The Open University of Israel
Department of Mathematics and Computer Science

Cross-Hardware GPGPU implementation of Acceleration Structures for Ray Tracing using OpenCL

Advanced Project in Computer Science (Course 22997)
Open University of Israel
Computer Science Division

Prepared By:
Timur Sizov

Prepared under the supervision of Dr. Mireille Avigal

06-06-2016

Background

Ray tracing is a basic algorithm for rendering 3D objects and scenes. Its basic principle is to determine a color for each pixel by casting a ray from the pixel, determining the closest point where that ray hits the object and determining the color of pixel according to the surface that was hit. While the algorithm is very simple and being able to render high quality images, its high demand of processing resources made it impractical for massive use. It could take a whole night to ray-trace a single high-quality image.

The most important optimization to make Ray Tracing fast enough for any practical use is utilization of **Acceleration Structures** – Data structures for spatial sorting that enable reducing the quantity of Ray-Object intersection tests for each ray. Unlike brute force approach, when each ray has to be tested against each object in the scene, usage of Acceleration Structures allow fast elimination of irrelevant objects that are far enough from the ray to be considered, and finding small amount of candidate objectswhich are close to the ray and therefore have a bigger chance to intersect with the ray.

Another property of the Ray Tracing algorithm that can be exploited for reducing rendering times is that it is "parallelism-friendly" –Each ray or group of rays can be processed as a separate independent task. This allows recruiting the power of multi-processor hardware for accelerating Ray Tracing.

An example of such hardware is the **Graphics Processing Unit** – The GPU. The GPU is a processor designated for massive data-parallel computations. Mostly it consists of several multi-processors that execute large amounts of parallel threads.
While initially those processors were designed for raster graphics processing, it turned out that those processors were useful for other massive data-parallel calculations as well.

Programming for GPU became significantly simpler with emergence of **General Purpose Graphics Processing Unit (GPGPU) platforms**. Those platforms include API and runtime libraries for creating and running programs on supporting graphics hardware. Introduction of those platforms in mid/late 2000's opened a large research area for adapting Ray Tracing related algorithms and acceleration data structures for those technologies to utilize the additional processing power and parallelism offered by those units.

The dominating GPGPU platforms up to date are NVIDIA CUDA and OpenCL.
CUDA is a proprietary platform which can be used only with hardware manufactured by NVIDIA. OpenCL is an open standard which can be implemented by any hardware vendor, and ideally, any program that was written for OpenCL platform should run on any hardware that supports OpenCL.

In majority of the studies up to date the acceleration structures are implemented using NVIDIA CUDA for testing and comparing their performance – Since the platform is well documented, contains many useful libraries, and that the platform is optimized for

specific hardware, makes it a good choice for development. However, those advantages come at the price of being able to use those implementations with NVIDIA hardware only.

This software project is a **cross-hardware** implementation of commonly used acceleration structures on a cross-hardware GPU platform - OpenCL. The final product of this project is a class library that provides API for using acceleration structures

Contents

List of Figures

1. Introduction

1.1. The Ray Tracing Algorithm

Ray Tracing is a rendering algorithm that generates images by tracing the path of light through pixels in an image plane and simulates the effects of the interaction between light and the surface of objects in the rendered scene. [5]
For each pixel the ray tracer finds the object that is "seen" from its position, in the direction where the pixel is "looking" and calculates the contribution of that object to color of the pixel, considering intersection point of viewing ray (a "line of sight" of the pixel) and the object, surface normal, material properties, and other information.

The basic Ray Tracing algorithm is composed of three basic stages, which are being performed on one pixel at a time: [2]

- **Ray Generation** – Computing the origin and direction of each pixel's viewing ray, based on the chosen camera model
- **Ray Intersection** – Finds the closest object that intersect the viewing ray
- **Shading** - Computing the pixel color based on the results of ray intersection.

If we implement the stages above we will get a very basic ray tracer with limited capabilities. A development by Turner Whitted in 1979 introduced additional important part of the algorithm: A recursive tracing of additional rays for simulation of light interaction with surfaces of objects [6]. This is a part of the Shading stage of the Ray Tracing algorithm, which is now extended to casting rays from surface point to incident directions to determine how much light comes from each direction and recursive tracing of light backward from the surface to the light sources.

Due to importance of this development, the term Ray Tracing refers to Whitted Ray Tracing and vice versa, and the basic algorithm without the recursive tracing is called **Ray Casting**.

The next step beyond Whitted Ray Tracing is **Distribution Ray Tracing.** [7]
The Distribution Ray Tracing is a technique that uses multiple viewing rays. Perhaps the first motivation to those techniques is **Anti-Aliasing**, but they also enable complex effects such as Depth of Field, Soft Shadows and Motion Blur. Those techniques often involve casting rays from random sample points of single pixel which is actually applying the **Monte Carlo Method** to Ray Tracing. For this reason, Distribution Ray Tracing is sometimes called **Stochastic Ray Tracing**.

So far we can observe that the more "evolved" the algorithm becomes, the more rays we will need to cast. Naive casting of millions of rays and checking each ray for intersection with each primitive is obviously not practical, so we need some kind of spatial data structure that will help eliminating intersection checks with objects which are too far away from the ray, and significantly reducing the already large amount of computational work.

The wide variety of spatial data structures, or acceleration structures, include **Uniform Grids**, **Bounding Volume Hierarchies**, **Octrees**, [1], **K-D Trees** [9] and **BSP Trees** [3] – Each data structure has its own characteristics, and while choosing the best performing acceleration structure we should consider factor like scene characteristics – Whether it is sparse or dense, the construction time, traversal time, available effective algorithms for construction and traversal, and more.

This project provides implementation of GPGPU versions of two acceleration structures: Bounding Volume Hierarchy, and Two Level Grid. The implemented construction and traversal algorithms are the most effective algorithms to date.

1.2. Ray Tracing and GPGPU

Attempts to utilize graphics hardware for general computations and Ray Tracing in particular, were made shortly after the appearance of GPU units on the mass market, in early 2000's.
This was before the development of GPGPU technologies, so the computation problems had to be formulated in terms primitives of graphic APIs such as OpenGL in order to utilize the GPU.

One of the earliest works on the subject of GPGPU Ray Tracing is "Ray Tracing on Programmable Graphics Hardware" by Timothy J. Purcell et al [18]. This work proposes a mapping of Ray Tracing to the GPU programming model and analyzes various aspects of implementation and provides results comparison.

Later, with the appearance of GPGPU technologies such as CUDA and OpenCL, additional works appeared, that utilized those technologies [19, 20].
There were also attempts to use hybrid architecture – To utilize GPU and CPU simultaneously, to achieve interactive speed of Ray Tracing [21].

In addition to academic efforts, hardware vendors such as NVIDIA and Intel constantly encourage developers and researches to use their technologies to implement Ray Tracers. One of the ways to do so is providing specialized frameworks, such as NVIDIA OptiX.

At this day, the research on GPGPU Ray Tracing continues to produce more and more effective algorithms for construction and traversal of acceleration structures.

2. Requirements

2.1. Scope

The final software product of this project is a **Cross-Hardware Class Library** which provides an API for using GPGPU implementations of acceleration structures for Ray Tracing.
The variety of acceleration structures is limited to:

- Bounding Volume Hierarchy
- Two Level Grid

The provided set of functions is limited to acceleration structure related operations, and doesn't include implementation of shading related algorithms.

Ideally, cross-hardware libraries should provide full portability and perform equally well on any hardware. In practice however, portability issues always can arise, possibly due to bugs in device drivers, and varying implementation of open standards by different hardware vendors. Since it is not practical to test the library for large variety of hardware, the implementation does not guarantee full absence of portability issues.

2.2. Functional Requirements

The general requirement of the API provided by the library is to provide functionality that will suffice to implement a basic Ray Tracer. Going to a lower granularity level of functional requirements will yield functions described below.

2.2.1. Rendered Scene related operations

- The API shall provide functionality for loading 3D models in format which is widely used in the 3D modeling industry.

- The API shall provide functionality for retrieving data about the scene. Access should be provided for the following data:

    - Retrieving primitives (Triangles) by index
    - Retrieving models by index
    - Retrieving general data – Number of models, number of primitives, and other scene items.

2.2.2. Acceleration Structure related operations

- The API shall provide functionality to construct the implemented data structures using processing power and parallelism of GPGPU hardware. The data structures shall be constructed according to the scene that the user desires to render, according to parameters defined by user when applicable.

- The API shall provide functionality to determine the closest intersection of a given ray with primitive in the scene. The resulting data shall include:
    - Index of the intersected primitive with the scene
    - Distance between intersection point and ray origin
    - The normal of the surface of intersected primitive at the intersection point

  The functionality shall utilize the processing power and massive parallelism of GPGPU hardware to maximize performance.

2.3. Non Functional Requirements

2.3.1. Portability

The software shall be able to compile and run on any GPGPU hardware that supports the selected cross-hardware platform on which the software is implemented. Since there is no practical way to avoid portability issues, the effort that is required to put the software in use for specific hardware shall be limited to resolution of specific portability issue.

2.3.2. Extensibility

The software design shall allow adding implementations of additional acceleration structures as part of future work. Also, the software design shall allow easy optimization and migration to higher versions of the underlying platform.

2.3.3. Documentation

The functionality and classes provided by the library shall be documented well enough for the user to be able to understand the functionality and use it correctly.

3. GPGPU Programming Model

In order to deduce constraints introduced by GPGPU platform and hardware, we need to get familiar with the programming model ofa low level GPGPU cross-hardware platform - OpenCL. Similarly to the proprietary platform, NVIDIA CUDA, OpenCL execution and memory model is designed to conform to graphics hardware architecture.
The following sections shall describe the programming, memory and execution models in deeper detail. Since OpenCL is an open standard, in this work I will arbitrary refer to AMD APP SDK documentation [15] – A useful implementation of OpenCL standard by widely known graphics hardware manufacturer.

3.1. Execution Model

OpenCL exposes a C API for GPU interoperability. The language and the API are sometimes referred to as OpenCL C [13]
In OpenCL, we typically write a program that starts it's execution on CPU – It starts execution from a regular *main* function, and when necessary, calls a function that should run on GPU. A function that should be executed on GPU is called a **Kernel**. When we want to say that some part of program should run on CPU or some data should be stored in the main memory (RAM), we say that the part runs on **Host**, and the data is stored in **Host Memory.** On the contrary, when we want to say that some part of program should

run on GPU, we say that the runs on **Device**, and uses data that is stored in **Device Memory.**
This is due to that the GPU programming model assumes that the Device is a coprocessor for Host, and Device and Host maintain different separate memory spaces.

The kernels are executed N times in parallel, by N threads over N – dimensional computation domain.
The execution of these threads is arranged in a hierarchy: The threads are grouped to one, two or three-dimensional **Work Groups**.
Due to specifics of hardware architecture, Work Group size is limited – Therefore we may need to run several Work Groups in a Global Work Group. The global size also can be specified in one, two, or three dimensions.

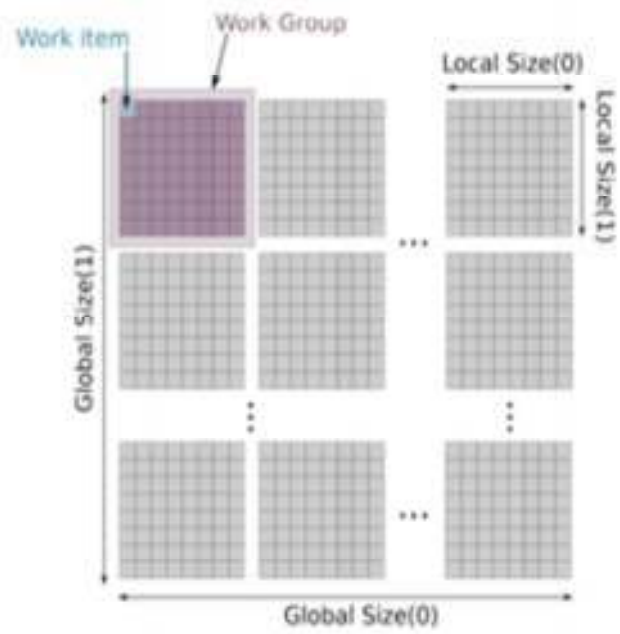

Figure 1 OpenCL Work Groups and Work Items hierarchy [15]

The threads are executed in a so-called **SIMT – Single Instruction Multiple Threads** fashion, while many executing threads execute the same instruction using different data items. The smallest execution unit that can be executed on device is a group of threads. In In OpenCL implementation, this group is called **Wavefront**. The amount of threads in this group is hardware dependent, and may vary from device to device: In devices manufactured by AMD the wavefront size is often 64, while in some Intel devices the Wavefront size is 16.
The relation of wavefront to workgroup is that work group consists of one or more wavefronts. Each wavefront can be executed independently of the rest of the wavefronts, even if they belong to the same workgroup. In AMD OpenCL, therefore, work group threads 0 – 63 will be executed in wavefront 0, threads 64 – 127 – on wavefront 1, etc. As can be concluded, for optimum hardware usage, the work group size should be a multiple of wavefront size.

3.2. Memory Model

The memory hierarchy of OpenCL was designed to loosely resemble the physical configuration of ATI and NVIDIA hardware. [13] According to the hierarchy, each thread has its own private memory space, called **Private Memory**.
Further, each work group has a **Local Memory** which is shared by the working items executing in the same workgroup.
Finally, at the top of the device memory hierarchy there is a **Global Memory**, which can be accessed by all executing kernels as well as the host, and the **Constant Memory** – A read-only region for host-created and allocated objects that will not change during kernel execution. [15]

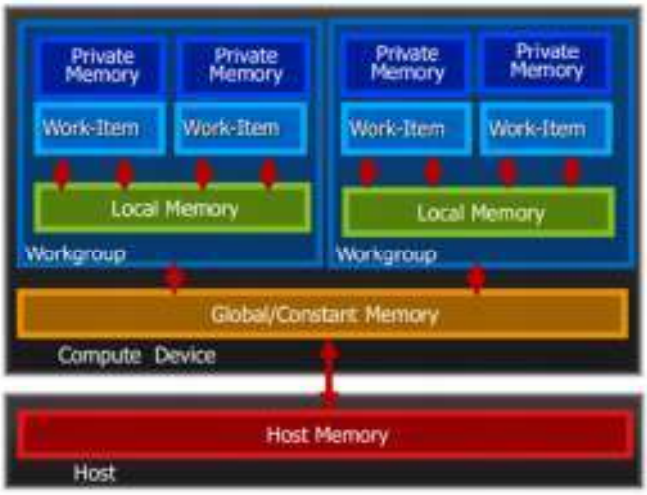

Figure 2 OpenCL Memory Model [15]

3.3. Constraints and Pitfalls of GPGPU Programming

The GPGPU architecture marginally differs from the traditional "Computer" architecture and those differences should be considered when programming for GPGPU platform. Familiarity with the details of the GPGPU architecture and best practices is important for effective utilization of the hardware and gaining the best performance for programs and algorithms – Those practices are actually the main guidelines for a good design for a GPGPU program.

The most important points to consider about GPGPU programming are summarized below:

- **Flow Control -** As mentioned before, each thread in wavefront or warp executes the same instruction at any given time. Diverging branches of execution, such as *if* statements result in serial execution of both control branches. There are optimizations for these cases, implemented in various technologies but in general branch divergence should be minimized for maximal performance.
- **Host – Device Memory Transfer –** The memory transfer is very slow, and the recommended practice is to avoid such memory transfers, or just to preload data once, perform several actions on it and fetch the results from the device at once.

- **Recursion –** Recursion is supported only by the newer versions of GPGPU APIs and more modern graphics hardware. This is due to hardware architecture, that doesn't feature a call stack. The most recent API versions do support recursion, although it is still a better practice to avoid it and use iterative versions of algorithms where possible.
- **Registers Economy –** The local variables of kernels physically reside in a designated registers. Quantity of those registers may change between various models of specific hardware. In case when the quantity of registers becomes insufficient, the variables that have no room in registers are "spilled" to local memory – resulting in slower access to those variables, and reduced number of active wavefronts during execution. Since such a situation result in performance penalties, register usage is a significant factor to consider when implementing an algorithm on GPGPU.

While avoiding recursion and host-device data transfer are significant factors that affect algorithm design, the branch divergence and register usage have significant influence on kernel execution time. Since the modern GPU exhibit a **Single Instruction Multiple Threads (SIMT)** execution model. This means that a single instruction is executed by multiple threads on different data items. This implies that flow control divergence reduces performance because all the control branches get executed serially, leading to waste of computing resources.

The usage of private memory registers has a direct effect on an important GPGPU specific performance metric: **Kernel Occupancy**. Kernel Occupancy is the ratio between the number of actually running threads, and the maximal number of threads that can run simultaneously on the hardware. For example, consider a GPGPU device has 4 compute units, each can run 64 threads simultaneously (A single OpenCL wavefront). Each compute unit also has 16 KB of registers. In order to utilize all of the execution capacity of the device, each thread can use no more than 256 bytes of memory for local variables. In case when a kernel uses more than that amount of memory, say 384 bytes, we can run only 42 threads simultaneously – Because that is the amount of threads that uses up all of the registers of compute unit. Therefore, we have a 42/64 = 65% occupancy in our example.
Maximizing occupancy motivated development of designated design guidelines and techniques that help reducing the impact of GPGPU hardware constraints and maximize hardware utilization and kernel occupancy.

4. Essential Background for Ray Tracing

The purpose of this chapter is to provide background and introduce terms which are essential for understanding basic Ray Tracing and proceeding forward with acceleration structures.
Reminding the short introduction of Ray Tracing in section 1.2, **Ray Tracing** is a rendering algorithm that generates images by tracing the path of light through pixels in an image plane and simulates the effects of the interaction between light and the surface of objects in the rendered scene.

