A modular GPU raytracer using OpenCL for non-interactive graphics

Henrique Nunes Jung[*] Vinicius Jurinic Cassol[†]

Universidade do Vale do Rio dos Sinos, Escola Politécnica, Brazil

Abstract

We describe the development of a modular plugin based raytracer renderer called *RenderGirl* suitable for running inside the OpenCL framework. We aim to take advantage of heterogeneous computing devices such as GPUs and many-core CPUs, focusing on parallelism. We implemented the traditional partitioning scheme called *bounding volume hierarchies*, where each scene is hierarchically subdivided into axis-aligned bounding boxes, so a ray may only need to traverse a subset of geometry by traversing the BVH and discarding objects it surely cannot hit, optimizing the rendering process. These structures were implemented on a many-core high parallel architecture suitable for OpenCL, which needed a specific binary tree structure implementation ready for stackless traversal on GPUs. RenderGirl is split between two main portions: Core and Interface, where the Core portions provide the bulk of ray-tracing operations and manage the communication with OpenCL; and the interfaces provide communication with a given host program, seeking modularity. In this paper we describe our results and performance gains with our partitioning scheme.

Keywords: GPGPU, OpenCL, computer graphics, raytracing

1 Introduction

Platforms for general purpose computing using GPUs started when NVIDIA[1] released its GPGPU platform called CUDA[2], which is capable of running only on NVIDIA hardware. OpenCL[3] was born sometime later using the label *heterogeneous computing*, in order to specify that the platform was supported by many kinds of hardware and computer processors, including CPUs and non-NVIDIA GPUs, which could be implemented by any vendor interested in developing hardware focused on parallel computation.

Within this context, this work aims to provide a modular plugin-based raytracer software that takes advantage of the parallel capabilities of modern GPUs while maintaining hardware agnosticism by using the portability OpenCL provides to hardware vendors and developers. By being modular, it's also capable of running inside larger 3D suites such as Blender, this way we can delegate other tasks to the host 3D program and focus on the rendering tasks. We developed a C++ application and a reference plugin for Blender, used to validate this architecture. The code has only portable system calls and OpenCL function calls.

There are several acceleration structures meant to increase the efficiency of rendering, which are called *spatial data structures*. Some of these structures are *bounding volume hierarchies* (BVH), *binary space partitioning*, trees, quad-trees and octrees. They hierarchically subdivide a scene so that the queries for their objects become faster [1, Chapter 14.1]. For RenderGirl, we developed a variant of the BVH partitioning structure suitable for stackless traversal on the GPU.

Our contributions include:

- A renderer independent of a given 3D software.
- A raytracer designed to take advantage of parallel computation capabilities of modern GPUs and other OpenCL-capable devices.
- A modular architecture with replaceable components that communicates with a known interface.
- Usage of acceleration structures suitable for storing scene information inside OpenCL architecture.

2 Related Work

Applications that provide GPU rendering capabilities for non-interactive graphics include Blender Cycles[4] , Octane render [5], V-Ray renderer [6], Redshift [7] and Indigo renderer [8]. Indigo and Cycles both claim to run using the OpenCL framework. Cycles is the only one that is free software; originally it only supported CUDA-capable devices, but newer versions of Blender ship with Cycles that works with limited features also on OpenCL, although the developers specify that the CUDA platform is more mature [9].

Recent related work on the computing literature includes: Áfra and Szirmay-Kalos propose a novel traversal algorithm for BVH that doesn't use a stack, and can be executed both on CPU and GPU platforms. They introduce the concept of *bitstack*, which is an integer used in the place of a stack [8]. Kalajanov and colleagues propose an acceleration structure using hierarchical two-level grids implemented in CUDA, which can eliminate problems arriving from using a single uniform grid for subdividing a scene. They call it the "teapot in a stadium" problem, where a great amount of objects is allocated on a single cell of the uniform grid [2]. Ravichandran and colleagues propose a parallel divide and conquer raytracing suited for GPUs. Divide and conquer ray tracing removes the need of creating an explicit acceleration structure once per frame, instead it creates an approximation using bound boxes [4]. Shumskiy and Parshin write a comparative study of ray-triangle intersection algorithms, how they perform on GPU hardware and how BVH acceleration structures can be optimized for each one. They point out that some of their results differ from generation to generation of the same line of GPUs due to changes in the micro-architecture. [5]. Wong and colleagues propose an optimization method for GPU ray tracing by dividing objects into view-sets based on light and camera position, the BVH is then built using these views-sets, this way the amount of triangles on the BVH is reduced [7]. A great portion of these works deal with acceleration structures for GPU raytracing - e.g. kd-trees, BVHs -, which indicates the complexity of these structures on GPU hardware.

