Graz University of Technology
Institute for Computer Graphics and Vision

Master's Thesis

Evaluation of the Softshell GPU execution model for real-time rendering and image processing applications

Philip Voglreiter
Graz, Austria, January 2013

Advisor
Prof. Dr. Dieter Schmalstieg
Institute for Computer Graphics and Vision, Graz University of Technology

Referee
Dr. Bernhard Kainz
Institute for Computer Graphics and Vision, Graz University of Technology

I want to thank my parents for making my studies
possible at Graz University of Technology.
Furthermore, I want to thank my girlfriend for bearing
with me despite the little time I could spend with her
while I was working on this thesis.
Finally, I want to thank Bernhard Kainz for the
patience he had in supervising all of my projects over
the past four years.

Some days you're the pigeon, some days you're the statue.

Bruce Dickinson

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, Austria	**January 31, 2013**	
Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, Österreich	**31. Jänner, 2013**	
Ort	Datum	Unterschrift

Abstract

The recent ascent of parallel programming languages, in combination with the wide-spread availability of powerful graphics hardware, inspires the development of new programming paradigms. In this work we evaluate *Softshell*, which currently belongs to the most advanced parallel programming models. We select real-world applications and adapt them for this Application Programming Interface (API). The evaluated algorithms arise from the General Purpose Computation on the Graphics Programming Unit (GPGPU) field. In GPGPU, affordable and simultaneously powerful GPUs act as massively parallel co-processors in a consumer Personal Computer (PC). NVidia's Compute Unified Device Architecture (CUDA) currently represents the most flexible GPGPU architecture available. *Softshell* extends this flexibility further by immediately setting up on top of CUDA and extending its capabilities via a three-tier scheduling model. Synthetic test cases for *Softshell* already show excellent performance. In this work, we benchmark *Softshell's* capability to improve performance and flexibility of *"Ray Casting With Advanced Illumination"* and *"Particle Based Volume Rendering"*. These algorithms are very well suited due to their vastly different requirements in terms of dynamic scheduling strategies. Furthermore, we discuss *"Dykstra's projection algorithm"* as an example where classical parallelization as well as *Softshell* are incapable of providing significant performance improvements over the linear counterpart.

Keywords. GPU programming, Softshell, CUDA, volume rendering, volume illumination, total variation, image processing

Kurzfassung

Durch die Kombination aus den jüngsten Entwicklungen im Bereich der parallelen Programmiersprachen und die weite Verbreitung von leistungsfähiger, parallel rechnender Hardware entstehen immer innovativere Programmiermodelle. In dieser Arbeit werden wir *Softshell*, derzeit eines der fortgeschrittensten Modelle dieser Art, auf seine Leistungsfähigkeit hin untersuchen. Synthetische Tests haben bereits die Vorteile von *Softshell* aufgezeigt. Wir wenden *Softshell* jedoch auf verschiedene Beispiele aus dem Bereich "General Purpose Computing on the Graphics Programming Unit" (GPGPU) an, um Erkenntnisse in einer realitätsnahen Umgebung zu gewinnen. Im Bereich GPGPU werden Grafikprozessoren als stark parallelisierte Koprozessoren eingesetzt. Derzeit bietet NVidia's "Compute Unified Device Architecture" (CUDA) das flexibelste "Application Programming Interface" (API) für parallele Programmierung. *Softshell*, welches auf CUDA aufbaut, bietet ein dreistufiges scheduling Modell. In dieser Arbeit evaluieren wir die Leistung des Modells anhand der Algorithmen *"Ray Casting With Advanced Illumination"* und *"Particle-Based Volume Rendering"*. Diese Algorithmen eignen sich hervorragend in diesem Kontext durch ihre drastisch unterschiedlichen Anforderungen an dynamisches scheduling. Weiters behandeln wir *"Dykstra's projection algorithm"* als Beispiel dafür, dass Algorithmen, welche nur auf den ersten Blick effizient parallelisierbar sind, auch durch *Softshell* nicht zwingend effizienter arbeiten.

Acknowledgments

This work was partly funded by the Austrian Science Fund (FWF): P23329 and by the European Union in FP7 VPH initiative under contract numbers 600641 (Go-Smart) and 223877 (IMPPACT).

Contents

List of Figures

Part I

Overview

Chapter 1

Introduction & Motivation

Facilitating GPUs as co-processors in common PCs is on a steady rise. Given their raw parallel computing power, this is no surprise at all. Further, GPUs tend to surpass Moore's Law, which states that computing power (measured in FLOPs: floating point operations per second) of central processing units (CPU) roughly doubles every eighteen months. The development of the computational model as well as increase in computational power and memory capabilities over the last years provide a steady stream of new possibilities to exploit the features of GPUs. Proportional to the influx of fresh algorithms and techniques, which benefit from massively parallel computing, the demand for more sophisticated and dynamically self-regulating execution models grows as well. Oppositely, the state of the art execution model for GPU applications comprises of rather rigid structures. Ongoing research shows promising new ways to increase not only flexibility, but also overall computational performance in this field. Unfortunately, implementations of recent models can only build on top of present APIs and often suffer from the lack of the required direct hardware support. Still, although optimization is only executed on a software-only-level, recent results are very promising and might shift the prevalent parallel computing paradigms.

In this thesis, we cover two important topics related to advanced GPU computation methods. Based on the Softshell [46] framework, our first contribution is an attempt of describing policies for modifications to exploit the capabilities of the framework at the best possible rate. We will do so by combining theoretical methods with detailed examples, taken from the areas of computer graphics and vision. The second contribution of the thesis combines the application of the proposed methods to realistic datasets with benchmarking versus their plain CUDA counterparts. Previously, only results highlighting the advantages over CPU implementations were shown in the original work.

Before we give a detailed description of the algorithms we evaluate, we provide a step-wise introduction into the GPU computing model in general in the following sections. We provide an overview tackling recent developments and we will conclude this section by showing up possible problems when using the current model.

1.1 GPU Computing Background

1.1.1 GPGPU

During the last years, graphic processing units have become more and more powerful in terms of computational capabilities. Until recently, the video games industry was practically the only - but still quite successful - driving force behind those advances. Due to the main application area of GPUs, computing possibly billions of operations on fragments or vertices per second, they have not only developed in terms of raw computing power. Naturally, the operations required from a GPU show only minimal data interdependence between primitives to attend to. Hence, parallel execution suggests itself as conflicting situations while storing data rarely, if at all, occur. Indeed, GPUs of earlier generations were massively parallel processors chained to a static, single-purpose processing pipeline. Over time, this static structure was weakened during several iterations.

The first step towards making the computational power available outside of the predetermined, standard graphic operations was the introduction of programmable shaders. They provide execution of user-defined operations on the same graphic primitives as used before (geometry, vertices, fragments). Although, this shows signs of meaningful progress towards programmability of the graphics pipeline, it still only provides entry points into the rigid graphics pipeline whose circumvention at this point was not an option. The general environment, optimized for processing graphics primitives, remains intact and huge parts of the pipeline remain basically unchanged. Nevertheless, some groups of researchers [7] managed to transform algorithms to fit into the fixed function graphics pipeline by using programmable shaders. Sophisticated translation schemes allowed to transform algorithms to facilitate the available memory and computation structures. As the details of how those models are actually implemented in hardware vary a lot depending on the GPU manufacturer, we will only describe the architecture relevant to this work in Section 1.1.2.

Modern General GPGPU interfaces go even further. They offer direct access to the powerful parallel processing units on the hardware via drivers and APIs. Contrary to shader programming, this approach is not restricted to certain entry points into the rendering pipeline. In fact, access to the processors on the GPU is immediate. Moreover, memory management is not only restricted to structures used in graphics applications. Instead, structuring may happen very similar to CPU programming. Consecutively, a plethora of algorithms may be executed in parallel on the GPU. Still, on top of the plentiful benefits of using parallel GPU hardware, we also need to discuss the restrictions caused by its special heritage.

The basic requirements for GPUs have always been: executing homogeneous operations on heterogeneous data. Rendering geometry, for example, requires the application of the same transform matrix onto thousands, or even millions of vertices. Another example for this kind of parallelism is alpha blending of textures, where each fragment is constructed from several lookups on textures. This programming model is called single instruction, multiple data (SIMD). The hardware does not provide the ability to perform heterogeneous tasks in parallel. Closely related are two terms describing memory access patterns. “Scatter” denotes arbitrary write locations during execution. For example, vertex operations often alter the location of vertices. “Gather” refers to reading data from arbitrary memory locations. This pattern is highly preferable since coherency measures rarely have to be imposed. An example of such an operation is image filtering, where information from several pixels is condensed into a single result pixel.

The applicability of the SIMD model to an algorithm is closely related to so called “data parallelism”. Data parallelism, often also referred to as concurrency, describes data independence between single threads of execution. As all threads in a SIMD environment execute the same operation, there is the possibility of interference during write operations to the same memory locations. For multi-core CPUs, locking mechanisms such as mutual exclusion (mutex) objects normally resolve this problem. But in a massively parallel environment, this is an unacceptable method. Considering a situation where possibly several thousands or even millions of threads have to wait for permission on writing to the same memory location. Thereby, the behavior of the program becomes linear as only one thread is allowed to store data in a certain memory location at a time. Instead, it is desirable to identify possibilities to alter existing algorithms in a way that interdependencies vanish. Each thread in the parallel environment needs to write to a different location in memory, effectively removing the aforementioned race conditions and achieving true data parallelism.

This computing paradigm leads to a rather rigid programming model as to circumvent interdependencies. Also, the requirements to the GPU architecture itself restricts the manifoldness of possible computing models and universality of the used hardware.

Nevertheless, GPGPU experienced an enormous increase in popularity during the last years. This ascent in interest demands innovative concepts to catch up with the advanced computing models and comfort of standard CPU programming, which had time to develop over the last decades.

1.1.2 Cuda Basics

A very prominent programming language for GPGPU is NVidia’s CUDA [10]. This language introduces capabilities for accessing the GPU as a co-processor via extensions to standard programming languages such as C or C++.

1.1.2.1 GPU Architecture

Early GPUs facilitated specialized hardware to embrace operations in the fixed-function Graphics Pipeline. Each step of the pipeline was executed by single-purpose hardware specialized on execution of one of the main rendering tasks: rasterization, clipping, etc.. Due to high demand for more flexibility, liberalization of the static structure was inevitable. Programmable shaders, which are functions restricted to graphics primitives, were introduced. They provide entry points between stages in the so far rigid rendering pipeline and allow including post processing effects on generated fragments and vertices. The involved rendering pipeline is depicted in Figure 1.1.

Figure 1.1: Standard OpenGL rendering pipeline. Blue boxes show programmable shader stages. Yellow boxes indicate fixed-function stages. Picture taken from [35]

With the introduction of the GeForce 8 series, paradigms of GPU design have changed. The fixed function hardware has been replaced by multi-purpose processors, the so-called streaming multiprocessors (SMP). Each GPU operates several of those SMPs, each of which actually performs parallel computation by facilitating several CUDA processors. The CUDA processors operate in single instruction, multiple data (SIMD) mode. This means that each CUDA processor within an SMP either executes the same operation as all others, or is idle until the control flow of the executed program again enables concurrency. Switching from the fixed-function render hardware to the new SMP model also induces adaption of the render pipeline. The newly introduced model is depicted in Figure 1.2. The ability to execute each single step involved in the render pipeline implicates the potential to conduct not only heterogeneous, but even arbitrary functions. Continuing the liberation of programmability of the graphics hardware by providing a completely open interface has led to what we now refer to as GPGPU.

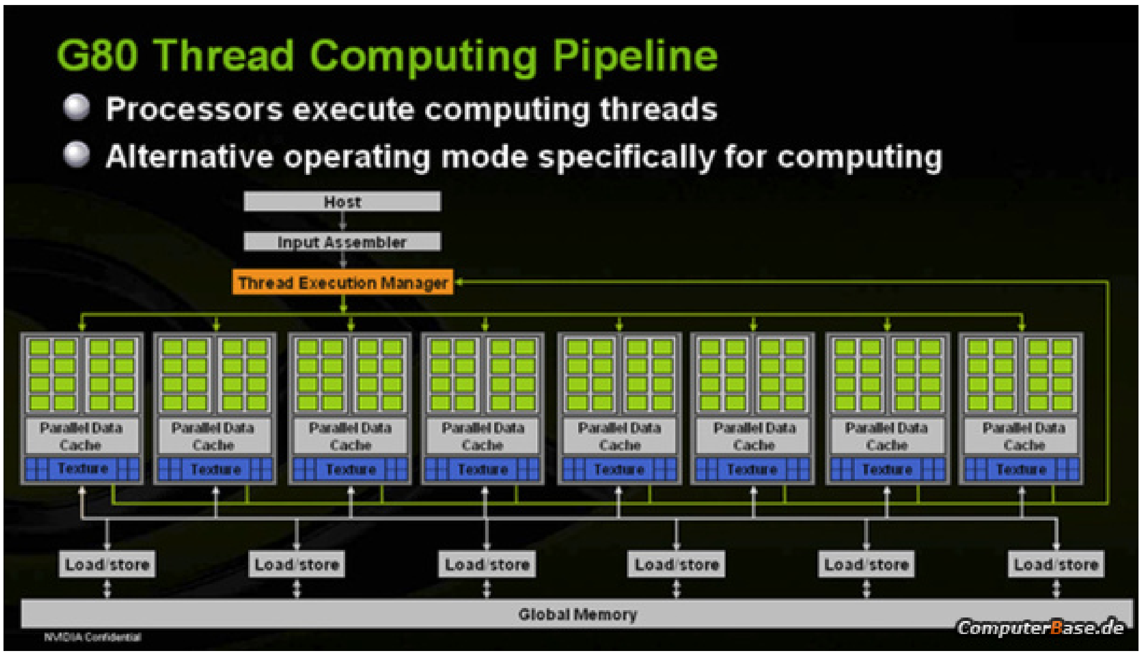


Figure 1.2: Overhauled render pipeline using multi-purpose processors (boxes with gray frames) instead of fixed-function hardware. Image taken from [17]

1.1.2.2 Memory Model

In this section we will discuss, which layers of memory the CUDA architecture provides. Further, we elaborate the scope of accessibility. Before we go into detail concerning GPU memory, we need to consider the general memory structure when using the GPU as a co-processor in a CPU-based program. Besides the memory available to the CPU, normally referred to as RAM, we may now also access an additional address space on the GPU. From now on, we will refer to the RAM on the CPU as host memory. The synonym we will use for memory on the GPU is "device memory". This gets more clear when considering the GPU as a co-processor (an additional “device”) performing tasks scheduled by the CPU (the “host”). The two mentioned address spaces are strictly separated. Consistently transferring small portions between these address spaces can quickly become a performance bottleneck due to the limited transfer bandwidth and speed. To circumvent this shortcoming, memory is copied in large chunks between host memory and device memory whenever possible. As a further consequence, neither host nor device may directly access memory on their counterpart. We will now proceed with a description of the individual types of GPU related memory at our disposal.

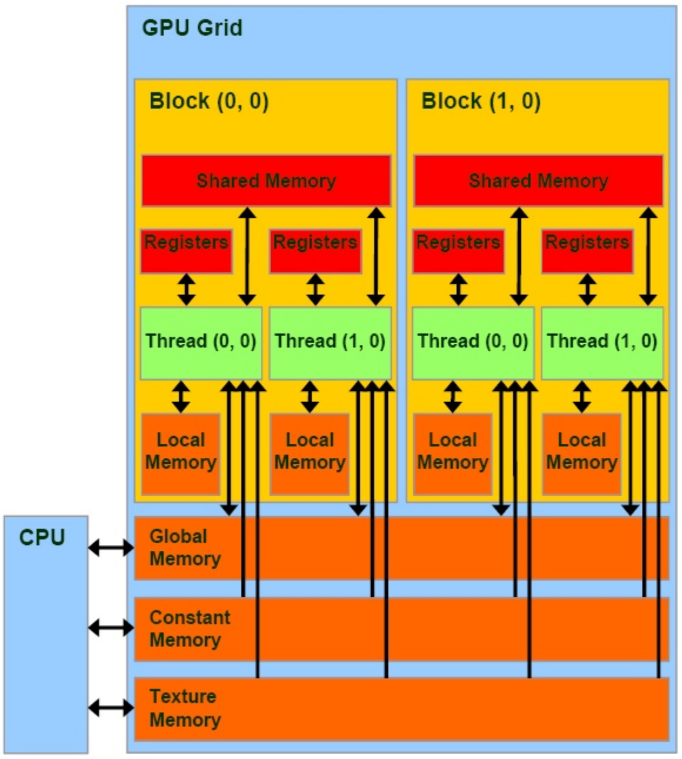
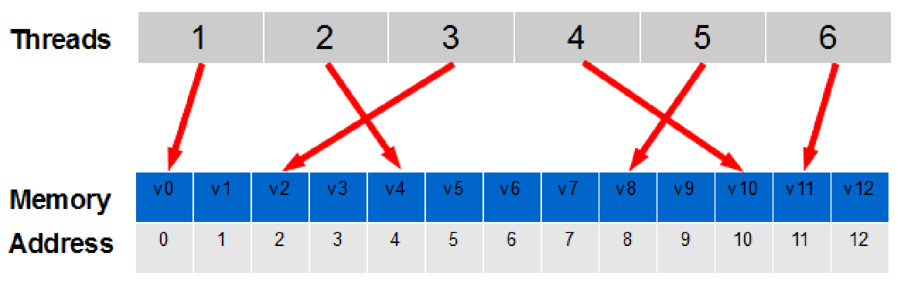


Figure 1.3: Overview over the CUDA memory architecture. The arrows indicate data transfer directions. One-sided arrows indicate read-only functionality. Picture taken from [52]

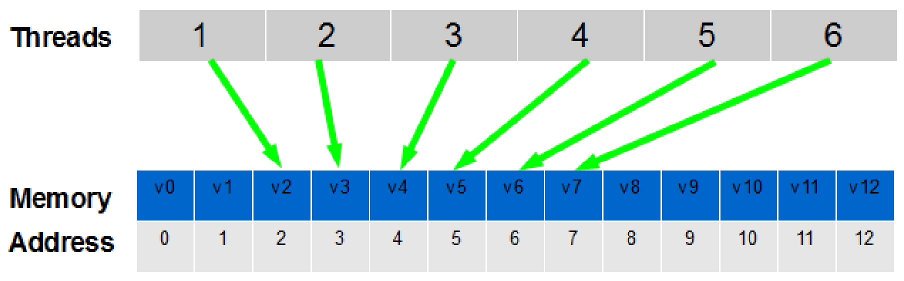
1.1.2.3 Memory In Global Scope

Although the GPU facilitates several multiprocessors, providing large, global data spaces to store data to operate on is an obvious requirement. Accessing memory of this type, called global memory, is generally considered to be rather slow compared to memory portions related more closely to the single processors. As a consequence, CUDA programmers try to avoid large amounts of global memory accesses and instead try to fall back to other scopes. Frequently used data should, if the size restrictions allow for it, be loaded into faster memory scopes. The size of globally accessible memory ranges from up to 512 megabytes in the first CUDA-capable GPU series, the GeForce 8800 GTX until up to four gigabytes in the recent GeForce 600 series.

Global Memory The global memory block is the main location when it comes to storing considerable amounts of data for read and write access. Although NVIDIA introduced caching for global memory with compute compatibility 2, hiding the latency of memory access still happens first and foremost via achieving a high occupancy. There are two caching layers in total. All operations concerning the global memory operate on an always available L2 cache. Additionally, the shared memory (see section 1.1.2.4) can be configured to act as an additional L1 cache. Still, fast context switching provides a great mechanism to hide the rather high latency of global memory access, which is somewhere between 400 and 800 clock cycles. Concerning the access patterns, it is highly advised to transform memory accesses into coalesced patterns. Coalescence induces that concurrently executed threads, all of which perform a memory read or write at the same time, access consecutive addresses in memory. Figure 1.4 provides a visualization of both types of access patterns.



(a) Non-coalesced memory access



(b) Coalesced memory access

Figure 1.4: Figures showing the difference between coalesced and non-coalesced memory access for one concurrent memory operation. Figure 1.4(a) shows a non-coalesced access pattern of a single instruction. The addresses are non-consecutive, and thereby decrease performance of memory transfers. Figure 1.4(b) shows the exact opposite. Addresses are aligned in a way which supports a high throughput.

Constant Memory The name of this part of the memory already suggests its properties. Once values have been set from the CPU, this portion of the memory solely provides read-only access during execution on the GPU. Constant memory aims at storing variables, which remain constant throughout computation, and therefor only comprises of 64 kb, but has 8 kb of cache at its disposal.

Texture Memory Although located in global memory, texture memory offers distinguished features. It specializes on spatial locality rather than memory locality. That is, the cache for texture memory works different than the normal global memory cache does. The most outstanding feature of texture caches is automatic interpolation in hardware when fetching data. Further, texture memory provides automatic normalization of data, as well as built-in boundary treatment. Texture memory was meant to be mostly read-only, although it is possible to write to it using sophisticated methods we do not cover in this thesis because they are not relevant for the proposed methods.

1.1.2.4 Memory In Multiprocessor/Thread Scope

CUDA operates on several SMPs per graphics card, so memory closely related to processors shows some differences compared to so-far known CPU caches and registers. First, each SMP needs to operate on its own set of local memory. Each of the SMPs consists of 32 distinct CUDA processors, of which each needs registers. Furthermore, concurrently running threads may want to communicate information directly rather than via slow global memory accesses.

Registers Similar to CPUs, registers provide very fast access interfaces, but only a limited amount is available per processor. Each running thread operates on its own set of registers tied to the currently assigned CUDA processor. Each set is completely independent from those related to different threads, even from threads running concurrently on the same SMP. This separation obviously leads to a quantitative limitation of registers per thread due to the high number of CUDA processors (up to 1536 on the GTX 680). The most recent graphics cards offer 63 usable registers per thread, while those of the upcoming generation are supposed to include 255. The CUDA compiler also provides parameterization of the maximum usable number of registers for each thread. This is useful for performance optimization due to the possibility of increasing the occupancy of the SMPs.

Local Memory Local memory operates as a fallback in case an application runs out of SMP resources, for instance when it exceeds the maximum number of available registers per thread. Another case where the compiler resorts to local memory is declaration of arrays in threads. If addressing is impossible via registers, local memory is used for storing the array. The denotation “local memory” might be misleading. The specified areas actually reside in global memory, although the compiler itself handles the addressing when necessary. As stated before, frequent global memory access has a noticeable impact on performance and should be avoided.

Shared Memory Sometimes threads within a local scope (called a “block”, see Section 1.1.2.6) might need to communicate or exchange data. So far, the only way to do so is via global memory access, which we already stated to be slow. Until now, all threads would need to issue memory access operations for the same location in global memory. Fortunately, CUDA offers means to circumvent such situations in the form of shared memory. This special kind of memory is available per SMP and offers barrier-free addressing on the whole SMP. Consequently all threads concurrently processing on the same SMP have access to the shared memory. This is especially useful if all threads in a local scope operate on the same data. Each value only needs to be loaded from global memory once, upon which it is stored in shared memory and accessible by all threads. Threads may even collaboratively load data for the whole scope. Concerning speed, shared memory clearly beats global memory. The latency of shared memory is up to 100 times lower than that of global memory, depending on the explicit GPU architecture. Shared memory is sized between 16kb in previous generations (Tesla GPU architecture [34]) and 48kb (Fermi GPU architecture [34]).

1.1.2.5 Computing Model

In the following sections we describe how to distribute the concurrently working threads into scopes, which again are spread to the SMPs by the CUDA runtime. We will not only discuss how to set up the computing grid, but also give a glimpse of how internal thread dispatching works in practice. Understanding the presented facts is a crucial preparation for grasping the impact of the Softshell framework. On the CPU, multi-threaded applications usually involve separating an algorithm into distinct tasks, for instance user input, rendering, networking and much more. The single cores of a standard multiprocessor work independently of each other, handling the execution of such heterogeneous tasks quite well. Each of the threads is tied to designated data to work on and upon a context switch, a single thread is paused and the necessary registers (program counters, ALU registers, etc) are stored for later continuation. On GPUs, the origin and resulting restrictions of the hardware demands a different model. First and foremost, GPU acceleration strength is not to be found in execution of heterogeneous tasks. Rather, all of the processors execute the same operations on heterogeneous data. This computation model is called “Single Process, Multiple Data”, or shorter: SPMD. For instance, addition of two arrays word by word utilizes the mentioned model. Earlier in this thesis, we have referred to similar conditions as “data parallelism” (Section 1.1.1).

1.1.2.6 Computing Grid

Upon identifying data parallelism in an algorithm, defining the computing grid follows as a next step. In most cases, the specific details detach from algorithmic paradigms. In fact, sometimes data structures suggest a specific type of subdivision. More importantly, the computing grid defines distribution of threads over the plentiful processors found on a GPU. The fragmentation of the whole amount of threads is twofold. On one hand, we need to partition the given problem into portions, which we distribute over available SMPs. Those so called “blocks” each contain a certain amount of threads, which are then distributed over the single CUDA processors contained in an SMP. In the next paragraphs we will provide details of how to exactly establish these structures.

Blocks Blocks are subject to two restrictions. On one hand, the maximum number of threads in a block is 1024 ever since compute capability 2 was introduced. On the other hand, blocks may be up to three-dimensional. When having in mind that we still operate on devices mainly used for three dimensional rendering, the origin of this second constraint unfolds. Further, the reasons behind having multi-dimensional blocks provides apparent benefits. For instance, when manipulating volumetric datasets, the three dimensions become handy. Blocks do not have to use all three dimensions. Processing images, for example, suggests usage of two-dimensional blocks. Other algorithms, such as the mentioned piecewise sum of two arrays, even get along with using only one-dimensional blocks. It is worth mentioning that this block structure is unique throughout a CUDA program and cannot be altered while the GPU works on it. However, it is possible to alter the structure between launching single instances of an algorithm.

Grid Once the structure of blocks has been established, the final grid, as it is issued on the GPU, is easy to define. We now simply need to divide the extent of the given problem by the block size. Of course we need to account for the dimensionality of our blocks and problem. Following this procedure, we achieve a two-layered structure. We have a grid, which contains a certain number of blocks in up to three dimensions, and each of the blocks contains a certain amount of threads, again in up to three dimensions. The final structure of the grid is depicted in Figure 1.5. The grid in this figure shows two-dimensional blocks and a two-dimensional grid. As soon as the grid and block structures are set, we can launch a CUDA computation function, also called “kernel”. Similar to blocks, the structure of the grid remains static during computation on the GPU, but can be changed in between launches. We need to introduce another term related to the threads in execution. “Occupancy” describes the degree of utilization of the available processors. In Section 1.1.2.4 we introduced shared memory and its limited size per SMP. This imposes a restriction on the maximum number of blocks simultaneously assigned to a single SMP. Similarly, the number of registers per SMP is limited as well. Hence, the number of registers used in a thread impact the number of blocks fitting in an SMP. Consecutively, as registers are assigned per-thread, the block dimensions also influence occupancy. All these restrictions combined result in measurability of the maximum number of active warps (and thereby also threads) per SMP at any given time for a GPU program, which is called occupancy.

