Distributed Ray Tracing In An Open Source Environment (Work In Progress)

Gunnar Johansson* Linköping University
Ola Nilsson† Linköping University
Andreas Söderström‡ Linköping University
Ken Museth§ Linköping University

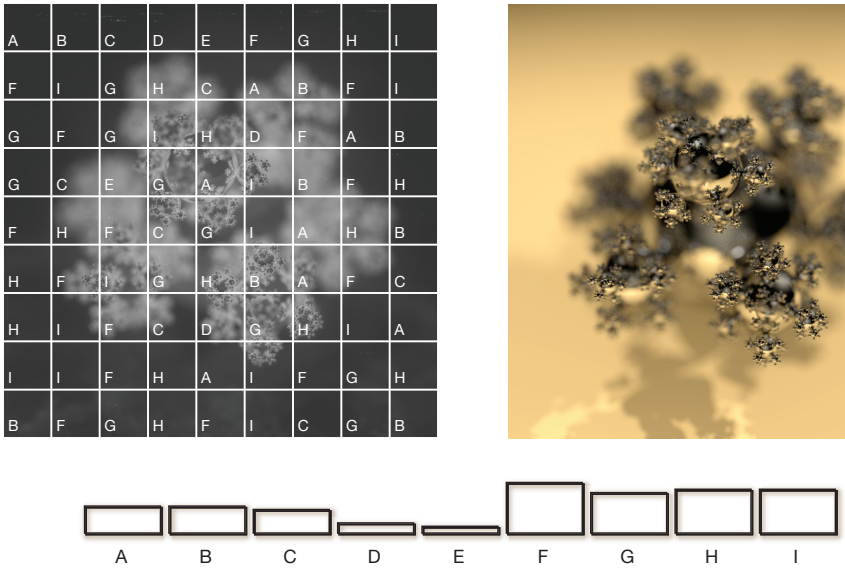

Figure 1: Network distributed rendering of a scene with simulated depth of field. Left: The partitioning of the scene and the node assignment is shown, the grey scale denotes complexity of the rendering measured in time spent per pixel. Right: The ray traced final image, note that a large amount of samples per pixel are needed to sufficiently sample this scene (512 rays per pixel). Below: Bar-chart depicting the load balancing measured in number of buckets rendered: A: Athlon XP 2.1GHz, B: P4 1.7GHz, C: Celeron 2.8GHz, D: PIII 700MHz, E: PIII 450MHz, F: Athlon MP 2.1GHz, G: Athlon MP 2.1GHz, H: PPC 970 2.5 GHz, I: Opteron 2.4 GHz.

Abstract

We present work in progress on concurrent ray tracing with distributed computers using "off-the-shelf" open source software. While there exists numerous open source ray tracers, very few offer support for state-of-the-art concurrent computing. However, it is a well known fact that ray tracing is computationally intensive and yet prevails as the preferred algorithm for photorealistic rendering. Thus, the current work is driven by a desire for a simple programming strategy (or recipe) that allows pre-existing ray tracing code to be parallelized on a heterogenous cluster of available office computers - strictly using open source components. Simplicity, stability, efficiency and modularity are the driving forces for this engineering project, and as such we do not claim any novel research contributions. However, we stress that this project grew out of a real-world need for a render cluster in our research group, and consequently our solutions have a significant practical value. In fact some of our results show a close to optimal speedup when considering the relative performances of each node. In this systems paper we aim at sharing these solutions and experiences with other members of the graphics community.

