Parallel Visualization and Compute Environments for Graphics Clusters

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Magnus Strengert

aus Stuttgart

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. K.-L. Ma
Mitberichter:	Prof. Dr. D. Weiskopf
Tag der mündlichen Prüfung:	18. Juni 2010

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2010

Contents

Abstract

Parallelism evolved into the primal driving force for progressing the performance of nearly all of today's processing platforms, ranging from mobile devices, over personal computers, to the world's most powerful supercomputers. Whether it is simultaneously executing multiple threads on a modern *central processing unit* (CPU), or utilizing the massively parallel processor arrays of recent *graphics processing units* (GPUs); whether it is scaling performance by employing a multiplicity of each of such processor in a single system, or using large cluster environments composed of hundred of thousands of such processing units, parallelism forms the hardware foundation to approach almost all of today's most demanding compute and visualization challenges. However, making highly efficient use of all the theoretically available distributed processing power remains the key challenge to successfully scale the actual performance to solve a given task.

This thesis addresses the challenges of efficiently leveraging all levels of parallelism inherent to graphics cluster systems—an interconnected compound of processing nodes equipped with GPUs—for the purpose of distributed visualization as well as parallel general purpose computing. Various algorithms and techniques have been developed that are well suited for parallel execution and also serve as building blocks for utilizing the next higher levels of parallelism. The introduced single-pass volume ray casting technique allows for near-optimal usage of the parallel processor array of a single GPU while still maintaining a high level of flexibility. GPU-based pyramidal filtering techniques enable a new, highly adaptive volume sampling algorithm, which is well suited for high output resolutions. Both techniques allow to offload visualization workload from the CPU, which in turn opens up the CPU's parallel processor cores to be leveraged for other tasks, such as data compression or image processing. For that purpose, a highly optimized alpha compositing operator has been introduced allowing to utilize both, the vector parallelism and thread parallelism of modern CPUs. Combining those methods provides the basis for a well-balanced, distributed visualization environment, that uses all available processing units inside a single node and across graphics cluster systems. Likewise, those building blocks also allow to address some of major challenges of these higher levels of parallelism, such as data distribution, inter-node communication, as well as workload balancing.

Besides the focus on efficiently utilizing GPU clusters, this thesis also makes important contributions towards bridging the *Programmability Gap*—referring to the lagging support of programming models and developer tools targeting such massively parallel hardware architectures. By means of an existing compute programming language targeted for a single graphics processor only, an extended programming environment has been introduced that demonstrates seamless scaling of compute applications from a single GPU up to large graphics cluster setups, while still providing a single, consistent programming interface to the developer. From the large-scale parallelism of GPU cluster systems back to low-level parallelism exposed by the processor array of a single GPU, new parallel-aware developer tools are presented, which for the first time allow for single-step debugging of all programmable pipeline stages of modern graphics processors.

While designed in the context of GPU clusters, many of the introduced algorithms and tools also apply to other already existing or upcoming processor platforms, such as FPGA-based accelerator cards, many core CPUs, as well as hybrids combining processing cores from today's CPUs and GPUs. In a more general sense, the concepts may be applicable to any architecture that share the common underlying principle of parallelism.

Zusammenfassung

Parallelität hat sich zur Triebfeder für das weitere Wachstum an Rechenleistung heutiger Computersysteme entwickelt, sei es im Bereich mobiler Geräte, bei Arbeitsplatzrechnern oder den weltweit schnellsten Hochleistungsrechnern. Egal, ob beim gleichzeitigen Bearbeiten mehrerer Prozesse auf einem modernen Hauptprozessor (CPU) oder beim Verwenden der hoch parallelen Recheneinheiten moderner Graphikprozessoren (GPU), sei es beim Vervielfachen der Rechenleistung durch die Verwendung mehrerer solcher Prozessoren in einem einzelnen Computer oder durch die Anwendung großer Rechennetzwerke mit mehreren hunderttausend Prozessoren, parallele Datenverarbeitung bildet die Grundlage zur Bearbeitung vieler der anspruchsvollsten Problemstellungen der heutigen Zeit. Die effiziente Verwendung all der theoretisch zur Verfügung stehenden verteilten Rechenleistung ist jedoch die zentrale Herausforderung, um weiterhin erfolgreich die tatsächliche Leistung zum Lösen einer Aufgabenstellung zu erhöhen.

Diese Dissertation befasst sich mit der effizienten Verwendung aller Formen von Parallelität von Graphik-Clustern—einem vernetzten Verbund von Computern, ausgestattet mit Graphikprozessoren—zur verteilten Visualisierung, sowie zur parallelen Datenverarbeitung. Verschiedene Algorithmen und Methoden wurden entwickelt, die speziell auf eine parallele Bearbeitung zugeschnitten sind und auch als Baustein für die Verwendung weiterer Arten von Parallelität dienen. Das vorgestellte Verfahren zur Darstellung von Volumendaten mittels Raycasting ermöglicht die nahezu optimale Verwendung der parallelen Recheneinheiten eines einzelnen Graphikprozessors und ermöglicht trotzdem ein hohes Maß an Flexibilität. GPU-basierte Pyramidenfilter bilden die Basis für ein neues, hoch adaptives Verfahren zur Volumenvisualisierung, das sich besonders für eine hochaufgelöste Bildausgabe eignet. Beide Verfahren erlauben den Hauptprozessor von Visualisierungsaufgaben zu entlasten, was im Gegenzug eröffnet, die parallelen Recheneinheiten der CPU für andere Aufgaben, wie zur Datenkompression oder zur Bildverarbeitung, einzusetzen. Zu diesem Zweck wurde ein hoch optimierter Alpha-Compositing Operator entwickelt, der sowohl die Vektor-Parallelität als auch die Thread-Parallelitäet moderner CPUs ausnutzen kann. Die Kombination dieser Verfahren bildet die Grundlage für eine ausbalancierte, verteilte Visualisierungsumgebung, die alle Recheneinheiten eines einzelnen Computers aber auch eines ganzen Graphik-Clusters ausnutzten kann. Die vorgestellten Bausteine dienen ebenso dazu, einige der wichtigsten Herausforderungen solcher verteilten Systeme zu bearbeiten, wie zum Beispiel die Datenverteilung, die Datenkommunikation, sowie die Lastverteilung.

Neben dem Schwerpunkt der effizienten Verwendung von Graphik-Clustern, umfasst diese Dissertation auch wichtige Beiträge, um die Verwendung und Programmierbarkeit solcher hoch parallelen Rechenarchitekturen zu vereinfachen. Am Beispiel einer vorhandenen Programmiersprache zur Datenverarbeitung auf einem einzelnen Graphikprozessor wurde eine erweiterte Programmierumgebung entwickelt, die es Anwendungen erlaubt, nahtlos von einer einzelnen GPU bis hin zur kompletten Architektur eines Graphik-Cluster zu skalieren, aber gleichzeitig eine einheitliche Programmierschnittstelle für den Entwickler bereitstellt. Von der Paralleliät eines Rechenverbunds zurück zu der Paralleliät eines Graphiksprozessors werden neue Verfahren zur Unterstüztung von GPU-Programmieren vorgestellt, die es zum ersten Mal ermöglichen, Programmfehler in allen Einheiten eines modernen Graphikprozessors gezielt aufzufinden.

Obwohl alle hier vorgestellten Verfahren im Kontext von Graphik-Clustern entwickelt wurden, können viele der vorgestellten Algorithmen und Verfahren auch auf andere bereits existierende oder kommende Rechenarchitekturen übertragen werden. Dazu gehöhren zum Beispiel FPGA-basierte Beschleunigerkarten, CPUs mit sehr vielen Rechenkernen, hybride Prozessoren, die Recheneinheiten von heutigen CPUs und GPUs kombinieren, und ganz allgemein, jedes System, das auf der gleichen Grundlage von Paralleliät aufgebaut ist.

Chapter 1
Introduction

Today the world's fastest supercomputers utilize hundreds of thousands processing units to achieve unpredicted compute performance—even clearly exceeding the magic mark of 1 petaFLOP (a quadrillion floating point operations per second). Modern graphics processors combine hundreds of stream processing units on a single chip to parallelize graphics tasks as well as general purpose computations. And the typical central processing units nowadays expose four physical processing cores, with eight and more to come. Furthermore, additional scaling may easily be introduced by stacking up multiple CPUs and/or GPUs into a single system. Obviously, parallelism evolved into an ubiquitous key characteristic of any modern processing platform, ranging from the smallest mobile devices, over everyone's personal computer, to professional workstations, up to large-scale high performance cluster systems.

The enormous pace at with parallelism was adopted—especially outside the originating field of high performance computing—becomes obvious when comparing the aforementioned systems of today with the ones available at the time this thesis was started: In 2004, the world's most powerful computer was composed out of five thousand processors (with an overall performance of 0.035 petaFLOPs), graphics units exposed at most 16 highly specialized shader units primarily suitable and solely designed for parallelizing graphics tasks only, and central processing units just introduced the concept of virtual cores for optimizing to some extent the execution of two concurrent threads on a single physical core. Scaling workload across two GPUs in tandem was still out to come.

Alongside this rapid technological change in hardware comes the inevitable need of efficiently utilizing all such increasingly parallel processing resources in software. Even more so, as parallelism nowadays emerges as the number one driving force for any significant improvement in overall compute performance—replacing the previously prevalent advances in raw clock rates that were of direct benefit to any process, sequential or parallel alike. Consequently, the general challenges of parallel programming become an inherent part of any software development for modern compute platforms, whether targeting a complete cluster environment or a single processor.

It is the focus of this thesis to address these challenges in the specific context of GPU cluster architectures—a network interconnected compound of GPU-enabled processing nodes that exposes all the diverse levels of parallelism introduced above: From multi-core CPUs, over the massively parallel processor arrays of modern GPUs, to multiple (hybrid) processors per node, up to the complete interlinked cluster environment. Motivated by the constant need to provide means to handle, analyse, and explore the ever increasing data sizes—often the immanent result of extensive utilization of parallelism during computational simulation themselves—the very specific architecture of “graphics clusters” is primarily targeted at large-scale scientific visualization; tackling problem statements that by far exceed the capabilities of a single graphics workstation. With the upstream trend of general purpose computations on graphics processors, GPU cluster systems additionally start to draw more and more attention in the core field of high performance computing as well—a trend recently manifested by the first GPU-enabled supercomputer entering the top 500 of the world’s most powerful cluster installations.

For both fields of application, scientific visualization as well as computation, this work tackles the challenges of GPU cluster computing: On the one hand, focusing on algorithmic aspects and the design of suitable methods to scale visualization techniques and GPU-based general purpose computations into distributed system architectures. On the other hand, targeting the difficulties of parallel programming, by specifically designed cluster-aware programming languages and custom tailored developer tools for the massively parallel execution environment of modern graphics hardware. Combined with the shared goal to move towards efficient parallel visualization and compute environments for graphics clusters.

1.1 Outline

Chapter 2 introduces the basic principles of interactive visualization in the context of modern graphics processing units. Discussing the hardware characteristics of current GPU generations and the emanating highly parallel execution environments in detail provides the necessary fundamentals for designing efficient algorithms for this class of processors in the following two chapters.

Using the example of volume visualization, Chapter 3 follows the approach of deriving scientific visualization techniques directly from the underlying physical principles—in this context from the interaction of light with participating media. With single-pass volume ray casting an efficient mapping of such techniques onto the parallel processing arrays of graphics hardware is introduced; enabling interactivity while still retaining the flexibility to be adapted to various different types and variants of volume rendering.

Chapter 4 continues with the design of highly scalable algorithms for graphics hardware with the focus set to image processing. Originally motivated by the necessity to come up with highly efficient image operators to enable interactivity on high resolution output devices, such as large display walls, GPU-based pyramid filtering techniques form the basis to high-throughput image operators for zooming, burring, or scattered data interpolation. Based upon these GPU-aware operators the examples of efficient depth-of-field rendering as well as highly adaptive volume sampling demonstrate accelerating more complex imaging and visualization techniques via pyramid filters.

Building upon the work of the previous two chapters, Chapter 5 extends parallelism from the processor array of a single graphics unit to complete cluster environments. Key challenges in scaling performance as well as manageable data sizes into distributed cluster architectures are addressed with the overall goal on creating an efficient, well-balanced system environment for interactive distributed visualization on graphics clusters.

Efficient scaling of workload across GPU clusters continues to set the focus for Chapter 6, while shifting the context to distributing general purpose computations for high performance computing. Centered around a custom programming environment—specifically designed with the various levels of parallelism of graphics clusters in mind—the discussion addresses the topics of programmability, usability, and efficiency of such a generic language-based approach to cluster computing. Exemplary fields of application of this approach span the areas of numerical linear algebra, global illumination, as well as data reconstruction from tomographic imaging.

Targeted at the challenges of parallel programming for graphics processing units, Chapter 7 introduces custom tailored developing tools for shader programming. By maintaining the concepts of parallelism exposed by graphics processors throughout the process of debugging and profiling, important key characteristics of shader execution can be made available to the developer and allow for improving the overall development cycle of GPU-based graphics applications. While being specifically designed to support shader development, the derived general concepts for parallel-aware debugging/profiling tools are also applicable to other use cases of graphics processors, but may also apply to any other processing platforms sharing the underlying principle of parallelism.

The last chapter summarizes the presented algorithms and techniques and relates the thesis' contributions to most recent developments as well as possibly upcoming trends in the field of distributed visualization and parallel general purpose computation on graphics cluster systems.

Chapter 2

Interactive Visualization

Visualization is the procedure of providing inside to complex, abstract data by the means of a visual, graphical representation. Its use cases are widespread, ranging from the pure communication of complex information, over revealing correlations between various data sources or repetitive experiments, to the exploration of not (yet) understood data. Shared among all these applications is the common goal of providing meaninful, comprehensible, fast, but yet precise and accurate access to vast amounts of raw data with the purpose of reducing overall time-to-discovery.

2.1 Visualization Pipeline

A conceptual formalism to the process of visualization was first introduced by Haber and McNabb [HM90] by defining a linear sequence of fundamental visualization tasks, nowadays commonly referred to as the *Visualization Pipeline.* According to their model, the transformation of the raw input originating from *Data Acquisition* to the final visualization output, typically in form of images or image sequences—covering both, video animations as well as interactive applications—can be split into the three major visualization stages of *Filtering*, *Mapping*, and *Rendering* (see also Figure 2.1).

Filtering Originally referred to as *Data Enrichment/Enhancement*, the first stage deals with pre-processing and refining the raw input data to come up with a well-defined source data stream, i.e. the actual *Visualization Data.* In the strict sense of enrichment or enhancement, common operations include data interpolation/approximation as well as augmenting the input data by further derivable quantities—for example, vorticity or shear might offer an additional valuable perspective to the underlying raw vector data. In excess of those operations, the more general term of *Filtering* further emphasis the pre-processing character of this pipeline stage. On the one hand, this refers to filtering in terms of data convolution, like

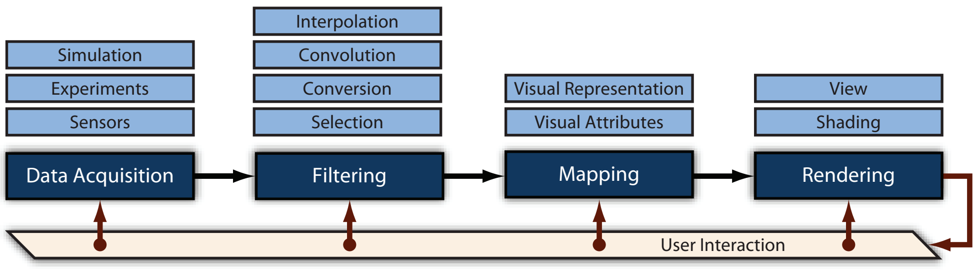


Figure 2.1: Overview of the Visualization Pipeline.

de-noising or data smoothing. On the other hand, this also covers filtering in terms of selection, i.e. the reduction of the input in respect to the field of interest, such as a planar cut through a volumetric field, the focus to a very specific data range, or the limitation of three-dimensional data to a given surface.

Mapping With *Visualization Data* being still plain numerical information, the stage of *Mapping* represents the actual transformation from abstract data into a visual representation. In its most direct form, such a transformation might already be implicitly imposed by the origin of the input data, like for example geo-spatial satellite data naturally map to a polygonal hight field, or might be directly derived from common conceptual associations, e.g. using arrow glyphs for vector data. However, other types of data might not necessarily expose such direct mapping techniques and even if so, such mappings might not always offer a truly effective visualization result.

Besides mapping to geometric attributes, various other attributes might serve as target for the mapping process of which color is one of the most prevalent mapping destinations. Thermographic cameras, for example, typically map infrared radiation to color information—thereby also utilizing the universal association of colors and temperature.

In particular for multi-field visualization techniques—depicting various input fields concurrently—mapping the data fields to distinct output attributes is key to enable perceivability as well as visual data correlation.

Rendering In the final stage of the *Visualization Pipeline* the geometric representation and visual attributes are transformed to the final visualization result, i.e. the pixel data of the output image. Discussion of this *Rendering* process is continued in more detail in Section 2.2.1.

Interaction Once all parameters of the various pipeline stages are fully configured, the visualization system is capable of generating a single static output.

However, barely a single visualization challenge allows to pinpoint all parameters optimally once in advance nor might a single, even optimal image be sufficient to fully capture the nature of the complete input data. Even more though, in case of dealing with new, unfamiliar visualization input being able to iteratively adapt the visualization settings is paramount for exploring such data. Hence, visualization is commonly understood as a feed back system at which the user interacts with any of the *Visualization Pipeline*'s stages based on the currently received output. Apart from changing "visualization" parameters only, in some scenarios feedback can even go back to the actual data acquisition (*Steering*). By doing so, the visualization process no longer acts as pure post-processing stage to the data acquisition, but rather serves as instrument to control data generation, such as simulations or sensor imaging, in a profound way.

Interactivity The effectiveness and usability of user interaction is tightly coupled with interactivity—the ability to provide timely feedback to user requests. In this sense, the term "timely" defines the visualization system to be at least fast enough to enable an explorative work flow. Depending on the actual visualization task, the intended usage, as well as the target stage of user interaction in the visualization pipeline absolute required feedback times to achieve interactivity may vary highly. A fact, that is also reflected by the broad usage of the term *Interactive Visualization* in literature—labeling systems that require multiple seconds per image up to real-time visualization with multiple images per second.

To be referred to as being interactive, a visualization system must at least meet the aforementioned criteria for the *Rendering*; thus, enabling interactive navigation through the input data. Nowadays, however, *Interactive Visualization* is commonly expected to provide interactive feedback up to the *Filtering* stage or the pipeline.

In this work, two major approaches are employed to achieve such a high level of interactivity: Firstly, the application of specialized graphic processing units (*GPU*s) to execute not only the *Rendering* process, but also the *Mapping*, the *Filtering*, and even suitable compute intense portions of simulation codes (*Data Acquisition*). And secondly, the utilization of GPU cluster systems for targeting interactivity for very large visualization challenges. While more obvious for the latter approach, parallelism is the common fundamental principle to both techniques, as detailed for graphics processing units in the following section.

2.2 Graphics Processing Units

Originally introduced as very simplistic graphical output buffer paired with a very basic set of 2D manipulator functions, graphic processing units evolved into highly powerful, specialized processors primarily aimed at 3D rendering, i.e. converting polygonal geometrical data into raster graphics. Predominantly driven by gaming, the rendering capabilities do, however, apply to various other fields of application and in particular to interactive visualization.

The underlying concept of GPU rendering—shared across all those use cases—is given by the model of a *Rendering Pipeline* that is discussed in more detail in the next section, followed by a brief introduction to the actual highly parallel implementation of such a pipeline in hardware. Finally, the chapter closes by taking an entirely different view onto that massively parallel hardware design with the focus on executing non-graphics algorithms on graphics processors.

2.2.1 Rendering Pipeline

From a developers point of view the process of converting the command stream of render calls issued on the *host*, executed on the CPU, to the final pixel data written to the frame buffer of the GPU follows a well-defined, sequential series of processing stages, commonly referred to as the *rendering pipeline.* This section briefly outlines such a pipeline on the basis of the *OpenGL rendering pipeline* for DX10 class graphics hardware (see Figure 2.2). Although the pipeline does change over time to adapt/extend to advanced flexibility and increased feature sets, the core of this pipeline is shared among all pipeline models for polygonal rendering—software, fixed-function hardware, as well as programmable hardware alike.

Host Conceptually, the process of rendering is triggered by issuing a draw call on the host application, thereby declaring a set of vertices and their topology, e.g. points, lines, triangles, quads, polygons, or composed strips of some of these primitive types. The vertex data covers all specified vertex attributes defining the shape and appearance of the scene/object to render; this may include attributes like position, color, normals, or texture coordinates, but also any other application-specific, generic attributes attached to each vertex.

Vertex Processing After the data specification on the application side, the first rendering stage to process the input data is *vertex processing.* As indicated by the name, this stage works on a per vertex basis, i.e. each vertex is processed independently and without interrelations to any other vertex from the set or primitive. As the *host*-specification of the vertex attributes may provide the data immediately with the draw call or may also index previously written memory, each vertex gets its attributes fetched via the *input assembly* first. The actual processing of the vertices continues by executing a *vertex shader* per input vertex or alternatively following the (often emulated) fixed-function vertex processing stage. The output

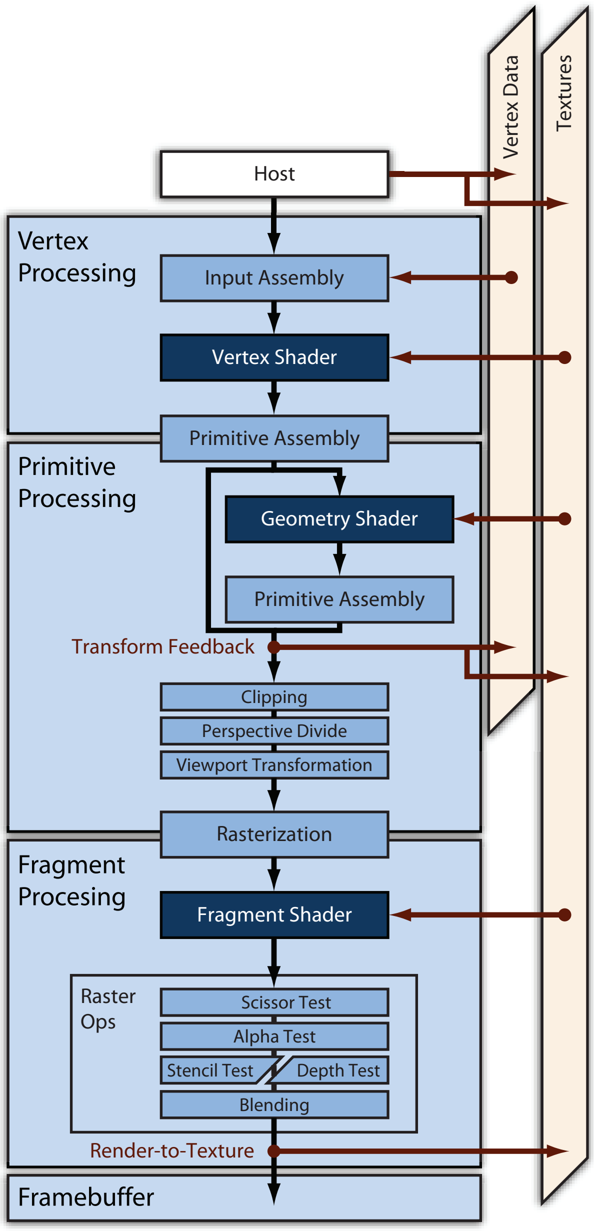


Figure 2.2: Schematic overview of the rendering pipeline effective for DX10 class graphic processing units. Alongside the actual pipeline (blue) the interrelation to the most important render data is depicted (red).

in both cases are “transformed vertices” that all reside in a common coordinate system (*clip coordinates*), i.e. unnormalized homogeneous coordinates of already projected vertices. Generation of this output may be performed fully application-specific via the programmable vertex shader, however, this process often follows the terminology of the fixed-function vertex stage: Assuming the input vertices to be given in a local coordinate system of the corresponding object (*object coordinates*), the overall transformation can be decomposed into an affine transformation of the object-specific coordinates into a common, global coordinate system (*world coordinates*) and a projection accounting for the camera’s characteristics to capture the three-dimensional space on a two-dimensional screen. Apart from the vertex positions, any other attached attribute may also be transformed or even newly generated on a per-vertex basis.