Arbitrariness in Problem Size So far, we have seen the textbook definition of grids and blocks. In reality, problems appear in arbitrary shapes and sizes. Consecutively, we need to face a few more details. In general, subdividing the total number of threads into portions fitting the specified block structure does not yield an integer. More commonly, especially when working in two or three dimensions, blocks at the borders appear fractioned into a part containing threads, and one that is empty. When we recall the fact that the block dimensions are unique throughout an application, we clearly need to define means

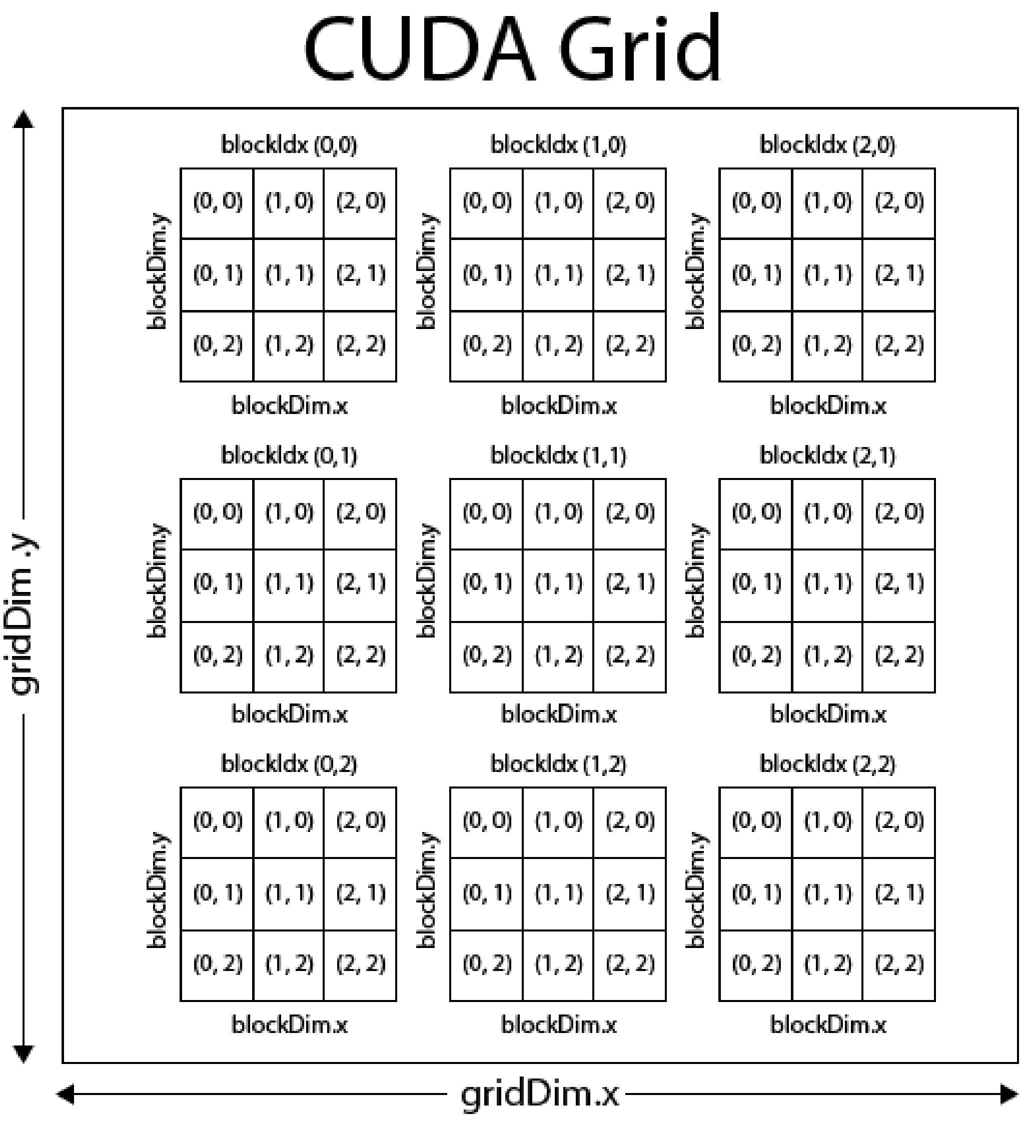


Figure 1.5: An exemplary two-dimensional CUDA computation grid. Picture taken from [52]

for handling these borders. Luckily, in CUDA, this is not a complicated task. In fact, we do not need to worry about this issue during definition of blocks and the grid at all. The trick is to define the grid to cover the minimal amount of blocks needed to hold all the threads in each dimension and allow a few threads to remain idle. Suppose we want to tackle a three dimensional problem with the size

$$problemDim = (n_x, n_y, n_z). \tag{1.1}$$

Further, suppose we use blocks with three dimensions and the following size:

$$blockDim = (b_x, b_y, b_z). \tag{1.2}$$

We now need to define a grid consisting of three dimensions, which, as we mentioned, accommodates to both *problemDim* and *blockDim*, and we also do not want to waste too many available resources. Obviously, we accomplish the optimal solution if the blocks at the borders contain at least one thread. Thus, we define the grid size to be

$$gridDim = (\lceil \frac{n_x}{b_x} \rceil, \lceil \frac{n_y}{b_y} \rceil, \lceil \frac{n_y}{b_y} \rceil). \tag{1.3}$$

Note that this definition does not depend on the actual number of dimensions used by the problem. Even if we neglect one or two dimensions in the grid, the optimality as well as the correctness of equation 1.3 still holds. Each of the thread relates to a unique identifier in the grid. To prevent the execution of threads in border regions, which exceed the actual problem size, we exploit the identifier to determine a possible transgression beyond the border and abort execution for single threads.

1.1.2.7 Program Execution

In the last sections we have explained how CUDA handles memory as well as how to separate a parallelizable problem into portions fitting the GPU architecture. The following section concerns actual program execution and provides some more details of the factual execution on the SMPs.

Kernels In short, the term “kernel” paraphrases the launch of a program in CUDA. Preceding the launch of a kernel, we need to make sure the required device memory is set up and populated with required data values. A kernel call, just like a normal function call, provides capabilities of including function parameters. The grid dimensionality, as discussed in section 1.1.2.6, is a mandatory parameter and inseparably built into a kernel call. Further values are optional. Once a kernel has been launched, there is no way to interrupt its execution.

Warps There is one more detail to the parallel execution we did not discuss yet. We discussed the structure of blocks and mentioned their capability of containing a maximum of 1024 threads. Each SMP contains up to 48 single concurrently working processors. Consecutively, a further subdivision of the blocks to accommodate to the actual hardware conditions is inevitable. We did not discuss this topic so far due to its detachment from the user-defined computing grids. Warps, contrary to the dynamic block and grid dimensions, have a fixed size of 32 threads. A CUDA programmer does not need to worry about separating the blocks in threads manually. Rather, CUDA seizes control over this step. We will not discuss actual warp dispatching (separation into and issuing of half warps to accommodate to the 48 computing units per SMP, warp scheduling, dual issuing of half warps and more) in detail. For more information on this topic, please refer to [10]. Contrary, we need to discuss the impact of this additional layer on designing the computing grid. Reaching optimal occupancy of the execution units on the lowest scale is one of the main goals. Warps may only draw threads from a single block, leading to the inability to gather surplus threads from distinct blocks in a single warp. Therefore, vacant threads within a warp constitute wasted resources as there is no facility, which enables filling in the gaps. Designing the block extents to be separable into an integral number of warps is highly advisable. Although this restriction holds for block borders, multidimensionality of blocks remains unconcerned. Subdivision of blocks into warps does not happen per-dimension, but rather by means of the total amount of threads in a block. Concluding, the total amount of threads within a block should be ideally a multiple of 32 for optimal occupancy.

Thread Distribution Upon issuing a kernel on its corresponding grid, the CUDA runtime seizes control of actual parallel execution and. One of the responsibilities of the runtime lies within distribution of the blocks over the SMPs. Each SMP is capable of holding up to 16 blocks in compute capability 3 at any given time. A large enough occupancy of each SMP is crucial when it comes to hiding memory latency by context switching. The so called "compute work distributor" acts as a unit globally responsible for block distribution over all available SMPs. Initially, this functional unit fills all the SMPs with as many blocks as they can hold. We need to note that blocks cannot, under any circumstance, migrate to another SMP. Consecutively, assigning a block to a SMP is a final decision and might hamper performance in certain cases. Following the completion of a block in an SMP in a certain SMP, the compute work distributor issues a new block on the SMP if additional ones are available. Now we take a look at how SMPs handle thread distribution over their computing units. Warps, as we mentioned in the previous graph, obtain their threads from single blocks. Subdividing each block into warps is the subsequent necessary step. The SMP is in charge of this task. Once the SMP has completed the subdivision, the warps are ready for execution. Each SMP features an innate scheduler whose responsibilities aim at choosing a warp for execution. At each cycle, the SMP fetches a single instruction for a warp, which is marked as ready. For instance, a warp waiting for a memory transfer or a long floating point operation to complete is not ready for execution on the SMP. Instead, the warp scheduler chooses warps, which do not fall in this category for execution. At any time, arbitrarily many threads of a warp may stall. The reason is not only found in waiting for data from memory, where all threads stall at once. The necessity for some threads to skip one or more execution(s) can also result from taking different branches of execution, but more on that in section 1.1.4.2. Another possible reason for staying idle results from early thread termination. Sometimes, single threads have to stop computing, often simply because the algorithm demands it. Early stopping of a thread may have another reason. We have shown that it is often necessary to launch a grid a bit larger than actually necessary for a given problem. In this case, the surplus threads at the borders stop computation after checking their unique identifier. This usually happens within the first few operations in form of an "if-then-else" clause, but the exact location and execution is up to the programmer.

1.1.3 Compute Capabilities

In this Section we describe the evolution of compute capabilities (CC) containing the feature sets of the different released GPU architectures. The first available compute capability, 1.0, offered only two-dimensional grids and three dimensional blocks. The size of grids in x and y was at most 655535, while blocks could contain at most 512 threads, with x and y dimensions being bound by 512 and z by 64. Only 768 threads, spread over 24 warps, were allowed per SMP. Shared and local memory were restricted to 16 kilobytes per SMP or thread, respectively. CC 1.1 first introduced atomic operations in global memory, with CC 1.2 introducing atomics to shared memory. In 1.2, the maximum number of warps per SMP also was increased to 32, leading to at most 1024 present threads. With CC 1.3, double precision units were first introduced in Cuda. The next major revision, CC 2.0, enhanced most of the previously mentioned capabilities. Grids were extended by a third dimension and block sizes increased to contain 1024 threads at most, doubling the maximum allowed values for x or y. The number of warps per SMP increased to 48, and consecutively the number of resident threads adapted to 1536. Furthermore, shared memory size per SMP was lifted to 48kb and local memory per thread to 512kb. In addition, atomic operations in global and shared memory were extended by floating point atomics. The most recent compute capabilities, 3.0 (consumer GPUs) and 3.5 (server rack GPUs), further bolstered the hardware resources, increasing the register count maximally available per thread and overall on SMPs.

In terms of functional units, CC 1.0 to 1.3 offered eight cores for integer and floating point operations, two special function units (SFU, used for functions such as sin, cos, sqrt,...), two texture filtering units and one warp scheduler per SMP. One instruction is issued at a time (single-issue). With CC 2.0, dual-issue was introduced. Two warps selected by the two warp schedulers per SMP are split in half (half-warps) and issued concurrently over the now available 32 arithmetic operators, four SFUs and four texture filtering units. With CC 3.0, the dual-issue of half-warps was revised and issuing of complete warps was introduced. As a consequence, arithmetic units, SFUs and texture filtering unit sizes have doubled to cope with the changes.

1.1.4 Arising Problems

In the last sections we have discussed how CUDA actually executes threads. In this section we will focus on the problems that may arise from this computation model. This is not a complete list, but the selected topics nail the desire for advanced methods for GPGPU computing down.

1.1.4.1 Static Grid

We have demonstrated how to define the computing grid, along with defining the construction of blocks. We have also mentioned that those decisions are final. In the original CUDA framework, threads cannot transgress the border of the block, which they were assigned to. This can be problematic with regard to performance issues. Let us consider a worst case scenario. We assume a one-dimensional block of size b_x that the SMP divides into $n_{warps} = b_x/32$ warps. Next, we assume that all threads except for exactly n_{warps} can stop execution very early in the progress. In the worst case, all of the remaining active threads relate to different warps, i.e., each warp only contains one active thread. Consecutively, the warp can only facilitate a fraction of the available computing power since most of the threads in each warp remain idle. So, overall, a lot of the available resources are wasted due to the inability to move threads between blocks. If there was a way to exchange a finished thread for a working one, we could gather all active threads and all finished threads in two separate groups of blocks and effectively increase the occupancy of the SMPs.

1.1.4.2 Divergence

In sections 1.1.2.7 and 1.1.4.1 we mentioned the possibility of executing different branches within a warp. In 1.1.4.1 we focused on early thread termination. This is not the sole reason for diverging execution paths, but rather a special case of one of the following possibilities. The first, and most obvious cause are conditional statements, namely "if-then-else". In this case, all threads within a warp with a positive evaluation of the condition execute all operations found in the "then" branch, while the threads remaining in the warp stay idle. Contrary, in the "else" branch, the roles are reversed. Only the threads computing a negative evaluation can compute the same series of operations concurrently. The rest of the threads have to wait for completion of the branch. If we observe the behavior of a branch with i_{true} statements in the positive branch and i_{false} operations in the negative counterpart, the total runtime is *not* $max(i_{true}, i_{false})$, but rather $i_{true} + i_{false}$ due to the inability to execute heterogeneous operations in one cycle. In case of multiple branches depending on a single condition things only get worse. The consequences are even more extensive than it might seem at the first glance. Loops, such as "for" and "while", depend on such "if-then-else" conditions for evaluation of whether to continue the loop or not. Here, the "else" branch remains empty, but the "then" part, the loop body, usually contains quite a few operations. If the amount of required loop iterations diverges for threads in a warp, the outcome is pretty similar to the previously mentioned divergence. Some threads remain in the loop, while others exit it and remain idle until all threads leave the loop. Let us assume a loop body containing i_{body} statements and the maximum and minimum number of iterations are denoted as n_{max} and n_{min} respectively. The time a thread remains idle is then given by $(n_{max} - n_{min}) * i_{body}$. We now have two parameters, which influence the time wasted for idling. This can be especially problematic when both the divergence is extreme and large portions of an algorithm reside within such a diverging loop. What carries these difficulties to the extreme is the possibility of arbitrarily nested diverging statements.

1.1.4.3 Lack of Control

Another reason for thinking of advanced computation paradigms is the lack of control in the current model. Initiation of computation on a GPU presently demands that the algorithm finishes at all cost. Hence programmers cannot incorporate changing circumstances into currently executed algorithms. The consequences are manifold. For example, hard real-time constraints possibly might not met due to the inability to interrupt execution.

1.2 Volume Rendering

In this section we provide a crude overview of volume rendering in general and focus on previous scientific work in this field. We also highlight those approaches we employ for evaluating the Softshell In contrast to surface rendering methods, volume rendering techniques attempt to display datasets comprising gaseous clouds of sample points with according light transportation models. This physical model vastly differs from surface rendering due to the interaction of data points among the cloud. While in photo-realistic volume rendering, as used e.g. in video games, the physical model of the gaseous medium itself is crucial (fire, fog, clouds), approaches for scientific visualization focuses more on the light transportation model. In this thesis, we only provide an overview of the basics for the underlying physics. For more detail, please refer to the comprehensive information found in [19]. We also adopt the mathematical notation used in the mentioned book.

1.2.1 Volumetric Light Transport

Max *et.al* [30] provide a very good overview of techniques employed in approximating lighting models, some of which we pick out and expose. Before forming an equation for the interaction of samples with incoming light, we need to discuss different parameters involved. Energy of light is generally transported along straight lines, or light rays. Interaction with the gaseous medium results in diffraction of the light ray as well as alteration of the transported energy. We now describe the four main ways of interaction.

Emission: Emission describes active contribution of the volume to the energy of incoming light. A light ray hitting a sample, given its direction and energy, transgresses through the sample while maintaining the direction. Additionally, according to the amount of emission radiating from the sample, the energy is increased by the corresponding amount.

Absorption: Although absorption also preserves the direction of the light rays, the energy update is just the opposite of emission. Some of the transported energy is lost due to physical properties of the medium. In real physics, light energy is of course not lost, but rather converted into another form - e.g. heat. Modeling this transformation as a loss is acceptable in volume rendering as the goal does not reflect in modeling energy theories, but rather visual effects.

Out-Scattering: Light transport through a medium not necessarily always results in direction preservation. Quite contrary, physical properties of particle rich media also allow for reflections of incoming light into several directions, referred to as out-scattering. Basically, if a light ray hits such a particle (or sample), the energy is scattered spherically around the point. Representing this property analytically is hard to achieve. Consecutively, approximating the properties by a certain amount of outgoing light rays whose energy depends on the incoming light ray must suffice.

In-Scattering: In-Scattering results from light emitted by several sources. Not only more than one light source contributes to the energy leaving a sample. Out-scattering also plays a major role in this process. Light energy from surrounding sources is gathered in a sample and affects outgoing light by increasing its total energy.

1.2.2 Illumination Models

We are now ready to state the comprehensive volume rendering equation including all of the above effects and consecutively describe each of the terms.

$$\begin{aligned}\omega * \nabla_x I(x,\omega) = &- (\kappa(x,\omega) + \delta(x,\omega)) * I(x,\omega) + \\ &q(x,\omega) + \\ &\int_{sphere} \delta(x,\omega') * p(x,\omega',\omega) * I(x,\omega')\mathrm{d}\Omega\end{aligned} \tag{1.4}$$

The radiance $I(x,\omega)$ is a way of describing the light energy emitted from an area per angle per time unit. ω relates to the light direction while x refers to the spatial position of the sample in question. Calculating the dot product of light direction and the gradient of radiance results in the desired luminance of a sample. The functions κ, δ and p relate optical properties to the material of the volume. The spherical integral captures incoming light resulting from scattering. Here, ω' denotes the direction of light originating from sources unrelated to the original light source with direction ω. Hence, the integral covers all incoming light (in-scattering) caused by refraction of surrounding samples (out-scattering).

1.2.2.1 Local Illumination

Due to the vast computational complexity of approximating the whole volume rendering equation (1.4), commonly used variants neglect some of the terms. In most approaches, only emission and absorption are considered as a trade off between image quality and performance. These models are often referred to as local illumination ([13],[27]) as they do not capture global influences from indirect illumination. For even further increased performance, models only embracing emission *or* absorption are popular. If we rewrite equation 1.4 to its non-differential form (i.e., if we integrate along the light direction, we achieve the volume rendering integral:

$$I(D) = I_0 * e^{-\int_{s0}^{D}\kappa(t)\mathrm{d}t} + \int_{s0}^{D} q(s) * e^{-\int_{s0}^{D}\kappa(t)\mathrm{d}t}\mathrm{d}s \tag{1.5}$$

Local illumination models describe an approximation of 1.5 Usually, one or more light sources emit light of certain colors, while each sample absorbs some of the incoming light. Prominent approaches for local illumination were proposed by Phong [36] and Blinn [4](also called Blinn-Phong model). The latter is an improvement over the former induced by exploiting a very similar method but with a lower computational complexity. Consecutively, the Blinn-Phong model often is the preferred method. Both models incorporate ambient, diffuse and specular lighting. Especially the specular, or reflective, component requires some computational power. The Phong model requires setup of a vector depicting reflection of light, depending on the surface normal and the direction from the point in question to the light vector. The Blinn-Phong model replaces this sophisticated term by an approach involving a half-vector in between the light and the view vector, which is faster to calculate. Figure 1.6 depicts the vectors required for each of the two models and highlights the difference in required effort.

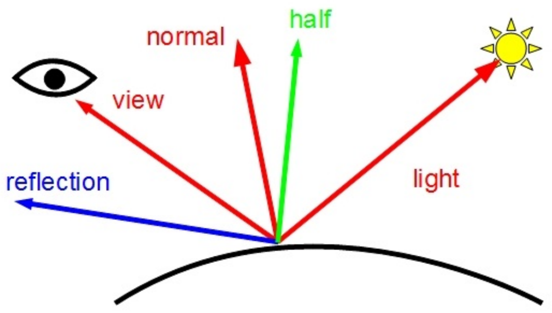


Figure 1.6: Reflective illumination calculation. The blue reflection vector is used in the standard Phong model and requires mirroring the light vector with the normal vector as axis. The green vector is used in the Blinn-Phong model and can be computed with significantly less effort.

1.2.2.2 Global Illumination

Often, simple absorption plus emission provides too little optical information. Using the so far described model can result in images representing the desired insight into a dataset insufficiently. Consecutively, more complex approaches capturing the features inherent to the dataset is necessary. When recalling the volume rendering equation (1.4), we now obviously also need to approximate the spheric integral related to incoming light from scattering throughout the dataset. Different approaches exist, each of which assumes distinct simplifications to make the underlying problem more tractable. Single scattering treats the medium as a low-density volume. Consecutively, the probability of multiple scattering is minimal. Quite contrary, some other approaches assume the opposite. Light is scattered throughout the volume so often that the approximation converges towards a diffuse model.

1.2.3 Data Representation

Volumetric datasets come in vastly different representations. Most often, the source of the datasets decree the optimal way of describing the spatial topology of collected data

points. Hence, we also need to discuss possible origins of the datasets in question.
In many scenarios, medical imaging devices (computer tomography, magnet resonance imaging, ultrasound) record three dimensional scalar fields of parts of the human body. Most commonly, these devices provide a set of two dimensional slices. Except for ultrasound imaging, which generally produces unaligned slices with non-rectilinear grids, the datasets comprise a rectilinear, uniform grid of sample points. Usually, the devices operate on adjustable, but per-dataset invariant spacing. Close to flawless slice alignment is easily achieved considering the precise control of the patient table. Given the nature of these slice-aligned datasets, adopting the rectilinear grid structure is advisable.
Another prominent resource of volumetric data is simulation of physical phenomena. Methods such as the finite element method ([47],[54]) for approximation of differential equations of the applied element method involving prediction of continuous as well as discrete behavior ([20]) heavily rely on unstructured grids. Approaches, which facilitate these methods, often incorporate temporal variance, e.g. heat distribution or flow.

1.2.4 Basic Approaches

Generally, all volume rendering algorithms relate to two different categories. In general, volumetric samples need to be sorted according to the view direction to enable for capturing mutual occlusion and performing composition of consecutive samples.
Image based approaches solve the problem of sample visibility in the image space. Most approaches implicitly sort the samples along the view direction via construction of viewing rays per image fragment. The probably most prominent approach in this category is volume ray casting. The ever-increasing performance of GPUs in combination with its visual quality are the driving forces behind continuing interest in the method. Many more algorithms with an image-based foundation exist (Section 2.3).
In contrast, object-based approaches rely on more explicit ways of sorting the volumetric primitives.

1.3 Dykstra's Projection Algorithm

This technique stems from convex analysis, more specifically from simultaneous projections onto convex sets in n-dimensional space to determine a point in their intersection. Finding such a point ad-hoc is generally deemed intractable. A preliminary assumption for this method to work is the presence of a projection operator on a single set, while the operator for projecting on the intersection of sets is deemed to be unknown. In the basic case involving two convex sets, the method chooses a point of one of the sets and projects them onto the counterpart. In the next step, the roles of the sets switch, hence projection alternates between the two sets. In case of more than two intersecting sets, the algorithm is enhanced by alternately projecting a point on the border of one set onto all others. Consequently, Dykstra's algorithm performs iterative alternating projections onto single convex sets of a set of intersecting convex sets. For two convex sets, the following Definition denotes the corresponding optimization problem:

Definition 1.3.1 *Dykstra's Projection Algorithm*

$$\begin{aligned}&A, B \ldots \ \textit{convex sets} \ \subset \mathbb{R}^n\\&x \in \mathbb{R}^n\\&\textit{minimize}\|x - r\|^2, \ \textit{subject to } x \in A \cap B\end{aligned}$$

In contrast to other projection methods, Dykstra's algorithm provably possesses strong convergence. Furthermore, Dykstra's algorithm does find a specific point in the intersection depending on the initialization r, whereas other methods detect arbitrary points within the intersecting region. The method described is used in many signal processing tasks, be it one dimensional (speech recognition) or two dimensional (image processing).

1.4 Algorithm Choice

After we have introduced the environment we work in it is time to motivate the choice of the algorithms that we will evaluate in this thesis. Each of the algorithms inherently suffers from one or more of the shortcomings described in Section 1.1.4. We will briefly summarize the single issues here. For a more elaborate description of the algorithms we refer to the Chapters 3, 4 and 5.

1.4.1 Raycasting with Advanced Illumination

Generally, global illumination effects in volume rendering applications are computationally intense. Recently, an iterative method [48] was proposed, which strongly decreases the memory consumption of the effect. The authors propose iteratively creating a deep shadow map with a scan line approach. Nevertheless, the method suffers from alternating GPU and CPU computation. By employing Softshell, we will show how to efficiently avoid the intermediate steps on the CPU. In effect, our modification reduces the number of memory transfers between device and host and further increases GPU occupancy with a more dynamic computing grid (Section 1.1.4.1).

1.4.2 Particle Based Volume Rendering

The basic volume rendering approach proposed in [50] inherently suffers from warp divergence (Section 1.1.4.2). Although the approach already is rather fast, we will transform the algorithm in a fashion strongly benefitting from Softshell's scheduling environment to gain even more performance.

1.4.3 Dykstra's Projection Algorithm

Finding points in the intersections of large quantities of convex sets is intense in terms of computational effort, but also memory consumption. The corresponding optimization problem suffers from warp divergence (Section 1.1.4.2) in three interleaved scopes. Consecutively, the algorithm is most suitable for extensive testing of Softshell's reconvergence strategies.

Chapter 2

Related Work

2.1 Advanced Computing Models on the GPU

Due to the importance of the Softshell [46] execution model for this thesis, we discuss its most important ideas in this section. Please note that Softshell indeed is just a model, autonomous from the specific underlying hardware and corresponding programming language.

