Volume Rendering Optimisations for Mobile Devices

by

Antonio Nikolov

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

May 2015

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Antonio Nikolov

May 20, 2015

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Antonio Nikolov

May 20, 2015

Acknowledgments

I would like to thank my supervisor Dr. Michael Manzke for taking me on this dissertation and introducing me to the world of volume rendering. I would like to extend my thanks to Tom Noonan and Niall Mullally for giving me a head start in the development of the application. My deepest gratitude goes to Sarah and Giovanni for their support during the good times and the bad times throughout the project.

ANTONIO NIKOLOV

University of Dublin, Trinity College
May 2015

Volume Rendering Optimisations for Mobile Devices

Antonio Nikolov

University of Dublin, Trinity College, 2015

Supervisor: Dr. Michael Manzke

Historically volume rendering has been associated with lacking the performance of traditional mesh-based rendering to achieve interactive speeds. The parallelism in modern CPUs and GPUs has allowed volume rendering to become a viable choice in movie special effects, destructible environments in games and medical imaging. Different software techniques have been developed to unlock the potential of the massively parallel architecture of the GPU. While a modern desktop hardware is more than capable of achieving volume rendering at interactive speeds, a modern mobile device is significantly lacking in power. A use for volume rendering on a mobile device would be a doctor using a device capable of reconstructing a 3D representation of a patients bone structure without stationary MRI scanner and then rendering the volume. This paper investigates the performance of ray casting against texture based volume rendering on mobile devices using OpenGL ES shaders as well as the performance of ray casting volume rendering using general purpose programming on the GPU (GPGPU) using CUDA. This project contributes a cross-platform volume renderer on Windows and Android with four performance tested implementations: ray casting

using shaders, ray casting using CUDA, single-threaded texture based and multi-threaded texture based volume rendering. Results show that generally the single-threaded approach is the fastest but produces lower quality image while ray casting using shaders is significantly faster than raycasting using CUDA.

Abbreviations

CPU - Central Processing Unit
CT - Computed Tomography
CUDA - Compute Unified Device Architecture
GPU - Graphics Processing Unit
FPS - Frames per Second
MRI - Magnetic Resonance Imaging
SoC - System on Chip

Summary

Chapter 1 *Introduction* sets the goals of the project and introduces the technical concepts used in the project. The goal of the dissertation was to develop a volume renderer that runs at interactive speeds on a mobile device architecture using modern OpenGL ES and GPGPU through CUDA programming. Chapter 2 *Previous Work* presents the current state of the art algorithms used in volume rendering for mobile devices. State of the art research uses both ray casting and texture based algorithms but need a workaround when representing a 3D texture with the outdated OpenGL ES 2.0 API which does not support 3D textures. An exposition of the various bottlenecks in the graphics pipeline is presented to put in perspective the ways of optimising the pipeline. Four different implementations of volume rendering were implemented in the project as described in Chapter 4 *Implementation*. Namely, ray casting using shaders, ray casting using CUDA, single-threaded texture based and multi-threaded texture based. Chapter 3 *Experiments* describes the tests that were carried out before the results of the experiments are presented. The implementation of the application is directed by the eventual testing rather than the other way around. Chapter 5 *Results and Evaluation* shows the visual output of the four different rendering methods used in the application as well as the performance tests for those methods. Evaluation on those results is also presented. Ray casting produces the best visual output while single-threaded texture based is the fastest. It was shown that the CUDA implementation performs significantly worse than its shader based equivalent.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **xi**

List of Figures **xii**

Chapter 1 Introduction **1**

1.1 Initial Goals . . . 1
1.2 What is Volume Rendering? . . . 2
1.3 Volume Rendering Concepts . . . 3
1.3.1 Ray Casting Based Volume Rendering . . . 4
1.3.2 Texture Based Volume Rendering . . . 6
1.4 Nvidia Shield Hardware Architecture . . . 9
1.4.1 Performance per Watt . . . 10
1.4.2 Quad Warp Scheduler . . . 10
1.5 Memory system . . . 12
1.6 CUDA Programming . . . 13
1.6.1 CUDA Programming Model . . . 13
1.6.2 Memory Hierarchy . . . 14
1.6.3 Disadvantages of Using CUDA on the Shield . . . 16

Chapter 2 Previous Work **18**

Chapter 3 Experiments **23**

Chapter 4 Implementation **28**
4.1 High Level Overview 28
4.1.1 Platform Dependent Code 28
4.1.2 Platform Independent Code 29
4.2 Ray Casting Using Shaders 30
4.3 Ray Casting Using CUDA 31
4.4 Textured Based Single-threaded 32
4.5 Texture Based Multi-threaded 36

Chapter 5 Results and Evaluation **37**
5.1 Visual Appearance 37
5.2 Performance Tests 42
5.2.1 Windows 42
5.2.2 Android 42
5.3 Evaluation 43

Chapter 6 Conclusion **49**

Bibliography **50**

List of Tables

3.1 The specifications used for testing on the desktop and mobile versions . . . 25
3.2 The volumes used in the project 27

5.1 Windows, CT Knee , 250 samples 43
5.2 Windows, CT Knee , 500 samples 43
5.3 Windows, Sphere, 250 samples 44
5.4 Windows, Sphere, 500 samples 44
5.5 Android, CT Knee , 250 samples 45
5.6 Android, Sphere , 250 samples 45
5.7 Multithreading: Windows, CT Knee, 250 samples 46

List of Figures

1.1 Ray casting volume rendering 5
1.2 Texture based volume rendering pipeline 6
1.3 Slicing the volume from the current viewpoint. There can be a minimum of 3 vertices and a maximum of 6. 7
1.4 Triangle fan. triangle are made by using the starting vertex and each subsequent two triangles: $\{v_0, v_1, v_2\}$, $\{v_0, v_2, v_3\}$... $\{v_0, v_7, v_8\}$ 8
1.5 Kepler GPU in the Tegra K1 Processor 11
1.6 Kepler Memory Hierarchy 12
1.7 Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel 15
1.8 Memory Hierarchy 16

4.1 Architectural overview 29
4.2 Three ways of getting the minimum and maximum points 33
4.3 Converting from uvt to Cartesian coordinates 34
4.4 Percentage spent in the proxy generation loop 36

5.1 Comparison of ray traced using shaders vs CUDA with 250 samples 39
5.2 Comparison of ray traced using shaders vs CUDA with 500 samples 40
5.3 Comparison of textured based with 250 and 500 samples 41
5.4 Using more opaque transfer function 42
5.5 Performance decrease of the shader versus the CUDA implementation when changing from 250 to 500 samples 47
5.6 Performance difference of ray casting using shaders over CUDA 48

Chapter 1

Introduction

1.1 Initial Goals

It is helpful to present the concrete goals of the dissertation to get a perspective of the project scope. The goals will also serve as a success criteria at the end of the project.

- Develop a volume renderer that will run on mobile devices at interactive speeds.
- Optimise the application to suit the system on chip (SoC) architecture of modern mobile devices.
- Test the suitability of ray casting volume rendering against texture based volume rendering.
- Test the suitability of ray casting volume rendering using CUDA on a mobile device with CUDA capabilities.

Volume rendering techniques are developed with a powerful desktop setup in mind so in order to test the suitability of existing techniques on a mobile device it was necessary to develop the application cross-platform for a desktop and for a mobile environment simultaneously in order to test the relative performance difference of each technique.

1.2 What is Volume Rendering?

Volume Rendering is the representation and visualisation of three-dimensional arrays of data representing spatial volumes. [DCH88] Volume rendering has found many uses in visualisation of data that is difficult to either gather or model with geometric primitives. [Fer04] Among notable example of the use of volume rendering include the generation of volumes by imaging a series of cross sections using Computed tomography (CT) and magnetic resonance (MR) scanners in medicine and visual effects which are volumetric in nature like fluids, clouds or fire.

The three dimensional nature means that visualising the dataset involves careful choice on how to map the data on a two dimensional surface. It is important to be able to view the data from different viewpoints and to shade based on the density and opacity values[1] in order to extract the features and surfaces within. Much of current research in volume rendering is concerned with the automatic extraction of such features. Instead, this dissertation focuses on optimisation in volume rendering as it has been traditionally been recognised as computationally expensive. Recent advances software techniques and especially in the hardware capabilities of modern graphical processing units (GPU) means that volume rendering has become a more viable choice in many circumstances in real-time rendering where the traditional mesh based approach to rendering would have been the only choice in the past. The hardware of mobile devices lacks in power so it would be several years before current techniques in real time volume rendering can be effectively applied there. Interactive 3D rendering on mobile devices is mainly constrained by limited power and memory capacity.

A class of volume rendering techniques called *isosurface extraction* involve imposing geometric structures where intermediate geometric representation of a surface is extracted first and then a conventional rendering approach is used to render the geometric primitives. The volume data is represented in a discrete binary form where a data element (or *voxel*) is either present or not. A disadvantage of this technique is the introduction of false positives artefacts into the final image as its nature involves intersecting geometric objects into the volume data whose edges may not necessarily intersect the data items in the volume. [WW92]

On the other hand, ray casting is a technique that attempts to generate an output image from a volume of continuous data values from 0.0 to 1.0 denoting the opacity[2] of the voxel. A

[1] more on those terms later

[2] in the context of graphics processing the term transparency is the opposite of opacity. In most cases when

ray is cast from the image plane and the values that it intersected are interpolated using a *transfer function*. It is a function which determines the mapping between the 3D volume space to the 2D image plane for each ray. The *sampling rate* is the measure of separation between successive contributing voxels. Non-uniform sampling can be achieved by defining a density property over distinct regions of the volume. The density describes the degree of compactness of the voxels. For example, sparsely populated regions or regions with empty space typically found at the front and back of the volume can have low density so they can be omitted from processing provided an algorithm is integrated to detect such empty regions.
A single voxel can be associated with more than just transparency. Depending on the desired image quality several other properties can be introduced such as colour, normal (facing direction), emissive, reflective and diffuse properties. [WW92]. Fitting geometric isosurfaces is known as indirect volume rendering whereas the use of ray casting is known as direct volume rendering because there is no explicit step to extract geometric surfaces as first proposed by [Lev88]. Due to the non-binary classification, small or poorly defined features are not lost
.

1.3 Volume Rendering Concepts

The transport equation for volume rendering is given by: [Lev88]

$$C_\lambda(x) = \sum_{k=0}^{n} c_\lambda(x + r_k)\alpha(x + r_k) \prod_{l=k+1}^{n} (1 - \alpha(x + r_1)) \tag{1.1}$$

where λ is a wavelength constant, $C_\lambda(x)$ is the final colour at position x, $c_\lambda(x + r_k)$ is the colour of the kth sample at position $x + r_k$ inside the volume, and $\alpha(x + r_k)$ is the opacity at position $x + r_k$. The equation neglects illumination. It computes the final colour by composing colour and opacities along a line.
The way the equation work is that for any given sample $x+r_k$ along the line – its contribution to the final colour will be higher than the remaining samples if it has higher opacity and the opacities of the remaining samples along the line are smaller. Inversely, if the remaining opacities along the line are higher than the current sample then the current sample will have

saying ”the opacity value of” is the same as saying ”the transparency value of”

less of a contribution.

1.3.1 Ray Casting Based Volume Rendering

This project implements a ray casting approach and a texture based to volume rendering. Ray casting is the process of generating a ray vector from a point in a certain direction. In the context of volume rendering, a ray is cast from the viewpoint of the centre of the camera towards the volume. The volume itself is enclosed in a bounding box which denotes the boundaries of the 3D texture that represents the volume. Typically, a bounding box with dimensions of -1 to 1 along the three primary axis is used.
The 3D volume stored in the host computer is loaded into RAM. It is usually stored in a specific format such as .raw or .pvm which determines how the data is laid out inside of the file. The 3D volume is written to a buffer and loaded onto the GPU only once at the initialisation stage of the application. Depending on the complexity of the implementation, additional data structures such as an octree may be loaded on the GPU and updated frequently to reflect changes in the viewpoint. The transfer function is usually stored in texture memory in the GPU. Depending on the dimension of the transfer function, it can be stored as a 1D, 2D or 3D texture. The user can also change the transfer function at runtime to render the volume differently. Shaders are created to to represent the algorithm that is to run the GPU. Depth testing and culling are enabled. The volume is traversed with ray casting on each frame of the application after initialisation.
Conceptually, there are four stages to the basic ray casting algorithm as illustrated in Figure 1.1. First, a ray is cast for each pixel of the screen. For each ray, the bounding volume is checked for intersection at this stage. Secondly, equidistant sample points are generated between the minimum and the maximum point of where the bounding box was intersected. Sample points are located between voxels so trilinear interpolation is applied using the neighbouring voxels. Thirdly, each sample point serves as a lookup into a transfer function to produce a colour value for that particular point in the volume. The shading of each sample can be more complicated such as applying local illumination using the gradient of the sample or using a 2D transfer function. And finally, the samples are composited in either front-to-back or back-to-front order using the opacity of each sample before the opacity and colour is supplied to the transport equation.

