Load Balancing Algorithm for Real-Time Ray Tracing of Dynamic Scenes

JINYOUNG LEE1, WOO-NAM CHUNG1, TAE-HYOUNG LEE1, JAE-HO NAH2, YOUNGSIK KIM3, AND WOO-CHAN PARK1

1Department of Computer Engineering, Sejong University, Seoul 05006, South Korea
2LG Electronics, Seoul 06772, South Korea
3Department of Game and Multimedia Engineering, Korea Polytechnic University, Siheung 15073, South Korea

Corresponding author: Woo-Chan Park (pwchan@sejong.ac.kr)

This work was supported in part by the Institute for Information and Communications Technology Promotion through the Korea Government [Ministry of Science, ICT and Future Planning (MSIP)] (Development of mobile GPU hardware for photo-realistic real-time virtual reality) under Grant 2016-0-00204, and in part by the Ministry of Science and Information and Communication Technology (ICT) (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2020-0-00312.

ABSTRACT In this article, we propose a load balancing algorithm that accelerates ray tracing effectively and simply manner. The algorithm was developed in a hybrid system consisting of a CPU and hardware dedicated to ray tracing. Tree-building is processed on the CPU, and rendering is executed by the ray tracing-dedicated hardware. Because these components operate independently of each other, the final performance in terms of frames per second (FPS) is determined based on the time spent on tree-building or rendering, whichever process takes longer. This characteristic of a hybrid system is reflected in the developed algorithm, which dynamically adjusts tree-build parameters at every frame, thereby minimizing the interval between tree-building and rendering times. These effects ultimately increase FPS performance. Experiments at a resolution of 1920 × 1080 involving various dynamic scenes indicated that FPS performance improved by an average of 75.3% when the proposed algorithm was used.

INDEX TERMS Load balancing algorithm, ray tracing, tree build.

I. INTRODUCTION

NVIDIA recently launched Turing GPUs, which support ray tracing—a rendering technique now also enabled in many games released for PCs and consoles [1]. These developments were followed by the release of other technologies that support ray tracing, such as AMD’s RDNA2 GPUs [2], Microsoft’s DirectX Raytracing API [3], and the Khronos Group’s open ray-tracing API called Vulkan Ray Tracing. These releases are likewise supported on AMD, NVIDIA, and Intel platforms [4]. The aforementioned product roll-outs point to the extensive use of ray-tracing technology.

A diagram of a typical ray-tracing pipeline is shown in Figure 1. In the tree-building stage, a tree is constructed based on geometric data on a frame to be displayed. In the rendering stage, ray tracing is performed using the constructed tree. This chain of processes is repeated to render dynamic scenes.

Many studies have been conducted to improve ray tracing performance. Such an improvement can be achieved through the acceleration of stages of operation in a system in which each stage is performed sequentially (e.g., Figure 1). The situation differs with respect to a system wherein each stage is conducted independently. For example, if the processing of the tree-building stage is incomplete even when processing at the rendering stage is rapidly completed because of its acceleration, rendering is placed on standby (which translates to waiting time) until tree construction is finished. In such a system, an important requirement is to reduce waiting time by balancing performance at each stage.
Waiting can also occur if the tree-building and rendering stages are run asynchronously. Although waiting is not necessary under asynchronous operation, the slowing down of tree-building owing to a given scene potentially diminishes tree quality; if scene topology changes, a rebuild is needed, which may result in waiting time [5]. The cost of tree-building and the resulting waiting time can be reduced through the construction of a shallow tree using axis-aligned bounding boxes (AABBs) of each primitive [6]. However, this method involves the use of fixed tree-building parameters, which in the end can cause waiting time depending on the scene being rendered.

With consideration for the above-mentioned issue, we developed a load balancing algorithm that can solve the performance degradation stemming from waiting time. The algorithm can reduce waiting time by controlling the speed with which tree-building and rendering are implemented. Such control is achieved through the dynamic adjustment of tree quality at each frame.