The algorithm consists of three basic stages, which can be modified and extended to achieve different rendering quality and effects:

- **Ray Generation** – Computing the origin and direction of each pixel's viewing ray, based on the chosen camera model
- **Ray Intersection** – Finds the closest object that intersect the viewing ray
- **Shading** - Computing the pixel color based on the results of ray intersection.

4.1. Ray Classification

The rays traced during the run of the algorithm can fulfill different purposes. We can classify the rays by their purpose for the rendering:

- **Viewing Rays –** The rays that being cast from each pixel. This ray is used to determine the closest intersection point with an object in the rendered scene. For this point we need to determine the color that it contributes to the pixel. The Viewing Rays are also called **Eye Rays** or **Primary Rays**.
- **Reflection Rays –** Rays that originate at surface of an object in mirror direction – represent the light reflected by surface.
- **Refraction Rays** –Rays that originate at surface of an object in the direction of the refracted light that passes through the object.
- **Shadow Rays** –Rays that originate from surface of an object towards light source

Sometimes the Viewing Rays are referred to as **Primary Rays**, and the Reflection, Refraction, and Shadow rays are referred to as **Secondary Rays.**

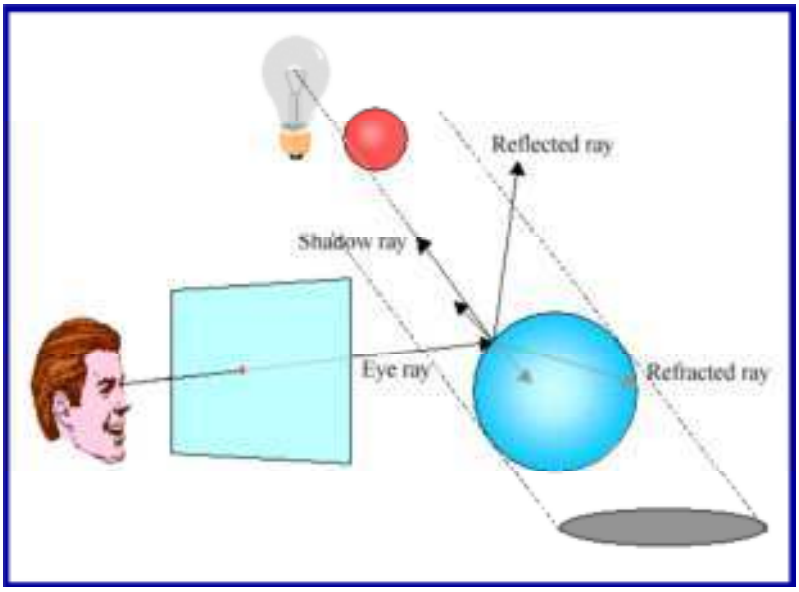

Figure 3 Classes of rays in ray tracing [3]

4.2. Recursive Ray Tracing: An Example

To get some idea about how a scene is being ray-traced, consider a following example:

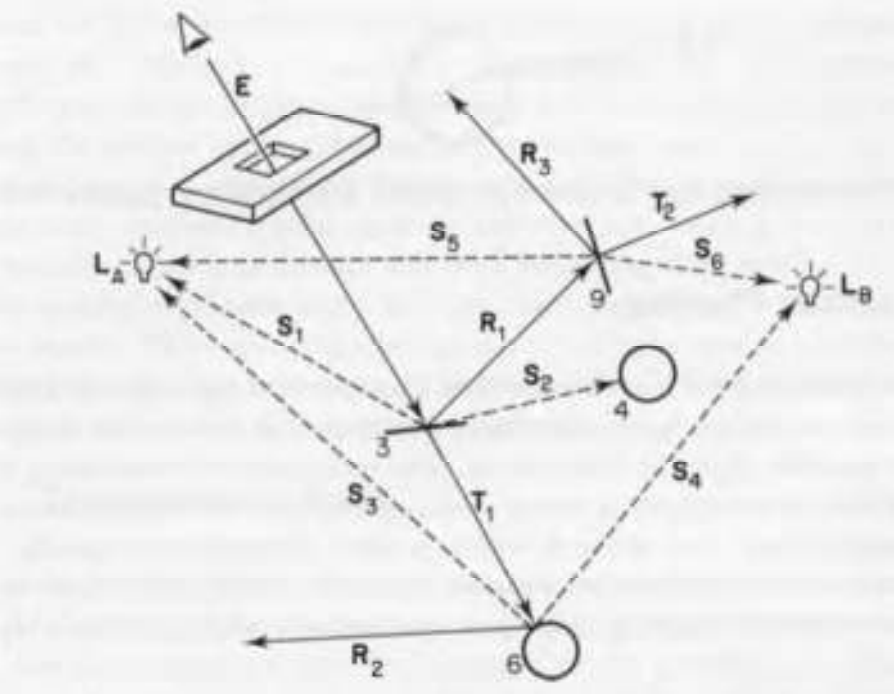

Figure 4 Ray propagation when rendering a scene [1]

The viewing ray **E** hits plane **3**, which reflects some amount of light and lets some light pass through itself (The plane has some degree of transparency).
At the next stage, shadow rays sent to each light source in the scene: Ray **S1** hits a light source directly, so the surface receives light from that source. Ray **S2** is sent towards light source **Lb**, but it is blocked by Sphere **4**, so the surface doesn't receive any light from **Lb**. Because the plane **3** is somewhat transparent and reflective, we cast a reflection ray **R1** and refraction (transparency) ray **T1** to find the light that the surface reflects and refracts. Ray **T1** hits sphere **6**, which is reflective. Then, yet again we cast shadow rays towards light source to determine the illumination of sphere **6**, and a reflection ray **R2**. According to our scene, the shadow rays hit their light sources and **R2** leaves the scene, so its contribution to the color of **T1** will be the background color. The ray **R1** hits plane **9** which is both reflective and transparent. Again, we send shadow rays to light sources, and reflection and refraction rays **T2** and **R3.**
(Notice that the shadow ray **S6** goes toward the light source through the surface).
The same process continues recursively, until the scene is completely rendered.

The process can be schematically described by a tree, which will give us another perspective on the recursive handling of rays:

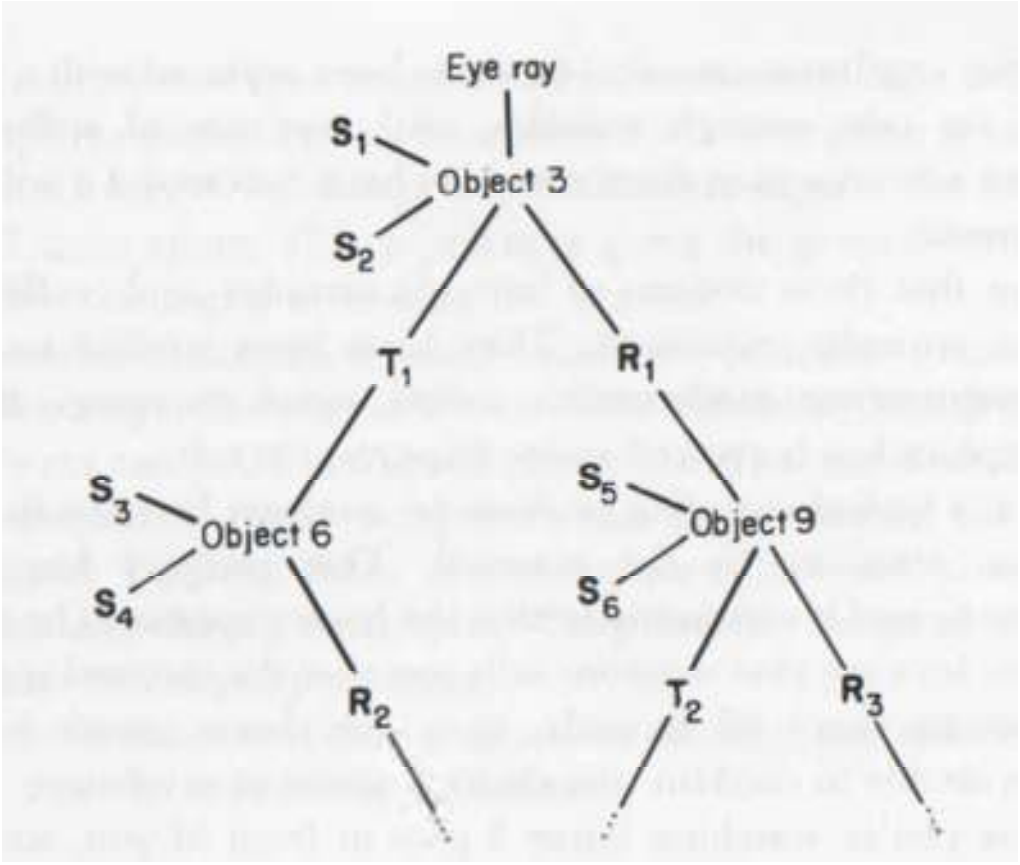

Figure 5 A tree representation of recursive ray tracing [1]

In this case we could stop following rays when all the secondary rays left the scene. In small scene like the one in the example this is not a problem. In larger scenes, however, an infinite number of rays may be generated and we will have to stop following rays when the contribution to the color of primary ray will drop below a certain threshold. This technique is called Adaptive Tree Depth Control.[1]

4.3. Basic building blocks of Ray Tracing

4.3.1. Ray Generation

The first stage in a Ray Tracing algorithm is the Ray Generation. The method we use for Ray Generation defines the **Camera Model** of a ray tracer one wants to build.In the most primitive case it will be straightforward – Cast a ray (Or several rays if we use the Distribution Ray Tracing) from the pixel to the direction of the normal of the **View Plane.** This will give the **Orthographic Projection**, which is very basic and limited – for example it will not make object appear smaller when they get farther from the camera. To get more realistic image we use **Perspective Projection.**

This is essentially a simulation of a **Pinhole Camera**. All the viewing rays are generated from a single point, called **The Center Of Projection** [3], that is located behind the view plane, and each ray crosses the center of the pixel (Or a point within the pixel, which was generated by a sample technique if we use Distribution Ray Tracing).

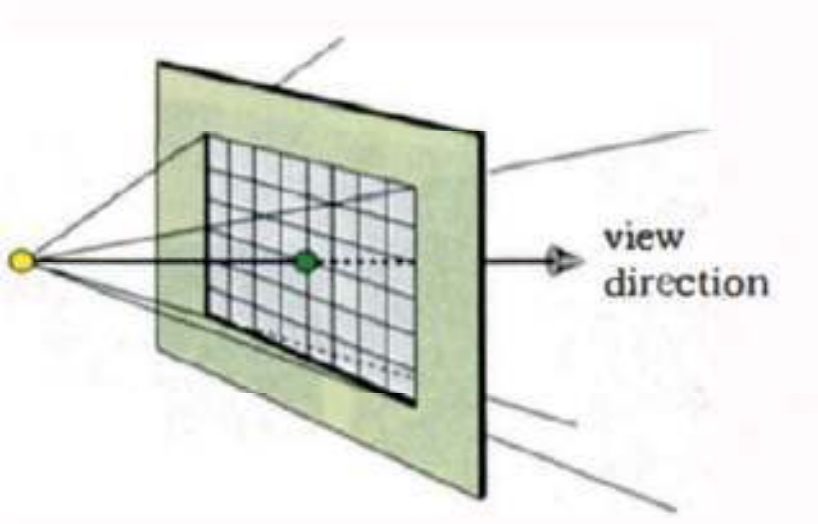

Figure 6 Perspective Projection [3]

Perspective Projection gives much more realistic results: Except for the property that perspective projection of an object becomes smaller when it gets farther away from center of projection [3] it has additional properties such as reduction of projected width when object is rotated, and more properties that makes Perspective Projection perhaps the most common camera model. Of course, Ray Tracing allows easy incorporation of additional viewing models such as **Non-Linear Projections** – A class of projections where the rays are not necessarily straight lines. Examples for such a model are **Fish-Eye** and **Spherical Panoramic Projection.**

4.3.2. Ray Representation

As in every aspect of Computer Science, we need a mathematical representation of entities we work with – In our case that would be the most basic element – The ray.
A ray is an infinite straight line that is defined by point *o* – the Origin, and unit vector *d*– the Direction. In addition, ray has a Ray Parameter *t* such as *t*=0 at the origin, so any arbitrary point on the ray can be expressed as:

$$p = o + td$$

Although we regard the ray as starting at its origin, we allow t to be negative, so the equation above will generate an infinite straight line. We consider negative values of t because it will be essential in in ray-object intersection calculations [3].

4.3.3. Implicit Surfaces

One of the strengths of Ray Tracing algorithm is the ability to render **Implicit Surfaces,** or **Mathematical Surfaces.**
An implicit surface is defined by a three dimensional scalar function $f(x, y, z)$, which can be evaluated at each point in space[25]. The surface is the set of points defined by:

$$f(x, y, z) = C$$

To determine whether a point is inside a body, we check whether:

$$f(x, y, z) < C$$

In practice, the constant C is usually equals to zero.

For example, a sphere is an implicit surface defined by the equation:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0$$

where (x_0, y_o, z_0) is the center of the sphere, r is the sphere radius, and (x, y, z) is the tested point.
Given the location of center and radius of a sphere, we can determine whether a tested point is located on the surface of the sphere by substituting the values into the sphere equation. If the equation holds true, the test point is located on the surface of a sphere. Similarly, if we would like to check whether a point is located inside a sphere, we again substitute values into the equation, and check whether the resulting value is smaller than zero:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 <= 0$$

In Ray Tracing, we use this function to test whether a ray hits the surfaces. In general, if we are able to find intersection of such function with a ray and find the surface normal, then we can ray-trace the surface. In rasterization algorithms we need to tessellate the complex surfaces to triangles in order to draw them. Since tessellation is an approximation of the actual surface, the rendering is less accurate than rendering of implicit surfaces.

4.3.4. Basic Ray-Object Intersection

As mentioned earlier, one of the virtues of Ray Tracing is the ability to render mathematical surfaces. All we need is just to be able to calculate ray-surface intersection. In terms of ray parameter *t*, our task will be to find an intersection point between a ray and some object in the scene, for which the ray parameter *t* is minimal.
The simplest example of ray-object intersection is a Ray-Plane intersection.

A plane is an infinite flat surface, which can be defined as a point *a*, which lies on a plane, and a normal *n*, a unit vector in perpendicular direction to the plane. This is sufficient to define a plane uniquely, because there is only one plane that passes through a given point and has the orientation specified by the normal. Since the plane is flat, all the points on the plane have the same normal, therefore if we pick an arbitrary point p on a plane, the vector (p – a) will lie on that plane, and it will be perpendicular to the normal.
We can then express the equation of a plane in terms of dot product of (p – a) and n as follows:

$$(p - a) * n = 0$$

In order to find intersection of plane and a ray we can substitute the ray equation into plane equation to get:

$$\big((o + td) - a\big) * n = 0$$

We can use this equation to calculate *t*:

$$t = \frac{(a - o) * n}{(d * n)}$$

Equation above is a linear equation, and since the equation will hold true only for one value of parameter *t*, the ray can cross a plane only once. After we calculated the parameter *t*, we can find the hit point by substituting it into the ray equation. Since for shading we will need not only the hit point but also the surface normal at the hit point we will need to calculate the surface normal at the hit point. Fortunately, in case of plane, the plane normal, *n,* is the surface normal at any point on the plane, so we don't need any extra calculations to determine surface normal at the hit point – we just use the plane normal, *n*. In case the ray does not hit the plane,

$$d * n = 0$$

The ray-plane intersection is as simple as ray-object intersections can go. Along with simple geometric primitives such as spheres or boxes, there are more complex surfaces, polygons, and even composite objects which composed from several primitives using methods such as **Constructive Solid Geometry.**

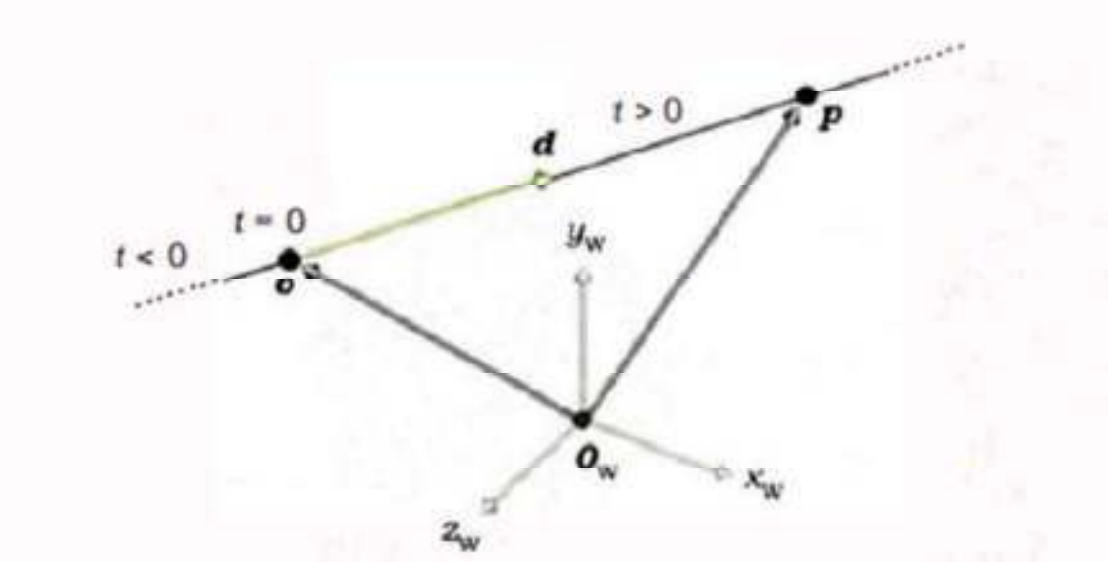

Figure 7Mathematical representation of a ray [3]

4.4. Acceleration Structures and Ray Tracing

As we may conclude from example in section 5.2, we need to trace very large quantities of raysin order to achieve good rendering quality. Even if we generate only viewing rays for image with resolution of 1024x1024, we already need to process over million rays. If we add effects such as field of depth and antialiasing, we may have to multiply that number by factor that may be 16 for example – So we reach over 16 million rays without even getting to tracing secondary rays.