The very idea of Softshell is to harness a framework for efficient execution of large quantities of threads executed in parallel. Research in this area has only recently received some attention due to the availability of massive parallel computing on GPUs. Recent research in this field mostly focused on exploiting programmable shaders in a general purpose fashion. The authors of [7] describe a programming language capable of applying a streaming programming model to high level shading languages: *Brook*. Although this model is as well platform-independent, it so far lacks the support of the most recently published GPGPU programming languages (i.e. CUDA, OpenCL). The underlying memory model, implementing the data streams, relies on the usage of floating point textures on the GPU. Implications of this model show eventually arising problems due to the fixed limit of the amount of available texture units on a GPU, thereby also limiting the number of maximally available streams. The execution unit in Brook coincidentally refers to as kernels as well as in CUDA. Kernels operate on streams, which they use as input as well as output data structure. The authors also show how to virtualize several aspects of GPU computing - namely kernel outputs and stream dimensionality. Further, they suggest extending the amount of virtualization on the GPU.

Approaches similar to [7] gained attention due to the availability of programmable shaders. Unfortunately, transforming general purpose algorithms to match graphics APIs used by shaders can often be a tedious task and generally imply restricting structures. The recent transition from programmable shaders to completely programmable multiprocessors on a GPU demands different methods to achieve high performance given the drastically changed circumstances. In the more recent approach published by [16] *et. al.*, the authors focus on the problem of thread divergence in modern GPGPU architectures. Algorithms can be grouped by blocks with different convergence behavior, i.e., divergence causes the control flow of a program to end up in different blocks. The authors exploit a model, which describes “domination” of blocks over different ones. Using this heuristic leads to the ability to determine divergence and react by splitting warps and regrouping threads into new, convergent warps again. Furthermore, the authors suggest possible hardware adaption for native support of their method and evaluate the induced increase in chip area.

2.2 Softshell Computing Model

Although Softshell is a generic model, applicable to many SIMD structures, we will from now on refer to the explicit implementation on top of CUDA. In Section 1.1.2.7 we summarize problems inherent to contemporary GPGPU architectures, specifically in CUDA. Softshell gradually alleviates these issues with its scheduling primitives and three tier scheduling model. In the next few Sections, we will proceed to give a detailed summary of the facts most important for later analysis of the algorithms we benchmark.

2.2.1 Scheduling Primitives

Similar to the basic CUDA model, Softshell disposes of a tiered entity system used for workload balance. In this subsection, we focus on not only describing the given entities, but also on relating them to the corresponding basic CUDA entities.

Procedures Procedures in Softshell basically describe the function, or algorithm, to be executed in parallel. In contrast to CUDA kernel launches, procedures may be executed on a global scale, i.e. on all available SMPs, but also locally on only a few SMPs. The very idea of procedures is concurrent execution of several, possibly vastly different, functions. Up until the most recent GPU generation, CUDA was not capable of executing several kernels at once.

Work Items Work items define the necessary input of single threads executing a procedure. Contained information is basically arbitrary, allowing for dynamic parametrization of procedures. Definition of the payload on a per-thread basis enables dynamic regrouping of threads executing a procedure, e.g. for re-convergence.

Work Packages Work packages correspond to a group of work items. Grouping the threads into those packages is required for actual parallel execution of the same procedure on an SMP, similar to blocks in CUDA. The togetherness of work items may change over the course of time. Further, the amount of threads per work package may be dynamic, leading Softshell to automatically regroup workpackages for optimized execution based on CUDA restrictions. Contrary, one can define a static number of work items per work package, denying the regrouping mechanism. The latter may become necessary if a fixed number of threads per work package is imposed by algorithmic circumstances.

Events In Softshell, events relate to initiating execution a procedure on the GPU. Therefore, work items need to be defined and are consecutively scheduled for execution by a defined procedure. Due to Softshell's ability to generate additional activity over the course of actions, newly created work items or packages are again associated with the corresponding event and its set of procedures. Events can either be triggered explicitly, via invocation controlled by the user, of implicitly by using e.g. timed sequences.

2.2.2 Compute Model

Softshell defines three tiers of scheduling. The responsibilities of each tier reflects in a different level of parallel execution.

Tier One The first tier distributes work over several GPUs. Upon invocation of an event, Softshell tries to optimize performance by considering the importance of an event (priority) as well as the current workload of all available GPUs.

Tier Two This tier is responsible for evaluation of and factoring in priorities of single work packages. These priorities may change over the course of execution, requiring Softshell to sort queued work packages according to their regularly queried priority. Each work package can exploit a distinct function for evaluation of the priority, eventually reacting to current computational circumstances. Newly arriving work packages, of course, can relate to a high priority compared to existing ones, leading to necessary earlier execution. Consecutively, the order of arrival is uncorrelated to the order of execution.

Tier Three This tier is where the definition of work items and packages is very beneficial. Softshell offers the ability to halt or even cancel execution of events. Consecutively, thread contexts need to be stored (similar to a CPU context switch) for an eventual later re-issue of the corresponding threads. Further, reconvergence of threads requires storing the contexts for regrouping.

2.3 Volume Rendering

2.3.1 Ray Casting with Advanced Illumination

In this subsection we will summarize the most important work related to both ray casting as well as global illumination approaches in the volume rendering environment.

2.3.1.1 Ray Casting

One of the most important methods for high-quality rendering of volumetric datasets is ray casting [28],[38]. Over the years, many methods for optimizing its performance have been proposed in [25], [15], [39] and [24]. Besides this behemoth, other image-based approaches of course exist as well. As we do not aim at listing a comprehensive collection of those algorithms, we will briefly skim through a few selected approaches. For example Z-Sweep, proposed by Farias *et. al.* [14] bases on perpendicularly sweeping the image plane through an unstructured grid. Upson *et. al.* suggest subdividing a voxelized dataset into a large amount of sub-volumes and using solid texture mapping in combination with atmospheric attenuation for improved visual results. The method proposed by Yagel *et.al.* [53] exploits templated ray casting and projection onto a special plane, the “base plane”.

2.3.1.2 General Global Illumination

In [26], Kajiya *et. al.* describe an early method closely related to modeling light transport in non-scientific visualization. The underlying approach, ray tracing, is well-known for producing high quality illumination of surface-based scenes. The focus lies on rendering volumetric phenomena such as clouds, flames or particle systems. The authors also exhibit ways of approximating light scattering in animated, dynamically evolving media.
In relation to single scattering, which imposes extremely low density in the volumetric dataset, Behrens *et. al.* [2] propose exploitation of texture mapping hardware for shadow calculations during volume rendering. Despite the lack of explicit lighting calculations, the approach still achieves a shaded appearance of the volumetric data. Methods as proposed in [12] and [23] exploit the exact opposite - an extremely dense model collapsing towards diffuse shading. These approaches show sensitivity to inhomogeneous data.
Even more sophisticated approaches exist. These incorporate physical models, which utilize multiple scattering to approximate full radiative scattering. Gutierrez *et. al.* improve photon mapping [22] by introducing dynamic refraction of photons traveling through a medium. Further, the authors consider curved photon trajectories, which leads to highly realistic rendering of physical phenomena.

2.3.1.3 Deep Shadow Maps

Highly related to the global illumination algorithm we evaluate in this thesis, Lokovic *et. al* [29] introduced a global illumination method called deep shadow maps. These extend ordinary shadow maps by another dimension and enable modeling light refraction and albedo in a volumetric sense. Although the technique aims at primitives such as hair or fog, Sundén *et. al.* [48] modify the approach for scientific volume visualization. This algorithm along with its modified version (Chapter 3) is one of the algorithms we evaluate in this thesis. The authors propose an axis-aligned plane-sweep technique for simultaneous volumetric rendering and iterative update of the deep shadow map. In contrast to previous approaches, this method enables immediate integration into a ray casting framework and covers light scattering in conjunction with volumetric shadowing. Performing basic ray casting in an image based plane sweep fashion solely changes the order of pixels processing during parallelization and does not affect the rendering algorithm at all. Basically speaking, a single iteration of the sweep produces a slice through the volume whose orientation is determined by the corresponding scan line in the image plane and the view direction related to the line. The authors now exploit the correspondence of this slice to the matching slice in a deep shadow map. When adapting the scan line progress direction according to the position of the light source, consecutive ray casting slices also relate to consecutive slices in the deep shadow map whose data only rely on previous iterations. Hence, due to compositing of the previous slice with the next one, the slice in question at all times contains the composite information of the required parts of the deep shadow map. This approach alleviates the problems arising from the massive memory consumption of the deep shadow map as the volume collapses into an iteratively updated two-dimensional shadow map. In between scan line iterations coherency of the data in the iterative shadow map must be maintained. One reason is the lack of alignment between shadow map and light source (respectively, its direction). Furthermore, the spherical light distribution of point lights invalidates perpendicular propagation of shadow information from one slice to the consecutive one. The authors propose CPU-based coherency measures while slices are generated on the GPU in parallel.

2.3.2 Particle Based Volume Rendering

Usually, object based volume rendering techniques resort to sorting of the rendering primitives. The projected tetrahedra approach (Shirley *et.al.*, [44]) and its GPU extension (Maximo *et.al.*, [31]) perform visibility sorting in object space, similar to the Painter's Algorithm. Similarly, hardware assisted visibility sorting (Callahan *et.al.*, [8]) resorts to partial sorting of volumetric cells for consecutive rendering. In other algorithms (Westover [51], Mueller [32], Schlegel [43]) the volumetric samples are projected onto the screen using a well-defined geometric shape (e.g. discs).
Particle based approaches, as proposed by Sakamoto *et. al.* [42], or Voglreiter *et.al.* [50] treat discrete samples as infinitesimal small or fragment-sized light-emitting sources and promote projection onto single fragments of the screen. The approaches efficiently circumvent depth sorting of samples by employing a form of supersampling. Other methods, such as Csébfalvi and Szirmay-Kalos [11] or Grossman [18], have a similar basic structure.

2.4 Dykstra's Projection Algorithm

The general algorithm was proposed by Boyle and Dykstra in [5]. We use this algorithm in relation to solving problem occurring in computer vision. Pock *et.al.* [37] propose exploiting Dykstra's algorithm for approximating the well investigated Mumford-Shah functional [33]. The equation is used in various computer vision tasks such as segmentation or tracking and approximates a set of piecewise smooth functions. The authors suggest solving the minimization problem from definition 1.3.1 for per-pixel feature vectors in an image.
Bauschke *et.al.* [9] provide comprehensive information on different approaches for finding points in the intersection of convex sets of arbitrary dimensionality. Further, Bauschke *et.al.* have proven the convergence of the algorithm in correlation with Bregman Projections in [1]. In [3], the authors discuss robust stopping criteria for the algorithm. A generalization of the method to an infinite number of convex sets intersecting and simultaneously changing the order of the sets from cyclic to arbitrary is given by Hundal *et. al.* in [21].

Part II

Advanced GPU-Accelerated Algorithms

Chapter 3

Ray Casting with Advanced Illumination

3.1 Basic Raycasting

Ray casting is probably the best known volume rendering algorithm thus far. This method most closely relates to the rendering equation (equation 1.4 and 1.5) and produces high quality images. Further, the method approaches three dimensional data structures in a most general way without correlating the data to geometric structures. Generally, input data comprises of a regular three dimensional voxel grid and spatial subsampling happens via interpolation methods (nearest neighbor, trilinear, tricubic). In the next few subsections we briefly describe the basic volume ray casting algorithm. We will focus on an abstraction of the concepts necessary to develop the global illumination approach. For more details concerning the theory behind ray casting and different optimization strategies, please refer to the related work provided.

3.1.1 Ray Generation

Given the basic idea of image based volume rendering, assigning one ray to each displayed pixel is self-evident. Perspective display of the volume is a desirable mechanism, thus we apply a pinhole camera model, involving the generation of a perspective viewing frustum. The spatial position of the camera, or eye point, will be referred to as (p_{eye}). For a well posed camera model we further define the view direction cam_{view}, the up-vector cam_{up}, and to avoid ambiguities also the perpendicular vector facing to the right, cam_{right}. Based on this camera coordinate system we define a viewing plane whose normal vector resides parallel to cam_{view} (or equivalently: the view plane being parallel to the plane spanned by cam_{right} and cam_{up}). Perspective projection also involves the definition of the opening angles of the camera in both x and y direction, denoted as the field of view (fov_x and fov_y). Typical values for fov_y are 45 and 60 degrees respectively. The corresponding fov_x can be calculated by multiplying fov_y by the ratio $screenHeight/screenWidth$. Given the field of view and the desired screen resolution, we calculate the boundaries of the so far infinite view plane. Therefor we assume that the distance from p_{eye} to the view plane is exactly one. Consecutive definition of the spatial positions of the four view plane corners is simple trigonometry and will not be discussed here.

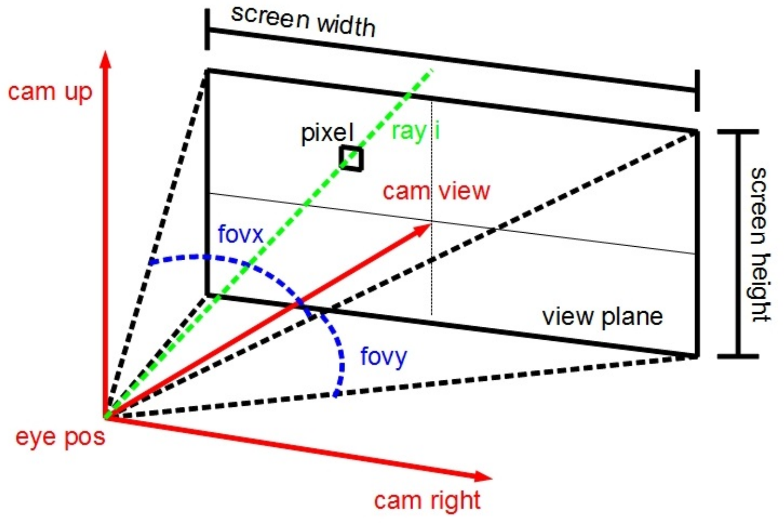


Figure 3.1: Ray casting camera model with all described parameters

Subdivision of the view plane corresponding to the image resolution yields 3D positions of the pixels, which we now use for defining the ray direction:

$$dir_{ray,i} = (pixel_i - p_{eye})/\|(pixel_i - p_{eye})\| \tag{3.1}$$

Using 3.1 we now define the ray equation:

$$pos_{ray,i}(t) = p_{eye} + t * dir_{ray,i} \tag{3.2}$$

Equation 3.2 provides the opportunity of performing an intersection check with the bounding box of the volumetric data we want to render. Only rays with a positive outcome need further attention.

3.1.2 Volumetric sampling

The next step in ray casting involves sampling of the data according to the ray equation. Therefor, a fixed sampling distance t_{step} is applied to equation 3.1. Volumetric data usually consists of one dimensional scalar values rather than color information, implying usage of transfer functions, which translate the sampled values into false colors (C_i) including an opacity (α_i) channel. Attaining the well-known semi-transparent representation of volumetric data is tied to the way samples along the ray are composited, or in other words, in which direction rays are traversed. Two accumulation approaches exist. The first one, called ”back-to-front”, starts at the voxel furthest away from the eye point and finishes at the one closest to the camera. One of the main advantages of this scheme is the traversal of *all* data points along the ray, but this also imposes a high computational cost. The compositing formula is given as

$$C_{out} = C_{in} * (1 - \alpha_i) + C_i * \alpha_i \tag{3.3}$$

where C_{in} denotes the so far accumulated color, C_i and α_i the transfer function lookup of the current sample and C_{out} the composite value.

”Front-to-back” traversal, in contrast, samples along the view direction and starts off at the voxel closest to the camera. Although, in this approach, one needs to keep track of α over the course of the compositing process, this minor disadvantage is toppled by the ability to abort sampling upon α reaching a pre-defined threshold, called early ray termination. This leads to a decrease in computational effort as not necessarily the whole volume is traversed by each ray, but rather only the parts providing sufficient information. The update formula of front-to-back traversal is denoted as

$$C_{out} = C_{in} + (1 - \alpha_i) * \alpha_i * C_i \tag{3.4}$$

$$\alpha_{out} = \alpha_{in} + (1 - \alpha_{in}) * \alpha_i \tag{3.5}$$

In case of front to back compositing, the final color is multiplied with the resulting α and blended over the background.

3.2 Illumination

In sections 1.2.2.1 and 1.2.2.2 we present an overview of local and global illumination algorithms. Specifically in our ray casting approach, we rely on the local Phong-Blinn ([4]) model as we deem it to create a proper trade-off between image quality and performance. For global illumination, we use a heavily altered version of the approach described by [48], which we will now proceed to describe.

3.2.1 Iterative Deep Shadow Maps in Softshell

Optimal facilitation of Softshell requires adaption of the basic algorithm. The next few sections will concern a step by step description of the required alterations as well as mathematical satisfaction for the given steps.

3.2.1.1 Goals

Before we state the actual algorithm, we need to contemplate how to maximally facilitate the Softshell model and minimize the overhead caused by shadow map operations. Originally, the approach requires one kernel launch per advance of the scan line and coherency measures regarding the shadow map in between. In our adaption, we utilize Softshell’s ability to launch additional workpackages directly from the GPU. Hence, coherency of the shadow map cannot be maintained in an intermediate step, which leads us to rethink the character of scan line positioning. Furthermore, the given approach shows poor utilization of the GPU due to the low amount of threads working concurrently. As we understand from the original work, each pixel on the scan line corresponds to a single thread. Even at very high resolutions, the number of threads is simply too low to saturate the available processors on the GPU. We will show how to effectively incorporate a slightly different parallelization method to increase occupancy on the GPU.

3.2.1.2 GPU Utilization

We need to develop a different way of parallelizing the basic ray casting algorithm to be able to fully utilize the available hardware. In the original approach, a single pixel of the scan line shows a distinct correspondence to a thread operating on the GPU. Consecutively, a lot of hardware resources are condemned to be idle. We fill this gap by not only parallelizing the scan line, but also the sampling process along the rays. As we learned in the CUDA introduction, block sizes should be a multiple of 32, so our options for the number of parallel samples per ray are limited. This could lead to issues with early stopping mechanisms as are used in standard ray casting. Interrupting the processing of a single ray upon meeting the stopping criterion is a valid option for standard ray casting, but in contrast, computation of the shadow map requires all samples along a ray, including those which would be skipped by early stopping. These samples may not contribute much to the visual performance of the currently active ray, but it is likely that consecutive rays will need the additional parts of the shadow map for correct visualization. Combined with the proposed parallelization approach, the necessity of traversing all samples leads to a high occupancy of the available hardware and also avoids some of the inherited warp divergence. Still, the samples of the last warp of a ray will, with a very high probability, at least partially exceed the volume boundary. These undefined samples will simply be assigned to the background color to avoid computational issues during compositing. We empirically experienced that in most cases a complete parallelization of all samples does not lead to the best possible performance in CUDA. Allowing a few iterations (depending on the step size used in ray casting and the volume extent, but usually somewhere between 1 to 4) partially alleviates the computational overhead caused by Softshell.

Compositing of the parallel computed samples in a fashion as is used in conservative ray casting approaches is possible, but not advisable. Compositing comes in two fashions - either sample A ”over” sample B, or sample A ”under” sample B. The over and under correlate to the view direction. The over operator implements front to back, while the under operator relates to back to front compositing. Usually, sample compositing is performed iteratively on ordered samples, inducing usage of a loop. But only one of the threads would have to compute the composite value while all others would be idle. Instead of wasting a lot of resources, we resort to a well known principle in parallel computing, namely reduction. Many of the ways for optimizing reduction over several warps require associativity of the underlying operator. Both operators come in a computationally efficient non-associative version, which relies on ordered samples, and an associative approach involving more complex mathematical operations. We found that the disadvantages of the more complex associative operator outweigh the advantages of optimized reduction. Consecutively, we have to resort to the standard variant of reduction, shown in figure 3.2.

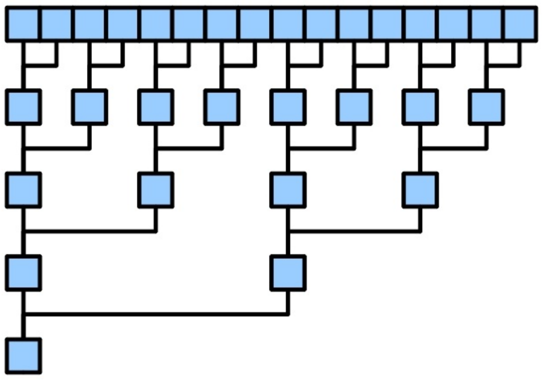


Figure 3.2: Reduction method as used in our sample compositing approach. Sample order is retained while benefiting from parallelism of the technique.

3.2.1.3 Single Launch

Abolish Interdependence: In a first step we invalidate the necessity of launching a single kernel per scan line iteration. Let us assume for now that light sources are only allowed to be directional and their directional vector is parallel to the y axis of the viewing plane. Obviously, the pixels of the scan line are now completely independent from each other as light only seeps through to the directly successive rays along the y axis. Consecutively, we do not need to synchronize the shadow map anymore and we actually now operate with an array of mutually independent, one dimensional shadow maps. This generalization allows us to advance the pixels of the scan line independently from each other. Figure 3.3 depicts the progression of the scan line, or rather the single fragments of the scan line due to alignment with the light source. For the sake of simplicity of consecutive discussions, we will neglect the component of the light vector, which lies parallel to the oriented shadow map. We will show how to efficiently involve influence of this component in the last part of this section.

Softshell: Modifying the algorithm in the shown way enables usage of Softshell's ability to launch additional workpackages. Initially, we launch as many workpackages as there are pixels along the view plane's x axis. Each workpackage consists of $n * 32$ threads, each of which is responsible for a sample along the ray and a certain, but low (refer to section 3.2.1.2 number of iterations along the ray. From then on, each workpackage is responsible for individual progression along the scan line direction. Advancing happens via calculation of the parameters for the consecutive pixel upon finishing work on the current one, and subsequent launch of a new work package in the same event, filled with the matching parameters. Incorporating the described approach annihilates the need for interruption of the GPU processing, which was mainly required to apply coherency measures for the shadow map on the CPU. But so far, the algorithm is only able to operate with directional lights illuminating along the y-axis of the viewing plane (or the x-axis, if the obvious alterations are applied).

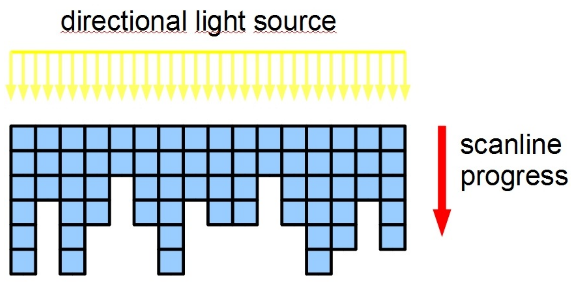


Figure 3.3: Scan line advancing per-pixel due to alignment with a directional light source.

Arbitrary Scan Lines: Restricting the directional light sources as we did in 3.2.1.3 hampers flexibility of the approach at an intolerable level. Therefore we propose an improvement over the original algorithm by allowing rotation of the general scan line direction. Of course, this leads to the necessity of taking a closer look at the mapping from view plane pixel to fragments of the scan line. Figure 3.4 shows problematic areas of an arbitrary scan line implying ambiguities in the mapping process.

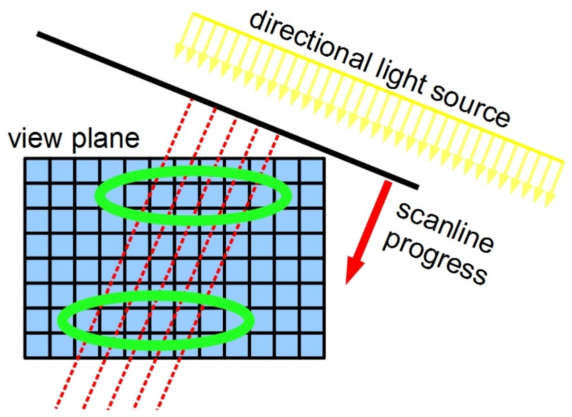


Figure 3.4: Scan line with an orientation not aligned to a view plane axis. The areas in green highlight problematic regions in need of further attention.

Resolving the issue occurring with the oriented scan lines requires us to make a few observations first. Initially, it is easy to verify that all scan line fragments (depicted as red dashed lines in figure 3.4), which firstly hit the view plane at the first line of its x-axis are mutually equidistant. Transgression of this sampling equidistance to the remaining fragments eases up the further progress, although their first hit is unlikely to proceed exactly through a pixel center. Consequently, for easier depiction, we assume a virtual extension of the view plane in a fashion leading to invariably all scan line fragments first hit the view plane on the same axis. The equidistance of the fragments distributed over the scan line now also implies centering at the firstly hit pixels. For rendering, we neglect the virtually added parts of the scan line. This extension indicates yet another observation. The equidistance causes a periodicity of occurrence of the problematic regions. Depending on the angle between the main axes of the view plane and the direction of incoming light, periods either occur along the x or y axis of the view plane. Fortunately, similar problems are a well researched topic in computer graphics. Rasterization of lines is a very basic requirement for many rendering algorithms, leading to numerous approaches. But not all of them qualify for our needs. Retaining the mutual independence of fragments of the scan line requires the chosen approach to neglect effects, which are often desired in rendering. Techniques such as anti aliasing directly imbued in the algorithm disable resolving of the ambiguities. In fact, we need a method implementing a binary decision whether a pixel corresponds to the current line. Bresenham's [6] algorithm perfectly fits our needs while also showing extraordinary computational performance. Figure 3.5 shows the final procedure of matching.

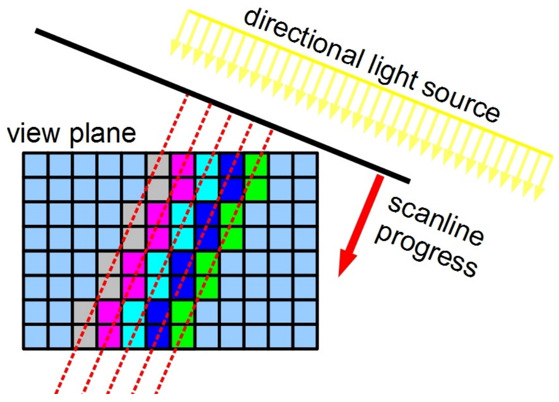


Figure 3.5: Final matching of scan line fragments and pixels in the view plane. Different colors in the view plane depict different fragments.