To implement the proposed algorithm, we built a hybrid system that comprises a CPU and an FPGA that can process tree construction and rendering stages independently through the CPU and ray tracing-dedicated hardware, respectively. Experiments featuring four dynamic scenes were carried out to evaluate the algorithm. The results showed that performance in terms of frames per second (FPS) improved by 48% to 100%. The considerable intuitiveness and simplicity of the algorithm translate to an easy application for performance improvement in various platforms.

The rest of the paper is organized as follows. Section II presents ray-tracing acceleration studies and tree-construction methods. Section III describes the proposed algorithm and system. Section IV discusses the experimental environment and the results of the algorithm evaluation. Section V concludes the paper.

II. BACKGROUND

A. RAY TRACING ACCELERATION

In some ray tracing studies, the performance was improved via the acceleration of the rendering stage. In [7], the researchers proposed a highly efficient, scalable, and modular hardware architecture that achieves real-time ray tracing. The architecture includes the triangle intersection unit, ray generation and shading unit, and Kd-tree traversal unit. The system created in [8] was derived from the hardware structure presented in [7] and features a fully programmable single-chip structure and implementation. Other researchers put forward a highly parallel multi-core processor architecture [9] and a tile-based parallel ray-tracing system that runs with 128 programmable multi-core [10]. In [11], the researchers designed a high-performance multi-chip architecture in which multiple chips share a single core memory. Finally, in [12], RayCore’s RTU (ray-tracing unit) was developed in such a way that enables the implementation of the entire ray-tracing stage in hardware.

B. TREE CONSTRUCTION METHODS

The surface area heuristic (SAH) algorithm was presented in [13] to determine the cost of generating a kd-tree. The SAH, which is a sorting-based kd-tree algorithm, is defined in (1) as follows:

$$SAH(p) = K_T + K_I \left( \frac{V_L C_L}{V} + \frac{V_R C_R}{V} \right),$$

where $K_T$ represents the cost of tree traversal for a current node, and $K_I$ denotes the cost of an intersection test on rays and primitives. SAH(p) is determined based on the conditional probability that the current node can be divided into two sub-nodes and that the number of primitives can be divided into each sub-node. This conditional probability is calculated by dividing the area of a sub-node space by the current node space (V in the equation above). In (1), $V_L$ and $V_R$ respectively refer to the left and right sub-node spaces where the current node space is divided by p. $C_L$ and $C_R$ refer to the costs of the intersection test on primitives in the left and right sub-node spaces, respectively. The kd-tree constructed via the sorting method is optimized and of very high quality, but generating it entails a long time.
Many studies improved performance by accelerating the construction of kd-trees. In [14], the researchers proposed a scanning algorithm that can identify approximated SAH costs. In [15], the authors developed a GPU kd-tree construction algorithm that uses spatial median and SAH algorithms to process upper- and lower-level nodes, respectively. In [16], a highly parallel binning method for fast kd-tree construction was put forward.

RayCore’s TBU (tree-build unit) [12] is underlain by a hybrid method that combines binning and sorting approaches in the operation of kd-tree construction hardware. In the current work, the hybrid method was implemented as software. Figure 3 shows an example of a kd-tree generated via hybrid kd-tree construction. Scene splitting is performed using a binning method until the number of primitives for high-level nodes goes below a certain threshold, and low-level nodes are generated via sorting for the primitives of split scenes. Fast tree construction is possible using the binning method, and an excellent quality of trees is created using the sorting method. Compared to the exact SAH kd-tree construction [13], the quality degradation of the trees constructed through the binning and sorting approach is insignificant, but kd-tree building proceeds rapidly.

An alternative to using a kd-tree is employing a BVH (bounding volume hierarchy). Construction time is minimized by repeatedly merging agglomerative clustering pairs to build BVHs [17]. Researchers proposed a BVH-building hardware architecture that enables the fast construction of high-quality BVHs using parallel locally ordered clustering [17], [18].