Acceleration Structures are used in Ray Tracing to increase rendering speed by eliminating irrelevant objects and reduce the set of objects that are tested for intersection to a small number. Acceleration structures include BVH trees, KD trees, Uniform Grid, Two Level Grid, BSP trees, and many more. Each structure performs differently for different scene, so there is no single answer for the question "Which structure performs best". This project provides implementation of two acceleration structures: BVH and Two Level Grid. These structures are conceptually simple (Although the implementation itself may be tricky at several points), and there are advanced construction algorithms for GPGPU for construction and traversal for these structures.
Each of those structures perform differently with various scenes and have different ratio of construction and traversal times.

Now, after getting familiar with the basic building blocks of Ray Tracing, we are ready to proceed with acceleration structures.

5. Software Design

5.1. Top Level Architecture

The functions for interaction between host and device are provided by the OpenCL API. This API is a C language API which provides functionality for compiling and running OpenCL kernels on device, managing device memory, data transfers between host and device, as well as gathering device information, compiling the kernels out of source files, initializing devices, and other routine tasks.
Since usage of the C API results in hefty amounts of repetitive code, the functions were encapsulated into C++ objects that provide more robust and elegant functionality to the user, while the implementation of those functions contains all the "dirty work".
This approach significantly simplifies the code for the algorithms, and simplifies error reporting and management.
In order to achieve better functionality separation and make the design cleaner, the OpenCL wrapper is compiled as a separate class library.

The algorithm logic is contained in the Algorithms library – This library shall contain the main Acceleration Structure construction and traversal algorithms, as well as their building blocks such as geometric operations, parallel sorting, and others.

The top level architecture is summarized in the figure below

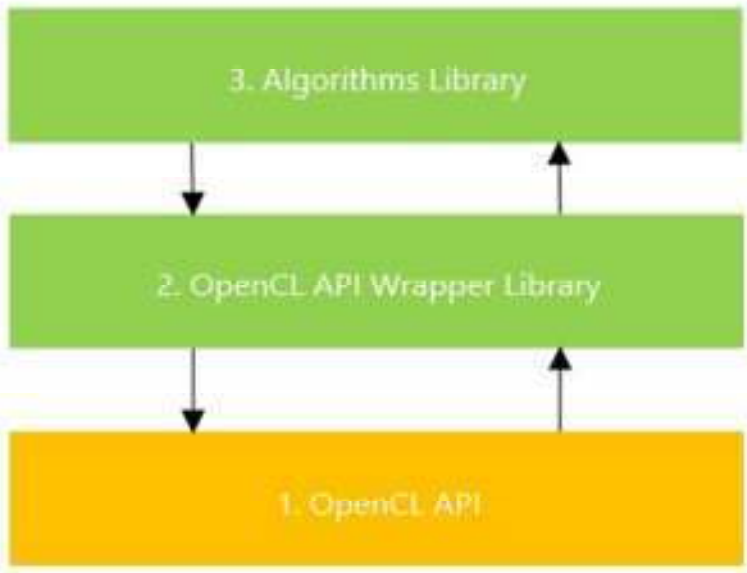

Figure 8Acceleration Structures Library - Top Level Architecture

5.2. OpenCL API Wrapper Library Design

Before proceeding with design of any OpenCL application, we first need to get familiar with OpenCL related entities and their relations.

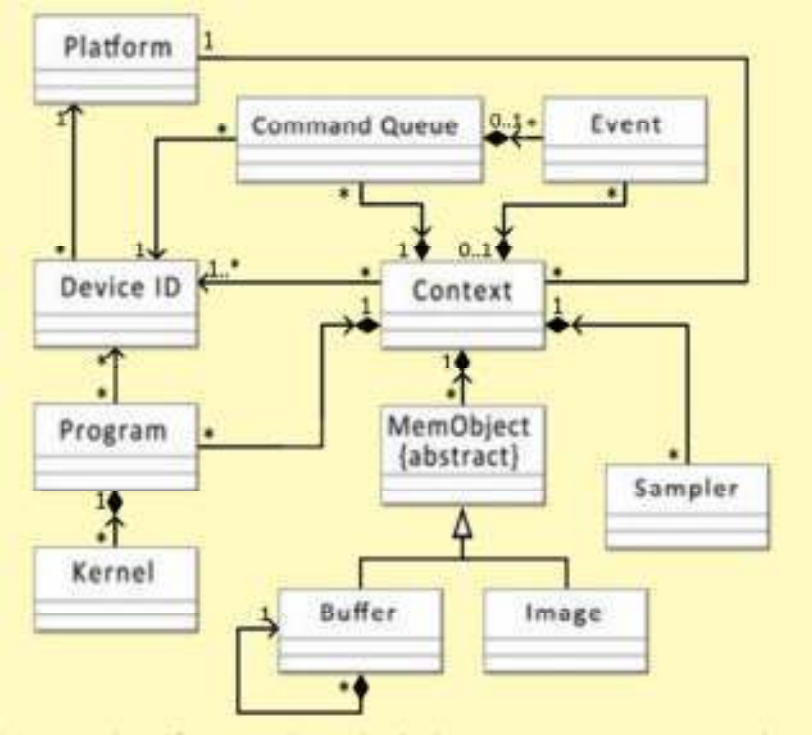

Figure 9 OpenCL Class Diagram, as per OpenCL Quick Reference

The entities in figure above are used in general OpenCL application workflow as follows:

- Selecting a OpenCL platform to work with
- Selecting device to run kernels on, among the devices associated with the platform
- Creating the OpenCL Context object using the selected platform and device
- Using the Context object to create Command Queue – An object that manages execution of device operations, and synchronization objects (Events) if necessary
- Using the Context object to compile a program and create the Program object
- Retrieve particular kernel that the user desires to run from the Program object
- Using context to allocate device memory (Creating Buffer objects)
- Using Command Queue to initiate device operations such as data transfer from host to device and kernel execution

The classes included in OpenCL API wrapper library encapsulate pure OpenCL objects. The wrapper classes simplify the interface with those objectsand manage the underlying resources in RAII style (Resource Acquisition Is Initialization).

Following class diagram illustrates the classes in the wrapper library and their relations:

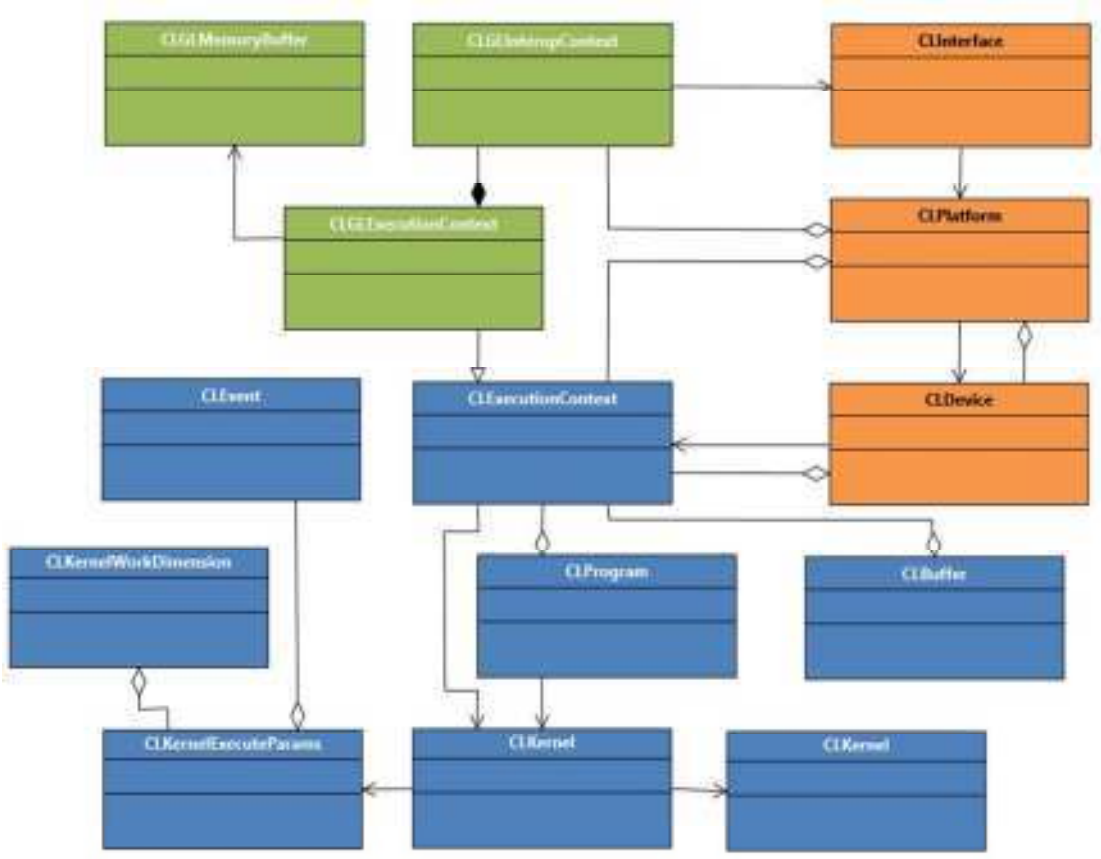

Figure 10 OpenCL API Wrapper Library Classes

The classes shaded in Orange are responsible for obtaining the platform and devices available on the machine, and perform basic initializations of OpenCL.
The classes shaded in Green encapsulate functionality of the OpenCL/OpenGL interop standard extension. Since this functionality is essential for effective rendering of pixel buffer produced by the Ray Tracing algorithm, the API provides this functionality.
The classes shaded in blue are responsible for compilation and execution of OpenCL programs, creating synchronization objects and device memory allocation.

5.3. Algorithms Library

The primary purpose of the Algorithms Library is to provide classes and functionality to construct the Acceleration Structures and traverse them. In addition, the library provides utility classes for algorithms which are the building blocks for Acceleration Structure construction and traversal algorithms. Those building blocks may be used by programmers for implementation of additional Acceleration Structures.
The design provides an abstract interface for Acceleration structures for that purpose, to fulfill the Extensibility requirement.

The implementation of the Algorithm Library classes uses the OpenCL API Wrapper library for GPGPU device related operations.

The classes in the Algorithms Library and their relations are presented in following figure:

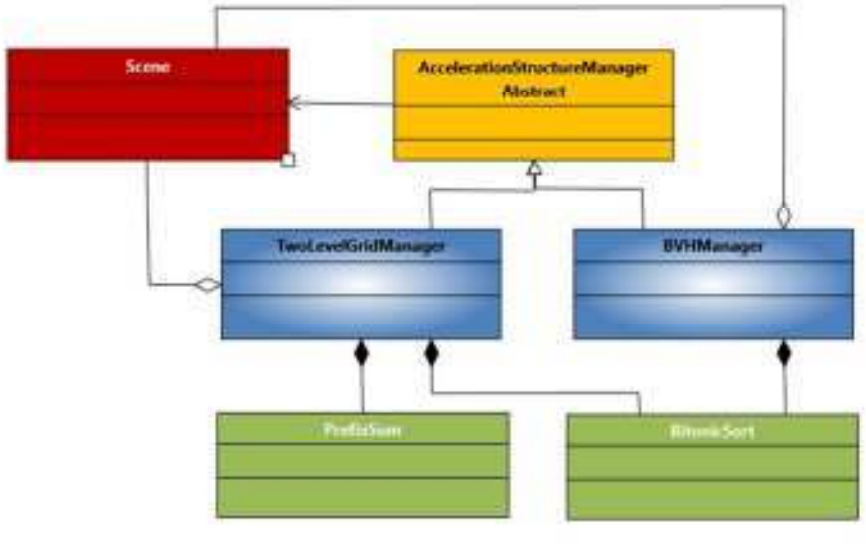

Figure 11 Algorithm Library Classes

The AccelerationStructureManager class is an abstract interface for generic acceleration structure and its related operations. The Algorithm Library contains implementation of two acceleration structures: The Two Level Grid and Bounding Volume Hierarchy, encapsulated in classes TwoLevelGridManager and BVHManager respectively.
The implemented algorithms required some other algorithms that were needed for implementation: The Parallel Prefix Sum and Bitonic Sort. The classes that implement those algorithms (Shaded in Green in the figure) are also part of the Algorithm Library API, for extensibility.

In addition, the library provides a basic data structures commonly used in Ray Tracing: That includes geometric primitives, camera data, ray-object intersection, and more. Since the code of those structures is designed to run on both host and device, the design of those structures is C language style structs.

Another important feature provided by the library is the Scene operations. Those operations are implemented in class Scene, which encapsulates host and device memory that contains the scene, and is responsible for loading the scene from file and manage loading the scene data to device.

6. Acceleration Structures

6.1. Uniform Grid and Two Level Grid

The Uniform Grid is perhaps the simplest acceleration structure. All that is needed to construct it is to divide the scene to cells, and associate each cell with an object that resides inside the cell.

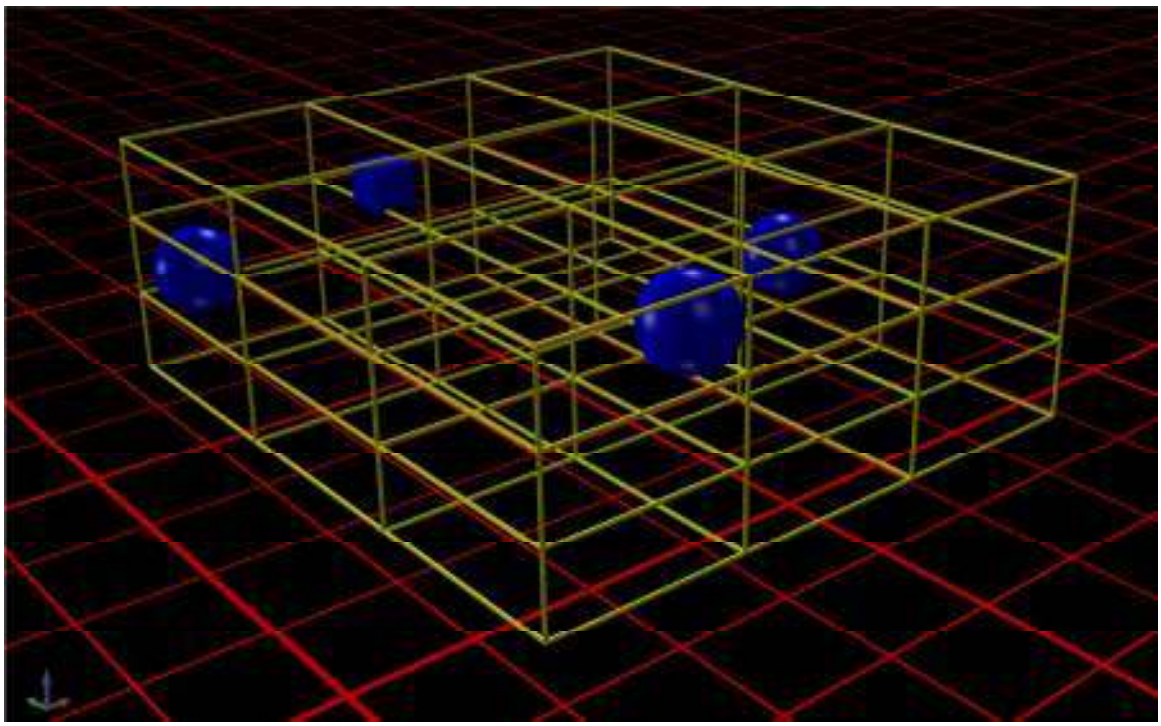

Figure 12Objects in a scene divided to 3x3 Uniform Grid

During the traversal process, we determine the cells that the examined ray passes through. Those objects then become candidates for closer ray-object intersection.

Uniform grids are said to be more efficient for dense scenes, and less efficient for sparse scenes – due to the ineffective space skipping by the traversal algorithm.
To make the traversal more effective, we could divide the grid to sub-grids, with different resolutions depending on amount of primitives in the sub-grid.
This approach was adapted for GPGPU by Kalojanov, Billeter and Slusallek, who proposed a **Two-Level-Grid** algorithm [34] to improve performance of Uniform Grids.
In this section we will analyze the implementation of this algorithm.

6.1.1. Grid Construction

The general steps to construct a simple Uniform Grid are:

- Calculate the Axis Aligned Bounding Box (AABB) of the scene
- Divide the resulting box into a grid with specified resolution
- Associate each primitive in the scene with a grid cells that contain it

In order to store the constructed Uniform Grid we will need:

- An array of references to primitives in the scene – The Reference Array
- A structure for storing the cells, while each cell contains a range of indexes of the Reference Array, that point to primitive contained in the cell.