After the vertices are transformed, render primitives are constructed according to the user-specified topology in the *primitive assembly*, which acts as bridge between the first and the second processing stage. Compound input-primitive types are decomposed into their basic rendering primitives as in most cases only points, lines, and triangles are supported further down the pipeline.

Primitive Processing Operating on the input stream of the issued primitives of a draw call, the fixed-function section of *primitive processing* is focused on filtering the input set of primitives to only those that possibly contribute to the final image; that is, primitives residing inside the field of view (*frustum culling*) and (optionally) facing the camera (*back face culling*). In case a primitive intersects with view frustum, the input is cut and limited to the segment lying inside the frustum (*clipping*)—possibly requiring re-triangulation for triangle primitives. Subsequent to this process, the homogeneous coordinates are normalized by *perspective division* (*device coordinates*) and brought into relation of the actual rendering view port (*window coordinates*) in preparation of the following rasterization process.

Introduced with the DX10 class pipeline, *primitive processing* offers an optional programmable element prior to the aforementioned fixed functionality: Operating directly on the output of the primitive assembly, the *geometry shader* is executed once per primitive. Each instance has access to all “transformed vertex attributes” of its assigned primitive. Based on this input, a geometry shader may output a newly generated list of primitives of a single pre-defined type, that is not necessarily identical to the input type. Per shader instance an arbitrary 1-to-n relationship between its input and the list of output primitives can be created. If this shader stage is enabled, the generated primitives again pass through *primitive assembly* to form a new set of basic render primitives.

Also newly introduced with the DX10 class pipeline is the optional output of “transformed vertex data” prior to clipping. The data is streamed into a buffer object that can be reused as input vertex data or texture data. The former mechanism effectively introduces the ability to loop over the processing of vertex/geometry data, e.g. enabling iterative mesh refinement techniques like subdivision surface methods [CC78, DS78].

Acting as bridge to the final processing stage, the remaining set of primitives are passed to the *rasterization* process, which is responsible to convert each primitive into a discrete set of elements according to the raster of the target image. In this process, the vertex attributes are interpolated in order to equip each output element with a distinct set of all attributes according to the elements' relation to the vertices of the primitive. Henceforth, each of these elements is explicitly bound to a fixed coordinate in the final image (*image space*). In order to distinguish between an image's pixel elements—defined by a position and a color only—and the elements generated by the rasterization process, the latter are referred to as *fragments*—a candidate pixel with all interpolated input attributes and further generated attributes, e.g. depth information, attached.

Fragment Processing The third and final processing stage operates on a per fragment basis and is split again into a programmable section and a subsequent configurable fixed-function block.

The former is represented by the *fragment shader* that is responsible to derive the final color and final depth of the fragment based on the interpolated input attributes. Available for all three shader stages, but of particular importance to fragment processing, is the ability to access external pixel/fragment data from memory via textures. Analog to the ability to loop over vertex processing, such a mechanism enables iterating over fragment processing by redirecting the final output of fragment processing to memory—referred to as *render-to-texture* mechanism—and reuse this data in a following render process.

The fixed-function segment of the fragment pipeline is executed after the *fragment shader* and consists of a series of tests that determine whether the processed candidate fragment is written as pixel to the output buffer, i.e. the frame buffer for display or an off-screen memory target. Those tests include conformity to a pre-defined rectangular area (*Scissor Test*) and comparisons to a given transparency bound (*Alpha Test*), fragment counter (*Stencil Test*), and depth value (*Depth Test*). In case a fragment passes all these tests, the final color is combined with the already existing color information at the fragment's position in image space. This might simply be a replace operation, but may also be a linear combination of the two colors.

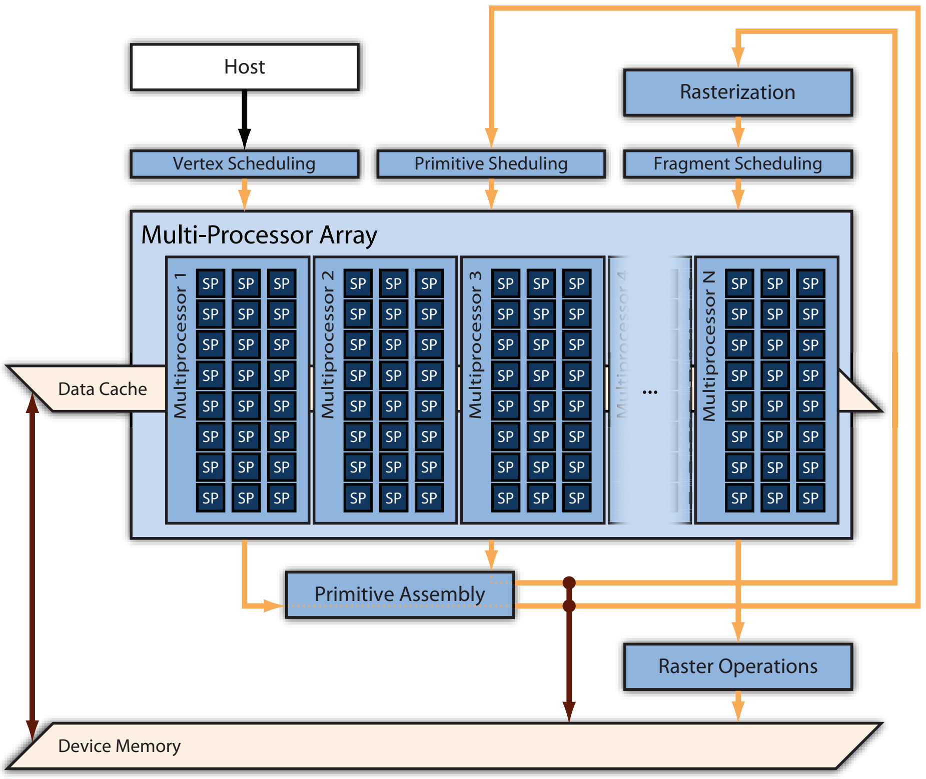


Figure 2.3: High-level schematic overview of a unified shader hardware architecture for the DX10 class rendering pipeline. The orange data flow resembles the full three-staged rendering pipeline. The layout is adapted from NVIDIA's G80 block diagram [NVI06a].

2.2.2 Hardware Design

The actual implementation of the rendering pipeline in graphic hardware significantly changed over time. At first, only fractions of the pipeline, i.e. the fragment stage only, were actually cast to dedicated silicon, while the remaining sections were executed in software—alongside the actual host application*. Initial designs for a complete hardware implementation of the rendering pipeline—fixed-function and later on programmable alike—exhibit distinct hardware units for the different stages of the pipeline, with each unit being highly specialized for the given task. Texturing, for example, was introduced for fragment processing only and extended to other stages significantly later on. With increased flexibility of each unit, however, the exposed feature set of these specialized processors ultimately converged.

*Even nowadays, such hybrid approaches are commonly found in areas with specialized requirements to die size and/or power consumption, as for example typical for embedded graphics units for mobile devices or entertainment electronics.

Driven by the transition to the DirectX 10 pipeline, graphic hardware designs of the major vendors hence shifted to utilize the same units for vertex, geometry, and fragment processing; a hardware layout typically referred to as *unified shader architecture*.

Common to nearly all generations of graphics processing units is the underlying concept of parallelism. As already mentioned during the discussion of the rendering pipeline (see Section 2.2.1) the elements of each stage are processed conceptually independent from each other—making the processing of vertices/primitives/fragments on the corresponding pipeline stage an embarrassingly parallel task. As a consequence, graphics hardware utilized multiple, parallel processing units ever since (see Section 5.1 for a more detailed discussion of applied parallelism in the context of graphic hardware).

To conclude this very brief introduction to graphics hardware, Figure 2.3 depicts a schematic layout diagram of a unified shader architecture for the Dx10 class rendering pipeline, typical to modern graphics processing units at the time of writing. The center of such units is comprised of a highly parallel processor array composed of a multitude of *stream processors* (*SP*), each capable of processing either vertices, primitives, or fragments. The rendering pipeline is mapped onto this architecture by a “virtual” pipeline that loops processing over the central processing array multiple times—once for each shader stage. All other blocks of the rendering pipeline—typically not matching the functionality, execution model, or memory access patterns of the unified stream processors—are centered around the processor array as distinct, highly specialized hardware units. The execution order is effectively constructed by directing the various graphic elements between the processor array and the specialized function units, according to the rendering pipeline.

2.2.3 The GPGPU Perspective

Initially triggered by mainly academic research interest in porting non-graphic applications to suit the rigid pipeline model actually designed for rendering, a complete new field of application for graphic processors emerged—nowadays commonly referred to as *General-Purpose computation on Graphics Processing Units* (*GPGPU*). From the very start the motivation and the goal was to utilize the broad parallelism offered by graphic processors to speed up processing—thereby exceeding the classical task of polygonal rendering. While initial efforts were forced to remodel general purpose algorithms to the underlying paradigm of graphical primitives, i.e. vertices, primitives, and fragments, the GPGPU perspective onto graphics hardware significantly evolved up to the present.

Based on the unified shader architecture discussed in the previous section, the GPU—more specifically, its central processor array—is understood as a general-purpose many-core processor capable of executing any parallel algorithm suitable to the underlying execution model. By this level of abstraction—and by conceptually hiding all graphics-centric fixed-functionality around the processor array—the programming model for GPGPU computing is fully separated from graphic processing. In fact, modern GPUs even feature specialized compute capabilities, firstly or solely introduced to the general purpose usage of the graphic processor. Such features, for example, include full double precision arithmetics, atomic instructions, or shared memory across groups of stream processors.

Along with these hardware changes (or to some extent also considered as the driving force) specialized programming paradigms and programming interfaces came up that support a more general application development model for graphics processors. A more detailed introduction to such programming languages, specifically focused on GPGPU programming, is given in Section 6.1.

In summary, the GPGPU's perspective on graphics hardware is a central, highly parallel processor core with specific hardware modules for graphic *and* compute purposes around. The former guarantees efficient polygonal rendering, while the latter enables a far more broad spectrum of algorithms applied to graphics processors. Among others, the diverse areas of GPGPU application nowadays span weather forecasting, molecular dynamics, medical imaging, photo-realistic rendering [†], video processing, database operations, computational finance, or cryptography.

Further discussion of GPGPU methods and the involved parallelism is given in Chapter 6.2 in the context of parallel GPGPU development environments for GPU cluster systems.

[†]Considered to be a GPGPU application—even though part of the field of graphics processing—as the fundamental concepts are completely different to the polygonal rendering paradigm of the rendering pipeline.

Chapter 3

Direct Volume Visualization

Used as a prime example for the process of deriving and implementing an interactive visualization pipeline for a given visualization challenge, volume rendering—the process of displaying three-dimensional scalar input data—is introduced and discussed in detail in this chapter. Thereby, this thesis focuses on *direct volume rendering* techniques that interpret the input data to represent a volumetric, gas like media that interacts with the light passing through it. Consequently the considered methods require the transport of light to be evaluated along each ray passing through the given volume. The contrary class of *indirect volume rendering* methods—using polygonal surface representations of feature criteria defined on the volumetric data—are not further covered.

Starting from the actual physical properties of light transport through volumetric participating media, two different optical models suitable for interactive volume visualization are derived in Section 3.1. Based on these physical descriptions of the visualization process, Section 3.2 details on the building blocks necessary for constructing a visualization pipeline for interactive direct volume rendering targeted for GPU execution. Finally, the last section (Section 3.3) details on an actual implementation of volume visualization that well suits the current architecture of graphics processors and enables interactive rendering for both previously introduced optical models.

3.1 Radiative Transfer in Participating Media

Spectral radiance is commonly used as the basic parameter for light transport in computer graphics as it holds the property of being constant along a line in space in total vacuum. In other words, for such an optimal optical system the radiance is completely independent from the distance between the emitter and detector. And it suffices to survey light interaction only at boundaries or when passing through participating media. In detail, the *spectral radiance* L_ν for wavelength ν along a ray denotes the *radiant energy* Q per unit time and a single frequency with respect to a solid angle and projected source area. Thus, it is defined as

$$L_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) = \frac{dQ}{dt d\nu d\Omega \cos\theta dA}$$

with $\boldsymbol{r}(\lambda)$ being the location in space parameterized by the distance λ from the ray's origin and $\boldsymbol{d}$ being the ray's direction for the radiance transport. The term $\cos\theta$ accounts for the projection of incoming radiance with respect to the ray's direction of light transport.

In contrast to the aforementioned optimal optical system, in case light transport passes through participating media, like clouds, steam, or hot gas, the spectral radiance is influenced by a multitude of natural phenomena caused by the interaction of light waves with the particles of the media. In the following solely the effects of absorption, spontaneous emission, and elastic scattering are considered. Accordingly, further optical phenomena, e.g. fluorescence, stimulated emission, inelastic scattering, or polarization of light, are not taken into account for the sake of simplicity.

Extinction The considered effects causing a loss of radiance during light transportation are two-fold: First, absorption accounts for the loss of radiant energy that is converted to a different form of energy, e.g. heat or electromagnetic energy. Second, out-scattering covers the amount of radiance that is scattered in other directions than the considered radiance transport along the ray. For both effects the amount of diminution in radiance along a ray segment is linear dependent to the source radiance. In case of absorption this correlation can be written as

$$dL_\nu^\kappa(\boldsymbol{r}(\lambda), \boldsymbol{d}) = -\kappa_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) L_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) d\lambda \tag{3.1}$$

with κ being the *absorption coefficient*. With σ being the *scatter coefficient* of the participating media the radiance loss due to out-scattering of light is given as

$$dL_\nu^{os}(\boldsymbol{r}(\lambda), \boldsymbol{d}) = -\sigma_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) L_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) d\lambda. \tag{3.2}$$

Closely related to the absorption of radiance is the notion of the *optical depth* τ, which is a description for the transmittance of a medium. The *optical depth* is a qualitative measurement of the radiance loss during light transport through a medium due to absorption. Thus, it can be defined as the negative ratio of the absorbed radiance along a path with respect to the radiation at the source. In case no energy gets absorbed by the medium, i.e. $dL^\kappa = 0$, the *optical depth* is defined as $\tau = 0$; stating the medium is transparent. In contrast, if a medium absorbs all incoming source radiance, i.e. $dL^\kappa = -L$, the resulting *optical depth* is

$\tau = 1$; stating the medium is opaque. By rewriting equation (3.1) and integrating along a ray segment from λ_1 to λ_2 the *optical depth* τ can be written as

$$\tau_\nu(\lambda_1, \lambda_2) = -\int_{\lambda_1}^{\lambda_2} \frac{dL_\nu^\kappa(\boldsymbol{r}(\lambda'), \boldsymbol{d})}{L_\nu(\boldsymbol{r}(\lambda'), \boldsymbol{d})} d\lambda' = \int_{\lambda_1}^{\lambda_2} \kappa_\nu(\boldsymbol{r}(\lambda'), \boldsymbol{d}) d\lambda'.$$

The optical *transparency* T of a participating media with respect to the *optical depth* is then defined as

$$T_\nu(\lambda_1, \lambda_2) = e^{-\tau_\nu(\lambda_1, \lambda_2)} = e^{-\int_{\lambda_1}^{\lambda_2} \kappa_\nu(\boldsymbol{r}(\lambda'), \boldsymbol{d}) d\lambda'} \tag{3.3}$$

with $T = 0$ denoting a completely opaque medium and $T = 1$ specifying a completely transparent medium.

Emission In contrast to the extinction of radiance, emission covers for effects that increase radiance when light interacts with participating media. Two effects are considered: Spontaneous emission covers the phenomena of emittance of radiant power due to excitation, i.e. energy transition to visible light. In-scattering accounts for incident radiance that is collected from all incoming light directions and that is scattered into the direction of the light transport.

For the phenomena of spontaneous emission the rate of augmentation of transmitted light in a medium can be expressed by adding the radiance contributed by the luminous particles in the media. In contrast, to the discussed extinction effects the change of radiance along a ray segment due to emission is solely dependent on the material properties of the participating media and therefore independent of the source radiance. Thus, the change in radiance can be denoted as

$$dL_\nu^\epsilon(\boldsymbol{r}(\lambda), \boldsymbol{d}) = \epsilon_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) d\lambda = g_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) d\lambda \tag{3.4}$$

with ϵ being the *emission coefficient* already given in radiant power per solid angle and projected area. Since spontaneous emission is the only source of radiant power in such a simplified optical model like it is discussed here, this input radiance is also commonly called the *source term* g.

The effect of in-scattering collects all contributions of incoming radiance L' that affect the light transport in the ray direction. The distribution of incoming radiance to outgoing directions is specified by a *phase function* p that models the underlying optical light distribution properties of the participating media. For the case of elastic scattering, i.e. the energy and wavelength is conserved during scattering, the *phase function* is only dependent on the location $\boldsymbol{r}$, the ray's direction of light transport $\boldsymbol{d}$, and the incoming light direction $\boldsymbol{d'}$. Taking again the *scattering coefficient* σ into account, the overall change in radiance due to incoming scattering can be defined as

$$dL_\nu^{is}(\boldsymbol{r}(\lambda), \boldsymbol{d}) = \left(\int_\Omega \sigma_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) p(\boldsymbol{r}, \boldsymbol{d'} \to \boldsymbol{d}) L'_\nu(\boldsymbol{r}, \boldsymbol{d'}) \cos\theta' d\omega'\right) d\lambda.$$

Radiative Transfer Equation By applying all of the aforementioned effects to the radiance transport along a line in space, the *equation of radiative transfer* can then be written in its integrodifferential form as

$$\begin{aligned} dL_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d}) &= dL_\nu^\kappa(\boldsymbol{r}(\lambda),\boldsymbol{d}) + dL_\nu^{os}(\boldsymbol{r}(\lambda),\boldsymbol{d}) \\ &\quad + dL_\nu^\epsilon(\boldsymbol{r}(\lambda),\boldsymbol{d}) + dL_\nu^{is}(\boldsymbol{r}(\lambda),\boldsymbol{d}) \\ &= -\left(\kappa_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d}) + \sigma_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})\right) L_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})d\lambda \\ &\quad + g_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})d\lambda \qquad (3.5) \\ &\quad + \left(\int_\Omega \sigma_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d}) p(\boldsymbol{r},\boldsymbol{d}' \to \boldsymbol{d}) L_\nu'(\boldsymbol{r},\boldsymbol{d}')\cos\theta' d\omega'\right) d\lambda. \end{aligned}$$

Depending on the applied optical model and the form of the optical coefficients used, various highly different approximations for this equation were proposed for many fields of applications, e.g. astrophysics, hydrodynamics, as well as computer graphics. For instance, realistic surface rendering in computer graphics is often based on the *rendering equation*, introduced independently by Immel et al. [ICG86] and Kajiya [Kaj86]. It is a specific version of the *equation of radiative transfer* stated in (3.5) that only accounts for emission and reflectance, i.e. surface in-scattering. With such an optical model, the *rendering equation* may be written in its non-spectral form for surface position $\boldsymbol{x}$, the incoming direction $\boldsymbol{d}$ from the surface, and incoming direction $\boldsymbol{d}'$ to the surface as

$$dL(\boldsymbol{x},\boldsymbol{d}) = g(\boldsymbol{x},\boldsymbol{d}) + \int_\Omega p(\boldsymbol{x},\boldsymbol{d}' \to \boldsymbol{d}) L'(\boldsymbol{x},\boldsymbol{d}')\cos\theta' d\omega'.$$

The resulting radiance at position $\boldsymbol{x}$ directed along $\boldsymbol{d}$ for an optical model accounting for emission and reflectance is consequently given by

$$L(\boldsymbol{x},\boldsymbol{d}) = L^\epsilon(\boldsymbol{x},\boldsymbol{d}) + \int_\Omega p(\boldsymbol{x},\boldsymbol{d}' \to \boldsymbol{d}) L'(\boldsymbol{x},\boldsymbol{d}')\cos\theta' d\omega'.$$

3.1.1 The Volume Rendering Integral

A different approximation of the *equation of radiative transfer* is commonly applied for direct visualization of volumetric scalar data sets. The three-dimensional data field acts as participating media that interferes with light from the background that passes through the media on its way to the observer. Several presumptions are made to map arbitrary 3D scalar input to the optical properties of the participating media and to allow for an efficient calculation of the radiance transport along a ray through the volume.

Premises The optical model is reduced to account for emission and absorption, but completely neglects any effect of scattering inside the media. By doing so, the radiance transport becomes solely dependent to the amount of radiance added or lost due to interaction with the media along the direction of the ray; very costly considerations of radiance changes due to events apart from the ray's line of sight are avoided. Applying the emission-absorption model to the integrodifferential *equation of radiative transfer* (3.5) leads to the following simplified description of light transport, written in its differential form as

$$dL_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}) = -\kappa_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d})L_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d})d\lambda + g_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d})d\lambda. \tag{3.6}$$

The *absorption coefficient* κ and the *source term* g denote the optical properties of the media and both need to be specified in order to determine the characteristics of light transport inside the volume. While the exact material properties are often unknown, for the purpose of direct volume visualization a suitable mapping of the scalar input data field to these optical properties is required. In this context, a suitable mapping denotes an appropriate mapping for the visual impression of the generated image under a given task, rather than trying to mimic the true physical properties of the involved materials that may often lead to less expressive graphical representations of the input data. For volume visualization this mapping step is denoted as *classification*, which is commonly specified in detail via *transfer functions* (cp. Section 3.2.2). Depending on the order of application of the *transfer functions* with respect to the interpolation of the input scalar field, one distinguishes between *pre-* and *post-classification*. With the former the transformation is applied before interpolation and can therefore be implemented as a pre-processing step to the input data, as mentioned for example by Levoy [Lev88]. In contrast, the latter uses already interpolated scalar values as input to the *transfer function*. In the general case, the two classification models lead to different results.