**III. THE PROPOSED ALGORITHM AND SYSTEM**

**A. THE PROPOSED ALGORITHM**

![Flow diagram of the hybrid ray-tracing system with our load balancing algorithm. TB time refers to tree-building time. RD time refers to rendering time.](image)

To solve the load imbalance problem described above, we developed a load balancing algorithm that can reduce waiting time via adaptive adjustments to tree quality in each frame. This adaptive adjustment is carried out in such a way as to raise or lower the appropriate value of tree parameters based on the measurements of tree-building and rendering times for a previous frame. Figure 6 shows the flow diagram of the load balancing algorithm.

Tree quality can be adjusted in many ways [19], including setting a tree’s depth to a certain value, establishing the volume of each leaf node by using the max primitive number, and changing the $K_T$, $K_I$ values in (1) [6]. The present research adopted the adjustment of the max primitive number, which is a universal method. If this number is 1, the depth of a tree performance improved when tree-building was carried out on the CPU presented in Figure 4.

In such a system, a critical requirement is to balance the processing performance of each stage. Under imbalanced processing, one of the stages in the chain is placed on standby. To illustrate, if the rendering of a current frame has been finished, but the tree for the next frame has yet to be built, the rendering stage must be paused until the tree-building is completed. Figure 5(b) shows a case wherein waiting time is prevented through adjustments to the performance of each stage. The approach is expected to improve performance by as much as $L$ compared to Figure 5(a).
is at its deepest, and build time is at its longest, but rendering can be completed within the shortest period because it can produce a high-quality tree (best tree quality). The higher the max primitive number, the shorter the tree-building time and the longer the rendering time, with tree quality at a diminished.

**B. THE HYBRID RAY TRACING SYSTEM**

Figure 7 shows the overall organization of the hybrid ray-tracing system designed in this work. It can be largely divided into tree-building and rendering stages. The tree-build stage is performed on the CPU by operating a TBU [12] as software. The rendering stage is carried out via hardware through the use of an RTU [12] on the FPGA board. An RTU has excellent performance per chip area because it implements the entire stage of ray tracing as hardware. The entire stage includes the setup processing, ray generation, traversal & intersection, hit-point calculation, and shading units.

In the system, trees are built in parallel on the CPU, and rendering is performed in parallel on the FPGA board. The CPU and FPGA board operate independently of each other. The specific operation is described as follows.

The CPU reads geometry data of the dynamic scene to be rendered from system memory and builds a kd-tree. Our kd-tree structure includes static and dynamic acceleration structures (ASs) for static and dynamic primitives, respectively. The static AS is built once at the beginning of the process and reused at every frame, whereas the dynamic ASs are rebuilt at every frame. The constructed kd-tree is transferred to the local memory of the FPGA, which performs rendering based on the transmitted kd-tree. During rendering, the CPU constructs a kd-tree for the succeeding frame. In other words, when the FPGA executes rendering for the n frame, the CPU builds a tree for the n + 1 frame.

**IV. EXPERIMENT**

**A. EXPERIMENTAL ENVIRONMENT**

Figure 8 shows the system used in the experiments. This system is composed of a host computer and an FPGA board. We used an Intel Core i9-9920X (3.50 GHz) CPU for the host computer and VCU118 [20] (FPGA: Xilinx Virtex UltraScale + 9P) as the FPGA board. The host computer and the FPGA board were integrated using the PCI Express interface.

**TABLE 1. Utilization of implemented RayCore on the FPGA.**

| Block                | LUT   | Register | Block Memory | URAM  | DSP  |
|---------------------|-------|----------|--------------|-------|------|
|ilinx vixt ultra-scale+ 9p| 60.15%| 39.64%   | 59.79%       | 14.69%| 12.68%|

In our system, 12 RTUs are operated on the Xilinx Virtex UltraScale + 9P board at 200 MHz. The utilization figures for a look-up table, register, block memory, unified random access memory, and digital signal processor are presented in Table 1.