Memory conflicts: Standard approaches usually employ a fixed size shadow map. We already showed that, in our approach, the shadow map actually is a loosely coupled conglomerate of single shadow arrays per ray. We now further exploit this characteristic for finally resolving issues occurring with evenly spaced shadow textures. First of all we need to synchronize the ray front to be able to determine a starting point for shadow map operations. Figure 3.6 shows the behavior of rays as seen in vanilla ray casting.

We need to synchronize the rays to a common start by determining the point of the bounding box closest to the view plane. Virtual parallel translation of the view plane into this point yields the optimal planarly synchronized ray front, as our future shadow map cannot possibly be closer to the eye point under any circumstances. Analogically, translating the view plane into the point of the bounding box furthest from the eye point results in the maximum extension of the view plane. Figure 3.7 shows what we have achieved so far.

We denote the distance between the eye point and the two respective planes as $t_{s_{near}}$ and $t_{s_{far}}$. Using the difference $\Delta t_s = t_{s_{far}} - t_{s_{near}}$, we now deduce the shadow map resolution. As we have already shown, it is possible to orient the normal vector of the scan

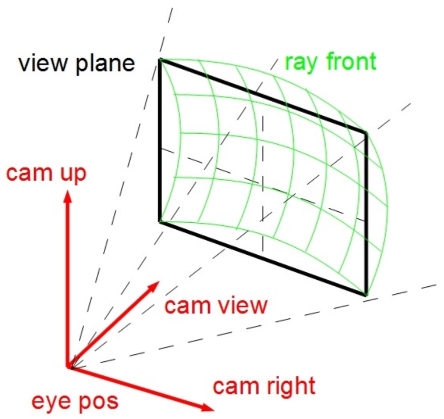


Figure 3.6: Ray front for common ray casting. The front shows hyperboloidal behavior at synchronous ray extensions.

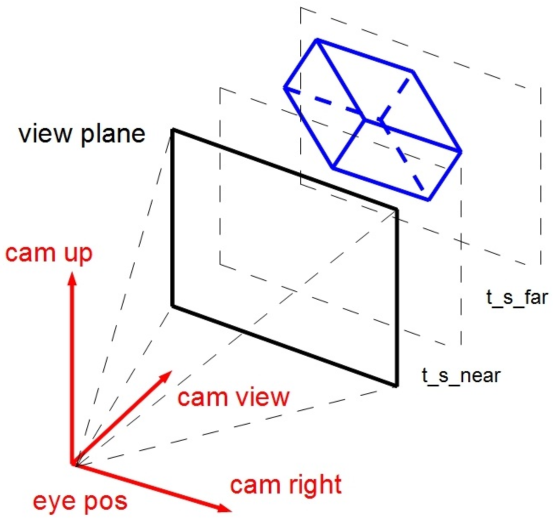


Figure 3.7: Synchronization of the ray front to planes parallel to the view plane, depending on the bounding box of the volume.

line to coincide with the light direction relative to the view plane and further fragments of the scan line operate mutually independently. Consecutively, our situation can be described as shown in figure 3.8. Let us first assume the 2-dimensional case as shown in the figure. Our first observation relates to the number of samples projected on the shadow array, which is parallel to the central ray. Clearly, the central ray requires the least samples, as the projected distance equals the sample distance during ray casting. Consecutively, depending on the scan line orientation, either the uppermost or the lowermost ray projects the samples with minimal distance along the plane (line) we project onto. Next, we generalize to the three-dimensional case. In the viewing frustum created by our perspective viewing model and the two shadow clipping planes $t_{s_{near}}$ and $t_{s_{far}}$, the largest possible distance is clearly contained in the lines from eye point through the view plane corners $c_1 \cdots c_4$. We can write this as:

$$t_{s_{max}} = (t_{s_{far}} - t_{s_{near}}) / \langle \frac{cam_{view}}{\|cam_{view}\|}, \frac{c_i - p_{eye}}{\|c_i - p_{eye}\|} \rangle \tag{3.6}$$

In other words, the four corner pixels on the view plane project the most samples onto the view plane. Note that this is the maximum possible size we will ever encounter rather than the optimal size for a distinct spatial orientation of a volume. It is easy to observe that we can further restrict the value by also incorporating the extension of the bounding box projected onto the view plane. Due to the definition of the shadow frustum via the bounding box extension in view space, the maxima hold for arbitrary orientation of the scan line.

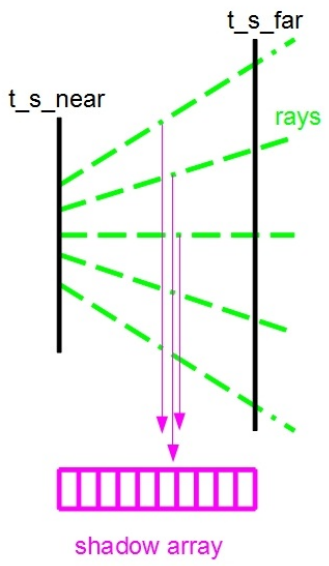


Figure 3.8: Projection of the samples during the sweep of bunch of rays related to a single fragment onto the scan line. Each of the magenta lines denote samples with equivalent distance from the starting point.

Achieving non-conflicting memory writes is crucial for parallelization in CUDA. Figure 3.8 shows the problematic situation during projection, resulting from equidistant sampling along rays with different angles relative to the line to project onto. We have already shown how to identify the ray that projects the most samples onto the shadow array. In the next step, we calculate the resolution of the shadow map as

$$\#samples = \lceil \frac{t_{s_{max}}}{stepsize} \rceil \tag{3.7}$$

where $t_{s_{max}}$ comes from equation 3.6 and *stepsize* is the step width used in ray casting. Hence, we provide enough space for projecting all samples of the most demanding rays into distinct bins. Yet, this setup alone does not suffice yet for complete data parallelism in the shadow map. Consider, for example, the central ray of the view plane and an arbitrary sample along its path. The projected extension of this sample in the shadow map is clearly larger than what was originally reserved. As a result, lookups as well as writebacks are problematic. The former require reading and compositing several several samples from the shadow map, which is in itself not a big problem. Contrary, the writebacks pose a bigger issue as we would need to distribute the value over several memory locations while accommodating for the proportional contribution. Given that we compute many consecutive samples concurrently, this inevitably leads to one or more writing conflicts per sample.

We now propose a method for resolving the writing conflicts. Recall that all fragments of the scan line are mutually independent. Hence, we only require an approach resolving the problem in 2D. We denote the direction of the central ray i of the sweep corresponding to a single scan line fragment as dir_{center_i}. The angle between this central direction and the upper or lower limit, respectively, will be referred to as ϕ. In the simplified case of a scan line parallel to the x-axis of the view plane, or equivalently, perpendicular to a light with a direction perpendicular to the x-axis, ϕ is equivalent to $fov_y/2$. Note that ϕ is consistent for all fragments if we again consider the virtually expanded view plane. Initially, we solve the problem for the first rays of each fragment. From the fact that these rays require the most fine-grained resolution we infer that a stepsize of one in the shadow map is the best choice. Avoiding memory conflicts for the consecutive rays requires adaption of the sampling stepsize used by the ray casting algorithm such that each sample uniquely relates to a distinct entry in the shadow map. Figure 3.9 visually explains these requirements. We derive the formula for the updated stepsize as follows. First, we denote the projected shadow map sample size of the starting ray in relation to the originally used stepsize in ray casting:

$$z_0 = cos(\phi) * stepsize_0 \tag{3.8}$$

and conclude the updated stepsize with normalized direction vectors:

$$\begin{aligned} stepsize_i &= \frac{z_0}{cos(\phi_i)} && (3.9) \\ &= \frac{z_0}{\langle dir_{center_i}, dir_{ray_i} \rangle} && (3.10) \end{aligned}$$

Applying the updated step sizes to the ray casting process leads to a synchronized ray front over the whole sweep. Consecutively, we achieve distinct, uniquely determined memory locations for writing.

We need to discuss another problem before finalizing the algorithm. Although we orient the scan line to be perpendicular to the light direction in screen space, this is not necessarily the case for the plane spanned by the shadow map in object space. We again consider influence on both read and write operations for the shadow map. Fortunately the remaining offset, which can be described as a vector parallel to dir_{center_i}, is not involved in the write operations. The ambiguity of read operations is also easily resolved. All we need to do is interpolate two samples of the shadow map resulting from the previous ray according to the offset. Yet - another issue requires examination. Updating the shadow map in a warp is guaranteed to be consistent due to the distinct memory locations and non-interference of the samples. Contrary, warp borders still pose a problem. Suppose we have two warps for a ray. Further suppose that the first warp finishes its calculation before the second is chosen for execution. Now all threads of the first warp have already updated the shadow map. But one or more of the already overwritten values might be needed by one or more threads of the second warp. This border discrepancy is easily resolved by using two buffers, one used for input and one solely for output. After each progression step of a scan line fragment, we swap these buffers and effectively resolve the issue. The additional memory does not pose a problem for standard applications, but might be problematic for very high resolutions. In these cases splitting the scan line into several passes for fulfilling memory constraints is an option when using the proposed method.

3.2.1.4 Final Algorithm

We are now prepared to condense the previously described modifications into algorithmical shape. Algorithm 1 sets up the necessary data on the CPU. We calculate the orientation of the scan line with respect to the view plane and the light direction. Exploiting the orientation, we derive the necessary amount of fragments on the scan line to span the whole view plane in the desired direction. As we have shown in the previous sections, we can exploit a simple Bresenham rasterization for determining the consecutive pixels of each fragment. Further, the rasterization is valid for all fragments, so we pre-calculate a fixed series of progression steps for the fragments and use these later on on the GPU. To finalize the preprocessing, we compute depth and width of the shadow map as was described. In the next preparational step, we launch several workpackages on the GPU each of which initializes per-fragment-data and launches actual execution later on. This process is shown in algorithm 2. We need to reset the shadow map as well as set up the parameters of the first ray of each scan line fragment. Next, we emit one workpackage per scan line fragment. This leads us to algorithm 3, which depicts the part responsible for actual ray casting and shadow map updates. We determine the required amount of iterations to span the whole gap between the near and far hit of the bounding box with respect to the number of parallel samples per workpackage. In each iteration we first perform normal ray casting including local illumination before compositing the shadow map from the last step with the currently calculated value for a certain position in a back to front fashion. Consecutively, we apply

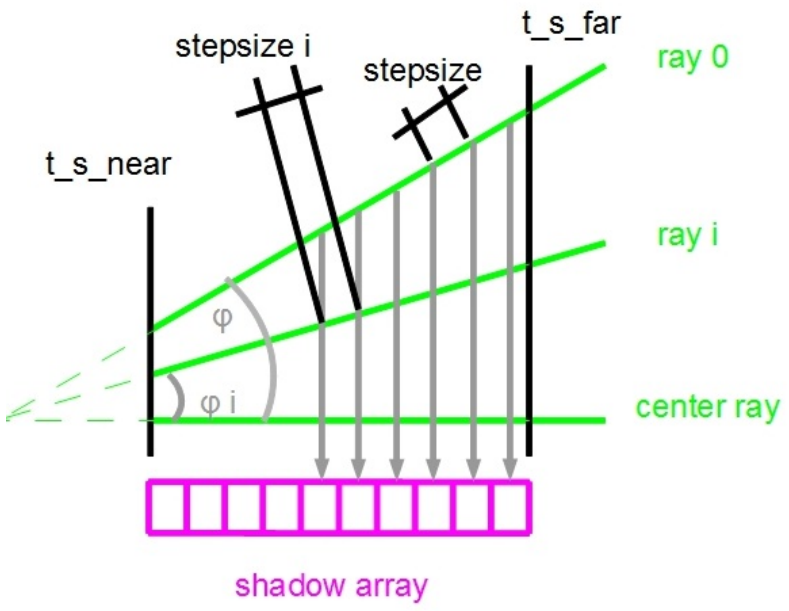


Figure 3.9: Projection of the samples during the sweep of bunch of rays related to a single fragment onto the scan line. Accommodated stepsize in ray casting contributes to conflict-less memory access.

the currently active information from the shadow map with the currently active sample, leading to the desired global illumination effect. Note that in our current implementation the shadow map consists of color and opacity to achieve a realistic colored shading. After finalizing the color and alpha information for the current sample, we perform reduction of these values over all threads of the workpackage. This control flow has a highly preferable memory footprint as we need not carry over color and opacity to the next iteration. Finally, after all iterations were processed and we thereby finish the whole ray, we update the parameters required for the next ray and emit a workpackage responsible for filling the next pixel of the current scan line fragment. Obviously, re-emission of a workpackage after finishing the current one is superior to looping over the pixels in a single fragment. The described approach leads to a re-evaluation of the workload distribution both in Softshell and CUDA, while rigid looping can lead to imbalances in SMP occupancy.

Algorithm 1 Preprocessing operations. May be called before each frame, e.g. due to camera or light transform

```
procedure SHADOW MAP SETUP($Camera, ViewPlane, LightDir, BBox, stepSize$)
    $scanLineAngle = angleInViewPlane(ViewPlane, LightDir)$
    $dominatingAxis = getDominantAxis(scanLineAngle)$
    if $dominatingAxis == x$ then
        $numFragments = windowWidth/cos(scanLineAngle)$
    else if $dominatingAxis == y$ then
        $numFragments = windowHeight/cos(scanLineAngle)$
    end if
    $setupBresenham(ViewPlane, scanLineAngle)$

    $tMapNear, tMapFar$
    $bBoxExtent(ViewPlane, BBox, tMapNear, tMapFar)$
    $cosViewPlaneCorner = dot(Camera.viewDir, (ViewPlane.c1 - Camera.pos))$
    $maxFrustumExtent = (tMapFar - tMapNear)/cosViewPlaneCorner$
    $shadowMapDepth = maxFrustumExtent/stepSize$
    $shadowMapWidth = numFragments$
end procedure
```

Algorithm 2 Initialization tasks on the device. These operations are necessary before actual rendering, but work on the GPU in parallel already. The input data shown is actually stored on the GPU rather than functional input.

```
Require: $Camera, ViewPlane$
Require: $Volume, BBox$
Require: $Scanline, NumFragments$
  procedure WORKPACKAGE:INIT($ShadowMap$)
      for all $fragment \in NumFragments$ do
          $clearShadowMap(ShadowMap)$
          $pixelPos = firstPixel(fragment, Scanline.Bresenham)$
          $rayStartPos = startPos(Camera.pos, ViewPlane, pixelPos)$
          $rayDir = normalize(rayStartPos - Camera.pos)$
          $emitSoftshellRayWorkpackage(rayDir, Scanline.Bresenham)$
      end for
  end procedure
```

Algorithm 3 Workpackage performing ray casting and global illumination update per pixel (fragment) on the GPU. The algorithm shows the code executed per thread of the workpackage

```
Require: $RayDir, StartPos, NumSamples$
Require: $TNear, TFar, StepSize$
Require: $SMap$
  procedure WORKPACKAGE:RAYCAST
      $numIt = ((tFar - tNear)/stepSize)/samplesPerWP$
      $colorAcc = \{0, 0, 0\}, alphaAcc = 0$
      for $it = 0 \cdots numIt$ do
          $currPos = StartPos + (it * SamplesPerWP + threadID()) * RayDir$
          $currColor, currAlpha = transferFunction(currPos)$
          $applyLocalIllum(currPos, currColor)$

          $sMapIndex = projectOntoSMap(currPos)$
          $updateSMapBackToFront(sMapIndex, currColor, currAlpha)$
          $composite(currColor, currAlpha, SMap[sMapIndex])$

          $colorAcc, alphaAcc = parallelReduction(currColor, currAlpha)$
      end for

      if $threadId() == 0$ then
          $newRayDir, newStartPos = progressScanLine()$
          $newStepSize = updateStepsize(StepSize, newRayDir)$
          $emitSofftshellRayWorkpackage(newRayDir, newStartPos, newStepSize)$
      end if
  end procedure
```

Chapter 4

Particle Based Volume Rendering

4.1 Methodological Overview

The main idea of PBVR is to construct a dense field of light-emitting, opaque particles inside a volumetric dataset. The field resembles a discrete sampling of the volumetric data and is used in consecutive rendering. Rather than aligning samples to the pixels of the screen, as used in image-based approaches, the particles randomly distribute over the whole volume, only related to the underlying grid in terms of interpolating the scalar value. By simulating the light emission of these opaque particles with respect to mutual occlusion, we perform object-based volumetric rendering. Sakamoto [42] and Csébfalvi [11] provide a good overview and mathematical background on the topic, but we will mainly focus on our adaption of the algorithm.
PBVR involves two major topics. First, a proper particle distribution inside the volume needs to be generated. Section 4.1.1 provides detail on our generation approach. Second, those particles need to be projected onto the image plane, which we describe in Section 4.1.2. Finally, we will describe how to optimally adapt the base method for Softshell.

4.1.1 Particle Generation

We propose a probabilistic process to generate a proper particle distribution. Uniformity of the distribution over the volume is a very desirable effect. Non-uniform distributions can easily lead to unwanted and disturbing visual artifacts such as streaks or very dense regions (clusters) and very sparse regions (holes) respectively. We further aim at parallelization of the generation process as well as superb computational performance for on-the-fly generation instead of moving it to preprocessing. Parallelization requires spatial subdivision of the volume into manageable parts. We choose to focus on tetrahedral grids due to the easy conversion to this representation. The converse, e.g. voxelization of tetrahedral grids, generally introduces an error caused by necessary interpolation. Further, tetrahedral grids allow for a locally adaptive level of detail in the base grid. On the fly generation of the particle field offers some desirable benefits. Firstly, the preprocessing is minimal, leading to almost immediate rendering. Secondly, we need not store the particles on the GPU as we simply recreate them each frame.

Particle Distribution over Cells Initially, we define a maximum for the number of generated particles, denoted as p_{max}, for the whole model. Note that, given the probabilistic nature of the approach, the maximum amount of particles is rarely fully exhausted. Exploiting the grid structure of the input volumes, we now determine the amount of particles disposable in regard to specific cells, p_{cell}. Recalling the uniformity criterion, the proportion of cell volume V_{cell} to the total volume of the grid V_{grid} is an obvious choice. Therefore, the number of particles per cell is

$$p_{cell} = V_{cell}/V_{grid} * p_{max}. \tag{4.1}$$

If we generate particles per cell using 4.1, we clearly do not distribute particles in a strict uniform fashion throughout the volume. At a first glance, this might look like a deficiency. Rather than distributing single particles, this approach beneficently separates the volume in regions (i.e., the tetrahedrons) with a fixed density (i.e., p_{cell}) resembling the expected value of particles of a total uniform distribution. The benefits of this method are at hand. Firstly, parallelization is only possible for distinct regions. Further, during parallelization, the workload distribution is invariant except for updated parameters. Next, we need to consider particle distribution over cells and resulting influence on the global distribution. Requirements for those single distributions do not change - uniformity stays a necessity. The exact modalities of uniformly random positioning inside the boundaries of tetrahedrons will be treated in the next few paragraphs. Examining the concatenation of such single particle fields yields the following observations. Firstly, global behavior of this distribution can barely be distinguished from a real uniform distribution. Following the idea of splitting average densities over spatial subregions, the resulting field does represent one possible outcome of a global uniform distribution. Secondly, overall uniform distributions still possess an inherent, albeit small, probability of generating clusters. Although the per-cell uniform fields are no exception, they operate on a completely different spatial scale. Visual artifacts produced by a single cell generally occupy only a very small portion of the screen, in contrast to deficiencies in global distributions possibly covering larger parts. Not only clustered particles are problematic in a global scenario. Considering completely unrestricted positioning of particles, distributions used for any two consecutively rendered frames are likely to vary quite a lot. Using two such vastly different distributions suggests considerable visual movement in the volumetric data. Contrary, using the mean values harness the generation process more tightly, efficiently removing such suggested movement.

Particle Position Any assumptions of specific parameters of the cells lead to restrictions concerning the usable input data or further preprocessing. Avoiding both cases, we use a most general approach for describing positions within tetrahedrons. Barycentric coordinates perfectly adapt to our requirements, only depending on the vertices of the cell. Fortunately, barycentric description of a point in three dimensions is unique for tetrahedrons, which are also referred to as simplex. Barycentric coordinates for polytopes, three dimensional polyhedrons with more than four vertices, generally represent non-uniquely determined points. Let us denote a tetrahedron as a set of four vertices:

Definition 4.1.1

$$T = \{V_i, i \in \{1,2,3,4\} \| V_i \in \mathbb{R}^3\} \tag{4.2}$$

and the barycentric coordinates describing a point P relative to a tetrahedron T:

Definition 4.1.2

$$b_i \in \mathbb{R}, i \in \{1,2,3,4\} \tag{4.3}$$

$$P = \sum_{i=1}^{4} b_i * V_i \tag{4.4}$$

The scalar parameters b_i describe the contribution of a vertex V_i to a point in a barycentric sense. No restrictions of the barycentric parameters in equation 4.4 induce positioning of points throughout all of $\mathbb{R}^3$. Constructing solely those points positioned within the boundaries of a tetrahedron is possible by introducing the following constraints:

Definition 4.1.3

$$\forall i \in \{1,2,3,4\} : b_i \geq 0 \tag{4.5}$$

$$\sum_{i=1}^{4} b_i = 1 \tag{4.6}$$

From 4.6 we deduce that any one of the b_i can be defined via the remaining three:

$$b_i = 1 - \sum_{j \neq i} b_j \tag{4.7}$$

Continuing from 4.7, we additionally infer the benefit of only having to generate three of the parameters randomly.

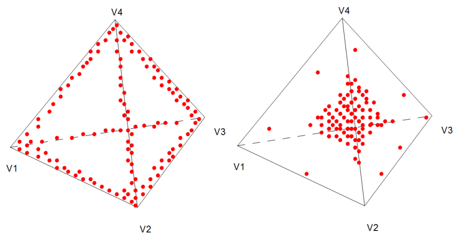


Figure 4.1: Generated particles clustered near the edges of a tetrahedron on the left and near the center on the right. This effect is created due to using an incorrect distribution

Random generation of the four parameters is possible in many ways. Although quite a few approaches sham statistically correct average values, many of them still introduce disturbing visual patterns. The most straight-forward way is random generation of all four parameters and dividing them by their sum. We can express this as:

Definition 4.1.4 *Barycentric parameters generated as random numbers over the interval* $(0,1)$ *with consecutive division by their sum:*

$$b_i = \frac{X \sim U(0,1)}{\sum_{i=1}^{4} E(b_i)} \tag{4.8}$$

$$E(b_i) = \frac{0.5}{2} = 0.25 \tag{4.9}$$

However, the particles tend to concentrate in the cell centers, inducing disturbing visual artifacts. The border regions of the cell remain very sparse. Another candidate method directly exploits the summation criterion. After generation of one parameter, the remaining constraint is updated. Consecutively, we again denote the expected values of each parameter:

Definition 4.1.5 *Barycentric parameters generated as random numbers exploiting usage of an updated remainder:*

$$E(b_1) = 0.5, E(b_2) = 0.25, E(b_3) = 0.125, E(b_4) = 0.125 \tag{4.10}$$

Randomizing the mapping of vertices to parameters asymptotically equalizes the expected values to 0.25. Still, this method suffers from a problem similar to the straight-forward method. Particles cluster at the edges connecting the vertices while central areas remain sparse. Figure 4.1 illustrates both problematic approaches. While examining both of the approaches more closely, we deduce a further requirement we did not consider yet. The generated parameters, in both cases, are *not* mutually independent. In definition 4.1.4, each parameter relies on the sum of the grand total, while in 4.1.5, the interdependence obviously results from restricting the intervals. We infer that mutual dependence results in visually perceivable tendencies and incorporate these observations in the final approach.

Rocchini *et. al.* [41] propose a correct method for generating random points in tetrahedra. The method bases on folding a cube and its parameter space into a tetrahedron in two steps. This procedure boils down to the following definition:

Definition 4.1.6 *Barycentric parameters generated as random numbers in a cube and consecutively folded twice to result in a tetrahedral space:*

$$(b_1, b_2, b_3) = \begin{cases} (b_1, b_2, b_3) & b_1 + b_2 + b_3 \leq 1 \\ (b_1, 1 - b_2, 1 - b_2 - b_3) & b_1 + b_2 + b_3 > 1, b_2 + b_3 > 1 \\ (1 - b_2 - b_3, b_2, b_1 + b_2 + b_3 - 1) & b_1 + b_2 + b_3 > 1, b_2 + b_3 \leq 1 \end{cases} \tag{4.11}$$

Summarizing, the particles are generated uniformly distributed in a parallelepiped. Points outside of the tetrahedron are transformed inside. This method generates a patternless uniform random distribution of coordinates within the constraints of barycentric parameters.

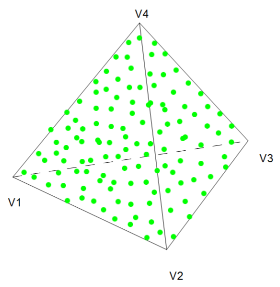


Figure 4.2: Uniform random distribution of particles over a tetrahedral cell showing no visual patterns

Particle Scalar Value Particle positions alone are not enough for volume rendering yet. We further need to calculate the scalar value correlated to the spatial position we just described. Given we operate on unstructured grids, bilinear spatial interpolation as used for regular voxel grids (see section 3.1.2) is beyond question. Instead, we define a method relating a particle's scalar value to the vertices of the tetrahedron it belongs to:

Definition 4.1.7 *Particle scalar value in relation to the vertices of the corresponding tetrahedron:*

$$\begin{aligned} s_P &= \alpha_1 * s_{V_1} + \alpha_2 * s_{V_2} + \alpha_3 * s_{V_3} + \alpha_4 * s_{V_4} \qquad (4.12)\\ &= \sum_{i=1}^{4} \alpha_i * s_{V_i} \\ &\textit{subject to} \\ &\forall i \in \{1,2,3,4\} : 0 \leq \alpha_i \leq 1 \qquad (4.13)\\ &\sum_{i=1}^{4} \alpha_i = 1 \qquad (4.14) \end{aligned}$$

In a sense more related to three dimensional space, definition 4.1.7 resembles a linear interpolation of scalar values corresponding to corners of a tetrahedron. We observe that definition 4.1.7 is highly related to 4.4 combined with 4.6. In fact, we can not only use the parameters for spatial linear interpolation of positions, but arbitrary values related to the vertices. Fortunately, a particle's scalar value requires exactly such a spatially related interpolation, which leads us to simply re-using the already available interpolating influence quantity.