CR Categories: I.3.2 [Computer Graphics]: Graphic Systems—Distributed/Network Graphics D.1.3 [Programming Techniques]: Concurrent Programming—Distributed Programming

Keywords: distributed ray tracing, render farm, open source

*e-mail: gunjo@itn.liu.se
†e-mail: olani@itn.liu.se
‡e-mail: andso@itn.liu.se
§e-mail:kenmu@itn.liu.se

1 Previous work

Rendering of images lies at the very heart of computer graphics and the increased desire for photorealism often creates a computational bottleneck in image production pipelines. As such the subfield of global illumination has been the focus of numerous previous publications, most of which are beyond the scope of this paper. To mention some of the most important; Ray tracing [Appel 1968; Whitted 1980], beam tracing [Heckbert and Hanrahan 1984], cone tracing [Amanatides 1984], radiosity [Goral et al. 1984], path tracing [Kajiya 1986], metropolis light transport [Veach and Guibas 1997], and photon mapping [Jensen 1996]. While some of these techniques offer better performance than others a prevailing problem seems to be that they are computationally intensive. In this paper we have chosen to focus on the popular ray tracing technique extended with photon mapping. However, even this relatively fast method can crawl to a halt when effects like depth of field or caustics are added. Unlike the usual pin-hole camera, realistic camera models usually require a significant increase in the amount of samples. The same is true when ray tracing caustics of translucent materials. Especially the latter has been a major issue in our group, since we are conducting research on large-scale fluid animations.

The algorithms constituting ray tracing and photon mapping are of-

ten referred to as being "embarrassingly parallel". Nevertheless a large body of work has been devoted to this topic [Lefer 1993; Freisleben et al. 1997; Stone 1998; Lee and Lim 2001], but relatively little has found its way into open source systems. *Yafray*[1] has some seemingly unstable support for multi-threading, but currently no distributed rendering capabilities. *POV-Ray*[2] has some support for distributed computing through unofficial patches. Finally, for *Blender*[3] some unmaintained patches and utilities appear to provide basic network rendering. PBRT[4] is another ray tracer; without distribution capabilites but with excellent documentation and modular code design. Given the state of the aforementioned distributed systems we choose - in the spirit of simplicity and modularity - to work with PBRT as opposed to competing ray tracers. We then extend PBRT with distribution capabilities which is straightforward given its modular design.

2 System

Two fundamentally different strategies exist for concurrent computing. The first approach is *threading* which involves spawning multiple local threads (i.e. "lightweight processes") sharing the same execution and memory space. In the context of ray tracing these threads can then cast rays in parallel. The second strategy is to employ *message passing*, which provides parallelism by communication between several running processes each having their individual execution and memory space. The former is more efficient when considering shared memory architectures, but is obviously not extendable to clusters of computers. For this reason, we use a message passing technique which has optimized strategies for both shared memory systems and clusters.

2.1 Software and Implementation

We have developed a modular distributed rendering system based on simple modifications of pre-existing open source components. The system can be configured to run on almost any computer hardware using very little effort, creating a powerful rendering cluster at virtually no cost. The core component of our system is the "physically based ray tracer", *PBRT*, by Pharr and Humpreys [2004]. We have extended PBRT with OpenMPI [Gabriel et al. 2004] and LAM/MPI [Burns et al. 1994; Squyres and Lumsdaine 2003] to support concurrent computations on both shared memory and distributed architectures. Brief descriptions of these core components follow:

- PBRT is open source for non-commercial use and offers an advanced light simulation environment. It deploys a strictly modular design and the source code is well documented.
- MPI (Message Passing Interface) [Forum 1994] is the de facto standard for distributed scientific computing. The implementations we used are OpenMPI and LAM/MPI which offer full compliancy to the MPI-2 standard.

To simplify the administration of our cluster, we use the *warewulf cluster solution*[5]. The warewulf server makes it possible for nodes to boot off the network while automatically downloading a specialized Linux configuration. Using this solution, virtually any machine connected to the network can be rebooted into the cluster and automatically register with the server. In addition, it is easy to maintain an up-to-date installation across the nodes and poll CPU and memory usage.