[*]e-mail: henriquenj@gmail.com
[†]e-mail: vjcassol@unisinos.br

[1]NVIDIA website http://www.nvidia.com/
[2]CUDA homepage http://www.nvidia.com/object/cuda_home_new.html
[3]OpenCL on Khronos Group website https://www.khronos.org/opencl/

[4]Cycles website http://www.blender.org/manual/render/cycles/
[5]Octane website https://home.otoy.com/render/octane-render/
[6]VRAY website http://www.chaosgroup.com/en/2/vray.html
[7]Redshift website https://www.redshift3d.com/products/redshift
[8]Indigo website http://www.indigorenderer.com/
[9]Cycles GPU rendering page http://www.blender.org/manual/render/cycles/gpu_rendering.html

2.1 Contextualization

Our work differs from the previously mentioned publications by focusing on modularity and portability, both of hardware and operating system.

3 Methodology

We developed a GPU raytracer using the OpenCL framework called *RenderGirl*[10] licensed under LGPL. We try not to pose any restriction on the use of the software regarding its host program, so anyone can implement their own interfaces.

The host program of RenderGirl is Blender, which provides all the base functionality of a 3D software suite. RenderGirl connects with Blender through its Python API and implements a *RenderEngine* interface. The host software performs all communication with the final user through its own GUI.

4 Architecture

RenderGirl is composed of a *core* that communicates with a given *interface*.

The core portion is a static library that provides an API for receiving the 3D scene structure - vertices, triangles, objects, cameras and lights - and the OpenCL device selection. It performs all of the communication between OpenCL and the interfaces, handling its context. The output of the render is an array of pixels with the rendered frame. The core also provides the Log Subsystem dedicated to log operations. Most of the interactions with the raytracer occur using a shared object called *RenderGirlShared*, which is implemented as a shared singleton. Scene structures are managed by a singleton called *SceneManager*, providing an API for setting up a scene using RenderGirl data structures; it's also responsible for subdividing the scene into BVH structures and making it suitable for OpenCL memory models.

The interfaces can be anything from executables to dynamic libraries, depending on the plugin interface provided by the host 3D program. Interfaces link themselves to the core at compile time and translate all the requests from the host program. They can pick individual features to support, and a given interface may support only a subset of features of the core. Three interfaces are provided as examples, they are:

- *Console interface*: The simplest interface. It is linked to the core and compiles as an executable. It only provides options for OBJ loading and does not output any image file. It's useful for porting the Core to other platforms and making sure it's working.
- *wxWidgets interface*: compiles as a standalone application with a GUI for loading OBJ files and render models, using the wxWidgets[11] toolkit. The user can choose attributes of the scene like camera position, color of light source and the output image format. It's likely to be deprecated over the Blender plugin interface due to the amount of dependencies it carries.
- *Blender Plugin*: the Blender plugin is the interface we provide as reference design to other interfaces for RenderGirl.

4.1 OpenCL scheduling model

OpenCL dispatches each execution of the raytracer by running a piece of code called *kernel*. Each kernel runs once inside a *work-item* which belongs to a *work-group*. The exact nature of the execution will be hardware dependent - e.g. a work-item can become a thread -, we assume that each work-item can work independently with no knowledge of other work-items, and therefore the kernels do not perform any synchronization.

We launch one work-item executing one kernel per pixel of the image. Each kernel will then build a ray based on pixel location within the image to be tested for collision against the BVH structure, where each ray may reach a leaf node, in which case it must test against all the geometry of that object. This approach works well in parallel because it does not require any synchronization among work-items, hence it's well suited for GPUs.

4.2 Scene structures

In order to make a 3D scene fit into an OpenCL compatible structure, we must not rely on complex data structures such as linked lists and dynamic arrays. Pointers can only be used with a feature called *shared virtual memory*, however it's only available on OpenCL 2.0, and NVIDIA still has its implementation running on 1.2, so we didn't want to rely on SVM.

Figure 1 shows how the 3D geometry is organized within the target device. The structure is similar to the OBJ file format. The core concept are the *global buffers*, they collectively describe the geometry of the entire scene. E.g. the vertices buffer contains all vertices for the entire scene. The vertex buffer is indexed by the triangle buffer. The triangle buffer is then divided into regions that compose the objects. An object's metadata array describes where each object starts on the triangle buffer and where it ends.

