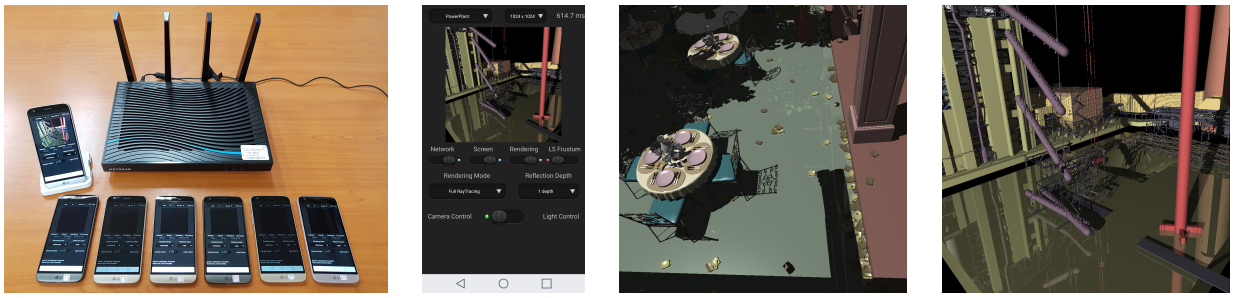
Effective Ray Tracing of Large 3D Scenes through Mobile Distributed Computing

Woong Seo
Dept. of Comp. Sci. and Eng.
Sogang University
Seoul, Korea
wng0620@sogang.ac.kr

Yeonsoo Kim
LG Electronics
Seoul, Korea
yeonsoo1101.kim@lge.com

Insung Ihm ✉
Dept. of Comp. Sci. and Eng.
Sogang University
Seoul, Korea
ihm@sogang.ac.kr



(a) Tested mobile cluster system (b) Master device (c) San Miguel (10,501K) (d) Power Plant (12,749K)

Figure 1: Mobile distributed GPU ray tracing. Through a master/slave system performing tile-based rendering, in which each slave machine keeps a full copy of size-reduced rendering data on limited graphics memory, we were able to achieve a good efficiency of distributed ray tracing in a mobile cluster with six slaves.

ABSTRACT

Ray tracing large-scale 3D scenes at interactive frame rates is a challenging problem on mobile devices. In this paper, we present a mobile ray tracing system that aims to render large scenes with many millions of triangles at interactive speeds on a small-scale mobile cluster. To mitigate performance degradation due to excessive data communication on mobile and wireless networks with still high latency, we employ a tile-based rendering strategy where each participating mobile device keeps an entire copy of the necessary rendering data. To realize such a system, we compress the 3D scene data to a size loadable into graphics memory, which enables an effective mobile GPU ray tracing. Also, by using a careful interaction scheme between the master and slave devices in the mobile cluster, we enhance the efficiency of the mobile distributed GPU ray tracing markedly.

CCS CONCEPTS

• **Human-centered computing** → **Mobile computing**; • **Computing methodologies** → **Ray tracing**; **Distributed algorithms**;

KEYWORDS

Mobile GPU ray tracing, mobile distributed computing, large-scale 3D scene, kd-tree compression, master/slave system

ACM Reference format:
Woong Seo, Yeonsoo Kim, and Insung Ihm ✉. 2017. Effective Ray Tracing of Large 3D Scenes through Mobile Distributed Computing. In *Proceedings of SA '17 Symposium on Mobile Graphics & Interactive Applications, Bangkok, Thailand, November 27-30, 2017 (SA '17 MGIA)*, 5 pages.
https://doi.org/10.1145/3132787.3139206

SA '17 MGIA, November 27-30, 2017, Bangkok, Thailand

ACM ISBN 978-1-4503-5410-3/17/11.
https://doi.org/10.1145/3132787.3139206

1 INTRODUCTION

Distributed/parallel computing has long been an effective tool to visualize large-scale datasets which require substantial computational resources for interactive manipulation. Recent advances in mobile technology suggest to explore mobile clusters, composed of ubiquitous smartphones and tablets connected through mobile and wireless communication networks, for solving large-scale and grand-challenge problems (refer to [Arslan et al. 2015] for a survey of some mobile cluster systems). While previous results on interactive ray tracing in distributed systems usually considered networked workstations and PCs (for instance, [Wald et al. 2003]), very few work has been reported that exploits mobile devices.