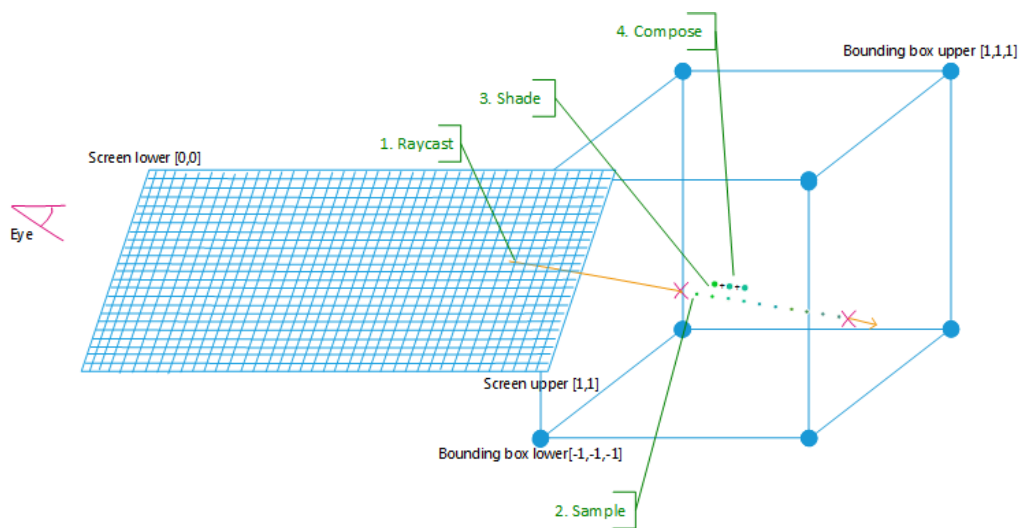


Figure 1.1: Ray casting volume rendering

Algorithm 1 illustrates how the rendering equation (1.1) is solved by ray casting.

Algorithm 1: Ray Casting

```
1  def raycastVR(pixelPos, eyePos, maxRaySteps, rayStepSize, volume,
   transferFunction)
2  begin
3      finalColor ← black
4      currPos ← pixelPos
5      direction ← normalise(currPos - eyePos)
6      accumAbsorption ← 0
7      for i ← 0 to maxRaySteps:
8          currColor ← black
9          textureCoord ← currPos transformed into texture space
10         density ← sample(volume, textureCoord)
11         currColor ← sample(transferFunction, density)
12         opacity ← currColor.alpha
13         if (accumAbsorption + opacity) higher than 1.0:
14             finalColor = finalColor + currColor * (1.0 - accumAbsorption)
15         else:
16             finalColor = finalColor + currColor * opacity
17         accumAbsorption = accumAbsorption + opacity
18         currPos = currPos + (direction * rayStepSize)
19         if not intersect(currPos, bounding box) or accumAbsorption higher than 1.0:
20             break
21 return finalColor
```

1.3.2 Texture Based Volume Rendering

The end goal in texture based volume rendering is the same as when using ray casting: render the volume from the current viewpoint. This is achieved differently by generating proxy planes that slice the volume rather than shooting a ray through the volume for each pixel. Each of the proxy planes represents a 2D slice of the volume. All the proxy planes are blended using the transport function to create the final picture on the screen.
Figure 1.2 illustrates the conceptual stages involved in a texture based approach.[fer] The

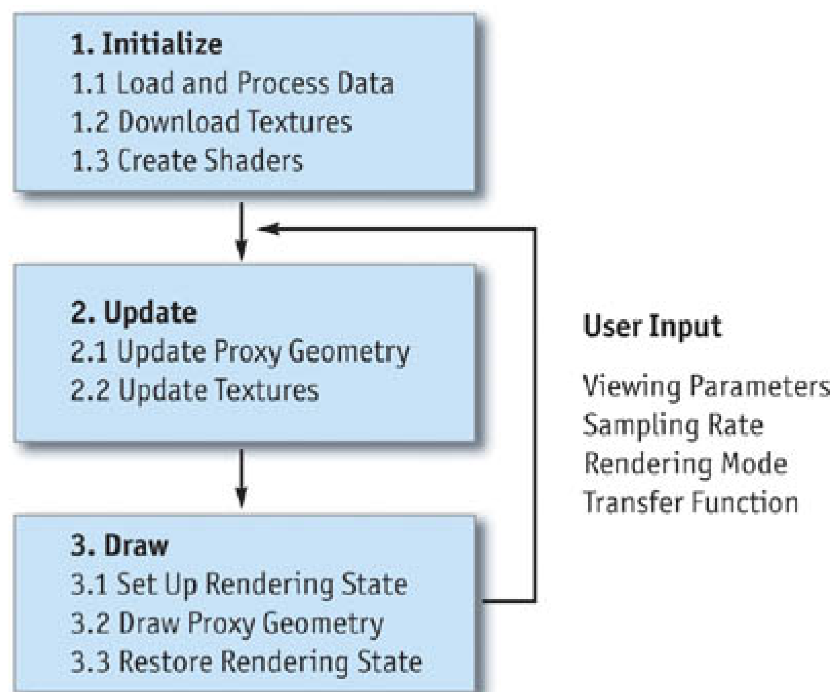


Figure 1.2: Texture based volume rendering pipeline

initialisation stage is the same as in the ray casting approach. The update stage involves the update and render loop of the application. Every time the viewing parameters change, the proxy geometry has to be recomputed and reloaded onto the GPU. It consists of a set of polygons usually stored as triangles or as is the case in my implementation – triangle fans. A triangle fan (Figure 1.4) is easy to use to represent a proxy plane as the algorithm that generates a proxy plane relies on a centre point and projecting outwards towards the edges of the bounding volume to generate the vertices of the proxy plane. The proxy geometry slices through the volume perpendicular to the viewing direction as shown in Figure 1.3 A

proxy polygon is computed by first by first projecting a plane perpendicular to the viewing plane. Then this plane is intersected with the edges of the volume bounding box to create the vertices of the polygon. Those vertices are sorted in a clockwise or counterclockwise direction around the centre. Sorting is required in order to create the correct buffer for the GPU. In the draw stage, setting up the rendering stage involves binding the buffer with the proxy geometry to the GPU. Lighting and culling and depth buffering are disabled and the alpha blend function is setup according to equation (1.1).
Each vertex of a proxy plane represents the texture coordinate inside the volume texture. All the proxy planes are then blended in back-to-front order.
The sampling rate determines how many proxy planes will represent the volume. It is determined by the resolution of the volume, the viewing angle and the maximum number of samples allowed. Usually, texture based approaches produce lower visual fidelity than ray casting when the sampling rate is the same thus a higher sampling rate is desired.

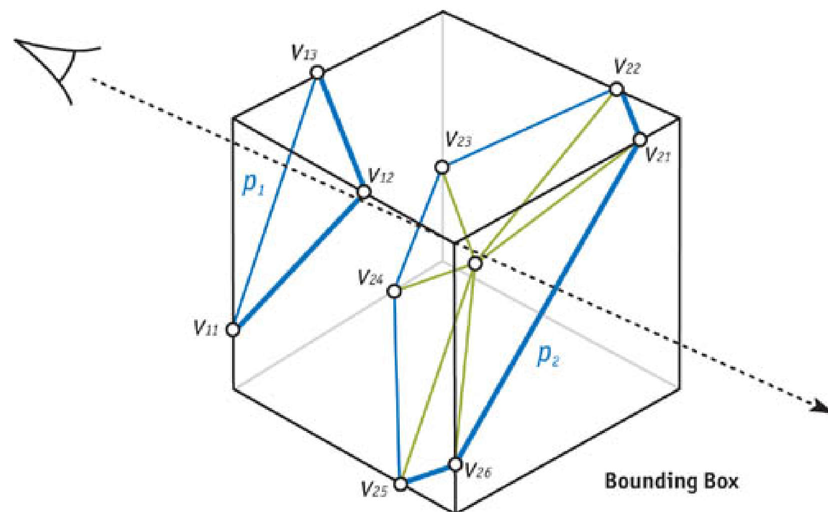


Figure 1.3: Slicing the volume from the current viewpoint. There can be a minimum of 3 vertices and a maximum of 6.

For correct behaviour the proxy planes used for slicing have to be view-aligned. Algorithm 2 provides a high-level explanation of how to compute the proxy geometry in view space. It

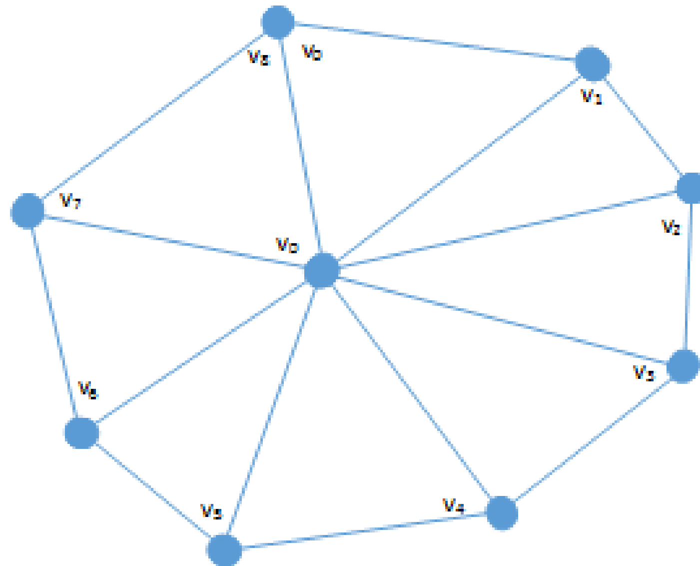


Figure 1.4: Triangle fan. triangle are made by using the starting vertex and each subsequent two triangles: $\{v_0, v_1, v_2\}$, $\{v_0, v_2, v_3\}$... $\{v_0, v_7, v_8\}$

transforms the vertices of the bounding box from object space to view space coordinates.

Algorithm 2: Geometry generation for textured based volume rendering

1 bbVerticesView ← verticesLocal * modelViewMatrix /*bounding box vertices*/
2 bbEdgesView ← edgesLocal * modelViewMatrix /*bounding box edges*/
3 (min,max) ← compute min and max z coordinates of bbVerticesView
4 distanceSample ← (max - min) / maxSamplingPts
5 **for** *each sample in back-to-front order***:**
6   intersectionPtsProxy ← get intersections with bbEdgesView
7   centreProxy ← average the points in intersectionPtsProxy
8   sortedProxy ← sort points counterclockwise
9   triangleFanProxy ← tesselate sortedProxy
10   vertexBuffer ← add triangleFanProxy

Line 3 in the algorithm is somewhat ambiguous. It does not specify whether the min max z coordinates are supposed to be of the vertices themselves or the closest and furthest points of the cube relative the view point. In the implementation section, a discussion of both approaches is given. Line 4 makes a simplification for the total number of sampling points. The algorithm does not account for the spacing between the voxels and assumes a constant

spacing to ease testing. This constraint has the implication that volumes of different sizes will essentially be represented equally within the same bounding box constraints.
The most significant bottleneck in texture slicing is rasterization. The rasterizer produces many fragments for each the proxy plane. The sampling rate greatly influences the load on the rasterizer. The higher it is the higher the number of proxy planes generated. Also transparent proxy geometry cannot leverage early depth culling to reduce the number of fragments produced.
Another possible bottleneck is the generation of the proxy planes. Assuming the viewing angle relative to the volume changes on each single frame then all the proxy geometry has to be recomputed anew. For example, if there are 1000 sampling points then each of the steps in Algorithm 2 has to be computed 1000 times.

1.4 Nvidia Shield Hardware Architecture

The Shield boosts a system-on-chip processor(SoC) with the Tegra K1 mobile processor incorporating the powerful Nvidia Kepler GPU architecture [Nvi14a]. The CPU is based on the ARMv8 architecture and the Kepler GPU has 192 CUDA cores. According to Nvidia, the Tegra K1 processor delivers performance comparable to that of PS3 and XBOX 360. The Kepler architecture delivers up to three times the performance per watt compared to its predecessor Fermi. [Nvi12] There are several features that make the Kepler architecture get more GPU utilisation without the need for higher clock speeds.

Dynamic Parallelism allows the GPU to generate new tasks dynamically without the scheduling and control of the CPU. This allows larger portions of the application to run entirely on the GPU.

HyperQ is a feature which enables multiple CPUs to launch work on a single GPU simultaneously reducing CPU idle time. It increases the total number of connections between the host CPU and the GPU, allowing 32 hardware-managed connections compared to only a single one on the Fermi architecture. It allows multiple CUDA streams, threads within a process or multiple processes to run separately at the same time. Previously, applications encountered false synchronisation across tasks because they were serialised on the single connection.

Grid Management Unit (GMU) is the dispatch and control system which manages and priorities grids to be executed on the GPU. A *grid* is an array of result blocks where each block consists of hardware work threads. [LNOM08] It can pause dispatching grids and queue pending suspended grids until they are ready to execute thus providing the flexibility needed to enable Dynamic Parallelism.

The Kepler GPU in the Tegra K1 processor is identical to the Kepler architecture used in high-end systems. It includes optimisations specifically targeting mobile systems with emphasis on conserving power. [Nvi12]. The highest-end Kepler GPUs consist of up to 4992 CUDA cores and consume up to 300W [Nvi14b] whereas the Kepler in the Tegra K1 consists of 192 CUDA cores and consumes less than 2W. The logical organisation of the hardware can be seen in Figure 1.5. The Tegra K1 Kepler consists of one Graphics Processing Clusters (GPC), one Streaming Multiprocessor (SMX) and one memory interface. The high-end counterpart consists of four GPCs, eight SMX and four memory controllers.

