited on page 27.)

Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358876.358882. (Cited on pages 21, 22, and 23.)

Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 67–77, New York, NY, USA, 2006. ACM. ISBN 3-905673-37-1. doi: http://doi.acm.org/10.1145/1283900.1283912. (Cited on page 23.)

Hansong Zhang, Dinesh Manocha, Tom Hudson, and III Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: http://doi.acm.org/10.1145/258734.258781. (Cited on page 39.)

Sergey Zhukov, Andrei Iones, and Grigorij Kronin. An ambient light illumination model. In George Drettakis and Nelson L. Max, editors, *Rendering Techniques*, pages 45–56. Springer, 1998. ISBN 3-211-83213-0. (Cited on page 24.)

Index