In an effort to investigate the feasibility of mobile cluster computing in interactive visualization of large-scale 3D scenes, we have developed a ray tracing system that aims to render large scenes with more than ten million triangles at interactive frame rates on a small-sized cluster made of up to a dozen mobile devices. Some of the main features of our rendering system are as follows. First, distributing bulky 3D scene data dynamically among processing units

during rendering is not appropriate for the current mobile cluster system due to its high transmission latency. Thus we adopted a tile-based master/slave rendering scheme where each slave mobile device, storing a full copy of all necessary scene data, repeatedly renders the tile area assigned to it, and transmits the 2D tile image back to the master mobile device.

Second, in order for a mobile device to be able to load as large a 3D scene as possible, we applied a space-efficient scene representation technique that substantially reduces the size of rendering data that must be handled. In particular, we extended the kd-tree representation scheme, proposed by Choi et al. [Choi et al. 2013], to further reduce the memory requirements without any serious degradation of rendering performance. Third, considering that the data communication between mobile devices is not so efficient as in the PC cluster environment, we paid special attention to minimizing performance drop due to ineffective interactions between the master and slaves.

2 COMPRESSION OF 3D SCENE DATA

2.1 Space-efficient representation of kd-tree

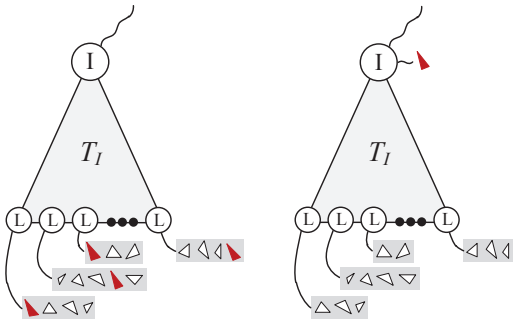
The kd-tree is one of the most essential spatial data structures that accelerate ray-object intersection computations. Whereas it is routinely used for efficient ray tracing, the kd-tree has a well-known drawback in which a large number of triangles that intersect with splitting planes are repeatedly duplicated into subvolumes during the kd-tree construction. This causes to increase the actual number of triangles that must be handled by the resulting kd-tree structure, often leading to inefficient, large and tall trees with high triangle redundancy (see Table 1).

Table 1: Sizes of example 3D scenes. The sizes of the kd-trees generated by the standard construction algorithm based on a surface-area heuristic (SAH) [Wald and Havran 2006] and represented with the compact data structure of Wald [Wald 2004] are compared with those of the respective geometry data. Here, the geometry of each scene with normals and texture coordinates at vertices was stored using the 'indexed face set' method. As is confirmed in the table, due to the replication of triangles during kd-tree construction, the resulting acceleration structure often imposes a substantial spatial overhead.

	No.'s of triangles (K)	No.'s of vertices (K)	Size of geom. (MB)	Size of kd-tree (MB)
Soda Hall	2,167.5	1,438.1	153.1	86.0
SanMig-7M	7,095.0	3,984.9	237.5	335.7
San Miguel	10,500.6	6,093.5	352.0	590.8
Power Plant	12,748.5	5,731.5	480.6	603.7

2.2 Extending the kd-tree representation method

In order to relieve the memory problem, Choi et al. [Choi et al. 2013] proposed a space-efficient kd-tree construction and representation scheme that allows an inner node to optionally store a reference to a triangle that would otherwise be duplicated in an excessive number of leaf nodes in the standard representation (refer to Figure 2). With a slightly modified kd-tree traversal algorithm, it was shown that their method markedly reduced the memory requirements for representing the tree structure, while effectively avoiding a serious degradation of the ray-tracing performance.



(a) Standard scheme (b) Choi et al.'s scheme

Figure 2: Augmented inner nodes for space-efficient kd-trees ([Choi et al. 2013]). In their method, Choi et al. had the option of storing a reference to a triangle with high redundancy in the root of a proper subtree instead of leaving multiple copies in the leaf nodes, which greatly reduced the space required for representing the subtree.