A SMX unit features 192 fully-pipelined floating-point and integer arithmetic logic units. It has IEEE754 compliant single- and double-precision arithmetic with fused multiply-add (FMA) operations. The special function unit (SFU) is used for fast approximate transcendental operations like square root and sine. There are eight times the number of SFUs in Kepler compared to the Fermi architecture.

1.4.1 Performance per Watt

The design philosophy focuses on consuming less power and generating less heat than its predecessor. One way of achieving this was to use the primary GPU clock rate within the SMX unit instead of the twice as fast shader clock. Running at higher clock rates allows more throughput with fewer copies of the performed by the execution units. This is an area optimisation but the added clocking logic means the cores require more power and dissipate more heat. Nvidia chose to optimise for power at the expense of area cost by increasing the number of processing cores running at lower clock speed.

1.4.2 Quad Warp Scheduler

The SMX schedules threads in groups of 32 parallel threads called *warps*. Four warps can be issued and executed concurrently and two independent instructions per warp can be selected each cycle. To optimise the warp scheduler, there is hardware logic to register scoreboarding

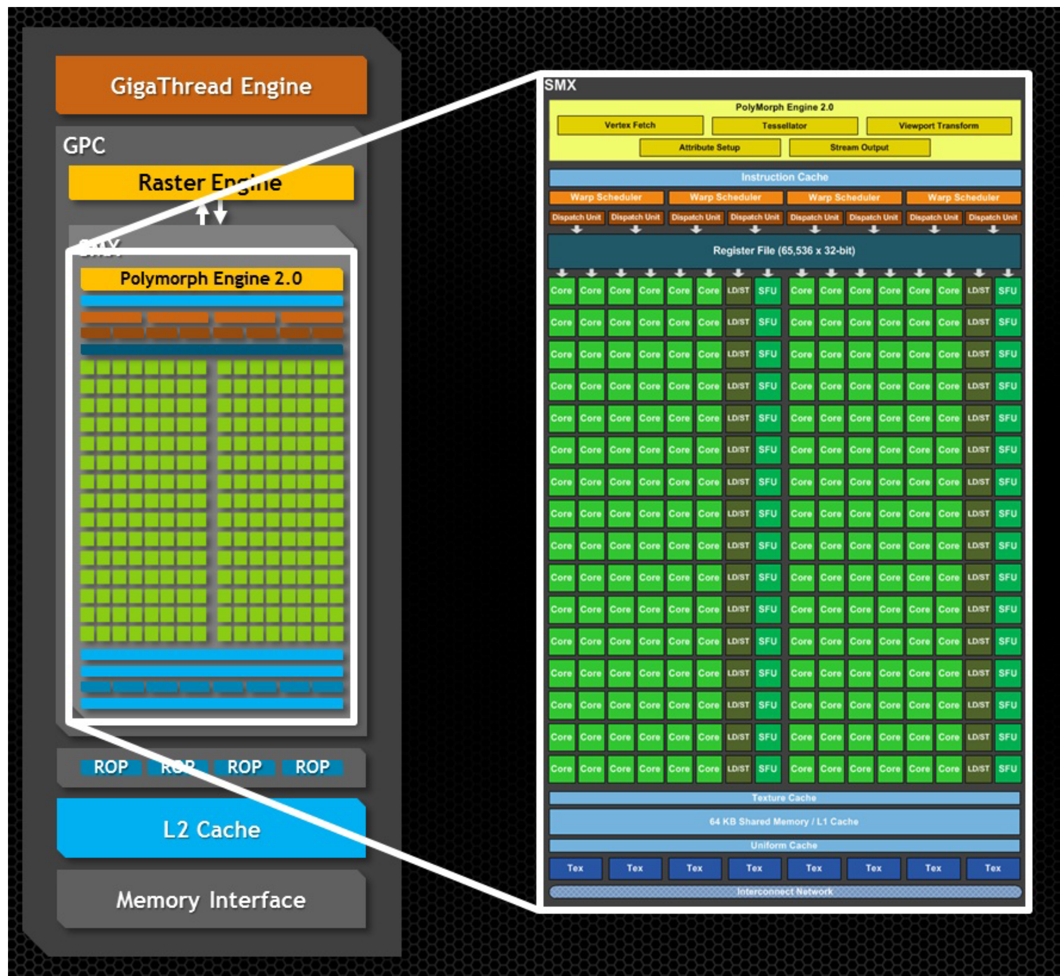


Figure 1.5: Kepler GPU in the Tegra K1 Processor

for operations with long latencies like texture read and write. *Scoreboarding* is a term used to describe dynamically scheduling instructions so they can be performed out-of-order.[Tho65] The scoreboard monitors waiting instructions to be dispatched and once all source operands are available it dispatches the instruction.

Another optimisation is scheduling on a inter-warp level and thread block level. The latter is performed by the ”PolyMorph Engine 2.0” shown in Figure 1.5. Improving from the previous architecture, Kepler replaces the complex hardware stage which prevents data hazards with a simple hardware block that extract the pre-determined instruction latency information which is then used to mask out warps for eligibility at the inter-warp scheduling level. Most of the work is left to the compiler to determine upfront the execution readiness of each instruction

and encode the information in the instruction.

1.5 Memory system

A diagram of the Kepler memory system can be seen in Figure 1.6. Each SMX has 64KB of on-chip memory that includes the L1 cache and the Shared Memory with configurable splits as 16KB/48KB, 32KB/32KB or 48KB/16KB. The bandwidth of the shared memory is 256B per core clock. The read-only data cache is used for data that is known to be constant for the duration of a function. The cache is directly accessible for general load operations. It lessens the working set footprint of the Shared/L1 cache path. It supports full speed unaligned memory access patterns. The use of the read-only cache can be managed by the compiler or the programmer through the "const __retrict" keyword. The L2 cache has 1536KB of memory. In the high-end Kepler architecture, the L2 cache serves as unification point between the SMXs on the GPU by providing load, store and texture requests. The GPU architecture supports

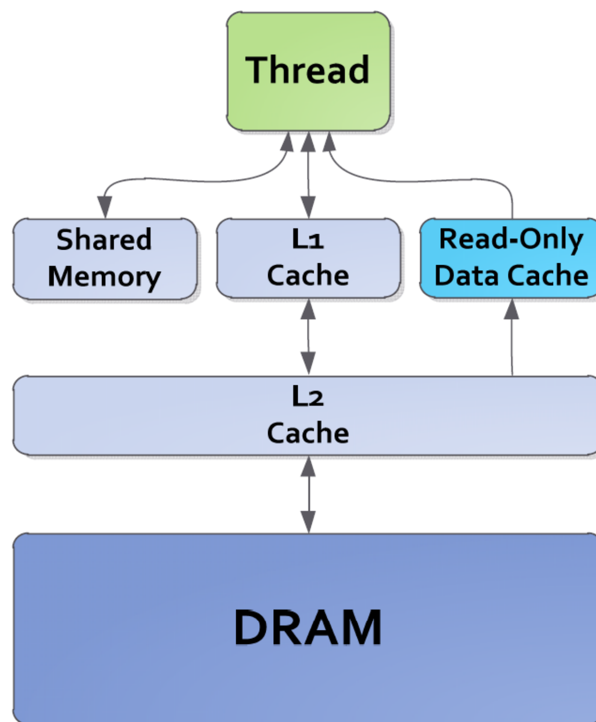


Figure 1.6: Kepler Memory Hierarchy

the DX11.2, OpenGL 4.4 and CUDA 6.0 APIs.

1.6 CUDA Programming

CUDA (Compute Unified Device Architecture) is a parallel programming platform created by Nvidia and it is available in most of today's Nvidia GPUs. CUDA allows general purpose programming to utilise the power of the graphics processor and exploit its massively-parallel architecture to perform computational tasks which lend themselves for parallelization.[Nvi08a]
The programmable GPU has evolved into a highly parallel, multithreaded processor with very high computational horsepower.[Nvi08b] It specialises in compute-intensive, parallel tasks such as graphics rendering. The memory bandwidth of a GPU is many times higher than that of a CPU. For example, an Intel processor from the Ivy Bridge microarchitecture supports a theoretical memory bandwidth of 60 GB/s and Nvidia's GeForce 780 Ti has a theoretical memory bandwidth of 350 GB/s. A GPU is designed for data processing rather than data caching and flow control. Both volume ray casting and texture-based volume rendering are intensive data-parallel computations with high ratio of arithmetic operations over memory operations. There is lower requirement for sophisticated flow control hence memory latency can be hidden with little to no data caching by the sheer amount of calculations at once. Data-parallel processing maps data elements to parallel processing threads. [Nvi08b]
In traditional 3D rendering, large sets of vertices are mapped to parallel threads.

1.6.1 CUDA Programming Model

The CUDA programming model was introduced in 2006 and allows developers to create general purpose applications that run on the GPU. It comes in an environment which uses the C programming language. It is designed to overcome the challenge of developing applications which transparently scale to leverage the increasing number of processing cores while maintaining a low learning curve for programmers familiar with the C. There are three key concepts which are abstracted by the programming model: thread groups, shared memory and barrier synchronisation. These abstractions provide fine-grained data parallelism and thread parallelism nested within task parallelism. The programmer is guided into subdividing the problem into sub-problems that can be solved in isolation at once by blocks

of threads. Each block of threads can be scheduled on any of the available multiprocessors within the GPU so that a compiled CUDA program can run on GPUs with differing number of cores. For example, the Quadro K2000 GPU and the Tegra K1 GPU used for testing in this project have 384 and 192 CUDA cores respectively but the application itself has the same code regardless of the number of cores. This scalability brought by the programming model allows the GPU as a viable option in a wide market range.
The thread hierarchy is split into threads, blocks and grids. The *thread* is a set of instructions which run in isolation of other threads. The *thread block* contains threads which are expected to reside on the same processor core within the GPU and must share the memory resources of that core. Currently, a thread block can contain up to 1024 threads. Blocks are organised into grids where each *grid* is logically mapped to a single streaming multiprocessor within the GPU. The programmer does not specify the number of grids. Instead they are automatically scheduled by the hardware based on the specified number of blocks and threads and the available number of hardware streaming multiprocessors.
Figure 1.7 illustrates the logical hierarchy. A *kernel* is a C like function that will execute a CUDA thread N number of times where N is the total number of threads scheduled to be run. For example, a thread block size of *16x16* withing a grid of *32x32* will result in 4096 threads within the grid. If there are 8 grids then the total number of unique threads running concurrently will be 32768. Typically in ray casting volume rendering, each pixel of the screen is assigned a separate thread. That means, the screen width and height are used to calculate the grid and block sizes. Thread blocks execute independently. That means, it must be possible to execute them in any order either in parallel or in series depending on the synchronisation imposed. This requirement allows thread blocks to be scheduled in any order across the available cores.

1.6.2 Memory Hierarchy

Thread within a block can communicate by sharing data through shared memory and synchronising execution to avoid race conditions. Synchronisation points are invoked at places in the kernel where shared memory is accessed by calling the CUDA function **__syncthreads()**. There are different memory spaces within the CUDA programming model that a single thread can access. How fast a memory access is depends on the type of memory which designates the locality of the data accessed. Figure 1.8 illustrates the memory hierarchy. Each thread

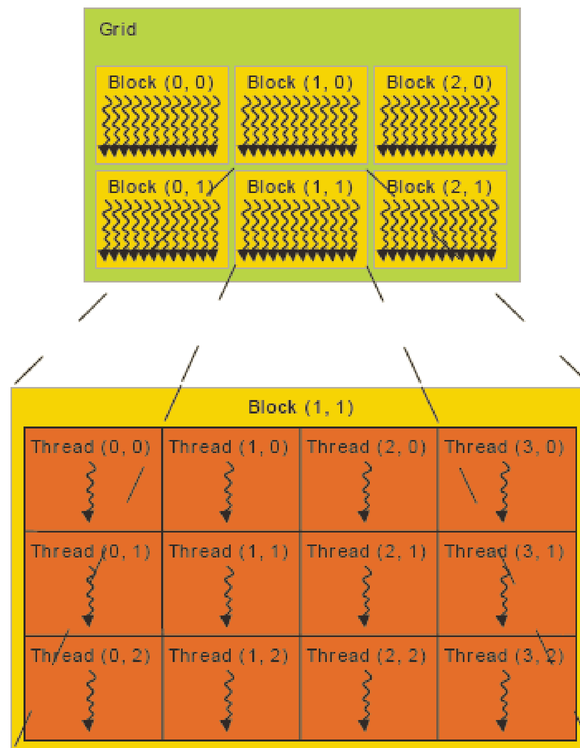


Figure 1.7: Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel

has a private local memory. Each thread block has a shared memory available to all threads of the block. Finally, global memory is shared by all threads within the kernel. One can assign read-only global memory which is called *constant memory*. It is cached so access to it is fast. Per-block shared memory is persistent only within a single kernel launch whereas global memory is persistent across the kernel launches.

