Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

GPGPU Accelerated Visualization of Complex Geometries

Bachelorarbeit
Teil 2

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Bernhard Manfred Gruber

Betreuer: Dipl.-Ing. (FH) Alexander Leutgeb, RISC Software GmbH, Hagenberg
Begutachter: FH-Prof. DI Dr. Werner Backfrieder

Hagenberg, September 2013

Acknowledgments

I want to thank my professor Herwig Mayr for putting in a good work for me at the RISC software company GmbH. On the technical site, I also want to thank my supervisor at RISC, Alexander Leutgeb, for the excellent support on all subjects of the project. Personally, also a thank you for several interesting discussions we have held after work. Furthermore, I want to give thanks to the RISC company itself, for providing me with high quality equipment and an amazing work station during my three month internship which made working a pleasure. Finally, also a thumb-up to Michael Hava for having the most in-depth C++ knowledge of a human being I have ever met. I learned a great deal by reading your code.

Kurzfassung

Diese Arbeit stellt eine detaillierte Dokumentation über das Berufspraktikum des Autors bei der RISC Software GmbH dar. Nach einer kurzen Einführung in das Unternehmen sowie das Projekt Enlight werden die Ziele des Praktikums erläutert, welche verschiedene Aspekte der Implementierung eines Raycasters mit OpenCL betreffen, der in den drei Praktikumsmonaten erstellt wurde.

Im Anschluss an die Einführung werden grundlegende Themen sowie Basiswissen behandelt, das in späteren Kapiteln benötigt wird. Dazu gehört ein Grundverständnis über Raycasting, reguläre Gitter als Beschleunigungsstruktur, Raycasting implizit beschriebener Geometrie über boolsche Subtraktion und OpenCL als Technologie für GPU Beschleunigung.

Bevor sich die Arbeit in die eigentliche Implementierung vertieft, wird der zum Praktikumsbeginn bestehende Prototyp detailliert analysiert. Dabei wird vor allem auf fortgeschrittene Algorithmen zur Optimierung des eingesetzten Raycasting Verfahrens eingegangen.

Anschließend fokussiert sich die Arbeit auf verschiedene OpenCL Programme, deren Ziel es ist GPU beschleunigt Bilder in ähnlicher Qualität wie die existierende Implementierung zu erzeugen. Auf Vorteile und Schwierigkeiten während der Implementierung der OpenCL Programme wird ebenfalls eingegangen.

Die abschließenden Laufzeitvergleiche der funktionierenden OpenCL Raycaster mit der existierenden CPU Implementierung werden noch durch Erfahrungen mit Entwicklungswerkzeugen rund um OpenCL und einer kurzen Diskussion über noch offene Probleme abgerundet.

Abstract

This thesis provides a detailed coverage of the authors internship at RISC Software GmbH. After a short introduction to the company and the Enlight project, the goals of the internship are discussed, which address various aspects of an OpenCL ray caster implementation created during the three months working period.

In succession to the introduction, fundamental topics required in further chapters of the thesis are covered. These include the principle of ray casting, regular grids as acceleration structure, ray casting implicitly described geometry using boolean subtraction and OpenCL as technology for GPU acceleration.

Before the thesis deepens into the actual implementation, the existing prototype at the start of the internship is analyzed in detail. This includes advanced algorithms to optimize the used ray casting approach.

The primary focus then lies on several OpenCL programs with the goal of reproducing the visual output of the existing CPU implementation using GPU acceleration. Advantages and difficulties of developing with OpenCL are encountered during the explanations of the implementations.

The final benchmarks of the working OpenCL ray casters are than rounded off by experiences made with various development tools around OpenCL and a discussion of the still remaining problems.

Contents

1 Introduction

1.1 About RISC Software GmbH

The RISC Software GmbH is a limited liability company and part of the software park Hagenberg located in Upper Austria. It has been founded in 1992 by Bruno Buchberger as part of RISC, which was founded five years before and stands for Research Institute for Symbolic Computation. While the institute conducts basic research in the fields of mathematics and computer science, the RISC Software GmbH dedicates itself to the application of mathematical and computational research. Besides a small set of software products, the RISC Software GmbH's main focus lies on providing customized software solutions using state-of-the-art and even cutting-edge technologies.

The company is divided into four departments (called units) designating its primary competences:

- Logistic Informatics
- Applied Scientific Computing
- Medical Informatics
- Advanced Computing Technologies

The company operates profit-orientated with a non-profit character as, according to the company agreement, all profits will be reinvested into research and development [4].

1.2 About Regio 13 Enlight

Regio 13 is a political program to sustainably improve the contestability of regional companies, economic growth and employment inside of Upper Austria. The program is co-funded by the European Union with a total budget of 95.5 million euro. The program's duration is from 2007 to 2013. Within this time, companies may apply for financial subventions by the government (Land Oberösterreich) for projects meeting a defined list of requirements.

The Enlight project is conducted by the Applied Scientific Computing unit of the RISC Software GmbH and funded entirely by the Regio 13 program. The project is based on a ray casting prototype developed by Michael Hava during his three month internship in 2011 [6]. The project officially started on 2011-07-01 and is scheduled to be finished at the end of 2013. The project proposal from 2011 defines the following goals for Enlight [5] :

- Development and exploration of new algorithms for visualization and modification of very complex and detailed geometries with file sizes of several gigabytes. Therefore, an optimized ray casting method will be used in favor of traditional and wide-spread rasterization, as the latter is inefficient per design for this dimension of geometries.
- The design of a software system architecture capable of using modern many core processors and GPUs as efficient as possible concerning parallel scalability of visualization.

- Development of a prototype and base library for demonstrating the performance potential on the use case of subtractive manufacturing.

- Additionally, the base library should be the foundation for the further development of a general and industrially applicable software library.

Besides the goals from the initial proposal, several derived goals and restrictions have been defined:

- The visualized model always consists of an initial volume (stock) from which further volumes are subtracted. Every volume must be water-tight[1] and consist only of triangles.

- To allow visualizing frequent geometry changes, such as in subtractive manufacturing, the software should be capable of handling at least ten scene updates (adding ten subtraction volumes) per second. This has a significant impact on the chosen data structure used to represent the scene.

- Furthermore, the internal data structure should be able to handle up to 100,000 subtraction volumes.

- The visualization should be interactive. Interactivity is described in the proposal by referring to an external article with at least ten frames per second [7].

- The visualization should be as precise as possible.

- The ray casted image should also contain depth values in order to merge it with OpenGL rendered geometries.

1.3 Initial situation and goals of the internship

The internship started on 2013-04-04 and lasted until 2013-06-30. As the project has already been running for two years at the time of joining the RISC Software GmbH, the major work of the project has already been done. The created application is console based offering a variety of commands. Several mesh formats (STL, PLY and PTM) have been implemented to allow loading geometry from the file system. The triangles are processed and already organized in a special kind of data structure (discussed in chapter 3.2). An additional window is used to show a preview of the loaded meshes using OpenGL (no subtraction of the volumes). Rotating and zooming the preview is also possible using cursor draging and mouse wheel scrolling. Via corresponding commands on the console ray casting can be enabled, which opens a second window showing the ray casted result. The underlying ray casting implementation runs on the CPU and does already perform well. A further CUDA implementation was in progress.

According to the job description of the internship agreement, the primary objective is to implement a ray caster based on the existing code infrastructure and data structures which utilizes modern hardware architectures (multicore CPUs and GPUs) as effectively as possible considering parallelism using OpenCL. Starting from this statement, several smaller subgoals have been defined during the internship:

- Implementation of an OpenCL infrastructure which allows running GPU accelerated ray casting algorithms and integrating it into the existing code base. Thus, corresponding commands have to be added to the console application and connected with the OpenCL infrastructure.

[1] A water-tight volume (sometimes also called solid) is a volume which is fully enclosed by a surface. In other words, the surface does not contain any leaks. If the volume was filled with water, no water could leave the volume.

- Implementation of a ray caster processing single rays independently. Therefore, the provided data structures, geometries, camera settings, etc. from the application core have to be processed to generate a valid output which can be handed back for successful display.

- Implementation of a ray caster processing rays grouped together in ray packets. This variant showed significant performance improvements on the CPU. Therefore it should also be evaluated on the GPU.

- Porting the implemented OpenCL infrastructure as well as the ray caster algorithms to a new prototype which will serve as the base of the final product.

- Implement out of core ray casting for geometries larger than the available GPU memory.

- Implement selective anti aliasing of the calculated image (canceled).

1.4 Chapter overview

After this short introduction to the company and the organizational part of Enlight, chapter 2 will continue with technical fundamentals about the knowledge fields Enlight encompasses. After a short overview about ray casting and acceleration structures, a specialized ray casting algorithm will be discussed using object entry and exit counting to determine the visual surface. A final coverage of GPGPU computing with OpenCL finishes this chapter, providing the information required to understand the subsequent algorithmic designs and presented implementations.

In chapter 3 a general overview of the existing application infrastructure is given. After the big-picture design is shown, the used data structure for ray casting is explained in more detail. Furthermore, the algorithm for adding newly loaded subtraction volumes to the data structure is presented, as the chosen strategy has a great impact on the ray casting algorithms, which is discussed at last.

The main chapter 4 then presents the implemented OpenCL infrastructure as well as several ray caster implementations. This chapter is organized chronologically showing the progress and evolution of the algorithms during the internship. It starts with the creation of an OpenCL environment and continues with several ray caster implementations and versions of them. The chapter ends with porting the created code to a new prototype and integrating out of core capability.

The final chapter 5 rounds off by showing several benchmark results on different scenes and compares the OpenCL implementation with the preexisting and finished CPU ray caster. Furthermore, experiences during development are shared and existing problems discussed.

2 Fundamentals

2.1 Raycasting

Ray casting is primarily an imaging technique and was first described by Scott Roth in 1982 as an effective method of rendering Constructive Solid Geometry (CSG) objects. Although ray casting may be used for different purposes, we will focus on generating a two dimensional image from a three dimensional scene. The basic principle therefore is shown in Figure 1.

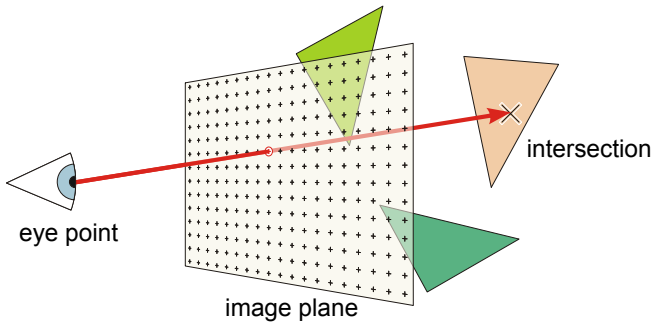

Figure 1: Principle of ray casting.

Generating an image of a three dimensional scene of objects is done by sending rays from a defined eye point through the image plane into the scene. The image plain resembles the two dimensional output image which should be generated. Each output pixel is transformed into a three dimensional position on the image plane which is passed through by one ray originating from the eye point. The eye point itself is usually positioned centered before the image plane to achieve a perspective view of the scene. Each ray is then guided through the three dimensional space containing the scene objects trying to find an intersection. In the case an object is hit, a color value is calculated for the intersection point on the object using e.g. materials parameters or textures of the surface. The pixel on the image plane, through which the ray was shot initially, is then set to this color.

Ray casting is different from the wide-spread rasterization used for most 3D applications and computer games today as it approaches from the image side instead of the objects, the scene is composed of. Rasterization takes all the objects placed in the scene, which have to be composed of primitive shapes like triangles, and transforms them from three dimensional space to the two dimensional space of the output image where each object is then rasterized onto the pixels of the output image.

Apart from a lot of technical differences, both methods also differ in their complexity. Mark Kilgard describes the asymptotic complexity of rasterization with $\mathcal{O}(b*m)$, whereas an accelerated ray tracing implementation using a favorable data structure achieves $\mathcal{O}(logb * p)$, where b is the number of objects in the scene, m is the average number of pixels per object and p is the number of pixels in the scene [8, p.48]. The Enlight proposal additionally points out, that the size of computer screens increased far slower over the past decades than the size of computer memory, computational power and the size of industrial Computer-aided design (CAD) and Computer-aided manufacturing (CAM) models [5]. Therefore, ray casting might be beneficial when used to render large scenes composed of huge amounts of triangles.

2.2 Regular grids

Regular grids are a type of data structures frequently used in ray casting/tracing to accelerate the traversal of rays through the scenes. Alternative approaches are often tree based such as kd trees, octrees, binary space partitioning (BSP) or bounding volume hierarchy (BVH). Each of these techniques has its own strengths and weaknesses depending on the scene and application. However, especially kd trees and grids (and multilevel grids) have been primarily focused in recent work about interactive ray casting/tracing [12, ch.1]. While kd trees usually perform amazingly well on static scenes, dynamic or animated objects still present a challenge as the used data structure has to be rebuilt after every scene change, which can take up to several seconds for larger sets of geometries [12, ch.1]. Regular grids are simpler structures for partitioning space and accelerating ray casting. They can be created quicker than a (balanced) tree, thus allowing more frequent scene changes. Nevertheless, kd trees may be up to a magnitude faster when it comes to the actual ray traversal [12, ch.1]. As Enlight focuses on visualizing dynamic scenes (for subtractive manufacturing) allowing at least ten subtraction volumes to be added to the scene per second, regular grids appear promising for being used in this case.

Building a regular grid data structure is simple. Initially, a bounding box is created around the scene. This box is then divided into equally sized cubes, which may force the bounding box to be enlarged a bit to fit a whole number of cubes in each dimension. These cubes are called cells. For each triangle of the scene all spanned cells are calculated and the cells store a reference to that triangle.