To enhance the space reduction effect further, we extend their kd-tree construction and traversal algorithms in such a way that up to two triangles may be referenced at an inner node. Our extension is based on the observation that the inner node layouts adopt an 8-byte alignment rule for efficient caching of the tree information, and the lower 32 bits of the *T-reference node*, i.e. the inner node with a reference to a triangle, are not used. In the original method, at most one triangle was permitted per inner node because an excessive amount of triangle references put on the inner nodes may slow down ray tracing seriously.

Be noted that the triangle references in the inner nodes incur a high frequency of, possibly unnecessary, early ray-triangle intersection computations during rendering. In addition, they cause too frequent divergent branches during the kd-tree traversal process, which is in particular critical when implemented on a high performance PC GPU. On the other hand, the GPU on the mobile device usually has a much lower parallel processing capability than the PC GPU. Therefore, the strategy of storing up to two triangle references on mobile platforms has the advantage of saving more space compared to the drawback of slowing down. This is more true when the ray tracer is implemented on a mobile GPU that usually has relatively limited graphics memory and memory bandwidth.

Table 2a shows how much memory usages for the geometries and kd-trees have been reduced further by our extension. In building the tested kd-trees, we set the pair of occupancy and frequency thresholds $(\tau_{occu}, \tau_{freq})$ to $(0.5, 0.4)$ in favor of achieving greater kd-tree-size reduction while keeping the frame-rate degradation at a reasonably low level (please refer to [Choi et al. 2013] to understand

what these control parameters are for). Also, for a good balance between the kd-tree-size reduction and ray-tracing speed, we let at most 8 triangle references in total be placed on inner nodes on the path from each leaf node to the root node of the kd-tree.

Table 2: Compression of 3D scene data. In these tables, "Standard" indicates the kd-tree construction and representation method that is based on [Wald and Havran 2006] and [Wald 2004]. On the other hand, "Choi et al." implies the method proposed in [Choi et al. 2013]. The percentages in parentheses show how the respective methods reduced the memory requirements compared to the standard SAH-based kd-tree.

(a) Sizes of the constructed kd-trees (unit: MB).

	Standard	Choi et al.	Ours
Soda Hall	86.0	50.1 (58.3%)	41.8 (48.6%)
SanMig-7M	335.7	202.7 (60.4%)	167.2 (49.8%)
San Miguel	590.8	342.2 (57.9%)	285.3 (48.3%)
Power Plant	603.7	321.0 (53.2%)	252.7 (41.9%)

(b) Total sizes of the scene data (unit: MB).

	Standard	Choi et al.	Ours
Soda Hall	239.1	203.2 (85.0%)	194.9 (81.5%)
SanMig-7M	573.2	440.2 (76.8%)	404.7 (70.6%)
San Miguel	942.8	694.2 (73.6%)	637.3 (67.6%)
Power Plant	1,084.3	801.6 (73.9%)	733.3 (67.6%)

Overall, our extension yielded a kd-tree-size reduction of 50.2% to 58.1% compared to the sizes of the standard kd-trees given in Table 1. Compared to the case when at most one triangle reference is placed per inner node, it also achieved an additional memory reduction up to 11.3%. Interestingly, the sizes of the geometry data also decreased slightly because culling more redundant triangles from leaf nodes enhanced spatial coherence in the triangle index lists, which in turn allowed the storage of a significant proportion of triangle indices in the *2-byte leaf mode* provided by the kd-tree layouts (refer to [Choi et al. 2013] for this leaf-node representation method). As a result, the total memory requirements to store the entire scene data were significantly reduced, as indicated in the Table 2b. As noted previously, it is inevitable that the ray-tracing frame rates drop to some degree because of additional computations for handling the inner nodes having triangle references. However, the timing results in Table 3 indicate only a slight decrease in rendering speed compared to the method of [Choi et al. 2013], which, considering the increased complexity of the kd-tree traversal algorithm, is quite encouraging.

3 EFFICIENT INTERACTIONS BETWEEN THE MASTER AND SLAVES

Our distributed mobile ray tracer employs a master/slave rendering model with one mater device and multiple slave devices on a mobile communication network, in which the image area is subdivided into a set of 2D tiles forming a pool of rendering tasks. The master is responsible for dynamically sending tile indices to and collecting rendered tile images from the slaves. Before the master starts to handing out tile indices to the slaves, each slave loads an entire copy of the necessary rendering data including the scene geometry and the compressed kd-tree in its graphics memory. All ray-tracing computations are performed on the GPUs of the slaves as initiated by the master.