CUDA threads execute on a physically separate *device* (i.e the GPU) that acts as a coprocessor to the *host* processor (i.e the CPU) running the C program. While the CUDA kernels execute on the GPU, the application that calls the kernels runs on the CPU. The CUDA programming model assumes that the host and the device maintain their own separate memory spaces in RAM. The host manages allocation, deallocation of device memory and data transfer between host and device memory.

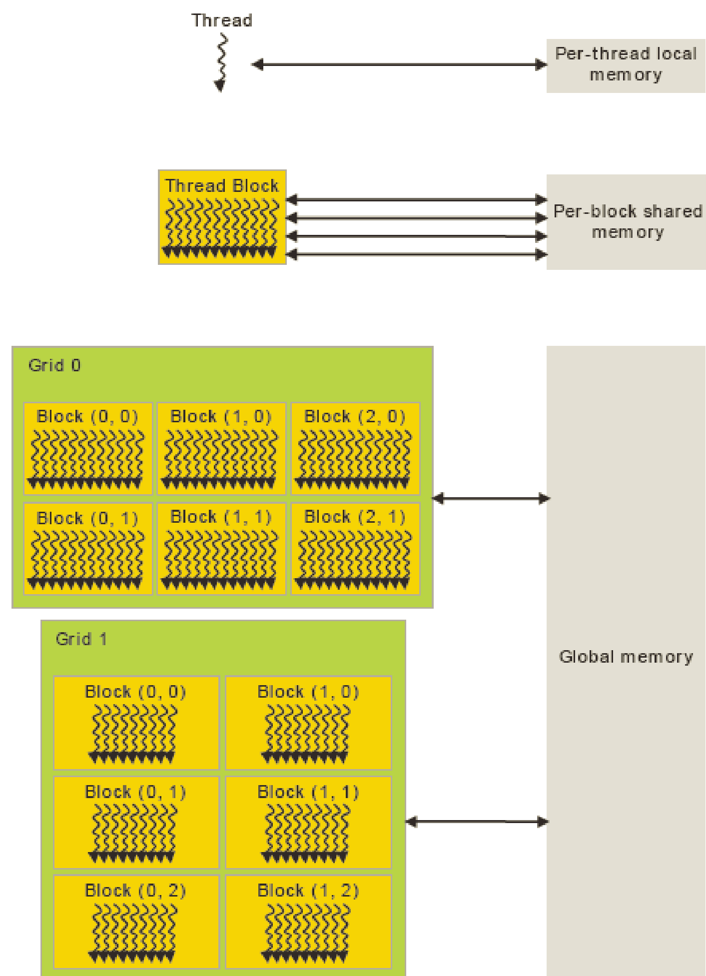


Figure 1.8: Memory Hierarchy

1.6.3 Disadvantages of Using CUDA on the Shield

Specific to this project, the biggest disadvantage of using CUDA was the limited support of the technology for the Tegra K1 processor. Developing CUDA for the Shield requires a Linux environment because the libraries that are needed to compile the CUDA code for the Shield are not available on Windows. Moreover, the only programming environment available is Eclipse CDT which has been known to have serious bugs that hamper productive development [Ecl10].

My personal opinion on the environment is that it is barely functional. I found that sometimes it would cache an old version of my project and compile that instead of what I currently

trying to build. That brought about confusion on whether I am testing my current project version or an old one. In order for me to verify what version I am testing, I had to place version number in print statement in the code that I changed everytime I tried to compile. Sometimes the program will be packaged without the runtime library and once the program runs it will complain that it cannot find the library, prompting another attempt at compiling the program.

The text editor does not give suggestion on syntax errors and available resources, further slowing down the coding process.

Eclipse would oftentimes lose connection to the Shield through the USB cable, rendering it impossible to upload a new build of the program onto the device. The only reliable temporary fix for this problem that I found was to restart the operating system. The CUDA code has to be compiled manually in the command line manually outside of Eclipse separate of the compilation of the application code. I would not recommend to develop CUDA on the Shield until the development kit for the device improves to include a proper environment that supports CUDA development.

Chapter 2

Previous Work

One of the reasons I chose to investigate graphic performance on mobile devices was the introduction of OpenGL ES 3.0 [Smi12] in 2013 to the latest iterations of mobile devices. The "ES" stands for Embedded Systems denoting that the API was designed for devices like smartphones, video game consoles and tablets. Previous research on volume rendering on mobile devices: [NJOS12], [MF12], [RA12], [MW08] dated back from 2012 at the latest. The implementations in those papers use the older OpenGL ES 2.0 that lacks some of the capabilities of modern OpenGL such as 3D texture support which is used extensively in volume rendering.

[NJOS12] compare several methods of direct volume rendering[1] with the goal of real-time interactivity on mobile devices using OpenGL 2.0. Their proposal comes from the need to have energy efficient applications by minimising the work done in the shaders. The algorithm is based on 2D texture slicing and their reason for not going with 3D texture slicing instead is twofold. 3D texture were not supported by then current OpenGL and it requires the use of proxy geometry that needs to be recomputed everytime the view angle changes.

The volumetric data is represented by placing each image slice in the 3D volume into a 2D mosaic configuration as one big 2D texture image as implemented in [CSK^{+}11] The solution for rendering the model is a texture slicing approach where the set of slices are projected perpendicular to the view direction. Equation(1.1) is computed by ordering composing the slices in a front-to-back order in the framebuffer using alpha-blending. The stack of slices is computed only once and remains static thereon. Hence, it can be cached in the GPU's

[1] ray casting is a type of direct volume rendering

memory. In order to render the model the front face of the cube must remain perpendicular to the view direction and the slices are view-aligned. To rotate the volume it is the texture coordinates that are updated. Without rotation, the 3D texture coordinates (s, t, r) are defined as follows:

$$(s, t, r) = (\pm\frac{1}{2}\frac{d}{X}, \pm\frac{1}{2}\frac{d}{Y}, \pm\frac{1}{2}\frac{d}{Z}) + (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}) \tag{2.1}$$

where (X, Y, Z) are the dimensions of the volume and d is the diagonal of the volume. The implementation relies on blending to be enabled. Equation(1.1) is computed by a blending factor and the OpenGL blending function:
glBlendFuncSeparate(GL_ONE_MINUS_DST_ALPHA, GL_ONE, GL_ONE, GL_ONE)
The vertex shader is provided with the texture transformation matrix according to the current view and the 3D texture coordinate is multiplied by the matrix to generate the rotated vertex texture coordinate. In the fragment shader, the mosaic texture is sampled according to the 3D texture coordinates. The resulting sample is used as a lookup into the transfer function texture. Fragments lying outside the volume are those with texture coordinates lesser than 0 and higher than 1. The `disgard` function is very costly in OpenGL 2.0. It is used to remove fragments from computation. Instead, an identity value of zero is assigned to the alpha of the current fragment ensuring that the blending function will not modify the framebuffer. The researchers compared their solution to the ray casting solution as described in [CSK$^+$11]. They made several conclusions based on the result gathered. The performance linearly depends on the number of fragments processed. The texture slicing approach nearly doubles the performance obtained by the ray casting solution but the at the cost of rendering quality. The researcher believe that their solution will scale better as the number of processing cores in mobile GPUs increase than the ray-casting approach.
[MF12] implemented ray-based volume renderer for mobile devices using WebGL 2.0. WebGL is a Javascript wrapper for OpenGL ES that runs cross-platform on web browser. Their implementation is a single-pass rendering pipeline that is able to handle dynamic transfer functions. The volume renderer was designed with the purpose of visualising medical images and has a transfer function widget that allows the user to change the values dynamically. They claim to achieve real-time visualisation of high-resolution volumes. They compare their implementation against a multi-pass ray-based volume renderer [KW03] that was written to target the out dated fixed-function OpenGL. The results of their experiments show that their algorithm significantly outperforms the multi-pass ray caster by a factor of two. They

contribute this speedup due to the significantly less texture fetches and efficient traversal. The efficiency of the algorithm mainly comes from the simplicity of the fragment shader which does not do any illumination. There is a single optimisation that checks the current ray against the bounding box enclosing the unit cube of the volume. A limitation that only allowed them to do hundred loop iterations in the fragment shader meant they could not implement more sophisticated shading.

[RV06] discuss the different bottlenecks present in the GPU The slowest part of the graphics pipeline is the bus from system memory to the graphics card memory. A smartphone GPU shares the same *memory bus* as the other components on the same chip thus the same constraint is very applicable to mobile devices but I would have to do further testing to conclude that memory transfer is the slowest part of the graphics pipeline. *Triangle throughput* is a measure of the speed of triangle assembly in the vertex shading stage of the graphics pipeline. It is mainly limited by the complexity of the vertex shader and the triangle setup phase which constitutes the steps taken to prepare the geometry[2]. *Rasterization* is the process of converting the scene geometry to pixels. In volume rendering, there is a lot of rasterization performed because the dataset is made up of slices of two dimensional textures which have to be mapped to the screen. A single texture slice is accessed multiple times thus techniques such as space-skipping are used to reduce the area that needs to be traversed. One of the most time consuming operations performed is texture lookup. The *texture cache size* is a limiting factor thus lower resolution textures may be used that fit into memory but that may reduce the resulting image quality. The *fragment shader* performs operations on the pixels that were previously rasterized. Simple fragment programs such as one that performs only a texture lookup, generally do not become a bottleneck but complex operations such as ligthing and multi-dimensional transfer functions can reduce the framerate. A framework that addresses those factors should be expected to perform well.

The core algorithm aims to exploit skipping parts of the volume which are void. In medical imaging for example, only 4% to 40% of the volume is occupied by regions of interest. The novelty of the algorithm comes from the space-skipping division in two stages: course division using bricking and a finer one using octrees. The steps are based on an analysis of the bottlenecks encountered in the pipeline. Bricking is the process of chopping the volume into texture bricks. The bricks area loaded into memory to serve as data for the volume rendering algorithm. The bricks address the bus and texture size bottlenecks.

[2]or in the case of volume rendering, the volumetric dataset for vertex processing

The bricks contain the original scalar values before applying the transfer function. This allows to change the transfer function on the fly. As mentioned before, the transfer function is the function that determines how the voxels along the casted ray are interpolated to determine the final colour of the pixel. Data in adjacent bricks overlaps by one pixel in every direction for correct interpolation of the transfer function. For bricks of b^3 and an overlap of n voxels, the memory overhead is $(3n/b) \times 100\%$.
Bricks are loaded into video memory as 3D textures. One of the reasons for specifically targeting OpenGL ES 3.0 is because it supports 3D textures whereas version 2.0 does not. There is a requirement that textures size be the power of 2 so if the brick does not divide evenly into brick dimensions, partially empty bricks will be added in every direction. In this implementation, the bricks are chosen as small as possible to improve the chance of bricks being completely void after applying the transfer function which is beneficial. On the other hand, smaller bricks introduce larger overhead due to the overlap needed for interpolation. The optimal brick size is defined depending on the available texture memory and nature of the dataset (e.g density).
Octree generation is then performed within each brick. An Octree is a spatial data structure which recursively divides the dataset into subtrees to speed up traversal and reduce the rasterization bottleneck. [Mea82] It is ideal for datasets which do not need to be recomputed on a regular basis.
Every octree node corresponds to a cuboid part of the voxel volume in total of eight parts. The octree is generated and traversed on the CPU. Its purpose is to reduce the workload on the GPU. Let a cell be defined as the a cube whose eight sides represent the regions of the overlapping voxels. For each element of the cell, if the a value representing the colour contribution is 0 after applying the transfer function then the cell is completely transparent. If all the elements of a cell are 0 then the octree of the cell is void and not send to the GPU for further processing. Each node has a value r of the ratio of visible to total data within the cube ($r = 1$ means completely filled). This ratio is calculated by averaging the ratios of the children of the current node. If the ratio is 0 then the the node is not drawn. If the ratio is above a certain threshold then the node is completely drawn.
Between the two stages, early-ray termination is performed to each brick. To perform this action, the volume has to be traversed in front-to-back order. The two equations which

determine when to terminate the ray are

$$C_{i+1} = (1 - A_i) \times a_i \times c_i + C_i \tag{2.2}$$

and

$$A_{i+1} = (1 - A_i) \times a_i + A_i \tag{2.3}$$

Where A, C denote the opacity and colour of the accumulated values along the ray. a and c denote the opacity and colour of applying the transfer function on the current voxel. When A_i approximates 1, the ray is terminated.

Chapter 3

Experiments

The experiments carried out were aimed at analysing the performance and visual quality of ray casting and texture based volume rendering. To have a baseline for reliable comparison it was necessary to develop the application for a desktop environment which supports CUDA as well as for the Shield mobile device. If an algorithm performed poorly than what was expected consistently across both platforms, it can be concluded that the algorithm or its implementation is inherently worse. But if an algorithm performed proportionally different across both platform then it can be concluded that the hardware is the impeding factor. In total four implementations were developed and tested:

1. Ray casting using shaders
2. Ray casting using CUDA
3. Textured based single-threaded
4. Texture based multi-threaded

The quality of the rendered image was measured purely by observation. The performance metrics that were measured included the frames per second (FPS).
The reason I chose to implement a texture based multi-threaded approach as well was to test whether the bottleneck in the application was in the application stage or the rasterization stage. If the work the CPU is alleviated by multi-threading the generation of the proxy geometry and the FPS increases while keeping the load on the GPU the same then this would be an indication that the bottleneck is on the application side.