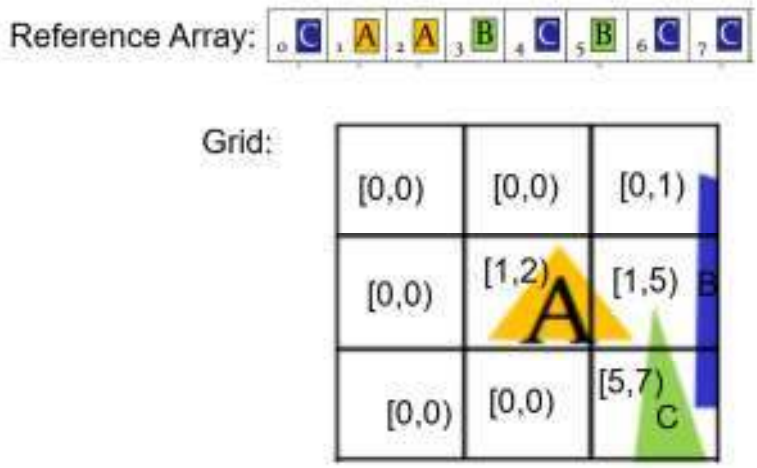

Figure 13Uniform Grid Representation: The Reference Array, and each grid cell contains an interval of indexes in Reference Array, that contains indexes that point to primitives contained by the cell [34]

The typical way to choose resolution of a grid is according to the following calculation:

$$R_x = d_x \sqrt[3]{\frac{\lambda N}{V}}$$

$$R_y = d_y \sqrt[3]{\frac{\lambda N}{V}}$$

$$R_z = d_z \sqrt[3]{\frac{\lambda N}{V}}$$

Where:

d_x, d_y, d_z – Extent of the box along X,Y and Z axes respectively
V – Volume of scene bounding box
N – Number of primitives in the scene
λ – Grid density – A user chosen constant

For each primitive we first determine which cells it overlaps. This operation may be performed in parallel (Each GPU thread performs this operation for each primitive, for example), and the output will result in a set of cell-primitive pairs.

At the next stage, those pairs need to be sorted by cell index.
Once the set is sorted, we may trivially extract the Reference Array and the index ranges for each cell.

To modify this structure as a Two-Level Grid, we divide each grid cell to *leaf cells* and our reference array will store *leaf-primitive* pairs. The top level cells will store the range of indices in the reference array, which are associated with its sub-cells, as in figure below:

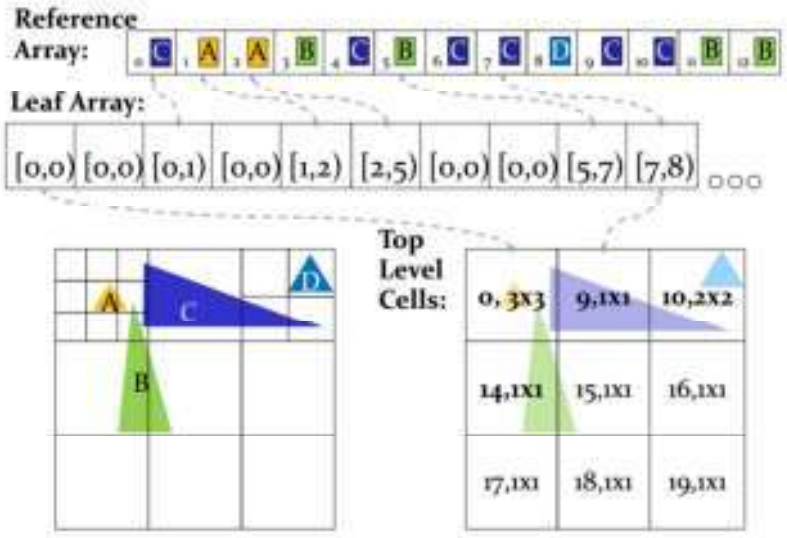

Figure 14Two-Level Grid Representation: Each top level cell contains range of indices of the Reference Array, which associates leaf cells with primitive indices [34]

For example in figure 12, top level cell 0 overlaps with objects A, B and C.
The particular top level cell 0 is split to 3x3=9 leaf cells.
The reference array, as mentioned before, contains references to objects, **sorted by leaf index that overlaps with it.**
Since we examine first top level cell which contains 9 leaf cells in our example, the first 9 cells of leaf array will contain index ranges that indicate indexes at which the value of reference pair is a reference to object that overlaps the top level cell 0.
This way we can track easily which object overlap any given leaf cell:
Since the range in leaf array at index 0 is [0,0), the leaf cell is empty, and so it is with the cell at index 1 as well.
At index 2 however, we see that the range in reference array is [0,1). Therefore, objects that overlap leaf cell at index 2 will be referred in reference array, at index 0. The cell at index 0 in reference array points to object C. Therefore, if ray hits this object, we would test intersection between the ray and object C.

If we take leaf cell at index 5, we can see that the range in leaf array at index 5 is [2,5), which leads us to examine reference array at indices 2,3, and 4.
We further see that the reference array entry at index 2 points to object A, index 3 points to object B and index 4 points to object C. Therefore, for leaf cell at index 5, the candidates for intersection will be objects A, B and C.

Now as we understand how do we store and query the two-level grid, we can look at the construction algorithm and examine it in detail.

```
    b ← COMPUTE BOUNDS()
2:  r ← COMPUTE TOP LEVEL RESOLUTION()
    t ← UPLOAD TRIANGLES()
4:  data ← BUILD UNIFORM GRID(t, b, r)
    A_tlp ← GET SORTED TOP-LEVEL PAIRS(data)
6:  n ← GET NUMBER OF TOP-LEVEL REFERENCES(data)
    tlc ← GET TOP-LEVEL CELL RANGES(data)
8:                          ▷ COMPUTE LEAF CELL LOCATIONS
    G ← (r_y, r_z), B ← r_x
10: A_cc ← ARRAY OF r_x * r_y * r_z + 1 ZEROES
    A_cc ← COUNT LEAF CELLS <G,B>(b, r, tlc)
12: A_cc ← EXCLUSIVE SCAN(A_cc, r_x * r_y * r_z + 1)
      ▷ SET LEAF CELL LOCATIONS AND CELL RESOLUTION
14: tlc ← INITIALIZE <G,B> (A_cc, tlc)
                          ▷ COMPUTE REFERENCE ARRAY SIZE
16: G ← 128, B ← 256
    A_rc ← ARRAY OF G + 1 ZEROES
18: A_rc ← COUNT LEAF REFS<G,B>(t, b, r, n, A_tlp, tlc)
    A_rc ← EXCLUSIVE SCAN(A_rc, G + 1)
20: m ← A_rc[G]
    A_p ← ALLOCATE LEAF PAIRS ARRAY(m)
22:                                 ▷ FILL REFERENCE ARRAY
    A_p ← WRITE PAIRS<G,B>(t, b, r, n, A_tlp, tlc, A_rc)
24: A_p ← SORT(A_p)
    leafs ← EXTRACT CELL RANGES< >(A_p, m)
```

Figure 15 Two Level Grid Construction Algorithm [34]

Lines 1-3 – Initialization

This stage of the algorithm is straightforward – It is the basic preparation of data for processing.

The scene bounding box can be calculated at the stage when the scene to be rendered is loaded to the host memory from file.

Counting of primitives in the scene also can be performed at the loading stage.

This gives us that the grid resolution can be calculated immediately after loading the scene from file, since we have all the needed variables after the loading stage.

Uploading triangles, however needs some more consideration: We could take the straightforward approach of storing 3 vertices for each triangle in the scene, but storing the mesh as indices and vertices will consume significantly less memory.

If we look ahead, we can identify a requirement that we will need to retrieve a triangle by index. To satisfy this requirement we can use an abstract data structure for the scene which will store the triangles as vertices and indices, and provide a function to retrieve a triangle in running order.

In context of the discussed algorithm, uploading triangles will mean uploading the structure that holds the triangles from host memory to device memory.

Line 4 –7 -Build Top-Level Grid

The name of the step is generic, and needs some detailed explanation.

To build a top-level grid we first count how much cells are overlapped by each triangle. We store this result in an array, where the count of overlapped cells per triangle is stored at the index of that triangle in the scene. Then, we run the **Prefix Sum** algorithm on that array. This allows us to determine how many top-level cell-triangle pairs do we have – Their quantity will be at the last cell of the result of Prefix Sum.

Next we allocate an array of a required size that can accommodate all the pairs, and run a kernel that will calculate for each triangle in parallel, which cells does it overlap, and write the pairs into the allocated pairs array, while the key of the pair is the cell index and the value of the pair is the primitive index.
Yet again we use the result of Prefix Sum algorithm to determine the base index in pairs array, where pairs for each triangle should be written – The base index in pair array for triangle at index *i* will be the value in the Prefix Sum output array at the index *i.*
Line 5 in the algorithm is the sorting of pairs by cell index. For sorting we may use variation of **Radix Sort** that supports key-value pair sorting, similar variation of **Bitonic Sort** (Like what was used in this project), or any other parallel sorting algorithm.
After we have the sorted pairs, we calculate the cell ranges (Line 7 in the algorithm) – For each cell*c* we calculate the start index – Index of the first pair in sorted pairs array for cell *c*, and the end index – Index of the first pair in sorted pairs array for cell *c+1*. To calculate the ranges, we load chunks of the pairs array into shared memory, and for each pair at index *p* we check whether the next pair, at index *p+1,* is associated with different cell than this pair. If so, we write the index *p* as closing index for the cell associated with pair *p*, and as opening index for the cell associated with pair *p+1*.

At this point, we have constructed a single level grid. We have the following data available:

- Top Level Reference Array – Array of top level cell-primitive pairs, sorted by cell index
- Top Level Ranges Array – Array that contains ranges of indices in the Reference Array, for each cell: For each cell at index *c,* a value in range array at index *c* contains index range in Reference Array that is associated with the cell *c* – Just as shown in figure 13.

Line 8 –22 –Prepare data and memory for leaf cells
The data that we've calculated so far gives us the possibility to calculate resolution for each cell – that is, how many leaf cells each top level cell should contain (Line 11).
This can be simply done in parallel – For each top level cells we check how many primitives does it contain (*range.high – range.low*) and calculate the top level cell resolution using the same principle as we did when calculated the top level resolution.
The kernel may update the top-level cells, and update a counter array that contains counts of leaves for each top level cell.
Calculating Prefix Sum (Or Exclusive Scan, as per algorithm on line 12) on the array of counters will give us the total number of leaf cells in the grid, and the index of the first leaf for each cell.
Then, we count the amount of leaf-primitive pairs, using the same principle as when we build the top-level grid. This time however, we need more complex method to determine whether a primitive overlaps a leaf cell. To save time we can test whether the bounding box of a primitive overlaps the leaf cell. If it does we increase the leaf-primitive pair count for the checked top-level cell. We run Prefix Sum on the resulting array to determine the amount of leaf-primitive pairs, so we can allocate memory that can accommodate the pairs.

Line 22 – 25 – Finalizing the construction of Two-Level Grid
Now when we have allocated memory to accommodate the reference array, we can generate the leaf-primitive pairs. At this stage we will need to use more precise and more complex test, so that our pairs will be accurate. In this project, the method that was used relied on **Separating Axis Theorem**. Unlike what we did in pairs counting stage, we use Box-Triangle intersection test to determine whether a leaf cell intersects a primitive. We perform this test for each primitive, against each leaf in top-level cell.
The resulting pairs are written into the Reference Array.
At the next stage, the leaf-primitive pairs are being sorted, similarly to the analogous stage in building the Top Level Grid (Line 24).
Finally, the leaf cell ranges are extracted from the sorted leaf pairs array.

This process is roughly illustrated in the following figure:

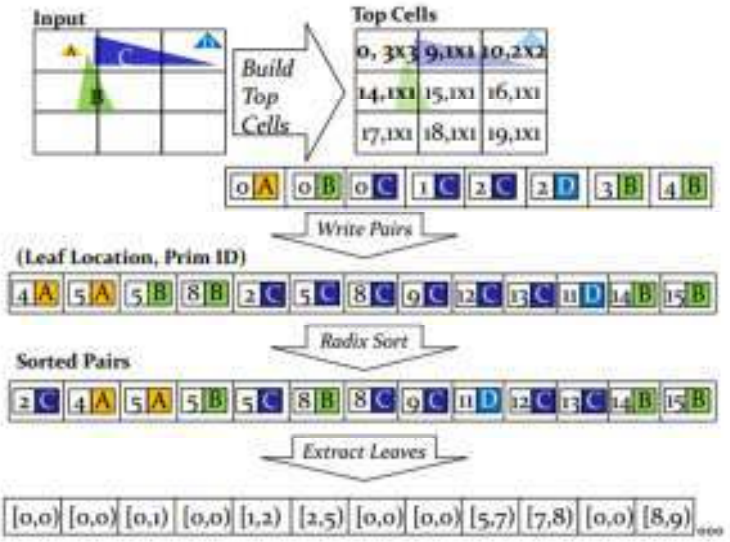

Figure 16 The process of construction of Two Level Grid [34]

6.1.2. Grid Traversal

Grid traversal is about determining which cells does the ray pierce, and test intersections between the ray and primitives that contained in those cells. The figure below illustrates this in 2D:

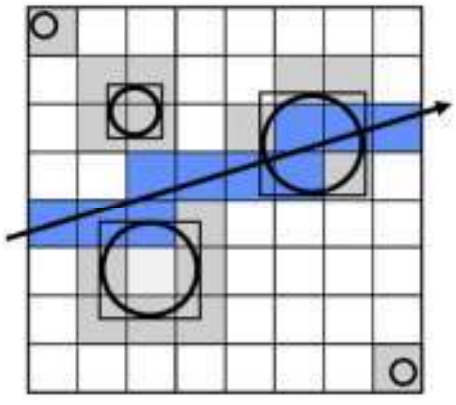

Figure 17 2D illustration of grid traversal [3]

For traversal, we will need to compute:

- The initial value of parameter *t* of the ray – At which ray enters the bounding box of the scene. This value will be the initial value of *tCurrent* – A variable that will be used to track the current *t* at the current traversal position
- Next ray intersections with horizontal and vertical edges, in terms of ray parameter *t*. We will store those values as*nextX*and *nextY*.
- The difference between each vertical and horizontal edge to the next one along the ray. We will store those values as *deltaX*abd*deltaY*.

In addition, we keep track of the current cell that is visited, by storing the current coordinates of the cells: X and Y, and the direction of increment of the ray: *dX*and *dY*
The following figure illustrates the meaning of the data:

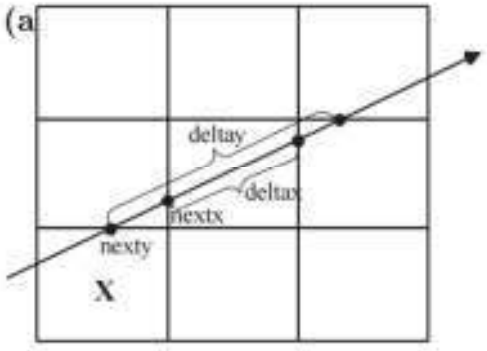

Figure 18 Data for grid traversal – A 2D illustration [2]

During traversal, we determine the closest intersection with an edge. We do this by finding *min(nextX,nextY)*. If the minimal value is *nextX*, the closest edge is vertical, and the traversal will advance along the X axis. Otherwise, the closest edge is horizontal, and the traversal will advance in the direction of Y axis. For example,if this edge happens to be vertical, we advance *nextX*by *deltaX* and advance the X cell coordinate in direction of *dX*. If the closest edge is horizontal, we do the same with variables related to Y axis. Then, when we have the current cell coordinates we check the primitives in that cell for intersection with the ray.

All that is needed for expanding the algorithm to 3D is to add calculations for *nextZ, deltaZ* and cell index along Z similarly to how it was done for X and Y axes.

