Real-Time SAH BVH Construction for Ray Tracing Dynamic Scenes

Dmitry Sopin[1], Denis Bogolepov[1], Danila Ulyanov[2]
[1]N.I. Lobachevsky State University of Nizhni Novgorod
[2]R.Y. Alekseev State Technical University of Nizhny Novgorod
sopindm@gmail.com, denisbogol@gmail.com, danila-ulyanov@ya.ru

Abstract

This work is aimed at the development of effective algorithms for building of full SAH BVH trees on GPU in real-time. In this work it is presupposed that all the scene objects are represented by a number of triangles (the so-called "triangle soup"), at the same time the arbitrary changes in the geometry are allowed in the process of rendering. The proposed methods have allowed more than tenfold increase performance compared to the best GPU implementation known.

Keywords: *Ray Tracing, Acceleration Structures, BVH, SAH, Real-Time, Dynamic Scenes, GPGPU, OpenCL.*

1. INTRODUCTION

The ray tracing algorithm was traditionally used in computer graphics for image synthesis of high quality. For getting the results a large amount of computation is needed. That is why for a long time using of the method in interactive applications and real-time systems seemed impractical. The main difficulties were related to two phases of work: building of acceleration structures and visualization based on ray tracing. For the effective solution of the highlighted tasks the programmable graphics accelerators can be used, which have turned into high-performance general-purpose processors over the past few years.

The first tries to implement the ray tracing on the GPU allowed the processing of the scenes with *static* geometry only [1]–[3]. This restriction allowed to build accelerating structure at the stage of pre-processing of the scene and use it on the GPU to make the rendering faster. The extensive research has shown that on standard consumer hardware ray tracing in real-time is possible for all major acceleration structures including uniform and hierarchical grids, kd-trees and bounding volume hierarchies (BVH). The technology has already found application in many fields of tasks but still has been of little use in applications with *dynamic* geometry, such as computer games, simulators and virtual reality systems.

The considerable attention of contemporary research is associated not only with the ray tracing methods but also with algorithms allowing the quick building of effective acceleration structures which would provide the support for dynamic geometry. In this case the formulation of the problem is changing radically because now it is necessary to take into consideration not only the efficiency of data structures at the stage of rendering but also to take into account the time needed for its construction which often appears the main limit for the working speed. The recent works have shown that the supporting for dynamic scenes with using of all the main accelerating structures can be possible [4]. However, in some cases the animation is put under some restrictions, in particular, only the *hierarchical* movements of the primitives or *deformable* scenes are allowed.

In this work for faster rendering BVH trees were chosen because of their advantages. First, this structure provides the most "dense" approximation of the geometry of the scene with a minimum number of nodes – it requires a minimum number of steps in the construction and traversing of the tree. Secondly, in the process of tree building only the centroids of triangles are used, that is why the situation with intersection of the split plane by a triangle is excluded – the primitive always belongs to the one descendant only. Another useful property of BVH trees is in the opportunity for their *updating* instead of a full *rebuild* which is used in several works where the animation is possible only with few limitations. Nevertheless, the processing of the scenes with arbitrary animation is possible only due to the presence of algorithms for fast building of BVH from scratch.

The high performance of ray tracing directly depends on the quality of the tree the best criterion for which serves the *surface area heuristic* (SAH). This heuristic was first proposed in [5] and is defined as follows: at each step of the recursive construction of the tree in the process of splitting of the set of triangles into two parts L and R the cost of that splitting is being computed:

$$SAH(T \to (L,R)) = K_T + K_I \left[\frac{SA(L)}{SA(T)} N_L + \frac{SA(R)}{SA(T)} N_R \right]$$

Here, $SA(X)$ stands for the area of the bounding volume of node X, N_X stands for the number of triangles in X, when K_I and K_T stand for implementation options that determine the cost of the test for intersection and traversal of the tree. The high effect of rendering is achieved through minimizing of the cost of splittings in the process of construction of data structures.