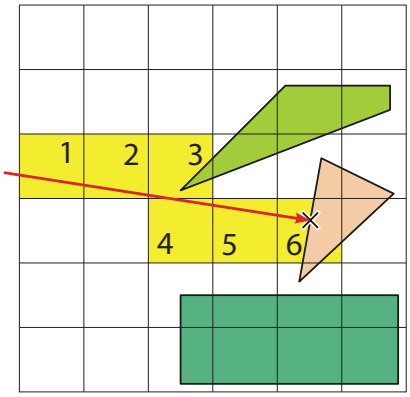

Figure 2: Traversing the cells of the grid using a DDA variant until an intersection has been found.

During ray casting, the rays have to traverse the cells of the grid in order to find potential geometry they might intersect with, cf. Figure 2. A fast algorithm for traversing regular grids using a single ray is given by John Amanatides and Andrew Woo [1]. Their algorithm is a slight modification of the digital differential analyzer (DDA) which is used for the rasterization of lines. As finding the pixels, which are essentially squares inside a regular 2D grid, covered by a line is actually the same problem as finding the voxels hit by a ray, this adapted 3D DDA algorithm can be used for efficient traversal of regular grids. Furthermore, as the algorithm's start point can be determined and the voxels (grid cells) are traversed incrementally, no depth sorting has to be performed across the hit cells and the traversal can be stopped on the first found intersection. Inside each cell, all triangles have to be intersected and the intersections have to be sorted by depth in order to find the correct hit and visible surface.

2.3 Boolean raycasting

Enlight focuses on subtractive manufacturing. Therefore, the scene is composed of an initial stock volume from which further volumes are subtracted, cf. goals in chapter 1.2. This scenario can be typically found in CSG modeling, where complex models are build from boolean combinations of simpler models. CSG models can be efficiently rendered using ray casting, by using a tree with the volumes as leaves and the boolean operations applied to them as nodes. The rays are then traversed through the scenes and have to resolve the tree on every volume entry or exit. Enlight only supports a subset of CSG by only allowing boolean subtraction from the initial stock volume. However, by inverting the stock volume via inversion of the surface normals and defining the world space as being filled instead of being empty, the problem can be even more simplified as the scene now only consists of subtraction volumes.

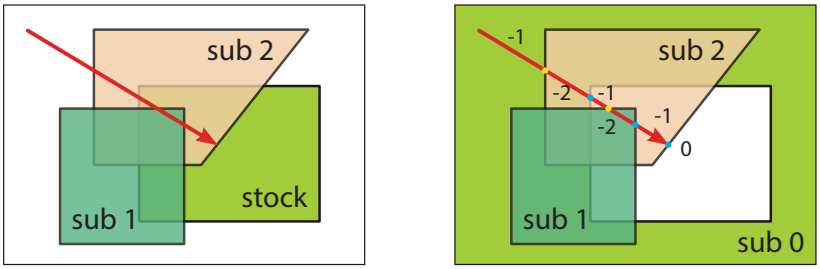

Figure 3: Ray traversing through a scene containing one stock and two subtraction volumes. By inverting the stock, which is shown in the left figure, into a subtraction volume, as shown in the figure on the right, the counting algorithm is simplified. The inversion is accomplished by inverting the normal vectors of the stock's surface. The ray starts inside the inverted stock volume with a counter value of minus one, is decremented on every volume entry and incremented on ever volume exit. The entries are shown as yellow dots and the exits as blue dots in the right figure. If the counter reaches the value zero, the surface has been found.

The algorithm for ray casting a scene composed of subtraction volumes is illustrated in Figure 3. Every ray starts outside the initial scene and maintains a counter which determines the number of subtraction volumes the ray is inside of/has entered. As outside of the original scene is inside the inverted stock, this value is initialized with minus one, corresponding to "inside one volume". The ray is then traversed through the scene. On every intersection where the ray enters a volume, the counter is further decremented, as the ray is now inside even more volumes. On every exit, the counter is incremented, as the ray is now inside one less volume, and checked if it has become zero. A counter value of zero on a volume exit corresponds to an intersection with the surface of the resulting model.

2.4 Packet casting

A common technique for accelerating ray casting and tracing is to accumulate squares of adjacent rays into ray packets and traversing the whole packets through the scene instead of individual rays. This technique has several advantages. Most importantly, a part of the parallelism between individual rays, which can be processed completely independent, is moved down into the traversing and intersection algorithms. As adjacent rays are likely to traverse through the same parts of the acceleration structure, the same cells of a grid in the case of Enlight, and hit the same triangles, the corresponding routines can be executed in a data parallel fashion on multiple rays. Modern CPUs and GPUs benefit from these scenarios and offer corresponding SIMD instructions (CPU) or an appropriate hardware architecture (GPU). Furthermore, coherent traversal and intersection

improves caching behavior via temporal locality[1]. Additionally, also traversal overhead is reduced by the number of rays inside a packet. However, traversing packets through acceleration data structures such as grids is more difficult and cannot be done using the 3D DDA algorithm presented in Chapter 2.2. Fortunately, an efficient solution is presented in the paper "Ray Tracing Animated Scenes using Coherent Grid Traversal" [12]. The algorithm is illustrated in Figure 4.

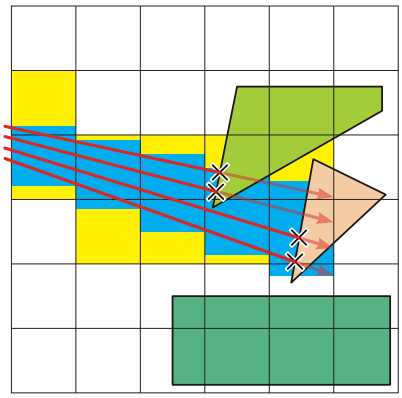

Figure 4: A ray packet traversing a regular grid slice by slice. The blue region is the axis aligned part of a slice (column) which is actually occupied by the packet and is determined by the corner rays of the packet. The yellow region is the blue region extended to full cell boundaries. Furthermore, the packet is always fully traversed until all rays have hit, although no intersections have to be calculated for rays which have already hit.

As the ray packet may now spans multiple cells it cannot traverse the grid cell by cell anymore. The next bigger aggregation of cells is a slice of the grid. To iterate the slices of a grid in a favorable order, the main/primary traversal axis is determined by finding the largest component of the direction vector of one of the innermost rays of the packet. The two other axes are called secondary axes. For each slice of the grid along this main traversal axis, starting at the side of the grid where the packet enters, the four corner rays are intersected with the front and back plane of the slice. The maximum and minimum values of the intersection points along the secondary axes describe the axis aligned region of the slice which is hit by the packet. This region is shown blue in Figure 4 and further extended to full cell boundaries shown in yellow. Then, all objects of the cells of this extended region have to be intersected with all rays of the packet. This routine can be easily vectorized by accumulating several rays into SIMD registers and processing the intersection tests using vector instructions like SSE or AVX. Rays which have already hit a triangle may be masked out. A further optimization to this algorithm is to cull all triangles of the selected cell region of the slice against the frustum spanned by the corner rays.

2.5 OpenCL

OpenCL is an open and free standard for parallel, general purpose programming targeting mainly CPUs and GPUs. This chapter will provide a quick introduction into using OpenCL for executing programs on the GPU. For detailed information, please have a look at the OpenCL chapter in the first part of this thesis - GPGPU Computing with OpenCL.

OpenCL is specified by the Khronos Group [11]. Implementations of OpenCL however have to be provided by hardware vendors such as NVIDIA, AMD or Intel. These vendors provide (Software Development Kits (SDKs)) containing C header files and static libraries which can be used to write applications using OpenCL.

[1]Temporal locality refers to accessing the same location in memory frequently within a short time.

To allow OpenCL to support various kinds of hardware, a separate language is used to write programs to be run on devices supporting OpenCL. These programs are called kernels and have to be included with the application which wants to use them, which is called the host application. This host application uses the OpenCL API to communicate with an OpenCL implementation present in the system. An application using OpenCL typically begins with selecting one of the available OpenCL platforms on the system which correspond to installed OpenCL drivers. A system may have different drivers for different devices from which the application has to choose one. Then, a device is selected supported by this platform. E.g the system's GPU on the NVIDIA platform. Furthermore, a context is required which provides a container for various OpenCL objects like buffers or kernels as well as a command queue which is used to enqueue operations for execution on a device. To execute an OpenCL program on the device, the source code of the program, written in OpenCL C, has to be passed to the OpenCL driver for compilation and linking at runtime. On success, kernels can be extracted providing entry points to the OpenCL program. A kernel can be any function of the written OpenCL program which returns void and is prefixed with the `__kernel` attribute. Kernels may also have arguments which can be set by the host application. On the host side, kernels can be enqueued on the command queue for asynchronous execution on a device, after their parameters have been set. For transferring larger amounts of data and for retrieving results from a kernel buffer objects can be created. These buffers can be asynchronously (default) or synchronously read and written from the host application by enqueuing corresponding operations on the command queue. Buffers can be passed as arguments to a kernel which can access the buffers via a pointer.

Although OpenCL is not tied to a hardware architecture, it does fit GPUs very well. A modern GPU typically consists of several hundreds or even thousands of cores able to process work in parallel. These cores are usually part of larger hardware compounds called streaming multiprocessors (SM). Each of these SMs has a small local memory which is shared between all cores of the SM and a large amount of registers which are allocated by the cores for executing kernels. The cores themselves do not have separate registers. Outside the SMs is the memory system connecting to the off-chip video memory on the graphics card, which can be accessed by all cores.

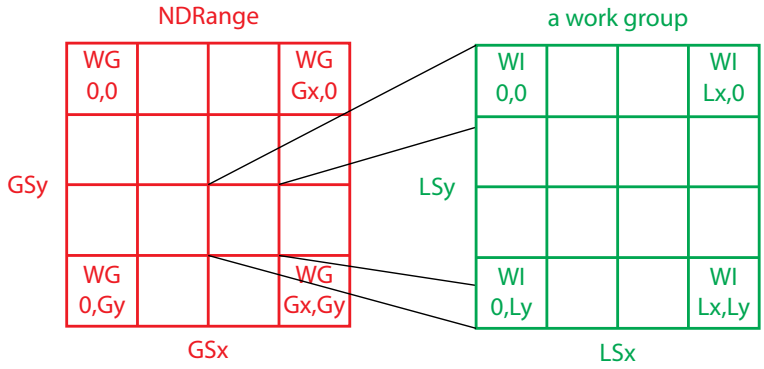

Figure 5: A two-dimensional NDRange consisting of $GSx * GSy$ work groups. Each work group consists of $LSx * LSy$ work items. A work item has two indexes in each dimension, the local index (relative to the work group) and a global index (relative to the NDRange), as well as the indexes of the group it is part of. The local work sizes are LSx and LSy whereas the global work sizes are $GSx * LSx$ and $GSy * LSy$.

OpenCL abstracts such massively parallel hardware architectures using several concepts. When the host application enqueues a kernel, it has to specify a so-called n-dimensional range (NDRange) which determines the dimension and size of the problem. E.g. a two-dimensional range with 800 × 600 elements. According to this range, work items are created, grouped together into work groups. The total number of work items in each dimension is called the global work size. The size of the work groups in each dimension is called the local work size. Each dimension of the global work size must be a multiple of the same dimension of the local one. A work item corresponds to a single execution of a kernel inside a single thread on the GPU. Each work item has several coordinates (ids) it can use to determine it's position inside the NDRange which is illustrated in Figure 5.

Work items inside a work group are executed on the same SM, share the residing local memory and may be synchronized. The off-chip video memory is called global memory in OpenCL, is used to store buffers and can be accessed by all work items and the host application. Variables placed in and pointers to the local memory have to be prefixed with the `__local` attribute, pointers addressing global memory have to be prefixed with the `__global` attribute.

3 Existing prototype

At the beginning of the internship the existing code base, shortly described in Chapter 1.3, was analyzed in order to understand how the ray casting application has been organized and implemented. The application is written in C++ using Microsoft Visual Studio 2010 and 2012 in 64 bit mode. Besides the provided compilers from Microsoft, also Intel's C++ compiler is used during development to benefit from stronger optimization. The code heavily uses C++11 features and AVX SIMD instructions, thus limiting the application to rather up-to-date compilers supporting C++11 and newer hardware. AVX is supported since Intel's Sandy Bridge and AMD's Bulldozer series. Furthermore, OpenMP is used as a technology for high level parallelization, OpenGL for visualization and the Microsoft Foundation Class (MFC), a C++ wrapper of the Win32 API, for window management and user interaction. Considering the implemented algorithms, single ray casting of scenes composed of subtraction volumes as shown in chapter 2.3 using the 3D DDA algorithm discussed in Chapter 2.2 has been used in the beginning. However, the initial approach has later been replaced by a highly optimized and parallel packet ray caster using the slice traversing algorithm presented in Chapter 2.4. The single ray variant can still be found in the code but is not used anymore.

Figure 6 shows screenshots of the existing prototype at the beginning of the internship.

3.1 Structure

Figure 7 shows a simplified class diagram of the most important classes used in the existing ray casting implementation. Most of them have been dealt with when extending the existing infrastructure by the new OpenCL ray caster.