Following the modular design of PBRT, our implementation is based on dynamically loaded modules. This simplifies the network abstraction layer significantly. PBRT is executed using the MPI runtime environment on every node in the network. Then, each node in turn loads different modules depending on its role. With this setup we have implemented a star shaped rendering cluster which has one master node that is connected to all the render nodes, see figure 2. The purpose of the master node is to distribute work, collect data, and assemble the final image. Since the complexity of different parts in the image is not known prior to rendering, the work distribution is a non-trivial problem, which is further complicated by the heterogeneity of the network itself. We chose to implement an automatic load balancing scheme where nodes query the master for tasks as the rendering progresses. First, the master partitions the rendering work into buckets, for example blocks of pixels or sets of samples. Then, each rendering node is assigned a bucket in sequential order. As soon as a node completes rendering, the finished bucket is sent to the master node for compositing. If there are unfinished buckets, the master returns a new bucket assignment to the render node and the process loops. This simple scheme is summarized in algorithm 1 and 2 for the master and render nodes, respectively. This strategy of dynamic task assignment is expected to automatically balance the load between nodes of different performance, providing good utilization of the assigned render nodes.

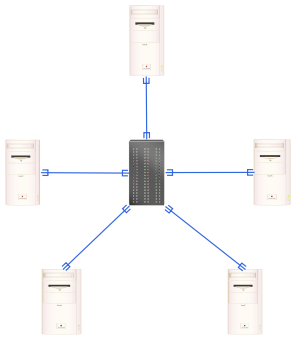

Figure 2: We use a simple star shaped network layout. The master node is shaded black and the render nodes white.

```
1:  Compute the buckets for distribution
2:  Assign initial buckets to the nodes in sequential order
3:  while there are unfinished buckets do
4:      Wait for a bucket
5:      Add received bucket to image
6:      if there are unassigned buckets then
7:          Assign the next unassigned bucket to sending node
8:      else
9:          Send terminating signal to sending node
10:     end if
11: end while
```

Algorithm 1: RENDERMASTER()

In practice, the implementation consists of a few lines of changes in the PBRT core files and additional modules that implement our distribution strategies outlined in algorithm 1 and 2. Our implementation currently supports distribution of either:

[1] http://www.yafray.org
[2] http://www.povray.org
[3] http://www.blender3d.org
[4] http://www.pbrt.org
[5] http://www.warewulf-cluster.org

```
 1: Initialize sampler and preprocess the scene
 2: Wait for initial bucket assignment
 3: while node is assigned a bucket do
 4:     while bucket is unfinished do
 5:         Fetch a sample from sampler and trace ray
 6:         Add sample result to bucket
 7:     end while
 8:     Send finished bucket to master
 9:     Wait for new bucket assignment or termination signal
10: end while
```

Algorithm 2: RENDERNODE()

Pixel blocks consisting of $\{R,G,B,A,weight\}$ tuples.

Sets of samples consisting of $\{x,y,R,G,B,A\}$ tuples.

Photons consisting of light samples $\{x,y,z,R,G,B,n_x,n_y,n_z\}$.

where capital letters denotes the usual color channels, lower case is used for coordinates and n_i denotes normal vector components. When distributing blocks of pixels, the render nodes will return a block of finalized pixels to the master upon completion. This leads to low, fixed bandwidth demands that are proportional to the block size. However, this also requires that the nodes sample in a padded region around the block for correct filtering of the boundary pixels. This means more samples in total, introducing an overhead growing with smaller block sizes. If we assume that the block is quadratic, with side l, and the filter has a filtering radius of r, then the overhead is given by $4rl + 4r^2$. In practice this means that a block size of 100×100 together with a typical filter radius of 2 already gives an overhead of $4 \times 2 \times 100 + 4 \times 2^2 = 816$ pixels out of 10.000, or roughly 8 % per block. This overhead needs to be carefully weighted against the better load balancing achieved by smaller block sizes. For example a block size of 10×10 pixels with filter radius 2 introduces an overhead of 96 % per block.

Adversely, the distribution of samples does not introduce a similar computational overhead, since the samples are independent of each other. However, this approach leads to higher bandwidth demands. More specifically, the bandwidth required for sample distribution compared with pixel distribution is proportional to the number of samples per pixel. For some scenes this can be a considerable number (see for example figure 1).

The two previous distribution strategies deals exclusively with the casting of rays within the ray tracer. However, the celebrated photon-map extension is also easily distributed, with nearly the exact same code. We have parallelized the pre-proccessing step which samples light at discrete points in the scene; the light estimation can be parallelized using one of the two aforementioned modules.