The first attempt to interactively build the SAH BVH trees was made in work [6], in which the author has adapted the *binned* technique that was originally proposed for *kd*-trees, and effectively implemented it on the multi-core CPUs. Subsequently, this implementation was further developed for the Intel MIC architecture, where the best timing estimates were obtained [8]. Recent work [7] has shown that the construction of SAH BVH tree on the GPU can be also possible, but the specified timing estimates turned out to be even higher than the similar estimates for older CPU [6], despite the use of more powerful architecture. Thus, at the present time there are no effective methods for construction of SAH BVH trees on the GPU, which would provide the opportunity for rendering of arbitrary dynamic scenes.

2. PROBLEM STATEMENT

This work is aimed at development of effective algorithms for building of full SAH BVH trees on GPU in *real-time*. In this work it is presupposed that all the scene objects are represented by a number of triangles (the so-called "*triangle soup*"), at the same time the arbitrary changes in the geometry are allowed in the process of rendering.

From a conceptual point of view the algorithm of the tree building is analogous to the algorithms which are described in the works [6]–[8]. Still in contrast with the works mentioned we don't use the simple heuristics similar to "Median Splits" and "LBVH" because they lead to decline in the quality of the generated data structure and consequently to the worse rendering performance. This work proposes a number of methods for the effective mapping of the general algorithm to the architecture of modern GPUs what allowed to accelerate the tree building up to 10 times compared to the best known GPU implementation [7].

3. BUILDING OF THE SAH BVH TREE

3.1 The Basic Algorithm

The process of the tree construction is represented in the form of the set of tasks the subsequent performing of which is realized through the concept of the task queue. The algorithm can be described by the following sequence of steps:

1) The *root* node is added to the task queue that contains all geometric primitives.
2) The first node in queue becomes *current*.
3) The current node is split into 16 bins along all the three axes. For every bin the number of geometrical primitives is calculated and the bounding volume is computed.
4) The optimal split plane is calculated by using SAH.
5) The current node is split into two new nodes containing geometrical primitives located to the left and to the right from the selected plane respectively.
6) For every new node the number of geometrical primitives is compared with the specified threshold number (we used 4). If the number of the primitives exceeds the threshold then the corresponding node is added to the task queue.
7) The current node is removed from the task queue. If the queue isn't empty then we go to the step 2.

3.2 The Adaptation for GPU

The main objective of this work was mapping of the general algorithm on the architecture of modern graphics processors.

The most simple way is mapping of a single task on one *work group* (here and below we use the terminology of OpenCL standard). In this case, we will repeat getting of the same result as in work [7]: low performance on the first levels of the tree, as soon as only some part of the available cores will be used in calculations, and the decline in productiveness at the last levels of the tree associated with an increase in overhead costs needed to support a large number of small tasks.

The following relatively simple method was proposed in work [8] and implemented for the Intel MIC architecture: for large nodes (with the number of primitives bigger than the specified threshold) the resources of the whole processor are used, while the other nodes are processed according to the previous scheme – a single task for a work group. This approach has proved effective for the architecture of Intel MIC, but when mapped on the GPU architecture there appears a number of problems. As in the case with the previous approach, there is a reduce in productivity at the low levels of the tree, and there an "intermediate layer" appears – a part of the levels at which a single task is not able to load the entire graphics processor, but the number of tasks is not large enough to project them onto a work group.

3.2.1 The General Scheme

For an effective use of modern GPU we have developed an improved algorithm for building of SAH BVH tree, which is largely free from the problems mentioned above. The general scheme of the algorithm is as follows:

1) At each step of the construction of the tree *all* the available tasks are fulfilled, regardless of their size. As a result, we are able to utilize the resources of the GPU to the full as well as to reduce the overhead costs associated with the transfer of tasks from the CPU.
2) For the processing of the nodes containing a sufficient number of geometric primitives (we used a threshold of 256), several work groups are used on a node (we used $N/512$, where N – the number of geometric primitives in the node). In conjunction with the optimization from paragraph 1), this approach enables us to use the number of resources close to the optimum.
3) The construction of the tree is divided into 3 stages: the processing of *large* nodes (more than 32K of primitives), the processing of *secondary* nodes (from 256 to 32K primitives) and the processing of *small* nodes (less than 256 primitives). This approach makes it possible to use the special modifications of the algorithm at each level, what results in the possibility to reduce the overhead costs and optimize the utilization of the GPU resources.