To expand the algorithm for Two Level Grid traversal algorithm, we perform traversal of inner grid at each traversal step of the top level grid, using the same principle.

```
1.  struct Contact generate_contact(conststruct Ray* ray, const char* scene, structGridData* gridData,
2.                          structTopLevelCell* topLevelCells,uint2* leavesArray,uint2* pairsRefArray)
3.  {
4.  //Calculating the entry and exit t values for each axis
5.  float tx_min, ty_min, tz_min;
6.  float tx_max, ty_max, tz_max;
7.
8.  float a = 1.0f / ray->direction.x;
```

```
9.  tx_min = ((a >= 0.0f ? gridData->box.bounds[0].x : gridData->box.bounds[1].x) - ray->origin.x) * a;
10. tx_max = ((a >= 0.0f ? gridData->box.bounds[1].x : gridData->box.bounds[0].x) - ray->origin.x) * a;
11.
12. a = 1.0f / ray->direction.y;
13. ty_min = ((a >= 0.0f ? gridData->box.bounds[0].y : gridData->box.bounds[1].y) - ray->origin.y) * a;
14. ty_max = ((a >= 0.0f ? gridData->box.bounds[1].y : gridData->box.bounds[0].y) - ray->origin.y) * a;
15.
16. a = 1.0 / ray->direction.z;
17. tz_min = ((a >= 0.0f ? gridData->box.bounds[0].z : gridData->box.bounds[1].z) - ray->origin.z) * a;
18. tz_max = ((a >= 0.0f ? gridData->box.bounds[1].z : gridData->box.bounds[0].z) - ray->origin.z) * a;
19.
20. //In case the ray doesn't hit the scene bounding box at all – Return invalid contact
21. float t0 = min(tx_min,min(ty_min,tz_min));
22. if (t0 > max(tx_max,max(ty_max,tz_max)))
23.         return NO_CONTACT;
24.
25. //Axis denoting constants
26. constintiX =0,iY =1,iZ = 2;
27.
28. //Calculating the coordinates of the cell at which traversal begins
29. intidx[3];
30.
31. if (isPointInside(&(gridData->box),ray->origin))
32. {
33.         //Case when ray starts inside the grid
34.         idx[iX] = clamp((ray->origin.x - gridData->box.bounds[0].x) * gridData->resX
35. / (gridData->box.bounds[1].x - gridData->box.bounds[0].x), 0.0f, gridData->resX - 1);
36.         idx[iY] = clamp((ray->origin.y - gridData->box.bounds[0].y) * gridData->resY
37. / (gridData->box.bounds[1].y - gridData->box.bounds[0].y), 0.0f, (gridData->resY - 1);
38.         idx[iZ] = clamp((ray->origin.z - gridData->box.bounds[0].z) * gridData->resZ
39. / (gridData->box.bounds[1].z - gridData->box.bounds[0].z), 0.0f, gridData->resZ - 1);
40. }
41. else
42. {
43.         CL_FLOAT3 p = ray->origin + (ray->direction * t0);  // initial hit point with grid's bounding box
44.         idx[iX] = clamp((p.x - gridData->box.bounds[0].x) * gridData->resX
45.  / (gridData->box.bounds[1].x - gridData->box.bounds[0].x), 0.0f, gridData->resX - 1);
46.         idx[iY] = clamp((p.y - gridData->box.bounds[0].y) * gridData->resY
47.  / (gridData->box.bounds[1].y - gridData->box.bounds[0].y), 0.0f, gridData->resY - 1);
48.         idx[iZ] = clamp((p.z - gridData->box.bounds[0].z) * gridData->resZ
49.  / (gridData->box.bounds[1].z - gridData->box.bounds[0].z), 0.0f, gridData->resZ - 1);
50. }
51.
52. // Ray parameter increments per cell in the x, y, and z directions
53. float delta[3];
54. delta[iX] = (tx_max - tx_min) / gridData->resX;
55. delta[iY] = (ty_max - ty_min) / gridData->resY;
56. delta[iZ] = (tz_max - tz_min) / gridData->resZ;
57.
58. float     next[3];
59. int       step[3];
60. int       stop[3];
61.
62. if (ray->direction.x> 0.0f)
63. {
64.         next[iX] = tx_min + (idx[iX] + 1) * delta[iX];
```

```
65.             step[iX] = +1;
66.             stop[iX] = gridData->resX;
67.  }
68.  else
69.  {
70.             next[iX] = ray->direction.x == 0.0f ? FLT_MAX : tx_min + (gridData->resX - idx[iX]) * delta[iX];
71.             step[iX] = -1;
72.             stop[iX] = -1;
73.  }
74.
75.  if (ray->direction.y> 0)
76.  {
77.             next[iY] = ty_min + (idx[iY] + 1) * delta[iY];
78.             step[iY] = +1;
79.             stop[iY] = gridData->resY;
80.  }
81.  else
82.  {
83.             next[iY] = ray->direction.y == 0.0f ? FLT_MAX : ty_min + (gridData->resY - idx[iY]) * delta[iY];
84.             step[iY] = -1;
85.             stop[iY] = -1;
86.  }
87.
88.  if (ray->direction.z> 0)
89.  {
90.             next[iZ] = tz_min + (idx[iZ] + 1) * delta[iZ];
91.             step[iZ] = +1;
92.             stop[iZ] = gridData->resZ;
93.  }
94.  else
95.  {
96.             next[iZ] = ray->direction.z == 0.0f? FLT_MAX : tz_min + (gridData->resZ - idx[iZ]) * delta[iZ];
97.             step[iZ] = -1;
98.             stop[iZ] = -1;
99.  }
100.
101.// Top Level Grid Traversal Loop
102.while (true)
103.{
104.            structTopLevelCell cell = topLevelCells[getCellIndex(idx[iX],idx[iY],idx[iZ],
105.                                    gridData->resX,gridData->resY,gridData->resZ)];
106.
107.            float minimal = min(next[iX],min(next[iY],next[iZ]));
108.            int axis = 0;
109.            while(minimal != next[axis])
110.            axis++;
111.
112.            //Only in case when top level cell has leaf cells, we process the top level cell further
113.            if (!(cell.resX == 0 || cell.resY == 0 || cell.resZ == 0))
114.            {
115.                        //Process the cell
116.                        struct AABB cellBox;
117.                        cellBox.bounds[0].x = gridData->box.bounds[0].x + idx[iX] * gridData->stepX;
118.                        cellBox.bounds[0].y = gridData->box.bounds[0].y + idx[iY] * gridData->stepY;
119.                        cellBox.bounds[0].z = gridData->box.bounds[0].z + idx[iZ] * gridData->stepZ;
120.                        cellBox.bounds[1].x = cellBox.bounds[0].x + gridData->stepX;
```

```
121.                    cellBox.bounds[1].y = cellBox.bounds[0].y +gridData->stepY;
122.                    cellBox.bounds[1].z = cellBox.bounds[0].z +gridData->stepZ;
123.                    result = processTopLevelCell(ray,scene,&cell,&cellBox,leavesArray,pairsRefArray);
124.                    if (result.contactDist> 0.0f)
125.                            return result;
126.            }
127.
128.//Advancing traversal in the direction of the selected axis
129.            next[axis] += delta[axis];
130.            idx[axis] += step[axis];
131.
132.            if (idx[axis] == stop[axis])
133.                    return NO_CONTACT;
134.        }
135.}
```

Figure 19 Top Level Grid Traversal

In lines 4-23 the traversal method determines intersection points with planes formed by each face of bounding box of the scene. This method is actually a slab based box-ray intersection. The figure below illustrates this: For instance for x axis, t0 and t1 will be stored as *tx_min*and*tx_max* respectively.

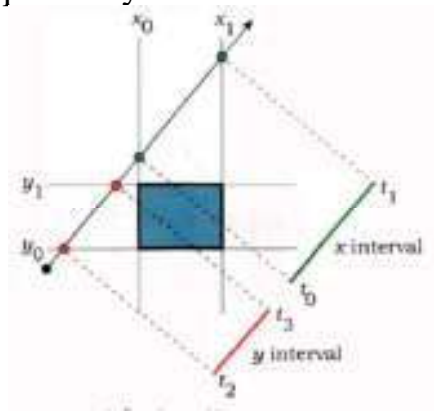

Figure 20 Illustration of intersectiong ray with lanes formed by the faces of AABB [3]

We use results of this calculation to determine whether the ray hits the scene bounding box at all (If it doesn't, we can just report that there is no contact), and for calculating the increment of ray parameter *t* along each axis – The deltas.

For storing data per-axis, such as *next* and *delta,* we use arrays of size 3, and associate each axis with an index. The constants, iX,iY, and iZ are defined in line 26.
This representation is very convenient for reducing control flow divergence in the traversal loop.

In lines 28-50 we calculate the index of the cell at which the traversal is started.
Two cases must be considered here:

- Ray origin is outside the box – Then traversal starts at the cell that contains the point where ray hits the scene bounding box
- Ray origin is inside the box – Then traversal starts at the cell that contains the ray origin.

OpenCL *clamp* function is used to clamp the result of the calculation to valid range of cell indices along each axis.

In lines 52 – 56 we calculate the deltas: We use the intermediate results of Ray-Box intersection to determine the delta along each axis. Considering figure 20, in order to calculate delta for X axis, we take the interval length – (t1 – t0) (*tx_max – tx_min* according to our code) and divide it by the number of grid cells along the X axis.

In lines 62-99we determine the next traversal point along each axis, the increment of indices of cells, and the boundary at which traversal is stopped. We calculate the values according to the direction of the ray along the axis. In case direction of the ray along an axis is 0, we assign infinitely large value to the corresponding *next* value, so it will never be chosen as a direction of traversal.

The main traversal loop (lines 102-134)is where the actual work happens – On line 107 we choose the minimal value for next intersection, and use it to choose the axis along which traversal should be advanced (Lines 108-110). We can see now how representing data per axis in arrays and denoting axes with indices helped reducing control flow divergence – Once we determine the index of the minimal *next* value, we use it to update the values for the appropriate axis.
We examine a cell only if it contains objects (And therefore, has leaf cells). If it doesn't – The traversal simply advances along the selected axis (Lines 129-130). If current index reached the *stop* value after the increment, *NO_CONTACT* is reported.

If the cell is not empty, we first compute the bounding box of the cell (Lines 116-122) and call the cell processing function.

The cell processing function is very similar to the top level traversal function described in figure 19 – Since it traverses a single level Uniform Grid. The only difference is in action that performed inside the traversal loop in case the cell is not empty:

```
1.  struct Contact result;
2.  result.normalAndintersectionDistance.x = 0.0f;
3.  result.normalAndintersectionDistance.y = 0.0f;
4.  result.normalAndintersectionDistance.z = 0.0f;
5.  result.normalAndintersectionDistance.w = FLT_MAX;
6.  bool contactFound = false;
7.
8.  while (true)
9.  {
10.         float minimal = min(next[iX],min(next[iY],next[iZ]));
11.         int axis = 0;
12.         while(minimal != next[axis])
13.                 axis++;
14.
15.         //Process the leaf
16.
17.         intleafIndex = getCellIndex(idx[iX],idx[iY],idx[iZ],
18.                 topLevelCell->resX,topLevelCell->resY,topLevelCell->resZ) + topLevelCell->firstLeafIdx;
```

```
19.         uint2leafRange = leavesArray[leafIndex];
20.         for (;leafRange.x<leafRange.y; leafRange.x++)
21.         {
22.                 uint2refPair = pairsRefArray[leafRange.x];
23.                 struct Triangle triangle = getTriangleByIndex(scene,refPair.y);
24.                 struct ContactnewContact = triangleIntersect(triangle, ray->origin,ray->direction);
25.                 if (newContact.contactDist> 0 &&newContact.contactDist<result.contactDist)
26.                 {
27.                         contactFound = true;
28.                         result = newContact;
29.                         result.materialIndex = MESH_HEADER(submesh)->materialIndex;
30.                 }
31.         }
32.
33.         if (contactFound)
34.                 return result;
35.
36.         next[axis] += dt[axis];
37.         idx[axis] += step[axis];
38.
39.         if (idx[axis] == stop[axis])
40.                 return NO_CONTACT;
41. }
```

Figure 21 Top Level Cell Traversal Loop

We retrieve the range from leaf array. For each index in range, we retrieve the primitive index from reference array, and then retrieve the triangle itself from the scene.
Once we have the triangle, we can intersect it with the ray.
If intersection is found, and the intersection distance is the minimal so far (Since for ray tracing we need the **closest intersection**) we save it as potential result.
Eventually, the function will return the closest contact found, since the structure of Two Level Grid guarantees that the first contact we found in a top level cell is the closest one to the ray origin.

6.2. Bounding Volume Hierarchy (BVH)

Bounding Volume Hierarchy is a hierarchical grouping of 3D objects, where each group is associated with a bounding volume that contains the objects within the group.
The objects are usually contained in leaf nodes, while each internal node represents a group of objects. Each internal node and the root are associated with a bounding volume (In this case it is an Axis Aligned Bounding Box, but can be any geometric primitive in general) that fully contains the objects within a group that the node represents.
BVH tree is a binary tree, therefore each node has two nodes or less.

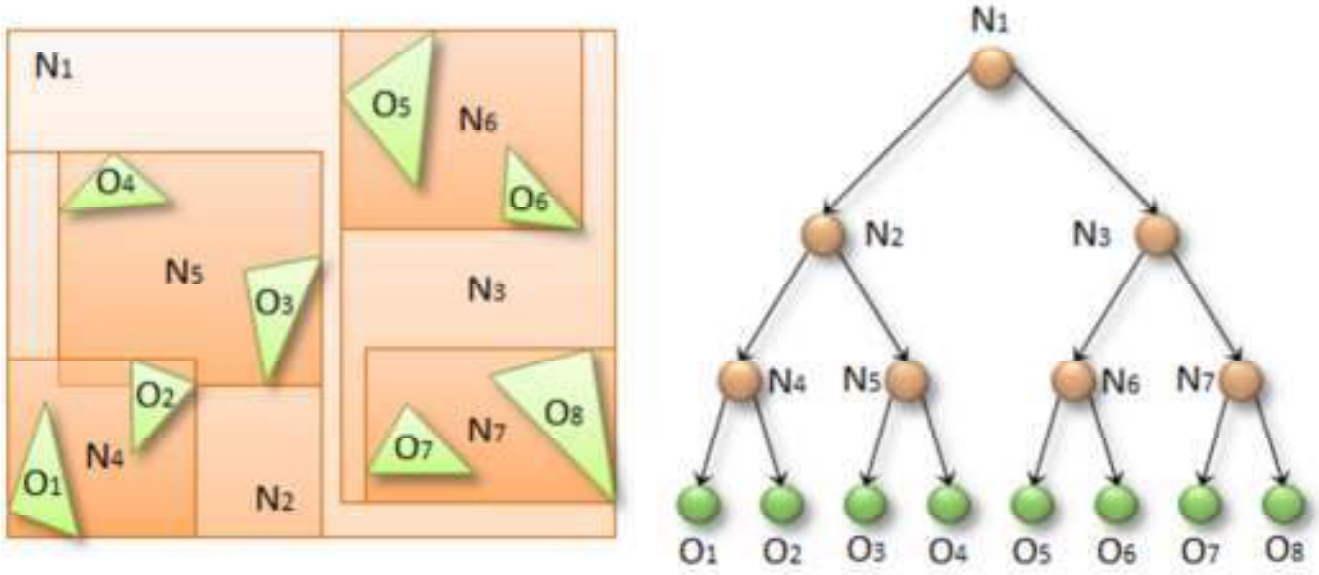

Figure 22Relation of arrangement of object in a scene to BVH tree [36]

As shown in the example in above, the leaf nodes – for instance O1 and O2 contain objects. Their parent internal node, N4 is associated with a bounding volume that contains its children. Moving one level up bring us to N2, which in turn is associated to bounding volume that contains the leaf nodes: O1, O2, O3, and O4. Finally, the root node N1 is associated to a bounding volume that contains the entire scene.

In context of Ray Tracing, we check the tested ray for intersection with bounding volumes of each child of the root – If there is an intersection with only one of the children, the traversal continues in the respective direction. In case ray pierces both of the children, the traversal continues in an arbitrary direction, while the other child is pushed into a stack, and will be revisited after the other branch was examined.

6.2.1. BVH Construction

Among the BVH construction algorithm, I've chosen a method developed byTero Karas of NVIDIA Research. The method relies on sorting objects in some order and use the sorting for effective choosing of close objects to build the hierarchy. This approach of ordering nodes is called **LBVH** – Linear BVH. The particular method by Tero Karas also maximizes the utilization of parallel hardware [35].
Since maximizing parallelism is a key factor for our subject, this is the method that was chosen for implementation.

According to the method, one of the candidates for order by which we sort the object is called **Z - Order Curve**, which we will use here. The Z-Order curve is a function that maps multidimensional data (Three dimensional in this case) to one dimension, while preserving locality. The property of preserving locality is useful for the purpose of building BVH – We can use Z-Order to map 3D objects to array of one-dimensional data, sort it, and obtain sorted objects while the locality is preserved – The neighbor objects in the sorted array will be close to each other in 3D space as well.

Using this method therefore promises that objects located close in 3D space are also close in the hierarchy – A property that contributes to reducing traversal times.

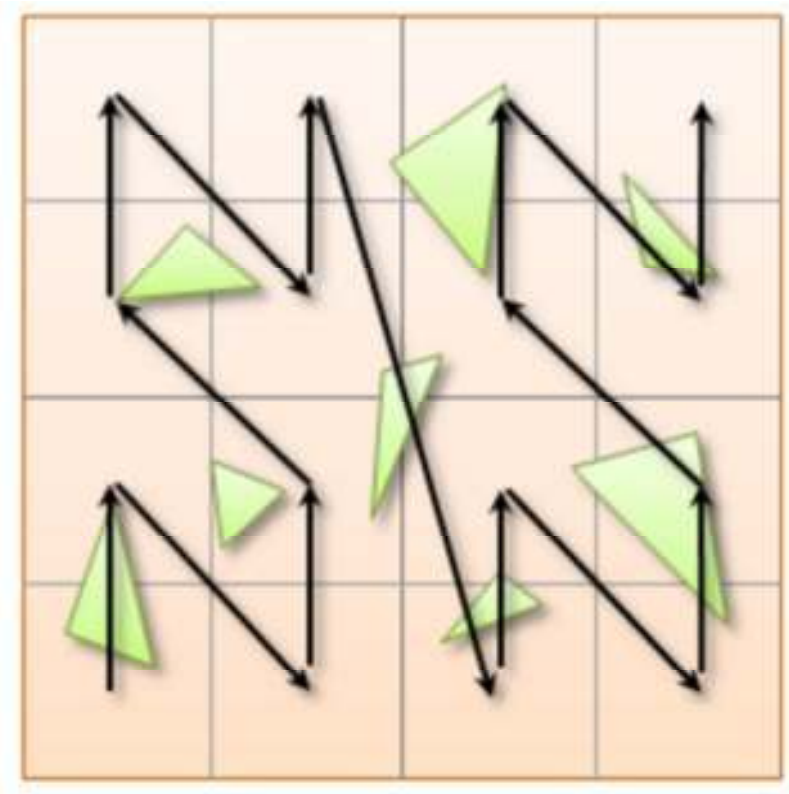

Figure 23 Z-Order Curve in 2D space [36]

The Z-order curve is defined in terms of Morton codes.
The function that calculates Morton code for a point in 3D space is actually the mapping function that maps 3D point to single dimension integer number.
In order to calculate a Morton code of a point, we:

- Normalize the point coordinates relatively to the scene bounding box, so the coordinate values will be between 0 and 1.
- Take a fractional part of the fixed-point binary representation of each coordinates and insert two gaps after each bit
- Interleave the bits of all coordinates, to form a single binary integer.