Table 3: Ray tracing time variation according to different kd-trees (unit: ms). The times for ray-tracing an image of $1,024 \times 1,024$ pixels on a single mobile device are compared. The Power Plant scene could not be rendered with the standard kd-tree due to the lack of GPU memory space.

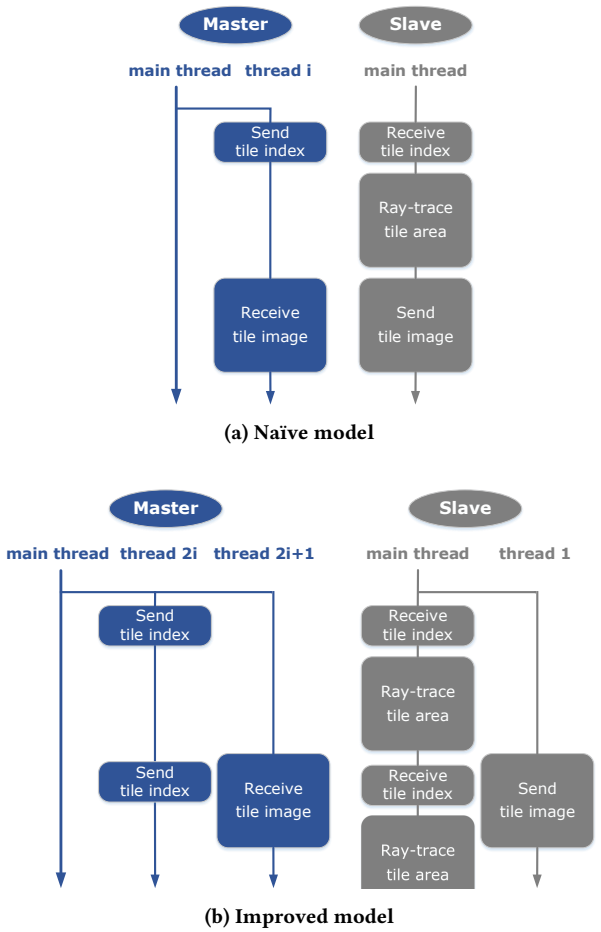
	Soda Hall	SanMig-7M	San Miguel	Power Plant
Standard	2,547.3	3,210.4	4,117.7	-
Choi et al.	2,648.2 (104.0%)	3,664.7 (114.2%)	4,194.9 (101.9%)	2,985.0 -
Ours	2,769.8 (108.7%)	3,868.5 (120.5%)	4,283.5 (104.0%)	3,093.7 -

Figure 3a illustrates a simple interaction model between the master and the ith slave, where a single thread on each side handles both communication and GPU ray tracing. Compared to the PC environment, however, we observed that the communication latency between mobile devices is quite high and sometimes irregular, easily becoming a major bottleneck in the entire distributed rendering pipeline. In particular, the mobile GPU had to stop rendering while the ray-traced tile image was being transmitted, leading to inefficient usage of computational resources. Therefore, in designing our distributed system, it was very important to hide the network latency of mobile communication as much as possible.

Figure 3b shows an improved interaction scheme. On the master side, two threads are created for each added slave, where one thread, thread 2i, is responsible for sending a tile index as soon as the ith slave is available, and the other, thread 2i+1, is dedicated to receiving the rendered tile image from the slave. On the other hand, the ith slave creates an extra thread only responsible for transmitting the rendered image tile while the main thread focuses mostly on the GPU rendering task. In this way, we were able to overlap the rendering computation and data transmission as much as possible on the slave side, enhancing the overall frame rates (see Table 4).