2.2 Hardware

Currently, our cluster is composed of existing desktop computers used by the members of our group, in addition to a collection of very old retired computers collected in our department. The performance ranges from a Pentium III 450 MHz to a dual core AMD Opteron 280, giving a very heterogeneous cluster. The majority of the machines are connected through 100 Mbps network cards and a few are equipped with 1000 Mbps cards. Finally, to better evaluate the impact of the very inhomogeneous hardware we cross-test on a strictly homogeneous shared memory system with high network bandwidth and low latency; an SGI Prism with 8 Itanium II 1.5GHz CPUs, 16 GB of RAM and InfiniBand interconnects.

3 Benchmarks

To assess the performance of our render systems we have benchmarked several different tests. By varying both the raytraced scene and the bucket distribution we evaluate the load balancing. Figure 3 shows the result of applying the load balancing scheme as described in algorithm 1 and 2 for both pixel and sample distribution. The pixel distribution strategy is very close to optimal, and clearly outperforms the sample distribution approach. Our tests indicate that this is due to the currently inefficient, but general, computation of the sample positions.

To measure the speedup achieved by the cluster, we measured the rendering times of a benchmark scene at low resolution at each render node. This gives a relative performance measure for the nodes that can be accumulated for a given cluster configuration. Figure 4 shows the speedup achieved by the cluster for respectively pixel and sample distribution, and figure 5 shows one of the corresponding load distributions. Since we can configure the cluster with an arbitrary combination of nodes, we chose to generate four different host-lists and plot the speedups achieved by increasing the number of nodes from each list. As can be surmised from Figure 4, the pixel distribution shows a clear advantage over the sample distributions. This is in agreement with the results shown in Figure 3. However, the pixel distribution deviates from the optimal linear speedup when the cluster reaches an accumulated relative performance of 5-6 units. This illustrates the main weakness of our simple load balancing scheme. Currently, the master assigns buckets to the render nodes in a "first-come-first-served" manner. Consequently it is possible that a slow node is assigned one of the last buckets, making the final rendering time depend on the execution of this single slow render node. Our measurements indeed show that fast nodes typically spend a significant amount of time idle, waiting for the final buckets to complete. A better strategy for load distribution or re-distribution is clearly needed to improve performance. This is left for future work, but we have several promising ideas.

Figure 6 shows a frame from one of our gigantic fluid animations (a water fountain) placed in the "Cornell-box". The fluid interface is represented by a sparse level set data structure and directly raytraced. Note that the complex fluid surface requires a very large amount of recursive steps in the ray tracing to accurately account for the light transmittance. The lighting conditions are very demanding due to a high amount of indirect lighting and color bleeding. The scene is rendered in a resolution of 1600×1600 with photon mapping using the final gather step. We stress that both the ray tracing and photon shooting scale well even for a scene as complex as this.

Our final benchmark is presented in Figure 7. It shows how the rendering times are affected by the number of buckets for a given scene. As expected, the pixel distribution strategy suffers from a large overhead induced by many small buckets, while the rendering times using a sample distribution approach is left almost unaffected. Also note the initial local minima in rendering time for the pixel distribution, caused by the good load balancing resulting from an optimal bucket size.

4 Future Work and Conclusions

As can be seen from figure 7 the input parameters to our system greatly affect performance. A practical extension would be a friendly user interface to the not always intuitive parameter space. The user should only be concerned with adding a scene description to a queue. For this we need to design heuristics or algorithms that from a given set of parameters {image size, scene complexity, cluster size, cluster node performance, *etc.*} finds the optimal settings and forwards an augmented scene description to the system.

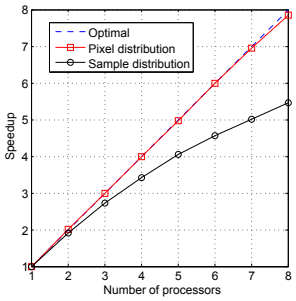

Figure 3: Benchmarking the scene in figure 1 at resolution 900 × 900 with 16 samples per pixel partitioned using 81 buckets. The rendering is performed on an SGI Prism shared memory machine with 8 CPUs. Note how the speedup is close to optimal for pixel distribution, while the sample distribution is suffering from overhead in the computation of the sample positions.

