A General Two-Level Acceleration Structure for Interactive Ray Tracing on the GPU

Rafael Huff
Fraunhofer IGD
TU Darmstadt

Tiago Neves
Universidade do Minho

Thomas Gierlinger
Fraunhofer IGD

Arjan Kuijper
Fraunhofer IGD
TU Darmstadt

André Stork
TU Darmstadt

Dieter Fellner
Fraunhofer IGD
TU Darmstadt

***Abstract*—Despite the superior image quality generated by ray tracing, programmers of time-critical applications have historically avoided it because of its computational costs. Nowadays, the hardware of modern desktops allows the execution of real-time ray tracers but requires a specialized implementation based on specific characteristics of each application, such as scene complexity, kinds of motion, ray distribution, model structure and hardware. The evaluation and development of these requirements are complex and time-consuming, especially for developers with no familiarity in rendering algorithms and graphics hardware programming.**

The aim of our work is to provide a general and practical method to efficiently execute interactive ray tracing in most systems. We considered the most common aspects of current computer graphics applications, like the use of a scene graph and support to static and dynamic objects. In addition, we also took into account the common desktop hardware. This led us to the development of a special acceleration structure and its implementation on the GPU. In this paper, we present the development of our work showing the combination of different techniques and our results.

I. Introduction

Several areas of computer graphics can benefit from the quality of photorealistic rendering, like virtual reality, animation, modeling and even fluid simulation. The reproduction of the natural behavior of light provides images very close to real life and allows good space and depth perception of virtual objects. Ray tracing is considered one of the best physically based rendering technique for simulating light transport but its highly associated computational costs are generally prohibitive to time critical circumstances of some computer graphics areas.

Ray tracing is a computationally very demanding algorithm because it calculates the intersection of each ray of light with every object in the scene and for this reason acceleration data structures are adopted, such as Bounding Volume Hierarchies (BVH), KD-Trees and Grids [9]. The ray tracing computing times restricted its applicability to few off-line solutions in the past, frequently focused on obtaining a single high quality and full resolution image. Therefore, researches were mostly drawn toward the efficiency of the acceleration structures traversal and their building time was sometimes neglected.

Over the years, faster hardware has also been a way to improve ray tracing computing times. Aside from costly parallel solutions in PC Clusters, there were few options to desktop users. This scenario is changing with the introduction of more parallelism in current PCs. Multiprocessor CPUs are a common basis and the power of modern multi-core GPUs is getting unveiled to developers in the past years. This improvement of performance brings the possibility to use ray tracing for interactive scenes but it is necessary to rethink the acceleration structures used so far. As a result of the constant modification of objects in interactive scenes, the spatial structure must be rebuilt or updated at each frame. Consequently, not only the traversal time is important but also the rebuild time; it is then necessary to find a *trade-off* between both. This tradeoff leads to no consensus about the best acceleration structure for interactive scenes. It depends on *specific* characteristics of each application, like scene complexity and kinds of motion, and for this reason it is difficult to find a general solution for every case. Therefore, although the current hardware allows the use of ray tracing for interactive rendering, developers still avoid it partially because of the lack of a *general* solution, combined with the need of graphics hardware knowledge and implementation of a ray tracing from scratch.

The goal of our work is to provide a *general* ray tracing algorithm that can easily be used by most applications. In order to accomplish that, we *combine* different rendering techniques and acceleration structures. Unlike the majority of recent researches in ray tracing, our objective is not the development of a specialized acceleration structure for maximal rendering performance of a *specific* application, but a solution that would effectively fit in general.

II. Related Work

A large variety of special data structures has been used to accelerate the intersection tests of ray tracing, like BVH, KD-Trees and Grids. Even though KD-Tree is still the best structure to accelerate ray tracing in static scenes, BVHs tend to be more flexible to incremental updates in dynamic scenes [9]. The use of these hierarchical acceleration structures usually implies corresponding stack-based traversal algorithms. Unfortunately, even the latest GPU architectures are poorly suited for implementing such algorithms [6].

A way to efficiently traverse a BVH on the GPU without a stack was presented in [7]. In this work, previous implementations of grids and KD-Trees on the GPU were outperformed but better performance than a well-optimized CPU ray tracer was not confirmed. Later, a better performance than CPU ray tracers was achieved using stackless KD-Tree traversal

and packet tracing in [2] and [6]. However, these techniques worked for static scenes only.