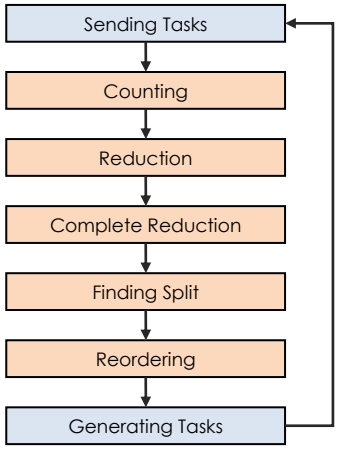

Figure 1: Construction of the tree level.

Figure 1 illustrates the most general scheme for building of the level of the tree (which is used at the stage of processing of the large nodes). Of all the presented stages only the sending of tasks and the generation of the new tasks are implemented on the CPU. These stages are the least labor-consuming, so their porting to the GPU is not reasonable.

3.2.2 The Generation of the Tasks for GPU

As it was stated earlier, in our implementation for processing of one node the several work groups are used, while several groups of nodes are processed simultaneously. This allows for high loading of the GPU, but, unfortunately, leads to the fact that there is no simple way to determine the amount of work for each specific work group. Therefore, as the additional parameters for work group we pass two data arrays – the number of the node being processed by the work group, and the number of this work group in the node. This information combined with data about nodes (the number of the primitives, the number of the first

primitive, bounding the volume of the node) allows to determine the amount of work for each work group.

3.2.3 Counting

The second stage in the construction of the tree level is counting of the number of primitives in each bin and of the bounding volumes (AABBs) of these primitives for each work group (this is correct as each work group processes the primitives from one node of the tree). In this case, the bins for all the three coordinate planes are calculated – as it was proved by the experiments in certain scenes it makes a significant (up to 10-fold) performance increase. Stage can be divided into two parts:

1) The conversation of information about geometric primitives into the information about the bins of tree nodes.
2) The application of the algorithm for parallel reduction to this data.

To implement the first part we have used the approach similar to that used in works [7] and [8] – the primitives of each work group are divided into parts consisting of 16 elements (in accordance with the number of bins) and each of the mentioned parts is processed by 16 consecutive threads ("*halfwarp*" in the terminology of NVIDIA CUDA). This algorithm can be described by the following pseudo code:

```
for i in 1 to 16 do
  if bin(triangle[i]) = threadId
    bin[threadId].append(triangle[i])
```

Using 16-passes on the 16 elements may seem superfluous, but it has the following advantages:

1) It is optimally mapped on the GPU SIMD architecture.
2) It minimizes memory conflicts (eliminates "*bank conflicts*" and provides "*coalesced*" access).

As an alternative solution to this problem we can propose the calculation by each data stream of the bin for an associated triangle and the subsequent reduction of these data. This approach leads to an increase in required memory (about 16 times) and the significant computational costs associated with the subsequent reduction.

3.2.4 Reduction

With the implementation of this phase the classical algorithm of parallel reduction was used. However, in the process of adaptation one significant change was made.

Firstly, we do not use a *variable* number of iterations of the algorithm. For the processing of the "average" levels of the tree just *one* iteration is enough, for the processing of "high" levels, we used an additional kernel of "final reduction", reducing all the available sets of bins in one (for "small" levels this step is not needed at all). The losses of productivity are not important for this approach, since most of the computational burden falls on the steps of calculation and reduction. The result is a simpler algorithm and reduced the amount of data exchange between the CPU and the GPU.

3.2.5 Search of Optimal Splitting

This is the one of the least resource-consuming parts of the algorithm. On the base of the available data on the bins the optimal split plane is calculated in it with the help of SAH. It was implemented on the GPU to reduce the amount of the traffic between the CPU and the accelerator.

3.2.6 Reordering

The last resource-consuming stage of the tree construction is the reordering of the nodes' elements in accordance with the found splittings (the elements on the left of the split plane are moved to the beginning of the array, the elements on the right – at the end). As a basis for solving this sub problem we used the *radix sort* algorithm, effectively implemented for the GPU in work [11].