**FIGURE 8. Hybrid ray-tracing system.**

**TABLE 2. Dynamic scenes for the benchmark test.**

| Scene       | Number of Primitives |
|-------------|-----------------------|
| Sword       | 37k primitives        |
| Wood        | 30k primitives        |
| Flight-sim  | 29k primitives        |
| Water-wave  | 12k primitives        |

Figure 9 shows the four dynamic benchmark scenes used in the experiments. There are four types of Sword (37k primitives), Wood (30k primitives), Flight-sim (29k primitives), and Water-wave (12k primitives). The experiments were conducted at a resolution of 1920 × 1080.

Table 2 shows detailed information about each dynamic benchmark scene, which includes information on scene primitives (static, dynamic), the number of textures and lights,
and the total number of frames. We measured rendering time, tree-building time, and FPS in units of the frame for each dynamic benchmark scene and obtained an average value for the total frames on the basis of the measured values. To ensure accuracy in experimentation, we replicated the procedures five times and obtained the average from these replications.

### B. EXPERIMENTAL RESULTS

#### TABLE 3. Experimental results according to changes in tree quality.

| scene       | leaf-num (#) | RT (ms) | TT (ms) | RT-TT (ms) | FPS |
|-------------|--------------|---------|---------|------------|-----|
| Sword       | 1            | 96.5    | 200.9   | -104.4     | 5.2 |
|             | 2            | 97.0    | 187.5   | -90.5      | 5.3 |
|             | 4            | 97.9    | 183.2   | -85.3      | 5.4 |
|             | 8            | 100.9   | 163.5   | -62.7      | 6.1 |
|             | 16           | 107.1   | 144.1   | -36.9      | 6.9 |
|             | 32           | 126.1   | 124.5   | 1.7        | 7.5 |
|             | 64           | 171.1   | 119.2   | 51.9       | 6.1 |
|             | 96           | 228.1   | 108.7   | 119.5      | 4.6 |
| Wood        | 1            | 36.4    | 119.7   | -83.3      | 8.3 |
|             | 2            | 36.5    | 114.5   | -78.0      | 8.7 |
|             | 4            | 36.6    | 109.0   | -72.4      | 9.2 |
|             | 8            | 37.0    | 95.3    | -58.3      | 10.5|
|             | 16           | 38.2    | 83.2    | -44.9      | 12.1|
|             | 32           | 42.2    | 72.4    | -30.1      | 13.9|
|             | 64           | 53.4    | 67.0    | -13.6      | 15.0|
|             | 96           | 63.5    | 60.5    | 3.0        | 15.4|
| Flight-sim  | 1            | 27.9    | 67.7    | -39.7      | 14.9|
|             | 2            | 28.0    | 66.3    | -38.2      | 15.3|
|             | 4            | 28.4    | 64.5    | -36.0      | 15.6|
|             | 8            | 29.8    | 58.5    | -28.7      | 17.4|
|             | 16           | 33.3    | 50.9    | -17.6      | 19.7|
|             | 32           | 39.6    | 42.3    | -2.6       | 22.1|
|             | 64           | 52.8    | 40.0    | 12.7       | 20.1|
|             | 96           | 65.3    | 40.0    | 25.3       | 17.0|
| Water-wave  | 1            | 27.9    | 49.9    | -22.7      | 20.4|
|             | 2            | 27.9    | 46.2    | -18.3      | 22.1|
|             | 4            | 28.0    | 43.5    | -15.5      | 23.5|
|             | 8            | 28.3    | 29.8    | -10.1      | 33.4|
|             | 16           | 29.8    | 25.9    | 3.9        | 33.1|
|             | 32           | 34.9    | 22.2    | 12.7       | 28.5|
|             | 64           | 47.3    | 27.4    | 19.9       | 21.2|
|             | 96           | 59.4    | 26.2    | 33.2       | 16.9|

Table 3 shows the rendering time, tree-building time, the interval between them (RT–TT), and the FPS in accordance with changes in tree quality. Each value is the average for all the frames. Leaf-num refers to the number of primitives that a leaf node has. Tree quality was adjusted by dividing leaf-num into eight levels: 1, 2, 4, 8, 16, 32, 64, and 96. The best tree quality arises from a leaf-num value of 1; as this value grows to 96, tree quality decreases.