The central class containing most of the application logic is `DebugView`. It inherits `CWnd` from the MFC and is the main window of the application containing the OpenGL visualization. Besides all graphical user interactions, like zooming and rotating, using the cursor, also the console commands are parsed and processed by this class. It further contains an instance of `Camera` which is updated according to user interaction. Also the regular grid holding all loaded meshes is stored by `DebugView`. Finally, a ray caster wrapper class is also available (`RayCasterWrapper`), which delegates calls to the actual ray casting implementation. When the application has been started, meshes can be loaded from the file system via corresponding console commands. The loaded meshes are processed by the classification algorithm and merged into the grid, cf. Chapter 3.3. Ray casting can be triggered either by issuing a command on the console or changing a camera property via GUI input (zoom, rotation). In both cases, `DebugView` eventually invokes `castWithRayPackets()` from the ray casting implementation wrapper and passes a reference to the grid and current camera settings. The wrapper delegates this call to the corresponding method in `RayCaster` where the actual ray casting code is executed. Initially, several camera values are retrieved and a `RayPacketFactory` is created. This factory relies on the older `RayFactory` used for the initial single ray casting algorithm. Furthermore, an instance of `PixelGrid` is created which will be later used to store the ray casting results and will be returned to the calling `DebugView` class, where it is used to render the casted image. After this setup, the requested output image area, sized according to the width and height of the camera, is divided into a grid of square packets and iterated over by two nested for loops. The square size is adjustable and currently set to eight. The outer loop iterates over the rows of the packets and is executed in parallel using an OpenMP directive. Within the inner loop, `RayPackets` are created using the corresponding factory. Each packet is traversed through the grid using an instance of

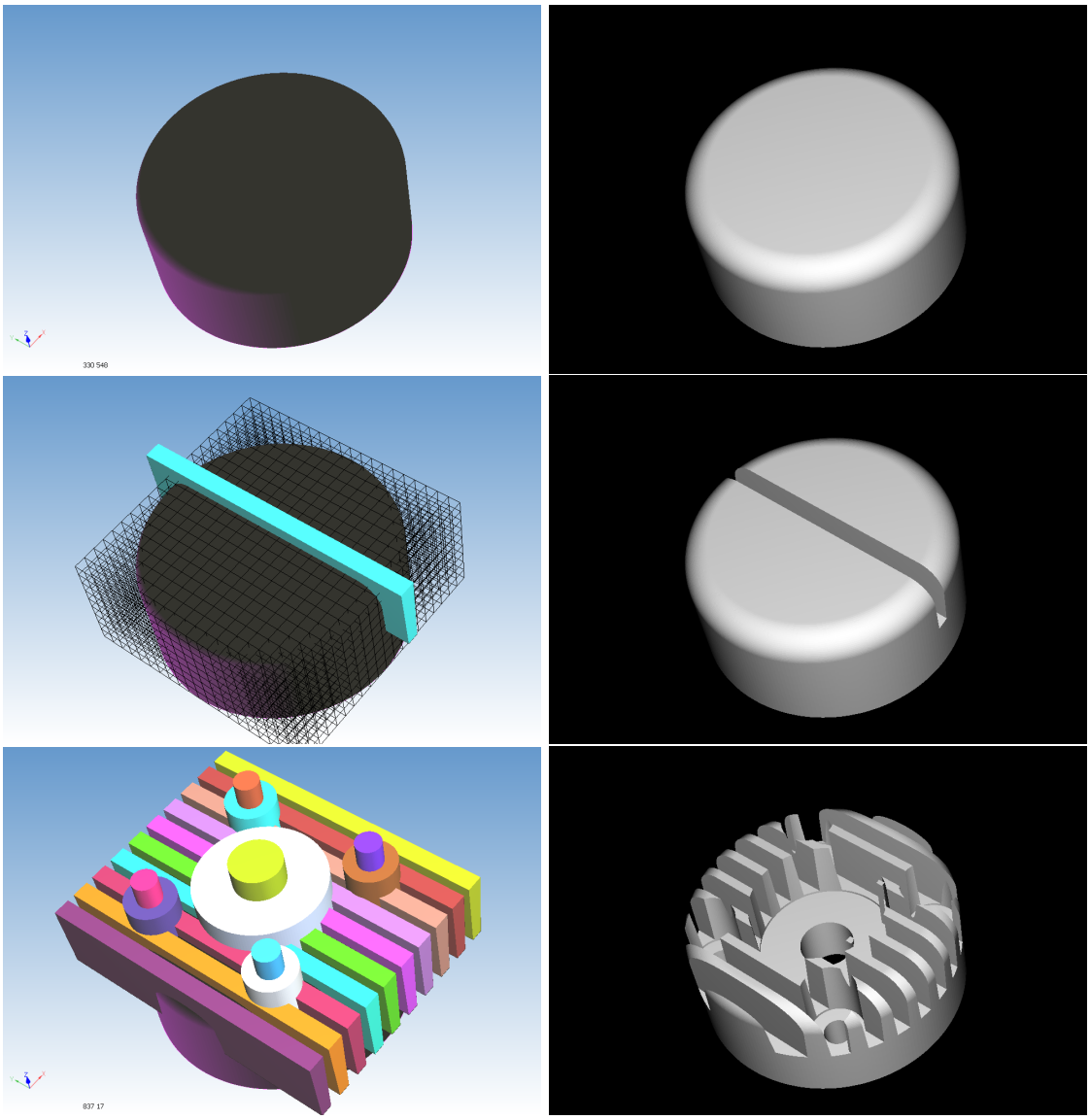

Figure 6: Screenshots of the existing prototype. The images on the left show the OpenGL visualizations of the loaded meshes. The images on the right show the ray casting result using the existing CPU packet ray caster. The first row only shows the loaded stock volume. The second row shows the stock volume and one subtraction volume. The OpenGL visualization also shows the grid. Note that the grid only has to cover the stock volume. The third row shows several subtraction volumes cut out of the stock forming a cylinder head.

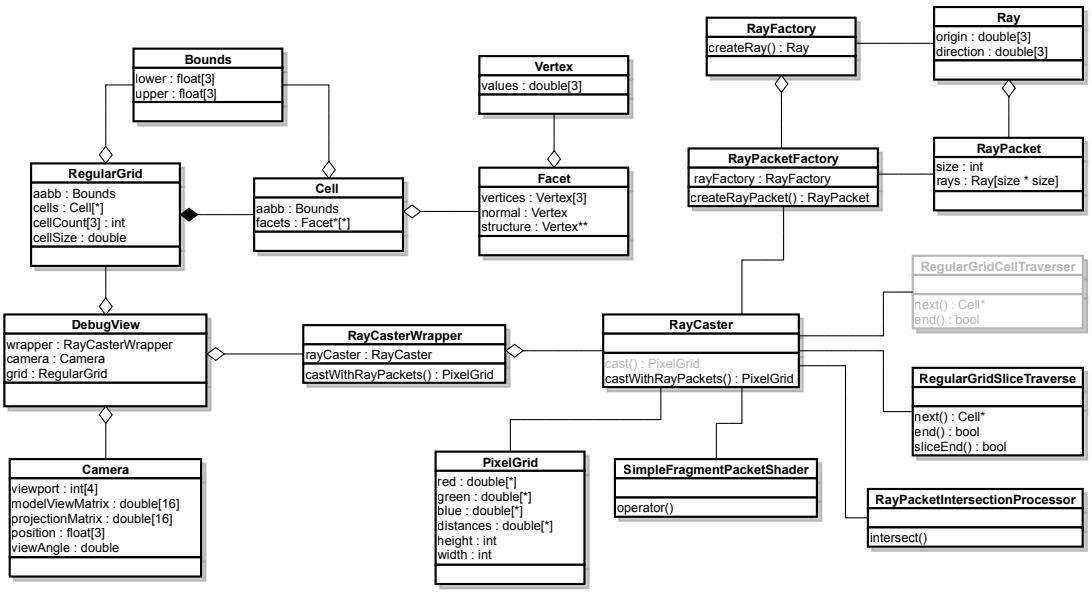

Figure 7: Simplified class diagram of the most important classes involved in ray casting at the beginning of the internship.

`RegularGridSliceTraverser`. After the slice traverser has been initialized with the packet, which determines traversal axis and grid entry point, a while loop retrieves cells from the slice traverser using its `next` method until `end` returns true. On every slice end, the traversal may be early aborted if all rays of the packet have already hit. For each cell the slice traverser returns, the ray packet has to be intersected with the referenced triangles of this cell. This is done using an instance of `RayPacketIntersectionProcessor`. This process consumes most of the required CPU time. Therefore the `intersect` routine and all subroutines are highly optimized and highly consist of AVX vector intrinsics. The intersection test for the packet is performed in parallel using SIMD after all triangles of the cell have been culled against the frustum defined by the corner rays of the packet. Details are discussed later in Chapter 3.4. For each intersection, the normal vector of the hit surface and the distance from the eye point are retrieved. The normal vector together with the initial ray direction of every ray of the packet is used by the `SimpleFragmentPacketShader` to calculate a color value for the ray. Currently, the scalar product of both vectors is used to measure a gray scale value, achieve simple shading. The distance of the intersection point to the eye point (camera position), from which the ray originated, is later translated into the normal distance of the intersection point to the image plane which corresponds to the depth value which would have been generated by OpenGL if the scene has been rendered traditionally. After all ray packets have been processed, the final `PixelGrid` instance containing the color and depth values is returned back to `DebugView` for display.

3.2 Ray casting data structure

The data structure maintaining the geometry and accelerating the ray casting algorithm is implemented by several classes. To start with, the `RegularGrid` class is a simple container for cells. It also stores some meta information such as the grids bounding box, the number of cells in each dimension and the size of a cell. The cells are stored in a continuous array. Therefore, 3-dimensional cell coordinates have to be mapped to the linear memory region.

The cells themselves are simple bounding boxes containing references (pointers) to the triangles (facets) contained within the cells. Although the bounding box is implicitly given by the cell's coordinates inside the grid and the grid's bounding box, the box is kept precalculated as it is often needed by the ray casting algorithm.

A `Facet` itself consists of its three vertices, a normal vector as well as the `structure` pointer. The latter references the mesh (subtraction volume) this facet belongs to. The importance of this value is elaborated later when discussing the details of the intersection routine in Chapter 3.4.

3.3 Classification

Every time a mesh is added to the grid, the triangles of the mesh have to be mapped to the cells of the grid. When the mapping is complete, the cells are classified into one of three categories. Cells which are occupied by triangles of the mesh surface are surface cells. Cells inside the mesh are inside cells and contain no triangles. Cells outside the mesh are outside cells and contain no triangles too. For the ray casting algorithm, only surface cells are relevant. The left sketch in Figure 8 illustrates the classification of a mesh.

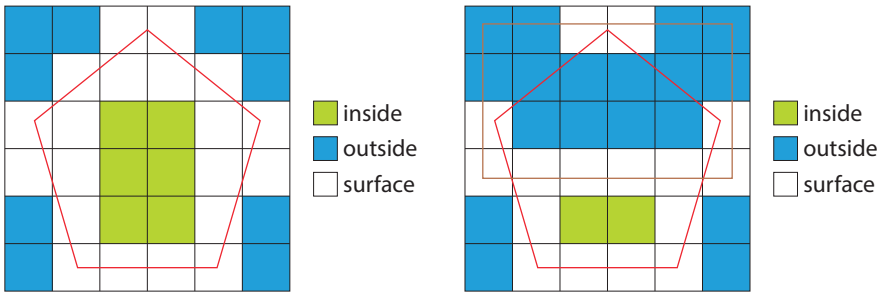

Figure 8: Principle of classifying cells of a grid according to the added mesh. Only surface cells are relevant for ray casting. The sketch on the left side shows the classified stock volume. On the right side the classification result, after a subtraction volume has been added, is shown.

When a new mesh is added to the grid, the mesh is again classified and merged into the existing cell classification. Limiting the modifiability of the scene by only allowing the addition of subtraction volumes simplifies the rules for merging a new mesh into the grid as shown in Table 1.

merge		Cell class of subtraction volume		
		outside	surface	inside
Cell class before	outside	outside	outside	outside
	surface	surface	surface	outside
	inside	inside	surface	outside

Table 1: Table of different merge scenarios and their outcome.

Cells which are outside the added subtraction volume remain unchanged. Surface cells of the added volume become surface cells except they where outside cells before, and cells inside a subtraction volume always become outside cells, as they lie outside the resulting geometry. The result of merging a subtraction volume into the grid is visualized in the right sketch in Figure 8.

As we can see, cells which where surface cells before and contained triangles have now changed to outside cells and are therefore no longer relevant for ray casting. As a result, the increase of triangles in the surface cells by adding new volumes has been compensated by exclusion of some surface cells. This reduction of potential triangles for intersection with an increasing number

of subtraction volumes is vital, as it enables scalability, allowing even scenes with thousands of subtraction volumes to be ray casted efficiently. In fact, some scenes can even be casted faster with an increasing number of subtraction volumes as the number of relevant surface cells and therefore triangles decreases.

However, this kind of reduction has a significant consequence on the ray casting algorithm. As subtraction volumes are divided into grid cells and some of them discarded, the volumes are no longer water-tight. This is a problem for entry/exit counting ray casting algorithms such as the one discussed in chapter 2.3. Therefore, an adapted version of this algorithm has to be developed, capable of handling open volumes. Chapter 3.4 discusses a method of getting around this problem.

3.4 Improved counting algorithm

The entry and exit counting ray casting algorithm for boolean subtraction volumes discussed in Chapter 2.3 maintains a counter value for each ray along the full traversal of the grid cells. However, by using the classification technique introduced in Chapter 3.3 to eliminate irrelevant triangles, it is no longer possible to keep states between multiple cells, as some of them might not contain relevant triangles for intersection but might be important for state changes. For example, an entry into a subtraction volume inside an outside cell. Consequently, all required states of the ray can only and must be determined upon entry of a surface cell.