This algorithm can be divided into two main parts – the prefix summation of the indices of the record (the place in the array where the element is to be placed) for all the geometric primitives of the node and strictly speaking the reordering of the array elements. Traditionally these parts are performed in several passes which entails additional costs of memory because of the need to store the results of intermediate calculations and they require additional calculations (associated with the launch of a prefix summation on the "global" level [12]).

In our algorithm the *atomic* operations on the global memory were used instead of the "global" prefix summation. As a result the additional steps for computing of the prefix sum on a global level have been replaced by *one* atomic operation for the work group on the local level what allowed to reduce the memory consumption and the number of computations.

3.2.7 Generating new tasks

It also belongs to the least resource-consuming parts of the tree building. At this stage the new nodes (corresponding to the obtained splittings) are added into the resulting tree and, depending on their size, the new tasks are generated. At the stages of processing of large and medium-sized nodes the nodes with the number of primitives which is less than a predetermined threshold (we used the 32K for the large and 256 for the medium ones) are added to the task queue belonging to the next step. On the stage of processing of small units they are considered to be the leaves (here as the threshold value is used 4).

4. TRAVERSING

4.1 The Basic Algorithms

So there are two main approaches to the realization of traversing of the tree on GPU: the *stack-based* and the *stackless*. The stackless approach was widely spread when only shaders were available for the purposes of ray tracing. As soon as writing data from the fragment shader (kernel) was possible only at the current fragment of the output texture, for realization of the full stack it was necessary to use the multi-pass schemes of low efficiency. With the advent of such instruments as CUDA and OpenCL it became possible to use the stack because we can easily get the access to the global device memory for reading and writing. Still to compare the performance we have realized the both variants.

4.1.1 The Stackless Algorithm

The stackless algorithms are normally based on the preprocessing of the tree and providing of additional information for the traversing of the tree. The algorithm based on using of the *escape indices* serves as an example of that approach for BVH tree [9]. The information about the node that is passed the next is counted and saved for every node. The algorithm showed some good results on simple scenes but with the scene getting more complex (what means the tree getting bigger) it turned out to behave much worse than the stack one. Besides that in the situation when the tree is built on every frame it is necessary to take into

consideration any additional processing of the tree because it must be done every time.

4.1.2 The Stack Algorithm

When traversing with a stack we can choose to which node of the tree we are to go further. For the ray tracing tasks the tree is being traversed in *front to back* order, what means that on every step you need to go to the nearest node. In this way the search for the nearest hit is done quicker. For traversing of the tree with a stack it is necessary to organize separate stacks for each thread. For that we need to allocate the fixed-size array in the *private* (in the terms of OpenCL) memory sufficient for storing of the stack of the maximum possible length the same as it is described in work [10].

4.1.3 The Stack in Registers

Though the organization of the private stack for every thread doesn't appear to be a problem anymore still any incoherent references to the external memory are undesirable. We have realized the variant of the traversing with a stack in the GPU registers. Thus for the stack element only 2 bits from the register will be given. With this approach the traversal algorithm has got three main states: to visit the right neighboring node, to visit the left neighboring node and also to go up the tree. A set of these states provides sufficient information for the traversing of the tree in the same order as with a normal stack. However, because of the growing complexity of the kernel code and the need to prepare and read the additional information for each node of the tree, such an approach in the current implementation did not provide for a significant advantage in comparison with the normal stack.

5. RESULTS

To estimate the operating time of the described algorithm the different scenes have been used with complexity from 10K to 3M of triangles. The tests have been conducted on a computer with the GPU NVIDIA GeForce GTX 480 being run by the Linux operating system (with 270.41.06 version of the video driver).