By further assuming the optical properties of the participating media are independent of the direction of radiance transportation, the relationship between the scalar input field $s(\boldsymbol{x})$ and the required optical properties using *post-classification* can be defined as

$$\kappa_\nu(\boldsymbol{r}(\lambda)) = \tilde{\kappa}_\nu(s(\boldsymbol{r}(\lambda))) \tag{3.7}$$

$$g_\nu(\boldsymbol{r}(\lambda)) = \tilde{\kappa}_\nu(s(\boldsymbol{r}(\lambda)))\,\tilde{c}_\nu(s(\boldsymbol{r}(\lambda))) \tag{3.8}$$

with $\tilde{\kappa}$ and $\tilde{c}$ being the *transfer function* for absorption and radiant emission power, respectively. The derivation of the *source term* in equation (3.8) is based on the assumption of a *local thermodynamic equilibrium* that states the *emission coefficient* to be equal to the *absorption coefficient* times the spectral radiance, given by Plank's law (see Chandrasekhar [Cha50] for a more detailed discussion). Thus, the emission can be expressed in terms of the absorption coefficient times the accordingly scaled radiant emission power. An alternative derivation of the same equation is given by Sabella [Sab88], who introduces a *density emitter model* that holds the same particles responsible for absorption as well as emission. In doing so, the *absorption coefficient* $\tilde{\kappa}$ specified with the *transfer function* can also be interpreted as particle density. On the one hand, this density provides a probability that a photon gets absorbed on its way through the volume. One the other hand, it also specifies the number of particles per volume that emit radiance. Thus, the radiant energy emitted by each particle $\tilde{c}$ is additionally weighted by the particle density, i.e. the *absorption coefficient.* In the graphics community this coupling is often referred to as *associated colors*, according to the definition by Blinn [Bli94]. Alternative descriptions of the *source term* also exist: Wilhelms and Van Gelder [WG91] uncouple both optical quantities to avoid the restrictions implied by the multiplicative combination; Wittenbrink et al. [WMG98] lessens the restrictions for *pre-classification* by introducing an opacity weighted color interpolation scheme.

Finally, by further assuming a constant *absorption coefficient* over the complete color spectrum, the simplified radiance transport equation for volume rendering can be derived by substitution of the optical properties in equation (3.6) by the corresponding quantities specified via the *transfer functions* (3.7) and (3.8), leading to the *volume rendering equation* in its differential form:

$$\frac{dL_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})}{d\lambda} = -\tilde{\kappa}(s(\boldsymbol{r}(\lambda)))L_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d}) + \tilde{\kappa}(s(\boldsymbol{r}(\lambda)))\tilde{c}_\nu(s(\boldsymbol{r}(\lambda))). \tag{3.9}$$

Integral Solution In order to determine the residual spectral radiance of light transport passing through a volume on its way from the initial source of radiance to the observer, the *volume rendering equation* needs to be evaluated along the ray's segment inside the media. Thus, the resulting radiance can be found by integrating the ordinary differential equation (3.9) along the complete distance a ray is located inside the volume. Following the description given by Max [Max95], the analytical solution for the *volume rendering equation* can be obtained by applying an integrating factor. Therefore, equation (3.9) is rearranged to

$$\frac{dL_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})}{d\lambda} + \tilde{\kappa}(s(\boldsymbol{r}(\lambda)))L_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d}) = \tilde{\kappa}(s(\boldsymbol{r}(\lambda)))\tilde{c}_\nu(s(\boldsymbol{r}(\lambda))), \tag{3.10}$$

which matches—written in this form—the more general description of a special class of ordinary differential equations, that can be expressed as

$$\frac{dL(\lambda)}{d\lambda} + f(\lambda)L(\lambda) = g(\lambda). \tag{3.11}$$

Differential equations of such type, can be made integrable by multiplication of an integration factor $e^{v(\lambda)}$. In case the function $v(\lambda)$ fulfills the two conditions

$$v(\lambda) = \int_0^\lambda f(\lambda')d\lambda', \qquad \frac{v(\lambda)}{d\lambda} = f(\lambda),$$

the left-hand side of equation (3.11) multiplied by the integration factor can be rewritten by applying the product rule:

$$\begin{aligned}\left(\frac{dL(\lambda)}{d\lambda}+f(\lambda)L(\lambda)\right)e^{v(\lambda)} &= g(\lambda)e^{v(\lambda)}\\ &= \frac{d}{d\lambda}\left(L(\lambda)e^{v(\lambda)}\right).\end{aligned}$$

Analogously, the transformation of the *volume rendering equation* given in equation (3.10), using an integration factor with function $v(\lambda)$ being $\int_0^\lambda \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'$, leads to

$$\frac{d}{d\lambda}\left(L_\nu(\boldsymbol{r}(\lambda),\boldsymbol{d})e^{\int_0^\lambda \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}\right)=\tilde{\kappa}(s(\boldsymbol{r}(\lambda)))\tilde{c}_\nu(s(\boldsymbol{r}(\lambda)))e^{\int_0^\lambda \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}.$$

The derived equation can now be integrated to evaluate the radiance transfer along the ray inside the volume; by definition, that is from its point of entrance at $\lambda = 0$ until the exit position at $\lambda = \xi$:

$$L_\nu(\xi)e^{\int_0^\xi \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}-L_\nu(0)=\int_0^\xi \tilde{\kappa}(s(\boldsymbol{r}(\lambda)))\tilde{c}_\nu(s(\boldsymbol{r}(\lambda)))e^{\int_0^\lambda \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}d\lambda.$$

Rearranging the previous equation to solve for $L_\nu(\xi)$, i.e the residual radiance at the exit position of the volume, results in the integral form of the radiance transport, which is also commonly called the *volume rendering integral*:

$$\begin{aligned}L_\nu(\xi) = {} & L_\nu(0)e^{-\int_0^\xi \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}\\ &+\int_0^\xi \tilde{\kappa}(s(\boldsymbol{r}(\lambda)))\tilde{c}_\nu(s(\boldsymbol{r}(\lambda)))e^{-\int_\lambda^\xi \tilde{\kappa}(s(\boldsymbol{r}(\lambda')))d\lambda'}d\lambda.\end{aligned} \tag{3.12}$$

By using the notion of the *optical transparency* T, as introduced previously in equation (3.3), the *volume rendering integral* (3.12) can be reformulated to a slightly shorter form, giving

$$L_\nu(\xi)=L_\nu(0)T(0,\xi)+\int_0^\xi \hat{\kappa}(\lambda)\hat{c}_\nu(\lambda)T(\lambda,\xi)d\lambda \tag{3.13}$$

with the $\hat{\kappa}$ and $\hat{c}_\nu(\lambda)$ denoting the shorter form of the corresponding quantities defined as $\hat{\kappa}(\lambda)=\tilde{\kappa}(s(\boldsymbol{r}(\lambda)))$ and $\hat{c}_\nu(\lambda)=\tilde{c}_\nu(s(\boldsymbol{r}(\lambda)))$.

The optical analogy to the mathematical description of the *volume rendering integral* corresponds to the underlying effects of the emission-absorption model. The first term accounts for the absorption of the incoming radiance from background light, which gets extenuated by the *transparency* of the media encountered along the ray's way through the volume. The second term denotes the contribution of radiance due to emission. For the complete ray segment located inside

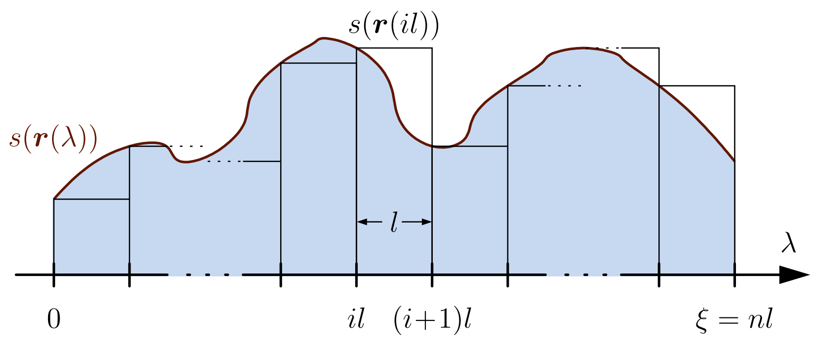


Figure 3.1: Approximation of the scalar function $s(\boldsymbol{r}(\lambda))$ along the ray section inside the medium. The distance is split into n uniform segments of length l. The ith slab is defined to span the interval $[il, (i+1)l]$ with a constant approximation of the scalar function using $s(\boldsymbol{r}(il))$ as basis.

the volume the emitted radiance of each position λ is integrated. As the effect of absorption also attenuates the emitted radiance along the remaining section of the ray through the volume, this portion of the transported radiance is scaled according to the volume's *transparency* between the actual location of radiance emission and the ray's exit point.

Discretization The task of Volume Visualization requires the evaluation of the *volume rendering integral* along all rays that transport radiance through the volume in the direction of the observer. However, for the general case no closed-form solution for the integral equation (3.13) exists. Williams and Max [WM92] propose an analytic solution under the assumption that the scalar field as well as the transfer function is piecewise integrable, i.e. both are defined as piecewise linear intervals. This solution can be further extended to transfer functions based on sums of Gaussians, as described by Kniss et al. [KPI$^+$03], but apart from those special cases a numerical approximation is required.

The underlying concept for the most commonly applied numerical integration is to split the ray segment located inside the volume into n uniformly sized segments of length $l = \xi/n$. In this way, each of the n slabs is defined to span a distinct interval $[il, (i+1)l]$ of the complete distance the ray interacts with the medium. Without losing generality, $i = 0$ denotes the sub-segment starting at the ray's entry point of the volume at $\lambda = 0$, whereas the upper bound of slab $i = n - 1$ is the ray's exit location at $\lambda = \xi$.

First, only considering the absorption terms of the *volume rendering equation*, the *transparency* of an arbitrary interval $[\lambda_1, \lambda_2]$ can be rewritten to tightly align with the borders of the newly introduced slabs. By definition of the optical *trans-*

parency (3.3) the arbitrary interval can be formulated as

$$\begin{aligned}T(\lambda_1,\lambda_2) &= T(\lambda_1,\lceil\lambda_1/l\rceil l)\times T(\lceil\lambda_1/l\rceil l,\lfloor\lambda_2/l\rfloor l)\times T(\lfloor\lambda_2/l\rfloor l,\lambda_2)\\ &= e^{-\int_{\lambda_1}^{\lceil\lambda_1/l\rceil l}\hat{\kappa}(\lambda')d\lambda'}\times e^{-\int_{\lceil\lambda_1/l\rceil l}^{\lfloor\lambda_2/l\rfloor l}\hat{\kappa}(\lambda')d\lambda'}\times e^{-\int_{\lfloor\lambda_2/l\rfloor l}^{\lambda_2}\hat{\kappa}(\lambda')d\lambda'}\\ &= e^{-\int_{\lambda_1}^{\lceil\lambda_1/l\rceil l}\hat{\kappa}(\lambda')d\lambda'}\times\prod_{i=\lceil\lambda_1/l\rceil}^{\lfloor\lambda_2/l\rfloor-1}e^{-\int_{il}^{(i+1)l}\hat{\kappa}(\lambda')d\lambda'}\times e^{-\int_{\lfloor\lambda_2/l\rfloor l}^{\lambda_2}\hat{\kappa}(\lambda')d\lambda'}.\end{aligned}$$

The inner term denotes the portion of *transparency* that can be expressed in terms of slabs that are fully enclosed by the arbitrary interval; the outer terms account for the conditionally necessary remaining border segments. Substitution in equation (3.13)—considering that the locations $\lambda = 0$ and $\lambda = \xi$ already align per definition with slab borders—gives

$$\begin{aligned}L_\nu(\xi) = L_\nu(0)&\prod_{i=0}^{n-1}e^{-\int_{il}^{(i+1)l}\hat{\kappa}(\lambda')d\lambda'}\\ &+\int_0^\xi\hat{\kappa}(\lambda)\hat{c}_\nu(\lambda)e^{-\int_{\lambda}^{\lceil\lambda/l\rceil l}\hat{\kappa}(\lambda')d\lambda'}\prod_{j=\lceil\lambda/l\rceil}^{n-1}e^{-\int_{jl}^{(j+1)l}\hat{\kappa}(\lambda')d\lambda'}d\lambda.\end{aligned}$$

Assuming a piecewise constant approximation of the scalar function $s(\boldsymbol{r}(\lambda))$ for each slab—and therefore also for $\hat{\kappa}(\lambda)$ and $\hat{c}_\nu(\lambda)$—as denoted in Figure 3.1, allows to simplify the absorption terms further, leading to the estimate of

$$L_\nu(\xi)\approx L_\nu(0)\prod_{i=0}^{n-1}e^{-\hat{\kappa}(il)l}+\int_0^\xi\hat{\kappa}(\lambda)\hat{c}_\nu(\lambda)e^{-\hat{\kappa}(\lambda)(\lceil\lambda/l\rceil l-\lambda)}\prod_{j=\lceil\lambda/l\rceil}^{n-1}e^{-\hat{\kappa}(jl)l}d\lambda.$$

With respect to the optical properties the sub-term $e^{-\hat{\kappa}(\lambda)(\lceil\lambda/l\rceil l-\lambda)}$ on the right-hand side denotes the local effect of self-attenuation of emitted radiance inside a single ray segment. In detail, it accounts for the loss of emitted radiance of a slab due to absorption effects in the same slab. For sufficiently small segment sizes l the limit of this absorption effect is

$$\lim_{l\to 0}e^{-\hat{\kappa}(\lambda)(\lceil\lambda/l\rceil l-\lambda)}=1,$$

thus, the self-attenuation of emitted radiance is commonly neglected. With this and by approximating the remaining integral with a Riemann sum—again under the premise of a piecewise constant *source term* per slab—the residual radiance can be written as

$$L_\nu(\xi)\approx L_\nu(0)\prod_{i=0}^{n-1}e^{-\hat{\kappa}(il)l}+\sum_{i=0}^{n-1}\hat{\kappa}(il)\hat{c}_\nu(il)l\prod_{j=i+1}^{n-1}e^{-\hat{\kappa}(jl)l}.$$

Even though, the *volume rendering integral* is already fully discretized in this form, an additional simplification is commonly applied to further reduce computational complexity. Therefore, the material property of the optical *opacity* α is introduced as being well defined as

$$\alpha(\lambda_1, \lambda_2) = 1 - T(\lambda_1, \lambda_2) = 1 - e^{-\int_{\lambda_1}^{\lambda_2} \kappa(\boldsymbol{r}(\lambda'))d\lambda'}$$

with respect to the optical *transparency* as previously described in equation (3.3). Analogously, the approximated *opacity* of slab i—with assumed constant $\hat{\kappa}(\lambda)$—can be written as

$$\alpha_i \approx 1 - e^{-\kappa(il)l} \approx -\kappa(il)l,$$

whereas the second approximation again implies the slab size l to be sufficiently small. In doing so, the final estimate for the *volume rendering integral* (3.13) in a discretized form can be written as

$$L_n \approx L_0 \prod_{i=0}^{n-1}(1-\alpha_i) + \sum_{i=0}^{n-1} \alpha_i C_i \prod_{j=i+1}^{n-1}(1-\alpha_j) \tag{3.14}$$

with C_i, L_i being defined as $C_i = \hat{c}_\nu(il)$ and $L_i = L_\nu(il)$ respectively. Or in the equivalent expanded form, giving

$$L_n = \alpha_{n-1}C_{n-1} + (1-\alpha_{n-1})\Big(\alpha_{n-2}C_{n-2} + (1-\alpha_{n-2})\Big(\dots$$
$$\dots \alpha_1 C_1 + (1-\alpha_1)\Big(\alpha_0 C_0 + (1-\alpha_0)L_0\Big)\dots\Big)\Big).$$

Depending on the order of evaluation of equation (3.14), two iterative descriptions of the process of volume rendering can be derived. As this order intuitively depicts the direction in with the slabs of a ray are processed, the two variants are usually called *back-to-font* and *front-to-back* compositing. The former follows the direction of light transport from the incident radiance from the background light L_0 to the ray's exit location. The resulting radiance estimate L_n can be evaluated by iterating i in the range of $[0, n-1]$ using

$$L_{i+1} = \alpha_i C_i + (1-\alpha_i)L_i \tag{3.15}$$

as *back-to-front* compositing operator. In the terminology of Porter and Duff [PD84] this corresponds to the operator "A over B", which specifies the placement of a transparent foreground A in front of background B. In this sense, volume rendering is often referred to as iterative compositing of new ray samples, i.e. the foreground, to the already combined samples, i.e. the background.

In case of *front-to-back* compositing the order of evaluation is reversed, but the overall approximation of the resulting radiance remains the same. With $\alpha_{[n..i]}$ being the accumulated opacity for the interval of slab n to slab i, the *front-to-back* compositing with i iterating from $ni-1$ to 0 can be written as

$$\begin{aligned} L_{i-1} &= (1-\alpha_{[n..i]})\alpha_i C_i + L_i \\ \alpha_{[n..i-1]} &= (1-\alpha_{[n..i]})\alpha_i + \alpha_{[n..i]}, \end{aligned} \tag{3.16}$$

using $L_{n-1}=0$ and $\alpha_{[n..n-1]}=0$ for initialization. Compared to *front-to-back* rendering this evaluation variant requires to explicitly keep track of the accumulated *transparency*. Nevertheless, both compositing schemes are commonly applied in volume rendering depending on the favored rendering technique (Section 3.2).

3.1.2 The Kubelka-Munk Optical Compositing Model

The *volume rendering equation*, as described in the previous section, builds upon an optical model without any scattering effects; thus, the system's light distribution can be completely modeled by solely evaluating the radiance transport along the rays through the participating medium in the direction of light propagation towards the observer. An exact and complete solution of the integrodifferential equation of *radiative light transfer* (3.5), however, would require to capture the total distribution of radiance of a scene in order to determine the incident radiance due to *in-scattering* for a single location in space.

A less computationally expensive alternative was firstly introduced by Schuster [Sch05] in 1905 for representing atmospheric scattering of light, i.e. light transmission through gas that holds fine particles of matter that have a significant effect on the light distribution—not accurately representable via an emission-absorption model. Hence, he based his optical model on the effects of absorption, emission, *and* scattering, but assumes that the scattered radiation is not arbitrarily distributed in space, but only spreads forward and backward with respect to the direction of light transport. This description of light transport in a participating media is also commonly referred to as *two-flux approach*, due to the inherent interrelation between the radiance transport along pairs of bi-directional rays.

With such an underlying optical model, the interaction of radiance transport along a line through participating media, i.e. a single thin layer of material of thickness χ, can be expressed as a system of two differential equations; each equation accounting for one ray along that line, oriented in exactly opposite directions of each other. In the following, the radiance connected to the two rays will be denoted as

$$I_\nu(\lambda) = L_\nu(\boldsymbol{r}(\lambda), \boldsymbol{d}), \qquad J_\nu(\lambda) = L_\nu(\boldsymbol{r}(\lambda), -\boldsymbol{d}),$$

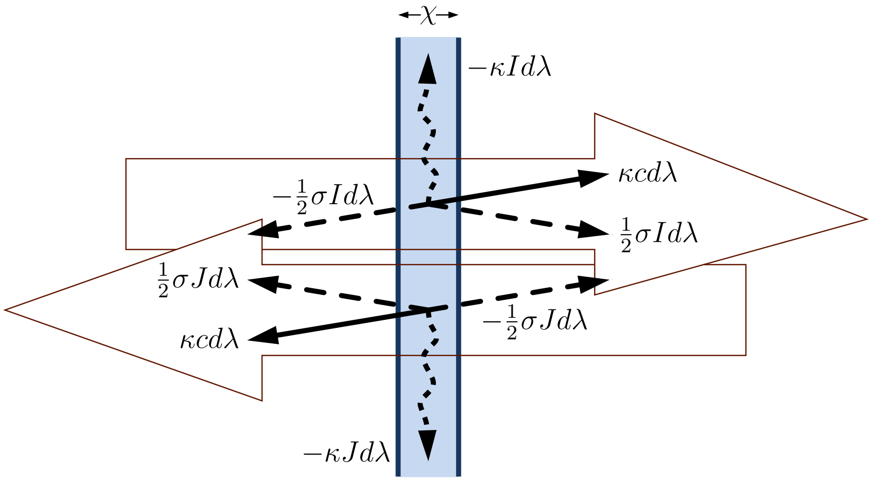

χ
$-\kappa I d\lambda$
$\kappa c d\lambda$
$-\frac{1}{2}\sigma I d\lambda$
$\frac{1}{2}\sigma I d\lambda$
$\frac{1}{2}\sigma J d\lambda$
$-\frac{1}{2}\sigma J d\lambda$
$\kappa c d\lambda$
$-\kappa J d\lambda$

Figure 3.2: Bi-directional light transport through a thin layer of media described in a two-flux optical model. The radiance of the two opposing rays I, J is decreased due to absorption (dotted arrows) and increased due to emission (solid arrows). The scattered radiance is equally split into a forward- and backward-directed fraction (dashed arrows); again contributing to the light transport in the two directions.

to distinguish between the direction of light transport. Considering, at first, only one of the two rays with the radiant energy I, the absorption radiance along the thin layer can be expressed using the *absorption coefficient* κ as $-\kappa I d\lambda$ (3.1) and the emission is either given by the *source term* $g d\lambda$ (3.4) or more specifically as $\kappa c d\lambda$ (3.8). By only considering the radiance transport along the single line in space—but in both corresponding ray directions—the lost radiance due to *out-scattering*, denoted as $-\sigma I d\lambda$ (3.2), is assumed to be scattered in equal parts in the two directions along the line. Thus, the incident radiance due to *in-scattering* is simplified to match half of the radiance scattered out by the light transport in the opposite direction (Figure 3.2). Combining the contributions of all mentioned effects—neglecting the parameters for the various coefficients—leads to the system of differential equations proposed by Schuster [Sch05], that couple the light transport along a pair of bi-directional rays as

$$\begin{aligned} -\frac{dI_\nu(\lambda)}{d\lambda} &= \kappa_\nu\Big(c_\nu - I_\nu(\lambda)\Big) + \frac{1}{2}\sigma_\nu\Big(J_\nu(\lambda) - I_\nu(\lambda)\Big) \\ \frac{dJ_\nu(\lambda)}{d\lambda} &= \kappa_\nu\Big(c_\nu - J_\nu(\lambda)\Big) + \frac{1}{2}\sigma_\nu\Big(I_\nu(\lambda) - J_\nu(\lambda)\Big). \end{aligned} \tag{3.17}$$

The change of sign between dI and dJ denotes the difference in change of radiance for the two rays in a common direction with respect to the ray parameter λ.