Zhou et al. [10] presented the first real-time KD-Tree algorithm on the GPU. Their work led to a GPU ray tracer for dynamic scenes that outperformed state-of-the-art multi-core CPU ray tracers. Even though they accomplish very high performance, their stack-based approach limits the depth of the tree [4]. Each thread requires a local stack which is allocated using a fixed-size array in thread-local memory. For well balanced KD-Trees it works fine, but due to the local memory size limit, deep structures would not fit in this approach. Finally, current methods for traversal and intersection are tested against theoretically obtainable limits in [1].

As previously mentioned, there is no optimal single acceleration structure technique for *every* scene. The division of trees in separated levels and the combination of different structures to outperform a single one on the CPU is not new [5], [8] but they have been neglected on the GPU.

In our work, we implemented a *two-level hierarchical tree* able to combine different acceleration structures in order to support scenes with diverse characteristics in one single general solution. Therefore, the depth of the tree cannot be limited and the stack-based techniques presented in [10] are not appropriate. Moreover, the previous techniques in [2], [6] are also not suited because they cannot handle dynamic scenes. Consequently, we present a new solution able to combine dynamic objects and a stack-less approach on the GPU in order to handle a broader set of scenes.

The structural information presented in a scene graph can also improve the construction and speed of ray tracing acceleration hierarchies [3]. Based on that, we integrate the structural information provided by OpenSG scenes in our tree. This information is also used to handle dynamic scenes. We reference OpenSG because it is C++ compliant and has cross-plataform capabilities.

III. System Design

The efficiency of a ray tracing algorithm relies usually on two main factors: adopted acceleration structures and hardware. In Section III-A, we present the design of an acceleration structure that can fit in general applications and its implementation on current desktops is shown in section III-B.

A. Adapted Acceleration Structure

Since the objective of our work is to provide a general solution for ray tracing, we implement a structure that handles both static and dynamic scenes. Moreover, our solution also deals with dynamic scenes in which some objects remain static for a long period of time. In order to accomplish that, we use a similar approach as [8], where a two-level hierarchy encapsulates geometries in its own structure and connects them with a top structure. Build and update time for non-deformable objects is then eliminated by storing their geometries in separated precomputed acceleration structures. In our case, we use BVH because it is competitive in performance when applied to dynamic scenes and it is a simpler acceleration structure to build and maintain.

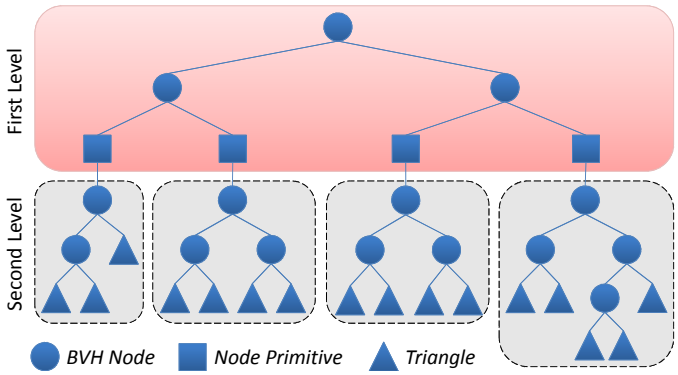

Fig. 1. Two-level acceleration structure

Our BVH is a binary tree composed of primitives. A primitive is either a node or a leaf. Nodes have a bounding box and two children - other primitives. Leaves are at the bottom of the tree and represent geometry elements - the triangles. We also implement axis aligned bounding boxes (AABB) for a faster traversal. With the aim to implement the aforementioned two-level hierarchy, we extend the implementation into four classes:

1) *BVH Primitive* - a BVH element abstract class inherited by *BVH Node*, *Node Primitive* and *Triangle*;
2) *BVH Node* - a default BVH node with an AABB and two children of type *BVH Primitive*;
3) *Node Primitive* - a class associated with an OpenSG geometry node and a child of type *BVH Primitive*;
4) *Triangle* - the leaf element representing a triangle of the geometry mesh.

BVH Node and *Triangle* are the usual BVH node and leaf, respectively. *Node Primitive* is a new concept which is an adaptation from the OpenSG scene graph. It is meant to make the project more versatile and prepared for dynamic scenes. *Node Primitives* are a mixture of leaf and node; on the one hand they are interpreted and treated as leaves, on the other hand they also have children. In fact, the final result can be interpreted as a BVH of BVHs, i.e., there is a primary BVH where the leaves are *Node Primitives* and each leaf has another BVH attached to it where the leaves are *Triangles*, as shown in Figure 1.