The desktop implementation was used to see how optimisation changes in the code and the difference in the the performance of the algorithms was represented in relative terms to the mobile implementation. For example, if Textured based single-threaded was 2.5 times slower on the Shield and Texture based multi-threaded was still slower but only 1.5 times then this is an indication of better CPU utilisation.
The hardware specifications [Int13], [qua], [Klu14] of both platforms can be seen in Table 3.1. There are some interesting differences to note. Even though the number of cores is the same on both platforms, the desktop version can run 8 separate threads simultaneously. This has implication in the textured based multi-threaded approach as the desktop version will be able to run twice as many threads which compute the proxy plane generation. The other notable difference is the maximum theoretical performance. It is measured in FLOPS (FLoating point Operations per Second).[Dr.12] It takes into account the available parallelism in the microprocessor. It is useful to the extent of having an idea of the limits of GPU but it does not take into account the factors that affect a typical application like memory access, caching, etc. and other criteria which may vary significantly across different applications.
Special care was taken to ensure that the testing parameters were consistent for the four implementations on both devices. These include the following:

screen resolution The resolutions tested were the Shield native resolution 1920x1200 and a lower resolution of 500x500.

camera projection A perspective projection of 45 ° field of view.

camera position The camera position was fixed to a position which ensured that most of the screen space would cover the volume.

bounding box size The bounding box was fixed to 2 units cubed (i.e minimum vertex of [-1,-1,-1] and a maximum vertex of [1,1,1]) for all volumes.

opacity threshold Fixed at 100%. This is a number which control the maximum allowed saturation of each pixel. It also has an effect on early ray termination.

Parameter	Desktop	Mobile
Operating System	Windows 7	Android 5.0.1
CPU	Intel Xeon E3-1240	Tegra K1 ARMv8 Cortex-A15
Clock Speed	3.30 GHz	2.5 GHz
# of Cores	4	4
# of Threads	8	4
Cache Size	8MB	2MB
Instruction Set	64 bit	64 bit
Memory Bandwidth	21GB/s	?
Max Power Consumption	80W	2W
RAM	16GB	2GB
GPU	Nvidia Quadro K2000	Tegra K1 Kepler GPU
CUDA Cores	384	192
Clock Speed	954 MHz	900MHz
Memory Size	2GB GDDR5	2GB shared with CPU
Theoretical Performance	732 GFLOPS	365 GFLOPS
Memory Bandwidth	64GB/s	17GB/s
Max Power Consumption	51W	2W
OpenGL	4.3	4.4
CUDA	6.0	6.0

Table 3.1: The specifications used for testing on the desktop and mobile versions

maximum ray steps/number of slices It is the number of steps permitted before the algorithm is forced to terminate. It is an easy way to control the rendering speed at the expense of visual quality. For regions of the volume which are mostly or totally void of information(i.e close to 0% opacity) then the algorithm will mostly likely terminate due to hitting this number rather than the maximum opacity in the case of ray casting. When using the texture based approach, the number is always hit. I chose to factor in the diagonal length of the bounding box to calculate the number to account for viewing directions that would generate the longest rays: when the camera direction goes through one of the vertices and

the vertex on the opposite side on the diagonal. Hence the formula is:

$$maxRaySteps = N \times \frac{diagonal\,length}{axis\,length} \tag{3.1}$$

$$= N \times \frac{\sqrt{(-1-1)^2 + (-1-1)^2 + (-1-1)^2 + (-1-1)^2}}{2} \tag{3.2}$$

$$= N \times 1.73 \tag{3.3}$$

where N is the number of rays along one of the axis and the axis length is 2 because the calculation is made in object space and the length of one of the axis of the bounding box there is 2 (-1 to 1).

ray step distance/sampling rate It is the distance between each sampling point and the distance between each proxy plane in the ray casting and the texture based approaches respectively. It is calculated by dividing the diagonal length of the bounding box by the maximum number of rays.

number of threads The texture based multi-threaded approach parallelises the generation of the proxy geometry. Each thread is given an equal amount of proxy planes to calculate. The number of threads was fixed to be the number of cores available on the machine.

CUDA block size The block size is the number threads assigned to a block. With very little testing I found that allocating the maximum permissible number of *32x32=1024* gives the best performance. Each thread calculates the transport equation for a single pixel.

CUDA grid size The grid size is the number of blocks per grid. To extract the maximum level of parallelism the screen size is divided by the block size: $block\,size = 1920/32 \times 1200/32 = 60 \times 38 = 2280$.
The volume was made to spin on its y-axis with constant speed and constant camera position for 10 seconds for to achieve consistency for each test.
Separately, there were two volumes that were tested as listed in Table. 3.2. The voxel spacing parameter is ignored to stay consistent with the constant sampling rate. It describes the spacing between voxels where X is the distance between of the centers of the voxels

along the x-dimension, Y is the spacing in the y-dimension, and Z is the spacing in the z-dimension. [ITK14]. Rendered images of the volumes can be seen in the results section.
An important exclusion was testing of volumes which exceed the available memory space. The limited 2GB space of the Shield and the sharing of the memory between the CPU and the GPU means that volumes roughly the size of 500MB would barely fit in memory. The other 1GB is in use by the operating system and system daemons. In fact, when a volume is loaded on the Shield GPU it would occupy twice the memory on the same physical memory due to the shared memory. Usually bricking is the technique used to divide a volume which fits on the local RAM memory but not the GPU memory. Because of the shared memory, there isn't a reason for using bricking with the exception where streaming of the volume directly from the hard drive is implemented. This would add extra layer of complexity to the application and so it is out of the scope of the project.

Name	**Description**	**File Size**	**Resolution [x,y,z]**	**Voxel Spacing [x,y,z]**
CT Knee	A CT scan of a human knee	25 MB	[379, 229, 305]	[1.0, 1.0, 1.0]
Sphere	A hollow ball with interesting patterns	32 KB	[32, 32, 32]	[1.0, 1.0, 1.0]

Table 3.2: The volumes used in the project

Chapter 4

Implementation

As mentioned before, a significant effort was made to develop the application on two different platforms: Windows and Android. I was supplied with skeleton implementation of a ray casting volume renderer which I adapted to run on both systems.

4.1 High Level Overview

The application API abstracted the part of the code which were platform independent from those which were platform dependent. The code in the latter category needed to be developed separately on both systems. Figure 4.1 illustrates the components of the application. OpenGL Mathematics (GLM) was used as a helper library to perform the graphics maths calculations. [GLM15] Having two different platform to develop on means there are program components that have to be developed separately for each platform. Specific to this project: handling of events, display, loading of assets and performance monitoring were platform dependent.

4.1.1 Platform Dependent Code

Context/Window Manager

The context manager handles the creation and update of the display. On windows it is implemented using the GLFW library. [GLF15]

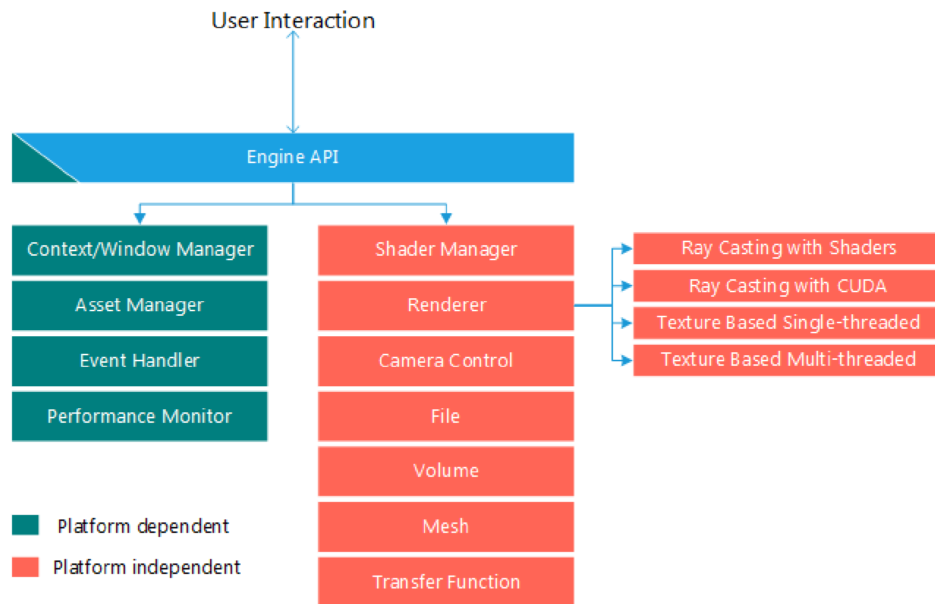


Figure 4.1: Architectural overview

Asset Management

The asset manager handles loading of files from the filesystem.

Event Handling

The event handler handles incoming events from the user as well as internal events.

Performance Monitoring

The performance monitor abstracts the collection of statistics such as memory usage, frames per second (FPS), global and local timers. For this project, only FPS and a global timer were implemented.

4.1.2 Platform Independent Code

Camera Control

The camera class controls how the projection and transformations of the viewpoint.

File

The file class represents any loaded file from the filesystem.

Volume

The volume class is a representation of a RAW file which is formatted as the internal representation for a 3D volume. It also handles the generation and usage of OpenGL textures to represent the volume on the GPU.

Transfer Function

The transfer function class is a representation of a Voreen XML transfer function.

Shader Manager

The shader manager handles the creation and usage of OpenGL shader programs.

Renderer

The renderer handles the rendering of the volume. It also abstracts the writing of uniform variables to the shaders. It has functionality to switch the rendering modes between the four implementations.

4.2 Ray Casting Using Shaders

The implementation of ray casting uses a single-pass shading. Single-pass means that all the calculations for a single pixel are made using a single shader program. On the other hand, a multi-pass volume renderer [KW03] will render front and back faces of a unit cube and rays are generated using textures rasterized in the first pass as lookups in the fragment shader. Calculations are made in model space because the sampling rate is specified relative to the size of the bounding box of the volume. The fragment shader is implemented as in Algorithm 1. One of the current disadvantages of this implementation is it would not render the volume when the camera is inside of the volume. It relies on all of the vertices of the bounding box to be infront of the camera even when OpenGL clipping is disabled. In such cases, clipping

could be integrated in the fragment shader which checks the position of the camera against the current pixel z position with a simple subtraction and if the result is negative then the current pixel is ”moved” to the position of the clip plane.
The vertex shader supplies the interpolated position of the vertices of the bounding box in model space to the fragment shader. The texture coordinate for the pixel is calculated from the current position of the sample along the ray. The position is in model space so to convert it to texture space the following formula is used:

$$texCoord = (position + 1.0) * 0.5 \tag{4.1}$$

giving a number in the range [0,1] assuming that the bounding box is specified between [-1,-1,-1] to [1,1,1] in model space.
The main reason for using OpenGL ES 3.0 is the native support for 3D textures which was not available in previous versions of the API. This means the volume texture can be supplied to the fragment shader in a straightforward manner without resorting to extra layer of complexity which converts 2D coordinates to 3D coordinates in the fragment shader. It is bound to the OpenGL texture buffer as would be the case in a the standard OpenGL implementation. Due to limitations of the OpenGL ES 3.0 API, the transfer function is represented a 2D texture rather a 1D texture as support for the latter does not exist. The second coordinate of the 2D texture is always of size 1 and when the texture is sampled in the fragment shader, the second coordinate is always 0.

4.3 Ray Casting Using CUDA

Harnessing the massive computational power of the GPU can be achieved through general purpose programming as well instead of shaders. The CUDA programming model allows the developer to apply their skills in C programming.
I adapted the sample CUDA implementation for volume rendering presented by [Nvi15] into the API I was building. I had to change a portion of the code so it would work on the Windows and the Android platform. Largely, it is the same as shown in Algorithm 1 with one notable exception. The position of the pixel is calculated based on the block and thread ids rather than interpolating the vertices which happens automatically in the shader implementation.

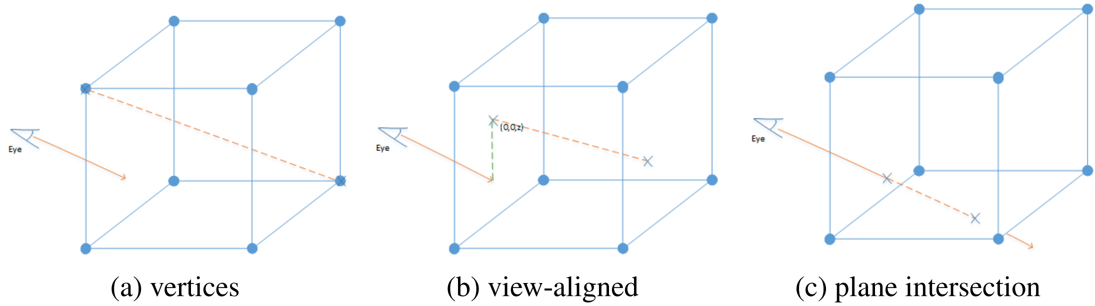
Preparing the 3D volume for use in CUDA takes several steps. The volume is loaded into a file and parsed by the Volume class. At this stage it is only represented as an array on the host side. The following procedure writes the raw volume from the application side to the CUDA memory space:

1. on the CUDA side, a pointer v is allocated with `cudaMalloc3DArray`
2. a *pitched pointer* is created using `cudaMallocPitch` [Nvi08c] and associated with v through the `cudaMemcpy3DParms` CUDA construct. A pitched pointer ensures that the memory it is pointing to is padded for irregular memory allocations to avoid bank conflicts where any two threads access different part of an array that spans multiple memory banks. `cudaMemcpy3DParms` is then used to make the actual copy from host memory to the CUDA shared memory space.
3. A CUDA `texture` construct is also created. Just like an OpenGL texture buffer object, the cuda texture is set to linear filtering and texture clamp instead of texture wrap.
4. Finally, the texture is bound the array v using `cudaBindTextureToArray`.