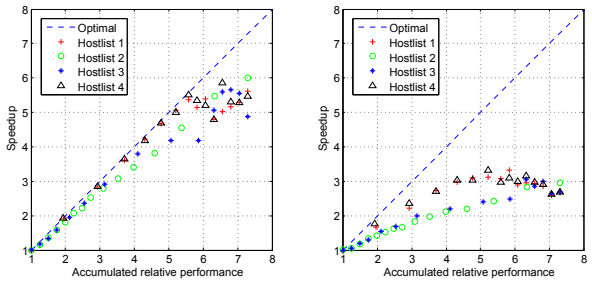

Figure 4: Benchmarking the scene in figure 1 at resolution 900 × 900 with 16 samples per pixel partitioned using 81 buckets. The rendering is performed on a heterogeneous cluster of 15 nodes. Left: Pixel distribution. Right Sample distribution.

Handling errors arising from computations in a distributed environment is a difficult problem. Implementations of MPI such as FT-MPI [Dewolfs et al. 2006] are completely dedicated to building stable systems. For a normal application such as ours the capabilities of OpenMPI are adequate but still require careful design. We wish to make our system as stable as possible.

Any parallel computation is bound to asymptotically saturate the network when using the layout described in 2. So far we have not seen this, but the amount of network traffic for distributing samples is considerable already at 10-20 machines. Thanks to our modular design it is easy to implement different layouts. This will also spread some of the computational overhead for the master node. We attribute the surprisingly bad results for sample distribution (figure 4 right) to the relatively expensive sample position computations. This can be efficiently solved by specialized routines for each sampling pattern. As can be seen from the load balancing graphs, figure 1 and 5 with a complex scene and a heterogenous cluster it is difficult to get really good utilization. Our simple scheme is stable and works asymptotically well (at least for sample distribution) but has practical drawbacks. We want to benchmark balancing schemes with better real world performance such as

- "*Round robin*" network style where each node only communicates with its neighbors and share its current task. This should decrease the network traffic to specific master nodes and distribute the workload better.

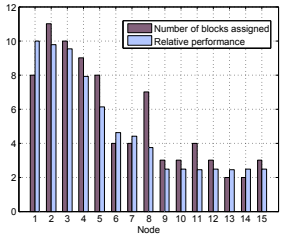

Figure 5: The load balance from the pixel distribution (figure 4 left) using a random host file.

- "*Optimal work load assignment*", currently each work unit is equally large. Using a preprocessing step where the time complexity of the scene is measured, as well as the relative performance of the nodes, a better work load assignment should be feasible. Stability issues when dealing with assumptions on performance and complexity needs to be considered though.
- "*Maximum utilization*", modify the current load balancing scheme so that when idle nodes are detected the distributed work units are recursively split into smaller parts and reassigned. This should ensure good utilization at a small overhead.

In this work, we have described how to use existing open source components to compose a high performance cluster used for distributed ray tracing with little effort and low budget. Initially, we have tested a simple load balancing strategy based on a first-come-first-served scheme. The tests indicate that for homogeneous configurations the resulting speedup is close to optimal. However, for heterogeneous scenes and cluster configurations sub-optimal results were reported stressing a need for more sophisticated strategies, as outlined above.

References

AMANATIDES, J. 1984. Ray tracing with cones. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 129–135.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, vol. 32, 37–45.

BURNS, G., DAOUD, R., AND VAIGL, J. 1994. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, 379–386.

DEWOLFS, D., BROECKHOVE, J., SUNDERAM, V., AND FAGG, G. 2006. Ft-mpi, fault-tolerant metacomputing and generic name services: A case study. *Lecture Notes in Computer Science 4192*, 133–140.

FORUM, M. P. I. 1994. MPI: A message-passing interface standard. Tech. Rep. UT-CS-94-230.