This problem mainly concerns the counter of each ray used to count the volume entries and exits along the ray in order to find the surface hit. To put it differently, each ray has to know inside how many subtraction volumes the ray is at the point where it enters a surface cell. However, if we have another look at the classified grid in Figure 8, we can see that each surface cell contains triangles from all subtraction volumes, called structures in code, inside which the ray can potentially be depending on its entry point. If the ray would be inside a subtraction volume which has no triangles inside the cell, the cell would have been classified as an inside cell of this volume and the cell would be skipped during ray casting.

Determining the number of subtraction volumes the entry point lies inside of, which is called inside counter, can be done by creating secondary rays. These rays are casted from the cell entry point of the initial ray, which is called primary ray for distinction, to a reference point on one triangle of each subtraction volume. The secondary ray must not hit other triangles of the same volume between the entry point and the triangle reference point. The used reference points on the triangles have to be inside the cell's bounding box and should not lie on a edge of the triangle, as numerical instability increases at the edges. A good candidate for such points is the centroid of the triangle clipped against the cell's bounding box. The angle between the secondary ray and the surface normal of the chosen triangle determines if the entry point is inside the volume the triangle belongs to. If the primary ray already hits a subtraction volume, no secondary ray is necessary for this volume and the primary rays direction and the surface normal of the hit triangle can be used instead. Figure 9 shows the determination of the inside counter on an example cell.

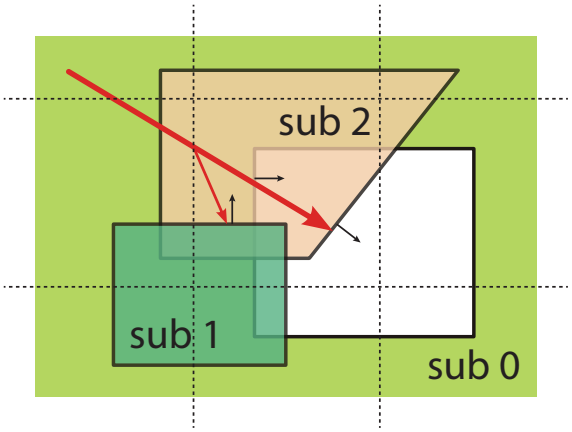

Figure 9: Determination of the inside counter upon cell entry. The primary ray hits the structures sub 0 and sub 2. Both surface normals enclose an angle with the ray's direction smaller than 90°. Therefore, the ray's entry point lies inside of sub 0 and sub 2. As the primary ray does not hit sub 1, a secondary ray is sent to a reference point on a triangle of sub 1. As the surface normal and the ray's direction enclose an angle larger than 90°, the cell entry point lies outside of sub 1. Therefore, the inside counter of the ray is initialized with minus two.

4 OpenCL ray casters

Before changing the existing prototype, a separate branch in SVN was created to allow experimenting with the existing code without disturbing the other members of the team. Before development was started, further tools were installed. As the GPU provided by RISC was from NVIDIA, NVIDIA's CUDA Toolkit (version 5.0) was installed which contains the required OpenCL header files and libraries. Furthermore, the OpenCL C++ wrapper provided by Khronos (cl.hpp) was downloaded from their website. Several weeks later, also Intel's SDK for OpenCL Applications 2013 was added, which provides a limited OpenCL debugger and some smaller extensions to Visual Studio [2]. A separate project was added to the existing Visual Studio solution and the necessary include and library paths configured in order to use OpenCL.

4.1 OpenCL ray casting driver

Creating a ray caster using OpenCL started by designing a small infrastructure which has to meet the following requirements:

- Only one class should be needed by `DebugView` (the main application class) in order to communicate with the OpenCL ray caster.
- As the OpenCL ray casters will be developed incrementally and several different ideas exists, a mechanism has to be designed to maintain multiple casters.
- A reload functionality has to the provided to reload an OpenCL caster at run time. As loading larger scenes may take up to some minutes and the OpenCL source code is read in and compiled at run time, this feature allows it to change the OpenCL source code of a caster at run time and reload it without having to relaunch the host application.
- Global options and important variables have to be exported to the main application where they can be adjusted using corresponding console commands. Changing the value of a variable may trigger a recompilation of the kernels or a reconfiguration of the OpenCL environment.
- If a ray caster crashes or OpenCL experiences any problems, the main application must be able to relaunch the complete OpenCL environment without having to be restarted.

Considering these requirements, a simple architecture has been designed shown in the simplified class diagram in Figure 10.

The central class and entry point of the OpenCL ray casting environment from the main application is the `OpenCLCasterWrapper` class. An instance of this class is maintained by `DebugView`, similar to the `RayCasterWrapper`, cf. Figure 7. The `OpenCLCasterWrapper` offers several methods for interacting with an OpenCL caster. Most importantly, it offers an `update` method taking a reference to the `RegularGrid` maintained by the main application which holds all relevant data of the scene. This method has to be called at least once before starting ray casting. The `raycast` method performs the main work of generating an image. The created image with depth information is also stored inside an object of `PixelGrid` and passed back to the main application.

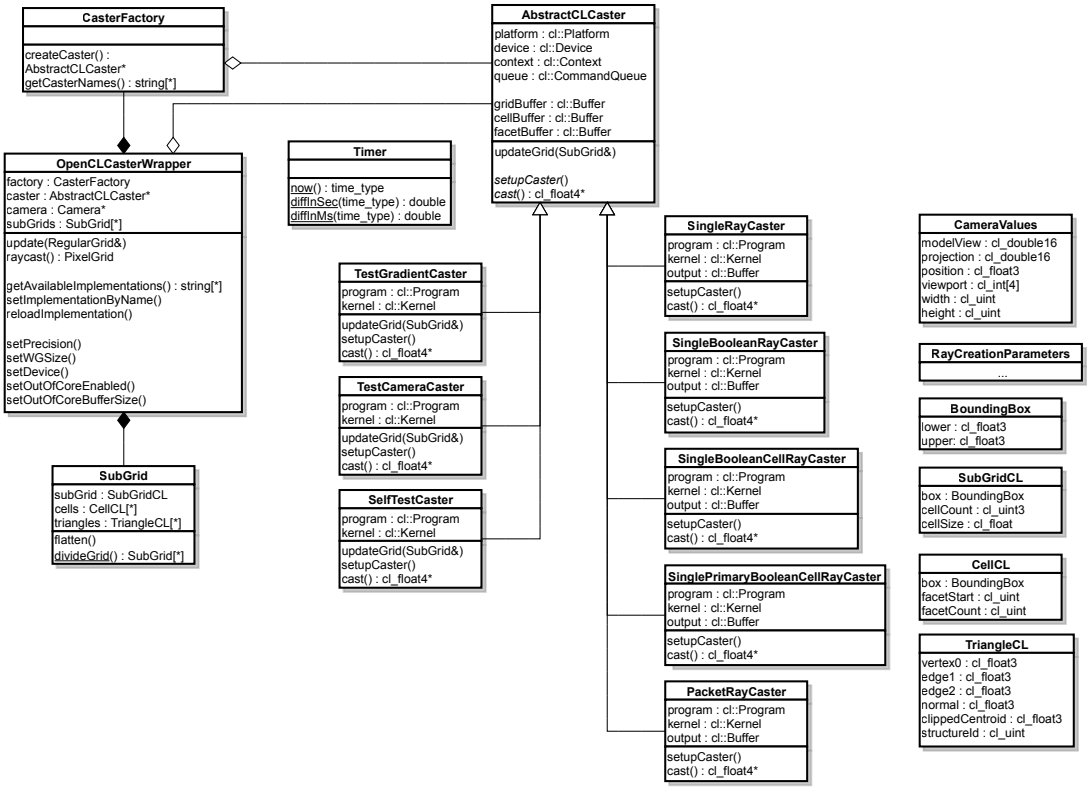

Figure 10: Simplified class diagram of the OpenCL ray casting driver.

For managing the underlying ray casting implementations, each caster is assigned a name. This assignment is managed by the `CasterFactory`, which is used by the `OpenCLCasterWrapper` to provide a list of available implementations and to allow selection and creation of casters by their name. The currently loaded caster is pointed to by the `caster` member. Furthermore, a reload method is provided for recreating the currently loaded caster.

Additionally, several options concerning OpenCL and the implemented ray casters may be set by the main application using the console. The `OpenCLCasterWrapper` offers further methods for configuring several values such as the floating point precision, work group size, OpenCL platform and device as well as enabling out of core casting and adjusting the out of core buffer size, cf. Chapter 4.13).

The `SubGrid` class is responsible for holding the full or a part of the grid and for transforming the grid's representation into buffers pushed to the GPU. Details are discussed with the caster implementations requiring this functionality.

The `Timer` class is a small helper class for measuring time durations between events. It is used to print the time required for certain operations inside the driver such as ray casting a frame, compiling a kernel or ray casting an out of core mesh in multiple passes.

Another important class is `AbstractCLCaster`. It is the base of all ray caster classes and is responsible for setting up the OpenCL environment. Every ray caster implementation inherits from this class and must implement the pure virtual functions `setupCaster` and `cast`. While `cast` is called directly from the `OpenCLCasterWrapper`, `setupCaster` is a template method inside `AbstractCLCaster` and is called during the casters construction, after OpenCL has been initialized. The derived caster implementation creates its kernels and further resources inside this method. The `updateGrid` method is implemented in the base class, as it works equally for all ray casting implementations which require the grid on the GPU (all non-test casters). Furthermore, `AbstractCLCaster` offers a variety of utility methods which can be used by derived casters.

On the right side of the class diagram of Figure 10 are several structures. `CameraValues` is used to transport relevant camera information from the `OpenCLCasterWrapper` into the caster implementations. This avoids dragging the `Camera` class and its dependencies deeper into the OpenCL project. The grid is also converted into one or multiple instances of `SubGrid` before it is passed into the caster implementations in order to reduce avoidable dependencies. The remaining structures are all used as data transfer objects between the OpenCL ray casters on the host application and the GPU. Therefore, only types defined in the corresponding OpenCL header are used, e.g. `cl_uint`, `cl_float3`.

4.2 TestGradientCaster

The first caster is a proof of concept to test the functionality of the OpenCL environment. It does not use the grid, loaded geometries or the camera's current values. The `TestGradientCaster` only loads the OpenCL kernel source code from a file and compiles it using corresponding helper methods from `AbstractCLCaster` inside the overridden `setupCaster` method. The `updateGrid` method is also overridden but empty, as no grid buffers have to be created and uploaded. Inside the `cast` method, an output buffer is created capable of holding a `cl_float4` value for storing red, green, blue and depth information for each pixel of the requested output image. The size of the output image is determined by the fields `width` and `height` of `CameraValues`, of which an instance is passed to the `cast` method. Then, the loaded kernel is enqueued on the command queue for execution on the GPU. The global work size of the kernel is only one dimensional and the number of output pixels rounded up to be a multiple of the maximum allowed work group size. Each kernel invocation then derives the x and y coordinate of the output pixels from the global id of the work item and writes a color value to the output buffer. The red channel of the color is the x coordinate divided by the output image's width, the green channel is the y coordinate divided by the height and the blue and depth channel are set to zero. After the kernel has executed, the result is read back to the host into an allocated `cl_float4` array, which is then returned by the `cast` method. The `OpenCLCasterWrapper` converts this array into an instance of `PixelGrid` which is then passed back to the main application for display.

The resulting gradient of the TestGradientCaster is shown in Figure 11.

Figure 11: Screenshot of the OpenCL kernel output of the TestGradientCaster.

4.3 TestCameraCaster

The next test caster is based on the `TestGradientCaster`. The goal of this caster is to test the generation of rays according to the current camera settings. Therefore, the `CameraValues` instance passed to the `cast` method is converted into a `RayCreationParameters` structure, which is passed as an argument to the kernel. This structure contains selected fields of the

model view and projection matrix as well as several precalculated values which are used to create the ray direction vectors for a given x and y coordinate on the image plane. The kernel itself stores the vertices of 12 triangles forming a simple cube placed at the coordinate space's origin in constant memory, which is a special part of the global memory. The kernel is enqueued with a two-dimensional global work size of the output image's width and height, both rounded up to be a multiple of the configured work group size. Each work item calculates the ray direction of the ray traveling through the pixel of the work group item's x and y coordinate on the image plane. With the origin at the camera position, this ray is intersected with the 12 triangles of the cube. The intersection routine used is an adapted version of the "Fast, Minimum Storage Ray/Triangle Intersection" approach presented by Tomas Möller and Ben Trumbore [10]. The intersections are depth sorted to find the correct intersection point and hit triangle. A color value for the pixel is then calculated using the scalar product of the ray's direction vector and the surface normal of the hit triangle. After the kernel has run, the resulting buffer is processed as by the TestGradientCaster.

The resulting image of a cube which rotates synchronously with the OpenGL visualization of the loaded scene is shown in Figure 12.

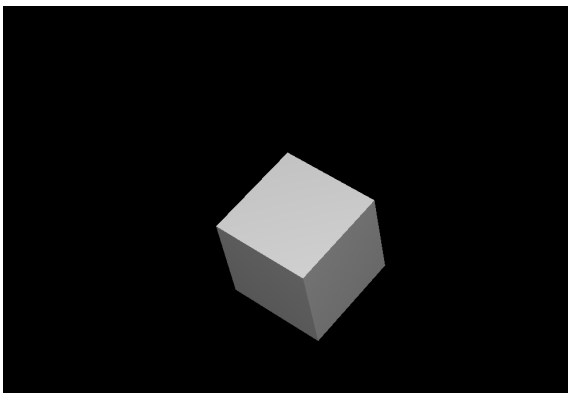

Figure 12: Screenshot of the OpenCL kernel output of the TestCameraCaster.