This calculation is illustrated in the figure below:

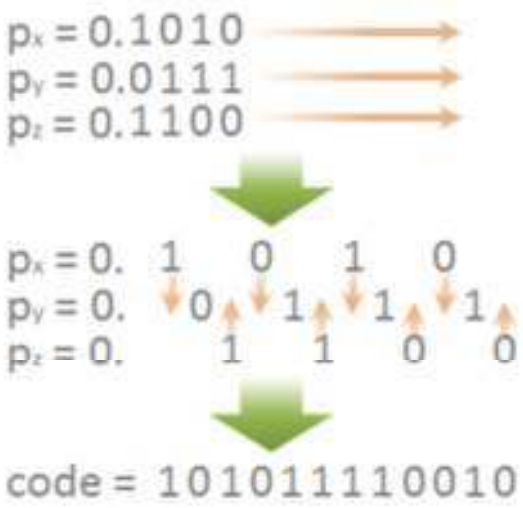

Figure 24 Morton Code Calculation [36]

Since we deal with 3D objects and not just with points, we can calculate the centroid of the bounding box of the object, and calculate its coordinates relative to bounding box of the scene.

After we have the sorted objects, we can think about each internal node as a linear range within the sorted array. The root contains the entire array, [0,N-1], while its children contain the range [0,*split*] and [*split*+1,N-1] respectively. This process could be continued in a top-down manner, while at each level we find the appropriate *split* value.

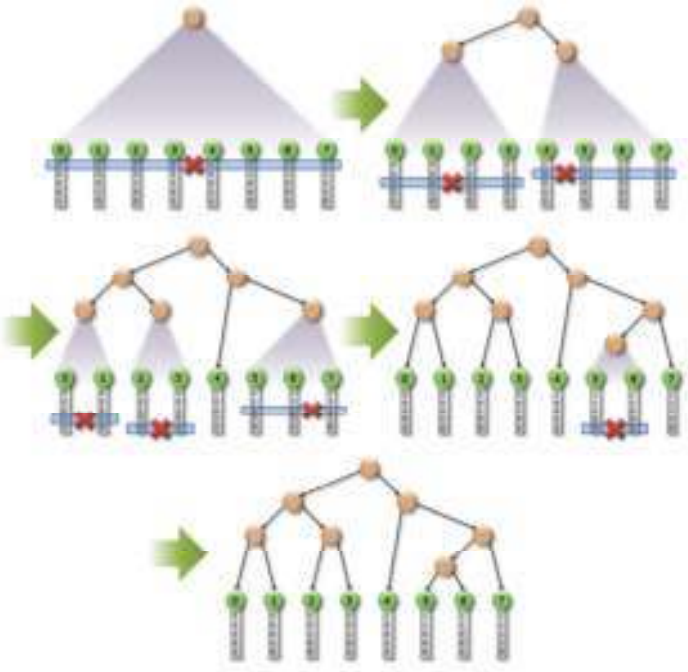

Figure 25 Internal nodes of a tree as linear range in a sorted array [36]

6.2.1.1. Paralellizing Tree Construction – Top Level Algorithm

When we try to parallelize the method described above, it becomes apparent that the dependence of each node on his parent is an obstacle in a way of effective parallelization. The method discussed here solves this problem, by numbering the internal nodes in a specific way that lets us process them in parallel without need in any information regarding the rest of the tree. The key issue here is to **determine the linear range of the array of sorted Morton codes, which is associated to an inner node without knowing any information about the rest of the tree**.
TeroKarras' method [35] suggests storing leaf nodes and internal nodes in two arrays, I and L respectively. The root of the tree would be located at I[0]. The children of the root, as well as children of any internal node will be located **according to the index of the value that splits the range associated with the internal node.**
To understand this layout better, consider the following figure:

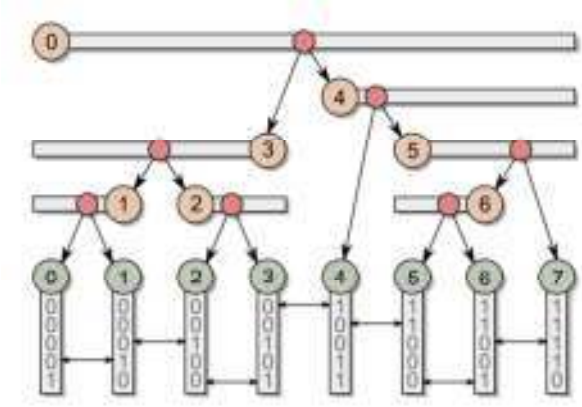

Figure 26Layout of BVH nodes in leaf and internal node arrays [36]

As illustrated in the figure above, the root is located at I[0]. In this case, the split value is found at index 3 in the array of sorted Morton codes. Therefore, the left child will be located at I[3], and the right child at I[4]. In general, for each parent node N with associated range [*low,hi*] we findthe *split* value and assign *split* as N.childA and *split+1*as N.childB. This process continues until leaves reached.

In order to construct the tree in parallel, each thread will have to obtain the following information independently:

- The range associated with an internal node at given index
- The split value of a given range

Before diving into the logic of how this information is obtained, we can examine the top level algorithm.
The algorithm presented below is a slightly modified version of the original algorithm [35]. In my version, the nodes stored in a single array of size 2N – 1 (N leaves + (N – 1) internal nodes). Leaf nodes are stored at indices [0,N-1] while the internal nodes are stored at the indices [N,2N – 1]. The root node will then be located at index N.
This approach simplifies the tree construction and reduces branching in tree traversal.

The high-level pseudocode of the algorithm **that runs at each thread** is presented below:

```
1.  void constructNode(structBVHNode* nodes,
2.  uint* mcToLeaves,
3.  uintnumLeaves,
4.  intidx)
5.  {
6.          //Range of this node
7.          uint2 range = determineRange(mcToLeaves,idx,numLeaves);
8.
9.  //Split of the range
10.         int split = findSplit(mcToLeaves, range.x, range.y);
11.
12. uintinternalNodeIndex = idx+numLeaves;
13. uint first = min(range.x,range.y);
14.         uint last = max(range.x,range.y);
15.         bool aIsLeaf = first == split;
16.         bool bIsLeaf = (last == (split+1));
17.         //split == first = 1? then leaf, otherwise inner node
18.         uintchild_A = mcToLeaves[split].y * aIsLeaf + (split + numLeaves) * !aIsLeaf;
19.         //split+1 == last = 1? then leaf, otherwise inner node
20.         uintchild_B = mcToLeaves[split+1].y * bIsLeaf + (split+1 + numLeaves) * !bIsLeaf;
21.
22.         //Some initializations for traversal and BB calculation that will come afterwards
23.         visitCounter(nodes[internalNodeIndex]) = 0;
24.         fillVector3(nodes[internalNodeIndex].boundingBox.bounds[0],FLT_MAX,FLT_MAX,FLT_MAX);
25.         fillVector3(nodes[internalNodeIndex].boundingBox.bounds[1],FLT_MIN,FLT_MIN,FLT_MIN);
26.         type(nodes[internalNodeIndex]) = INNER_NODE;
27.
28.         //Parent and Children Relations
29.         childA(nodes[internalNodeIndex]) = child_A;
30.         childB(nodes[internalNodeIndex]) = child_B;
31.         parent(nodes[child_A]) = internalNodeIndex;
32.         parent(nodes[child_B]) = internalNodeIndex;
33.
34. }
```

Figure 27 Top Level BVH Construction Algorithm

The algorithm takes the following data as input:

- Array of tree nodes
- Array of key-value pairs while the key is the Morton code, and the value is the index *i*of a primitive associated with a leaf which is also found at index *i*in the array of tree nodes.
- Number of leaves in the nodes array - N
- Index of the current internal node, in range of [0,N-1]

At line 7, we call the *determineRange* method that determines the range associated with this internal node, and at line 10 we determine the split index for that range. This index will be used to determine the children of this internal node.

In case a child is an internal node, the internal node index should be translated by adding *numLeaves* to it, to obtain the actual index of the child in nodes array.
Similarly, the index of this node is translated in line 12 to obtain the actual index where this node resides in nodes array.

At lines 13-20 we determine whether the children of this internal node are leaves:
If a value at split index equals to value at minimum or maximum indices of the range, it indicates that the respective child is a leaf (Following the layout in figure 26 shows why).
In lines 18 and 20, the value of the children is determined by calculation instead of *if* statement, to eliminate branching.

Lines 22-32 writes the data for this internal node: Fills the indices of the children, records child-parent relation between this node and children, initializes the bounding box of the node to default, and the visit counter that will be needed for bounding box calculation (Which will be performed later).

6.2.1.2. Determining the split value of a range

The split of a range of sorted Morton codes is determined according to **the highest bit that differs between Morton codes in a given range**. In order to determine that, we willuse a utility function that calculates the length of the **common prefixlength** of two Morton codes that is defined as follows:

$$\delta = clz(mc1\ XOR\ mc2)$$

While δ denotes the length of the common prefix of Morton codes *mc1* and *mc2*, and *clz*stands for *count lead zeroes*, - A function that is defined in OpenCL standard.

Then, we will use Binary Search to find the **highest Morton Code that has greater common prefix length (δ) than the common prefix of the values at boundaries of the range**.
For instance, in figure 26, we have an array of 8 sorted Morton codes. We will denote the array as *a.* The common prefix length of *a*[0] and *a*[7] is 0. The highest Morton code that has a longer common prefix with *a*[0] is *a*[3]: The MSB of all of the Morton codes in range [0,3] is 0, and hence the common prefix of those values is 1, which is greater than the common prefix of the values in range [0,7], which is, as mentioned before, equals to 0. Hence, the chosen index for splitting the range will be 3.

The partitioning method above works well because it classifies the points on either side of a plane in 3D. The splitting plane can be derived from the value at split index: To determine the axis perpendicular to the plane we calculate:

$$axis = \delta(rangeLow, rangeHi)\ mod\ 3$$

While possible results 0, 1, 2 denote axis X, Y, and Z respectively.

The position along the calculated axis is the coordinate for that axis of the point that was used to calculate the given Morton Code. To find it, we decode the Morton Code back to coordinates of a point, and take the coordinate value that relates to the axis.
The following pseudocode presents the *findSplit* function:

```
1.  intfindSplit( uint* sortedMortonCodes, int first, int last)
2.  {
3.      unsigned intfirstCode = sortedMortonCodes[first];
4.      unsigned intlastCode = sortedMortonCodes[last];
5.
6.      // Calculate the number of highest bits that are the same
7.      // for all objects, using the count-leading-zeros intrinsic.
8.
9.  intcommonPrefix = clz(firstCode ^ lastCode);
10.
11. // Use binary search to find where the next bit differs.
12.     // Specifically, we are looking for the highest object that
13.     // shares more than commonPrefix bits with the first one.
14.
15. int split = first; // initial guess
16. int step = last - first;
17.
18.     do
19.     {
20.         step = (step + 1) >> 1; // exponential decrease
21. intnewSplit = split + step; // proposed new position
22.
23.         if (newSplit< last)
24.         {
25.             unsigned intsplitCode = sortedMortonCodes[newSplit];
26. intsplitPrefix = clz(firstCode ^ splitCode);
27.             if (splitPrefix>commonPrefix)
28.                 split = newSplit; // accept proposal
29.         }
30.     }
31.     while (step > 1);
32.
33.     return split;
34. }
```

Figure 28 findSplit routine pseudocode

On line 9 we determine the length of common prefix of the boundaries of the range. We initialize variables for Binary Search on lines 15 and 16 so the range will be first partitioned in the middle, and perform the Binary Search to find the split: If the common prefix of the range [*first,split*] is longer than the common prefix of the range we accept the split proposal.

The function above ignores the case of duplicate Morton codes – which, however, can occur in practice. There are several workarounds that can be applied to handle this case, which will be discussed later.

6.2.1.3. Determining the range associated with internal node

After understanding the logic of splitting a range of Morton codes, now we can proceed to determining specific range of codes per specified internal node.

The input of the range calculation routine will be:

- Index of the internal node we are calculating for – Denoted as *i*,
- Array of sorted Morton codes – Denoted as *codes*
- Size of the array – Denoted as *size*

For any internal node with the index *i*, we must first determine the "direction" of the range related to the node *codes*[i].
If $\delta(codes[i], codes[i+1]) > \delta(codes[i], codes[i-1])$ it means that the range **starts**at index *i,* and continues rightwards.
Otherwise, the range **ends** at index *i*, and starts somewhere earlier at the left.

We determine the direction as follows:

$$d = sign(\delta(codes[i], codes[i+1]) - \delta(codes[i], codes[i-1]))$$

A result, which can be either 1 or -1, indicates a direction in which we look for, in relation to the index *i.*
The index *i*is actually one of the ends of the range: If *d* = -1, it will be the high end of the range, otherwise it will be the low end of the range.

To find the other end of the range, we will look for the last value that shares the same $\delta$with $codes[i]$, in the direction specified by *d*. We use Binary Search to do that, similarly to the way we used it to find the split value.

The following pseudocode presents the *determineRange* function:

```
1.  uint2 determineRange(uint2* list, intindex,uint size)
2.  {
3.  intdir,d_min,lastIndex = size-1;
4.   uint2 result;
5.   //In case root node encountered - Just return the entire range
6.   if(index == 0)
7.   {
8.            result.x = 0;
9.            result.y = lastIndex;
10.             return result;
11.  }
12.  //Calculating the direction
13. uintminone = list[index-1].x, precis = list[index].x, pluone = list[index+1].x;
14. uintlr[2] = {clz(precis ^ minone),clz(precis ^ pluone)};
15. dir = sign(lr.y - lr.x);
16.
17. //Look for an range to search in (power of two)
18. d_min = lr[dir< 0];
19. intl_max = 2;
20. inttestindex = index + l_max * dir;
21.  while(testindex<= lastIndex&&testindex>=0 &&clz(precis ^ list[testindex].x) >d_min)
22.  {
23.           l_max = l_max<< 1;
24.           testindex = index + l_max * dir;
25.  }
26.
27. //Go from l_max/2 ... l_max/4 ... l_max/8 ..... 1 all the way down
28. int l = 0;
29. for(int div = 2 ; l_max / div >= 1 ; div = div << 1)
30.  {
31.           //Calculate the ofset state
32.           int t = l_max/div;
33.           //Calculate where to test next
34.           intnewTest = index + (l + t)*dir;
35.           //Test if in code range
36.           if (newTest<= lastIndex&&newTest>= 0)
37.           {
38.                    intsplitPrefix = clz(precis ^ list[newTest].x);
39.                    //If the code is higher then our minimum, update the position
40.                    if (splitPrefix>d_min)
41.                             l = l+t;
42.           }
43.  }
44. result.x = min(index,index + l*dir);
45. result.y = max(index,index + l*dir);
46.   return result;
47. }
```

Figure 29determineRange function pseudocode

Lines 6-11 handle the case when we deal with the root – Then the function simply returns the entire range.
Lines 13-15 determine the direction of the range, and then we look for a range to search within, in lines 18-25. The binary search for the other boundary of the range is performed on lines 28-43

As it was the case with *findSplit* function, this function assumes that all the Morton codes are unique. This case is going to be addressed later.

6.2.1.4. Bounding Box calculation

The top level algorithm described in section **6.2.1.1** builds the hierarchy of the tree, but it doesn't calculate bounding boxes of internal nodes – An essential piece of data for tree traversal. Since the whole idea of the algorithm is to process nodes without having to know anything about the rest of the hierarchy, it cannot perform bounding box calculation because bounding boxes of parent nodes are derived directly from bounding boxes of their children (As union of their children, to be precise.)
This leaves us with the need in an additional processing stage – The bounding box calculation. The relation of bounding boxes of parent and child nodes do not allow such an effective parallelism as hierarchy building, but luckily this is not a calculation intensive stage that is assumed to be negligible compared to more significant stages of the algorithm.

In order to be able to calculate the bounding boxes, we will need to record parent-children relations, as it is done in algorithm in **figure 27** in lines 29 and 30. In addition, the tree node data structure must contain the bounding box itself.
Representing the box by vector of its minimal and maximal bounds is a good choice for fast box union calculation, and it also works well with intersection algorithms.
In addition, we will need to store a counter for each node. The counter will be increased by each thread that visits the node, in order to terminate threads that are not needed anymore. Those counters can also be stored in a separate array, per leaf, as an alternative representation.
The pseudocode of a kernel that calculates bounding boxes is presented below:

```
1.  void computeBoundingBoxes(structBVHNode* nodeBuffer, volatile uint* counters, uintleafCount)
2.  {
3.  uint current = parent(nodeBuffer[get_global_id(0)]);
4.  while(current != UINT_MAX)
5.  {
6.  volatile uint* counter_ptr = counters+(current-leafCount);
7.          atomic_inc(counters_ptr);
8.          if (*(counter_ptr) > 1)
9.          {
10.                 mergeBoundingBox(nodeBuffer,current);
11.                 current = parent(nodeBuffer[current]);
12.         }
13.         else break;
14. }
```

```
15. }
```

Figure 28 Bounding Box calculation kernel

The kernel starts its work at leaf node that resides at the index of the given thread, which is obtained by OpenCL *getGlobalId* function – Assuming that we run one thread for each leaf. At line 3 we obtain the parent of the leaf, and traverse up towards the root until root is reached, and the parent value has the value of UINT_MAX to denote that the node is a root node that has no parent.
At each internal node we increase the atomic counter at the location that corresponds to the index of the current node.
Since we store the leaves and internal nodes in the same array and the inner nodes reside in range [N,2N-1] as was specified in section **5.2.1.1**, we translate the parent index from range [N,2N-1] to range [0,N-1] as the counters are stored, and hence the calculation of a pointer in line 6. If it turns out that this is the first thread that visits the node – Terminate it, since it tells us that bounding box of only one child is calculated so far, and we don't have the complete information to compute bounding box for this node. If the visiting thread is the second one, then bounding box of this node can be calculated, and the thread proceeds further towards the root, either to signalize to the parent that it's child's bounding box was calculated, or to calculate the bounding box of the parent itself and continue the process in similar way. The utility method, *mergeBoundingBox* called at line 10 is merely calculating the union of bounding boxes of children of the given node and assigns the result to the bounding box of the given node.