All *Node Primitives* are at the same tree depth. This is due to the building method: a first BVH of *Node Primitives* is built and then, for each one, a new BVH is created with its *Triangles*. That separates the concept of scene graph primitives organization and primitives soup. The idea behind is that rigid objects will never need to update their BVH, so dynamic scenes of rigid objects would only imply an update of the first BVH - the one of Node Primitives. The other BVHs would not need to update because each *Node Primitive* has the OpenSG geometry node transformation matrix attached to it. When a ray traverses the tree it is multiplied by the inverted transformation matrix of each *Node Primitive*, enabling correct

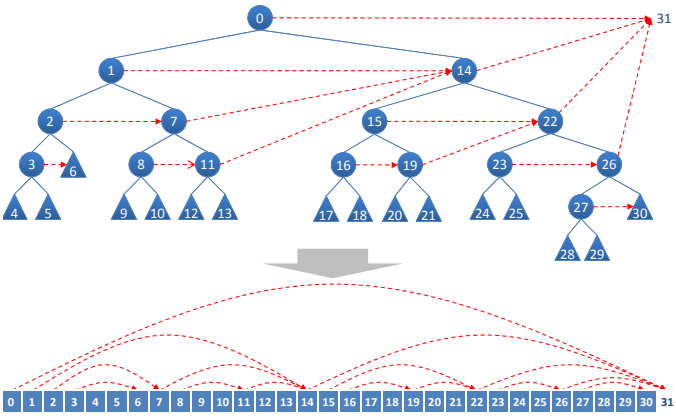

Fig. 2. BVH traversal example

and updated intersection tests.

The presented construction permits to have different acceleration structures between levels. KD-Trees can be used for better traversal of rigid object structures, for instance. The only requirement for a hybrid strategy is to apply a different construction method to each structure.

B. Implementation on the GPU

In order to reach real-time rates using ray tracing on a desktop PC, our solution is designed to profit from the GPU parallelism. To accomplish that, the BVH representation has to be converted into a more suitable and co-operating data representation because recursion is not supported on the GPU - given that there is no stack available. Moreover, pointers are not fully supported. Consequently, we adapt the work of [7], where the structure is converted to an array.

The idea consists of numbering every node in a deep first, left to right order. These numbers match the node position in the array. Moreover, each node has an escape index pointing to the node to escape to if no intersection is found. In [7] a texture is used to store the tree traversal representation and not the tree itself although GPGPU allows joining both into one single representation. It also states that leaves do not need to store any escape index, because their escape node is always the next one, and nodes that do not have another node to escape point to the end of the array - that happens when the ray misses the object and hits the background. Figure 2 shows the converted structure of Figure 1.

This BVH representation is achieved with some adaptations. Firstly, the concept of two-layers is removed by eliminating *Node Primitives* from the tree structure. Nonetheless, the information carried within them - such as the characteristics of objects' materials - was kept in a separated array, *M*. Thus, each BVH node points to its own material in this array. Secondly, triangle vertices information (position, normal, color and texture coordinate indices) are kept in another array, *T*. Each element of *T* corresponds to a triangle, so it has information about three vertices, and *T* has as many elements as the number of triangles in the tree. Each leaf of the tree points to its own element in *T*. Finally, three more arrays are

Fig. 3. Evaluated scenes: Sphere (left), Bunny(middle) and Dragon (right)

built: one with textures information (size, number of channels, etc.), *TextInfo*; another with texture coordinates, *TexCoord*; and a last one with the textures themselves, *Tex*. A *TexInfo* element denotes how to read and interpret textures from *Tex* - which are pointed to by a *TexCoord* element. Such information is directly related to the OpenSG Texture binding approach. Each element of *M* has a pointer to a *TexInfo* element and each element of *T* has pointers to three elements of *TexCoord*; a texture is read from *Tex* by combining the information provided by *TexInfo* and the *TexCoord*.

IV. Results

With the purpose of comparison, we implemented different versions of the ray tracer on the GPU and CPU with the above mentioned structure. On the GPU, CUDA and OpenCL were used. CUDA is the most adopted general purpose parallel computing architecture for GPUs but is restricted to nVidia graphic cards only. In contrast, OpenCL is a new royality-free framework for parallel programming intended to be portable across different hardware manufacturers or even different platforms. On the CPU, we first implemented a standard single-threaded version of the acceleration structure using pointers and recursion. Further, we implemented on the CPU the same array structure used on the GPU to be executed serially, called here *Emulated* version.