4 RESULTS

To demonstrate the effectiveness of the presented method, we first implemented a GPU-based full ray tracer using the OpenCL 1.2 API, which enabled to handle the three types of kd-trees constructed with and without the size-reduction methods applied. Then the proposed mobile ray tracer was implemented and tested on a mobile cluster built using LG G5 smartphones, each of which used the Qualcomm Snapdragon 820 chipset equipped with an Adreno 530 GPU and was connected to an IEEE 802.11ac-based wireless network (see Figure 1a and 5). For this smartphone, a graphics memory allocation error was encountered when trying to load rendering data larger



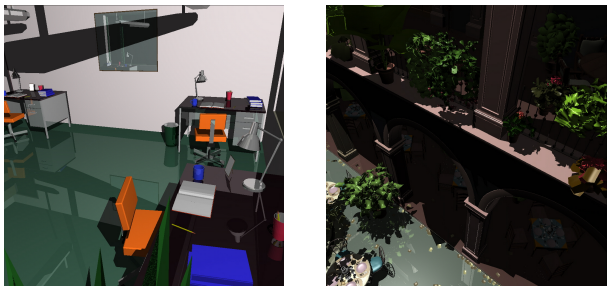
(a) Naïve model

(b) Improved model

Figure 3: Mobile interaction model between the master and the *i*th slave. By separating the tasks of image transmission and GPU ray tracing appropriately, the overall efficiency of mobile distributed rendering improved significantly.

Table 4: Rendering time comparison between the two master-slave interaction models (unit: ms). Images of $1,024\times 1,024$ pixels were ray-traced using tiles of 256×256 pixels. In this table, "N" and "I" respectively indicate the mobile interaction models that are illustrated in the figures a and b of Figure 3. The percentages in parentheses clearly show that the adopted interaction model significantly enhanced the timing performance on the mobile platform over the simple interaction model.

		# of participating slaves			
		3	4	5	6
Soda Hall	N	1,922.8	1,410.3	1,232.1	1,133.5
	I	1,642.9 (85.4%)	1,237.2 (87.7%)	1,011.9 (82.1%)	888.4 (78.4%)
Power Plant	N	1,257.2	978.2	874.7	762.2
	I	1,157.8 (92.1%)	859.3 (87.8%)	779.0 (89.1%)	670.1 (87.9%)



(a) Soda Hall (2,167K) (b) SanMig-7M (7,095K)

Figure 4: Example scenes and camera views tested. See also Figures 1c and d.

than roughly 950 MB. To perform fair evaluation, we selected four representative scenes and camera views with low to high geometric and rendering complexity (see Figure 1c & d and Figure 4), whose triangle numbers ranged from 2,167K to 12,749K.

Table 5a shows how effectively the distributed ray tracing time decreased as the number of participating slave machines increased, where each rendering time was averaged after five runs. In this table, the figures in parentheses indicate the efficiency which is defined as $\frac{\tau_1}{S\tau_S}$, where τ_1 is the execution time on one slave and τ_S is the time on S slaves. Despite the inefficient mobile communication environment, our distributed ray tracer was able to maintain an efficiency up to 80% when six smartphones participated in rendering. Note that these timings were obtained using a 4×4 partition of $1,024\times 1,024$ image meaning each tile is of 256×256 pixels. We also tested smaller tile sizes in a hope to achieve better dynamic load balancing, producing higher frame rates. However, in contrast to our expectation, the distributed rendering performance deteriorated rapidly. This is presumably because the communication overheads grew very fast as image tiles of smaller sizes were transmitted more frequently through the mobile network (see Table 5b).

5 CONCLUDING REMARKS

In this paper, we have demonstrated the possibility of using networked mobile devices, which are ubiquitous these days, for interactively visualizing large-scale 3D scenes. To the best knowledge of the authors, this is the first study on the mobile distributed ray tracing that can effectively handle 3D scenes with many millions of triangles. Unfortunately, we were not able to test our method against a mobile cluster of larger size due to the difficulty in building such a system. Even if more slave machines are available, simply adding them to the current form of mobile cluster with six slaves would not increase frame rates significantly due to fast growing communication overheads at the image transmission stage, to which the current mobile communication system is more vulnerable than the PC-based systems. A more balanced approach would be to build multiple small-scale mobile clusters made of, for instance, five to seven slaves, and to have each of the clusters render an

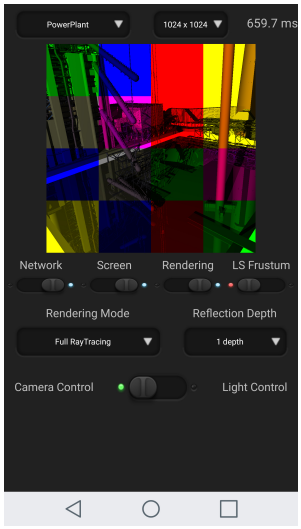


Figure 5: Tile-based mobile distributed ray tracing (Power Plant). Each different color represents a slave device that produced the corresponding tile image where six slaves participated in the distributed rendering.

interleaved frame, which we believe will increase the scalability of our distributed rendering system more effectively.