6.2.1.5. BVH Construction – flow summary

Now, as we have all the pieces, we are ready to put it all together. The algorithm on figure 27 in section 6.2.1.1 runs in separate thread.
Before that, however, we need to prepare the data and calculate the Morton codes for each primitive, and initialize the nodes. After building the hierarchy, we also have the stage of computing the bounding boxes.

Following is the summary of the BVH construction flow:

- Allocate arrays:
  - Array of 2N – 1 tree nodes
  - Array of N key-value pairs to contain mapping of Morton codes to leaf indices
  - Array of N atomic counters
- Initialization
  - Initialize data of the leaf nodes
  - Calculate the Morton code for each leaf, and store the Morton code – leaf index pair in the designated array
- Hierarchy construction: Run kernel that calls the function described in fig. 27
- Bounding Box calculation: Run the bounding box calculation kernel, described in fig. 30.

6.2.1.6. Handling Duplicate Morton Codes

The presented algorithms didn't cover the issue of duplicate Morton codes. However, since duplicate Morton codes are common in practice, this issue must be addressed.

The first way to handle this case is the way recommended in the paper that describes the algorithm[35]. According to the proposed approach is to concatenate the index of the Morton code in the sorted array with the Morton code itself – So called "Augmenting". Tero Karas suggests to apply this approach by falling back to indices$i$*I* and j while calculating $\delta(i, j)$, which lets us avoid storing the additional information.

Another way, which was used in my implementation of the algorithm, is the one proposed in a feedback to the article describing the algorithm online [36] – According to the solution proposed by the user, the *determineRange* method should be modified such that in case two identical Morton codes occur, the returned range will begin at the input index and end at the index of the first different element exclusively. The *findSplit*method also should be changed accordingly, such that in case that Morton codes at the indices which are the boundaries of the range are equal, the first index of the range will be returned as split value.

The third way to solve this problem is to utilize the approach of ranges, similarly to the way we used to find ranges in reference array to populate top level cells in Uniform Grid. To apply this approach we will need to store a range of indices of sorted array of Morton code – triangle pairs that shares the same Morton code with the associated leaf.
From construction algorithm perspective, it will work only with unique Morton codes. From traversal perspective, if a leaf reached, the ray is going to be tested for each primitive that shares the Morton code of the leaf. Those primitives will be exactly those that are associated with the Morton code-triangle pairs within the range stored in the leaf.

Those approaches do not have an immediate advantage or disadvantage – Testing which approach works best and easier to implement is a subject for future work.

6.2.2. BVH Traversal

The straightforward traversal algorithm is a recursive top-down tree traversal, where at each internal node we test whether a tested ray intersects with the associated bounding volume, and if it does, proceeds to the children recursively. If a recursion reached a leaf, we report a potential intersection between a ray and the object that is associated with the reached leaf.

Following discussion in chapter 4, it is obvious that we could achieve much better performance if we implement a smarter iterative traversal algorithm with specifics of GPGPU technologies in mind.

A study by Timo Aila and Samuli Laine [30] provides a thorough analysis and compares several ways to implement a tree traversal and discusses the efficiency of each of them.

According to the study, using the previously discussed Persistent Threads yields a better performance than relying on hardware scheduling. Their implementation relies on a traversal stack that resides on thread local memory in case of per-ray traversal. The next step in optimization might be getting rid of the stack. A study by Áfra and Szirmay-Kalo [31] is one of the latest proposals of a BVH traversal algorithm that doesn't feature a stack. However, performance tests of those algorithms show that those algorithms perform slower than the stack based algorithms, even though the traversal state of those algorithms consumes much less memory than for stack-based algorithms. This result is similar on earlier attempts to create a stack-free BVH traversal algorithm.

6.2.2.1. BVH Traversal Algorithm

Considering this information, a simple stack-based traversal would be good for a first experiment. The stack will be allocated in private memory per-thread.

```
1.  struct Contact bvh_generate_contact(struct Ray* ray,structBVHNode* bvh, uintrootIdx,const char* scene)
2.  {
3.  uintstack[32];
4.  uintstackPointer = 0;
5.  uintcurrentIdx = rootIdx;
6.  stack[stackPointer++]=UINT_MAX; //Push initial value
7.  structContactDataresult;
8.  result.contactDist= FLT_MAX;
9.  CL_UINT resMaterialIdx = 0;
10. do
11. {
12.         structBVHNode node = bvh[currentIdx];
13.         if (type(node) == INNER_NODE)
14.         {
15. //Processing internal node
16.                 uintchild_A_idx = childA(node);
17.                 uintchild_B_idx = childB(node);
18.                 struct AABB child_A_box = bvh[child_A_idx].boundingBox;
19.                 struct AABB child_B_box = bvh[child_B_idx].boundingBox;
20. //See which node children are pierced by the ray
21.                 bool aValid = AABBIntersect_private(&child_A_box,ray->origin,ray->direction) > 0
22. || isPointInside_private(&child_A_box,ray->origin);
23.                 bool bValid = AABBIntersect_private(&child_B_box,ray->origin,ray->direction) > 0
24.  || isPointInside_private(&child_B_box,ray->origin);
25.
26.                 if (aValid&&bValid)
27.                 {
28. //Both children intersect – Proceed to A, save B on the stack
29.                         currentIdx = child_A_idx;
30.                         stack[stackPointer++] = child_B_idx; // push
31.                 }
32.                 else if (aValid)//Only A intersects – Proceed to A, forget about B
33.                         currentIdx = child_A_idx;
34.                 else if (bValid)//Only B intersect s – Proceed to B, forget about A
35.                         currentIdx = child_B_idx;
36.                 else//No children intersect – Pop from stack
37.                         currentIdx = stack[--stackPointer];
```

```
38.             }
39.             else
40.             {
41.   //Processing Leaf: Get triangle from scene according to node
42.                         struct Triangle t = getTriangle(triangleIndex(node),scene);
43.                         struct Contact contactData = triangleIntersect(triangle,ray);
44.                         if (contactData.contactDist> 0 &&contactData.contactDist<result.contacDist)
45.                         {
46.                                 result = contactData;
47.                         //The resulting contact may be not the closest one – So keep looking on what is on the stack
48.                                 currentIdx = stack[--stackPointer];
49.                         }
50.             }
51.             while (currentIdx != UINT_MAX);
52.             //Finalizing the result
53.             if (result.contactDist == FLT_MAX)
54.                         result.contactDist = 0;
55.             return result;
56.   }
```

Figure 29 Stack-based BVH traversal

The algorithm is a standard stack-based iterative tree traversal. Inside the traversal loop we handle two cases: When the encountered node is a leaf node, we check intersection with the associated triangle. If intersection is found, we save the result and pop the next node from the stack. The reason not to terminate the traversal at this point is that since we look for the closest intersection, we don't have any guarantee that the intersection we've found is the closest one.
In case intersection is not found, we simply pop the next value from the stack.

When an internal node is encountered, we check intersection of the ray with its children: If both children intersect with the ray, we proceed arbitrary to the direction of child A, and save child B on the stack. If only one of the children intersect – We proceed in the direction of the intersecting child. Finally, when none of the children intersect we pop the next value from the stack.

7. Test Results

7.1. Rendered Images

In order to assess usability and performance of the OpenCL Ray Tracer Library, I've built a test application – A simple ray cast renderer which can render 3D models in Wavefront OBJ format.

Following test images were rendered with resolution of 1024x1024 pixels:

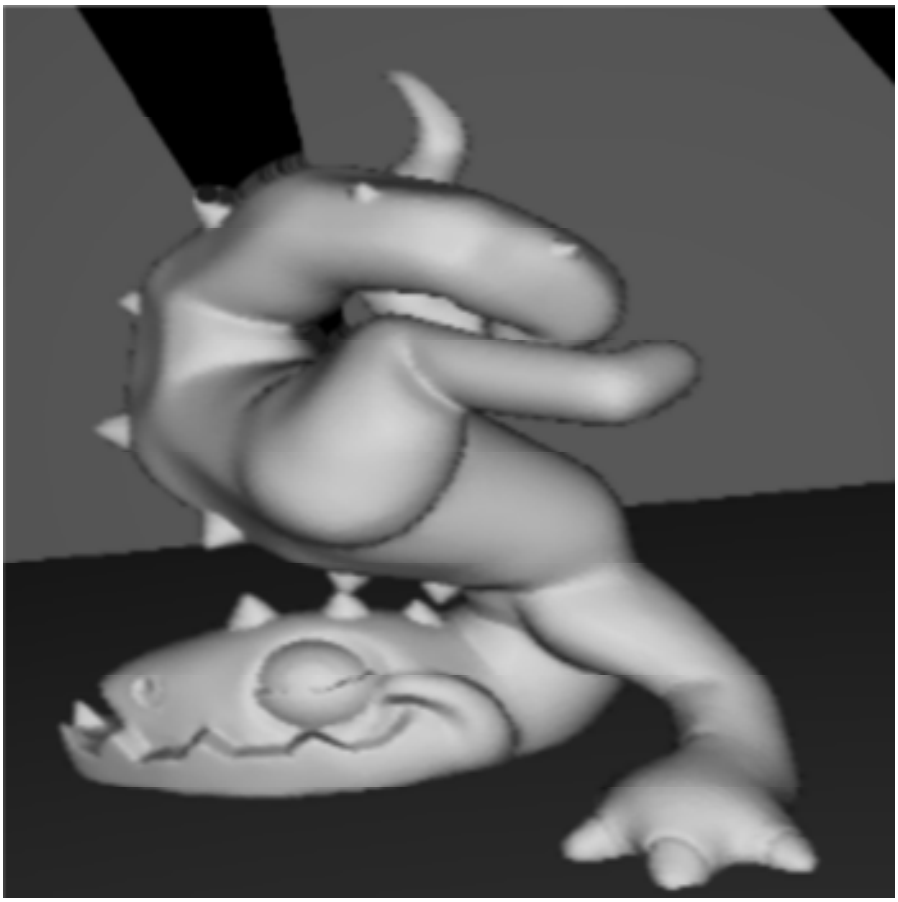

Figure 30 Clumsy Dragon, 50K trangles

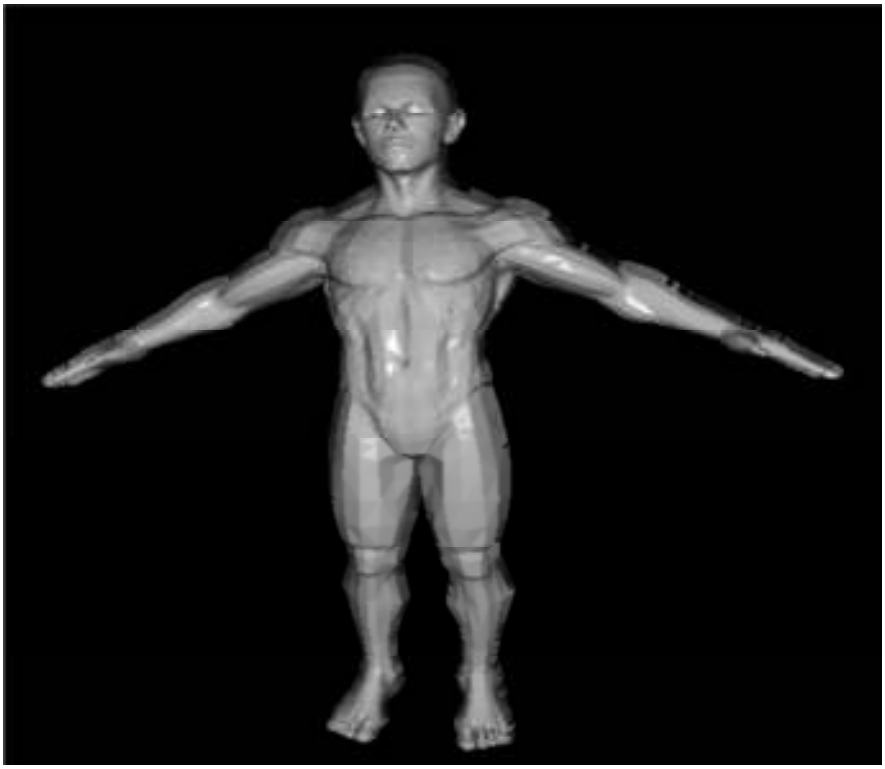

Figure33Male Figure, 39K triangles

Figure 31 Skull, 60K triangles

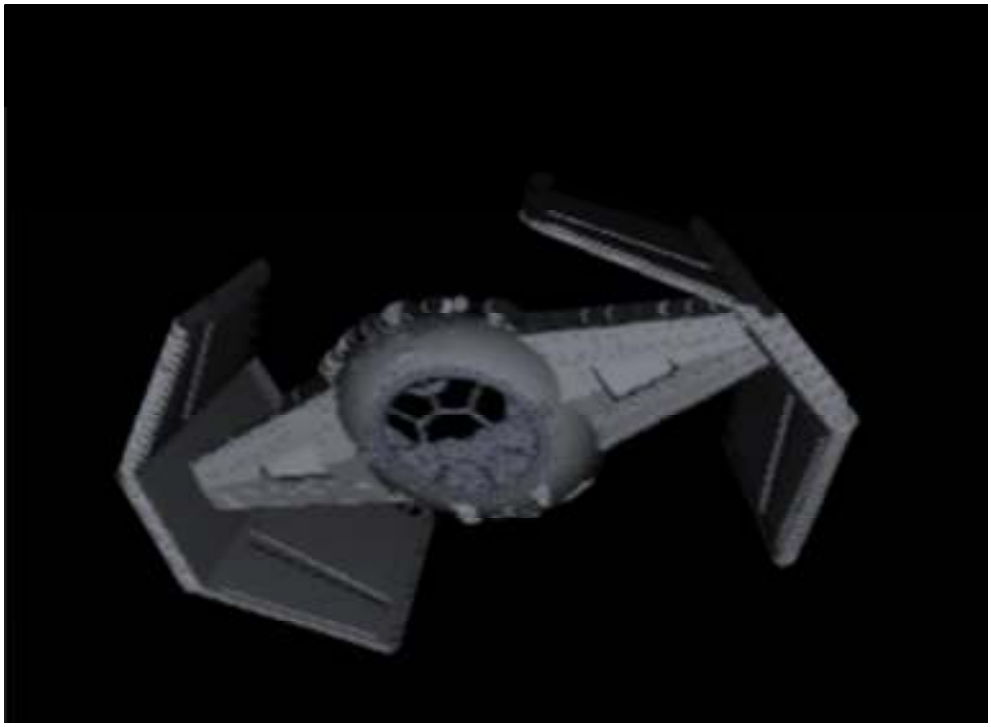

Figure 32 Spaceship, 150K triangles

Figure 33 Anime Character, 18K triangles

The images above demonstrate the capability of the library to render models of various sizes. The demo application featured movable camera, and considering that the testing hardware is a basic budget graphics card for mobile computers, the performance was usable. Of course, for more complex shading and introduction of shadow, reflection and refraction rays and supersampling will affect performance severely. As proof of concept however, the demonstration shows that the functionality provided by the library can be used to build an actual Ray Tracing renderer for practical uses.

7.2. Performance Observations

Kernel Name	Device Name	# of Calls	Total Time(ms)	% of Total Time	Avg Time(ms)	Max Time(ms)	Min Time(ms)
extractLeafCellsKernel	Capeverde	1	169.86963	46.72	169.86963	169.86963	169.86963
generateContactsKernel	Capeverde	1	106.89881	29.40	106.89881	106.89881	106.89881
writeLeafPairsKernel	Capeverde	1	43.58326	11.99	43.58326	43.58326	43.58326
ParallelBitonic_B8	Capeverde	102	24.05378	6.62	0.23582	0.48741	0.07111
prepareGridDataForLeaves	Capeverde	1	7.38341	2.03	7.38341	7.38341	7.38341
extractCellRangesKernel	Capeverde	1	5.57482	1.53	5.57482	5.57482	5.57482
ParallelBitonic_B2	Capeverde	13	2.80074	0.77	0.21544	0.33719	0.07496
ParallelBitonic_B4	Capeverde	12	2.55126	0.70	0.21260	0.35007	0.08267
group_prefixSum	Capeverde	6	0.35067	0.10	0.05844	0.12563	0.02830
global_prefixSum	Capeverde	3	0.19718	0.05	0.06573	0.09630	0.05022
writePairsKernel	Capeverde	1	0.15141	0.04	0.15141	0.15141	0.15141
prepareDataKernel	Capeverde	1	0.12281	0.03	0.12281	0.12281	0.12281
countLeavesAndFillCellKernel	Capeverde	1	0.04193	0.01	0.04193	0.04193	0.04193
updateTopLevelCellsWithLeafRange	Capeverde	1	0.02252	0.01	0.02252	0.02252	0.02252

Figure 34 Frame Rendering Times: Male Figure, using Two Level Grid

Kernel Name	Device Name	# of Calls	Total Time(ms)	% of Total Time	Avg Time(ms)	Max Time(ms)	Min Time(ms)
generateContacts	Capeverde	1	149.18044	95.86	149.18044	149.18044	149.18044
computeBoundingBoxes	Capeverde	1	3.91393	2.51	3.91393	3.91393	3.91393
ParallelBitonic_B8	Capeverde	40	1.57526	1.01	0.03938	0.07422	0.02689
buildRadixTree	Capeverde	1	0.35126	0.23	0.35126	0.35126	0.35126
ParallelBitonic_B4	Capeverde	5	0.21156	0.14	0.04231	0.04933	0.03982
ParallelBitonic_B2	Capeverde	6	0.20089	0.13	0.03348	0.03437	0.03230
calculateMortonCodes	Capeverde	1	0.19245	0.12	0.19245	0.19245	0.19245

Figure 38 Frame Rendering Times: Male Figure, using BVH

For smaller models (39K triangles in this example), when using Grid, the dominant part of the frame rendering is the grid construction, while for BVH, the structure traversal takes 95% of frame rendering time.
The BVH construction appears to be significantly faster than the grid, while the grid traversal performs slightly faster than BVH.