Particle Emission Probability The last few sections concerned generation of a dense particle field throughout the grid in a way akin to uniform distribution. Recall that we treat particles as opaque occluders. Simply projecting the whole field without further considerations at this point would lead to representing the object boundary. Obviously, we did not take opacity into concern yet. Consecutively we need to thin out the particle field with regard to cell, and particle opacity, respectively. Additionally, avoidance of visually perceivable patterns is still of concern.

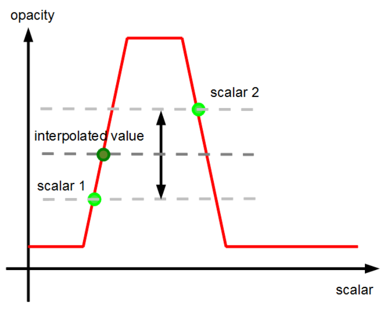
Combining the interpolated scalar value of a distinct particle with a transfer function, we initially determine the opacity of the volume at the given location. We next apply the rejection method [40] as basis of determining whether or not to actually emit and project the particle. Generally speaking, the determined opacity of a particle op_P, which is in the interval $(0, 1)$ describes the emission probability. Uniquely for each particle, we next generate a stochastic variable X within the interval $(0, 1)$ on the real line. Subsequently, we test X against op_P. Only if this test ends up positively, i.e., $X < op_P$, we accept the particle and proceed to emit and project it. If the test fails, we discard the particle. Using this method leads to adaption of the amount of particles towards the perceived density of the volume. The effect is not constrained on the cell-level. On the contrary, this method also accommodates linear transitions among opacities of vertices.

4.1.2 Particle projection and Image Generation

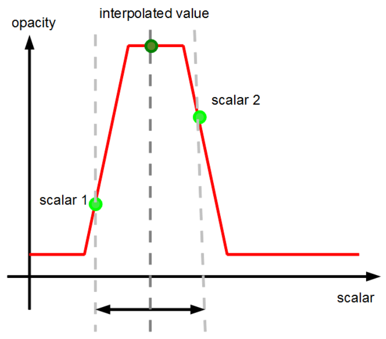
After concluding the particle generation process, we now tackle projecting the finalized particle field into image space. But not only the projection itself is of concern. Apparently, assigning a color value to each particle is just as fundamental.

Projection from Object Space to Image Space We use a standard camera model involving extrinsic (modelview matrix) and intrinsic (projection matrix) parameters. Combining both results in the so called modelview-projection (MVP) matrix incorporating homogeneous coordinates. Calculation of the image space coordinates involves multiplication of the homogeneous position of a particle with the inverse of the MVP matrix, subsequent perspective division, and finally, viewport transform. During this process, the distance to the view plane emerges as a ”by-product”. Importance of the depth buffer already appeared in the introduction, and it is now time to shed light on this circumstance. We maintain a depth-buffer (also referred to as z-buffer) with a resolution equaling that of the screen. Particles hitting the screen undergo a preliminary z-test as, for each fragment of the screen, only the particle closest to the camera is responsible for contributing to the image due to their opaqueness. Accordingly, particles closer to the camera replace previously recorded ones in the depth as well as the color buffer, effectively avoiding incorporation of sorting mechanisms.

Transfer function Our approach offers three different entry points for transfer function lookups. Two of these are well known in volume rendering and referred to as ”pre-classification” and ”post-classification” respectively. Pre-classification relates to performing classification in advance of interpolation. Accordingly, this approach requires interpolation of color values. Contrary, post-classification depends on interpolated scalar values already being present. The final color value results from performing a transfer function query with the interpolated scalar as input. In case of tetrahedral grids we interpolate the values of the corner points, be it assigned colors or scalars. In contrast to conservative volume rendering methods, our approach permits an additional option at an even later stage. In section 4.1.3 we will discuss the need for combining several pixels, or equivalently several



(a) pre-classification



(b) post-classification

Figure 4.3: The difference in interpolation between different classification strategies. Figure (a) shows pre-classification where final opacity and color values are interpolated. In (b) scalar values are interpolated using post-classification and the application of the transfer function is performed using the interpolated scalar.

projected particles, of the screen. Consecutively, classification is also possible after the mentioned step. Figure 4.3 illustrates the difference between pre- and post-classification. The last method mentioned did not attain results of the desired visual quality. Hence we prefer post-classification per default given its advantage in interpolation accuracy over pre-classification as well as its increased visual quality compared to classifying lastly.

4.1.3 Spatial superimposing

Projecting opaque particles on a view plane with the described z-buffer method neglects a lot of information required for volumetric, semi-transparent rendering. Overcoming this deficiency is solely possible by increasing the data available per pixel. While normal approaches resort to gathering information along the view direction, we use a different procedure. Instead of threading information in z-direction, we subdivide each pixel in a quadratical matrix of subpixels. This technique closely relates to supersampling, which is a commonly used technique in modern computer graphics. Capitalizing the increased resolution increases not only accuracy during projection. Furthermore, we drastically increase the information available per pixel due to multiple particles contributing to a single pixel. Here we rely on the probabilistic nature of the particle generation process. The likelihood of projected particles to hit different subpixels simultaneously while descending from diverse depth levels is of concern. Due to incorporation of transfer functions, this probability is sufficiently large for actually gathering samples in a sense which reflects volumetric data structures according to the absorption-emission model. Let us denote the (quadratic) number of subpixels per pixel as $l * l$ where l abbreviates the so called subpixel level. This parameter decisively describes the amount of spatial information gathered per pixel. We fully exploit the available samples by averaging over all subpixels for each pixel, including the unused ones. Including the non-present information, in this case, is crucial for the success of the technique. Subpixels that were not hit contribute in a sense reflecting the transparency of the volumetric data embracing a transfer function if we presume those subpixels to adopt the background color. Hence, if we also include those non-hit subpixels, we also account for regions of the volume with high transparency. Finally, we achieve the well-known semi-transparent appearance of volume rendering due to the non entirely covered subpixel matrix.

Translucency Obviously, the quantity of particles hitting distinct subpixels strongly influences translucency of a corresponding pixel. Hence both the total number of particles as well as the subpixel level impact the overall translucency setting. Automatic calculation of these parameters is indeed possible, e.g. when timing constraints need to be met. Of course, the transfer function integrates into this process. Further, the area of a projected cell is an additional relevant factor. Given both parameters, it is possible to estimate the numbers of particles hitting the screen related to a certain subpixel level. Despite this possibility, our prototype implementation incorporates fixed numbers for both particle quantity and subpixel level for more accurate performance measurement.

Assuming a GPU implementation of the algorithm, we also need to discuss algorithmic impact of these parameters. Let us start with the amount of particles. Evidently, each particle needs to be generated and possibly projected, consecutively. Hence we conclude a linear coherence of particle quantity and computational performance. On-the-fly generation of the particles leaves the memory requirement untouched. Concerning the subpixel level l, we observe a quadratic relation to consumed memory. Resulting from subdividing both color and depth buffer pixels into $l * l$ subpixels, this quadratic increase clearly bypasses the particle quantity in considerations related to automatic parameter adaption as well as available hardware.

Particle Depth Enhancement Until now, we do not incorporate a mechanism adapting the contribution of single particles to the total pixel value. Neglecting this problem leads to problems in visual perception, especially during rotations. Similar to front face culling, the rendered image suggests the presence of structures in the back of the volume. This mostly happens in situations where the amount of particles may be slightly too low for the chosen subpixel level. Methods relying on sorted samples, such as ray casting, inherently exhibit adaption of successive samples based on consumed transparency. Unfortunately, our approach lacks this feature, leading to the requirement of approximating the described effect. One possible way is sorting the gathered samples on basis of a pixel according to their depth. Subsequently, compositing as e.g. used in ray casting is possible. We promised an algorithm free of sorting, leading to a different point of view on the compositing process. Instead of treating the particles as light-absorbing, semi-transparent entities, we stick to the model of opaque particles. We now modify the absorption model accordingly. Basically, we assume a linear loss of energy for the particles traveling towards the view plane. Possible interpretations of this assumption reflect the non-empty space and thereby either slowing down the particle during projection, or otherwise possible loss of luminance caused by the semi-transparency. Due to the absence of a metrical order of the particles, we exploit availability of depth information. Relative to the distance covered, we weigh a particle's influence on the final pixel correspondingly. Obviously, particles, which origin further away from the viewing camera lose more energy on their way towards the screen than those closer to it. Furthermore, particles usually traverse a certain amount of empty space. This region not necessarily only spans the distance from view plane to the boundary of the volume. Empty space also results from incorporation of transfer functions. Cells with a very low, or even zero opacity further contribute to the void regions. Due to frequently changing transfer functions as well as rotation of the view point, the chosen mechanism needs to provide capabilities for rapid adaption to the changing circumstances.

To overcome all the posed issues, we analyze the recorded depth of each occupied subpixel z_{curr} and store minimum z_{min} and maximum depth z_{max} over the subpixel matrix of a pixel. Next, we determine the depth range corresponding to these values and perform a linear interpolation:

$$\zeta = (z_{max} - z_{curr})/(z_{max} - z_{min}) \tag{4.15}$$

ζ now describes the weight for a certain subpixel. The described approach covers all the topics of this section: empty space does not cause a loss in energy, be it regions not covered by the volume at all, or cells emitting (close to) no particles, as the weights of the closest particles is close to one. Further, particles adaptively lose energy according to the distance traveled.

4.2 Parallelization

In the last sections we provided a detailed description of how we generate the necessary particle field and how we efficiently aggregate the information into a quasi-3D representation. This section treats of describing a basic parallelization approach. Concerning the preprocessing depicted in the pseudo code 4, we see that it could be parallelized easily upon obtaining the total volume of the input. But, given the procedure is a nonrecurring and computationally non-demanding task, parallelization is indeed possible, but not crucial when having the overall performance in mind.

Algorithm 5 illustrates a summary of the related sections 4.1.1 through 4.1.2. Reflecting on the description of this recurring part - the rendering itself - the parallelization

Algorithm 4 Preprocessing: Determine maximum number of particles for each cell

```
1: procedure PREPROCESS($maxParticles$, $totalVolume$, $cells[\,]$, $particleCount[\,]$)
2:     $n = maxparticles$
3:     $v = totalvolume$
4:     for all $cells\ c_i \in cells[\,]$ do
5:         $v_i = calculateVolume(c_i)$
6:         $particlesCount[i] = n * v_i / v$
7:     end for
8: end procedure
```

capabilities of the proposed algorithm are obvious. Exploiting the data parallelism induced into the particle generation and projection process, we treat each of the volumetric data cells as a single thread on a GPU.

Algorithm 5 Generation and Projection in a strongly simplified form

```
Require: clear $colorBuffer$, $depthbuffer$
Require: $transferFunction$, $subpixelLevel$, $MVPmatrix$
 1: procedure PARALLELRENDERING($cells[\,]$, $particleCount[\,]$)
 2:     for $cellId = 0$ to $numCells$ do
 3:         $nExp_i = calculateExpectedNumParticles(cells[cellId])$
 4:         if $nExp_i > spaceSkippingThreshold$ then
 5:             $CUDAProcessCell(cells, particleCount, transferFunction, cellId)$
 6:         else
 7:             $continue$
 8:         end if
 9:     end for
10: end procedure
11: procedure CUDAPROCESSCELL($cells[\,]$, $particleCount[\,]$, $cellId$)
12:     for $i = 0$ to $particleCount[cellId]$ do
13:         $p_i = generateBaryCentricPosition(cells[cellId])$
14:         $s_i = interpolateScalar(cells[cellId], p_i)$
15:         $opacity_i = lookupOpacity(s_i, transferFunction)$
16:         if $rand \in (0,1) \leq opacity_i$ then
17:             $projectParticleOntoBuffer(p_i, s_i, transferFunction)$
18:         else
19:             $continue$
20:         end if
21:     end for
22: end procedure
```

The final superimposing step (refer to section 4.1.3) shows no direct algorithmic correlation to the previous parts and needs to be treated detached from the previous phases. From algorithm 6, we infer that each final pixel value may be computed completely independently from the values used by other pixels. Again, we have perfect data parallelism, leading to strong parallel capabilities.

Algorithm 6 Superimposing in a strongly simplified form

```
procedure SUPERIMPOSING($oututImage[\,][\,], pixelBuffer[\,][\,]\ depthBuffer[\,][\,]$)
    $l = subPixelLevel$
    for $x = 0$ to $width$ do
        for $y = 0$ to $height$ do
            $col = average(x * l \cdots (x+1) * l - 1, y * l \cdots (y+1) * l - 1)$
            $outputImage[x][y] = col$
        end for
    end for
end procedure
```

4.3 Shortcomings & Improvements with Softshell

The algorithms 4 and 6 are not qualified for further optimizations using Softshell. Both do not suffer from thread divergence due to loops, and both do not offer clear prioritization of certain parts. In algorithm 4 there is no loop at all, and we need to calculate all the volumes anyway. In algorithm 6, the two outer loops involving x and y represent the parallelization. The function, which averages over the subpixels corresponding to a single pixel, again consists of two nested loops. But the task of this function is to average over *all* subpixels, and thus performs exactly the same operations per pixel (i.e. thread). Further, the importance, and thereby a possible prioritization, of a single pixel over another one cannot easily be determined (there might not be a differentiation in importance at all). Additionally, neither of those algorithms needs to initiate additional computation depending solely on currently finished parts, which further limits exploitation of Softshell's capabilities.

Quite the opposite is the case for algorithm 5. Consider the loop responsible for particle generation in a single cell:

```
for $i = 0$ to $particleCount[cellId]$ do
    ...
end for
```

We have previously shown that the amount of particles for a given cell, denoted as $particleCount[cellId]$, is directly tied to the spatial size of the cell. If we consider grids from arbitrary sources, such as physical simulations, the cell sizes may show a very irregular behavior. Consecutively, the particle count varies by the same factor. Thus we come to the conclusion that, for many practical use cases, this loop may lead to drastic thread divergence in an unpredictable amount of blocks on the GPU.
Furthermore, the loop exhibits interlaced divergence due to the probabilistic rejection of particles at arbitrary iteration steps. Ideally, we want to completely avoid the generation of particles we will not project anyway. Doing so not only fights the mentioned divergence, but also reduces the overall count of loop iterations in almost all cases, short of completely opaque cells.

4.3.1 Transformed generation process

We will now proceed to transform the generation process from a uniform distribution with a rejection model towards generation of only those particles actually considered for projection. In section 4.1.1 we elaborated the impact of vertex scalar values, and especially the corresponding opacity from a transfer function, on a particle's emission probability. Summarizing, particles closer to a vertex related to a high opacity, have an adequate high probability of projection. In other words, vertices assuming sufficiently high opacity values attract the essential particles. We further observe the expected value of particles projected by a cell.

Definition 4.3.1 *Definition of the average number of projected particles for a given cell using a transfer function*

$$opacity_i = transferFunction(s_{V_i}) \tag{4.16}$$

$$E(p_{cell,projected}) = p_{cell} * \frac{1}{4}\sum_{i=1}^{4} opacity_i \tag{4.17}$$

Let us proceed through the necessary steps to derive definition 4.3.1. First, recall the the barycentric coordinates, which we use in particle generation. Barycentric coordinates inherently posses the Lagrange property as well as partition of unity.

Definition 4.3.2 *Interpolation properties of barycentric parameters*

$$s\colon V_i \in \mathbb{R}^3 \to \mathbb{R} \qquad \textit{(scalar field related to vertices)} \tag{4.18}$$

$$s_b(x) = \sum_{i=1}^{4} b_i * s(V_i) \qquad \textit{(Lagrange property)} \tag{4.19}$$

$$\sum_{i=1}^{4} b_i = 1 \qquad \textit{(partition of unity)} \tag{4.20}$$

From definition 4.3.2 we infer the equivalence of barycentric coordinates to an automatically normalized linear interpolation of an underlying function. This function represents the discrete scalar field of the input grid. Injectivity of the piecewise linear transfer function preserves these properties for the opacities as well. Due to the one-to-one relation between opacity and projection probability, the properties hold here as well. Definition 4.3.1 now directly follows from 4.3.2. To retain the probabilistic nature, we interpret 4.3.1 as average of a Gaussian distribution. Concerning σ of the Gaussian, we need to accommodate to the circumstances of the basic values. That is, cells with low average opacity should still, with a high probability, emit only very few particles, while cells close to the maximum must not excess this value. Consecutively, σ must depend on the proximity to the extrema. It is known that 99.7 percent of the probability mass of a Gaussian distribution are contained within the interval of $3 * \sigma$ to the left and right of the mean value. We deem this interval accurate enough to fit our needs. Let us now examine the situation around the extremal points.

4.3.1.1 Number of actually required particles

Under all circumstances, we need to avoid generation of an excess amount of particles, i.e. more than p_{max}. We also need to ensure disqualification of a negative number of particles for obvious reasons. Let us denote the average projection probability, equivalent to the average opacity, as $P_{proj}(cell_i)$. Relating this probability to 4.3.1 via equivalence, we receive a probabilistic measure for the average number of projected particles. Next we examine the relation between the average number of particles, maximum and minimum quantity of particles and σ. Therefor, we focus on three cases.

Case 1: $P_{proj}(cell_i) \sim 0$ Cells possessing this probability in most cases do not project particles at all. Especially when the approximation becomes equality, no particles qualify for emission, i.e. $E(p_{cell,projected}) = 0$. Subsequently, the Gaussian distribution must reflect this property as well. Therefor, we choose the $3 * \sigma$ interval:

Definition 4.3.3 *Standard deviation for* $P_{proj}(cell_i) \sim 0$

$$\begin{aligned} 3 * \sigma &= p_{max,i} * P_{proj}(cell_i) - p_{min,i} && (4.21) \\ &= p_{max,i} * P_{proj}(cell_i) - 0 \\ &= E(p_{cell,projected}) \\ \Rightarrow \sigma &= \frac{1}{3} E(p_{cell,projected}) && (4.22) \end{aligned}$$

We exploit the property of the minimum number of particles to be zero without exception. It is always possible, albeit unlikely, that not a single particle passes the rejection test. From definition 4.3.3, we see that the probability mass concentrates around the average value due to σ evaluating to a very small number. Actually, we define the probability mass from the left hand side and mirror it to the right. In the special case of $\sigma = 0$, we bypass the Gaussian adaption and proceed with using the unmodified, average value.

Case 2: $P_{proj}(cell_i) \sim 1$ The observations in the second special case are very similar to the previous one. Let us again denote σ for this case.

Definition 4.3.4 *Standard deviation for* $P_{proj}(cell_i) \sim 1$

$$\begin{aligned} 3 * \sigma &= p_{max,i} - p_{max,i} * P_{proj}(cell_i) && (4.23) \\ &= p_{max,i} - E(p_{cell,projected}) \\ \Rightarrow \sigma &= \frac{1}{3} * (p_{max,i} - E(p_{cell,projected})) && (4.24) \end{aligned}$$

In this case, we utilize the distance between average number of particles and maximum number of particles, defining the Gaussian from the right side. Again, the distance can evaluate to zero. Again we bypass the Gaussian curve in this case to avoid numerical issues.

Case 3: $0 < P_{proj}(cell_i) < 1$ This is the case we generally expect to occur. Naturally, the standard deviance is tremendously larger than at the borders due to the increased freedom of single particles caused by deploying the rejection-based method. Basically anything can happen - no particles being projected, all particles, as well as anything in between. Consecutively, we want our $3 * \sigma$ region to expand as far as possible. Hence we define the general case as

Definition 4.3.5 *Standard deviation for* $0 < P_{proj}(cell_i) < 1$

$$\Delta p = max(p_{max,i} - E(p_{cell,projected}), E(p_{cell,projected})) \tag{4.25}$$

$$3 * \sigma = \Delta p \tag{4.26}$$

$$\Rightarrow \sigma = \frac{1}{3} * \Delta p \tag{4.27}$$

We immediately see that this definition of σ automatically expands the Gaussian curve maximally within the constraints. Concluding, we state the final distribution:

Definition 4.3.6 *Final Gaussian distribution*

$$p_{cell,projected} \sim \mathcal{N}(\mu, \sigma) \tag{4.28}$$

$$\mu = E(p_{cell,projected}) = p_{max,i} * P_{proj}(cell_i) \tag{4.29}$$

$$\sigma = \frac{1}{3} * max(p_{max,i} - E(p_{cell,projected}), E(p_{cell,projected})) \tag{4.30}$$

4.3.1.2 Transformed particle positions

At this point, we can assume guaranteed projection of each generated particle. Hence, we need to comply with the original property of particle emission probability related to the vertices of a given cell. Basically speaking, we want vertices with a high opacity to "attract" particles while still maintaining the probabilistic nature of the generation process. The solution of this problem is at hand. We first evaluate the opacity of each vertex of a cell

Definition 4.3.7 *Opacity of the scalar* s_{V_i} *of a vertex* V_i

$$opacity_i = transferFunction(s_{V_i}) \tag{4.31}$$

Next, we apply the "attraction" modification to the barycentric parameters by multiplying the generated random parameters and the vertex opacity.

Definition 4.3.8 *Introducing the attraction term to the barycentric parameters:*

$$\forall i \in \{1,2,3,4\} : b_i \geq 0 \tag{4.32}$$

$$\sum_{i=1}^{4} b_i = 1 \tag{4.33}$$

$$\Rightarrow b_i\prime = opacity_i * b_i \tag{4.34}$$

Without adaption, we clearly cannot use three parameters generated with 4.3.8 for calculating the remaining one anymore. Doing so with definition 4.3.8 alone would result in overestimation of the fourth parameter. As the first three opacities definitely are ≤ 1, we would effectively reduce their contribution in relation to the fourth almost surely. Hence we would violate the assumption of the attraction of vertices proportional to their corresponding emission probability. Clearly, we need to introduce another term to reflect present proportions:

Definition 4.3.9 *Modifying the attraction term to incorporate relativity to the maximum opacity:*

$$b_i\prime = \frac{opacity_i}{\max\limits_{j \in \{1,2,3,4\}} (opacity_j)} * b_i \tag{4.35}$$

The factors still reside in the interval $(0, 1) \in \mathbb{R}$ due to our early exclusion of cells with low average opacity. Due to adhering to the maximum opacity of *all* vertices we can again rely on definition 4.1.6. Even if the left out vertex corresponds to the maximum opacity of a cell, definition 4.3.8 ensures maintaining the attraction on the generated particle due to lowering the remaining three.

4.3.1.3 Finalized Algorithm

Algorithm 7 condenses the newly derived details. Elimination of divergence at arbitrary loop iterations now permits complete utilization of Softshell's thread regrouping capabilities. Fortunately, single pixels, and thereby threads, are mutually completely independent, which leads to minimal effort for data exchange upon regrouping.

Algorithm 7 New Generation and Projection method, Softshell version

```
Require: $clear\ colorBuffer$, $depthbuffer$
Require: $transferFunction$, $subpixelLevel$, $MVPmatrix$
 1: procedure WORKPACKAGE:GENERATEANDPROJECT($cells[\,]$, $particleCount[\,]$)
 2:     $nExp = calculateExpectedNumParticles(cells[cellId])$
 3:     $n = applyGauss(nExp, \sigma)$
 4:     $opacities[:] = getOpacities(cells[cellId])$
 5:     for $x = 0$ to $n$ do
 6:         $b[:]$
 7:         $b[1 \cdots 3] = uniformRandom(3)$
 8:         $b\prime[:] = b * opacities[1 \cdots 3]/max(opacities[:])$
 9:         $b\prime[:] = foldTetrahedron(cells[cellId], b\prime[:])$
10:         $p_x = generateBaryCentricPosition(cells[cellId], b\prime)$
11:         $s_x = interpolateScalar(cells[cellId], p_x)$
12:         $opacity_i = lookupOpacity(s_x, transferFunction)$
13:         $projectParticleOntoBuffer(p_x, s_x, transferFunction)$
14:     end for
15: end procedure
```

Chapter 5

Dykstra's Projection Algorithm

Two versions of this algorithm exist. A linear approach may be parallelized on the pixel-level, while the second one reaches deeper and computes the projections in parallel as well. We will shift our focus from the theoretic background of projection onto convex sets towards the algorithmic properties of both approaches. In the next sections we will perform a deep analysis of possible ways to increase performance and at the same time show the inability to employ these strategies due to hardware constraints.

5.1 Problem Definition

Before we describe the algorithms, we first need to define the input parameters. Given an input image of size $dim1 \times dim2$ the number of points each of which we need to solve the optimization problem for will be referred to as $numPoints = dim1 * dim2$. Reasonable values from real-life scenarios for these first two input dimensions are located at around 500 pixels each. Although larger images may occur as well, we will use this value as reference in upcoming considerations. Next we define the dimensionality of the feature vector, of which one exists per pixel in the image, by $dim3$. A high dimensionality of the feature vectors is a priority to be able to achieve optimal results. 512 dimensional features are desirable, but one also needs to consider the available memory upon parallelization. In an initial step we need to determine the number of involved constraints and their actual shape. A feature vector with dimensionality $dim3$ results in at most $dim3 * (dim3 - 1)/2$ constraints for the optimization problem. Algorithm 8 shows the procedure of loading the constraints for a single input point.

The constraints represent the distances of the feature vectors with respect to the convex sets. Note that in this algorithm we already treat with the intersection of more than two sets. Clearly, the upper bound of the number of constraints, $dim3 * (dim3 - 1)/2$, may be reached in a worst-case scenario, but is generally unpredictable due to heterogeneity in the input images. Resulting from the possibly vast number of constraints, not only the computational complexity of solving these needs to be considered. Memory quickly becomes a serious problem on consumer graphics hardware due to the near-quadratic memory complexity caused by the constraints.

Algorithm 8 Determine the constraints for a single point

```
Require: $numPoints = dim1 * dim2$
Require: $dim3$
 1: procedure LoadConstraints($dim3$, $p[dim3]$, $constrI1[]$, $constrI2[]$)
 2:     $numConstraints = 0$
 3:     for $i1 = 0$ to $dim3$ do
 4:         for $i2 = i1$ to $dim3$ do
 5:             $sumX = \sum_{k=i1}^{i2} p[k].x$
 6:             $sumY = \sum_{k=i1}^{i2} p[k].y$
 7:             if $\sqrt{sumX^2 + sumY^2} > 1$ then
 8:                 $constrI1[numConstraints] = i1$
 9:                 $constrI2[numConstraints] = i2$
10:                 $numConstraints = numConstraints + 1$
11:             end if
12:         end for
13:     end for
14: end procedure
```

5.2 Linear Dykstra

This first section concerns the "linear" version of Dykstra's algorithm for solving the problem shown in 9 for an arbitrary amount of convex sets. Linear, in this case, does not relate to the actual implementation of the algorithm, but rather concerns the fashion in which constraints are treated. As the name suggests, this version computes the update steps for the constraints one by one in an iterative fashion. To do so, the algorithm facilitates two auxiliary arrays p and q, which contain the working copy of the data (p) and auxiliary values used in the update procedure (q). We may assume that the constraints have already been determined in a preliminary step (see algorithm 8). Typically for optimization, we define two parameters responsible for stopping the optimization algorithm. The first one contains the maximum number of available iterations. If *maxIt* is exceeded, timely convergence is unlikely and the execution is halted. Contrary, if the algorithm converges early, i.e. the error term is below a threshold *epsilon*, we stop execution as well.