FREISLEBEN, B., HARTMANN, D., AND KIELMANN, T. 1997. Parallel raytracing: A case study on partitioning and scheduling on workstation clusters. In *Hawai'i International Conference on System Sciences (HICSS-30)*, vol. 1, 596–605.

GABRIEL, E., FAGG, G. E., BOSILCA, G., ANGSKUN, T., DONGARRA, J. J., SQUYRES, J. M., SAHAY, V., KAMBADUR, P.,

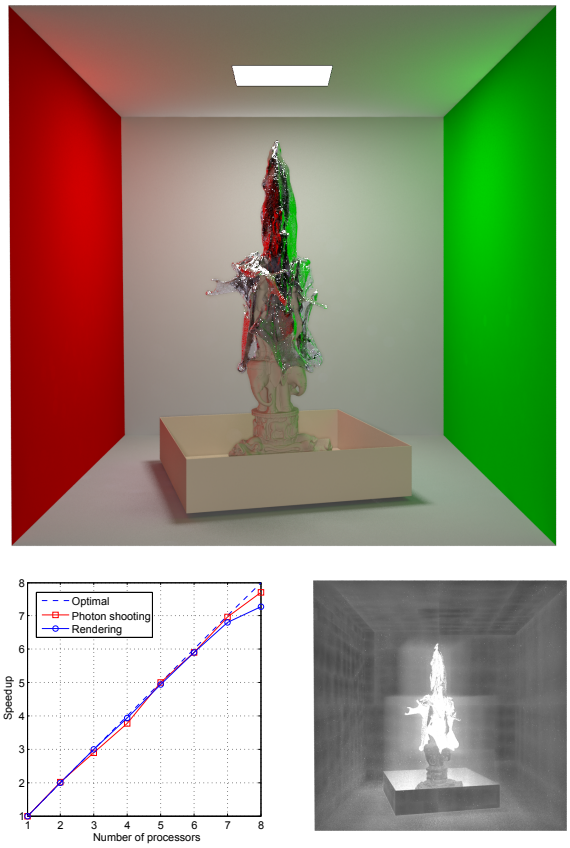

Figure 6: Above: A complex scene from a water simulation inserted into the "*cornell box*" containing several difficult components. Below left: The speedup from distributing the photon mapping preprocessing step and the ray tracing. Below right: Time complexity visualization of the scene measured in time spent per pixel, bright pixels indicate complex regions.

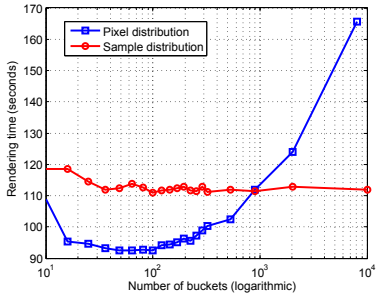

Figure 7: Benchmarking pixel distribution versus sample distribution with an increasing number of buckets.

BARRETT, B., LUMSDAINE, A., CASTAIN, R. H., DANIEL, D. J., GRAHAM, R. L., AND WOODALL, T. S. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97–104.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 213–222.

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, H. Christiansen, Ed., vol. 18, 119–127.

JENSEN, H. W. 1996. Global Illumination Using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, Springer-Verlag/Wien, New York, NY, 21–30.

KAJIYA, J. T. 1986. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 143–150.

LEE, H. J., AND LIM, B. 2001. Parallel ray tracing using processor farming model. *icppw 00*, 0059.

LEFER, W. 1993. An efficient parallel ray tracing scheme for distributed memory parallel computers. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, ACM Press, New York, NY, USA, 77–80.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SQUYRES, J. M., AND LUMSDAINE, A. 2003. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Springer-Verlag, Venice, Italy, no. 2840 in Lecture Notes in Computer Science, 379–387.

STONE, J. E. 1998. *An Efficient Library for Parallel Ray Tracing and Animation*. Master's thesis.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *ACM SIGGRAPH '97*, ACM Press, 65–76.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 255–264.