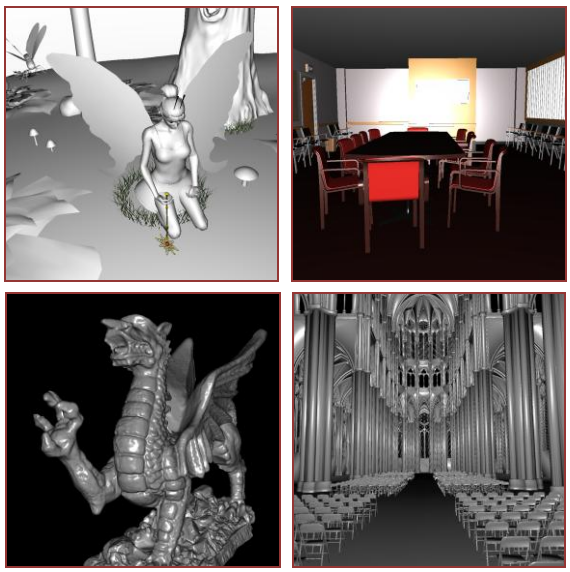

Figure 2: Sample frames from the test scenes (*Fairy Forest*, *Conference*, *Welsh-dragon* and *Cathedral*).

Table 1 contains the comparison of our results with the results from other well-known works.

Table 1: Comparison to other GPU SAH BVH builders: pure build time (*ms*) / rendering performance (*FPS*, 1024×1024).

Scene	Lauterbach [7] (GTX 280)	Wald [8] (Intel MIC)	Ours (GTX 480)
Toasters / 11K	N/A	11 / 105	13 / 83
Fairy Forest / 174K	488 / 21	31 / 29	40 / 25
Cloth / 92K	N/A	19 / 97	19 / 42
Dragon/Bunny / 252K	403 / 8	43 / 55	49 / 15
Conference / 284K	477 / 24	42 / 46	98 / 36
Welsh-dragon / 2.2M	N/A	N/A	362 / 20
Cathedral / 3.2M	N/A	N/A	697 / 14

6. CONCLUSION

In this work the task of building of full SAH BVH on GPU has been studied. The given methods for adapting of the general algorithm allowed to improve the results at more than 10 times compared to the best implementation known [7] (taking into account the difference in the hardware used – up to 5 times). Moreover, the obtained timing estimates are comparable with the estimates for the "compromise" structures (providing for rapid construction, but less effective for the ray tracing – Hybrid BVH, Two-level Grids), obtained in recent works [7], [13]. The current implementation allows to perform the rendering of arbitrary dynamic scenes of up to 800K of triangles and static scenes of up to 5M of triangles.

7. REFERENCES

[1] Timothy J. Purcell. *Ray Tracing on a Stream Processor*. Ph.D. dissertation, Stanford University, March 2004.

[2] Foley T., Sugerman J. *KD-Tree Acceleration Structures for a GPU Raytracer*. In Proceedings of the ACM SIGGRAPH/ Eurographics conf. on Graphics hardware, pp. 15–22, 2005.

[3] J. Gunther, S. Popov, H. Seidel, P. Slusallek. *Real-time Ray Tracing on GPU with BVH-based Packet Traversal*. In Proc. of the IEEE Symposium on Interactive Ray Tracing, 2007.

[4] I. Wald, W. Mark, J. Gunther, ... , P. Shirley. *State of the Art in Ray Tracing Animated Scenes*. Computer Graphics Forum 28(6), 2009, pp. 1691–1722.

[5] J. Goldsmith and J. Salmon. *Automatic Creation of Object Hierarchies for Ray Tracing*. IEEE Computer Graphics and Applications, vol. 7, no. 5, pp. 14–20, 1987.

[6] I. Wald. *On fast Construction of SAH-based Bounding Volume Hierarchies*. In Proceedings of the Eurographics Symposium on Interactive Ray Tracing, 2007, pp. 33–40.

[7] C. Lauterbach, … , D. Manocha. *Fast BVH Construction on GPUs.* Computer Graphics Forum, 28, 2, 375–384, 2009.

[8] I. Wald. *Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture*. IEEE Transactions on Visualization and Computer Graphics, 2010.

[9] Thrane N., Simonsen L. *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis University of Aarhus (2005).

[10] Zhou K., ... , Guo B. *Real-time KD-tree construction on graphics hardware*. ACM Trans. Graph. 27, 5, 1–11, 2008.

[11] Satish N., Harris M., Garland M. *Designing efficient sorting algorithms for manycore GPUs*.

[12] Harris M. *Parallel Prefix Sum (Scan) with CUDA*.

[13] J. Kalojanov, … , P. Slusallek. *Two-level Grids for Ray Tracing on GPUs*. In Proc. of the Eurographics conf., 2011.