The optimal leaf-num values of the scenes are as follows: 32 for the Sword and Flight-sim scenes, 96 for the Wood scene, and 8 for the Water-wave scene (Table 3). In common, the closer the interval between tree-building and rendering times to 0 (i.e., the more balanced the processing performance of the two stages), the higher the FPS.

#### TABLE 4. FPS measurement results of when applying the load balancing algorithm and without applying the load balancing algorithm.

| scene      | PPS without load balancing algorithm | PPS with load balancing algorithm | improvement rate |
|------------|--------------------------------------|----------------------------------|------------------|
| Sword      | 5.2                                  | 7.7                              | 48.0%            |
| Wood       | 8.3                                  | 16.6                             | 100.0%           |
| Flight-sim | 14.9                                 | 23.9                             | 73.8%            |
| Water-wave | 20.4                                 | 36.6                             | 79.4%            |

Table 4 shows the FPS results derived with and without the load balancing algorithm. When the algorithm was applied, the leaf-num setting value was controlled by applying an increase or decrease of 8 to reduce the interval between tree-building and rendering times. The maximum value of the leaf-num is 96, which is divided into 13 levels. When the algorithm was not applied, we set the leaf-num value to 1 to ensure the best quality as this configuration is a commonly used setting in building a high-quality kd-tree.

As shown in Table 3, the tree quality with the highest FPS differed in each scene in accordance with scene characteristics. Thus, when tree quality was set to a certain level, maintaining high FPS in all the benchmark scenes was impossible. Nevertheless, our load balancing algorithm produced high FPS performance under all the dynamic benchmark scenes because it adjusted tree quality adaptively by measuring the interval between rendering and tree-building times at the frame level.

The experiments indicated that FPS performance improved from 5.2 to 7.7 for the Sword scene, from 8.3 to 16.6 for the Wood scene, from 14.9 to 25.9 for the Flight-sim scene, and from 20.4 to 36.6 for the Water-wave scene. These amount to an overall performance improvement of up to 48% to 100%.

### V. CONCLUSION, LIMITATIONS, AND FUTURE WORK

#### A. CONCLUSION

In this article, we proposed a load balancing algorithm that can improve FPS performance by balancing tree-building and rendering performance. Our experiments showed an average performance improvement of 75.3%. Our approach is very effective for a system wherein tree-building and rendering are done independently, as is the case in a hybrid ray-tracing system. The findings showed that performance can be enhanced by identifying the appropriate trade-off point on the basis of the relationship between tree-building and rendering performance in generating a given tree quality.

#### B. LIMITATIONS AND FUTURE WORK

We focused on improving performance by controlling tree quality, which is one of the parameters of tree-building.
variable rate shading [21] and foveated rendering [22].

parameters, but no consideration was accorded to rendering construction on other platforms, such as GPUs or dedicated hardware. We believe our approach is applicable to tree environments where a CPU processes tree-building, as in our

given that this method is highly scalable, it is not limited to environments where a CPU processes tree-building, as it is not limited to

parameters. In future studies, we intend to probe into parameters that affect speed and image quality in rendering, such as variable rate shading [21] and foveated rendering [22].

The proposed method revolves around tree-building parameters, but no consideration was accorded to rendering construction on other platforms, such as GPUs or dedicated hardware.
YOUNGSIK KIM received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, Yonsei University, South Korea, in 1993, 1995, and 1999, respectively. He was with the System LSI, Samsung Electronics Company, Ltd., from 1999 to 2005, as a Senior Engineer. Since 2005, he has been with the Department of Game and Multimedia Engineering, Korea Polytechnic University. His research interests include 3-D graphics and multimedia architectures, game programming, and SOC designs.

WOO-CHAN PARK received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1993, 1995, and 2000, respectively. From 2001 to 2003, he was a Research Professor with Yonsei University. He is currently a Professor of computer engineering with Sejong University, Seoul. His current research interests include ray-tracing processor architecture, 3-D rendering processor architecture, real-time rendering, advanced computer architecture, computer arithmetic, lossless image compression hardware, and application-specific integrated circuit design.