Kernel Name	Device Name	# of Calls	Total Time(ms)	% of Total Time	Avg Time(ms)	Max Time(ms)	Min Time(ms)
writeLeafPairsKernel	Capeverde	1	11540.38637	56.47	11540.38637	11540.38637	11540.38637
extractLeafCellsKernel	Capeverde	1	7727.59393	37.81	7727.59393	7727.59393	7727.59393
prepareGridDataForLeaves	Capeverde	1	808.33615	3.96	808.33615	808.33615	808.33615
ParallelBitonic_B8	Capeverde	127	121.33185	0.59	0.95537	1.86741	0.32052
extractCellRangesKernel	Capeverde	1	116.90918	0.57	116.90918	116.90918	116.90918
generateContactsKernel	Capeverde	1	95.97259	0.47	95.97259	95.97259	95.97259
ParallelBitonic_B2	Capeverde	14	11.89570	0.06	0.84969	1.37007	0.33230
ParallelBitonic_B4	Capeverde	13	11.79111	0.06	0.90701	1.40741	0.34355
group_prefixSum	Capeverde	7	1.13141	0.01	0.16163	0.49570	0.02800
global_prefixSum	Capeverde	4	0.94504	0.00	0.23626	0.37363	0.18859
writePairsKernel	Capeverde	1	0.47570	0.00	0.47570	0.47570	0.47570
prepareDataKernel	Capeverde	1	0.35126	0.00	0.35126	0.35126	0.35126
countLeavesAndFillCellKernel	Capeverde	1	0.16993	0.00	0.16993	0.16993	0.16993
updateTopLevelCellsWithLeafRange	Capeverde	1	0.11437	0.00	0.11437	0.11437	0.11437

Figure 39 Spaceship (153K triangles) rendering kernel times using Grid

Kernel Name	Device Name	# of Calls	Total Time(ms)	% of Total Time	Avg Time(ms)	Max Time(ms)	Min Time(ms)
computeBoundingBoxes	Capeverde	1	770.16622	38.30	770.16622	770.16622	770.16622
generateContacts	Capeverde	1	750.88948	37.34	750.88948	750.88948	750.88948
buildRadixTree	Capeverde	1	478.29985	23.79	478.29985	478.29985	478.29985
ParallelBitonic_B8	Capeverde	51	8.96430	0.45	0.17577	0.25482	0.15674
ParallelBitonic_B4	Capeverde	6	1.02726	0.05	0.17121	0.17407	0.16933
ParallelBitonic_B2	Capeverde	6	0.97363	0.05	0.16227	0.16267	0.16163
calculateMortonCodes	Capeverde	1	0.56830	0.03	0.56830	0.56830	0.56830

Figure 35 Spaceship (153K triangles) rendering times using BVH

For larger model (153K triangles), when using Grid, the grid traversal becomes almost insignificant compared to overall construction time, while with BVH, although the relative amount of time spent in tree traversal becomes less significant, still takes 37% of rendering time. Comparing grid traversal with BVH traversal, for this model Grid traversal is faster by a significant factor of almost 8.

The impact of these results can be experienced when running the demo application: When using Grid structure, the grid construction stage takes some time (Before the image is rendered), but the refresh time between frames, when the camera moves is faster than in case when using BVH.
When using BVH, the construction time is much faster than construction time of a Grid.

For fully interactive scenes, when scene can change from frame to frame, BVH will still yield faster rendering times, because the total frame rendering time with BVH is faster than the total rendering time of Grid. Optimizing BVH traversal further, could make BVH perform significantly faster and more usable for interactive scenes.

8. Further work

This work is a foundation for further research in the field of Ray Tracing on cross-hardware GPGPU platform, and it opens a lot of issues that could be investigated further on.

8.1.1. Code optimization

Since the practical part of this work was done almost from scratch, there is a room to investigate how to make the implementation more clean and effective. Possible optimizations could be more extensive use of hardware intrinsics exposed by the OpenCL standard, manage memory according to platform guidelines, and optimize the implementation of algorithm building blocks such as intersection algorithms and sorting

8.1.2. Dealing with portability issues

Ideally, an OpenCL code that doesn't use hardware-specific extensions should run on any hardware that supports the standard. In practice, however, there are portability issues that affect results of calculations. Those issues can be related to bugs in hardware drivers, or incorrect implementations of the OpenCL standard. Identifying and resolving those issues are subject for future work.

8.1.3. Comparing effect of hardware characteristics on performance

The cross-hardware implementation opens a possibility to run the same algorithm on hardware with significantly different characteristics and examine the effect on performance of algorithms, particularly Acceleration Structures.

8.1.4. Extending the library with additional acceleration structures

The library is designed for extensibility, which can be used for adding more acceleration structures to the library. A possible candidate could be K-D Tree - The parallel BVH construction algorithm can also be modified for fast and effective K-D tree construction.Experimenting and getting this approach to work, by finding a suitable way to relate between primitives and points over which the K-D tree is constructed is a subject for future work

9. Code and Documentation

Code for this project can be downloaded at the following link:
https://github.com/kotturtech/OpenCLRayTracer

The HTML documentation of this project: Utility functions and API is included in project repository in electronic form.

In addition, the source code is included on a CD which accompanies this work.

10. Sources

[1] James Arvo, Robert L. Cook, Andrew S. Glassner, Eric Haines, Pat Hanrahan, Paul S. Heckbert, David Kirk, "*An Introduction to Ray Tracing*", Academic Press, 1989

[2] Peter Shirley, Steve Marschner, "*Fundamentals of Computer Graphics*", A.K. Peters Ltd. 2009

[3] Kevin Suffern, "*Ray Tracing From the Ground Up"*, A.K. Peters Ltd. 2007

[4] Ingo Wald, William R. Mark et al, "*State Of The Art in Ray Tracing Interactive Scenes",* SIGGRAPH 2007

[5] Wikipedia Contributors, Wikipedia – The Free Encyclopedia**https://en.wikipedia.org/wiki/Ray_tracing_(graphics),** 2010

[6] Turner Whitted, "*An Improved Illumination Model For Shaded Display"***,** Proceedings of the 6th annual conference of Computer Graphics and interactive techniques, 1979

[7] Solomon Boulos, Dave Edwards, et al, "*Packet-based Whitted and Distribution Ray Tracing*", University of Utah, 2007

[8] Ingo Wald, Philipp Slusallek et al, "*Interactive Rendering with Coherent Ray Tracing"***,** EUROGRAPHICS 2001

[9] Tim Foley and Jeremy Sugerman*,"KD-Tree Acceleration Structures for a GPU Ray tracer*", Stanford University, 2005

[10] Wikipedia Contributors, Wikipedia – The Free Encyclopedia**,** https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units 2015

[11] NVIDIA™ CUDA® - Home Page http://www.nvidia.com/object/cuda_home_new.html

[12] NVIDIA™ CUDA® Programming Guide http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3mY7Q054Y

[13] Jonathan Thompson, Kristofer Schlachter, "*An introduction to the OpenCL Programming Model"*, New York University, 2012

[14] Brian C. Budge, John C. Anderson et al, *"A straightforward CUDA Implementation forInteractive Ray-Tracing"*, 2008

[15] AMD APP SDK OpenCL User's Guide rev 1.0, 2015
http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf

[16] Wikipedia Contributors, Wikipedia – The Free Encyclopedia, topic: C++ AMP
https://en.wikipedia.org/wiki/C%2B%2B_AMP

[17] Microsoft Corporation, "*C++ AMP : Language and Programming Model*", Version 1.0, August 2012

[18] Timothy J. Purcell, Ian Buck, William R. Mark and Pat Hanrahan,*"Ray Tracing on Programmable Graphics Hardware",*ACM Transactions on Graphics volume 21, July 2002.

[19] Kristóf Ralovich, *"Implementing and Analyzing a GPU Ray Tracer"*, Budapest University of Technology and Economics, 2007

[20] Andrew D. Britton, *"Full CUDA Implementation Of GPGPU Recursive Ray-Tracing"*, A Master's Thesis, Purdue University 2010

[21] Brian Budge, John Anderson, Christoph Garth, Kenneth Joy, "*A hybrid CPU-GPU Implementation for Interactive Ray-Tracing of Dynamic Scenes*". Tech. Rep. CSE-2008-9, UC Davis, 2008

[22] Martin Zlatuška ,VlastimilHavran, "*Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms*", Journal of WSCG 18:1-3, 2010 69-76

[23] Thomas Schiffer, Dieter W. Fellner, "*Ray Tracing: Lessons Learned And Future Challenges*",Potentials, IEEE Volume:32 , Issue: 5 Sept.-Oct. 2013

[24] Kshitij Gupta, Jeff A. Stuart, John D. Owens, "*A study of Persistent Threads style GPU programming for GPGPU workloads*",Innovative Parallel Computing (InPar), 2012, 1-14

[25] Gábor Liktor, "*Ray tracing implicit surfaces on the GPU*", Computer Graphics & Geometry 10, 3 (2008), 36–53

[26] Martin Christen, "*Ray Tracing on GPU*", University of Applied Sciences Basel (FHBB)Diploma Thesis

[27] Artur Lira Dos Santos, Veronica Teichrieb, Jorge Lindoso, "*Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture",* WSCG (2014) Full Papers Proceedings: 22nd International Conference in Central

Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS Association, p. 203-212, 2014

[28] Solomon Boulos, "*Notes On Efficient Ray Tracing*", SIGGRAPH 05 ACM SIGGRAPH Courses, Article Number 10, 2005

[29] Tero Karras, "*Thinking Parallel, Part II – Tree Traversal On GPU*", NVIDIA Dev Blogs, 2012
http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/

[30] Timo Aila, Samuli Laine, "*Understanding Efficiency of Ray Traversal on GPUs*", Proceedings of the Conference on High Performance Graphics Pages 145-149, 2009

[31] Attila T. Áfra, László Szirmay-Kalo, "*Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing*", Computer Graphics Forum Volume 33, Issue 1, pages 129–140, February 2014

[32] Reza Fuad R,Mochamad Hariadi, "*GPU-Based Ray Tracing Algorithm Using Uniform Grid Structure*", Jurnal Ilmiah Ilmu Komputer UPH 05/2011

[33] Javor Kalojanov, Philipp Slusallek, "*A parallel algorithm for construction of uniform grids*",Proceedings of the Conference on High Performance Graphics 2009, Pages 23-28

[34] Javor Kalojanov, Markus Billeter, Philipp Slusallek, "*Two-Level Grids for Ray Tracing on GPUs*",Computer Graphics Forum Volume 30, Issue 2, pages 307–314, April 2011

[35]Tero Karras (NVIDIA), "*Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees*", High Performance Graphics 2012, June 2012

[36]Tero Karras (NVIDIA), "*Thinking Parallel –Part III, Tree Construction*", NVIDIA dev blogs, 2012,https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/

[37] Kun Zhou et al, "*Real-time KD-tree construction on graphics hardware*", ACM Transactions on Graphics Volume 27 Issue 5, December 2008

[38] Tim Foley, Jeremy Sugerman, "*KD-tree acceleration structures for a GPU raytracer*", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware Pages 15-22 ACM New York, NY, USA ©2005

[39] HWRT Seminar, Background Papers on Hardware Ray Tracing http://www.eng.utah.edu/~cs6958/papers/HWRT-seminar/

תוכן עניינים

בניין חשובות לאלגוריתמים. בנוסף לכך, כיוון שהפלטפורמה מיועדת לחומרה ספציפית, היא גם מציעה ביצועים משופרים לעומת פלטפורמות כלליות יותר. יתרונות אילו באים על חשבון כך שניתן להריץ את האלגוריתמים על חומרה המיוצרת ע"י חברת NVIDIA בלבד.

התוצר של הפרוייקט הנתון הוא מימוש אלגוריתמים ומבני נתונים מקביליים עבור אלגוריתם עקיבת קרניים על גבי פלטפורמה שאינה תלויה בחומרה – OpenCL . תוצר הפרוייקט הינה ספריית עזר המכילה אלגוריתמים ומבני נתונים מקביליים שניתן להיעזר בהם למימוש אלגוריתם עקיבת קרניים תוך ניצול משאבי חישוב מקבילי של מעבדים גראפיים מודרניים.

תקציר

מעקב קרניים (Ray Tracing) הוא אלגוריתם בסיסי לציור עצמים תלת-מימדיים. עבור כל פיקסל האלגוריתם יוצר קרן היוצאת מהפיקסל, מחשב את הנקודה הקרובה ביותר שבו הקרן פוגעת בעצם תלת מימדי וקובע את הצבע של פיקסל על פי נקודת פגיעה של הקרן בעצם תלת מימדי. עם זאת שהאלגוריתם הוא מאוד פשוט ובעל יכולת לצייר תמונות באיכות גבוהה, הוא גם מאד תובעני כלפי משאבים של המערכת (כח עיבוד), מה שמגביל את השימוש המעשי בו. לעיתים, ציור תמונה אחת באיכות גבוהה יכולה לקחת לילה שלם.

האופטימיזציה החשובה ביותר שנעשתה באלגוריתם עקיבת קרניים, ובעצם מאפשרת את השימוש בו באופן מעשי היא שימוש במבני נתונים למיון עצמים תלת-מימדיים במרחב, המאפשרים לצמצם משמעותית את כמות בדיקות לגילוי באיזה אובייקט הקרן פוגעת. מבני נתונים אלה נקראים גם Acceleration Structures. מבני נתונים אילו מאפשרים שלילה מהירה של אובייקטים שנמצאים רחוק מהקרן, ובדיקה של עצמים שנמצאים קרוב לקרן בלבד, בניגוד לגישת Brute Force, לפיה עבור כל קרן יש לעבור על כל העצמים ולבדוק האם הקרן פוגעת בו.

מאפיין נוסף של האלגוריתם הוא הידידותיות שלו לעיבוד מקבילי – ניתן לראות עיבוד של כל קרן כמשימה נפרדת, שהיא בלתי תלויה בעיבוד קרניים אחרות. לכן מימוש עקיבת קרניים על חומרה מקבילית נותן שיפור משמעותי בביצועים של האלגוריתם.

דוגמא החומרה כזו היא המעבד הגראפי - Graphical Processing Unit (GPU). ה - GPU הוא מעבד המיועד לחישוב מקבילי מאסיבי. בדרך כלל מעבדים אילו מורכבים מכמה ליבות עיבוד, ומותאמים לחשיובים וקטוריים באופן מקבילי. אמנם המעבדים האילו מיועדים במקור לעיבוד גראפי וראזטריזציה, התברר שניתן להישתמש במעבדים האלה לחישובים גם למטרות כלליות יותר.

השימוש במעבדים גראפיים לחישובים כלליים נעשה קל ונגיש יותר עם הופעת פלפורמות GPGPU – General Purpose Graphics Processing Unit . פלטפורמות אילו כוללות ממשק למפתחים, ספריות זמן ריצה וכלי פיתוח נוספים לכתיבת תוכנות למעבדים גראפיים. הופעתן של פלטפורמות אילו בשמצע-סוף שנות ה-2000 פתח תחום למחקר שמטרתו למצוא דרכים לנצל את משאבי החישוב המקבילי של מעבדים גראפיים כדי לשפר ביצועים של אלגוריתם עקיבת קרניים, והאלגוריתמים ומבני נתונים הרלוונטיים לעקיבת קרניים.

פלטפורמות GPGPU הנפוצות בשוק כיום הינן NVIDIA CUDA ו- OpenCL. NVIDIA CUDA הינה פלטפורמה שפותחה ע"י חברת NVIDIA ונתמכת ע"י חומרה המיוצרת ע"י NVIDIA בלבד. OpenCL לעומת זאת, הוא תקן שכל יצרן חומרה יכול לממש את הפונקציות המפורטות בו. תקן זה נתמך ע"י רוב ייצרני מעבדים גראפיים כיום, ותכניות הכתובות עבור פלטפורמת OpenCL יכולות לרוץ על כל חומרה שתומכת בתקן.

ברוב עבודות המחקר החלק המעשי ממומש על פלטפורמת NVIDIA CUDA , במטרה למדוד ביצועים ולהשוות אלגוריתמים, זאת כיוון שהפלטפורמה מתוחזקת היטב, ומספקת ספריות עזר רבות המספקות אבני

האוניברסיטה הפתוחה
המחלקה למתמטיקה ומדעי המחשב

מימוש מבני נתונים להאצת אלגוריתם עקיבת קרניים על פלטפורמת GPGPU שאינה תלוייה בחומרה ספציפית - OpenCL

פרוייקט מתקדם במדעי המחשב (קורס 22997) מוגש כחלק מהדרישות לקבלת תואר "מוסמך למדעים" M.SC במדעי המחשב באוניברסחטה הפתוחה
החטיבה למדעי המחשב

הוגש על ידי: טימור סיזוב

הפרוייקט הוכן תחת הדרכתה של
ד"ר מיריי אביגל

06-06-2016