5.3 Parallel Dykstra

There is another way to fight the divergence - parallelization of the algorithm on a different level. The "parallel" counterpart to the algorithm described above is able to attend to all constraints of a single point at once - hence each point facilitates $numConstraints_k$ threads, which perform summation and update in parallel. This alleviates the issues occurring at the central loop completely. One could think of it as switching the optimization loop with the constraints loop and moving the parallelization down towards the next nested level, i.e. the constraints loop. Consecutively, the former outermost optimization loop di-

Algorithm 9 Dykstra's Projection algorithm for a single point

```
Require: $maxIt$, $epsilon$
Require: $p[\,]$, $q[\,] = \{0\}$
Require: $points\{x, y, t\}[\,]$
Require: $numConstraints_k$, $constraintsI1_k[\,]$, $constraintsI2_k[\,]$
  procedure DYKSTRA($x$)
      $numIt = 0$ , $error = 0$
      while $numIt < maxIt$ & $error > epsilon$ do
          for $ii = 1 \ldots numConstraints_k$ do
              $i1 = constraintsI1_k[ii]$
              $i2 = constraintsI2_k[ii]$
              $sum_X = (1 + i2 - i1) * q[ii].x$
              $sum_Y = (1 + i2 - i1) * q[ii].y$
              for $j = i1 \cdots i2$ do
                  $sum_X = sum_X + p[j].x$
                  $sum_Y = sum_Y + p[j].y$
              end for
              $(sum_X, sum_Y) = shrinkage(sum_X, sum_Y)$
              $e_X = q[ii].x - sum_X/(1 + i2 - i1)$
              $e_Y = q[ii].x - sum_Y/(1 + i2 - i1)$
              for $j = i1 \cdots i2$ do
                  $p[j].x = p[j].x + e_X$
                  $p[j].y = p[j].y + e_X$
              end for
              $q[ii].x = q[j].x - e_X$
              $q[ii].y = q[j].x - e_Y$
              $error = max(error, \sqrt{e_X^2 + e_Y^2})$
              $writeBack(p_X, p_Y)$
          end for
      end while
      $projectOntoParabola()$
  end procedure
```

verges only on the block (or workpackage) level instead of the warp level. Thereby, the divergence in the number of optimization passes is not an issue anymore as well. In fact, we only have the divergence on the innermost level left, which in fact is less concerning than the outer loops.
In the parallel case, we need to aggregate the results from all constraints at once and perform the update with the compound value. Determining the values contributing to the compound is easy, we simply perform the same sum as before and apply a weight corresponding to the number of active constraints to describe the relative contribution to the aggregate value. The values resulting from the constraint update can then be consolidated e.g. via parallel reduction methods.

Part III

Results

Chapter 6

Evaluation

6.1 Environment Specification

We use a consumer PC operating with an Intel i7-3770 quad core CPU with 3.4 GHz, 8 Gb of main memory using Windows 7 64 Bit. We use an NVidia GForce GTX 470 with 1280 mb dedicated video memory running driver version 306.97, using the shader model 2.0 with CUDA 5. It provides 448 CUDA cores with a processor clock of 1215 MHz and a graphics clock of 607 MHz. The GPU is connected to a PCIe x16 slot. For our two rendering algorithms we provide render timings, averaged over multiple frames per input parameter set. This is necessary not only because PBVR is probabilistic in nature, but also due to the unpredictable internal scheduling of threads in CUDA. Induced by its sheer computational complexity, we analyze Dykstra's algorithm from an algorithmic point of view rather than raw performance.

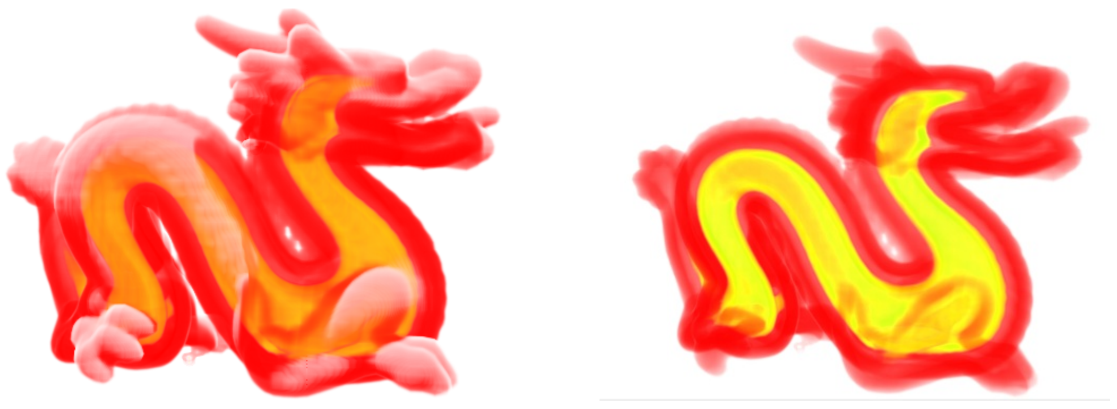
6.2 Ray Casting with advanced Illumination

This section shows the numerical evaluation of both the method proposed by Sundén *et. al.* [48] and our modified approach optimized towards Softshell. The method used in this section specifically aims at highlighting the power of the tier 2 scheduling of Softshell. For details concerning the exploited features, we refer to section 3.2.1.1 of this thesis. Besides showing numerical data recorded for two different datasets we will also elaborate the rendering environment.

6.2.1 Rendering Environment

Unfortunately, we cannot make any assumptions on the exact optimization methods used in the volume renderer facilitated in the original paper. For achieving meaningful results, we need to facilitate exactly the same underlying volume renderer. In consequence, we implement both methods on top of a very basic version of ray casting. In this environment where both methods build on top of the exact same ray caster we are able to effectively compare the performance of both illumination techniques, one using the described method employing Softshell, while the other one sets up directly on top of

CUDA.
The light source we use in our approach is directional and radiates vertically downward. While our algorithm is not impaired by arbitrary light directions due to arbitrary scan line orientation, the original approach requires an intermediate cache coherency step. In the special case of vertical lighting, we can effectively skip this intermediate step per scan line advance, leading to a most beneficial environment for the original approach.
We test two volume datasets. The first is the Stanford Dragon [45] with a 64x64x64 voxel grid and a 2 byte scalar resolution. Further, we use the Lobster CT scan [49] with dimensions 301x324x56.



(a) Global + Local Illumination (b) Local Illumination only

Figure 6.1: Dragon volume dataset rendered with global illumination and extreme specularity (6.1(a)). Note the self-shadowing caused by the jagged structures especially at the transition from head to body, which is missing in 6.1(b) where only local illumination is applied.

6.2.2 Visual results

The rendered images are, as expected, of very high quality for both datasets. We use a vertical camera opening angle of 60 degrees and a step width of 0.2 units. Note that this step width dynamically adapts to accommodate for a synchronized ray front and is actually lower in the closer a ray is to the center of the image.
Figure 6.1 shows one frame of the dragon dataset. Especially the smooth transition at the central loop from glossy material towards the head where self-shadowing slowly fades in highlights the capabilities of global illumination effects. Figure 6.2 contains the shadow map corresponding to Figure 6.1. The upper half contains the depth map as seen from the light source. The lower part depicts accumulated unshaded color used as contributing factor for the consecutive scans. Both parts represent the finalized iterative deep shadow map at the end of rendering a frame. Keep in mind that due to our invoked mechanisms the shadow map shown here actually is radially distorted around the view point of the camera. As we need not synchronize the whole ray front, but only the consecutive rays of

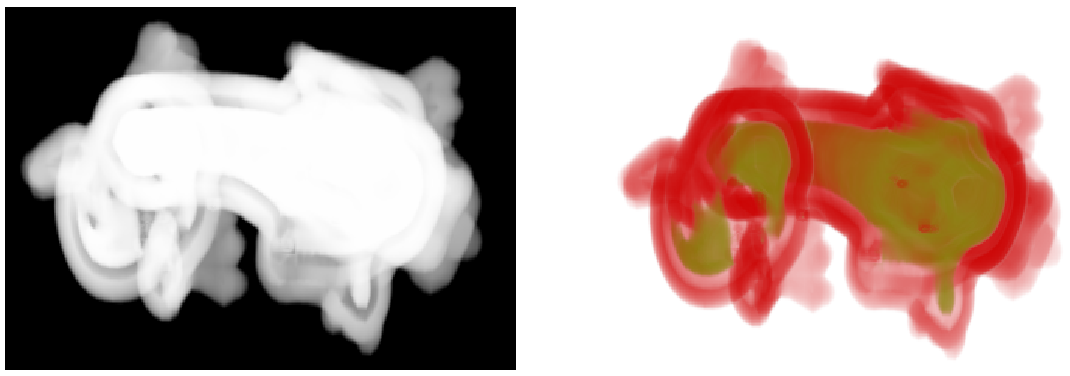

(a) Opacity Channel (b) Colour Channel

Figure 6.2: Shadow map corresponding to image 6.1. In the left image 6.2(a), we show the depth as seen from the light after the complete pass. The image to the right 6.2(b) depicts the accumulated color in the shadow map, again after a complete scan.

one fragment, the cause of this effect is obvious.
Figures 6.3 and 6.4 represent the same information as above for the lobster dataset. Unfortunately, this dataset provides only little self-shadowing due to barely overlapping structures.

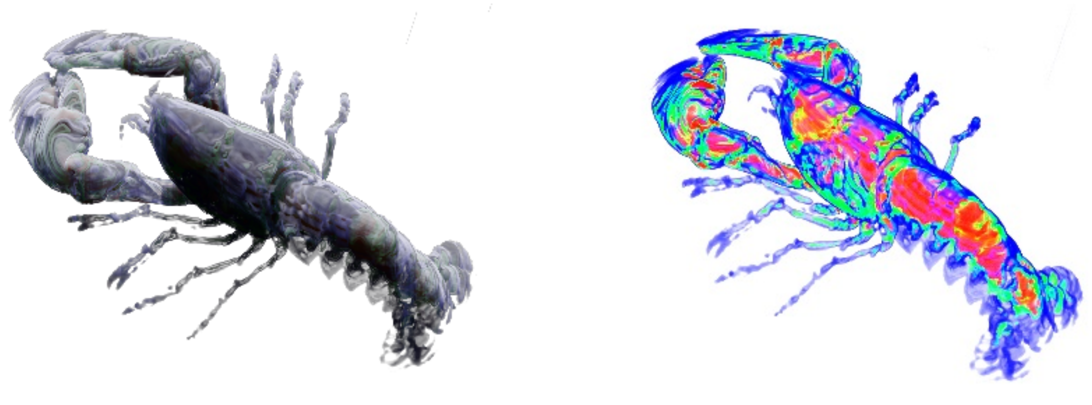

(a) Global + Local Illumination (b) Local Illumination only

Figure 6.3: Lobster volume dataset rendered with global illumination (6.3(a)). In this image we show less extreme light conditions to reflect the shadows imposed by structures in the dataset. We, again, show a comparison to local-only illumination for the same volume.

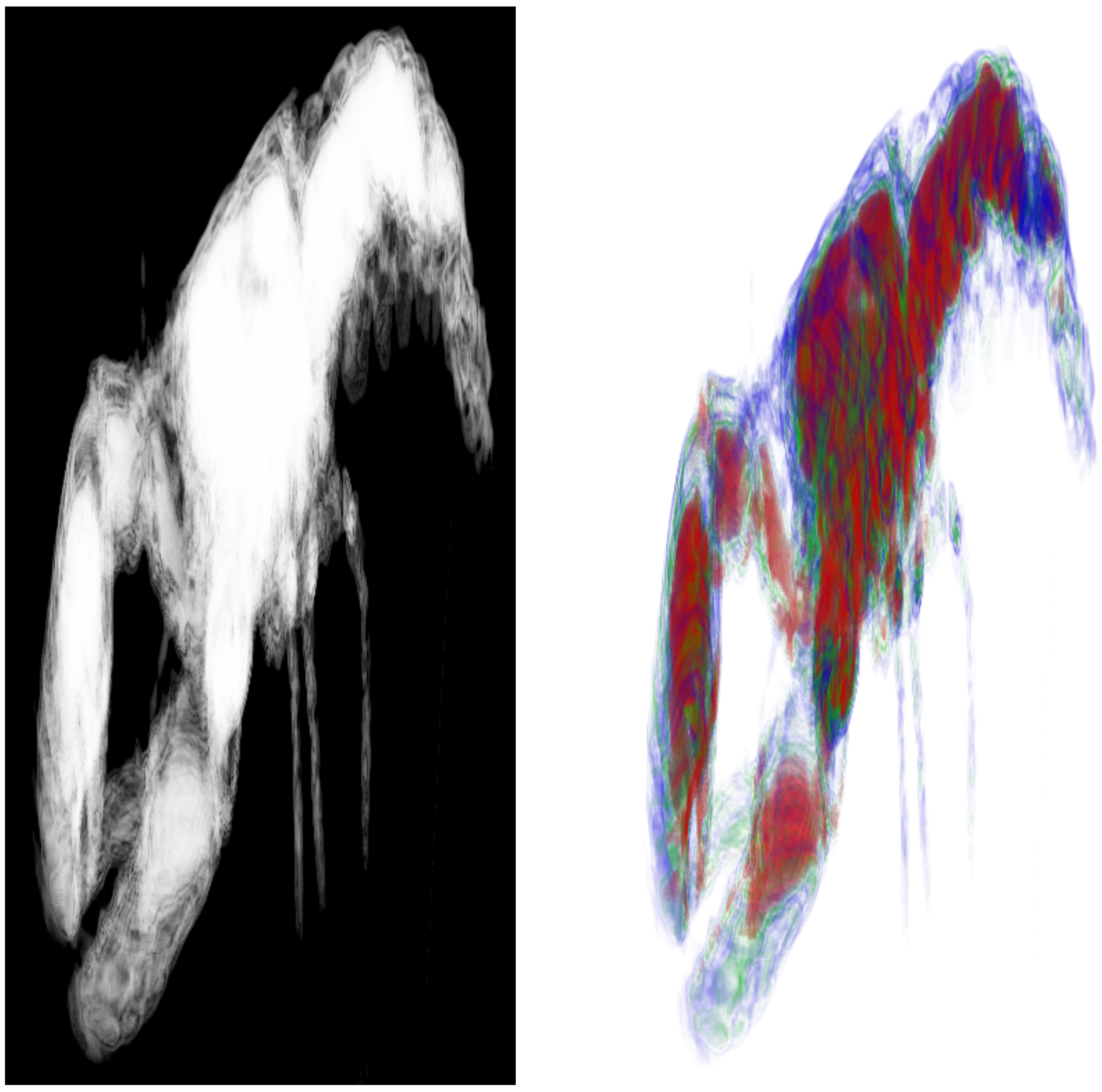


Figure 6.4: Shadow map corresponding to image 6.3. In the left image 6.4(a), we show the depth as seen from the light after the complete pass. The image to the right 6.4(b) depicts the accumulated color in the shadow map, again after a complete scan.

6.2.3 Numerical Results

Table 6.1 summarizes the average performance of both algorithms accumulated over ten frames for each resolution using the dragon dataset. It immediately stands out that our method suffers less from low utilization of the GPU resources, even at low values for window width and height. Further, we observe that due to the meager number of threads utilized in the original approach the performance is almost invariant to increasing the

resolution in the x-direction (i.e., the number of fragments on the scan line). Contrary, our approach easily circumvents this bottle neck due to the alternative parallelization route enabled by Softshell.
For the lobster dataset, for which the results are shown in table 6.2, the situation is even more severe. Especially at low resolutions the independence of the scan line fragments drastically increases the performance of our approach. The speedup varies in between 5 and 22 at the lowest number of scan line iterations (= 64) and 3.5 to 3.8 at 1024 iterations at the maximum resolution. To emphasize the impact of the improved approach, we also provide two graphs with a side-by-side comparison of the time taken for rendering in Figures 6.5 (Dragon) and 6.6 (Lobster). The Figures further highlight the benefit of scan line fragment independence. Note the distinctly more shallow curves, especially at lower resolutions.

Table 6.1: Numerical results (render time per frame in milliseconds) of both Softshell and plain CUDA for the dragon dataset. Horizontally we increase the number of fragments on the scan line (number of pixels in x-direction). Vertical increments denote the number of iterations of the scan line (image resolution in y-direction)

Softshell	**64**	**128**	**256**	**512**	**1024**
64	11,7 ms	15,1 ms	24,2 ms	41,4 ms	75,9 ms
128	20,1 ms	29,5 ms	47,5 ms	78,9 ms	141,2 ms
256	39,4 ms	58 ms	110,3 ms	189,7 ms	314,9 ms
512	79,5 ms	115,2 ms	218,4 ms	430,4 ms	756,8 ms
1024	164,9 ms	241,4 ms	458,3 ms	895,4 ms	1844,9 ms
CUDA	**64**	**128**	**256**	**512**	**1024**
64	152,8 ms	147,6 ms	144,4 ms	141,4 ms	140,3 ms
128	292,9 ms	311,5 ms	314,4 ms	285,1 ms	287,9 ms
256	580,9 ms	630 ms	742,7 ms	621,1 ms	615,3 ms
512	1153,3 ms	1288,3 ms	1537,2 ms	1534,3 ms	1530 ms
1024	2297 ms	2545,3 ms	3119,1 ms	3040,1 ms	3118,3 ms

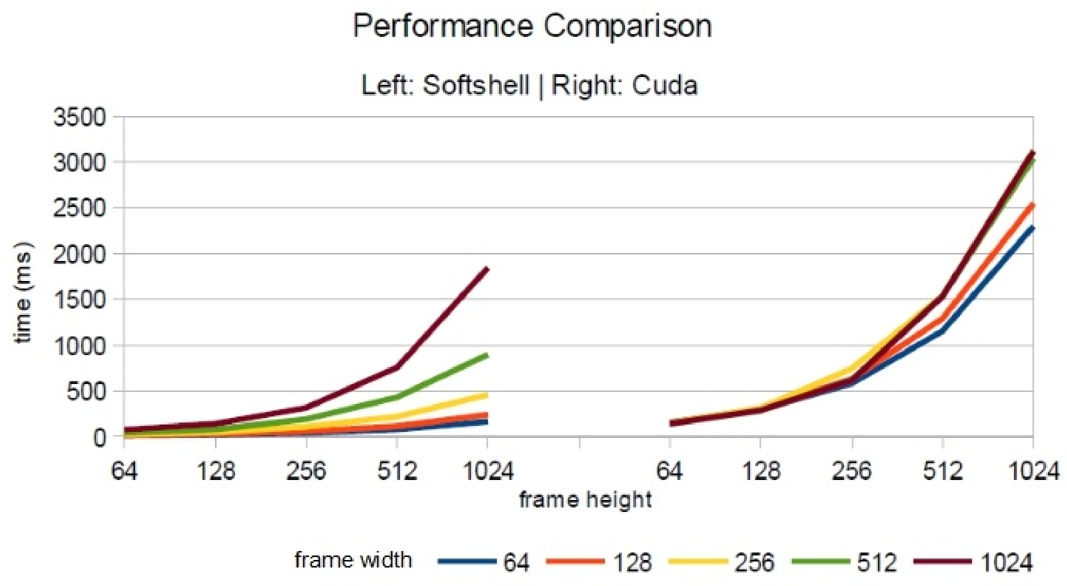


Figure 6.5: Dragon: Side-by-side comparison of the time taken per frame for both methods. Note that the seemingly exponential increase results from the exponential growth of the frame size. Each line depicts a constant value for the number of fragments on the scan line/horizontal resolution of the image.

Table 6.2: Numerical results (render time per frame in milliseconds) of both Softshell and plain CUDA for the lobster dataset. Horizontally we increase the number of fragments on the scan line (number of pixels in x-direction). Vertical increments denote the number of iterations of the scan line (image resolution in y-direction)

Softshell	**64**	**128**	**256**	**512**	**1024**
64	24,1 ms	29,4 ms	42,8 ms	64,2 ms	102,1 ms
128	42,8 ms	59,5 ms	91,6 ms	143,8 ms	217 ms
256	83,9 ms	110,4 ms	200,8 ms	337,7 ms	527,6 ms
512	161 ms	228,3 ms	417,5 ms	839,7 ms	1443,4 ms
1024	322,9 ms	433,3 ms	817,7 ms	1541,4 ms	3075,4 ms
CUDA	**64**	**128**	**256**	**512**	**1024**
64	536,4 ms	545,8 ms	541,8 ms	502,1 ms	524,1 ms
128	1044,1 ms	1194,3 ms	1211,3 ms	1055,2 ms	1056,2 ms
256	2064,7 ms	2341,4 ms	2883,6 ms	2379,6 ms	2362,5 ms
512	4122,5 ms	4655,4 ms	5752,8 ms	5790,7 ms	5800,2 ms
1024	8386,5 ms	9446,6 ms	11647,7 ms	11598,3 ms	11969,2 ms

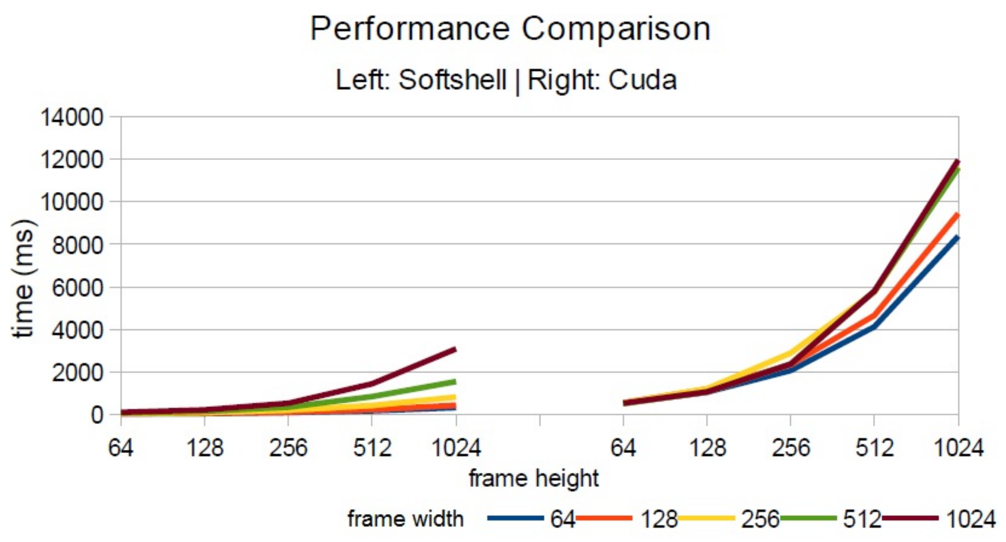


Figure 6.6: Lobster: Side-by-side comparison of the time taken per frame for both methods. Note that the seemingly exponential increase results from the exponential growth of the frame size. Each line depicts a constant value for the number of fragments on the scan line/horizontal resolution of the image.

6.3 Particle Based Volume Rendering

In this section we evaluate the PBVR algorithm presented in section 4.1 and especially the adaptions from 4.3. These changes fabricate a proper environment for Softshell's third tier scheduling by eradicating the interleaved warp divergence of the particle generation loop. We especially make a point of thread re-convergence caused by variable, bounded random particle amounts for the different cells. Note that for current GPU generations a concrete implementation of Softshell is only possible in Software, i.e. it sets up on top of a GPGPU language. Due to the lack of hardware support for the proposed concepts, each feature comes with a considerable price to pay in terms of additional computational effort on top of the actually executed algorithm. Especially thread re-convergence for loops is a costly feature as we will show in this section. Nevertheless, under the right circumstances, the caused overhead vanishes in comparison to a drastically improved overall performance.
The expected divergence of a warp of cells is directly proportional to the relative volumetric extent of cells and individual application of the transfer function on each cell. Based on this observation we now infer the worst case scenario for the algorithm. To create such a disadvantageous environment we need to minimize the overall divergence of threads. Minimal divergence results in mobilizing the re-convergence tests on the loops without actually improving performance because only a vanishing number of threads need to be re converged. Employing a uniform opacity distribution over all possible scalar values of the input volume would only make sense in few distinct, special cases of volume representation. As a consequence, the only meaningful measure we can facilitate is evening out the maximum number of particles per cell throughout the grid. In section 4.1.1 we state that each cell draws its maximum number of particles out of a pool available for the whole grid. The specific amount is proportional to the fraction of cell volume to total volume. Hence, to even out the particle proportions, all cells need to be of equal size. The easiest way to create such a grid is to simply tetrahedralize a rectilinear voxel dataset with respect to the inherent spacing in all three dimensions. Bear in mind that the actual number of particles still depends on the defined transfer function, but the homogeneous grid evens out the divergence by a large amount.

6.3.1 Visual results

The basic rendering algorithm did not change over the originally proposed methods. In Figure 6.7 we show the dragon dataset which was tetrahedralized from a voxel grid, whose performance we will examine in section 6.3.2. In Figure 6.8 we show a real unstructured grid. It depicts the simulation of cell death caused by simulated heat distribution. The performance of the algorithm using this dataset is discussed in Section 6.3.3.