4.4 SelfTestCaster

The self test caster is a simple test of the structures used in host application and OpenCL interaction. More specifically, the SelfTestCaster calculates the size of all structures used in memory transfers between the host and the GPU on the host and enqueues a kernel retrieving the corresponding values on the device side. These values are compared to each other and an error is yielded if any pair of them is not equal. The reason why problems might occur on this level is the varying struct member alignment and padding carried out by different compilers. Although the data alignment of OpenCL is based on the one of ISO C99, some additions have to be made for OpenCL's built in vector types, cf. OpenCL specification 6.1.5 [11]. In particular, OpenCL has built in vector types which have to be aligned to a memory address being an even multiple of their size and a power of two. Therefore, a `float3` value for example, with a theoretical size of 12 bytes, is aligned to a 16 byte boundary. However, the corresponding `cl_float3` typedef on the host application (defined in cl.h) is backed by a structure containing a four element float array. This ensures that both types, `float3` inside OpenCL and `cl_float3` on the host application, have the same size of 16 bytes, but it does not guarantee a proper alignment as the OpenCL `float3` type will be 16 bytes aligned and the `cl_float3` structure (an array of floats) only to a four byte boundary. This has significant consequences if such vector types are used within structures as the size of the structs may be different (typically larger) inside an OpenCL kernel than within the host application. Consequently, enforcing proper alignment is partially up to the programmer and has to be declared for the affected structures (e.g. with `__declspec(align(sizeof(cl_float3)))` using Intel's C++ compiler). As the kernel does not create an output image, an empty buffer is allocated on the host and returned.

4.5 SingleRayCaster

The SingleRayCaster is the first attempt of a full ray caster. It is based on the TestCameraCaster considering the host code, the ray creation inside the work items and the triangle intersection routine. However, instead of using statically coded geometry inside the kernel, the grid containing the loaded geometries from the main application is used. Therefore, an appropriate memory layout has to be defined which fits well to the concept of OpenCL buffers. To traverse the grid, the 3D DDA algorithm presented in Chapter 2.2 is implemented. No counting is performed and no distinction of triangles from different structures and their facing (normal vector) is made.

As the data structure holding the grid on the host application is to some degree hierarchical as the grid uses a dynamically allocated array of cells and cells hold arrays of pointers to facets, the first step before any transfer to the GPU can happen is to flatten the data structure into corresponding OpenCL buffers. Figure 13 visualizes the hierarchical grid has been flattened into linear buffers by substituting pointers by indexes.

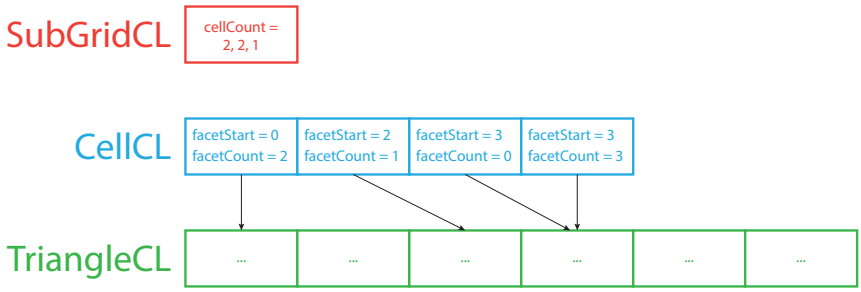

Figure 13: Layout of the OpenCL buffers used to store the flattened grid.

Three buffers are required to store the grid. The first one is rather small and only stores meta information about the grid. It contains a single instance of the `SubGridCL` struct which contains the `cellCount` member. This cell count holds the extents of the grid in all three dimensions and therefore determines the number of cells. The second buffer stores the cells as a large array of `CellCL` structures. Each of them may have multiple facets associated with them. Therefore, each cell has a `facetStart` index which points to the first of `facetCount` triangle belonging to this cell in the triangle buffer. The third buffer then holds a huge array of triangles which contains the geometric information of the scene.

Flattening the grid into these three buffers has to be done every time a surface cell changes, e.g. by loading a new subtraction volume. When an update of the grid occurs, the main application calls the `OpenCLCasterWrapper`'s `update` method and passes the reference to the new grid. The grid is then flattened by the `SubGrid` class, filling its members `subGrid`, `cells` and `triangles`, which are exactly the contents of the three required OpenCL buffers. The `SubGrid` instance is passed to the currently loaded caster by calling the `updateGrid` method. This method is implemented in `AbstractCLCaster` and creates the actual OpenCL buffer objects and transfers the content of the `SubGrid` to the GPU.

The kernel itself is again enqueued with a two-dimensional global work size of the size of the output image (one work item per pixel) rounded up to a multiple of the configured work group size. The kernel code for this first full ray casting approach starts in the same way as the TestCameraCaster. Each work item initially determines its ray direction. Then, the grid has to be traversed using the adapted 3D DDA algorithm based on John Amanatides and Andrew Woo presented in Chapter 2.2 [1]. This algorithm basically works by calculating the entry point of the ray into the grid as well as the corresponding cell of this entry point. Consecutively, increment values for each dimension are calculated used to determine the next cell from the current one. The collection of routines implementing this algorithm will be called cell traverser. After initialization, the ray casting kernel polls cells from this cell traverser until an intersection has been found or the grid

has been fully traversed. For each cell returned by the cell traverser, all triangles inside this cell, determined by the `facetStart` and `facetCount` members, are loaded from the triangle buffer and intersected using the same routine as in the TestCameraCaster. The intersection with the smallest distance to the ray origin (camera position) is selected, as triangles are stored in no particular order guaranteeing depth sorted iteration. Cells however are returned in correct order by the cell traverser. Therefore, the traversal is aborted if an intersection has been found inside a cell. The color of the output pixel is set like in the TestCameraCaster by calculating the scalar product between the ray direction and the hit surface's normal vector.

The output of the SingleRayCaster is shown in Figure 14. Although the generated image is far from being a correct visualization, one can clearly recognize the shape of the cylinder head used as ray casting input. For a correct output image cf. the existing CPU caster result in Figure 6.

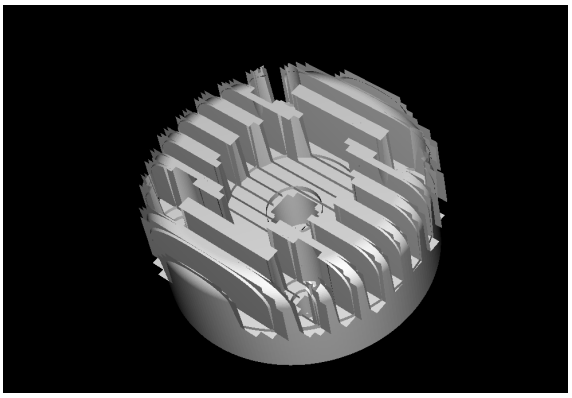

Figure 14: Screenshot of the OpenCL kernel output of the SingleRayCaster. The fringes sticking out of the cylinder head, especially visible at the top left, are due to hits on triangles of the subtraction volumes, as the nearest triangle hit, regardless of which volume, is taken for the pixel's color. These errors will disappear when cell based counting is implemented.

4.6 SingleBooleanRayCaster

The SingleBooleanRayCaster extends the previous SingleRayCaster by using a counter for each ray which counts the number of volume entries and exits. The counting is done along the full traversal axis as discussed in Chapter 2.3. As parts of the geometry are missing because only the triangles of surface cells are stored in the grid and transfered to the GPU, this caster will probably produce erroneous results. However, this implementation would work if the complete geometry was present in the grid. This caster can be seen as an evolutionary step between the already presented SingleRayCaster and the SingleBooleanCellRayCaster, which will be able to handle open geometries.

The SingleBooleanRayCaster class is equal to the one of the previous SingleRayCaster. Also the kernel code starts off equally by calculating the ray direction for each work item/pixel and traversing this ray through the grid. Besides declaring the counter and initializing it with minus one, the only code that has changed is the intersection routine of the ray with a cell polled by the cell traverser. The code still has to iterate over all triangles inside the cell and intersect the ray with each of them. However, instead of keeping the nearest intersection to the ray origin so far, we have to intermediately store all intersections of the ray with any triangle inside the cell. And here we hit a big problem in GPGPU computing with OpenCL, the lack of dynamic allocation.

Every kind of memory which is needed during a kernel execution has to be allocated beforehand. This accounts for both, global and local memory. Registers are handled a bit differently. Before a new work group is scheduled for execution on a streaming multiprocessor, all required registers for all work items of the work group have to be allocated in the register block on the SM. The number

of registers required for a work item must therefore be determined in advance by the compiler for all possible branches of the program, including all local variables, and is a decisive factor for how many work groups may be concurrently executed on a SM.

Let's consider the available options by an example. We take a common, small screen resolution of 1024×768 pixels. For each pixel a ray is created which needs a separate intersection buffer. As we do not know in advance how many intersections may maximally occur inside a cell, we have to define an allowed maximum. Let's say this value is a hundred. We actually had a long discussion about this in the office and thought about an even higher value, but a hundred proved to be sufficient in all used test scenes. Finally, we have to determine the size of the information required to hold an intermediate intersection for further processing. This must be at least the index of the hit facet, an unsigned integer, the distance of the intersection point from the ray's origin, a floating point value, and a boolean value, padded to four bytes, whether the ray enters the volume of the hit triangle or not. These three values make up twelve bytes. Let's calculate the total amount of memory required to hold a hundred intermediate intersection results per work item: $1024 * 768 * 100 * 12 = 943718400B = 900MiB$, which is a huge amount of memory for a GPU just for storing temporary data of an algorithm. Furthermore, this value does not include the grid with its cells and triangles. Consequently, storing the intermediate intersections in global memory is not acceptable.

Using local (shared) memory for storing the intermediate intersections per work item would also be possible. However, local memory is quite small with sizes of 16, 32 or 48 KiB per SM. As each work item requires 1.2 KiB ($12 * 100$) the largest local memory configuration would only allow 40 work items to run concurrently on a SM which is an extremely poor occupancy considering the high amount of potentially concurrently executed threads per SM, e.g. 2048 on a NVIDIA Kepler GPU [3, p.7].

The third option is using registers. Although 1.2 KiB per work item do not fit most GPU's hardware limits, e.g. NVIDIA Kepler limits a thread to 63 (GK104) or 255 (GK110) 32 bit registers [3], the kernel can still be launched. This shortage of registers is handled by compiler and driver by spilling the not available memory into global memory. As global memory is far slower accessed, register spilling usually causes a loss in performance. However, as the spilled area per work group is rather small, several tens or hundreds of kilobytes, and the area is always accessed by the same SM, this part of global memory benefits strongly from the L1 cache on the SM and may still achieve good performance on reads and writes. Paulius Micikevicius from NVIDIA held a detailed presentation about this subject in CUDA [9].

As a result of this small analysis, it has been determined that the intersection buffer will be kept as an array inside the kernel. It is then up to the compiler to try to fit the array into registers or spill it into global memory. This buffer is then used to store all intersections of the ray with triangles inside the current cell. Afterwards, the intersections inside the buffer have to be sorted according to their distance to the ray's origin as the counter has to be altered in the same order as the ray hits the triangles. OpenCL does not provide a sorting routine of any kind. Thus, a sorting algorithm has to be implemented. As the maximum size of the intersection buffer is defined as a hundred, a simple insertion sort will be used as it performs well on small arrays and has very little overhead considering additional variables. After the buffer has been sorted, all intersections are iterated again, this time in the order they ray hits them, and the counter is incremented or decremented according to the hit triangle's facing. If the counter reaches the value zero, the surface triangle has been found. The triangle id of the intermediately stored intersection is used to resolve the full triangle with normal vector and recalculate the exact intersection position. This information is then used to calculate the color value of the output pixel.

The output of the SingleBooleanRayCaster is shown in Figure 15. The image does not look much better than the one of the SingleRayCaster and is still incorrect as the grid contains open geometries leading to invalid counting. However, by implementing counting, an important step towards a working implementation has been made.

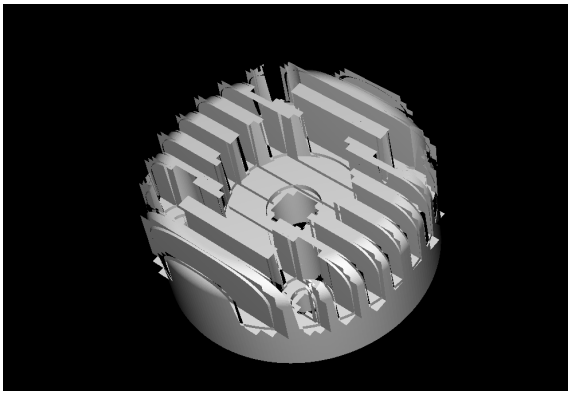

Figure 15: Screenshot of the OpenCL kernel output of the SingleBooleanRayCaster.

4.7 SingleBooleanCellRayCaster

The SingleBooleanCellRayCaster takes the previously described SingleBooleanRayCaster from Chapter 4.6 and moves the counter initialization from before the cell traversal into the cell itself as explained in Chapter 3.4. The main unknown is thus the number of subtraction volumes the ray, in particular its entry point, is inside of at its cell entry. Therefore, we have to decide for each subtraction volume if the entry point lies inside the volume or not. A simple solution is to take any triangle of a subtraction volume and create a secondary ray from the entry point to a point on this triangle, which has to lie inside the current cell. According to the enclosed angle of the triangle's normal and the created secondary ray, it can be decided if the entry point is inside or outside of the subtraction volume the triangle belongs to. However, as multiple triangles of the same volume may be potentially hit by the secondary ray, it has to be ensured, that the secondary ray is not intersected by another triangle of the same volume. This case can lead to wrong decisions, as e.g. the secondary ray may select a triangle of the back size of volume and also intersect the front. The resulting initial counter value is initialized with zero and decremented for each subtraction volume the entry point is inside off.