On the host, a pixel buffer is created the size of the screen. Crucially, `cudaGraphicsGLRegisterBuffer` is used to register the buffer object for access by CUDA. This way the output of the render kernel in CUDA can write directly to the pixel buffer allocated by OpenGL instead of writing to host then writing to OpenGL thereby saving memory bandwidth and time.
The CUDA render kernel does the ray casting calculations for each pixel and writes the results in the aforementioned pixel buffer. The information in this buffer still needs to be renderer through the graphics pipeline. For this matter, a very simple shader is invoked which renders a 2D plane directly without any model, view or projection transformations. The information in the pixel buffer is unpacked onto a texture buffer object. The fragment shader is supplied with this object to be draw onto the 2D plane as the final image.

4.4 Textured Based Single-threaded

The texture based algorithm works in view space. Alpha blending is enabled and configured for back-to-front blending. To achieve this, the OpenGL function `glBlendFunc(GLenum`



(a) vertices (b) view-aligned (c) plane intersection

Figure 4.2: Three ways of getting the minimum and maximum points

`sfactor, GLenum dfactor)` is called with parameters `sfactor=GL_SRC_ALPHA` and `dfactor=GL_ONE_MINUS_SRC_ALPHA` which translates to

$$C_{dest} = C_{src} + (1 - A_{src})C_{new} \tag{4.2}$$

$$A_{dest} = A_{src} + (1 - A_{src})A_{new} \tag{4.3}$$

where new corresponds to the current sample. Depth testing is disabled to let OpenGL correctly apply alpha blending.

The bounding box vertices and edges are transformed into view space. The next step is to get the minimum and maximum z intersections. There are three interpretations of this which I implemented and all three produce slightly different results when the volume is rendered. They are illustrated in Figure 4.2. I decided that intersection 4.2c gave the most accurate results based on observation of the rendered volume under angle. It is also the most computationally expensive method as it needs to find the exact intersections on the planes of the cube. I used the fast Möller-Trumbore algorithm for calculating ray-triangle intersection [MT05]. The advantage of this algorithm is that it does not need to compute the plane equation neither dynamic or precomptuted leading to CPU usage and memory savings. Algorithm 3 shows how the ray-cube intersection is performed. The Möller-Trumbore test return results in *uvt* coordinate space which is similar to barycentric coordinates. Lines 8-10 describe how to convert from uvt to 3D Cartesian space. *u* is the scale factor of the edge between v_1 and v_2, *v* is the scale factor of the edge between v_1 and v_3 and *t* is the distance from the origin to the intersection point. The conditions $u \leq 1$, $v \leq 1$ and $u + v \leq 1$ are fulfilled by the algorithm.

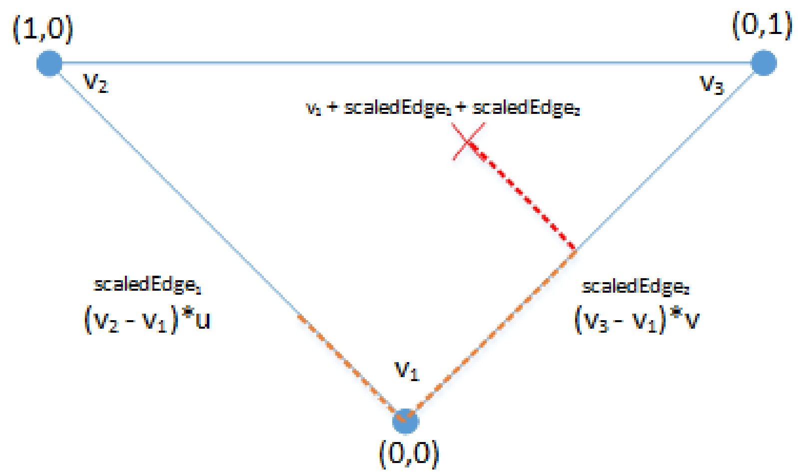


Figure 4.3: Converting from uvt to Cartesian coordinates

Algorithm 3: Ray-cube intersection test

```
1  bool getMinMaxPoints(cube triangles, Out: min, Out: max) begin
2      VEC3 origin = VEC3(0,0,0)
3      VEC3 dir = VEC3(0,0,-1)
4      list intersections;
5      for triangle in cube triangles:
6          VEC3 uvt
7          if intersect(triangle, origin, dir, Out: uvt):
8              VEC3 scaledEdge1 = (triangle.v2 - triangle.v1) * uvt.u
9              VEC3 scaledEdge2 = (triangle.v2 - triangle.v1) * uvt.v
10             VEC3 intersectionPt = triangle.v1 + (scaledEdge1 + scaledEdge2)
11             intersection.add(intersection)
12     if intersections.size not 2:
13         return false
14     distFromView1 = length(intersections[0])
15     distFromView2 = length(intersections[1])
16     if distFromView1 lessthan distFromView2:
17         min = intersections[0]
18         max = intersections[1]
19     else:
20         min = intersections[1]
21         max = intersections[0]
22     return true
```

The **for** loop in Algorithm 2 is computationally the most intensive part of the algorithm. For each sample, the intersections with the edges of the cube are calculated by testing the sample point against all the edges of the cube. Because the calculation is made in view space the following assumption can be made which simplifies the test:

> there is an intersection if the z coordinate of the sample point lies between the z coordinates of the two vertices of the edge being tested.

Sorting the points of the proxy plane around the centre involves iterating over all of the points and using insertion sort. The relation used for the sort algorithm is a *lessThan(a, b, centre)* operator which return *true* if *a* is smaller than *b*, *false* otherwise. It was adapted from [cia14].

This algorithm also makes the assumption that all the points lie on the same plane. In the case of this program, it is the x-y plane of the view.
Each of the generated vertices is written to a vertex buffer object in a triangle fan format and rendered using a simple 2D texture plane shader (the same way as with the CUDA implementation). OpenGL takes care of blending the proxy planes as long as they are rendered in a back-to-front order.

4.5 Texture Based Multi-threaded

Tracing the application side code of the textured based approach with Visual Studio's performance analyser quickly reveals that it spends 78.8% of its time in the proxy generation loop as shown in Figure 4.4. The whole loop is embarrassingly parallelisable, meaning that

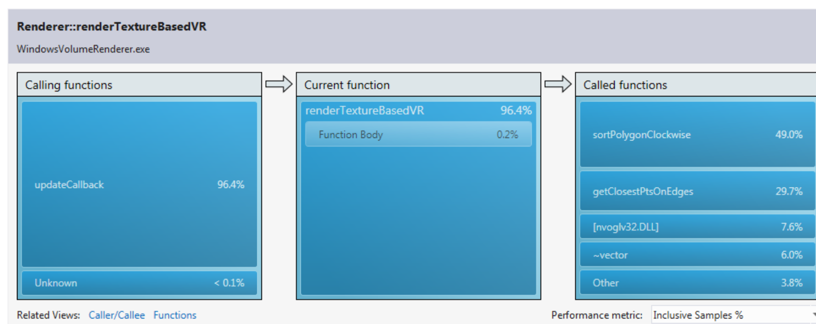


Figure 4.4: Percentage spent in the proxy generation loop

there is no data dependency whatsoever between any two samples in the loop. The code is open to be exploited using a multi-core implementation. In order to do so, the loop was split evenly among a set number of 4 threads. For example, if there are a 1000 proxy planes to be generated then each thread will compute 250. All the proxy planes are added to a list that has been resized beforehand to be as big as the number of samples so that each thread can ”slot in” a proxy plane in its position without having to explicitly use mutual exclusion on the list through a shared lock.
When performing the speed test comparison for the single-thread and the multi-threaded approaches it was revealed that there is no performance advantage when using multi-threading. This is an indication that the application side is not the bottleneck in the texture based approach.

Chapter 5

Results and Evaluation

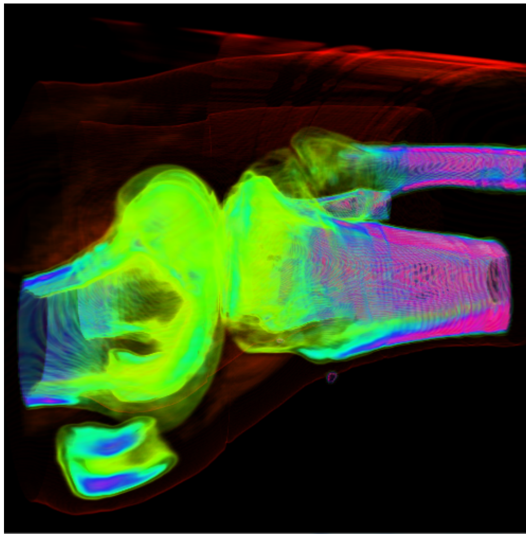
5.1 Visual Appearance

Figures 5.1, 5.2, 5.3 show the volume rendered at 500x500 resolution using the four different methods. Each volume has its own transfer function that highlight different regions of the volume and the transfer function is the same across the different methods.

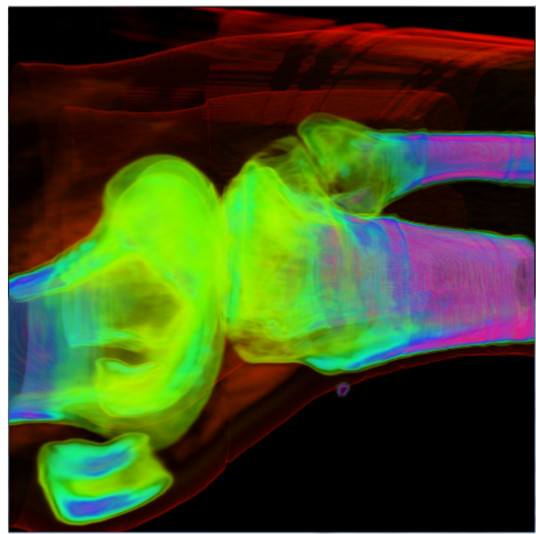
The two volumes display different visual qualities. Although larger in size, the CT Knee appears less dense than the Sphere volume. This is largely part to how the transfer function is specified and that is outside of the scope of the project. It controls which regions of the volume are considered dense by mapping transparency values to the density value stored in the individual voxels. The colour that is given to a particular density value does not affect the performance. But most importantly, the application fits volumes of all sizes within the same bounding box dimensions with the same sampling rate thus the volume size does not affect performance. Nevertheless, the size of the volume means the volume contains more information so intricate details,if desired, can be rendered more prominently by tinkering with the transfer function.

Figure 5.1 illustrates that the type of data itself and the shape of the geometry has an impact on which algorithm is better suited. The shader implementation produces a sharper, more contrasted image whereas the CUDA implementation produced a smoother image. Arguably, the CUDA CT Knee looks more appealing than the shader CT Knee and the reverse is true in the case of the Sphere. The CT Knee has a transfer function which produces regions with gradual transition in colours that is more suited to a smoother blending. On the other hand,

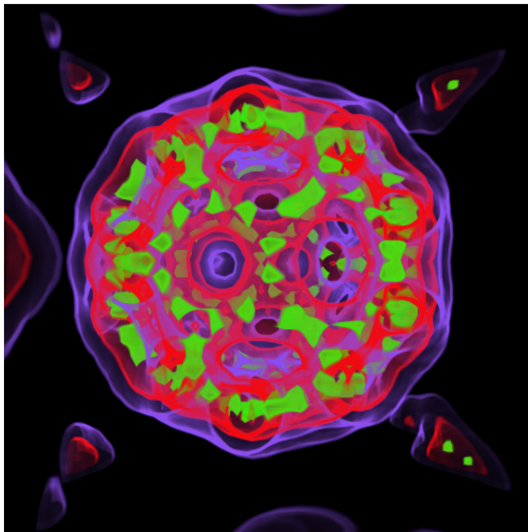
the transfer function of the Sphere produces regions with sharp transition in colours that is more suited to sharper rendering. In my opinion, the visual distinction between the both implementations comes from how CUDA and OpenGL handle linear texture interpolation with *cudaFilterModeLinear* and *GL_LINEAR* respectively.
Another interesting observation is that more samples do not necessarily produce better images. Again, it depends on interpretation and the intended effect. Comparing Figures 5.1and 5.2, using 500 samples on the CT Knee clearly produces a better outline of the knee joint. Looking at the Sphere, its transparency is lost when using higher samples. The overall transparency can be regulated by the opacity threshold which is set to 1.0 in all cases for performance testing purposes. But we can observe that the same effect can be achieved by lowering the sample rate thus increasing the performance.
The texture based approach in 5.3 produces less dense and arguably lower quality image for the same number of samples compared to the ray traced approach.