6.3.2 Numerical Results - Dragon

The first dataset we consider is a tetrahedralized version of the 64x64x64 voxel grid we already used in section 6.2. After tetrahedralization and tessellation the grid consists of 1.25 million cells. The sheer amount of cells may look discouraging at first, but does

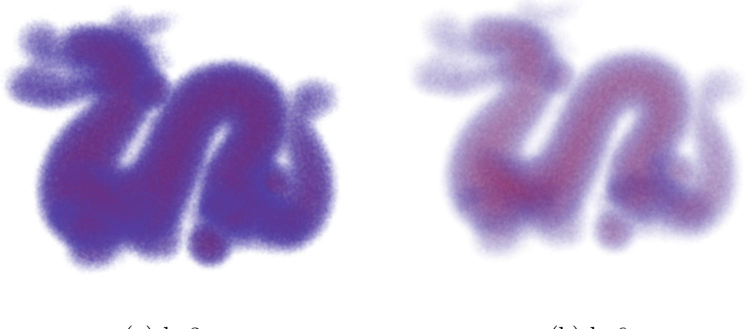

(a) l=3 (b) l=6

Figure 6.7: Stanford Dragon rendered with 60 million particles per frame, subpixel level 3 (a) and subpixel level 6 (b) on a resolution of 1200x800 pixels

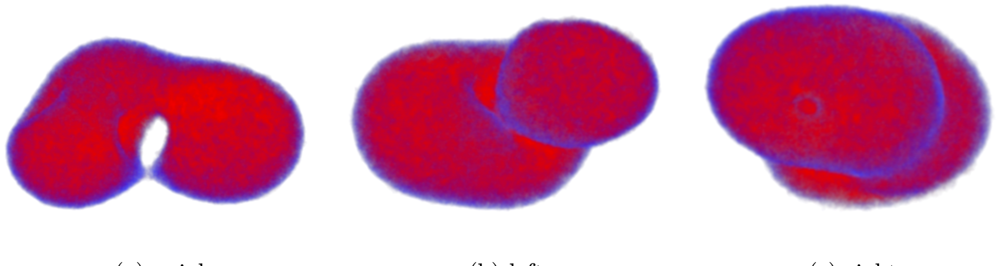

(a) axial (b) left (c) right

Figure 6.8: Tumor Ablation Simulation with 1.22 million cells in differently aligned views

not in the slightest pose a problem for the algorithm. Even the memory consumption of 78 megabytes for the volume alone easily stays within the hardware restrictions. We use a grid with such a high resolution to further reduce the divergence of the loop responsible for generating the particles. In table 6.3 we present the timing for frames rendered with certain particle pools. The render times represent execution for the same maximum amount of particles without using the tier 3 re-convergence on the left and with it on the right. The values represent average rendering time over 30 frames to accommodate to the probabilistic behavior of the algorithm. In Figure 6.9 we show the chart corresponding to table 6.3 for visualizing the transition point at which Softshell starts to outperform plain CUDA. This point is located at around two billion particles. Hence the maximum number of iterations per cell needs to be somewhere in between 1400 and 1800 for Softshell to be faster than plain CUDA.

For comparison, we use a lower resolution grid of the same voxel grid containing only roughly 300k cells. Although the relative divergence remains equal, lower tessellation increases the absolute difference in iterations with respect to the used transfer function. Therefor we expect a more favorable behavior of the re-convergence step due to the ability of actually gathering a larger quantity of active threads and ruling out more inactive ones. Examining table 6.4 leads to the conclusion that the increased absolute divergence drastically improves the re-convergence performance gain. In Figure 6.10 the two trend lines are barely distinguishable at lower particle amounts. Further, the point at which Softshell outperforms CUDA is located at a much lower particle amount. Given the large variance in time taken per frame and the already low timing, it is hard to say where exactly this crucial point is located. Again, we observe that the maximum number of particles of the conversion point is located somewhere around 1600.

Unfortunately, such large particle pools are more than unrealistic for volumes of this size. Although we observe increased performance at a certain point, much lower amounts of particles absolutely suffice for most datasets. Hence we observe that, for currently available GPUs, the third tier scheduling with its thread re-convergence in loops only increases performance if the absolute loop divergence is extremely high. Unfortunately, we cannot reproduce such a situation with the proposed algorithm and realistic input parameters for a uniformly sized collection of cells.

Table 6.3: Render times for different amounts of maximum particles for dragon dataset with 1.25 million cells. Render times given in milliseconds

Particles in million	Without T3	With T3
50	59 ms	142 ms
100	96 ms	174 ms
200	162 ms	230 ms
500	330 ms	382 ms
1000	590 ms	621 ms
2000	1095 ms	1090 ms
3000	1560 ms	1523 ms
4000	2015 ms	1942 ms

6.3.3 Numerical Results - Tumor Ablation Simulation

Unstructured grids comprise of much more than just tetrahedralized rectilinear grids. To really demonstrate the power of the third tier scheduling in a more general scenario, we examine the performance on the ablation dataset already presented in the original paper. Here, the circumstances completely differ from the previously examined grid. We again have a large quantity of cells, i.e. 1.22 million. But in contrast to the dragon dataset, the cell sizes vary vastly. The single cell sizes range from 0.1 to 111.8 volumetric units, with an average of 1.7 (note that we do not need to relate specific metrical units to the sizes). In table 6.5 we present the average timing of 30 frames with the same particle pools as before. Figure 6.11 again shows the trend line of performances with and without thread re-convergence. We observe a drastic performance increase when exploiting Softshell's tier three capabilities.

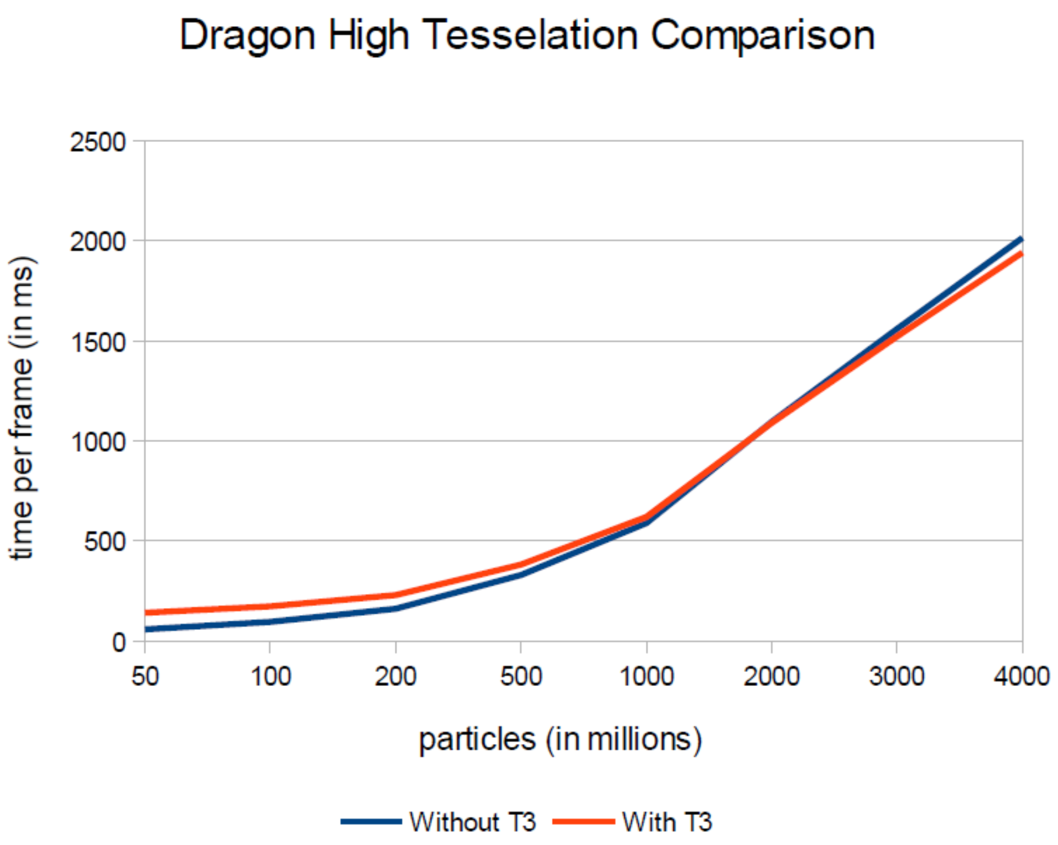


Figure 6.9: Trend line for the dragon dataset with 1.25 million cells. We observe the tier 3 model outperforming plain CUDA at around 2 billion particles.

Table 6.4: Render times for different amounts of maximum particles for the lower resolution dragon dataset with only 300k cells. Render times given in milliseconds

Particles in million	Without T3	With T3
50	31 ms	35 ms
100	56 ms	58 ms
200	95 ms	98 ms
500	209 ms	211 ms
1000	375 ms	371 ms
2000	695 ms	691 ms
3000	998 ms	980 ms
4000	1320 ms	1230 ms

The explanation of these extreme results is as follows. Without thread re-convergence, CUDA suffers from the strongly differing loop iteration counts (with a ratio of up to

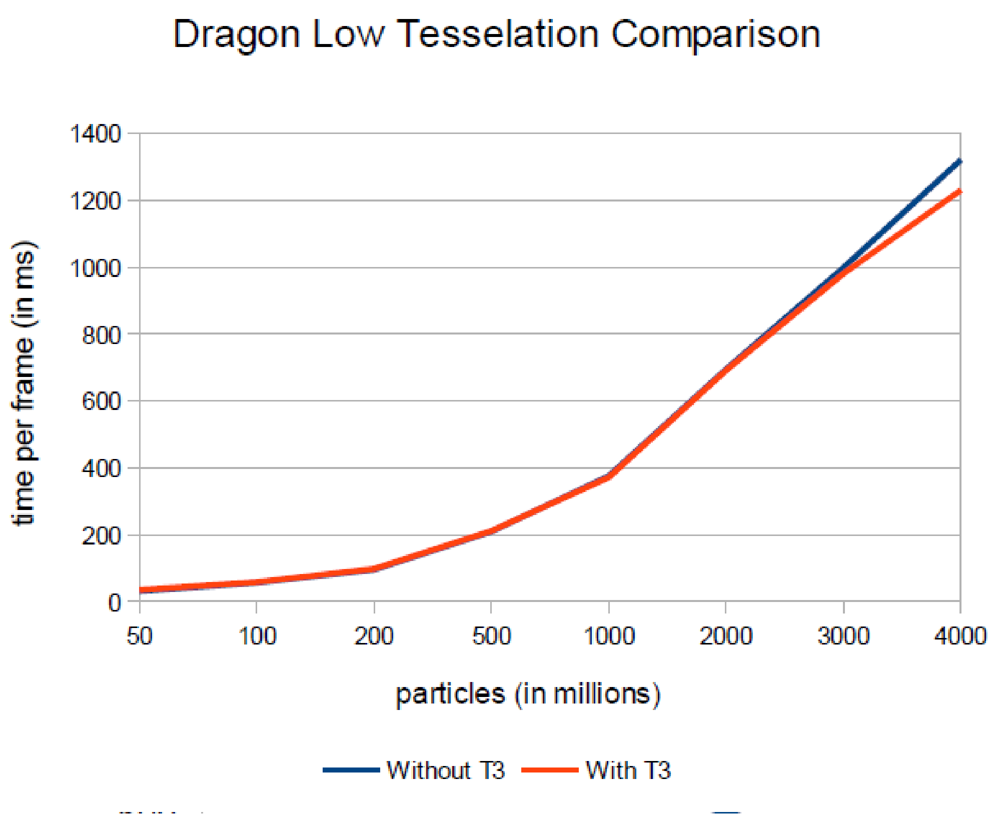


Figure 6.10: Trend line for the dragon dataset with 300k cells. For this grid we observe a much lower penalty for using tier three. Performance already increases in between 500 million and 1 billion cells, much earlier than with the higher resolution dataset.

1000:1). The assignment of cells (= threads) to warps completely neglects inherent cell sizes. This, on average, leads to quite unfavorable warp constellations. Furthermore, the occupancy is extremely high (1.22 million threads) to begin with. As a consequence of these observations, huge quantities of idle threads block resources for threads actually in need of resources. If we now fight the divergence using Softshell, the following happens. Threads, which have finished their work, are sorted out from the pool of all threads. Contrary, threads ready for executing further iterations are then grouped to form convergent warps again. Due to the extreme initial occupancy, the overall efficiency barely decreases in consequence of the still very large amount of remaining threads. But in contrast to before, all remaining threads actually exploit the available resources. Recall the average cell size of 1.7 and consider the induced amount of very small cells. Before, the likelihood of pairing small cells with large ones was considerably high, blocking resources throughout the GPU. Now the small cells are removed rather quickly as they finish their work early, and only the large ones remain. Already at a maximum of 200 million particles, which is quite realistic for this dataset, we observe a speed up of 1.8, not to speak of the more unrealistic values at the lower part of the table.

Table 6.5: Render times for different amounts of maximum particles for the ablation dataset with 1.22 million cells. Render times given in milliseconds

Particles in million	Without T3	With T3
50	95 ms	140 ms
100	148 ms	142 ms
200	264 ms	144 ms
500	530 ms	148 ms
1000	927 ms	161 ms
2000	1680 ms	185 ms
3000	2400 ms	212 ms
4000	3180 ms	243 ms

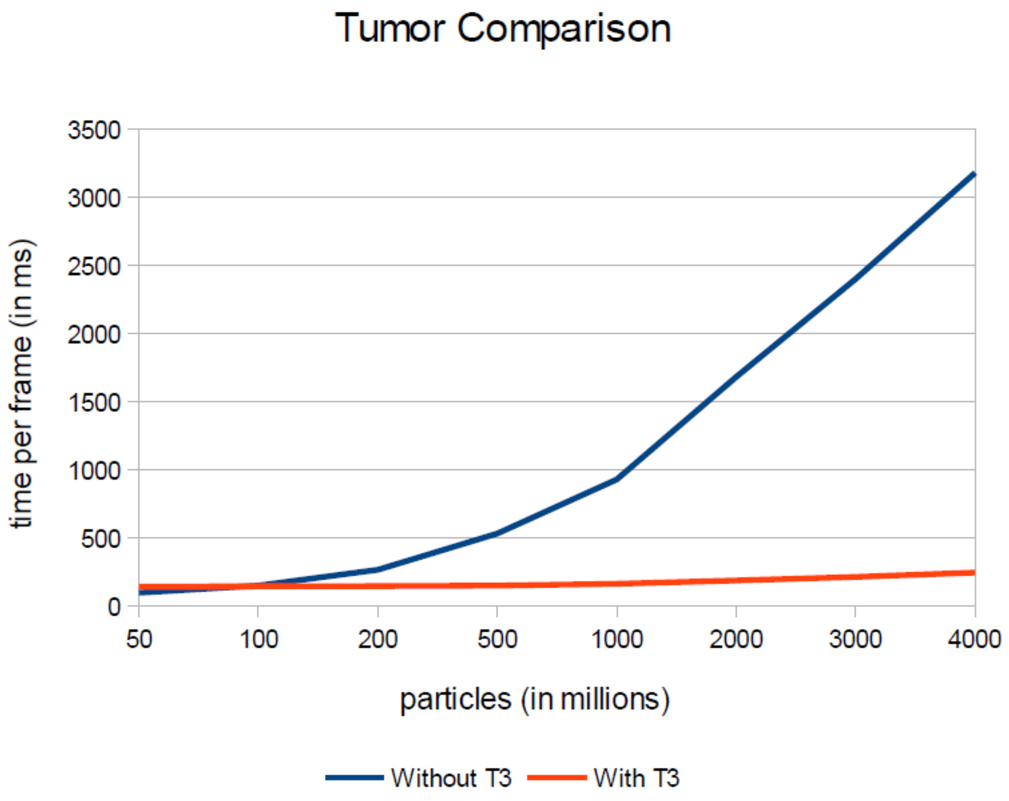


Figure 6.11: Trend line for the tumor ablation simulation dataset with 1.22 million cells. We observe that the CUDA performance decreases vastly in correlation to increasing particle amounts, while Softshell's re-convergence cushions the increasing divergence most effectively.

6.4 Dykstra's Projection Algorithm

We now come to the final algorithm we evaluate in this thesis, Dykstra's algorithm for finding points in intersection of convex sets. We will first analyze the "linear" algorithm in form of a standard CUDA implementation and show up the problems occurring. Next, we will show up possible entry points for Softshell and cater to the possible advantages while also highlighting the problems still remaining. Consecutively, we evaluate the "parallel" version of the algorithm and show up the inherent issues in this version.

6.4.1 Analysis - CUDA

Let us now proceed with an analysis of the algorithm from the viewpoint of a CUDA-implementation. The procedure shown in algorithm 9 needs to be called per point. Hence one point represents a single thread in the CUDA computing grid. We left out a few minor details irrelevant for our considerations due to their vanishing impact on the overall performance. Numerically we were able to achieve a speedup of up to two in the CUDA version for considerably high $dim1$ and $dim2$ ($dim1, dim2 \geq 200$) and low $dim3$ ($dim3 \leq 32$) over the CPU counterpart. Increasing $dim3$ quickly led to a vast drop in performance, leading to the CPU version outperforming the CUDA implementation. We now highlight the issues occurring both from the point of view of available memory as well as computational efficiency which in combination lead to the poor performance in CUDA.

Memory Analysis First we will analyze the memory consumption of the above algorithm. For parallel execution, we need to facilitate working copies of parts of the array, denoted by $p[\,]$, whose values need to be written back into the actual data array $points[\,]$. Additionally, in the linear case, we need the auxiliary array q . Further, during the second projection onto a parabola after the main optimization has finished, we extend the array p by another value, $p[\cdot].t$, and introduce an array f of the same size as p. We will not go into detail of the second projection due to its negligible impact on the whole process. In total, we get a memory consumption of

$$\begin{aligned}
size = {} & 3 * dim1 * dim2 * dim3 && \text{points}\\
& + dim1 * dim2 * dim3 && \text{f (second projection)}\\
& + 3 * dim1 * dim2 * dim3 && \text{working copies of p}\\
& + 2 * dim1 * dim2 * dim3 * (dim3 - 1)/2 && \text{auxiliaries q, for x and y}\\
& + 2 * dim1 * dim2 * dim3 * (dim3 - 1)/2 && \text{constraint indices}\\
= {} & dim1 * dim2 * (7 * dim3 + 4 * dim3 * (dim3 - 1)/2) && (6.1)
\end{aligned}$$

If we assume simple floating point resolution (single precision) of the values, we need to multiply $size$ by 4 to get the actual memory consumption in bytes. We immediately observe that $size$ strongly depends on the feature vector size $dim3$ due to the near quadratic dependence. The per-point memory consumption results from eradicating $dim1$ and $dim2$, which denote the number of input points, from equation 6.1:

$$size_k = 7 * dim3 + 4 * dim3 * (dim3 - 1)/2 \tag{6.2}$$

It is time to also consider the chosen numerical precision given its influence on the required memory. In most cases, floating point precision, i.e. 4 bytes per value, is sufficient and for actual memory consumption we simply multiply 6.2 by 4. Double precision results in an additional multiplicative factor of 2 on top. To get an idea of the impact of the feature vector dimensionality, we offer a listing of memory consumption related to $dim3$ in table 6.6 both for single as well as double precision. We immediately recognize the problems we will face - the memory consumption per point by employing reasonable feature vector dimensionalities induces the inability to completely transfer the whole optimization process for desired input dimensions $dim1$,$dim2$. Even for lower dimensional feature vectors we already need to split the complete process into several iteratively executed pieces. Due to the absence of interdependence between single features, this is no computational problem. We rather lose a lot of occupancy by splitting the input into several calls to fit on contemporary consumer graphics hardware. Further, the split induces linearization by iterating over all separated parts of the problem in different kernel launches. Unfortunately, we cannot circumvent the situation without the availability of significantly larger global memory.
Let us consider an example for visualizing the severity of the described issues. Assume the following input dimensions:

$$\begin{aligned} dim1 &= 500 \\ dim2 &= 500 \\ dim3 &= 128 \end{aligned}$$

Also assume that we have a bit more than one gigabyte of global memory at our disposal. This is a rather realistic value for current consumer GPUs. Further, we use single precision. The memory consumption per point in this scenario is given as

$$size_k = 133632 \text{ bytes}$$

from table 6.6. The total memory we need for all points then is

$$size = dim1 * dim2 * size_k = 31.11 \text{ Gb}$$

Consecutively, we need to split up our problem into 32 single calls with at most 1Gb memory consumption. We also observe that only roughly 7800 threads can be started per run. On modern GPUs, those threads lead to a far too low occupancy to make up for the vast amount of global memory accesses. Recall the discussion of latency hiding in CUDA from section 1.1.2.3. Lowering the number of global memory calls would be possible by facilitating shared memory. But considering the unpredictable structure of the input in combination with the limited amount of available shared memory, this strategy quickly becomes infeasible when examining the required memory per point. Further, notice that $dim3$ is only 128 in this example, while actually even higher dimensionality is demanded.

Table 6.6: Memory consumption per input point

dim3	numValues	mem (float)	mem (double)
2	18	72	144
4	52	208	416
8	168	672	1344
16	592	2368	4736
32	2208	8832	17664
64	8512	34048	68096
128	33408	133632	267264
256	132352	529408	1058816
512	526848	2107392	4214784

Structural Analysis Algorithm 9 shows the most important steps in the procedure. The first loop, responsible for the optimization iterations, inherently diverges due to the inhomogeneous input data. The number of iterations may range from only tens if convergence happens early, up to the several thousand, depending on the stopping criterion (i.e. the maximum number of iterations). The actual number of iterations is completely unpredictable except for the unlikely scenario where one has stored it for the exact same input during a previous run. This strategy would require vast amounts of memory in conjunction with executing the algorithm incredibly often to gather enough data. In the previous paragraph, we have already shown that memory is a serious issue as is. Hence estimation or even precise lookups of the number of iterations is out of question.
The second loop, in charge of iterating over the present constraints, diverges as well. The condition not only depends on the input data, but also on the maximum number of constraints we may encounter. Recalling that $maxConstraints = dim3 * (dim3 - 1)/2$, the loop iterations can be somewhere in between zero and $maxConstraints$ per point. The predictive strategy from above is again infeasible due to the dependence on the actual input. The expected divergence solely in this loop is considerably lower than before, but in return the loop execution occurs once per dykstra iteration.
We have two more loops interlaced into the previous ones, but in this case they are called successively rather than nested. The first one, calculating the sum, as well as the second one, which updates the working copies, both employ the exact same header. Unfortunately, we cannot aggregate them into one pass due to the calculations performed in between. On top of that, the loops clearly cause warp divergence again. The number of iterations in this case has an upper bound of $dim3$, which is, as already stated, preferably assigned to 512. It is very unlikely that all threads in a warp coincidentally handle constraints with the same size. Furthermore, both loops facilitate two global memory reads or writes respectively. Although shared memory would be clearly the level of preference, we already discussed the inability to facilitate it in the previous paragraph.
We now have to deal with three strongly diverging interlaced loops. In fact, we know the number of iterations for the second and third levels after launching algorithm 8. So, in

theory, we could try to rearrange the points to fight the divergence on the second level. But this strategy is not only computationally intense, but also does not fight the divergence on the first level. Rearranging the threads by incorporating both the second and third loop to achieve a less, but not necessarily non-divergent version closely relates to the knapsack problem, which is known to be NP-hard. Although approximative solutions exist, the sheer number of input parameters still makes this strategy rather complex. The required effort clearly does not pay off due to the inability to abolish divergence completely.
Let us again consider a very simple example to visualize the problematic situation. Assume the following parameters:

$$\begin{aligned} dim1 * dim2 &= 8 \\ dim3 &= 4 \\ maxConstraints &= 6 \\ maxIt &= 5 \end{aligned}$$

One of the possible scenarios of execution for these parameters is shown in Figure 6.12. Each of the columns represents one thread working on a distinct set of constraints. The colored bars depict the innermost for-loops. Note that, for the sake of simplicity, we only depict one of the two innermost loops. In fact, the situation is even worse due to duplicating each "bar", including the gaps, which causes even more idle time for some threads. The outlined white bars depict the phases causing threads need to stall, waiting for the remaining ones to finish the current sum/update. Even in this strongly simplified example, the abysmal situation we face with Dykstra's algorithm clearly stands out. The thick horizontal lines mark the periodicity of the outermost optimization iterations. Note that in the second to last iteration this periodicity slightly changes in favor of performance due to collapsing loops. One might think this is a favorable circumstance, but it in fact means that some of the threads in the warp do not work at all anymore. Also note the gaps caused by different amounts of constraints and also the different loop length for the sum/update procedure. Also, the depicted scenario does not even reflect the worst case, but rather the expected behavior. If we now extrapolate to a more realistic setup, e.g. $dim1 * dim2 * dim3 = 500 * 500 * 128$, the inefficiency is out of hand. Hence, without restricting the input data to well-behaving scenarios, we cannot efficiently avoid the divergence caused by the three inherently diverging nested loops.

To get a more concise idea of the severeness of the situation we provide another example with a slightly bigger dimensionality. Figure 6.13 shows the diverging number of constraints for 100 input points, recorded during actual execution of the algorithm. In this test scenario, $dim3$ was set to 16. We see that it is rather unlikely that the maximum number of constraints is actually reached, but the divergence on this level of execution remains extremely problematic as is. Note that due to the larger dimensions we unfortunately cannot provide a visualization similar to the basic case above. In Figure 6.14 we show the iteration count of the innermost loops. We cannot sync the bars to show the idle time in this case due to the sheer size of the graph, but the color coding remains the same. Even though the interleaved idle times are not shown in this graph, even the summed up

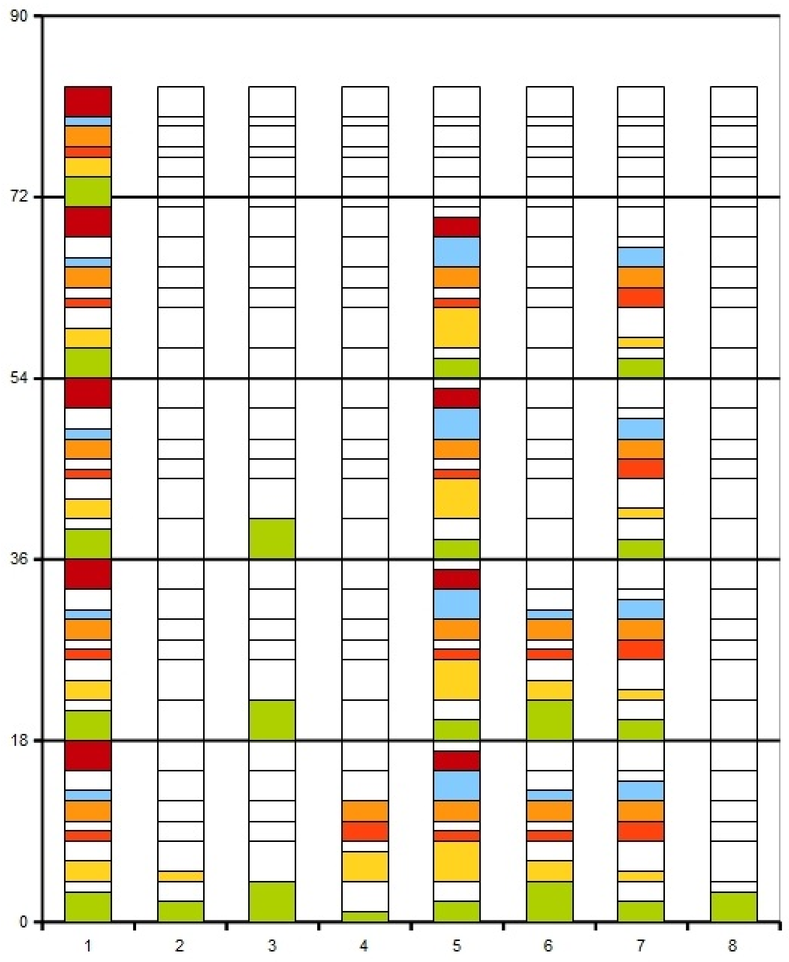


Figure 6.12: Loop divergence in Dykstra optimization of a possible setting for the simple example.

iteration counts give some insight into the reasons behind the poor performance of the parallelization. Keep in mind that the stacked bars only represent one of many dykstra optimization loop passes.