The concrete implementation of this approach makes use of additional data provided in the `TriangleCL` structure and is further based on an assumption guaranteed by the grid creation code of the main application. The `clippedCentroid` member of the triangle structure, cf. class diagram in Figure 10, contains the centroid of the resulted polygon after the corresponding triangle has been clipped to its cell's bounding box. This value is calculated by the `SubGrid` class when the grid of the main application is flattened into the OpenCL buffers. Furthermore, each triangle has a `structureId` member which is an identifier being equal among all triangles inside a cell belonging to the same subtraction volume (structure). It is additionally guaranteed, that triangles belonging to the same structure are stored continuously in the triangle buffer. Based on these circumstances the counter initialization algorithm is implemented as follows.

Inside each work item, for each cell returned by the cell traverser, all triangles of the cell are iterated. Additional variables keep track of the current structure id, the current distance to the nearest triangle of the current structure and its normal. In each loop where the structure changes, a secondary ray is created from the primary ray's entry to clipped centroid of the current triangle of the loop which has caused the structure change. The distance from the entry point to the clipped centroid as well as this triangle's normal is kept. All following triangles of the same structure are intersected with the created secondary ray. In case of an intersection, the distance to the nearest intersection of the secondary ray is updated as well as the normal by the one of the new, intersected triangle. Before the next structure change, the normal of the nearest triangle intersecting the created secondary ray can be used to determine if the entry point lies inside or outside of the current structure. If it is inside, the counter is decremented. The triangle iterating continues with the next structure analogously. After all triangles of the cell have been iterated, the initial counter value holds the number of subtraction volumes the primary ray's entry point is

inside of. The algorithm can now continue equally as the SingleBooleanRayCaster by intersecting all triangles, sorting them by their depth and processing the counter along the ray to find the surface hit.

The SingleBooleanCellRayCaster is the first ray casting approach able to deliver correct results. The output image of the kernel is shown in Figure 16. Apart from a few wrong pixels which occur because of numeric instability of the used float data type), the result already makes a satisfying impression. The errors do not occur on the CPU caster which uses double precision. Considering the code however, there is still room for improvement as the kernel iterates all triangles of the cell twice. One time to determine the inside counter and one time for the intersections. Furthermore, the few invalid pixels may be eliminated if double precision would be used for the intersection routines on the GPU too. These two subjects will be addressed in subsequent chapters.

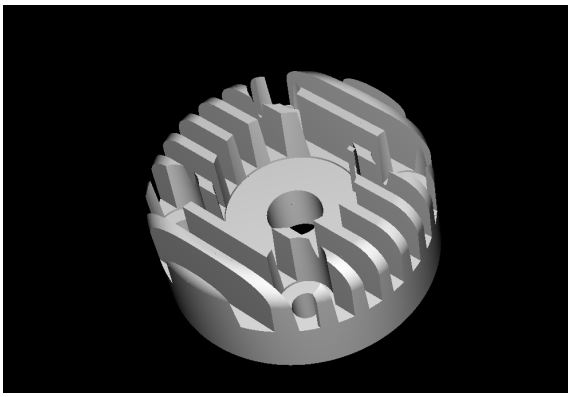

Figure 16: Screenshot of the OpenCL kernel output of the SingleBooleanCellRayCaster.

4.8 Double support in calculations

Although the `double` data type is very commonly seen in conventional languages like C++, Java or C#, only recent GPUs support double precision floating point arithmetic. The reason for this is that the higher precision on the cost of speed did not fit the initial requirements of a graphics processing unit, which was optimized for maximum throughput. With the gaining popularity of GPGPU computing, hardware vendors also integrate double precision support within their hardware. The NVIDIA Kepler architecture for example provides special double precision units in addition to their normal cores [3, p.8]. Some sensitive values of the `RayCreationParameters` make already use of double precision types, as the program is expected to be run on newer hardware. However, most of the calculations inside the kernel are done using simple floating point arithmetic.

The anticipated benefit of using double precision for grid traversal and intersection routines is to reduce the number of erroneous pixels. These pixels are most often the result of rays which shoot between adjacent triangles or angles between secondary rays and normals of almost exactly 90°, where the scalar product is a tiny positive or negative number. As OpenCL 1.2 is still very close to C99, no language features exist which would allow writing generic kernels. AMD has already proposed their OpenCL C++ static kernel language for standardization which would allow the use of classes, inheritance and especially templates, but it is not yet officially part of OpenCL and currently only supported on AMD systems. Therefore, the OpenCL preprocessor will be used together with a compiler argument to set the desired type. All floating point types involved in calculations inside the OpenCL code will be replaced by a corresponding token, e.g. `T`, which is common in C++ templates, and defined when the OpenCL compiler is invoked, e.g. using `-DT=double`. Furthermore, also derived types must be defined to allow switching vector types, like `T3` for `double3`.

This approach works well in practice. By recompiling the OpenCL kernel at run time, the used precision can even be changed on the fly without having to relaunch the main application. A corresponding console command has been implemented.

The SingleBooleanCellRayCaster with double precision already gets grid of most wrong pixels. However, the runtime of the kernel increased to over twice the amount of the single precision variant.

To also support older GPUs without double precision an additional preprocessor option was introduced which also affects the main C++ application. It allows forcing the floating point precision to single. This also overrides the double values of the transfer structures.

4.9 OpenCL source embedding

One of the big strengths of OpenCL is its portability across many different hardware platforms. This big advantage is achieved by delaying the compilation of the OpenCL source code until an application's actual execution on the target platform. However, this concept usually forces programmers which put their OpenCL code in dedicated files to ship parts of their source code with the compiled application. This scenario is undesired or even prohibited in commercial software development where the customer should not gain access to a product's source code. Furthermore, keeping the OpenCL source files in the correct place for the application to find them at run time requires additional deployment overhead. Fortunately, both problems can be solved by embedding, and optionally even encrypting, the OpenCL source code into the actual program. Although several easy solutions exist on different operating systems or technologies, like resource files on windows, no cross platform or C++ standard compliant method for storing an applications resources is available.

However, an OpenCL source file basically contains text and text can be statically stored inside an application using a string literal, which is an array of characters. It is therefore possible to store the required OpenCL code inside a large character array which is compiled into the executable. Fortunately, the corresponding OpenCL API call for creating an OpenCL program from source does not take the name of a file as argument but the actual source code. Therefore, the embedded character array can be directly passed to OpenCL.

Nevertheless, editing source code in a separate file is more comfortable for the programmer and better handled by IDEs concerning syntax highlighting and debugging than writing OpenCL code directly as string literals into the existing C++ code. Therefore, a separate conversion tool has been developed which reads in a given input file and outputs a C++ header file with a character array containing the content of the input file. This header can than be included and used by the existing C++ application. Modern build systems like MSBuild and Visual Studio also support the configuration of custom build steps, which allows the creation of includable headers from the OpenCL source files as part of the build process in Visual Studio. Visual Studio therefore calls the external conversion tool for every OpenCL source file which is part of the project before it starts compiling the C++ files.

This solution works for simple OpenCL source files. However, the preprocessor's `#include` directive allows an OpenCL source file to include other source files. If a source code containing includes was embedded directly, the application would fail to include the referenced files at run time after deployment. Therefore, a little bit more logic has to be added to the external conversion tool. Specifically, the OpenCL source code has to be scanned for `#include` directives and they have to be recursively resolved, as included files can include further files, before the text can be converted into the output character array. Furthermore, as this process drops the original file names and line numbers, proper `#line` macros have to be inserted to allow the compiler to report errors on their correct origin. Although this is basically a small implementation of a part of a preprocessor, an existing one could not be used, as conditional compilation should still be possible after the conversion by keeping `#ifdef` and similar directives.

4.10 SinglePrimaryBooleanCellRayCaster

The last single ray caster implementation takes the previous one, the SingleBooleanCellRayCaster from Chapter 4.7, and merges the two loops which iterate over the triangles of the current cell into one. Furthermore, secondary rays are only cast when the primary ray does not hit a structure.

The host code for this caster implementation is exactly the same as of the SingleBooleanCellRayCaster. The kernel starts equally by computing the ray direction and starting the grid traversal using the cell traverser. For each cell the initial counter value has to be calculated as well as all intersections. These two steps were executed after each other in the previous implementation and are merged now. Before iterating over the triangles of the current cell, the intersection buffer is declared together with a variable holding the current structure id, the start triangle index of the current structure as well as normal vector and distance to the nearest intersection of the current structure. Then, the algorithm starts to poll the triangles of the current cell from global memory. The `structureId` member is again monitored inside the loop over the triangles. On a structure change and on the first loop iteration, we save the index of the current triangle as start index of the new structure. During the triangles of this structure the primary ray is intersected with each triangle. If an intersection is found, the intersection is added to the intersection buffer and the distance from the primary ray's origin to the intersection point as well as the hit triangle's normal vector is stored. Before the next structure change and after the last triangle of a structure, the algorithm checks if the primary ray has intersected a triangle of the current structure. If this is the case we can determine if the entry point lies inside the current structure by using the normal vector of the hit triangle and the primary ray's direction. If the primary ray did not hit any triangle of the structure, we have to use secondary rays. In this case, the triangles of the current structure are iterated again. The start is easily found again as we have stored the start triangle index on each structure change for this reason. A secondary ray is created to the clipped centroid of the first triangle and all subsequent triangles of the same structure are intersected with the secondary ray. The nearest hit of the secondary ray is then used to determine if the primary ray is inside this structure or not, equally as in the previous implementation for all structures. After all triangles have been iterated, we have the buffer of intersections as well as the initial inside counter value and can continue with depth sorting the intersections and processing the counter along them to find the surface hit.

This approach has two advantages over the previous one. Firstly, each triangle of a cell is only requested once from global memory instead of two times which should improve performance. Secondly, using the primary ray for determining if the entry is inside a volume or not produces more stable results as many of the erroneous pixels have been disappeared. This is the final single ray casting variant which is also used as the main application's default ray casting routine. The output of the ray caster is shown in Figure 17.

Further optimization of this implementation is surely possible but would require the help of advanced tools like a GPU profiler. However, NVIDIA dropped OpenCL support with CUDA Toolkit 5.0, which was used during the internship. This decision was not officially announced but the topic is discussed in several internet forums [13].

4.11 PacketRayCaster

The PacketRayCaster approaches differently than the single ray variants and is primarily motivated by the huge performance boost achieved by the CPU implementation compared with the previous single ray CPU version. The general performance consideration is to keep a group of adjacent rays as coherent as possible to benefit from better vectorization and caching behavior. Furthermore, traversing whole packages instead of each single ray reduces the overall control logic.

Although the PacketRayCaster shares some code with the final SinglePrimaryBooleanCellRayCaster, a lot of changes are necessary to the kernel code. Fortunately, the host code does only

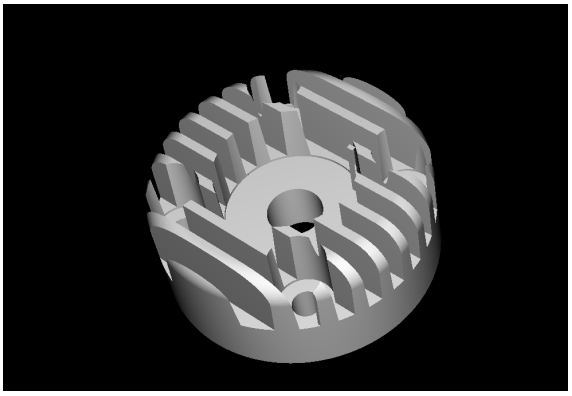

Figure 17: Screenshot of the OpenCL kernel output of the SinglePrimaryBooleanCellRayCaster.

require a little adjustment, which is to define a packet width and height which is used as the kernels local work size. The remaining host code stays the same, meaning the kernel is again enqueued using a two-dimensional range with the size of the output image where each dimension is rounded up to the next multiple of the defined packet width or height.

The kernel code then begins again by determining the ray direction for each work item using the `RayCreationParameters`. Afterwards, the four corner rays are marked by declaring an integer for each work item which is initialized to minus one and only set to a valid index, from zero to three, by the four work items in the corners of the work group, identified using their local ids. This integer can then be easily used to detect if the current work item is a corner ray or not. Furthermore, the four corner ray work items write their ray directions into a shared array in `__local` memory.

Now the traverser algorithm can be set up. The packets will be navigated through the grid slice by slice as described in Chapter 2.4. However, a traversal is only necessary if the ray packet actually hits the grid. This check is performed by trying to intersect the outer square of rays of the packet, called peripheral rays, with the bounding box of the grid. If any of them hits, a traversal is necessary. Therefore, a main traversal axis has to be determined which is done by selecting the axis where the corresponding component of the central ray's direction vector has the largest magnitude. All corner rays are then intersected with the front plane of the first slice of the grid. The main traversal axis as well as the front plane intersection points are stored in local memory. The slice traverser is now initialized.

The kernel then continues with the traversal loop, which polls cells from the slice traverser. The slice traverser keeps state information about the current slice and the current cell's coordinates inside the current slice. Initially, when the first cell is requested, the slice traverser intersects the four corner rays with the black plane of the current first slice. Using the maximum and minimum values of the intersection points on the front and back plane of the slice of the non-traversal axes, the rectangle on the slice which has been hit by the packet can be determined. This rectangle is extended to full cell boundaries and the slice traverser can return the first cell of this range. Subsequent cell requests can be satisfied returning further cells of this range until all cells of the hit slice region have been iterated. Then the slice traverser has to move on to the next slice by intersecting the corner rays with the back plane of the next slice Using the intersection points of the last slice's back plane as intersection points of the now current slice's front plane, the affected cell region can again be determined. This process is repeated until all cells of the last slice have been returned or the traversal is aborted.