Several extensions to this original optical model were proposed by Kubelka and Munk [KM31] in the context of colored paint coatings in order to weaken the presumptions made for deriving these equations. However, for the field of their special application the effect of emission is inexistent and is therefore typically ignored. While the initial model assumes the light transport to be independent to the angle between the ray direction and the surface normal of the material layer, Kubelka [Kub48] extended the model accordingly and proved the initial assumption to be correct only in case of perfectly diffuse illumination and dull materials. With K and S representing the geometric absorption and scattering coefficients already in respect to the thickness of the thin layer of material in the direction of light propagation, equation (3.17) can be rewritten by replacing the true optical coefficients based on the physical properties of the participating media as

$$\begin{aligned} -dI_\nu(\lambda) &= -\left(S_\nu + K_\nu\right) I_\nu(\lambda)d\lambda + SJ_\nu(\lambda)d\lambda \\ dJ_\nu(\lambda) &= -\left(S_\nu + K_\nu\right) J_\nu(\lambda)d\lambda + SI_\nu(\lambda)d\lambda, \end{aligned} \tag{3.18}$$

which are in general referred to as *Kubelka-Munk equations.*

A solution for this system of differential equations can be derived under the following two assumptions: First, the participating medium is composed of a single, homogeneous layer of thickness χ, that is coated atop of a perfectly black background surface, i.e no incoming radiance is reflected from the background at all. Second, the radiant power from scene lighting is exclusively emitted on the opposite side of the background layer. Thus, the layer of medium is defined to have a lit and an unlit surface. Additionally shifting the ray parameter λ to be aligned with the unlit side of the material layer at $\tilde{\lambda} = 0$, the solution to the *Kubelka-Munk equation* (3.18) can be written in term of the incident radiance on the lit surface layer $I(\chi)$ as

$$\begin{aligned} I_\nu(\tilde{\lambda}) &= I_\nu(\chi)\frac{a\sinh(bS_\nu\tilde{\lambda}) + b\cosh(bS_\nu\tilde{\lambda})}{a\sinh(bS_\nu\chi) + b\cosh(bS_\nu\chi)} \\ J_\nu(\tilde{\lambda}) &= I_\nu(\chi)\frac{\sinh(bS_\nu\tilde{\lambda})}{a\sinh(bS_\nu\chi) + b\cosh(bS_\nu\chi)} \end{aligned} \tag{3.19}$$

where a and b are defined as

$$a = \frac{S_\nu + K_\nu}{S_\nu}, \qquad b = \sqrt{a^2 - 1}.$$

For most practical purposes based on this optical model, the radiance transmitted and reflected by the layer of participating media is more of an interest for evaluation than the radiant energy inside the material itself. Therefore, the last

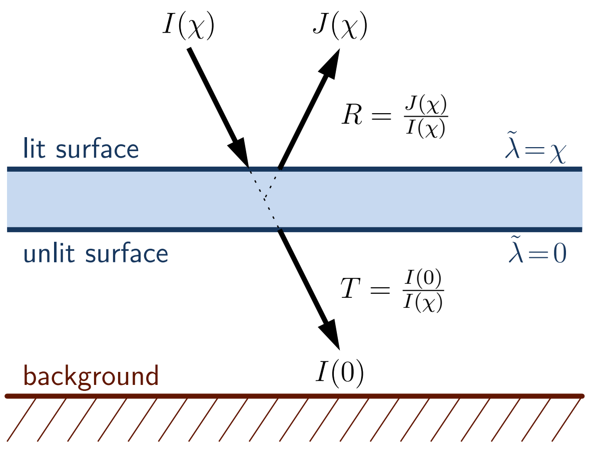


Figure 3.3: Two-flux approach for a single color layer of thickness χ coated on a perfectly black background. The reflectance R and transmittance T are both given with respect to the corresponding surface of emission.

set of equations are often expressed in terms of these two energy fluxes. With respect to the shifted ray parameter $\tilde{\lambda}$, the reflectance $R = J(\chi)/I(\chi)$ can be defined as the portion of the incoming radiance on the lit surface of the layer $I(\chi)$ that is reflected as the outgoing radiance at the same location in the opposite direction $J(\chi)$; whereas the transmittance $T = I(0)/I(\chi)$ is given by the portion of the incoming light that leaves the layer of material in the direction of the incoming light on the unlit surface side $I(0)$ (see Figure 3.3). Thus, for the two critical energy fluxes R and T the equations (3.19) simplify to

$$\begin{aligned} T_\nu &= \frac{b}{a\sinh(bS_\nu\chi) + b\cosh(bS_\nu\chi)} \\ R_\nu &= \frac{\sinh(bS_\nu\chi)}{a\sinh(bS_\nu\chi) + b\cosh(bS_\nu\chi)}. \end{aligned} \tag{3.20}$$

Kubelka [Kub54] further extended the optical model to account for multiple layers of participating media, of which each layer is assumed to be formed by a homogeneous material, but the properties of the individual layers may vary. Thus, by thinking of a volumetric media as a stack of multiple independent layers, this new model is able to evaluate light transport through a volume of media. However, by introducing a stack of multiple layers to the optical model requires the interaction between those layers to be accounted for in order to evaluate the overall radiance fluxes of reflectance and transmittance passing through all the layers.

Based on the discussed single layer model, for the case of two thin layers A

and B—possibly of variant material properties as well as thickness—the basic characteristics of light transport for each layer can be described in terms of their reflectance R_A,R_B and transmittance T_A,T_B. Without loosing generality, the test setup considered in the following assumes layer A be coated first; thus, A is underneath of layer B with respect to the normalized initial radiance source $I_A(\chi) = 1$, i.e. the unlit surface side of layer B is facing the lit surface side of layer A. According to the single layer model, the overall incident radiance of scene lighting is split by layer B into the reflected portion R_B, while the transmitted portion T_B passes through B and acts as incident radiance for layer A. Similarly, this portion T_B is divided further by layer A leading to a transmitted radiance portion of T_BT_A for passing through both material layers and to a reflected portion of T_BR_A, that is send back from the lit surface of A to layer B. The latter energy hits layer B on the unlit surface and—assuming the material to have identical optical properties when lit from either surface side—is again split into a transmitted portion $T_BR_AT_B$ and another reflected portion $T_BR_AR_B$. Since reflectance always covers for a portion of energy sent back to the other surface, the overall interaction between the two layers leads to an infinite series of reflected and transmitted light (compare Figure 3.4). In terms of scattering, this infinite progression of light between the two layers is an iterative solution to the *in-scattering* term of the differential *equation of radiative transfer* (3.5), simplified to match the premises of the two-flux model, i.e. defined as a one-dimensional interrelation between two distinct positions in space. Adding up the contributions for reflectance for the complete test setup, i.e. all energy leaving layer B on the lit surface side in the opposite direction of scene lighting, gives for a single wavelength ν

$$R_{A,B} = R_B + \sum_{n=0}^{\infty} T_B^2 R_A R_A^n R_B^n = R_B + \frac{T_B^2 R_A}{1 - R_A R_B}.$$

This geometric series converges due to the premise $|R_AR_B| < 1$, i.e the reflectance of at least one material is assumed to be non-perfect. Correspondingly, the total transmittance for the unlit surface side of layer A can be written as

$$T_{A,B} = \sum_{n=0}^{\infty} T_B T_A R_A^n R_B^n = \frac{T_A T_B}{1 - R_A R_B}.$$

By induction this combination of two layers of material can be extended to arbitrary numbers of layers, allowing to iteratively combine all individual layers to the overall characteristics of a complete stack of participating media. Given a stack of n material layers ordered in the sequence they were applied on top of each other, i.e. layer $i = 0$ is coated first on the background surface, the iterative operators can be classified again in terms of *back-to-front* and *front-to-back* compositing (cp. Section 3.1.1). In the former case, the layers are processed in the

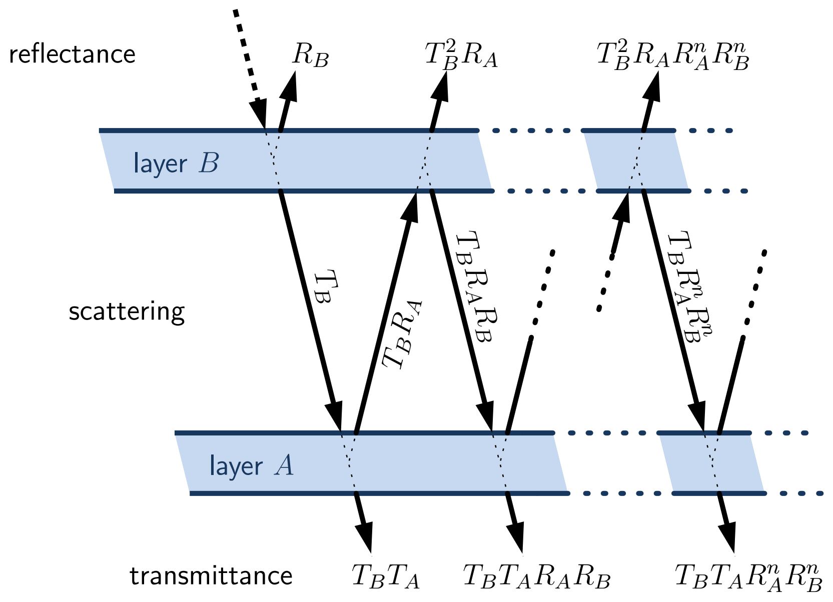


Figure 3.4: Light propagation between two layers of material in the *Kubelka-Munk* optical model. The overall reflectance and transmittance for the two-phase medium is given as an infinite series due to endless mutual scattering between the layers.

order they are varnished on a carrier surface; with i iterating from 0 to $n-1$ the *back-to-front* compositing operator is given as

$$\begin{aligned} R_{[0\ldots i+1]} &= R_{i+1} + \frac{T_{i+1}^2 R_{[0\ldots i]}}{1 - R_{[0\ldots i]}R_{i+1}} \\ T_{[0\ldots i+1]} &= \frac{T_{[0\ldots i]}T_{i+1}}{1 - R_{[0\ldots i]}R_{i+1}}. \end{aligned}$$

Analogously, the *front-to-back* operators for the two fluxes of reflectance and transmittance—with i iterating in the range $[n-1, 0]$—can be written as

$$\begin{aligned} R_{[n\ldots i-1]} &= R_{[n\ldots i]} + \frac{T_{[n\ldots i]}^2 R_{i-1}}{1 - R_{[n\ldots i]}R_{i-1}} \\ T_{[n\ldots i-1]} &= \frac{T_{[n\ldots i]}T_{i-1}}{1 - R_{[n\ldots i]}R_{i-1}}. \end{aligned} \tag{3.21}$$

While the operators differ for the order of evaluation, it can be shown that the overall transmittance T is independent of the processing direction, which is also an important optical property that has been verified experimentally.

3.2 Direct Volume Visualization Techniques

Independent of the underlying optical model chosen for emulating the physical effects of light propagation through participating media, a large number of different visualization techniques are published to efficiently map those computations to computer hardware. Nevertheless the majority share the common tasks of sampling the volumetric input data field (Section 3.2.1) and providing a suitable mapping of the input data to the required optical properties, e.g. color and opacity (Section 3.2.2). Major differences arise from implementing the integration of the volumes contribution to light transport through the media in direction to the observer in order to obtain a final image on the screen. Some of the most important techniques, especially in the context of interactive visualization applicable to hardware acceleration, are subsequently discussed (Section 3.2.3).

3.2.1 Data Sampling

In relation to the physical notion of a volume of participating media, the input data field for volume visualization is commonly interpreted to define some kind of scalar density distribution—or any other scalar quantity that may be mapped to the physical properties of emission, absorption, or scattering—in a three-dimensional domain. While the input data is normally only given at distinct positions in space, e.g. from sensor probes scattered in space during a physical experiment, the evaluation of light transport along an arbitrary line through the data domain requires to quantify the density at any position $\boldsymbol{x} = (x, y, z)$ in the three-dimensional Euclidean space R^3. Thus, the sampling of non-continuous input data must resort to data interpolation with an underlying function f of the form $f : R^3 \to R$. In more detail, given an input set of n samples at position $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}$ with corresponding scalar values of $s_0, \ldots, s_{n-1}$ the interpolating function f is required to satisfy the condition $s_i = f(\boldsymbol{x}_i)$ for all input samples S_i.

Scattered Data Interpolation Without imposing any restrictions to the given positions or their interconnection the set of input data samples is commonly called *scattered data* or *meshless data*. A multitude of methods exist to interpolate a scalar value at any position $\boldsymbol{x}$ from such an arbitrary point set, which are commonly referred to as *scattered data interpolation*. In the following, only few important fundamental techniques are further discussed:

By simply assigning the scalar value s_j of the closest input data sample S_j with respect to the Euclidean distance $d(\boldsymbol{x}, \boldsymbol{x}_i)$ from the interpolation position $\boldsymbol{x}$ to the input samples $\boldsymbol{x}_i$, an interpolation function f can be defined that fully satisfies the required conditions stated above. Due to its nature such type of interpolation is commonly called *nearest neighbor* interpolation or *point sampling*. The derived interpolating function f is discontinuous unless all input scalar samples are of the same value, i.e. $s_0 = s_i$ for all $i = 1 \ldots n-1$, and piecewise constant with value s_j in the region V_j that is defined as

$$V_j = \{\boldsymbol{x} \in R^3 \mid d(\boldsymbol{x}, \boldsymbol{x}_j) \leq d(\boldsymbol{x}, \boldsymbol{x}_i) \; \forall i = 0 \ldots n-1\}.$$

The obtained regions V_i, named *Dirichlet regions* [Dir50] or in the context of geometry processing more frequently referred to as *Voronoi cells* [Vor08], partition the space into convex polytopes in such a way that each cell contains exactly one input sample S_j. While the *nearest neighbor* interpolation scheme only takes the single input sample of the cell that enclose the interpolation position into account, alternative reconstruction schemes based on the *Voronoi cell* partitioning were presented; for example, the class of *natural neighbor* interpolation methods [Sib81] use a weighted combination of input samples, selected on the basis of the topology of the *Voronoi cells* around the interpolation position.

In contrast to selecting only a small set of spatially close input samples for interpolation, *inverse distance weighting*, also named *Shepard interpolation* [She68], uses all input data samples for deriving an interpolated combination for arbitrary positions. The influence and thus the contribution of each input sample to the final scalar interpolation result is defined to decrease with larger distances between the input position and the interpolation location. The interpolation function f is originally defined as

$$f(\boldsymbol{x}) = \frac{\sum_{i=0}^{n-1} w_i(\boldsymbol{x}) s_i}{\sum_{i=0}^{n-1} w_i(\boldsymbol{x})} \qquad \text{with} \qquad w_i(\boldsymbol{x}) = \frac{1}{d(\boldsymbol{x}, \boldsymbol{x}_i)^p},$$

where the parameter p with $p \geq 1$ controls the smoothness of the reconstruction. Related to this class of techniques is the concept of interpolation based on *radial basis functions* [Har71], that generalize the interpolation function to be of the form $f(\boldsymbol{x}) = \sum a_i h_i(\boldsymbol{x})$, where $h(\boldsymbol{x})$ may be chosen as some arbitrary—most often Gaussian-typed—function and a_i being coefficients defined in such a way that f meets the required properties for interpolation. This way, *inverse distance weighting* becomes a special case of interpolation using *radial basis functions.*

Grid-based Interpolation for Unstructured Data Finally, the last family of interpolation schemes for *scattered data* discussed here, introduce a partitioning of the space based on connecting the given input samples by line segments to form edges of cells, that ultimately form a grid structure with an explicitly defined topology. The interpolation at an arbitrary position can then be split in the tasks of identifying the corresponding grid cell, deriving the *local coordinates* relative to the vertices of the selected cell, and subsequently performing the data interpolation based on the scalar values attached to the cell corners, i.e. the input data

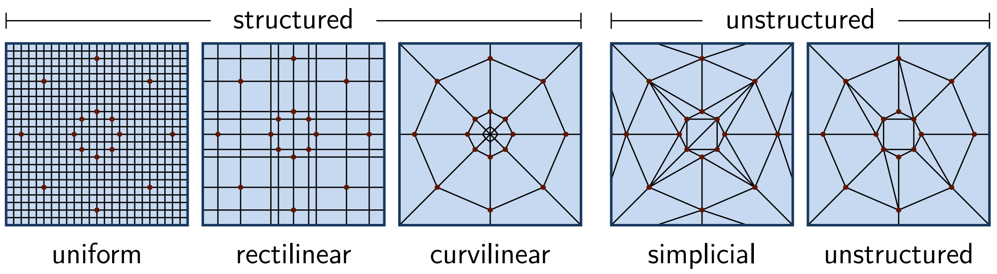


Figure 3.5: Classification of common two-dimensional grid types for a given input sample set (black dots). Additional sample positions (grid line intersections) need to be introduced to represent the input data with the regular layouts.

samples that span the cell. Thus, the basic shape of the cells of such a constructed grid are in general restricted to convex polyhedra, e.g. tetrahedra, cubes, prisms, or pyramids, in order to allow for a well-defined cell interpolation. If not further restricted in any way the derived partitions are referred to as *unstructured grids* (Figure 3.5). Often applied to the field of finite element analysis, such arbitrary grids offer high flexibility and allow local adaption to arbitrary volume border conditions, but also impose a high complexity in handling a multitude of different cell types and often cannot guarantee a continuous interpolation function across cell borders. Hence, a more restrictive sub-class, the *simplicial grids* (Figure 3.5), further presume the grid to be constructed solely from simplex shaped cells, i.e. only from tetrahedra in a three-dimensional space. Deriving such constrained partitions from arbitrary input samples is an research field of its own; with the most prominent technique being the *Delaunay triangulation* [Del34], resulting in a unique 'well-shaped' tetrahedral partition of the three-dimensional space for any given input set—that actually corresponds to the dual graph of the aforementioned partition of space by means of *Voronoi cells*. The term 'well-shaped' refers to the characteristics of the partition to be as equilateral as possible, that is the minimal angle of the cells in maximized.

In the context of volume rendering *simplicial grids* are commonly used for visualizing unstructured input data due to the inherent property of enabling a continuous linear interpolation across all grid cells. For a single tetrahedral cell with the vertices $S_0 \dots S_3$, an arbitrary position for interpolation can be expressed as a linear combination of the corner positions, giving

$$\boldsymbol{x} = \alpha \boldsymbol{x}_0 + \beta \boldsymbol{x}_1 + \gamma \boldsymbol{x}_2 + \delta \boldsymbol{x}_3.$$

As this equation is over-estimated for the three-dimensional space, a unique description of any given point can be derived by additionally demanding the coefficients to meet the condition $\alpha + \beta + \gamma + \delta = 1$. For any linear function, i.e. the

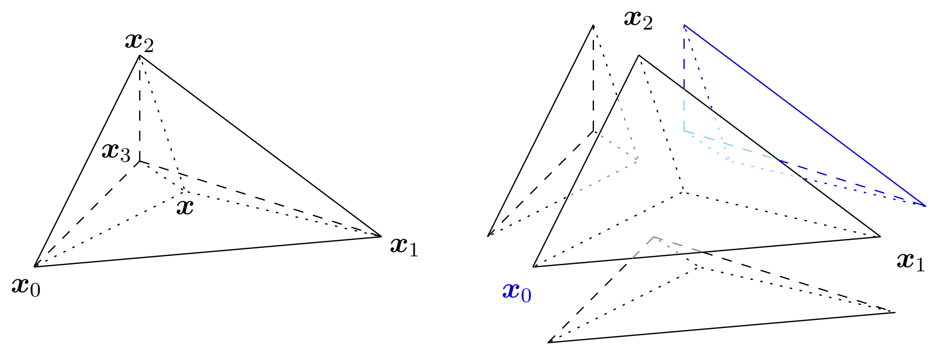


Figure 3.6: Partition of a tetrahedra into four sub-simplexes at position x. The *barycentric coordinates* are defined as the volumetric ratio between the sub-simplexes (right) and the original tetrahedra (left); here, each vertex is assigned to the opposed sub-simplex, i.e the simplex that is constructed without the vertex itself (depicted in blue for the vertex x_0).

interpolation function f, follows accordingly

$$\begin{aligned} f(\boldsymbol{x}) &= \alpha f(\boldsymbol{x}_0) + \beta f(\boldsymbol{x}_1) + \gamma f(\boldsymbol{x}_2) + \delta f(\boldsymbol{x}_3) \\ f(\boldsymbol{x}) &= \alpha s_0 + \beta s_1 + \gamma s_2 + \delta s_3. \end{aligned}$$

Thus, the identical coefficients also act as local coordinates for the interpolation of the scalar function. Originally derived from their relevance in identifying the center of gravity for a simplex with the coefficients attached as weights at the corresponding vertices, the local coordinates are commonly called *barycentric coordinates* or *barycentric weights* [Möb27]. Their exact values can be derived by solving the corresponding linear system of equations—coordinates of the positions in the three dimensional space plus the linear dependency of the coefficients. In geometric terms, this linear system describes the interpolation as volume-weighted combination based on the partitioning of the original simplex into $N+1$ sub-simplexes at the interpolation position in a N-dimensional space (Figure 3.6). The *barycentric coefficients* are defined as

$$\alpha = \frac{|\boldsymbol{x}\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3|}{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3|}, \quad \beta = \frac{|\boldsymbol{x}_0\boldsymbol{x}\boldsymbol{x}_2\boldsymbol{x}_3|}{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3|}, \quad \gamma = \frac{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}\boldsymbol{x}_3|}{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3|}, \quad \delta = \frac{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}|}{|\boldsymbol{x}_0\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3|},$$

whereas the operator $|\ldots|$ denotes the volume of the corresponding simplex. Consequently, the constructed ratios naturally sum up to one.

Some applications may require a more smooth interpolation function, especially of higher-order, e.g. to additionally guarantee a continuous gradient reconstruction from scattered data. In particular in the context of finite element analysis cubic

C^1 continuous interpolation schemes based on the Clough-Tocher method [WF87] play an important role for data reconstruction. In addition, also *natural neighbor* methods can be extended to provide higher-order reconstruction of irregular tetrahedral grids with continuous cell boundaries [BS04]. However, for interactive volume visualization such more sophisticated techniques are often too expensive and complex to evaluate; thus, they will not be further discussed here.

Grid-based Interpolation for Structured Data Unlike completely arbitrary scattered input data, in many cases—and especially in the specific field of volume visualization—the input data is actually often provided in a more regular structure, e.g. from regularly aligned sensor arrays such as photoactive CCD sensors or X-ray detector arrays in a CT scanners. In contrast to *scattered data*, where an explicit topology needs to be specified for defining a unique grid structure, such *structured data* already inherently provide a rectangular matrix structure as underlying topology. Thus, the input sample positions can be accessed by just using integer offsets i,j,k in a three-dimensional space. Finding the neighboring samples is trivially achieved by increasing or decreasing those offsets by one on each axis. Depending on the generality in specifying the input sample locations $\boldsymbol{x}_{ijk}$, we distinguish between *curvilinear*, *rectilinear*, and *uniform grids* (Figure 3.5).