4.3 Acceleration structures

Acceleration structures are commonly used on ray tracing in order to avoid expensive and inefficient brute-force ray-triangle intersection tests for the entire scene. With these structures, using a given ray, we can query for only a subset of triangles and perform the intersection tests on them.

Thrane and Simonsen conduct a study comparing three different acceleration structures and their implementations on GPUs: bounding volume hierarchies, kd-trees and uniform grids. On their experiments, they concluded that BVH outperforms the other two except for a few cases, citing the simpler nature of the traversal code with minimal branching as the most likely responsible for the good results [6]. After careful consideration of their results, we chose BVH for our own implementation.

The process of using acceleration structures is usually divided in two steps: *construction* and *traversal*. The BVH was originally proposed by Kay and Kajiya[3], they describe an implementation using convex hulls as bounding volumes for each level of the tree.

For our BVH in GPU, the *construction* portion is roughly the same as described by Kay and Kajiya. The BVH is built as a binary tree where at each level objects are sorted by their absolute position either in the X or Y axis, the collection of objects is then split into half, composing the two children of that node. The axis used for ordering are swapped at each level of the tree in an attempt to make a fair distribution of objects. This process of ordering and division repeats itself until the collection of objects is reduced to one, making it a leaf node, which then contains all the geometry of that particular object. The major difference between our approach is that we use *axis-aligned bounding boxes*(AABB) as bounding volumes instead of convex hulls, since the traversal code is less complex, although with a much less tightly fit bounding volume. Thrane and Simonsen mention that there's a great penalty of running complicated code on the GPU[6]. Every node holds an AABB fitting all the geometry of its child nodes, so a ray must only test a collision against the AABB, if the collision fails, the algorithm can discard that sub-tree and resume processing on the next sibling node. If the program reaches a leaf node, then the ray must be tested against all the geometry of that object within the leaf node. All the construction phase happens on the CPU.

On the *traversal* step, we implemented the traversal algorithm described by Thrane and Simonsen in their master thesis[6]. The main problem to be solved here is to overcome the lack of stacks on

[10]RenderGirl project page https://github.com/henriquenj/rendergirl

[11]wxWidgets website http://wxwidgets.org/

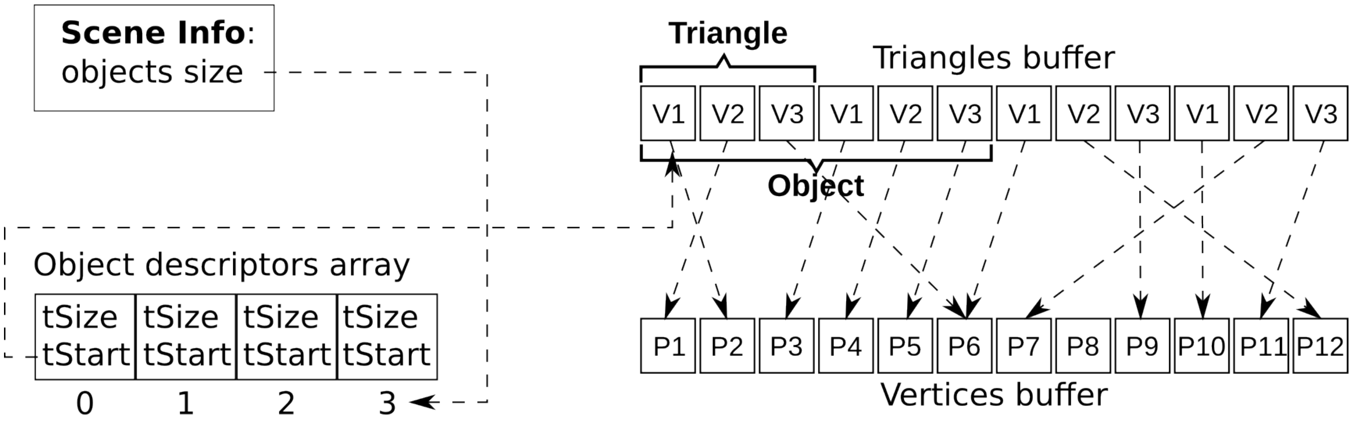

Figure 1: The relationship of each data buffer that composes the geometry within the OpenCL device. Dotted arrows represent an integer index that points to some other array. The BVH will be covered in its own section.