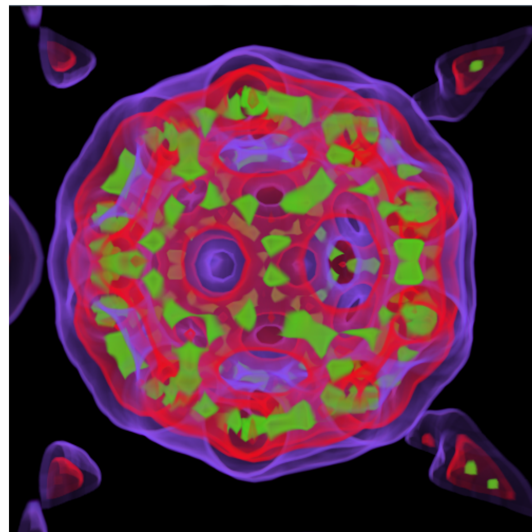
(a) CT Knee ray traced (shader)



(b) CT Knee ray traced (CUDA)

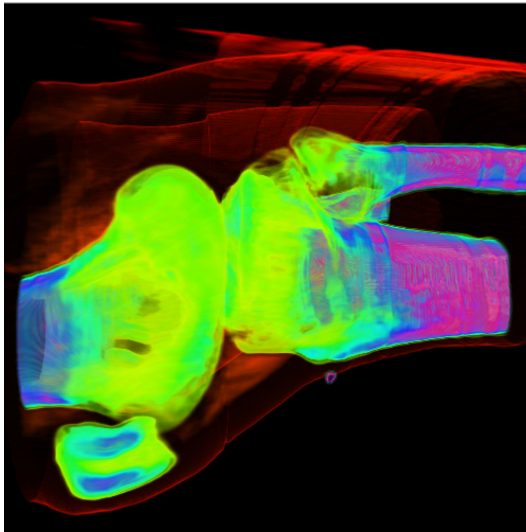


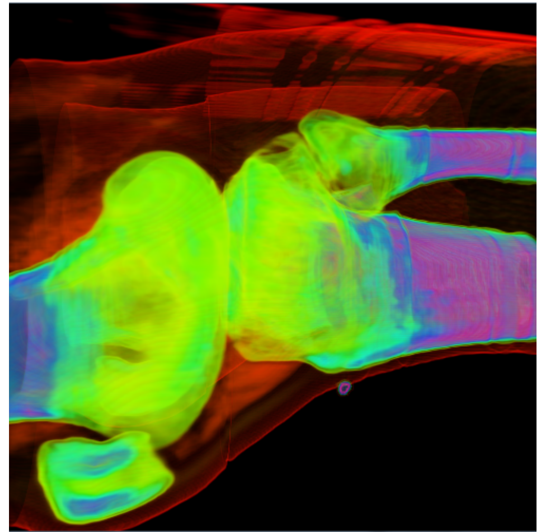
(c) Sphere ray traced (shader)

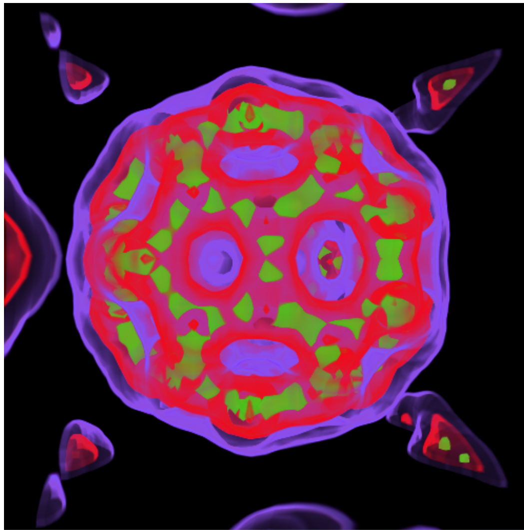


(d) Sphere ray traced (CUDA)

Figure 5.1: Comparison of ray traced using shaders vs CUDA with 250 samples


(a) CT Knee ray traced (shader)


(b) CT Knee ray traced (CUDA)


(c) Sphere ray traced (shader)

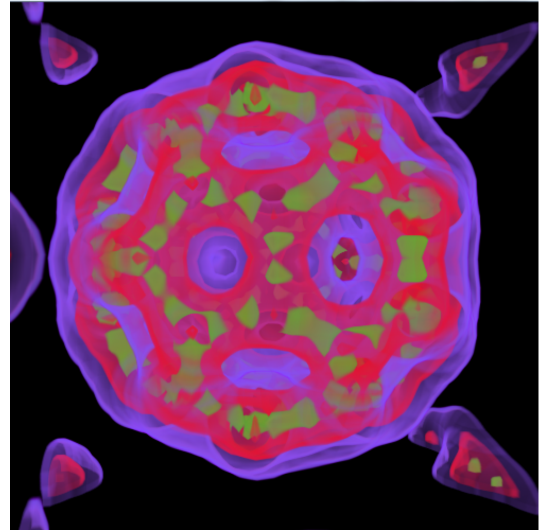
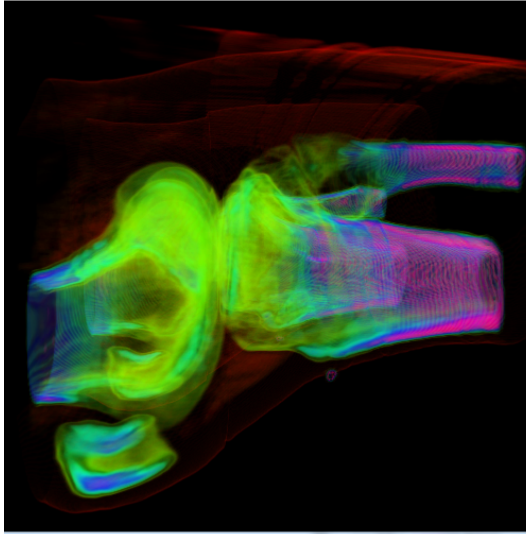
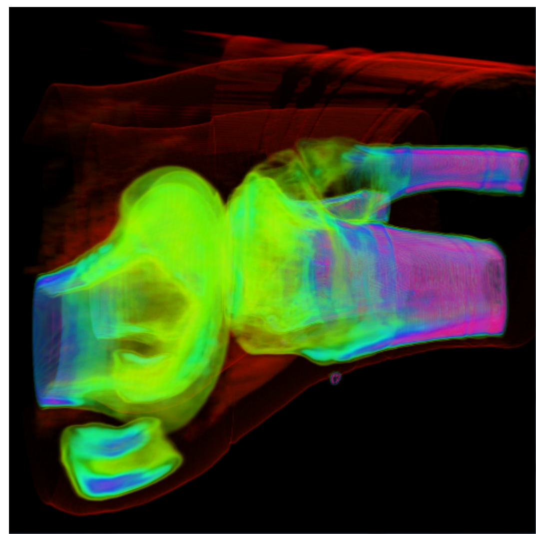

(d) Sphere ray traced (CUDA)

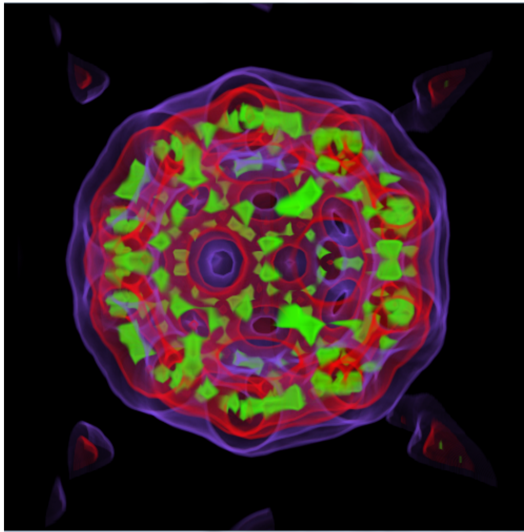
Figure 5.2: Comparison of ray traced using shaders vs CUDA with 500 samples



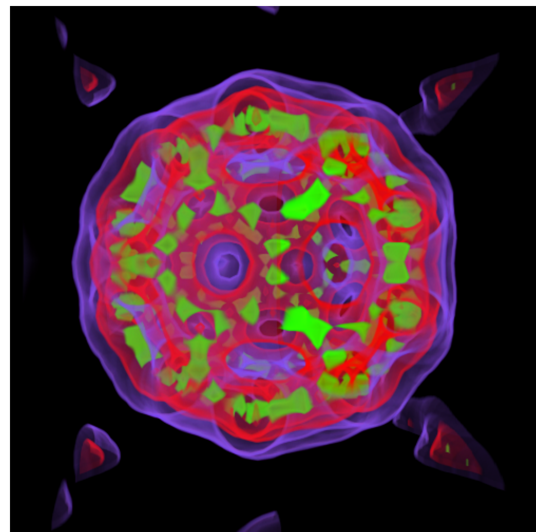
(a) CT Knee 250 samples



(b) CT Knee 500 samples



(c) Sphere 250 samples



(d) Sphere 500 samples

Figure 5.3: Comparison of textured based with 250 and 500 samples

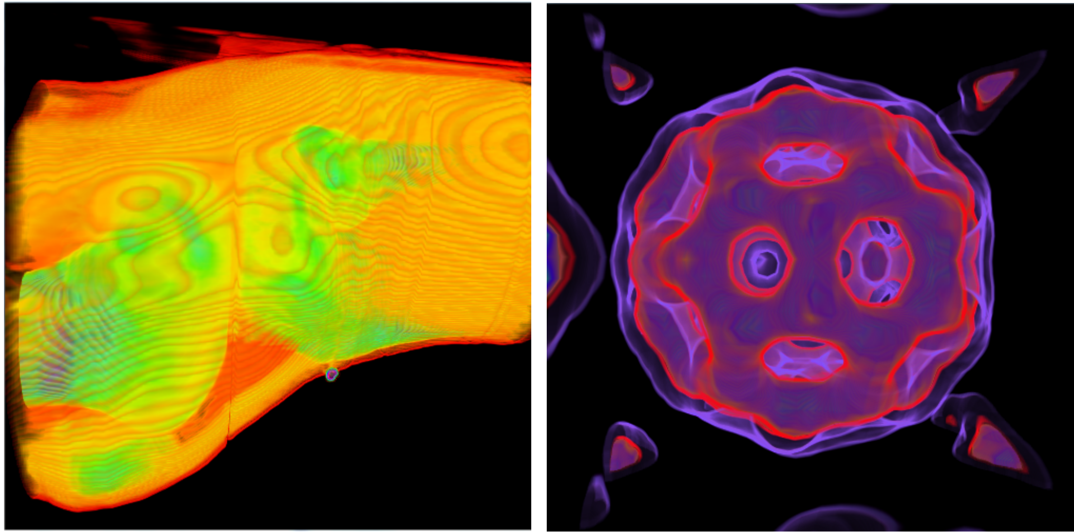


Figure 5.4: Using more opaque transfer function

5.2 Performance Tests

The transfer functions labelled "TF1" is the one used to produce the images in Figures 5.1, 5.2, 5.3. "TF2" produces more opaque images and the opacity threshold is hit faster than in "TF1". Output of how the volumes look with their respective "TF2" can be seen in Figure 5.4. "FPS" stands for frames per second.

5.2.1 Windows

Each algorithm was tested with 250 and 500 samples on two different resolutions with two different transfer functions as shown in Tables 5.1, 5.2, 5.3, 5.4. Note that there are 9.216 times more pixels to be drawn for the 1920x1200 resolution compared to the 500x500 resolution.

5.2.2 Android

Each algorithm was tested with 250 samples on two different resolutions with two different transfer functions 5.5, 5.6.

Algorithm	**FPS TF1 500x500**	**FPS TF1 1920x1200**	**FPS TF2 500x500**	**FPS TF2 1920x1200**
Ray Casting (Shader)	102	22	184	44
Ray Casting (CUDA)	39	8	39	8
Texture-Based Single-threaded	124	28	114	25
Texture-Based Multi-threaded	123	26	114	25

Table 5.1: Windows, CT Knee , 250 samples

Algorithm	**FPS TF1 500x500**	**FPS TF1 1920x1200**	**FPS TF2 500x500**	**FPS TF2 1920x1200**
Ray Casting (Shader)	62	14	120	29
Ray Casting (CUDA)	21	4	21	4
Texture-Based Single-threaded	62	14	60	13
Texture-Based Multi-threaded	61	14	60	13

Table 5.2: Windows, CT Knee , 500 samples

5.3 Evaluation

Several observations can be made based on the numbers gathered for the Windows and the Android version.