Starting by the least restrictive layout, the input samples of a *curvilinear grid* can be expressed as $\boldsymbol{x}_{ijk} = (x(i,j,k), y(i,j,k), z(i,j,k))$, where the coordinates in each dimension relate to an arbitrary function of all integer offsets; analogously, a full vector lookup table with i,j,k as indices can be stored to specify the geometry of such grids. Despite the underlying regular structure—in general assumed to be non-intersecting and non-degenerated—*curvilinear grids* offer the possibility to naturally describe non-rectangular spaces, i.e. they can be adapted to approximate cylindrical or spherical shapes. This flexibility, however, comes at costs of having varying spatial directions between neighboring cells. While this property does not intrinsically hinder C^0 continuous interpolation across cell borders, higher-order interpolation or algorithms based on a global description of positions and directions are hindered. One common approach to this are C-Space algorithms, that map the true curvilinear geometric layout, named physical space (P-Space), to a computational space (C-Space), i.e. a grid of the same topology but with directional uniformity, to ease computations. The actual mapping of directions from P- to C-Space is thereby usually implemented using the *Jacobian matrices.*

One possible candidate structure for such a computational space is the class of *rectilinear grids.* Their input samples conform to the more strict condition $\boldsymbol{x}_{ijk} = (x(i), y(j), z(k))$, where the coordinates of each dimension is only dependent to its corresponding integer offset; and the only valid cell type is restricted to cuboids. Hence, such grids feature directional uniformity, but still allow for quasi-

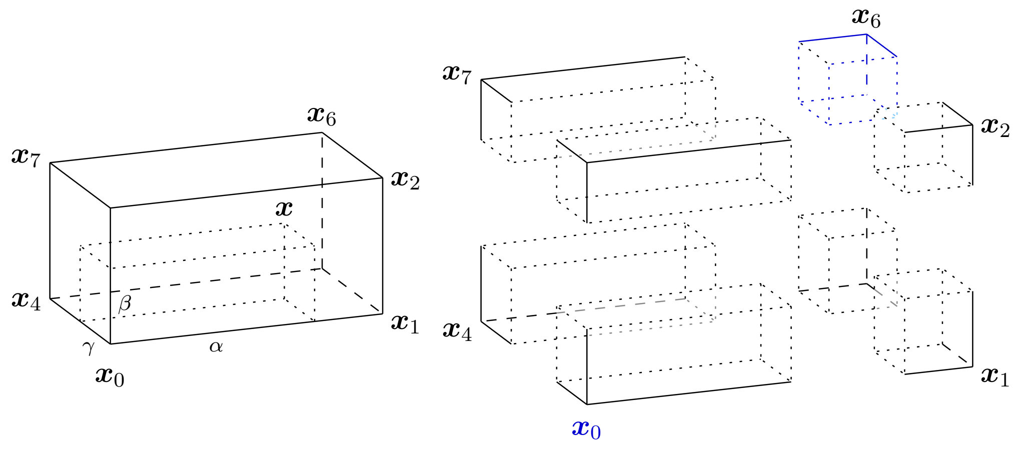


Figure 3.7: Partition of a cuboid cell into eight sub-cuboids at position $\boldsymbol{x}$. The *local coordinates* α,β,γ relate to the ratios of partition along each axes. For interpolation each vertex is weighted with the volumetric fraction of the opposed sub-cuboid (depicted in blue for the vertex $\boldsymbol{x}_0$).

local adaption of the grid to the underlying experiment or simulation.

Finally, the last discussed sub-type of *structured grids* further restricts the possible pattern of sample point locations to $\boldsymbol{x}_{ijk} = (x + i\Delta x, y + j\Delta y, z + k\Delta z)$, where Δx denotes a globally uniform cell size in the x-dimension and Δy, Δz are defined accordingly. In addition to directional uniformity, such grids offer uniformity in distances from cell to cell in each dimension separately as well as homogeneous cell volumes across the complete grid. In the special case of equilateral cells, i.e. $\Delta x = \Delta y = \Delta z$, one also speaks of a *Cartesian grid.* Despite the missing support for local adaption, *uniform grids* play a major role in interactive volume visualization due to its special set of characteristics, that enable highly efficient algorithms and computations, such as gradient calculation for example.

With respect to interpolation, *structured grids* also allow for an efficient linear C^0 continuous interpolation function, similar to the *barycentric interpolation* for simplexes. In fact, both schemes reduce to a linear interpolation along a line in the one-dimensional case. In a three-dimensional space, the corresponding *tri-linear interpolation* is derived by the tensor product of linear interpolations in the direction of the three axis. The used weights attached to the eight vertices of a cuboid for linear interpolation correspond to the *local coordinates* of the interpolation location $\boldsymbol{x}$ in relation to the cell's geometry. For a grid cell, spanned by the input samples $\boldsymbol{x}_0 \dots \boldsymbol{x}_7$, the local coordinates can be written as (see also Figure 3.7)

$$\alpha = \frac{|\boldsymbol{x} - \boldsymbol{x}_0|}{|\boldsymbol{x}_1 - \boldsymbol{x}_0|}, \qquad \beta = \frac{|\boldsymbol{x} - \boldsymbol{x}_0|}{|\boldsymbol{x}_3 - \boldsymbol{x}_0|}, \qquad \gamma = \frac{|\boldsymbol{x} - \boldsymbol{x}_0|}{|\boldsymbol{x}_4 - \boldsymbol{x}_0|}$$

Thus, the local position, but more importantly also a linearly interpolated scalar function, can be reconstructed by the interpolation function f, given in respect to the scalar values $s_0 \dots s_7$ attached to the cell vertices as

$$\begin{aligned} f(\boldsymbol{x}) = & s_0\,(1-\alpha)(1-\beta)(1-\gamma) + s_4\,(1-\alpha)(1-\beta)(\gamma) + \\ & s_1\,(\alpha)(1-\beta)(1-\gamma) + s_5\,(\alpha)(1-\beta)(\gamma) + \\ & s_2\,(\alpha)(\beta)(1-\gamma) + s_6\,(\alpha)(\beta)(\gamma) + \\ & s_3\,(1-\alpha)(\beta)(1-\gamma) + s_7\,(1-\alpha)(\beta)(\gamma). \end{aligned}$$

Analogously to the *barycentric weights*, the factors for weighting the scalar components for the *tri-linear interpolation* geometrically represent the volumetric ratio of the opposing sub-cuboid in relation to the unpartitioned cell (Figure 3.7).

In the context of volume rendering geared towards acceleration using graphics hardware, *uniform grids* are the most frequently used data structure for storing and processing input data. This is due to the natural mapping of these grids to the concept of textures, which also expose on-the-fly interpolation transparently performed by the graphics hardware during data access. Currently the typically available filters are *nearest neighbor* interpolation as well as well as *linear* interpolation for one, two, and also three dimensions. Beyond the directly supported interpolation mechanisms, a variety of alternative interpolation schemes for volume reconstruction, especially of higher order, were studied. Some examples among these include cubic B-spline interpolation [Lev88], windowed Gaussian filters [Wes90, LH91], or reconstruction based on weighted Chebychev approximations [Car93]. In the special context of GPU-based volume rendering, Sigg and Hadwiger [SH05] presented an approach to reduce the overhead for cubic texture filtering, that is also applicable to three dimensions by efficiently utilizing the hardware supported linear texture filtering. This method is discussed in more detail in Section 4.1 in the context of efficient convolutional operators for image processing.

A detailed qualitative comparison of those and other reconstruction methods is given by Marschner and Lobb [ML94]. More recently, Thèvebaz et al. [TBU00] use a frequency domain analysis for quality and performance comparison for a large number of reconstruction filters, while Möller et.al. [MMMY97] base their study on the analysis of the Taylor series expansion of the interpolation filters' convolution sum. However, while most of those more elaborate techniques impose an extensive computational overhead to volume sampling—in particular when mapped to graphics hardware—they still do not play a major role in interactive volume visualization.

3.2.2 Classification

The process of mapping the input data, or their continuously interpolated data samples, to the actual physical quantities of emission and absorption, is commonly implemented via *transfer functions* (cp. equations (3.7) and (3.8)). As the design of such functions exposes direct control over the final rendering results, the classification is actually the main interface for user interaction. Rather than remodeling the true optical properties of the captured volumetric probe, volume visualization and in particular the process of creating a suitable mapping function is more about visual data exploration. While the commonly used manual specification of the transfer functions basically equals a trial-and-error approach, a variety of concepts exist to aid this process: Ranging from specific user interface methodologies, e.g. based on design galleries [MAB$^+$97], thumbnail renderings [KG02], or spreadsheet representations [JKM01], to semi-automatic transfer function generation. Among others the latter class of concepts include automatic detection of isosurfaces [BPS97], data model driven identification of object boundaries [KD98], or user-guided generation of transfer functions based on genetic algorithms [HHKP96] or machine learning [TLM05].

Especially in the context of medical application, classification is tightly coupled to the area of volume segmentation; dealing with the task to explicitly identify and separate objects such as organs, blood vessels, or neoplasms. Subsequently, optical properties can be assigned according to the object classification. However, this approach is not further detailed in the scope of this work.

Multi-Dimensional Transfer Functions In contrast to physically based rendering, volume visualization commonly applies the assumption that the physical properties of emission and absorption of the participating media are conceptually independent of a ray's direction. As a consequence, transfer functions in their simplest form are in fact one-dimensional. However, classification of data solely based on the scalar input often fails to accurately capture surface boundaries and tends to severe aliasing artefacts in the vicinity of sharp features. In the context of isosurface rendering Levoy [Lev88] devised a specifically designed two-dimensional mapping function based on the input scalar value and the gradient magnitude to lessen the impact of aliasing. Following this basic idea, various new models for transfer functions in multi-dimensional spaces were proposed to enable the distinct classification of objects and their boundaries [KD98, KKH01, ŠBSG06].

Pre-Integrated Transfer Functions Besides the direct influence of the appearance of the final rendering result, transfer functions also have a direct impact on the quality of the reconstruction. This correlation is due to the dependency of the required sampling rate—according to the sampling theorem—for the correct evaluation of the volume rendering integral from both, the input data field and the user specified mapping function. In detail, it can be shown that the required sampling distance is bound by the product of the maximum Nyquist frequencies of the input data field and the transfer function (see Section 2.4.3 in [Kra03]). An alternative, less restrictive estimate is given by Bergner et al. [BMWM06]. They show that the upper limit of the sampling frequency is proportional to the product of the maximum gradient magnitude of the input data and the maximum Nyquist frequency of the transfer function. For high frequency mapping functions, however, both estimates imply the necessity of a very fine grained sampling; either leading to visual artefacts in case of undersampling, or severely increasing the rendering costs.

To overcome this issue, *pre-integration* evaluates the volume rendering integral, for a piecewise linear, continuous scalar function for pairs of sample values and a set of sample distances prior to the actual ray traversal. During rendering, this pre-computed table is used to estimate the contribution of the volume rendering equation between two consecutive data samples. In effect, the sampling of the data and the sampling of the transfer function is separated and, thus, the Nyquist frequency of the transfer function is no longer a factor for optimal sampling estimation. However, this only comes at costs of increased storage overhead for the pre-computed lookup table; in particular, for high data quantization, i.e. short or floating point input, calculating and storing a full table becomes infeasible.

Originating from the field of rendering tetrahedral meshes, *pre-integration* was first mentioned by Max et al. [MHC90] for the special case of transfer functions—and the products of color and alpha functions (cp. equation (3.8))—being limited to integral functions. Stein et al. [SBM94] and Williams et al. [WMS98] extended the technique to linearly varying absorption coefficients and piecewise linear transfer functions, respectively. The application to arbitrary transfer functions was first presented by Röttger, Kraus, and Ertl [RKE00] and was later on applied to GPU-accelerated slice-based volume rendering by Engel, Kraus, and Ertl [EKE01].

As changing the transfer function requires large sections of the pre-computed lookup table to be updated, fast evaluation of the *pre-integration* table is crucial for interactive exploration of volumetric data. Several acceleration techniques were proposed: Röttger and Ertl [RE02] parallelize computation to be executed on the GPU. Weiler et al. [WKME03] use an incremental approximation for larger sampling distances based upon a previously calculated table in order to speed up creation of three-dimensional lookup tables. Lum et al. [LWM04] further optimizes this strategy to be also applicable to a single table of constant sampling distance.

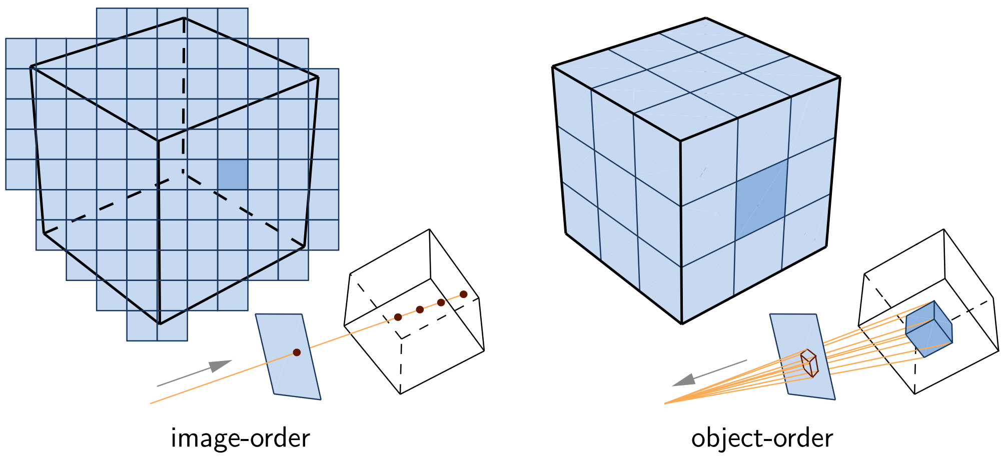


Figure 3.8: Schematic comparison of the two major volume rendering paradigms: *Image-order* techniques determine the volume's contribution on a per pixel basis, thus, calculation is aligned to the view plane. In contrast, *object-order* techniques project each volume element onto the image plane and accumulate their contributions.

3.2.3 Rendering

The pursuit of an efficient evaluation of the volume rendering integral spurred the development of various rendering techniques for volume visualization. Closely coupled to the available compute platforms, the algorithms are often geared towards specific hardware capabilities or are initially driven by the availability of hardware features. While a great variety of techniques and optimization strategies are published, only a few of the most fundamental concepts are discussed here; in general, these methods are grouped into two categories (see Figure 3.8):

Image-order techniques operate on a per-pixel basis and evaluate for each image element the corresponding light transport through the volume. Being the most prominent example of this class, *ray casting* [Lev88] directly implements this strategy by shooting rays from the eye through each image element on the view plane and following their trace through the volume. Along each ray the volume data is sampled, classified, and each sample's contribution to the volume rendering integral is accumulated according to the compositing equation (3.16), or (3.15) respectively. Due to its inherent parallelism—each ray can be handled independently—target platforms range from vector processors, to multi-core CPUs, to specialized SIMD volume rendering hardware [MKS98, PHK+99, RS00], to graphics hardware. Section 3.3 details on ray casting for GPUs.

In contrast of operating on a per-pixel basis, *object-order* techniques process each volume element, i.e. cells of a three-dimensional mesh or sample of a point

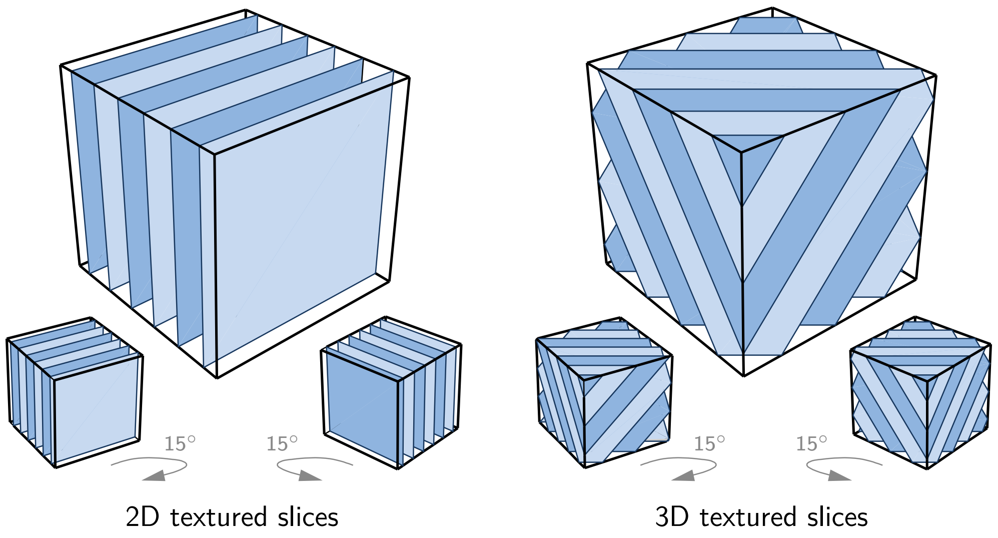


Figure 3.9: Slice based volume rendering paradigms: Volume aligned slices (left) allow to resort to bilinear texture mapping at costs of memory overhead and slice stack flipping during rotation. View aligned slices (right) overcome this issues provided that tri-linear texture mapping is available.

cloud, and determine its contribution to the final image by projecting the element onto the image plane; in this process the projection of a single volume element may contribute to an arbitrary number of pixels. Via compositing the projected footprints are combined in a depth sorted manner with respect to the volume element's location in camera space to form the final image.

Initially proposed for tetrahedral meshes—more generally, any mesh that can be decomposed into such a structure—*cell projection* [ST90] efficiently approximates the projection of volume cells by sets of semi-transparent polygons that mimics the cell's effects on light transport in image space; by doing so, the technique can benefit from hardware rasterization, which led to a variety of graphics hardware accelerated implementations for various mesh types [RKE00, WMFC02].

Splatting [Wes90] is an alternative way of estimating the projected contribution of a volume element by pre-computing two-dimensional footprints—called splats—of a three-dimensional reconstruction kernel. Each volume element is represented by a distinct splat in image space that are again combined by compositing for obtaining the final rendering result.

Driven by graphics hardware's capability for texture mapping, slice based volume rendering techniques [CN93, CCF94] were proposed to efficiently embed the evaluation of the volume rendering integral into the graphics pipeline. The volume is represented in object space by semi-transparent textured proxy geometry that are iteratively projected and composited using hardware acceleration.

Initially limited to bilinear texture mapping only, slicing was restricted to stacks of volume aligned planes oriented perpendicular to the volume's axis with the smallest included angle to the view direction; as a consequence, memory requirements are tripled due to the requirement of maintaining all three slice stacks covering the main axis. In addition, switching between the stacks leads to an abrupt change in sampling locations and, thus, may result in visible popping artefacts during rotation (see Figure 3.9 left).

With the availability of tri-linear texture mapping, view aligned slices became the standard for slice based volume rendering—effectively avoiding the issues of object aligned slices (see Figure 3.9 right). However, calculation of the proxy geometry—arbitrary cross-sections of a plane with a cube—become more costly and error-prone. Section 5.3.3 details on efficient ways to obtain the proxy geometry.

3.3 Single-Pass Volume Ray Casting

The techniques of slice based volume rendering efficiently map all performance critical tasks, i.e. the data sampling, the projection of the proxy geometry, as well as the compositing, to the rendering pipeline and, thus, utilize graphics hardware quite effectively; a property also reflected by experimental results that indicate slice based volume rendering on graphics hardware—simply mapped to the (virtual) fixed function pipeline—to be texture bound, i.e. limited by the hardware's performance in fetching data from the volumetric scalar field. In this sense, such simple slice based techniques are optimal in terms of performance. However, the implementation via a static stack of plain proxy geometry entails several disadvantages in terms of algorithmic flexibility:

In connection with projective projection, the actual sampling interval is not equivalent to the distance between two consecutive slices, but is also dependent on the included angle between the slice normals and the view direction. As a consequence, sampling quality varies in image space with degrading sample frequency from perpendicular to angular ray directions; solutions proposed to alleviate possible artifacts include *alpha correction*, three-dimensional pre-integration tables, or *shell rendering* [LHJ99]; however, they all introduce additional implementation and/or computational overhead.

In terms of flexibility, the mapping of the view rays to a set of fixed proxy geometry rules out any possibility to change the direction of light transport during light propagation, e.g. as it is common for refractive materials; thus, posing restrictions on the applicable optical models.

More importantly, this limited flexibility in algorithmic control of the volume sampling also impede optimizations targeted to reduce the overall texture load like for example *early ray termination* [Lev90] or *empty space skipping* [GR90, YS93]; especially, in respect of being texture bound, such optimizations are, however, crucial for further increasing the overall performance. Hence, various successful solutions were proposed to include such optimization mechanisms to slice based volume rendering; section 4.3.1 actually discuss such an approach in more detail. However, the simplicity and elegance of the original optimization strategy is often lost during the integration to a slice based renderer and commonly also requires considerable implementation effort.

Besides ray traversal and data sampling, also the remaining subtask of compositing is limited in terms of flexibility. As the slices are blended via raster operations, the possible compositing operations are restricted to those defined by the graphics pipeline. While the exposed operations for blending suffice to setup the compositing equations derived from the volume rendering integral or simple models like *maximum intensity projection* (*MIP*), the limited set of operations may hinder other techniques. For example, *local maximum intensity projection* (*LMIP*) [SSN$^+$98] or compositing according to the Kubelka-Munk model (see Section 3.3.3) cannot be expressed by the fixed raster operations' functionality.

Finally, another algorithmic disadvantage arises from the independent processing of consecutive volume samples along a ray. Triggered by issuing draw calls for distinct slices of the proxy geometry, the graphics pipeline does not allow for any communication between the processing of samples, in particular, along a single ray. As a consequence, classification or optical models, that are based on the sample history along a ray, must resort to costly re-computation of the necessary history of intermediate results for each new sample location. An example for such a repetitive computation is *pre-integration*: each new sample requires the data value of the previous one to fully define a slab's contribution to the integral; an slice based implementation, thus, needs to fetch each sample twice—a critical overhead for texture bandwidth, however, efficiently dampened by texture caches.

To overcome all mentioned issues, research focused on mapping *image-order* techniques, more specifically *ray casting*, to the graphics pipeline. In the same year Krüger and Westermann [KW03a] as well as Röttger and colleagues [RGW$^+$03], both proposed a similar multi-pass algorithm to implement *ray casting* on GPUs. Instead of aligning the proxy geometry with the actual sampling locations, their algorithms repeatedly render the front faces of the volume's bounding geometry while keeping track of the true integration positions as well as the accumulated color and opacity in intermediate buffers. For each new iteration, the intermediate buffers—storing all information that need to be passed from one sample to the next along the rays—get sampled according to the ray's screen space position, the new sample location is derived from this data, and results are written back to the data buffers after the ray integration. While such approaches effectively address all previously mentioned issues, they introduce additional texture costs for passing the data in between the samples and, thus, cannot fully achieve the performance of plain slice based volume rendering.

With the advance of programmable graphics hardware—in particular, with the support for dynamic flow control—a direct mapping of *volume ray casting* to graphics processors became possible that combines the performance of slice based approaches with the flexibility of casting rays in a single render pass. This work was published first in cooperation with Simon Stegmaier and Thomas Klein [SSKE05].

3.3.1 Basic Principle

Instead of mapping the tasks of ray traversal, data sampling, and integration to different stages of the rendering pipeline* increased programmability of GPUs enables to implement the complete evaluation of the volume rendering integral for a single ray in one instance of a fragment shader. Besides offering full programmability for all volume render stages, additional overhead for passing data—in particular additional texture fetches—between a ray's samples is effectively avoided.

Render Setup In order to generate the necessary fragments—each fragment equals a single ray—some proxy geometry needs to be rendered. For the rendering of a cuboid volume, this basically reduces to draw the front faces of the cuboid to generate a single ray per pixel. Conceptually equivalent to this approach is the usage of the back faces—in fact the identical set of fragments are generated—with the only difference that the rays are then traversed backwards. However, as the latter reverses the original notion of shooting rays from the camera into the scene, the following description follows the former approach.

In addition to the initialization of a matching set of fragments, the choice for rasterization of the front faces also enables to setup the initial start location for ray traversal and subsequently the ray direction. To provide the fragment shader with the necessary information, each vertex is attached with its object space location as texture coordinate. By doing so, polygon rasterization generates correctly interpolated object space positions for each fragment—resembling the ray's intersection with the front faces. In relation to the camera position the actual ray direction can be obtained by subtraction in the shader. Without loss of generality, the further discussion assumes the object coordinates of one of the bounding box vertices to be zero, whereas the opposing vertex is defined by the geometric

*For slicing and multi-pass ray casting ray traversal can be thought of being triggered by vertex processing, data sampling by fragment processing, and compositing—at least in case of slicing—by raster operations.