GPU hardware, since tree traversal algorithms are usually implemented using recursion. The key difference from traditional CPU implementations is that we build a fixed-order *traversal array* on the CPU and only iterate over it in the GPU, so we never transmit the tree to the OpenCL device but rather the iteration itself. The traversal array already contains the nodes we must intersect in the correct order. The BVH node on the traversal array has three properties: the AABB that fits all the geometry of that node and its children, the *escape index* and an index that points to an object in the global list of objects. The escape index represents the index where the traversal should resume if the current node's AABB fails to intersect with the ray. If the traversal should end, the escape index is equal to the size of the traversal array, acting as a global escape index. This can be visualized on figure 2. The object index is only valid in child nodes, and it's flagged as -1 when in a middle node. The traversal is done in a top-bottom left to right order. The algorithm for traversal is detailed on Algorithm 1.

Algorithm 1 BVH traversal on OpenCL device

```
1: Given a ray $r$
2: $tarray \leftarrow$ The traversal array
3: $s \leftarrow$ size of traversal array
4: $index \leftarrow 0$
5: while $index < s$ do
6:     $node \leftarrow$ element from $tarray$ at $index$
7:     if $r$ intersects with $node$'s AABB then
8:         if $node$ is a leaf node then
9:             Executes ray-triangle intersection with all triangles
               on this leaf node and return the nearest
10:        else
11:            $index \leftarrow index + 1$
12:        end if
13:    else
14:        $index \leftarrow node$'s escape index
15:    end if
16: end while
```

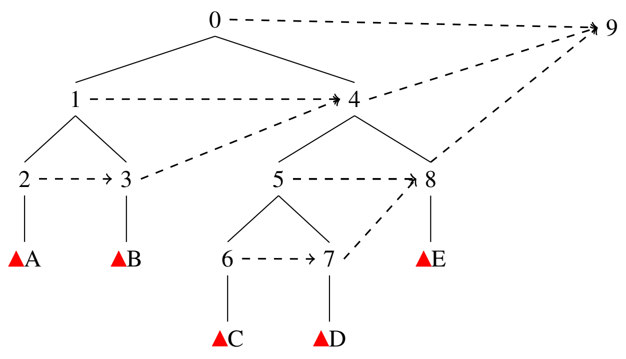

Figure 2: Visualization of the BVH as a tree. The numbers indicate the order of visiting. Dotted arrows represent where the traversal should resume if the ray fails to hit the AABB. 9 is the global escape index. Each leaf node holds an index to the global array of objects, here depicted as a triangle.

5 Results

We tested our raytracer implementation by running it from Blender using a variety of scene configurations. We aim to test different kinds of object distributions in order to test how the BVH performs on each case. We used the following test scenes:

- Scene 1: A simple scene depicting a table and some kitchen objects. It contains 21.424 vertices, 42,848 triangles divided into 28 objects.
- Scene 2: The same scene as before but with several evenly distributed tables. It contains 203.644 vertices, 405.242 triangles divided into 233 objects. See figure 3.
- Scene 3: A scene with a single whale object containing 8.432 vertices and 16.764 triangles. See figure 4.
- Scene 4: A scene depicting a cabin. It contains 93.570 vertices, 178.104 triangles divided into 250 objects. See figure 5.

The figure 6 shows ours results using the scenes described previously. All tests were executed on a NVIDIA GeForce GTX 970 using driver version 353.30. The OpenCL specification version used by this driver is 1.2.

From the graph we can see that the BVH works well with scenes that are already divided into several objects. Scene 1 and 2 are good fits for the algorithm because most leaf nodes have a low amount of geometry, and the increase of resolution plays little role in the rise of time. We can observe that the single object scene does not perform as well as the other two, since in scene 3 a single object holds all geometry, so we can observe that the BVH has little to no improvement, since all rays certainly hit the AABB covering the whale, and in the end the GPU has to test against all the geometry

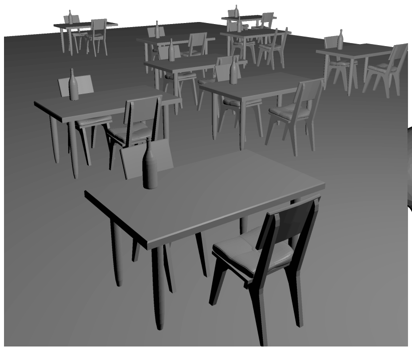
Figure 3: Scene 2 with the tables

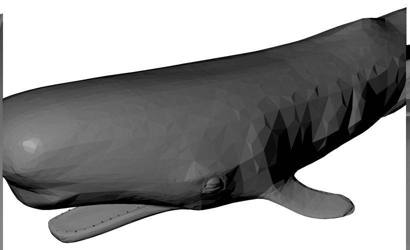
Figure 4: Scene 3 with the single whale object