For each cell returned by the slice traverser, the packet has to be intersected with the triangles inside the cell. However, before all rays inside the packet are intersected with all triangles, several optimizations are done. Each corner ray work item creates a plane using the next clockwisely adjacent corner ray direction from shared memory. The result are four planes bounding the frustum of the ray packet which allows culling of the cell's triangles against it. However, this

frustum culling step can only be done by one thread of the packet which has to actually hit the cell. A cell hit by the ray packet does not necessarily need to be hit by all rays of the packet.

The culling is done by iterating over all triangles of the current cell. Each triangle's vertices are then checked against all frustum planes. If all vertices lie outside of one plane, the triangle is culled. This approach does not cull all possible triangles, but it is fast and easy to implement as it does not require any intersection tests. The indexes of all triangles which are not culled are accumulated in a buffer in local memory, and therefore visible to all work items afterwards. However, the culling routine has to keep an eye on the centroids of the unculled triangles. As triangles of a structure lying outside the packet's frustum are culled, a secondary ray targeting a centroid outside the frustum of an unculled triangle may lead to wrong results, as a potentially, with the secondary ray intersecting triangle was removed during culling. Therefore, the culling routine has to ensure, that the first triangle of each structure provides a valid centroid lying inside the packets frustum. A further optimization is also possible during the culling routine. By also monitoring the structure changes we can determine the case in which all triangles of a structure are culled. If this is the case, a secondary ray can be used to determine if the frustum lies inside this culled structure which allows to skip the current cell entirely.

After the culling step completed, a barrier operation is executed to ensure that the culling thread has finished writing the indexes of the unculled triangles into local memory before any work item starts to read them. After this synchronization point, each work item can then proceed with the intersection routine taking the array of unculled triangle indexes in shared memory. The intersection is done equally as in the SinglePrimaryBooleanCellRayCaster, by iterating all triangles and intermediately storing the intersections in a buffer. The initial value of the inside counter is also determined in this step. Secondary rays are used if the primary ray does not intersect a subtraction volume. The intersections are then sorted again and the counter is processed along them to find the surface hit.

Although the packet ray caster produces visually equivalent results as the SinglePrimaryBoolean-CellRayCaster, cf. Figure 18, it is despite the optimizations slower than the single ray variant. The reasons are probably the large amount of synchronization points (barriers) encountered during the kernel execution, which are needed every time a shared state is written, which happens a lot inside the slice traverser. However, a detailed analysis requires the use of professional tools like profilers, which were not used during the internship, cf. end of Chapter 4.10 for the reason. Another, yet unsolved problem is that the packet ray caster crashes occasionally, especially when enqueued multiple times with short delay or when being compiled with double precision.

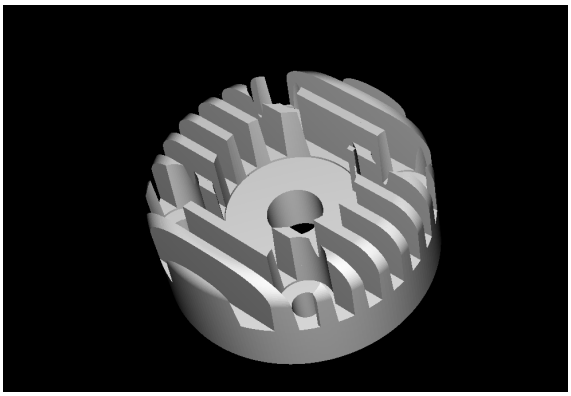

Figure 18: Screenshot of the OpenCL kernel output of the PacketRayCaster.

4.12 Migration to new host application

While the main development of Enlight happened using the code base of an existing, RISC internal framework designed for easy OpenGL integration and scene management, Michael Hava, one of the project members, was already working on a newer, leaner main application which would then also be used for the final product. The main design consideration of this new application was to get rid of legacy code, like the OpenGL framework and MFC, and to separate the application into components using a self-written C++ component framework by Michael Hava called CWC. Furthermore, by using the cross-platform GLFW library for window management and user interaction, the application may be run on all major platforms.

Although the complete main application was rewritten, the final port of the OpenCL caster suite to this new prototype did only require an adaption of the `OpenCLCasterWrapper` class. Of main concern were the `RegularGrid` class and the `PixelGrid` class, which where both replaced by other structures. Updates to the OpenCL driver now occur separately for each cell and may be executed concurrently by multiple threads. Therefore, thread safety of the `OpenCLCasterWrapper` 's methods had to be ensured which was implemented using C++11's new thread library. Furthermore, the concept of configurable driver options was generalized to key value pairs settable by the main application. Examples include floating point precision, currently used caster implementation, work group size, OpenCL platform or device and out of core casting. The final and largest change was also the output format of the casters, which was changed from an image with red, green, blue and depth channel to the hit triangle id and distance as well as the barycentric coordinates of the intersection point on the triangle. The actual color image is then calculated by the main application, separating the shading step from the ray casting algorithms.

4.13 Out-of-core ray casting

The final implemented feature is optional out-of-core ray casting, which affects all ray casters. The basic concept of out-of-core algorithms is to allow the processing of data structures which are larger than the available system memory. Concerning the GPU ray casting implementation, scenes should be supported where the grid representation is larger than the available memory on the graphics card. The available main memory is still a limit for the scene's size.

When designing an out-of-core approach for the grid based GPU ray casters, we are limited by the design of OpenCL. OpenCL requires, that all global memory objects, like the three buffers where the grid, cells and triangles are stored, have to be created and fully initialized before a kernel is executed. Furthermore, these objects cannot be changed during the execution of a kernel. Additionally, communication between the host application and a running kernel is one-way. No way exists for a kernel to communicate back to the host application. These constraints limit the design of an out-of-core caster as such, that all required information for a kernel has to be set up before the kernel is enqueued and the kernel cannot request further data during it's execution. Therefore, the grid has to be divided into smaller sub grids which have to be ray casted individually and merged into a final output image. Fortunately, dividing the grid causes no problems as the ray casting algorithms already operate on a cell basis. Also, merging the output images should inflict no problems, as every pixel contains depth information.

Out of core casting can be enabled on the console of the main application and an optional out of core buffer size may be set for testing purposes. If this value is not specified, it is set to the maximum available buffer size. When the main application passes the grid to the `OpenCLCasterWrapper` by calling the `update` method, the wrapper checks if out of core casting is enabled. If this is the case, it passes the grid to the static `SubGrid::divideGrid` method, which recursively splits the grid in half, changing the split axis on each recursion, until each sub grid requires fewer memory than the current out of core buffer size. These sub grids are then flattened and stored in the `subGrids` member of the wrapper.

When the scene is ray casted, the sub grids are depth sorted depending on the current camera position. A loop than iterates over the sub grids starting at the nearest. The sub grid is transfered to the GPU and ray casted. Each work item of a subsequent ray cast, the second to the last one, then checks the buffer containing the output image if the corresponding pixel has already been written to. If this is the case, the work item can exit. This approach ensures that no pixels are ray casted twice. Therefore, apart from the memory transfers of the sub grids which have to be done for every ray casted image, no additional overhead is produced.

5 Results

This chapter covers the final results of the ray casting development and provided appropriate benchmarks. Furthermore, experiences made during development, especially concerning the available tools, are shared. Finally, existing and unsolved problems are discussed.

5.1 Benchmarks and comparison with existing casters

The algorithms developed during the internship were finally tested on a few larger scenes which come closer to real world applications of Enlight. The cylinder head used throughout this thesis, cf. for example Figure 6, is often used for quick demonstration or debugging, as it is a rather small scene which can be loaded in roughly a second. Besides the cylinder head, a lot of other scenes, more or less fitting the purpose of Enlight, are available and can be loaded within the main application. From these scenes, two more have been chosen which show interesting behavior. The first additional scene is a Menger sponge, which is a fractal geometry. It is easily created using a programming language. Due to its recursive nature, Menger sponges provide an easy way of generating complex meshes just by adjusting a parameter of the generation script. In the following benchmarks a level five member sponge will be used. The second additional scene for ray casting will be a multi impeller, which is a technical component of many centrifugal pumps. Figure 19 shows screen shots of the addition scenes.

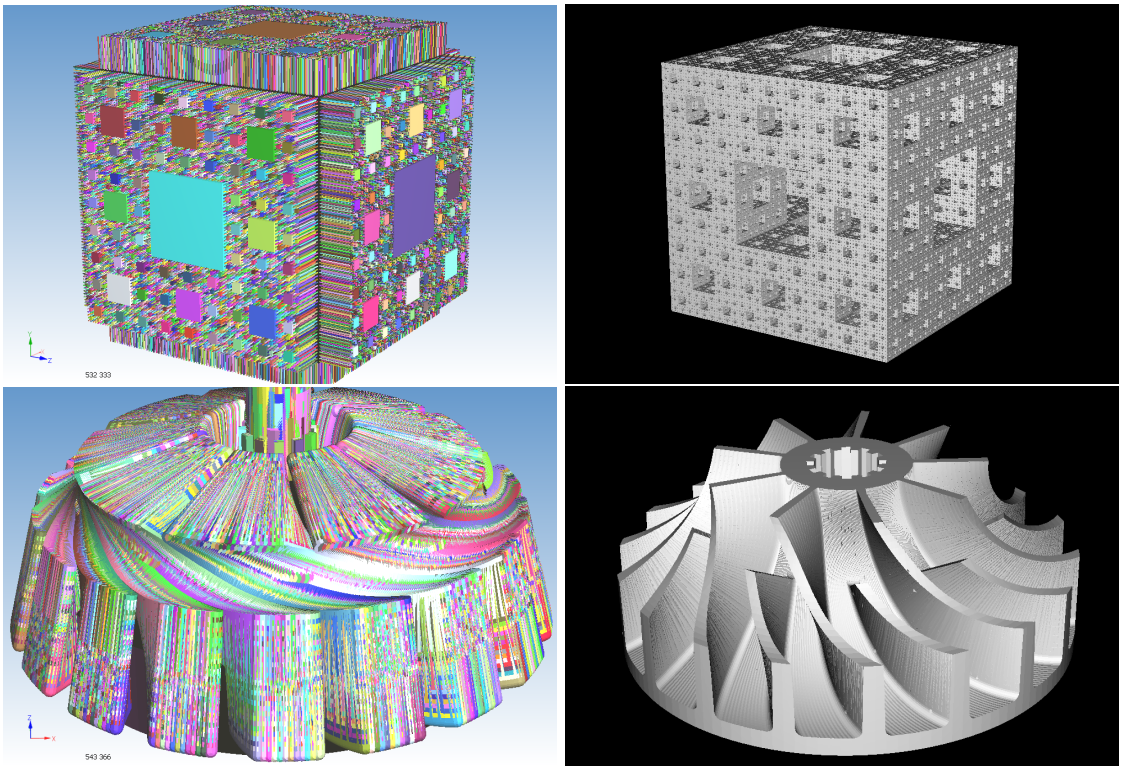

Figure 19: Screenshots of the OpenGL visualization and the OpenCL kernel output of the SinglePrimaryBooleanCellRayCaster of a level 5 Menger sponge and a multi impeller.

Scene	Vol.	Unique tri	Cell tri	Grid dim	Outside	Surface	Inside
Cylinder head	21	28000	63604	40 40 18	13392	8744	6664
Menger sponge 5	14044	168528	9582636	100 100 100	498480	501520	0
Multi impeller	4726	12510260	4010800	150 150 61	928405	139271	304824

Table 2: Geometric information about the used test scenes.

Table 2 shows various geometric information about the three scenes used for benchmarking. The cylinder head is rather small with only 21 subtraction volumes and 28,000 triangles. The Menger sponge level five has been chosen for its large number of subtraction volumes. A level six Menger sponge was also tested once, which is the first scene to exceed the 100,000 subtraction volume mark with roughly 110,000 cuboids. However, it consumes the full available system memory and forces the operating system to swap. As a consequence, the classification of the scene takes several hours to complete in which the used work station cannot be used for anything else. Therefore, the level 5 variant has been chosen. A further interesting behavior is that although the initial geometry of the Menger sponge only consists of around 0.17 million triangles, this number multiplies by almost 57 to 9.58 million when the triangles are assigned to the cells they intersect. This is one of the large drawbacks of using a grid as acceleration structure. The multi impeller provides a real world example of subtractive manufacturing. It mainly impresses by its large triangle count of 12,5 million. Furthermore, the multi impeller profits tremendously by the cell classification as only 11% of all cells are surface cells. The smaller number of cell triangles when compared with the original, unique ones is explained by the many subtraction volumes which partly stick out of the grid.

Each scene is loaded and then ray casted ten times by each chosen ray casting implementation using an output resolution of 1024×768 pixels. The average run time is shown in Table 3. All times are in milliseconds.

Caster	Precision	Cylinder head	Menger sponge 5	Multi impeller
SingleBooleanCell	single	25.7	40.7	114
	double	72.5	159.9	653.7
SinglePrimaryBooleanCell	single	25.6	41.6	123.2
	double	82.5	258.5	955.7
Packet	single	67.9	715.4	914.1
CPU AVX	double	44.9	165.8	316
Update time		130	11071	10455
Transfer time		4	177	334

Table 3: Run time of several casters, complete update time and memory transfer time on different test scenes. All times are in milliseconds.