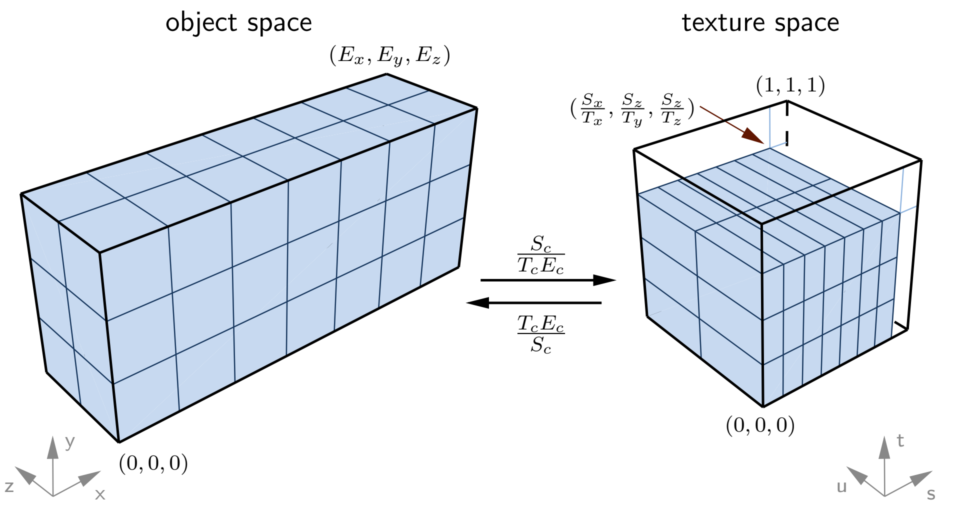


Figure 3.10: Translation between spaces for operating on a power-of-two texture.

volume extent E_c with $c \in \{x, y, z\}$, i.e. usually given by a combination of input data slices S_c times the slice distance D_c for each axis.

After the actual ray traversal—discussed in the next paragraph—the resulting color and opacity of volume rendering can simply be blended with the background image or rendering by raster operations.

Compared to slice based volume rendering and also multi-pass ray casting, this approach offers the additional benefit of posing fixed, minimal costs for vertex processing, rasterization, and raster operations. While for both alternative approaches these costs scale with the number of volume samples along a ray, single-pass volume ray casting only requires the front faces to be processed by these stages of the rendering pipeline exactly once. In particular in the context of modern GPU's unified shader architecture this characteristic might prove favorably in terms of efficient hardware utilization.

Shader Setup The programmable fragment shader stage of the rendering pipeline basically resembles a ray casting implementation analogous to a CPU variant. The main difference results from data sampling, which is consequently handled by texture fetches for a GPU implementation. Hence, the actual coordinates for sampling the input scalar field are required to be translated to texture space. Basically, this leads to two design options: First, the ray casting operates completely in object space and sample locations are solely transformed from object space to texture space for fetching input data. Second, the complete ray traversal is performed in texture space after the object space input data, i.e. the start positions and ray directions, are translated accordingly. From an algorithmic point of view both methods are identical, however, performance may vary significantly depending on the applied optical model.

The reason for this is founded in the transformation between object and texture space that is given in the most general case as affine map, i.e. a scaling combined with a translation. With the above stated pre-assumption on the object space coordinates, the transformation reduce to a scaling with the factor F_c, given as

$$F_c = \frac{S_c}{T_c} \frac{1}{E_c}$$

whereas T_c denotes the true dimensions of the volume texture—which might differ from S_c in case GPU capabilities are limited to power-of-two textures (see Figure 3.10). If non-power-of-two textures are supported the first term of S_c/T_c always equals one and, thus, can be neglected. In the general case, this transformation does not respect length or angles; as a consequence, geometric operations like the normalization of the ray direction or a vector dot product for lighting must be performed in object space. While not posing an additional costs for ray casting in object space, it does add an overhead for the variant working in texture space, as ray directions or gradient need to be translated back to object space.

Depending on the optical model the implementation variant requiring less transformations per ray iteration proofs to be favorable. For example, using the volume rendering integral as optical model (as described in Section 3.1.1) favours the second variant working in texture space—as it is described in the following—since it does not require any transformations per sampling step; opposed to the single mapping of the sampling position from object to texture space per iteration in the alternative version. However, for other optical models, especially when accounting for refractive media, object space ray casting is advantageous.

Fragment Shader The basic framework of ray casting implemented in a single fragment shader is described on the basis of a front-to-back, pre-integrated volume renderer written in GLSL that uses the Equation (3.16) for compositing. Via textures and uniform variables the input scalar field, the two-dimensional pre-integration table, and the user/volume parameters are passed to the shader:

```
uniform sampler3D VOLUME;
uniform sampler2D TRANSFERFUNCTION;
uniform vec3 geomCameraPos;
uniform vec3 scaleFactors;
uniform float samplingDist;

void main(void) {
```

In order to operate in texture space, the input data given by the rasterization, i.e. the ray starting locations, need to be transformed from object space to texture space. Correspondingly, the ray direction needs to be defined. Beside the pure mapping between the spaces, applying this transformation to the ray direction also ensures a constant uniform sampling distance with respect to voxel dimensions:

```
    vec3 geomPos = gl_TexCoord[0].xyz;
    vec3 pos = geomPos*scaleFactors;
    // Compute the scaled ray direction
    vec3 geomDir = samplingDist*normalize(geomPos - geomCameraPos);
    vec3 dir = geomDir*scaleFactors;
```

After the ray setup, the first sample is fetched in preparation of the pre-integrated ray integration. In case another classification method is applied, the special handling of the first sample may become obsolete:

```
    vec3 scalarOld = texture3D(VOLUME, pos).r;
    vec3 dst = vec4(0.0);
    vec3 src = vec4(0.0);
    pos += dir;
```

The actual ray traversal, including the data sampling, classification, as well as compositing, is iteratively performed until the ray leaves the volume. By simply storing the current sample in a register for further usage in the next pre-integration step, data can be communicated between the samples along a ray; efficiently avoiding recalculation or repetitive sampling.

The order of operations is arranged in a way to optimize performance, i.e. to allow for a maximum of independent arithmetic operations after a texture fetch to hide texture latencies. This is achieved, by interleaving the data sampling and the compositing operation, whereas the compositing is delayed by one iteration. In comparison to the classical order of sampling, classification, and compositing, this optimized approach allows for slightly increased performance of 5%–10% on average (benchmarked on a NVIDIA 9800GT):

```
    do {
        // Lookup new scalar value
        vec3 scalar = texture3D(VOLUME, pos).r;
        // Move one step forward
        pos += dir;
        // Perform blending
        dst += (1.0 - dst.a)*src;
        // Lookup color in pre-integration texture
        src = texture2D(TRANSFERFUNCTION, vec2(scalar, scalarOld));
        // Save current sample for pre-integration in next iteration
        scalarOld = scalar;
    } while((clamp(pos, vec3(0.0), vec3(1.0)) - pos) == vec3(0.0));
    gl_FragColor = dst + (1.0 - dst.a)*src;
}
```

The loop termination is based on the assumption that the volume completely covers the texture, i.e. valid coordinates span the range $[0 \ldots 1]$; however, this is only guaranteed for non-power-of-two textures. In case of power-of-two-textures the position is additionally required to be multiplied by a factor of T_c/S_c for the loop test to compensate for the void texture regions; apart from this minor change the same efficient loop test[†] can be applied.

Evaluation In order to evaluate single-pass ray casting in terms of practical considerations, a direct comparison of this technique against a screen aligned slice based approach is discussed. Both methods implement pre-integrated volume rendering using the identical two-dimensional classification table without any further optimizations with respect to quality or performance. In this way, the two variants can be considered as fundamental implementation of their respective class. Evaluation of this comparison focuses mainly on key differences in categories of quality and performance.

From a quality perspective, ray casting offers several algorithmic benefits for enabling superior image quality: First, the constant sampling distance along a ray. And second, the calculations are all performed on a full 32bit floating point precision. Both characteristics have a direct impact on image quality, however, their true effect on visually perceivable differences are highly dependent on sampling distances, data quantisation, and frequency distribution of the scalar field as well as the transfer function. Besides the challenge of objectively measuring the visual differences, such comparisons are inherently limited to draw conclusions based on two specific implementations rather than comparing the actual techniques; in particular, as a variety of methods exists to address the algorithmic disadvantages of slice-based methods. For the special case of the two exemplary implementations using no further optimizations, this clearly leads to a higher image quality for ray casting due to the less accurate reconstruction of slice based rendering (see Figure 3.11 bottom row). However, a slice based version of shell rendering—data samples are warped into the shell arrangement in the shader, while still using plane slices as proxy geometry—in combination with 32bit compositing using off-screen buffers, effectively leads to an identical evaluation of the volume rendering integral as the described ray casting approach. Thus, for practical considerations the assessment of quality is tightly coupled with the development and computational effort required to achieve sufficient—for comparison purposes, identical—results. In this sense, ray casting seems superior to slice based approaches, as an image

[†]Efficiency of this specific test arise from the mapping of the clamp instruction to the low-level saturate instruction supported by most modern GPUs. However, as the compile process is dependent on various factors, i.e. driver versions and hardware capabilities, this most efficient mapping cannot be always guaranteed from a high level perspective, like exposed by GLSL.

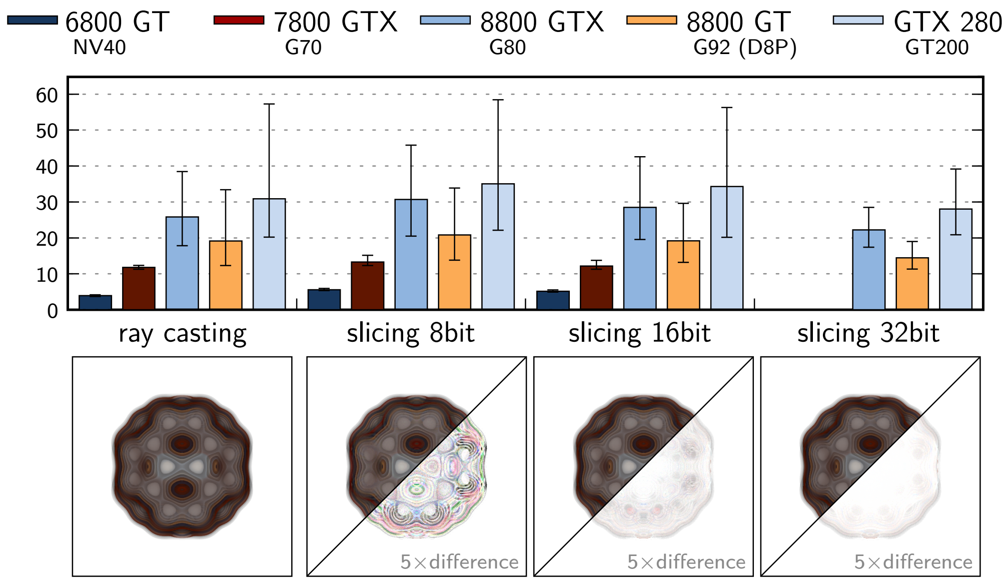


Figure 3.11: Performance comparison of ray casting and slice based rendering with increasing compositing precision. Representing each hardware generation capable of ray casting, benchmarks were performed on five NVIDIA cards. Results for rendering the 512^3 data set[‡] on a 512^2 viewport, sampling at least once per voxel, are given as average frames per second for a full rotation around the y-axis, including the range of render times as error bars. For visual comparison, the rendered images as well as the scaled color differences with respect to ray casting are shown in the bottom row.

quality is achieved that is only matchable by additional development as well as computational overhead on the side of slice based techniques.

From a performance perspective, ray casting nowadays shows competitive results against the slice based variant. Exposing only half the performance of slice based methods on the first capable hardware generation—as also reported in the original publication using a NV40 based card [SSKE05]—more recent hardware generations, such as the GT200 architecture, feature significantly less overhead for dynamic flow control and, thus, expose comparable performance for both techniques (cp. benchmarks given in Figure 3.11). This match in rendering speed ultimately reflects the similar algorithmic complexity of the two implementation variants, especially in terms of data fetches. Nevertheless, the benchmarks still show ray casting being slightly slower compared to slicing in case of 8bit and 16bit compositing. The reasons for this discrepancy are twofold: First, ray casting per-

[‡]The original 128^3 buckyball data set approximates the electron density inside a C-60 Buckminsterfullerene molecule. The data is courtesy of O. Kreylos, UC Davis.

forms slightly more texture fetches than the slice based approach due to constant sampling intervals; for the benchmark setup, the number of data samples differs on average by a factor of two percent. And second, compositing for ray casting is not executed on fixed function hardware, i.e. the raster operations, that run effectively in parallel to the fragment processing. While being the key to increased flexibility, that poses a small performance penalty. However, in case of using 32bit floating point accuracy for compositing results change. Full floating point blending currently poses a high computational overhead for raster operations. As a consequence, the bottleneck for slice based techniques shifts from being texture bound to be raster bound, which in turn leads to a sever drop in performance for the maximum achievable frame rate (cp. Figure 3.11 right). In contrast, ray casting does not depend on the on the compositing performance of the raster operations and, thus, outperforms slice based rendering.

Besides the categories of quality and performance, the already mentioned algorithmic flexibility of ray casting poses the major advantage in usability of ray casting. Practical applications for utilizing this advantage are given throughout the next two sections, including early ray termination, empty space leaping (both Section 3.3.2), continuous refraction, and a more elaborate spectral optical model (both Section 3.3.3). Further examples of this flexibility covering high-quality reconstruction of implicit surfaces, self-shadowing, image-based lighting, combinations of iso-surfaces with semi-transparent volume rendering, or clipping are discussed in the context of the original publication [SSKE05].

3.3.2 Acceleration Techniques

In the context of volume rendering being texture bound in the general case, the most promising acceleration strategies consequently focus on reducing the number of texture lookups. While data fetches—in particular, the tri-linear interpolation—were already the critical performance bottleneck in the earliest days of volume rendering using CPU based ray casting[§], several optimization techniques based on the paradigm of shooting rays were proposed early on. However, their implementation on graphics hardware via slice based algorithms are still tedious and expensive—even rendered impossible for some hardware generations. In the following two such techniques are discussed in the context of single-pass volume ray casting and compared to their possible slice based counterparts:

Early Ray Termination Being initially called adaptive ray termination [Lev90] or α-termination [DH92], the basic idea of this acceleration technique is to stop

[§]An indicator, that data interpolation is still a performance critical task for x86 processors nowadays, is given by Intel's Larrabee architecture [SCS$^+$08], that solely implements texture fetches as fixed functionality besides an otherwise completely software based graphics pipeline.

traversing prior to leaving the volume, in case further samples do not significantly contribute to the final composited color anymore; for front-to-back rendering (3.16) this criterion boils down to a sufficiently high accumulated alpha component $\alpha_{[n..i]}$, since all further ray traversal iterations at most contribute with a remaining factor of $1 - \alpha_{[n..i]}$. Efficiency as well as the introduced error are highly dependent on the chosen α-limit, the data set, and the applied transfer function—being most suitable for renderings exposing large volume areas of high data opacity.

In accordance to the initial implementation [Lev90], realization for single-pass ray casting simply results in a changed loop test for ray traversal—maintaining the acceleration techniques' simplicity and elegance:

```
    } while((dst.a < ertLimit) &&
            ((clamp(pos, vec3(0.0), vec3(1.0)) - pos) == vec3(0.0)));
```

For the benchmark setup described in Figure 3.11 early ray termination was measured on a NVIDIA GTX280. Adding the optimization with a limit of $\alpha_{[n..i]} < 0.95$ results in an overall increase of performance by a factor of 1.89—with a simultaneous reduction of required data fetches of 43.4%. Clearly showing the direct relation between performance and data samples. Detailed analysis of the results also indicate the expected small performance overhead for adding this acceleration technique to single-pass ray casting (see Section 7.2.2).

In contrast, embedding early ray termination into slice based volume rendering requires substantially more development overhead and exposes a severely higher computational costs. The most efficient method is based on the *early depth test* to avoid the fragment shader execution for already terminated rays [KW03a]. In an intermediate pass the ray termination condition is evaluated for each ray and the pixels' depth is set accordingly, i.e. set to zero in camera space in case ray traversal should be terminated. Fully rasterizing the front faces and switching the fragment shader for each intermediate step is, however, necessary due to the conceptual limitation of the depth test, allowing for early rejection only if no depth is written in the fragment shader. Even though, the intermediate pass is only writing depth—allowing for faster execution on many GPUs—state switches, increased rasterization load, and most importantly additional texture lookups for querying the accumulated alpha component leads to a less efficient implementation compared against single-pass ray casting.

Empty Space Leaping The second acceleration technique discussed here, exploits the fact that during user interaction spatio-temporal coherence for rays shot through the volume is very high. This coherence is utilized by skipping empty regions using information from the previously rendered frame and directly *leap* to the first ray sampling position that exhibits significant contribution to the

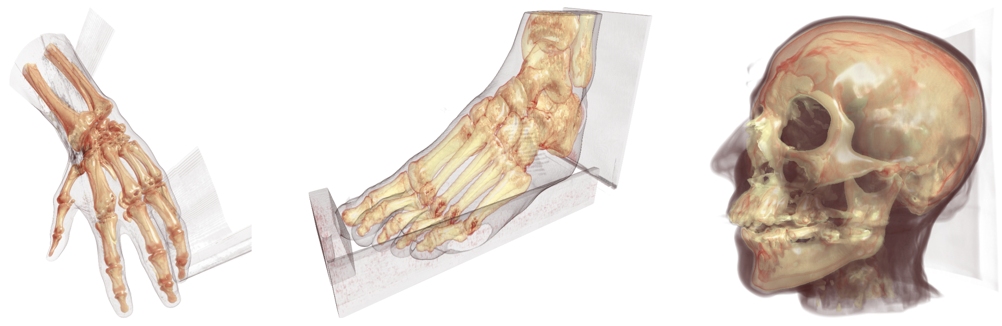


Figure 3.12: Various high-quality volume renderings exposing large empty volume regions in the surroundings of the actual visible object after classification.

final image. Ray traversal is then started at this location, reducing the volume sampling cost significantly. In particular, for iso-surface renderings, volume rendering with large empty regions after classification, or combinations of iso-surface and semi-transparent volume rendering (see Figure 3.12) these technique proofs to be auspicious. Being initially proposed in the early 1990s by Gudmundson and Randén [GR90] and Yagel and Shi [YS93] for CPU based ray casting systems, application of this technique to graphics hardware based rendering were unfeasible and/or impracticable for various early hardware generations. The first method utilizing GPUs for empty space leaping combined depth information obtained by slice-based volume rendering with a CPU based ray casting approach to speed up overall render times [WS01]. A fully hardware based approach was first published in cooperation with Thomas Klein and Simon Stegmaier [KSSE05]—following the original concept proposed by Yagel and Shi.

Following the schematic outline of the algorithm given in Figure 3.13, the accelerated ray casting proceeds as follows: In case no prior knowledge is available or applicable—the former is usual for the first rendered frame, the latter may be caused by user interaction breaking the assumption of coherence—the ray casting system is initialized by starting at the volume's front faces; ultimately, performing ray casting equivalent to the unoptimized variant. However, instead of writing only the final render result to the frame buffer, the location of the first relevant ray sample, in the following referred to as *hit position*, is stored in a second render buffer. Prior to rendering consecutive frames, these hit points are reprojected, i.e. transformed regarding to the new viewing parameters, to obtain a valid estimate for the ray starting locations of the next frame. As reprojection generally fails to provide data for all rays in the next frame due to occlusion (see white areas in projection results of Figure 3.13), those regions are filled with a conservative estimate, i.e. the location of the front faces. In turn, the newly generated hit positions are devised as start locations for ray sampling in the following frame.

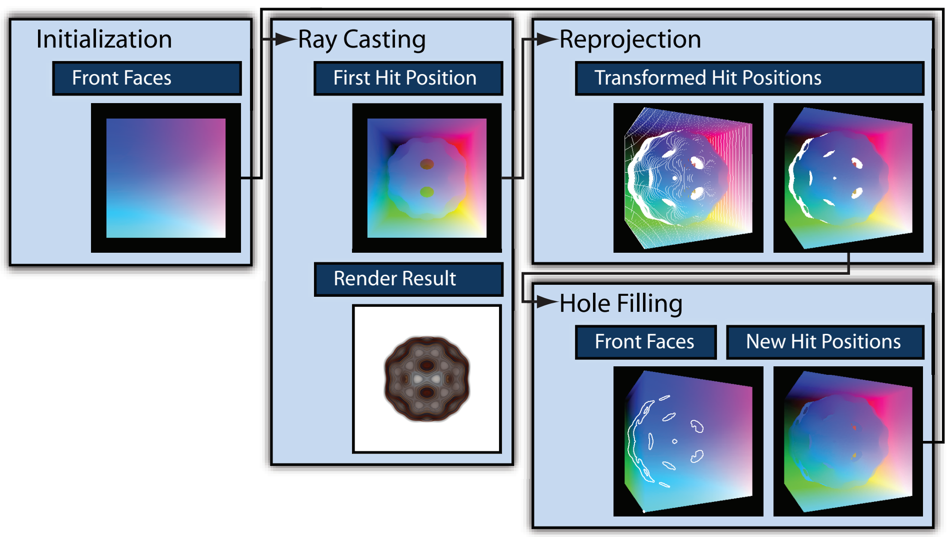


Figure 3.13: Schematic overview of GPU based empty space leaping.

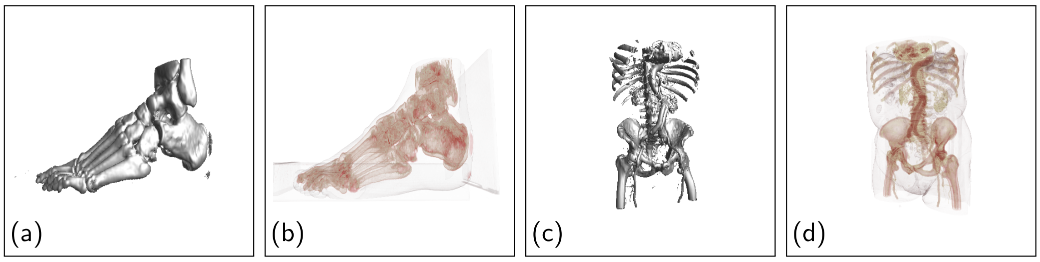
Reprojection Analogous to the *coordinates buffer* introduced by Yagel and Shi, the hit positions are written as object coordinates to the intermediate float buffer—actually including a small offset in the opposite ray direction to account for inaccuracies during reprojection. In this way, it is sufficient to initialize the render setup according to the new view parameters and to feed the positions through the vertex pipeline to compute the new coordinates of their screen space projection, i.e. the origin of the corresponding ray for the subsequent frame. This process is implemented as a separate render pass, again writing the transformed hit positions to a floating point render target.