Three scenes containing refractive and reflective materials were tested on a nVidia GTX280 GPU and on a Intel Core 2 Quad CPU 2.50GHz: Sphere (10K triangles), Bunny (70K triangles) and Dragon (870K triangles) - illustrated in Figure 3. The images were rendered at resolution 800x600 and had 3 levels of reflection. Soft shadows are previously calculated using Precomputed Radiance Transfer. Table I shows the average time per frame.

As we did not focus on optimizing a specific case, timings are not state of the art. Nevertheless, we can already achieve better results on one single GPU than [8] on 32 CPUs. The integration with some techniques of [4] will very likely speed up the computation. As expected, CUDA and OpenCL parallel versions are faster than both serial CPU versions. Furthermore, CUDA is generally faster than OpenCL and after evaluating the nVidia bandwidth test, we found out that this is probably caused by the considerably slower device to device memory copy of OpenCL. However, this is most likely a driver issue which might be solved in future driver releases. Nevertheless, the developed array structure improved considerably the frame rate on CPU.

TABLE I
AVERAGE TIME TO COMPUTE A FRAME (IN SECONDS)

	CUDA	OpenCL	CPU	Emulated
Sphere	0,105s	0,33s	17,9s	1,93s
Bunny	0,187s	0,515s	5,115s	2,01s
Dragon	0,41s	1,49s	13,25s	4,46s

In our tests, the number of triangles in a scene did not affect the performance considerably. In fact, a bigger scene could run faster than a smaller one if fewer first rays intersected the scene. It means that the two-level acceleration structure *has efficiently reduced* the number of ray-primitive intersection tests and the scene complexity has not influenced traversal time.

V. CONCLUSION AND FUTURE WORK

This paper presented our work towards a general and practical interactive ray tracing solution that tries to maximize the rendering performance on most applications. Our work considered the characteristics existing in most current systems, such as a scene graph structure, GPU hardware and dynamic and static objects. We presented the development of a two-level acceleration structure and its implementation on the GPU. Our results positively showed that the number of primitives in the scene did not influence directly the traversal time of the structure. We also showed that the same array representation of the acceleration structure - initially created to be executed on GPU - also improved considerably the traversal time when adopted on the CPU.

Future directions lead to a KD-Tree implementation comprising the acceleration structure of non-deformable objects and some tests with dynamic scenes. A hybrid solution is already supported by our approach and a faster traversal for structures that do not have to be updated could improve the rendering speed. In addition, the performance can be improved with a better approach regarding rays coherence.

ACKNOWLEDGMENT

This research was funded in part by the European Communitys Seventh Framework Programme, under grant MAXIMUS FP7-ICT-1-217039, and by the brazilian research agency CNPQ, under the project number 290118/2006-9.

REFERENCES

[1] Aila, T., Laine, S.: Understanding the efficiency of ray traversal on gpus. In: HPG, pp. 145–150 (2009)
[2] Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree GPU raytracing. In: SI3D, pp. 167–174 (2007)
[3] Hunt, W., Mark, W.R., Fussell, D.: Fast and lazy build of acceleration structures from scene hierarchies. In: IEEE Symposium on Interactive Ray Tracing, pp. 47–54 (2007)
[4] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH construction on GPUs. Comput. Graph. Forum **28**(2), 375–384 (2009)
[5] Müller, G., Fellner, D.W.: Hybrid scene structuring with application to ray tracing. In: ICVC, pp. 19–26 (1999)
[6] Popov, S., Günther, J., Seidel, H.P., Slusallek, P.: Stackless kd-tree traversal for high performance GPU ray tracing. Computer Graphics Forum **26**(3), 415–424 (2007)
[7] Thrane, N., Simonsen, L.O.: A comparison of acceleration structures for gpu assisted ray tracing. Master's thesis, University of Aarhus (2005)
[8] Wald, I., Benthin, C., Slusallek, P.: Distributed interactive ray tracing of dynamic scenes. In: PVG, pp. 77–86 (2003)
[9] Wald, I., Mark, W.R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S.G., Shirley, P.: State of the art in ray tracing animated scenes. In: STAR of EG, pp. 89–116 (2007)
[10] Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time kd-tree construction on graphics hardware. In: SIGGRAPH Asia, pp. 1–11 (2008)