The first thing one can notice on the benchmark results is that the single precision variant of the final SinglePrimaryBooleanCell caster is faster in all scenarios than the CPU implementation and produces visually equivalent results than the slower CPU variant which runs in double precision. Furthermore, the cylinder head and the level five Menger sponge can both be casted interactively with 39 and 24 frames per second (FPS). The multi impeller is more calculation intensive only achieving around 8 FPS. Nonetheless, these results are very delighting as they prove that scenes of this complexity can still be ray casted in reasonable time. The SingleBooleanCell caster is surprisingly a little bit faster on larger scenes. However, this small advance is compensated by more erroneous pixels. The packet ray caster is far behind all other ray casting approaches which mostly due to the high amount of synchronization points inside the kernel. Furthermore, switching from single to double precision causes a serious performance loss in all tested kernels. Although a run time decrease by a factor of two to three has been expected, as the Kepler architecture has only a third of double precision units than normal cores, the performance drop was even worse on larger scenes. Nevertheless, single precision has proven to deliver good and stable results in most cases.

Apart from the casting times, one should also notice the time required for an update of the OpenCL buffers. Although the memory transfer times seem reasonable, the time required for the complete update is enormous. The most time consuming operations during the update is the flattening process and the calculation of the clipped centroids. The latter could be avoided, if the centroids were precalculated when the triangles are added to the grid.

5.2 Development

During the internship various software tools have been used for developing OpenCL applications. Some of the experiences made might be of value for future developers and are therefore shared here:

NVIDIA Visual Profiler
This separate tool from NVIDIA is shipped with their CUDA SDK and provides a standalone debugger and profiler for GPGPU applications. Although mainly developed for aiding CUDA developers, it should support OpenCL until CUDA SDK version 4.1. However, this version has been tried without success.

NVIDIA Nsight
Nsight is NVIDIA's second GPGPU debugging and profiling tool and requires a free registered developer account before it can be downloaded. It provides excellent Visual Studio integration and does support OpenCL according to NVIDIA's website. However, Nsight requires the presence of at least two GPUs, where one is used for debugging kernels step by step and the second one for handling the display while the first one is busy. Nsight also allows remote debugging.

Intel SDK for OpenCL Applications
Intel also supports OpenCL for their CPUs and on chip graphic controllers. By installing Intel's SDK for OpenCL Applications, one can install additional Visual Studio plugins which allow to create dedicated OpenCL projects. Syntax highlighting for OpenCL source files is configured and a kernel file can be compiled offline using a new context menu entry to check for compilation errors without having to start the actual application. Furthermore, a simple debugger is integrated which allows to debug a single work item which has to be configured before the main application is started. However, this debugger is limited to OpenCL programs on the CPU and cannot be used in the case of crashes as other work items are processed in parallel to the debugging session of the selected one. Also break points do not work when set in files included by a kernel. In general, the tool makes a still immature expression although being useful.

gDEBugger
Graphic Remedy's gDEBugger is a standalone debugger, profiler and memory analyzer for OpenGL and OpenCL applications. It is additionally independent of the used hardware platform and freely available. The tool shows basic information for a process it is attached to like memory consumption of various objects inside open OpenGL or OpenCL contexts. It also offers a time line showing enqueued OpenCL operations like enqueued kernels or memory transfers and highlights their start time and duration. This allows a developer to optimize a device's occupancy. Although the possibility of creating break points at certain API calls and viewing the source code of kernel objects, stepping through kernel code is not supported.

AMD CodeXL
CodeXL is the name of AMD's flagship in OpenCL development and comes with excellent Visual Studio integration. The tool suite provides a sophisticated debugger which allows debugging all work items in parallel. Therefore, special multi-watch windows can be opened for tracking variables across all work items, shown in a tabular view. Breakpoints and

stepping are fully supported within Visual Studio. Furthermore, an application can be launched in profiling mode where CodeXL reads important performance counters from the GPU for each executed kernel. This information is very useful in hunting down bottlenecks and optimizing kernels for specific GPUs. Additionally, a static kernel analyzer is provided which can show the disassembly of a kernel and provides general static information such as consumed registers or required local memory. Unfortunately, CodeXL is limited to AMD cards only and has therefore not been used during the internship. However, it proved to be highly useful during the development of the algorithms presented in the first part of this thesis "GPGPU Computing with OpenCL"

5.3 Existing problems

Although the developed ray casters achieve good performance and provide visually satisfying results, there is still room for improvement. One of the biggest problems for the boolean counting kernels is the intersection buffer which requires 1.2 KiB of memory for each work item. This (relatively) large amount of memory cannot be held in registers and is spilled into the slow global memory. Some time was actually spent during the internship on thinking how this buffer could be avoided. A possible alternative is to organize the triangles inside a cell in an intelligent data structure which allows to iterated the triangles depth-sorted, e.g. a BSP tree. However, this would add additional overhead to the grid management routines as these data structures have to maintained or rebuilt each time geometry is added to the grid. Furthermore, data structures also add more complexity to the memory layout of the triangles. Finding a better solution to this problem would probably result in a larger performance boost.

A second problem is the memory access pattern. GPUs are designed for so-called coalesced memory reads, which means that consecutive work items should read from consecutive memory addresses from global memory. This is definitely not the case in all caster implementations, as cells and triangles are accessed at almost random locations. At this point is probably also room for optimization by maybe copying all triangles of a cell into shared memory using one large fetch across all work items instead of every work item requesting all triangles.

Finally, also the result of a profiler would be very valuable to detect bottlenecks which are not as obvious as the previous ones. However, profiling an OpenCL application on NVIDIA hardware seems to be tedious with the current tools provided by this hardware vendor. Unfortunately, no AMD graphic cards are available at RISC.

6 Summary and conclusion

The RISC Software GmbH is a limited liability company and part of the software park Hagenberg. It focuses on the practical application of research done in the corresponding RISC institute. In mid 2011, the project Enlight was started with financial help by the governmental Regio 13 program. The goal of Enlight is to create a ray casting solution for interactive visualization of complex geometries. Subtractive manufacturing is the main inspiration for the project. At the start of the internship in April 2013, most functionality was already implemented. The goal was therefore to try a different ray casting approach using GPGPU computing and OpenCL.

Ray casting is a wide spread technique for creating a two-dimensional image of a three-dimensional scene by casting rays from a camera position through the pixels of an image plane in space on which the final image should be projected. Ray casting has a different run time complexity than traditional rasterization, from which especially large scenes benefit. Ray casting also allows to easily generate images of implicit geometries like CSG models, where the scene is described by boolean combination of volumes. Counters are used in this case to count volume entries and exits until the surface hit is found. To accelerate ray casting several data structures are used to organize the scene more efficiently. One of them are regular grids, where the scene is subdivided into equally sized cubes. Grids are advantageous over other wide-spread data structures like kd trees for being easy and fast in construction, better facilitating dynamic scenes. During ray casting, grids are traversed by individual rays using a 3D variant of the DDA algorithm. Ray packets are guided through the grid slice by slice.

OpenCL is an open and free standard for parallel and general purpose programming targeting heterogeneous platforms. It is most often used to write programs, called kernels, for GPUs. To use OpenCL in an application, an SDK is required to provide the necessary header files and libraries. A typical OpenCL application starts by selecting an available platform and device as well as creating a context and a command queue. Kernels are written in OpenCL C and compiled at run time for the chosen device. Buffers may be created to pass larger blocks of data between the host application and a kernel. Kernels are executed in an n-dimensional range which determines the number of work items which should be executed. The memory model of OpenCL closely resembles modern GPUs by distinguishing between global, local, private (registers) and constant memory.

The existing prototype at the time the internship started is a C++ application built using Visual Studio and Intel's C++ compiler. It heavily uses AVX intrinsics to process ray packets as fast as possible in a SIMD fashion, thus limiting the application to newer processor types. The acceleration structure used is a regular grid. By classifying the grid cells every time a subtraction volume is added, the relevant cells for ray casting can be detected leading to a further reduction of intersection tests. By using subtraction volumes to express complex geometries, the ray casting algorithm has to use counters for volume entries and exists in order to find the implicit surface.

During the internship, several OpenCL ray casters have been developed together with a small OpenCL driver running these casters. Mainly single ray variants where focused, as they involve no synchronization between individual rays. Several advanced features were added such as double precision support, source file embedding and out of core ray casting. The built infrastructure was finally ported to a new prototype which will be used for public demonstration and provides the base for a final product.

The benchmark results show that the OpenCL implementation can definitely compete with the existing CPU variant with a speedup of two to four on different scenes using single precision. The

visual quality of the output can be considered equal with the CPU double precision implementation. During development, several tools have been tried and evaluated. However, a few problems still remain which could further improve the ray casting performance on GPUs.

In conclusion it can be said that the OpenCL port of the ray caster was successful. Ray casting offers a lot of parallelism due to its parallel nature which can be advantageously used in GPGPU computing. However, we also saw that writing programs for the GPU is not as easy as developing an algorithm on the CPU. Commonly used and well-established concepts like dynamic allocation and communication methods between threads are suddenly unavailable when programming using OpenCL. Different aspects like memory access patterns, branch divergence between threads and register footprint come into play which are usually neglected in traditional software development.

Although the general purpose graphics processing unit becomes more and more capable of executing non-graphical tasks and running algorithms very dissimilar to the ones the a GPU was initially designed for, GPUs are still not general purpose processors. A graphic card remains a highly specialized instruments with its main goal of accelerating computer graphics. Therefore, only a small subset of problems actually benefits from being run on a GPU. Ray casting is fortunately one of them. Although we have also seen, that less independently parallel approaches, like the packet ray caster, which requires a high amount of synchronization, lead to a drastic loss of throughput.

Finally, also the developer tools available for OpenCL are far from being as satisfying as their CPU counterparts. During the internship, Intel's VTune Amplifier has been used for optimizing CPU code and was amazingly helpful. Also debugging C++ code, even if run in multiple threads, is well supported by today's debuggers like the one integrated in Visual Studio. GPGPU computing is still a young discipline, also for vendors. The provided tools seem to be immature and unstable in several cases. Intel's OpenCL debugger for example, or AMD's CodeXL. Some tools are also only available on a certain kind of hardware like NVIDIA's Visual Profiler or AMD's CodeXL. Although consequent improvements and advances are being made, there is still a lot of work to do to make GPU development, especially with OpenCL, as convenient and easy as traditional CPU orientated software development.

Concerning the future of Enlight, the initial planning schedules the project's end to December 2013. During this time, a lot of innovative work has been made which has been submitted to various conferences. Especially the idea of classification was new and proved to largely accelerate the ray casting of scenes described by subtractive volumes. RISC is currently discussing the integration of the ray casting technique developed during Enlight into other customer products, thus bringing the gained knowledge to its application. Furthermore, a follow-up project is in consideration, which may evaluate ray casting using a different but promising new piece of hardware on the high performance market, Intel's many integrated core (MIC) coprocessor cards. By being designed for massively parallel acceleration using existing software technologies and languages, Intel's MIC architecture does not suffer from the design requirements of a GPU, as it consists of a huge array of conventional Intel x86 cores. Who wants to be annoyed by GPGPU computing then?

List of Figures

References

[1] John Amanatides and Andrew Woo. "A fast voxel traversal algorithm for ray tracing". In: *In Eurographics '87*. 1987, pp. 3–10.

[2] Intel Corporation. *Intel® SDK for OpenCL* Applications 2013*. 2013. URL: http://software.intel.com/en-us/vcsource/tools/opencl-sdk (visited on 2013-09-06).

[3] NVIDIA Corporation. *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Kepler TM GK110. The Fastest, Most Efficient HPC Architecture Ever Built*. Version 1.0. Jan. 28, 2013. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 2013-04-13).

[4] Dipl.-Ing. Wolfgang Freiseisen. *RISC Software GmbH*. RISC Software GmbH. 2013. URL: http://www.risc-software.at (visited on 2013-08-27).

[5] RISC Software GmbH. "Enlight - Visualisierung und Modellierung sehr komplexer, detailreicher Geometriemodelle". ger. Project proposal. Mar. 28, 2011.

[6] Michael F. Hava. "Ray Casting von triangulierten Objekten. Bachelorarbeit Teil 2". ger. Bachelor thesis. FH Oberösterreich, June 2011.

[7] David J. Kasik, Dinesh Manocha, and Philipp Slusallek. "Guest Editors' Introduction: Real-Time Interaction with Complex Models". In: *Computer Graphics and Applications, IEEE* 27.6 (2007), pp. 17–19. ISSN: 0272-1716.

[8] Mark Kilgard. *Ray Casting & Tracing*. University of Texas. Apr. 17, 2012. URL: http://www.slideshare.net/Mark_Kilgard/24raytrace-12595304#13776911378721&hideSpinner (visited on 2013-08-28).

[9] Paulius Micikevicius. *Local Memory and Register Spilling*. NVIDIA. 2011. URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf (visited on 2013-09-03).

[10] Tomas Möller and Ben Trumbore. "Fast, minimum storage ray-triangle intersection". In: *J. Graph. Tools* 2.1 (Oct. 1997), pp. 21–28. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487468.

[11] Aaftab Munshi, ed. *The OpenCL Specification*. Version 1.2. Khronos Group. Nov. 14, 2012. URL: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (visited on 2013-04-08).

[12] Ingo Wald et al. "Ray tracing animated scenes using coherent grid traversal". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 485–493. ISSN: 0730-0301. DOI: 10.1145/1141911.1141913. URL: http://doi.acm.org/10.1145/1141911.1141913.

[13] spectral. *CUDA 5.0*. Oct. 18, 2012. URL: http://ompf2.com/viewtopic.php?f=8&t=1019 (visited on 2013-09-04).

A Test environment

The following hardware and software configuration has been used for development, testing and benchmarks:

- Windows 7 x64
- Intel Core i7 Quadcore (8 virtual cores) at 3.4 GHz
- 8 GiB DDR3 RAM system memory
- NVIDIA GeForce GTX660 with 2 GiB GDDR5 RAM and 1344 CUDA Cores