6.4.2 Analysis - Softshell

We now proceed with analyzing possible entry points for Softshell's third tier scheduler to fight the divergence described above. Unfortunately, Softshell's capabilities only concern thread gathering on one of possibly several nested execution levels. Experimentally none of the three versions increased the performance. Rather, the massive overhead caused by fighting rapid divergence on all levels in many cases decreased performance compared to the plain CUDA version considerably. We will now describe the outcome of applying it to each of the levels separately and mutually exclusively.

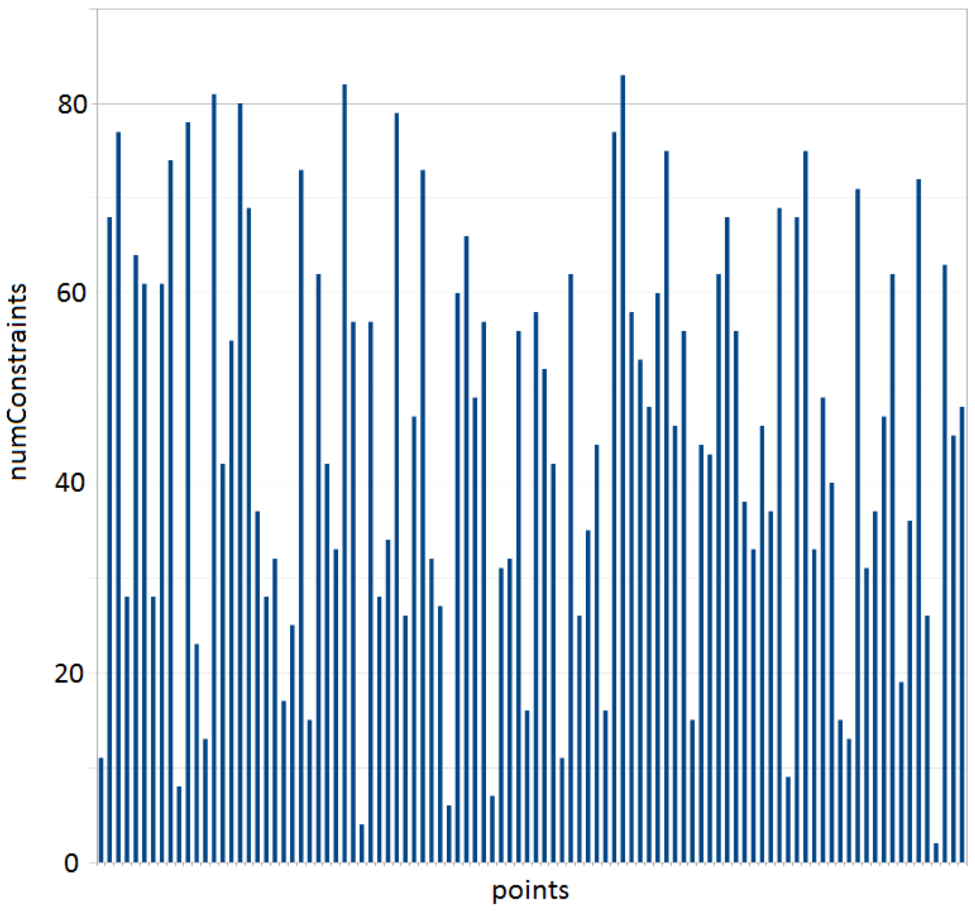


Figure 6.13: Number of constraints for 100 points with $dim3 = 16$.

Dykstra Iterations This is probably the most logical location to facilitate Softshell's divergence anti-measures. Due to the completely unpredictable nature and the high limit on the number of iterations we assumed to benefit most from the third tier when applied to this loop. Although thread aggregation works quite well, the resulting gain in performance is close to non-existent. The reason is at hand. Although we aggregate the threads, we already showed that the overall occupancy of the single SMPs of the GPU is too low right from the start. Removing even more threads, albeit inactive for the rest of the procedure, simply cannot vastly improve the performance in this case. We could only improve performance in this case if the number of threads fitting on the GPU memory wise would be dramatically bigger, mostly resulting from the sheer amount of global memory read and write operations.

Iterations over the constraints The next possible entry point for a scheduled loop is located at the iterations over the constraints. Placing the aggregation at these iterations, though, cannot increase performance too much as well. Softshell, as mentioned, can

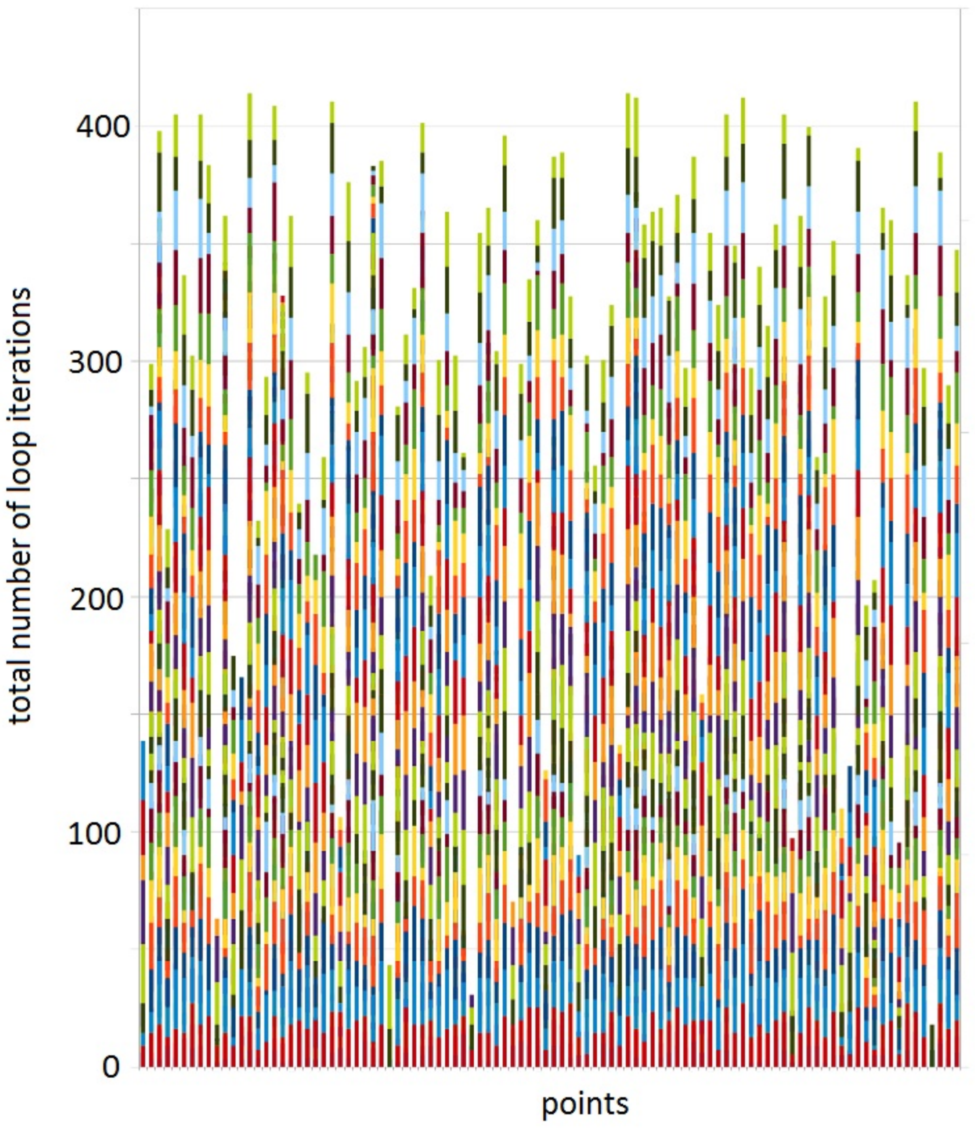


Figure 6.14: Sum and update loop iteration counts for the same 100 points from Figure 6.12 per optimization pass

only deploy its third tier scheduling to one loop. Consecutively, the outer loops remain untouched. This also means that the outer divergence goes by unnoticed. Furthermore, diverging threads on a deeper loop level, in this case the sum and update loops, are not checked for inactivity as well. Thereby, we regroup the threads on the central level without paying respect to the innermost loops and might even deteriorate the situation in the innermost loops. Again, the performance only in some cases increases marginally, but mostly suffers from the overhead introduced by the thread aggregation on software level. Partly due to the low occupancy and partly resulting from the drawback of divergence countermeasures on only one level, this strategy also fails.

Summation & Update At the innermost level we face a somewhat different situation. We have two loops with identical conditions unable to be merged due to their distinct operations. Both of the loops probably do not diverge as strongly as the ones before (recall the desired value of 512 for $dim3$). Let us start off with the first loop, the summation process. If we aggregate threads diverging from this loop condition, we indirectly also influence the behavior of the second loop. Threads still running guarantee to have a higher number of loop passes than those already finished. Due to the update process executing the absolute same iterations in terms of indices, this also slightly improves the performance of the second loop. Contrary, reconverging the threads during the update loop is the inferior choice as the beneficial influence from the co-dependence is lost. Altogether we face the exact same problem as before. We are inevitably forced to concentrate on one loop's divergence behavior while the others vastly diverge. Again, we are unable to drastically increase the performance.

6.4.3 Analysis - Parallel Dykstra's Projection Algorithm

Unfortunately, the aggregation of one update step's outcome is extremely problematic. Ideally we would perform a reduction on the values calculated per constraint in shared memory, which turns out to be absolutely impossible for the following reasons.
If we only want to sum up for the "interesting" values, i.e. between all sets of $i1$ and $i2$ for all constraints, we cannot efficiently perform a reduction as this method relies on all threads working on one (or more) arrays, which is equivalently shaped throughout a whole block. In this case, the indices denote the differing starting points in the array, hence reduction would be inefficient and we would have to restrain ourselves to a rather linear update procedure, quickly becoming inefficient for large values of $dim3$.
Performing real reduction in shared memory hence requires allocation of sufficient memory for all values (also the zeros outside the index bounds). In this case we could perform a normal reduction on multiple arrays. In fact, we would need $numConstraints_k * dim3$ memory locations per block. Already at $dim3 = 16$ only three blocks would fit into the shared memory of one SMP, while everything above 32 greatly exceeds the bounds on contemporary GPUs.
We could of course perform reduction in global memory. But as we already stated, accessing this type of memory is rather slow. Furthermore, if we consider the desired dimensionality 512 for $dim3$ again, this would induce the necessity of allocating up to 511 megabytes of global memory *for a single point* assuming single precision. Although the occupancy increases, the problem becomes too linear in nature. Even if we had vast amounts of memory at our disposal (e.g. huge GPU clusters), we could not increase the performance. Even with 2 gigabytes of global memory on a single graphics card and only 256 dimensions, we can merely start calculation for 31 points at once, which would lead to over 8000 launches for the desired input size of at least $dim1 * dim2 = 500 * 500$.
Besides the problematic situation related to memory constraints, we face another issue with this approach. For correct execution, synchronization of the blocks after each iteration of the optimization algorithm is inevitable. After each of those iterations, a common basis for all threads needs to be established. Otherwise, the outcome is unpredictable and simply incorrect. The synchronization of all threads working for one point's solution across

block borders is not only a tedious task, but further reduces efficiency of the algorithm as a considerable amount of threads needs to halt execution until the remaining ones have finished their work. Although warp divergence remains unaffected due to complete blocks stalling, hardware utilization possibly suffers from the effect. This is not only valid for very high values of $dim3$, but immediately occurs if not all constraints fit into one block. We cannot apply reconvergence strategies to this algorithm, and Softshell's capabilities of dynamic work creation also do not alleviate the issues of the consumed memory effectively. Hence, even the "parallel" version exceeds the capabilities Softshell offers.

Part IV

Conclusion

Chapter 7

Conclusion & Future Work

7.1 Conclusion

We have shown that it is possible to alter existing parallel algorithms in a fashion that they heavily benefit from Softshell's diverse feature set. For both evaluated algorithms we use completely different paradigms in the adaption process, fortifying the analyzed weak points of the algorithms with a combination of smart modifications of the base algorithm with techniques from parallel programming to create a new algorithm perfectly fitting the framework that Softshell provides. Although in the current GPGPU computing model it is only possible to implement Softshell purely in Software on top of CUDA, the numerical results of the new algorithms compared to their original counterpart range from pleasing to absolutely astonishing. Furthermore, we have shown that in special cases, even Softshell is incapable of lifting restrictions for parallel algorithms with generally poor performance. As a consequence, we deem Softshell to be a very powerful framework for boosting the performance of carefully selected parallel algorithms.

7.1.1 Raycasting with Advanced Illumination

The adaptions we made to this algorithm have shown that Softshell's capability to launch new work items from within GPU execution are a powerful tool for increasing performance of certain algorithms. We achieved a speed up of up to 20 for low resolutions. At higher resolutions, our approach involving Softshell still is up to four times faster than the originally proposed method. We also show that even very high resolutions of the involved shadow map are absolutely feasible. In our algorithm, we use a shadow map of the exact same resolution the sampling process of ray casting provides. Using a strongly optimized underlying ray casting algorithm might push the algorithm towards interactive frame rates even at high resolutions, suitable for medical visualization of data sets. To reach this goal, the fragment size could be slightly reduced (e.g. combine 2 fragments of the shadow map) while still achieving satisfying visual results.

7.1.2 Particle Based Volume Rendering

With this algorithm we have shown the power of reconvergence strategies in divergent loops. As expected, low divergence (refer to the tetrahedralized voxel grids) does not vastly increase the performance. Rather, the induced overhead of thread regrouping alleviates, if not exceeds, the performance gained. On the contrary, for datasets this algorithm was designed for (refer to the unstructured grid simulation), we achieve a very drastic increase in performance. This leads to the ability to use even more tesselated datasets with higher particle counts. Alternatively, the saved computing time could be used for improved illumination effects which the basic algorithm does not provide. Screen space ambient occlusion, performed on the supersampled frames, would be an option.

7.1.3 Dykstra's Projection Algorithm

For this case, we have shown that Softshell, despite its capabilities, is not able to solve issues for some algorithms. The inherent inefficiencies of this algorithm outweigh the performance increase by using Softshell drastically. We have deeply analyzed the algorithm for possible adaptions, but the basic structure of both the linear and the parallel version do not allow for gainful alterations. Quite contrary, the parallel version is even expected to perform worse than the linear one due to memory consumption deficiencies. The analysis not only shows up weak points of Softshell, but rather also emphasizes on the drawbacks of the underlying GPGPU computing model. Maybe, if the paradigms of parallel computing shift away from their vector-machine-like model, even algorithms like Dykstra's algorithm will be efficiently parallelized.

7.2 Future Work

Paradigm changes in GPGPU are quite unavoidable given its steady growth. Hence with changing circumstances in this area, frameworks such as Softshell need to adapt, or may even be rendered unnecessary due to immediate adaption in the underlying hardware. Due to the novelty of this area in information technology, the GPGPU community needs to keep up-to-date for creating cutting edge algorithms benefiting from the newly introduced technologies. This is not only valid for direct exploitation of basic GPGPU frameworks, but also for models such as Softshell, setting up on top.
Furthermore, there is a vast bulk of parallel algorithms which can benefit from Softshell and its features. In this thesis, we only scratched the tip of the iceberg in terms of the capabilities Softshell offers. For example, dynamic tesselation in advance of rendering would benefit from launching additional workpackages. View-dependent methods, such as view dependent levels of detail or view dependent refinement of progressive meshes could on one hand benefit from launching additional workpackages, but on the other hand also from dynamic prioritization related to the current view. The same techniques could find application in environment with real-time constraints, such as medical visualization. Dynamic prioritization could be used for reactive, but maybe also predictive strategies in volume rendering applications. Especially the image quality of PBVR could see some attention by employing an illumination approach, such as screen space ambient occlusion

in the subpixel buffer, or a more classic approximation of ambient occlusion on the mesh. All of these graphics related algorithms show potential for benefitting from Softshell, but the concrete results would require an extensive testing procedure along with algorithmic adaptions similar to what we have shown up in this thesis. Also, completely different fields of research deserve further investigation for algorithms with potential for improvement. Optimization problems with a simpler structure than that of Dykstra's Projection Algorithm could vastly benefit from thread reconvergence.

Bibliography

[1] H. H. Bauschke and A. S. Lewis. Dykstra's algorithm with bregman projections: a convergence proof. *Optimization*, vol. 48, pp. 409–427, 1998. Cited on page 27.

[2] U. Behrens and R. Ratering. Adding shadows to a texture-based volume renderer, 1998. Cited on page 26.

[3] E. G. Birgin and M. Raydan. Robust stopping criteria for dykstra's algorithm. *SIAM J. Sci. Comput.*, vol. 26, no. 4, pp. 1405–1414, 2005. doi:10.1137/03060062X. Cited on page 27.

[4] J. F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, pp. 192–198, 1977. doi:10.1145/965141.563893. Cited on pages 19 and 33.

[5] J. P. Boyle and R. L. Dykstra. A method for finding projections onto the intersection of convex sets in Hilbert spaces, 1986. Cited on page 27.

[6] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, vol. 4, no. 1, pp. 25–30, 1965. doi:10.1147/sj.41.0025. Cited on page 37.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pp. 777–786. ACM, New York, NY, USA, 2004. doi:10.1145/1186562.1015800. Cited on pages 4 and 23.

[8] S. P. Callahan, M. Ikits, S. Member, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, p. 2005, 2005. Cited on page 27.

[9] P. Combettes and J.-C. Pesquet. Proximal splitting methods in signal processing. In H. H. Bauschke, R. S. Burachik, P. L. Combettes, V. Elser, D. R. Luke, and H. Wolkowicz (Editors), *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, Springer Optimization and Its Applications, pp. 185–212. Springer New York, 2011. ISBN 978-1-4419-9568-1. doi:10.1007/978-1-4419-9569-8_10. Cited on page 27.

[10] N. Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007. Cited on pages 5 and 13.

[11] B. Csébfalvi and L. Szirmay-kalos. Monte carlo volume rendering. In *In Proc. of IEEE Visualization*, pp. 449–456, 2003. Cited on pages 27 and 45.

[12] C. Dachsbacher and M. Stamminger. Translucent shadow maps. In *Proceedings of the 14th Eurographics workshop on Rendering*, EGRW '03, pp. 197–201. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003. ISBN 3-905673-03-7. Cited on page 26.

[13] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 65–74, 1988. doi:10.1145/378456.378484. Cited on page 18.

[14] R. C. Farias, J. S. B. Mitchell, and C. T. Silva. Zsweep: an efficient and exact projection algorithm for unstructured volume rendering. In *Volviz 2000*, pp. 91–99, 2000. ISBN 1-58113-308-1. doi:10.1145/353888.353905. Http://www.odysci.com/article/1010112988958315. Cited on page 25.

[15] S. Frey, C. Müller, M. Strengert, and T. Ertl. Concurrent ct reconstruction and visual analysis using hybrid multi-resolution raycasting in a cluster environment. In *ISVC (1)*, pp. 357–366, 2009. Cited on page 25.

[16] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 407–420. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-3047-8. doi:10.1109/MICRO.2007.12. Cited on page 23.

[17] C. GmbH. Computerbase.de. `http://www.computerbase.de`, 2013. Cited on pages xix and 6.

[18] J. P. Grossman and W. J. Dally. Point sample rendering. In *In Rendering Techniques '98*, pp. 181–192. Springer, 1998. Cited on page 27.

[19] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. ISBN 1568812663. Cited on page 17.

[20] D. Hartmann, M. Breidt, v. V. Nguyen, F. Stangenberg, S. Höhler, K. Schweizerhof, S. Mattern, G. Blankenhorn, B. Möller, and M. Liebscher. Structural collapse simulation under consideration of uncertainty - fundamental concept and results. *Comput. Struct.*, vol. 86, no. 21-22, pp. 2064–2078, 2008. doi:10.1016/j.compstruc.2008.03.004. Cited on page 20.

[21] H. Hundal and F. Deutsch. Two generalizations of dykstra's cyclic projections algorithm. *Mathematical Programming*, vol. 77, pp. 335–355, 1997. doi:10.1007/BF02614621. Cited on page 27.

[22] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pp. 21–30. Springer-Verlag, London, UK, UK, 1996. ISBN 3-211-82883-4. Cited on page 26.

[23] H. W. Jensen and J. Buhler. A rapid hierarchical rendering technique for translucent materials. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, New York, NY, USA, 2005. doi:10.1145/1198555.1198592. Cited on page 26.

[24] D. Jönsson, P. Ganestam, A. Ynnerman, M. Doggett, and T. Ropinski. Explicit Cache Management for Volume Ray-Casting on Parallel Architectures. In *EG Symposium on Parallel Graphics and Visualization (EGPGV)*, 2012. Cited on page 25.

[25] B. Kainz, M. Steinberger, S. Hauswiesner, R. Khlebnikov, and D. Schmalstieg:. Stylization-based ray prioritization for guaranteed frame rates. In *Proc. Non-photorealistic Animation and Rendering (NPAR 2011)*, 2011. Cited on page 25.

[26] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pp. 165–174. ACM, New York, NY, USA, 1984. ISBN 0-89791-138-5. doi:10.1145/800031.808594. Cited on page 26.

[27] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 150–162, 2003. doi:10.1109/TVCG.2003.1196003. Cited on page 18.

[28] M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, vol. 8, no. 3, pp. 29–37, 1988. doi:10.1109/38.511. Cited on page 25.

[29] T. Lokovic and E. Veach. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pp. 385–392. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 1-58113-208-5. doi:10.1145/344779.344958. Cited on page 26.

[30] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995. doi:10.1109/2945.468400. Cited on page 17.

[31] A. Maximo, R. Marroquim, and R. Farias. Hardware-Assisted Projected Tetrahedra. *Computer Graphics Forum*, vol. 29, Issue 3, pp. 903–912, 2010. Cited on page 27.

[32] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pp. 65–ff. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0-89791-864-9. Cited on page 27.

[33] D. Mumford and J. Shah. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, vol. 42, no. 5, pp. 577–685, 1989. doi:10.1002/cpa.3160420503. Cited on page 27.

[34] Nvidia. *Fermi Compute Architecture Whitepaper*, 2010. Cited on page 10.

[35] OpenGL.org. Opengl wiki. `http://www.opengl.org/wiki/Rendering_Pipeline_Overview`, 2012. Cited on pages xix and 5.

[36] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, vol. 18, no. 6, pp. 311–317, 1975. doi:10.1145/360825.360839. Cited on page 19.

[37] T. Pock, D. Cremers, H. Bischof, and A. Chambolle. An algorithm for minimizing the mumford-shah functional. In *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 1133 –1140, 2009. doi:10.1109/ICCV.2009.5459348. Cited on page 27.

[38] H. Ray, H. Pfister, D. Silver, and T. A. Cook. Ray casting architectures for volume visualization. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, vol. 5, pp. 210–223, 1999. Cited on page 25.

[39] C. Rezk-Salama. Gpu-based monte-carlo volume raycasting. In M. Alexa, S. J. Gortler, and T. Ju (Editors), *Pacific Conference on Computer Graphics and Applications*, pp. 411–414. IEEE Computer Society, 2007. ISBN 978-0-7695-3009-3. Cited on page 25.

[40] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 0387212396. Cited on page 50.

[41] C. Rocchini and P. Cignoni. Generating random points in a tetrahedron. *J. Graph. Tools*, vol. 5, no. 4, pp. 9–12, 2000. Cited on page 48.

[42] N. Sakamoto, J. Nonaka, K. Koyamada, and S. Tanaka. Volume Rendering using tiny Particles. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pp. 734 –737, 2006. doi:10.1109/ISM.2006.157. Cited on pages 27 and 45.

[43] P. Schlegel and R. Pajarola. Layered volume splatting. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC '09, pp. 1–12. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-10519-7. doi:10.1007/978-3-642-10520-3_1. Cited on page 27.

[44] P. Shirley and A. Tuchman. A polygonal Approximation to direct Scalar Volume Rendering. In *Proceedings of the 1990 workshop on Volume visualization*, VVS '90, pp. 63–70. ACM, New York, NY, USA, 1990. ISBN 0-89791-417-1. doi:http://doi.acm.org/10.1145/99307.99322. Cited on page 27.

[45] Stanford University. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`, 2012. Cited on page 68.

[46] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg. Softshell: dynamic scheduling on gpus. *ACM Trans. Graph.*, vol. 31, no. 6, pp. 161:1–161:11, 2012. doi:10.1145/2366145.2366180. Cited on pages 3 and 23.

[47] W. Strang and G. Fix. *An analysis of the finite element method*. Prentice-Hall series in automatic computation. Prentice-Hall, 1973. ISBN 9780130329462. Cited on page 20.

[48] E. Sundén, A. Ynnerman, and T. Ropinski. Image Plane Sweep Volume Illumination. *IEEE TVCG(Vis Proceedings)*, vol. 17, no. 12, pp. 2125–2134, 2011. Cited on pages 21, 26, 33, and 67.

[49] University of Erlangen. The Volume Library. `http://www9.informatik.uni-erlangen.de/External/vollib/`, 2012. Cited on page 68.

[50] P. Voglreiter, M. Steinberger, D. Schmalstieg, and B. Kainz. Volumetric real-time particle-based representation of large unstructured tetrahedral polygon meshes. In *MeshMed*, pp. 159–168, 2012. Cited on pages 21 and 27.

[51] L. A. Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. Ph.D. thesis, Chapel Hill, NC, USA, 1991. UMI Order No. GAX92-08005. Cited on page 27.

[52] Wordpress. 3dgep. `http://3dgep.com/`, 2012. Cited on pages xix, 7, and 12.

[53] R. Yagel and A. E. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, vol. 11, no. 3, pp. 153–167, 1992. Http://www.odysci.com/article/1010112989689822. Cited on page 26.

[54] O. Zienkiewicz and R. Taylor. *Finite Element Method: Volume 1 - The Basis*. Butterworth-Heinemann, Oxford, fifth edn., 2000. Cited on page 20.