Note that whereas the presented method employs the space-reduced kd-tree structures for efficient distributed ray tracing, the scene geometry that represents the vertex and face information still requires a substantial amount of memory space as revealed in Table 1. We are currently developing a geometry compression method that is suitable for interactive mobile rendering. A successful combination of data reduction techniques for both components of scene data will generate a great synergistic effect for the ray tracing of very large 3D scenes on the ubiquitous mobile platforms.

In addition, to raise the overall frame rates of our rendering system, we are extending the GPU ray-tracing module to include the adaptive sampling technique [Kim et al. 2016] which was shown to reduce the total number of ray shootings, while introducing only a small deterioration in the rendering quality. Not only improve the computational speed, it would also help effectively reduce aliasing artifacts in ray-traced images through efficient supersampling.

ACKNOWLEDGMENTS

The test scenes are courtesy of the UC Berkeley Walkthrough Group (Soda Hall), Guillermo M. Leal Llaguno (San Miguel), and the UNC GAMMA group (Power Plant). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01006590).

Table 5: Computation time and efficiency of mobile distributed rendering. The ray tracing times (in ms) for rendering images of $1,024 \times 1,024$ pixels are reported.

(a) Rendering times using tiles of 256×256 pixels (4×4 block of tiles). The efficiency numbers in parentheses measure the scalability of our mobile distributed ray tracer.

	# of participating slaves					
	1	2	3	4	5	6
Soda Hall	2,769.8 (1.0)	1,502.7 (0.92)	1,085.2 (0.85)	876.6 (0.79)	670.1 (0.83)	607.9 (0.76)
SanMig-7M	3,868.5 (1.0)	1,996.6 (0.97)	1,504.3 (0.86)	1,092.4 (0.89)	913.9 (0.85)	811.6 (0.79)
San Miguel	4,283.5 (1.0)	2,368.7 (0.90)	1,642.9 (0.87)	1,237.2 (0.87)	1,011.9 (0.85)	883.4 (0.81)
Power Plant	3,093.7 (1.0)	1,660.9 (0.93)	1,157.8 (0.89)	859.3 (0.90)	779.0 (0.79)	670.1 (0.77)

(b) Rendering time variation according to tile block sizes (Power Plant).

	# of participating slaves			
	3	4	5	6
2×2	1,309.7	1,090.1	-	-
4×4	**1,157.8**	**859.3**	**779.0**	**670.1**
8×8	1,382.1	1,082.1	922.3	852.2
16×16	2,038.0	1,678.6	1,401.0	1,288.7

REFERENCES

M. Arslan, I. Singh, S. Singh, H. Madhyastha, K. Sundaresan, and S. Krishnamurthy. 2015. CWC: A Distributed Computing Infrastructure Using Smartphones. *IEEE Transactions on Mobile Computing* 14, 8 (2015), 1587–1600.

B. Choi, B. Chang, and I. Ihm. 2013. Improving Memory Space Efficiency of Kd-tree for Real-time Ray Tracing. *Computer Graphics Forum* 32, 7 (2013), 335–344.

Y. Kim, W. Seo, Y. Kim, Y. Lim, J. Nah, and I. Ihm. 2016. Adaptive Undersampling for Efficient Mobile Ray Tracing. *The Visual Computer* 32, 6-8 (2016), 801–811.

I. Wald. 2004. *Realtime Ray Tracing and Interactive Global Illumination.* Ph.D. Dissertation. Saarland University.

I. Wald, C. Benthin, and P. Slusallek. 2003. Interactive Ray Tracing on Commodity PC Clusters - State of the Art and Practical Applications. In *Proc. Euro-Par.* 499–508.

I. Wald and V. Havran. 2006. On Building Fast Kd-trees for Ray Tracing, and on Doing That in O(Nlog N). In *Proc. IEEE Symp. on Interactive Ray Tracing.* 61–69.