The transfer function does not affect the performance in the CUDA implementation. The transfer function has an effect on early ray termination as it changes the overall transparency of the volume. It would make sense that using *TF2* will be faster as it produces more opaque volume where the opacity threshold will be reached earlier along the ray. But if that is not the case as illustrated by the results gathered then one of the possible reasons is that the early ray termination test in the CUDA implementation does not work. The early ray

Algorithm	FPS TF1 500x500	FPS TF1 1920x1200	FPS TF2 500x500	FPS TF2 1920x1200
Ray Casting (Shader)	168	33	212	42
Ray Casting (CUDA)	52	9	52	10
Texture-Based Single-threaded	210	41	210	41
Texture-Based Multi-threaded	181	40	181	40

Table 5.3: Windows, Sphere, 250 samples

Algorithm	FPS TF1 500x500	FPS TF1 1920x1200	FPS TF2 500x500	FPS TF2 1920x1200
Ray Casting (Shader)	101	21	113	24
Ray Casting (CUDA)	26	5	26	5
Texture-Based Single-threaded	107	21	106	21
Texture-Based Multi-threaded	106	20	106	20

Table 5.4: Windows, Sphere, 500 samples

termination test can be described by the following algorithm:

```
1 if sample outside of bounding box OR
2 accumulated opacity bigger than opacity threshold:
3     stop sampling
```

Removing this check from the code reduces the performance as expected. For example, the performance of *Windows TF1 500x500 Ray Casting (CUDA)* drops from 39 to 31 FPS but it is the same performance drop when using *TF2*. This shows that the early ray termination does work but it is not a factor that contributes to the equivalence in performance of both transfer functions.

Algorithm	FPS TF1 500x500	FPS TF1 1920x1200	FPS TF2 500x500	FPS TF2 1920x1200
Ray Casting (Shader)	25	5	39	10
Ray Casting (CUDA)	10	2	10	2
Texture-Based Single-threaded	28	7	28	7
Texture-Based Multi-threaded	27	7	27	7

Table 5.5: Android, CT Knee , 250 samples

Algorithm	FPS TF1 500x500	FPS TF1 1920x1200	FPS TF2 500x500	FPS TF2 1920x1200
Ray Casting (Shader)	30	5	34	7
Ray Casting (CUDA)	12	2	12	2
Texture-Based Single-threaded	35	9	35	9
Texture-Based Multi-threaded	36	8	34	8

Table 5.6: Android, Sphere , 250 samples

The transfer function does not affect the performance of the textured based approach. This can be explained by the nature of the algorithm itself. It has to render all generated proxy planes regardless of how much of the plane surface actually contributes to the final image.

The Multi-threaded implementation does not perform faster than the single-threaded implementation. This suggests that the bottleneck is the rasterization stage rather than the application stage. Even at the lower 500x500 resolution both implementations perform similarly. Lowering the resolution even further thus switching from rasterization bottleneck to the application bottleneck does not "help" the multi-threaded outperform the single-threaded implementation. This suggests that the overhead of creating and maintaining the threads is higher than the boost of performance they provide. The current number of threads is fixed

to 4. If the workload of each thread is not high to overcome the overhead its brings then either lowering the thread count or increasing the number of samples will have a positive effect on the performance compared to the single-threaded implementation. On the contrary, it may be the case that not enough parallelism is extracted from the CPU thus increasing the thread count should increase performance. Table 5.7 shows that increasing the thread count actually lowers the performance. Further testing is needed to evaluate whether this approach

No. Threads	**1**	**2**	**4**	**8**	**16**
FPS	124	124	124	114	87

Table 5.7: Multithreading: Windows, CT Knee, 250 samples

has any merit at all or the implementation itself needs to be optimised.

The CUDA implementation perform significantly slower than its shader equivalent. This was unexpected given that both implementations use the same algorithm with the same parameters. Further testing revealed (using the NSight Performance Analyser [NVI14c]) that during runtime for the shader implementation 69% is spent performing the *SwapBuffers* function. In the CUDA implementation, 68% is spent on the *cudaMemset*, 25% on the *cudaGraphicsGLRegisterBuffer* and 14% on the *cudaGraphicsMapResources* functions. The latter two cannot be changed because they perform the necessary mapping of resources between CUDA and OpenGL each frame. *cudaMemset* is optional but highly desirable because it acts as the OpenGL equivalent *clearColor*. Removing the call to the function at each frame does not increase the FPS suggesting that the analyzer does not account for time spent rendering in the CUDA kernel.

The next step was testing the performance impact of texture fetching in the shader against CUDA implementations using *Windows CT Knee 500x500 250 samples* as an example. A simple test is removing all texture fetches and observe the rendering speed. The shader implementation became 4.25 times faster (from 102 to 434 FPS) and the CUDA implementation became 8.07 times faster (from 39 to 315 FPS). This suggests that, as currently implemented, the texture fetching in CUDA is slower than in a shader. For each loop iteration there is one fetch to a 3D texture and one fetch to a 1D texture. There is no way that the algorithm will perform correctly without either one.

So the only other way to improve texture fetching on CUDA is to investigate the memory layout and see the volume and the transfer function can be stored in a memory space. Cur-

rently, both textures are stored in texture memory. It is a constant memory with read-only access that is optimised for data locality and is cached in the texture cache.[LDD+12] But if cached, global memory has a higher bandwidth. From the speedup of disabling texture fetching, it can be concluded that the CUDA implementation is memory-bound.[LDD+12] presents optimization strategies for compute- and memory-bound algorithm for CUDA. According to this paper, low latency memory caches can be better utilised by reorganising the data into self-contained data structures and using multi-pass approach to process a subset of these self-contained data structures during each pass. In the context of volume rendering, this can be achieved by either bricking or spatial data structures.
Another consideration is the performance decrease when changing the number of samples. CUDA suffers worse performance hit when increasing the number of samples as illustrated in Figure 5.5.

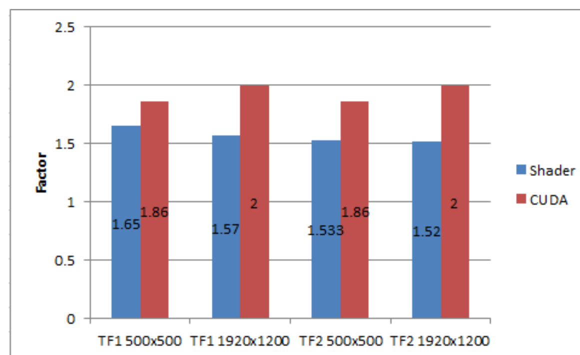
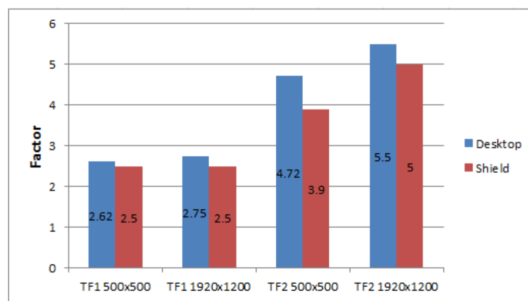


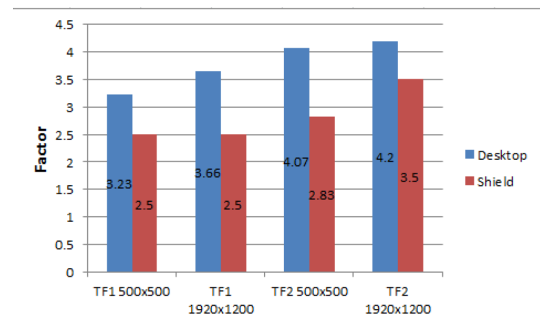
Figure 5.5: Performance decrease of the shader versus the CUDA implementation when changing from 250 to 500 samples

The ray casting shader implementation outperforms the texture based implementation for opaque volumes. The ray casting approach is affected by early ray termination whereas the textured based approach is not.

The performance difference between CUDA and ray casting is smaller on the Shield than on the desktop version. This is illustrated in Figure 5.6. An in-depth investigation of the two chosen hardware specifications is required to make a sound conclusion why this is the case.



(a) CT Knee



(b) Sphere

Figure 5.6: Performance difference of ray casting using shaders over CUDA

Chapter 6

Conclusion

The main goal of developing an interactive volume renderer has been achieved. A cross-platform OpenGL ES 3.0 volume renderer was developed. Moreover, comparison of ray casting versus texture based approaches was presented. It was shown that textured based rendering produces lower quality image and is faster for volumes exhibiting higher transparency. Ray casting produces higher quality visuals and is faster for opaque volumes. The current implementation of CUDA is significantly slower than its shader equivalent due to OpenGL shader programming still being computationally faster than CUDA when performing the same algorithm.

The biggest challenge was working with CUDA on the Shield. The development hurdles with the Linux environment were a serious detriment throughout the project.

Future work might include integrating a user interface for a fully-fledged application. This would merit the application for use in practice. Integration of the octree spatial data structure to do empty region skipping would be the next logical step as it has been proven a viable way of improving the traversal of the volume. Also, not all of the initial goals were met. An investigation into how each algorithm suits the Shield hardware architecture would have given insight on how to optimise specifically for that architecture in order to gain the most performance. Possible areas of investigations would be cache utilisation and measuring the power consumption for each of the implemented algorithms.

Bibliography

[cia14] ciamej. Sort points in clockwise order? `http://stackoverflow.com/a/6989383`, 2014.

[CSK+11] John Congote, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar Ruiz. Interactive Visualization of Volumetric Data with WebGL in Real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 137–146, New York, NY, USA, 2011. ACM.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, June 1988.

[Dr.12] Dr. Fernandez, Mark. Nodes, Sockets, Cores and FLOPS, Oh, My. `http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329`, 2012.

[Ecl10] Eclipse Contributors. Symbol could not be resolved errors. `https://www.eclipse.org/forums/index.php/t/550462/`, 2010.

[fer] GPU gems: programming techniques, tips, and tricks for real-time graphics, author=Fernando, Randima and Haines, Eric and Sweeney, Tim, journal=Dimensions, volume=7, number=4, pages=816, year=2001, chapter=39.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[GLF15] GLFW Contributors. GLFW is an Open Source, multi-platform library. `https://www.eclipse.org/forums/index.php/t/550462/`, 2015.

[GLM15] GLM Contributors. OpenGL Mathematics, 2015.

[Int13] Intel Contributors. Intel Xeon Processor E3-1240 (8M Cache, 3.30 GHz) Specifications. http://ark.intel.com/products/52273/Intel-Xeon-Processor-E3-1240-8M-Cache-3_30-GHz, 2013.

[ITK14] ITK Contributors. ITK/MetaIO/Documentation. http://www.itk.org/Wiki/ITK/MetaIO/Documentation, 2014.

[Klu14] Klug, Brian and Lal Shimpi, Anand. NVIDIA Tegra K1 Preview & Architecture Analysis. http://www.anandtech.com/show/7622/nvidia-tegra-k1, 2014.

[KW03] Jens Kruger and Rüdiger Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[LDD+12] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W Toga. CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer methods and programs in biomedicine*, 106(3):175–187, 2012.

[Lev88] Marc Levoy. Display of surfaces from volume data. *Computer Graphics and Applications, IEEE*, 8(3):29–37, 1988.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Ieee Micro*, 28(2):39–55, 2008.

[Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[MF12] M.M. Mobeen and Lin Feng. Ubiquitous medical volume rendering on mobile devices. In *Information Society (i-Society), 2012 International Conference on*, pages 93–98, June 2012.

[MT05] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[MW08] Manuel Moser and Daniel Weiskopf. Interactive volume rendering on mobile devices. 2008.

[NJOS12] José M Noguera, Juan-Roberto Jiménez, Carlos J Ogáyar, and Rafael Jesús Segura. Volume Rendering Strategies on Mobile Devices. In *GRAPP/IVAPP*, pages 447–452, 2012.

[Nvi08a] Nvidia. What is CUDA? http://www.nvidia.com/object/cuda_home_new.html, 2008.

[Nvi08b] CUDA Nvidia. Programming guide, 2008.

[Nvi08c] Nvidia Contributors. Memory Management. http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__MEMORY_g80d689bc903792f906e49be4a0b6d8db.html, 2008.

[Nvi12] Nvidia. Kepler GK110 Whitepaper. 2012.

[Nvi14a] Nvidia. NVIDIA Tegra K1 Whitepaper. 2014.

[Nvi14b] Nvidia. Tesla K80 GPU Accelerator Board Specifications. 2014.

[NVI14c] NVIDIA Contributors. Analysis Tools. http://docs.nvidia.com/nsight-visual-studio-edition/4.0/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis_Tools_Overview.htm%3FTocPath%3DAnalysis%20Tools|______0, 2014.

[Nvi15] Nvidia Contributors. CUDA Samples. http://docs.nvidia.com/cuda/cuda-samples/#volume-rendering-with-3d-textures, 2015.

[qua] NVIDIA Quadro K2000. http://www.techpowerup.com/gpudb/1838/quadro-k2000.html.

[RA12] Marcos Balsa Rodríguez and Pere Pau Vázquez Alcocer. Practical Volume Rendering in Mobile Devices. In *Advances in Visual Computing*, pages 708–718. Springer, 2012.

[RV06] Daniel Ruijters and Anna Vilanova. Optimizing GPU volume rendering. 2006.

[Smi12] Smith, Ryan. Khronos Announces OpenGL ES 3.0, OpenGL 4.3, ASTC Texture Compression, & CLU. http://www.anandtech.com/show/6134/khronos-announces-opengl-es-30-opengl-43-astc-texture-compres 2, 2012.

[Tho65] James E. Thornton. Parallel Operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, New York, NY, USA, 1965. ACM.

[WW92] Alan Watt and Mark Watt. *Advanced Animation And Rendering Techniques*. 1992.