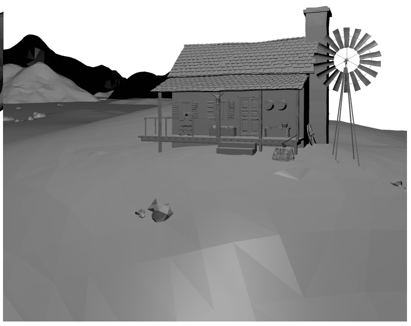
Figure 5: Scene 4 with the cabin

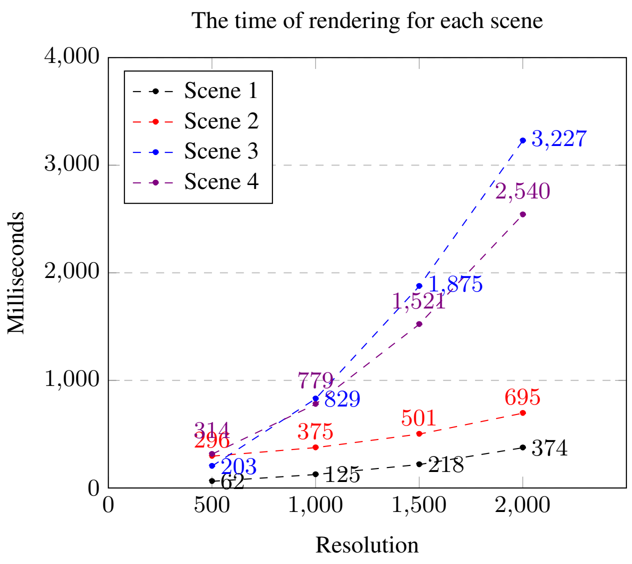

Figure 6: Rendering time of the four scenes across different resolutions. Resolution value represents both width and height.

of the object for almost all rays. This is due to the fact that the BVH partitions objects, but it does not split geometry, so it depends if the scene is already well divided into suitable chunks. An object with a large amount of triangles that covers most of the frame will certainly slow down any rendering, even if the triangle count of the scene is relatively low. Scene 4 also performs poorly even though it has a good amount of objects, the reason for this is that we have a floor object on the scene that contains a great amount of triangles and covers most of the frame, creating a situation like the whale scene.

6 Conclusions and future work

The BVH performed well within the scope of its ability, providing good speedups , even if with some known weaknesses of the acceleration structure. For a future development, we could implement an hybrid of kd-tree and BVH by partitioning objects that are above a given threshold of triangle count, which would speed up cases like the whale scene. Although we must still exercise caution on the complexity of the partitioning scheme, since the generation of the structures are done on the CPU side and it's serialized, while the traversal and ray intersection operations takes full advantage of the OpenCL device. For instance a BVH that takes 3 seconds more to generate must save at least 3 seconds worth of processing on the device, so a solution for the worst case of the BVH should taken into consideration the whole context of the application.

The plugin architecture was validated by embedding the Core on three different interfaces that can make use of the same library, e.g Blender can run the unmodified RenderGirl that is also embedded on the other two interfaces. We can also take advantage of all the facilities Blender provides, such as image saving, 3D geometry loading and triangulation features. This spare us of having to implement all from scratch, enabling our work to be focused on the rendering itself.

References

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on gpus. 30:307–314, April 2011.

[3] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *ACM SIGGRAPH computer graphics*, volume 20, pages 269–278. ACM, 1986.

[4] S. Ravichandran and P. J. Narayanan. Parallel divide and conquer ray tracing. *SIGGRAPH Asia 2013*, November 2013.

[5] V. Shumskiy and A. Parshin. Gpu ray tracing – comparative study of ray-triangle intersection algorithms. *GraphiCon'2012*, pages 61–66, October 2012.

[6] N. Thrane, L. O. Simonsen, and A. P. Ørbæk. A comparison of acceleration structures for gpu assisted ray tracing. Technical report, 2005.

[7] S.-K. Wong, Y.-C. Cheng, and S.-Y. Lii. Gpu ray tracing based on reduced bounding volume hierarchies. pages 1–6, July 2012.

[8] A. T. Áfra and L. Szirmay-Kalos. Stackless multi-bvh traversal for cpu, mic and gpu ray tracing. 33:129–140, November 2013.