Several different ways exist to realize this reprojection stage on graphics hardware: First, supported by NVIDIA cards only, a combination of *pixel buffer objects* and *vertex buffer objects* can be used to effectively implement the required *render to vertex array* functionality; yet, a distinct memory copy on the GPU as well as twice the memory footprint are required. Second, the *transform feedback* can be used to feed the data back as vertex input, but such an approach would require the ray casting to be performed completely in the vertex shader—hindering the hole filling process, as discussed shortly. And third, vertex texture lookups in combination with a dummy vertex array can be used to generate the required vertices with correct positions assigned in a vertex shader. Due to the restricted availability of some of the mentioned features on the targeted GeForce 6 based hardware, the latter variant was favored; besides hardware capabilities, even on more recent hardware generations, benchmarks indicate this technique still to be the most effective. With the introduction of *geometry instancing*, further optimization of the actual implementation could be achieved by replacing the dummy vertex array and issuing only a single point element with sufficient instances generated on the fly; however, this optimization is not considered in the following.

Splatting and Hole Filling Although after reprojection most pixels have assigned a valid estimate for starting ray casting the subsequent frame (see left reprojection result in Figure 3.13), two issues remain to be addressed:

The fist one is related to the discretization of the render targets that affects both, the input hit point positions and the transformed output hit points locations; the former is due to rasterization of the front faces during ray casting, the latter is caused by subsequent rasterization of the already discretized input data. As a consequence, the generated estimates in general contains pixels with no valid space-leaping information and in special cases—most likely in areas of high depth discontinuities—estimates may even be defective. Such errors are possible, in case an occluding surface—that is closer to the camera—is not accounted for due to input data discretization or rasterization during the reprojection phase; consequently the resulting estimate is based on a more far, in fact occluded surface. Several solutions for this problem have been proposed, including the use of point- and cell-splatting methods [YS93, WSK02]. Both splatting approaches exploit the fact that projecting points that exhibit a screen space footprint larger than a single pixel can solve the occlusion problem as long as the points are spread densely over the image plane. Depth sorting guarantees that for overlapping points the most conservative estimate, i.e. the point nearest to the camera, is chosen. Realization on graphics hardware is simply achieved by increasing the output point size for the point elements rendered during reprojection; the effect of this splatting approach can be seen by comparing the two projection results in Figure 3.13.

The second issue covers areas in the projected hit positions that were originally occluded in the untransformed input data (see remaining white spots in the right projection results of Figure 3.13). As the hit positions written during ray casting do not contain any valid information about these regions, a valid estimate cannot be derived from reprojection. Instead, the volume's front faces are used as the most conservative estimate for the ray start location of these regions (cp. outlined front face regions in Figure 3.13). A more progressive, but still valid estimate could, however, be achieved by filling the empty areas with the corresponding depth information of the closest border pixel for each distinct, continuous region. By means of pyramid image filters, as introduced in Section 4.1, such an intermediate pass could easily be integrated into the original system to optimize ray start locations—and thus, further reduce data fetches—for these occluded areas.



data set	size	h	no ESL	ESL	speed-up
Foot (a)	160×430×183	0.4	6.7	18.8	2.7
Foot (b)	160×430×183	0.8	9.7	13.2	1.4
Abdomen (c)	512^2×174	0.7	3.9	10.4	2.7
Abdomen (d)	512^2×174	0.7	5.6	6.8	1.2

Figure 3.14: Performance of the single-pass ray casting with and without empty space leaping (ESL). Shown are the average frames per second measured while rotating the data sets¶ around the y-axis. Benchmarks were taken while rendering in a 512^2 viewport using a NVIDIA 7800GTX graphics board. The sampling step size h is given with respect to the shortest edge of the voxels.

Evaluation With the focus on render performance, Figure 3.14 lists some benchmarks comparing render times with and without the empty space leaping technique. As the performance gain is related to the ratio between the saved data fetches opposed to the still required ones, efficiency of the optimization technique is severely higher for iso-surface rendering than their semi-transparent counterparts. For example, in the former setup the Abdomen data set (Fig. 3.14(c)) requires an average of 177 ray iterations to be performed without using empty-space leaping. In contrast, applying the optimization technique reduces the number of data fetches to 25 per ray.

However, parts of the significant reduction of data samples is in return used up by the introduced computational overhead. In particular, the reprojection step—in detail, the rendering of the point primitives—imposes severe costs; as a consequence, the effectiveness of the discussed approach—which may even end up to be negative—is strongly dependent on the data set, the classification, and the cost for sampling the empty regions. While alternative approaches, e.g. like octree based hierarchical methods [HSS+05] or proximity clouds [CS94], can be expected to offer superior performance gains, such techniques, however, often only

¶The used data sets are courtesy of the Department of Radiology, University of Iowa (foot) and Michael Meißner, Viatronix Inc. (abdomen).

come at costs of additional memory and/or several preprocessing costs. In order to improve the effectiveness of the discussed approach for semi-transparent volume rendering, a further render pass could be added to process the last significant ray sample; analogous to the empty space leaping pipeline for skipping regions in front of the visible object, such an approach would enable to skip void regions behind the object as well. In respect to early ray termination, such an extension would only be favorable if high number of rays actually pass through the complete object, which is again highly dependent on the render parameters.

In summary, the two discussed acceleration techniques both clearly indicate the flexibility of single-pass ray casting, especially in controlling the ray sampling. Beyond the shown two examples, other optimization strategies based on adaptive sampling, like for example β-acceleration [DH92], naturally map to graphics hardware via the paradigm of single-pass ray casting. However, it should not be left unnoted that some techniques—especially designed in the context of slice based volume rendering—cannot be embedded into ray casting efficiently; the most prominent example of this class of algorithms is the evaluation of volumetric light attenuation effects, e.g. volumetric shadows or translucency, via half-angle slices [KPHE02].

3.3.3 Spectral Volume Ray Casting

Although, the traditional volume rendering integral allows to create renderings that are suitable for the analysis of typical volume data sets in a large number of applications, the underlying emission-absorption model does not incorporate important optical properties that are typically observed for semi-transparent materials, such as glass or water. Among others, such effects include reflection, refraction, optical dispersion, or selective absorption. Thus, spectral color models received increased attention in computer graphics, and volume rendering respectively, in order to more accurately represent the true physical behavior of transparent media [NvdVS00, BMDF02, BMTD05]. Based on the work of Abdul-Rahman and Chen [ARC05], who employed the Kubelka-Munk optical model for CPU based volume ray casting, an extended GPU based variant of this technique utilizing the single-pass ray casting paradigm was published as part of this thesis. The work was carried out in cooperation with Thomas Klein, Ralf Botchen, and Simon Stegmaier.

While Section 3.1.2 details on the underlying Kubelka-Munk theory, Figure 3.15 motivates the application of this specific optical model to volume rendering more descriptively: Real-world semi-transparent materials, like for example colored liquids, glass, or foils, absorb light in a wavelength dependent manner—commonly referred to as selective absorption or spectral absorption. Received visible colors for such materials are, thus, a result of the fraction of wavelength of the incident

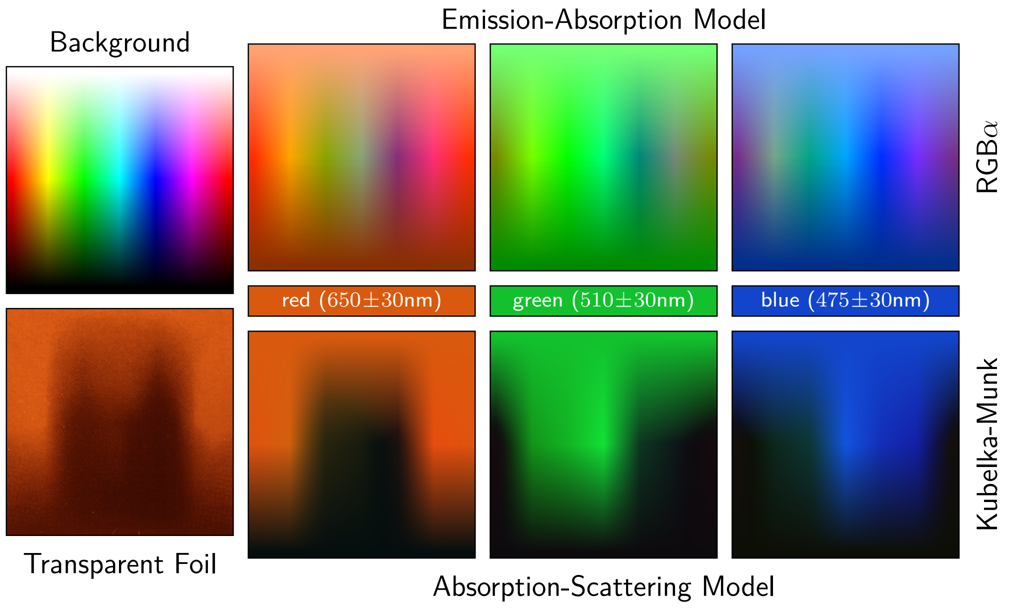


Figure 3.15: Comparison of the RGBα compositing model versus the Kubelka-Munk model. The test setup simulates a transparent layer of medium, which filters all light except the given wavelength, in front of a lid color spectrum (upper left). For comparison the photograph on the lower left side shows the experimentally obtained result for the 650nm case.

light that is not absorbed by the object. An example of such an effect is shown in the bottom, left image of Figure 3.15, that shows an actual scan of the specified background image seen through a transparent foil that completely absorbs all wavelength others than the red spectrum.

The emission-absorption model is commonly based on a constant absorption across the color spectrum, while the notion of colors is introduced by the emission term—theoretically complying with the analogon of a glowing gas of particles. As a consequence, visualizations based on the volume rendering integral, i.e. more specifically on the RGBα compositing model, fail to reproduce this spectral optical effect. Instead, the absorbed radiant energy is replaced with the emitted radiance, leading to a blending effect with the specified color; still exposing nearly the complete color spectrum in the final result (see Figure 3.15 upper row).

In contrast, the Kubelka-Munk model is based on wavelength dependent optical properties, such as scattering and absorption; ultimately, enabling to accurately reproduce the real-world effects of spectral absorption (as shown in Figure 3.15 lower row). In particular, the resulting images solely feature wavelengths according to the colored, semi-transparent material layer, while otherwise completely blocking the inverted color spectrum—mimicking the spectral filtering property of the used colored foil.

Spectral Absorption To include spectral effects into single-pass volume ray casting, the optical properties of the media need to be specified across the visible color spectrum, i.e. the process of classification is required to assign spectral properties to the given input scalar function. In detail, for each scalar value of the volume a corresponding set of spectral absorption and scattering coefficients S_ν and K_ν (cp. Equation (3.18)) has to be defined. To balance sufficient sampling of the color spectrum against render complexity, the number of independent wavelengths and their distribution in the spectrum need to be defined. While any arbitrary mapping can be used in general, the complexity of specifying the transfer function severely increases with the number of considered wavelengths as well. Thus, a common approach to bypass manually specifying the optical properties for each wavelength is to map ranges of scalar values to distinct materials for which the required properties are known in optics (for detailed information on measured spectral properties see [Gla94]). Alternatively, Bergner et al. [BMDF02, BMTD05] propose a specifically designed transfer function widget for spectral transfer functions—targeted for a different optical model.

The overall process of ray casting remains unchanged, with the only difference that instead of accumulating color and opacity values, for each considered wavelength ν the accumulated reflectance R_ν and transmittance T_ν is tracked along the ray. Thus, the classification and the volume integration based on alpha compositing is replaced by: First, a lookup of the Kubelka-Munk coefficients S_ν and K_ν for the sampled scalar value according to the spectral transfer function. Second, the evaluation of the reflectance and transmittance for the corresponding volume layer, i.e. volume slab of thickness χ, according to the Equations (3.20). And third, the Kubelka-Munk composition as defined by the Equations (3.21).

Although, an implementation of Equations (3.20) is possible in the fragment shader, with respect to performance such an approach proves not to be very efficient due to the computational complexity—shader execution maps among others to the costly evaluation of six exponential functions and five divisions. Fortunately, in case of ray casting the thickness of each volume slab is constant due to the uniform ray sampling, allowing to pre-compute Equations (3.20) for a finite set of scalar values and wavelengths. While this optimization follows the typical paradigm of balancing performance/memory versus accuracy, a feasible trade off in this case is given by storing the data in a 16bit floating point texture—as both functions are bound to the range of $[0 \ldots 1]$.

After the ray traversal, passing through the volume in a forward direction, the reflectance properties of the background layer needs to be taken into account to evaluate the initial light flux in the opposite direction. As discussed in Section 3.1.2, the background is assumed to be an opaque material layer reflecting the incoming light of the forward directed flux according the its spectral reflectance map R_ν^{bkg}. For compatibility to non-spectral rendering, this map may be derived from an RGB image as described by Glassner [Gla89]. Defining a spectrum for an RGB color triplet is highly ambivalent and in combination of the discrete sampling of the color spectrum the input colors can not be guaranteed to be reproduced perfectly; yet, the approximation is sufficiently good for this case.

The overall spectral reflectance map R^{img} of the rendered image for each wavelength ν is derived by a final compositing of the accumulated reflectance $R_{[n\ldots0]}$ and the background's reflectance map R^{bkg}; according to Equation (3.21) as

$$R^{\mathrm{img}} = R_{[n\ldots0]} + \frac{T^2_{[n\ldots0]} R^{\mathrm{bkg}}}{1 - R_{[n\ldots0]} R^{\mathrm{bkg}}}.$$

Although it is theoretically possible to perform spectral ray casting for all considered wavelengths ν in a single fragment program, hardware/software limitation hinder this approach. On the one hand, register file size may become an critical issue—effectively leading to costly swapping of temporary registers—on the other hand, execution of such a complex shader may exceed maximum instruction counts or maximum clock cycles per shader instance. Instead, each wavelength is processed independently—or if the optical model permits in small batches—and results are combined using additive blending via raster operations—after the illumination stage discussed next.

Post-Illumination The final step for deriving an image for display from the overall reflectance map is to introduce a light source that acts as radiation source for the forward directed light flux. Given the light source's color spectrum, the overall result equals the remaining spectrum after this incident light passed through the volume, got reflected by the background layer, and again traversed through the volume in the backward direction. By means of the overall spectral reflectance map Ri_ν^{img}, the whole process can simply be described as

$$I_\nu = R_\nu^{\mathrm{img}} L_\nu,$$

whereas L_ν denotes the color spectrum of the light source. In order to mimic natural daylight conditions, the spectrum of the CIE standard illuminant D65 [WS82] may be used for post-illumination. Due to the application of the lighting conditions after the actual ray traversal, the color spectrum of the used light source, however, can also efficiently be used for visual data exploration: A volume may expose completely different data structures under varying lighting conditions according to the volume's spectral absorption and reflection properties. As long as view parameters do not change, exploration is solely based on the reflectance map and, thus, can be performed interactively (see [ARC05] for more details).

Finally, the spectral image has to be converted to RGB color space for display. This is accomplished by converting the color spectrum I_ν to the XYZ color space using the tristimulus color matching functions, $\bar{x}(\nu), \bar{y}(\nu), \bar{z}(\nu)$ defined for the CIE 1964 standard colorimetric observer [WS82] by

$$I_{\mathrm{XYZ}} = \sum_\nu I_\nu(\bar{x}(\nu), \bar{y}(\nu), \bar{z}(\nu))$$

and transforming these colors to RGB space, according to Glassner [Gla89], by

$$I_{\mathrm{RGB}} = I_{\mathrm{XYZ}} \begin{bmatrix} 1.967 & -0.5388 & 0.064 \\ -0.548 & 1.938 & -0.027 \\ -0.297 & -0.130 & 0.982 \end{bmatrix}.$$

Spectral Dispersion Based on the analogon that the volume is composed of colored liquids or glass, another important optical effect closely coupled to such materials is refraction. Initial efforts were primarily focused on embedding refraction into ray tracing of semi-transparent polygonal surfaces, e.g. Kay and Greenberg [KG79b] or Whitted [Whi80]; in the context of volume rendering, Rodgman and Chen [RC01] introduced continuous refraction for discrete ray casting based on the optical mode of the volume rendering integral.

Although the Kubelka-Munk model assumes that the entire volumetric domain of colorant has the same refractive index, it does not prevent the introduction of refraction in an approximative manner: Under the assumption that two material layers feature different refraction coefficients, changing the direction of a flux accordingly between the two layers is more accurate than no directional change of the flux. Effectively, such an enhanced model is in no way less accurate than introducing refraction in the traditional volume rendering integral, which is is based on the Lambert-Bouguer law [WS82]—also assuming the entire volumetric domain to be of the same refractive index. Yet, even breaking the original assumption of a homogeneous material, adding refraction can effectively be used to provide additional visualization clues, like e.g. improved depth perception [RC06].

On an implementation side, adding refraction to a single-pass ray casting is straight forward, as the ray direction is stored in the register file for each shader instance separately and, thus, the direction can simply be adapted according to the law of refraction, also called *Snell's law.* In particular, the termination of ray traversal benefits from the general loop termination criteria used in Section 3.3.1, opposed to the alternative approach to render the back faces in order to efficiently construct a break condition [KW03a], which is only valid for linear rays.

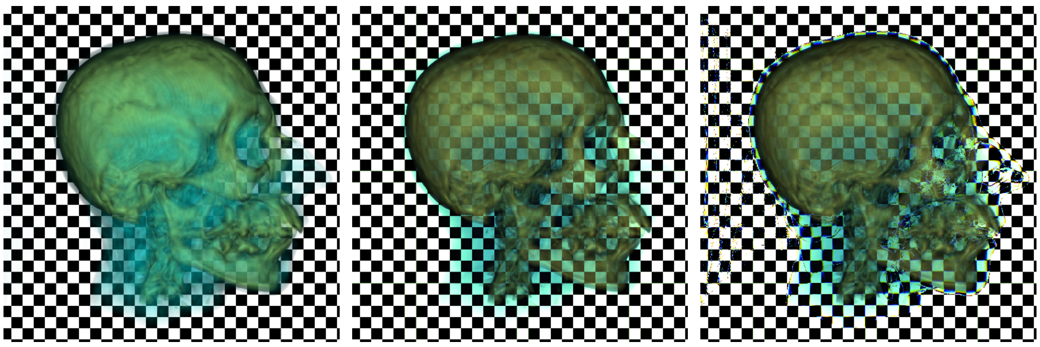


Figure 3.16: Comparison of RGBα volume rendering using pre-integration (left), spectral volume ray casting based on the Kubelka-Munk optical model (middle), and spectral volume rendering including refractive effects like dispersion (right).

With respect to the spectral classification process, the refraction index per material is also specified in a wavelength dependent fashion; this way, the effect of spectral dispersion—that is the separation of a light's spectrum into single-peak chromatic light due to wavelength dependent refraction—can additionally be captured by spectral volume ray casting. A sample application of refractive volume rendering exposing spectral dispersion is given in the following section.

Evaluation Spectral volume rendering based on the Kubelka-Munk theory poses a significant performance overhead compared to the standard RGBα compositing model according to the evaluation of the volume rendering integral. Being still far from interactive frame rates paired with a severely increased complexity of specifying a suitable transfer function, shifts the primary focus of such an elaborate optical model away from interactive data exploration towards more realistic, physically based volume rendering that accurately captures the underlying optical effects caused by light interaction with semi-transparent materials. Effects like spectral absorption, refraction, or dispersion—if applied in a purposeful manner—can add important visualization clues for an observer that help to communicate data content in one static image. An example of such an application is given in the image series of Figure 3.16 by comparing three different optical models under the same view conditions and similar classification parameters. In this setup, the optical model including all the aforementioned effects, tend to provide a more precise and descriptive representation of the data—properly indicating object boundaries as well as volume regions of high transparency. In particular, as the newly introduced optical characteristics mimic true, real-world physical behavior of the used materials, such effects are likely to be conceived easily by a human observer.

Chapter 4

Pyramidal Filtering Techniques

As presented in the previous chapter for the case of direct volume rendering, virtually any visualization pipeline exposes at least some stages that are subject to parallelism. This chapter details on the filtering stage of a visualization pipeline with the primary focus on constructing highly efficient filter kernels suitable to the parallel execution model of modern graphic processing units. All introduced filters are based on two fundamental concepts:

Firstly, all filters are constructed by design to be pyramidal filters as introduced originally by Burt [Bur81] and Mitchell [Mit87]. Due to their inherent logarithmic runtime complexity, pyramidal filters have always been in the focus of research for image processing. Likewise, also GPU-accelerated image processing as well as GPU-based rendering utilized pyramidal techniques already early on in the form of mipmapped textures—a pyramidal multi-resolution representation of pixel data targeted at high quality image minification with low computational overhead for data accesses. However, due to hardware limitations more general concepts of pyramidal filters on GPUs, enabling an increasing set of advanced filter kernels as well as a broader range of applications, was inhibited until more recently.

Secondly, all filters are solely based on bilinear interpolation as their fundamental basis operation, i.e. all introduced filters can be constructed by a sequence of bilinear filter kernels. As almost all graphics processing units expose bilinear texture filtering directly as hardware operation, this approach ensures an efficient execution on GPUs. Moreover, it is also this specific characteristic of a GPU's execution environment—exposing nearly identical access overhead for pixel data using point sampling or bilinear interpolation—that motivates the custom design of pyramidal filters for GPUs, which differ fundamentally from the traditional filter kernels targeted for other architectures, such as CPUs. For the class of Bartlett filters, for example, Section 4.1.1 compares the most efficient ways of implementing such filters for CPUs versus GPUs. This also serves as prime example, that applying optimizations valid for CPUs do not necessarily lead to the most efficient implementation of GPUs as well.

After introducing the general concept of GPU-based pyramidal filters in more detail in Section 4.1, the remainder of the chapter focuses on applying the derived kernels in the context of image processing as well as visualization. In particular, examples for the usage of these higher-order GPU-filter operations are given in the context of image manipulation for high-resolution display walls, scattered data interpolation, high-quality image post processing for depth-of-field rendering (all Section 4.2), but may also be used as building block for any other visualization pipeline. The latter will be demonstrated in Section 4.3.1 by extending the volume rendering pipeline introduced in Chapter 3 to support adaptive sampling/filtering in three dimensions.

4.1 Pyramid Filters based on Bilinear Interpolation

Pyramid filters operate on an input signal on multiple scales, that is on multiple resolutions of the input data in the context of image processing. The actual pyramidal filter operation is thereby closely coupled with the construction of the multi-level representation and can be described as an ordered sequence of fundamental convolutional operators. Depending on the relation between input and output resolution, those operations can by categorized into three classes: basic filters, analysis filters, and synthesis filters. The following three sections detail on each of these operations and introduce various filters derivable solely from bilinear filtering. Consequently all of the discussed filters are directly applicable to execution on graphics hardware and can be efficiently implemented via iterative application of texture filtering*.

The formalism to describe the filter operations as well as the implementation using graphics hardware were published first in cooperation with Martin Kraus [SKE06, KS07b]. Additional algorithmic improvements as well as further details on the normalization phase inherent to the proposed filtering mechanism were later on published by Kraus [Kra09b].

4.1.1 Basic Filters

Even though not considered to be a pyramid filter in a strict sense, the simplest class of convolutional operators, discussed in this context, processes the data on a single pyramid level, i.e. the filters do not perform a transition across layers of different sizes. Thus, they directly relate to classical filter operations in image processing. For the design of complex pyramid filter in Section 4.1.2 and Section 4.1.3

*While implementable on any GPU supporting bilinear texture filtering, efficiency of those algorithms is closely coupled to the availability of fast render-to-texture support—nowadays exposed by nearly all consumer graphics processors. That way, the complete pyramid construction can be executed on the GPU by means of ping-pong buffer usage.

such “basic” filters act as building blocks and an efficient implementation of such operations on the GPU is a crucial prerequisite.

In the following, a discrete convolution of the filter mask h and an image g is denoted as

$$f = h \otimes g$$

with the resulting image f being of the same size as input g. If the filter mask h is represented by a $n_i \times n_j$ matrix and image g is of dimensions $n_r \times n_c$, the resulting component at row index r and column index c of image f is defined by:

$$f_{r,c} = \sum_{i=1}^{n_i} \sum_{j=1}^{n_j} h_{i,j} g_{r-\lfloor i-n_i/2 \rfloor, c-\lfloor j-n_j/2 \rfloor}$$

Access to input elements of image $g_{r,c}$ with indices outside of the range of $[1 \dots n_r]$ or $[1 \dots n_c]$ can be handled according to the texture wrap modes of the graphics hardware texture unit—commonly supporting at least clamping, mirroring, or repeating. However, in the special case g represents a filter mask by its own, such elements are set to be zero.

2×2 Box Filter The most important building block for all of following filter operations is the 2×2 box filter, also called uniform, average, or mean filter:

$$h_{\text{box}}^{2\times 2} \overset{\text{def}}{=} \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Its importance is a direct consequence of the efficient implementation of this filter operation on graphics hardware. In fact, this filter maps to the specific case of a bi-linear interpolation with the local coordinates α and β equal $\frac{1}{2}$ (cp. Section 3.2.1), i.e. the interpolated position corresponds to the barycenter of the four input elements. Thus, the operation can be implemented via a single texture lookup.

An important characteristics of graphics hardware, however, is that the bi-linear interpolation effectively comes for free, i.e. does not pose any additional performance overhead compared to a single nearest-neighbor lookup. This is in strict contrast to the theoretical cost for CPU based implementations that expose a linear dependency—ignoring the effects of data caches—on the number of fetched data components. Hence, a nearest neighbor lookup for CPU based filter operations is approximately four times less expensive than a single bi-linear lookup. As a consequence, the efficient mapping of the $h_{\text{box}}^{2\times 2}$ filter to graphics hardware enables several GPU specific filter optimizations, that greatly differ from optimal solutions for other hardware environments.

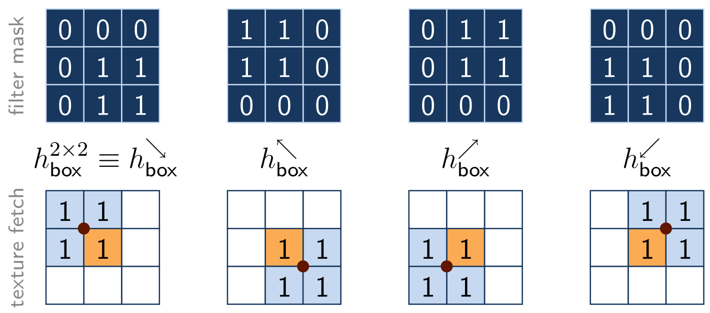

filter mask
texture fetch
0 0 0
0 1 1
0 1 1
1 1 0
1 1 0
0 0 0
0 1 1
0 1 1
0 0 0
0 0 0
1 1 0
1 1 0
$h^{2\times2}_{\text{box}} \equiv h^{\searrow}_{\text{box}}$
$h^{\nwarrow}_{\text{box}}$
$h^{\nearrow}_{\text{box}}$
$h^{\swarrow}_{\text{box}}$

Figure 4.1: All four variants of a 2×2 box filter. Filter results are written to the middle component (marked in orange); the corresponding zero-padded 3×3 filter mask (upper row) change—consequently leading to non-equivalent results—with used texture fetch location (red circles).

For even filter dimensions n_i and n_j, as it is the case for the $h^{2\times2}_{\text{box}}$ filter, discrete rasterization of the target buffer inherently leads to a shift of the output components, as the filter result is defined to match the input grid. For the 2×2 filter example, this equals to writing the result to the position of one of the input elements—introducing a diagonal shift by the length of half a pixel diagonal in the direction of the output element. In order to make this shift explicit, one can use zero padding of the convolutional mask, that is for for each 2×2 convolution mask $h^{2\times2}$ there exists an equivalent 3×3 convolution mask. In correspondence to the aforementioned definition of the convolution, the sampling point for the resulting component with indices r and c is located at the upper, left corner of the pixel specified by r and c in the original source image. Consequently, the corresponding zero-padded 3×3 filter mask for the $h^{2\times2}_{\text{box}}$ mask, is given as:

$$h^{2\times2}_{\text{box}} \quad \equiv \quad h^{\searrow}_{\text{box}} \quad \overset{\text{def}}{=} \quad \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

The arrow symbol in $h^{\searrow}_{\text{box}}$ indicates the inherent shift of the non-zero filter mask elements. Accordingly, three non-equivalent variants of this filter exist by shifting the mask in the remaining directions; the corresponding filter mask in the opposite direction is denoted as:

$$h^{\nwarrow}_{\text{box}} \quad \overset{\text{def}}{=} \quad \frac{1}{4}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

All four implementation options of a 2×2 box filter map to a single texture lookup on graphics hardware—with the difference of a shifted sample location for

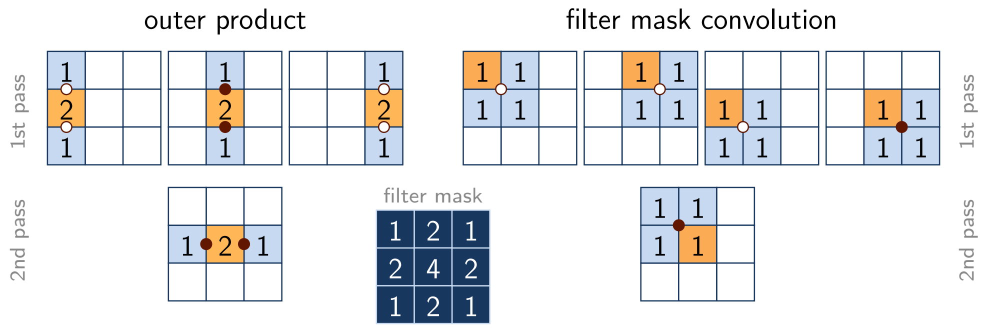


Figure 4.2: Implementation variants of a $h^{3\times3}_{\text{Bartlett}}$ filter for GPU execution. For both, the second pass works on the data stored the first pass; only solid circles mark costs directly related to the target component. In total, the variant exploiting separability requires four texture fetches, the version based on filter convolution only needs two.

the bi-linear texture fetch (see Figure 4.1). Based on these simple operations, more complex filters can be constructed by concatenation of convolutional operations, as shown in the following paragraphs.

3×3 Bartlett Filter Bartlett filters, also called triangular filters, are outer products of one-dimensional triangle, or tent, filters; thus, they are symmetrical and by definition separable. The 3 × 3 Bartlett filter is given as:

$$h^{3\times3}_{\text{Bartlett}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} \cdot \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix}$$

As indicated by the construction of the filter, a two-pass evaluation of the filter exploiting the separability enables an efficient implementation, requiring only six nearest-neighbor data fetches—opposed to the nine data lookups in the brute force implementation. In fact, such an approach is the most efficient one for a CPU based implementation and is typically applied for image processing filters in this context. By means of two output buffers used in a ping-pong fashion, such an approach can directly be mapped to graphics hardware. However, further optimizations are possible in a GPU environment: Due to the specific filter layout the fetch of three nearest-neighbor lookups can be replaced by two bi-linear texture fetches (see Figure 4.2 left). As a result, the overall cost for an application of a $h^{3\times3}_{\text{Bartlett}}$ filter is reduced to four bi-linear texture lookups per output element. Yet, the representation as a convolution of box filters leads to a more efficient implementation:

$$\begin{aligned} h^{3\times 3}_{\text{Bartlett}} &= \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \otimes \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \\ &= h^{\searrow}_{\text{box}} \otimes h^{\nwarrow}_{\text{box}} = h^{\nwarrow}_{\text{box}} \otimes h^{\searrow}_{\text{box}} \end{aligned}$$

A sequence of two convolutions with $h^{\searrow}_{\text{box}}$ and $h^{\nwarrow}_{\text{box}}$ is equivalent to applying a $h^{3\times 3}_{\text{Bartlett}}$ filter (see Figure 4.2 right), thus, this convolutional operator can be implemented most efficiently by only two texture fetches—one lookup for each box filter, as introduced previously.

Gaussian Filters Analog to the construction of the Bartlett filter, operators with larger support can be constructed similarly. According to the central limit theorem, sequences of Bartlett filters lead to approximations of Gaussian filters; for example, a $h^{5\times 5}_{\text{Gauss}}$ filter can be composed via two Bartlett filters:

$$\begin{aligned} h^{5\times 5}_{\text{Gauss}} &\overset{\text{def}}{=} \frac{1}{16^2}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \\ &\equiv h^{3\times 3}_{\text{Bartlett}} \otimes h^{3\times 3}_{\text{Bartlett}} \end{aligned}$$

From a performance perspective, the evaluation of such a filter of size n can be decomposed into $(n-1)/2$ convolutions of $h^{3\times 3}_{\text{Bartlett}}$ filters. Considering the most efficient implementation using two bi-linear interpolations for each Bartlett filter—as discussed in the last paragraph—a convolution with a $h^{n\times n}_{\text{Gauss}}$ filter can be evaluated with $n-1$ texture lookups only. In contrast, by exploiting separability the filter may only be evaluated by $2n-2$ bi-linear lookups or $2n$ nearest-neighbor data fetches.

4.1.2 Analysis Filters

Combining a convolution with a reduce operation, the output image of analysis filters are only half the size in each dimension with respect to the input image. In the context of pyramid filters, this relates to a transition of one pyramid level towards the top of the pyramid. The operation is in the following denoted as

$$f = h \otimes_{\downarrow} g,$$

which defines the components of f according to

$$f_{r,c} = \sum_{i=1}^{n_i} \sum_{j=1}^{n_j} h_{i,j}\, g_{2r-\lfloor i-n_i/2 \rfloor, 2c-\lfloor j-n_j/2 \rfloor}.$$

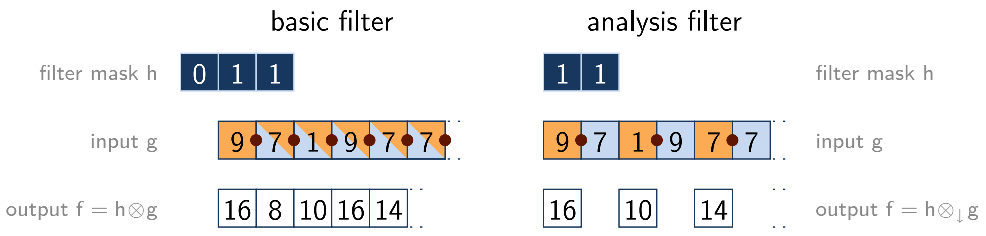


Figure 4.3: Comparison of basic convolution and analysis convolution for the one-dimensional case of a box filter with two pixels support.

Being a combination of a convolution and downsampling, the mapping to graphics hardware introduces a change of the sampling pattern, i.e. a more sparse pattern is applied. However, in case the convolution can be implemented by a single texture lookup, the same filter combined with the reduce operation can consequently also be implemented by a single texture lookup—performing in total only a fourth of texture fetches compared to the basic filter equivalent (see Figure 4.3).

In contrast to the basic filter operations, even filter dimensions are preferable for analysis filters to assure a uniform contribution of even and odd pixel rows and columns to the final image. Even though, this issue may be counterbalanced during filter mask design, it impedes the creation of well-defined filters.

Discontinuous Filter Commonly applied in the context of mipmap pyramids—at least in the standard case of power of two textures [GH03]—the simplest combination of downsampling with a convolutional operator uses the 2×2 box filter (cf. Figure 4.3), denoted as

$$h_{\text{box}}^{2\times2} \otimes_{\downarrow} g \equiv h_{\text{box}}^{\nwarrow} \otimes_{\downarrow} g.$$

Other important applications of this specific filter include the generation of the low pass coefficients of a discrete wavelet transformation using the *Haar* wavelet [Haa10]. A crucial characteristic—in particular in the context of *Haar* wavelets—is that the sequential application of this operation equals always a discontinuous box filter, i.e. with accordingly increased filter support; in reverse, the property entails this filter being classified as discontinuous.

On an implementation side, for each output element of the resulting image, a single texture lookup is necessary to implement this operation on graphics hardware. In combination with a reduce operation, the four aforementioned variants of 2×2 box filter implementation, however, additionally differ in size of the generated output image, i.e. the number of non-zero output rows and columns, due to the alignment of the texture sample pattern with the input image (see Figure 4.4). This is also the reason for using $h_{\text{box}}^{\nwarrow}$ in the above definition—in contrast to $h_{\text{box}}^{\searrow}$

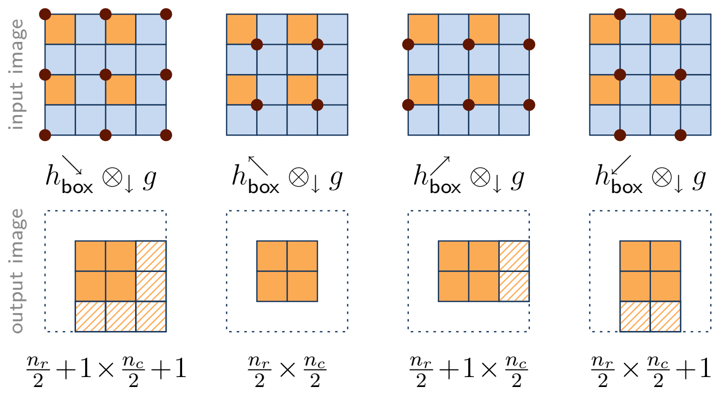

input image
$h^{\searrow}_{\text{box}} \otimes_{\downarrow} g$
$h^{\nwarrow}_{\text{box}} \otimes_{\downarrow} g$
$h^{\nearrow}_{\text{box}} \otimes_{\downarrow} g$
$h^{\swarrow}_{\text{box}} \otimes_{\downarrow} g$
output image
$\frac{n_r}{2}+1 \times \frac{n_c}{2}+1$
$\frac{n_r}{2} \times \frac{n_c}{2}$
$\frac{n_r}{2}+1 \times \frac{n_c}{2}$
$\frac{n_r}{2} \times \frac{n_c}{2}+1$

Figure 4.4: Comparison of implementation variants for combining a 2×2 box convolution with a reduce operation. Depending on the alignment of the texture fetch pattern, i.e. the used box filter implementation, the number of non-zero output rows and columns varies by one component on each dimension.

in the original publication [KS07b]. Defined that way the prerequisite of reducing the input image exactly by a factor of two for each application of the filter is met—enabling a non-degenerated pyramid structure.

C^0-Continuous Filter In contrast to the 2×2 box filter, expanding the filter support to 4×4 leads to a C^0-continuous analysis filter—for the limit of infinite iterations of the filter operations—due to the overlapping filter domains.

$$h^{4\times4}_{\text{box}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

This filter can be constructed by sequential application of an analysis filter and a basic convolution operation of the already reduced intermediate image—effectively leading to equal costs for both filter stages:

$$h^{4\times4}_{\text{box}} \otimes_{\downarrow} g \equiv h^{\searrow}_{\text{box}} \otimes \left(h^{\nwarrow}_{\text{box}} \otimes_{\downarrow} g\right)$$

Consequently, for each output element of the reduced image a total of four bilinear texture fetches is necessary for an evaluation on graphics hardware. Analog to the construction of the basic $h^{3\times3}_{\text{Bartlett}}$ filter, alternating the shift directions avoids introducing pixel shifts larger than half a pixel diagonal. Again, several non-equivalent design alternatives exist, e.g.

$$\tilde{h}^{4\times4}_{\text{box}} \otimes_{\downarrow} g \equiv h^{\nwarrow}_{\text{box}} \otimes \left(h^{\searrow}_{\text{box}} \otimes_{\downarrow} g\right),$$

which actually offsets the application of the 4×4 filter by one pixel along the shift axis in comparison to primal definition. However, due to the previously mentioned size differences of the generated intermediate image after the first convolution step (cf. Figure 4.4), the first implementation approach is favorable.

Exchanging the order of the basic convolution and the analysis filter also allows the construction of a C^0-continuous filter—in fact, it leads to a triangle filter—defined as:

$$h^{3\times3}_{\text{Bartlett}} \otimes_{\downarrow} g \equiv h^{\searrow}_{\text{box}} \otimes_{\downarrow} \left(h^{\nwarrow}_{\text{box}} \otimes g\right)$$

Performance costs, though, are higher due to the application of the basic filter operation to the input image, rather than the reduced one; overall costs per output pixel consequently increase to six bi-linear lookups. Even worse, the analysis of even and odd pixels become asymmetric, which is likely to result in flickering artefacts in the context of animated image sequences.

C^1-Continuous Filter More smooth filter operations can be constructed by combining multiple box convolutions with a reduce operation. For instance, the $h^{4\times4}_{\text{quadratic}}$ filter corresponds to a biquadratic B-spline in the limit of infinity. It is defined as

$$\begin{aligned} h^{4\times4}_{\text{quadratic}} &\overset{\text{def}}{=} \frac{1}{64}\begin{bmatrix} 1 & 3 & 3 & 1 \\ 3 & 9 & 9 & 3 \\ 3 & 9 & 9 & 3 \\ 1 & 3 & 3 & 1 \end{bmatrix} \\ h^{4\times4}_{\text{quadratic}} \otimes_{\downarrow} g &\equiv h^{\searrow}_{\text{box}} \otimes_{\downarrow} \left(h^{3\times3}_{\text{Bartlett}} \otimes g\right) \end{aligned}$$

and can be implemented efficiently via three bi-linear texture lookups only.

Higher-order B-spline analysis filter may be constructed by introducing additional box convolutions prior to the analysis filter, e.g.:

$$h^{5\times5}_{\text{cubic}} \otimes_{\downarrow} g \equiv h^{\searrow}_{\text{box}} \otimes_{\downarrow} \left(h^{3\times3}_{\text{Bartlett}} \otimes \left(h^{\searrow}_{\text{box}} \otimes g\right)\right)$$

4.1.3 Synthesis Filters

In contrast to analysis filter, synthesis filter combine a convolutional operator with an upsampling operation. Hence, in reverse of analysis filters, the dimensions of the input image to a synthesis filter are doubled—actually performing a transition to the next layer towards the base of the pyramid. Analog to the notion of analysis filters, this operation is written in the following way:

$$f = h \otimes_{\uparrow} g,$$

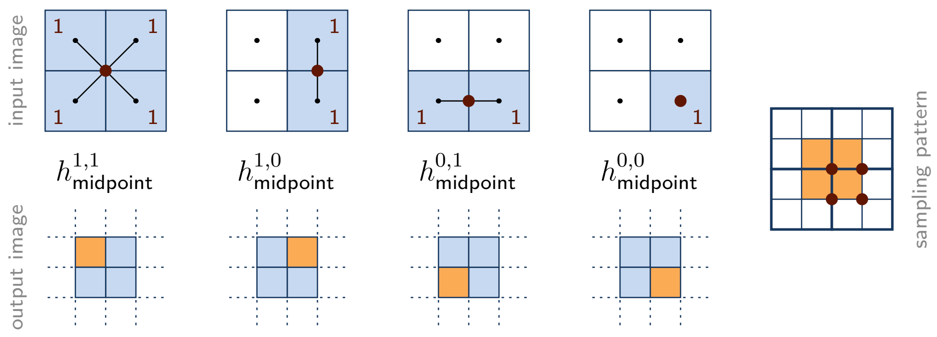

input image
1
1
1
1
$h^{1,1}_{\text{midpoint}}$
$h^{1,0}_{\text{midpoint}}$
$h^{0,1}_{\text{midpoint}}$
$h^{0,0}_{\text{midpoint}}$
output image
sampling pattern

Figure 4.5: Synthesis filtering according to midpoint subdivision: The resulting sampling pattern is grid-centered.

where the components of the output image f are given by

$$f_{r,c} = \sum_{i=1}^{n_i}\sum_{j=1}^{n_j} h_{i,j}^{r \bmod 2, c \bmod 2}\, g_{\lfloor (r+1)/2\rfloor - \lfloor i-n_i/2\rfloor, \lfloor (c+1)/2\rfloor - \lfloor j-n_j/2\rfloor}$$

with $h^{1,1}$, $h^{1,0}$, $h^{0,1}$, and $h^{0,0}$ being the four reconstruction filters that define the rule for generating a 2×2 output patch per input component—the superscript indices denote the location in the corresponding patch.

Given by its fundamental task, synthesis filter are inherently related to the field of data interpolation; in more detail, most of the constructed filter in this section directly correspond to well known subdivision schemes for creating smooth surface geometry (see Zorin et al. [ZSD+00] for more details) and are thus only introduced briefly for completeness.

Discontinuous Filter The simplest form of a synthesis filter is already provided by nearest-neighbor texture interpolation; consequently, it can be trivially implemented by a single texture lookup per output component. In the introduced notation, this basic downsampling operations maps to:

$$h^{1,1}_{\text{nearest}} \overset{\text{def}}{=} h^{1,0}_{\text{nearest}} \overset{\text{def}}{=} h^{0,1}_{\text{nearest}} \overset{\text{def}}{=} h^{0,0}_{\text{nearest}} \overset{\text{def}}{=} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

C^0-Continuous Filter Similarly, bilinear texture interpolation inherently leads a C^0-continuous synthesis upsampling operation. In correspondence to the midpoint subdivision scheme, the synthesis filter h_{midpoint} is defined as:

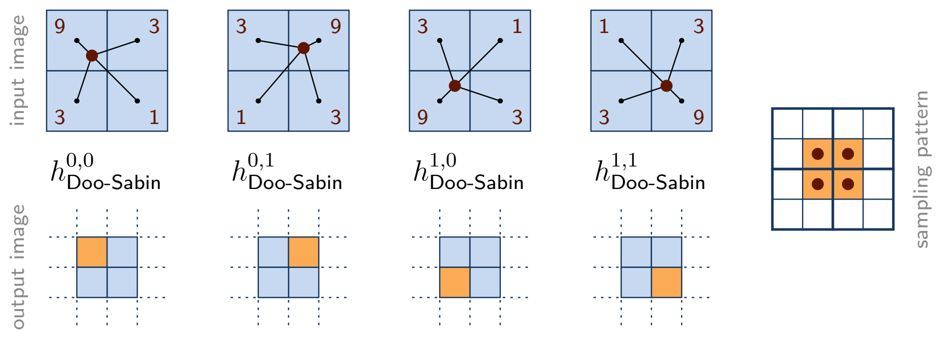

input image
9
3
3
1
$h^{0,0}_{\text{Doo-Sabin}}$
$h^{0,1}_{\text{Doo-Sabin}}$
$h^{1,0}_{\text{Doo-Sabin}}$
$h^{1,1}_{\text{Doo-Sabin}}$
output image
sampling pattern

Figure 4.6: Synthesis filtering according to Doo-Sabin subdivision: The resulting sampling pattern is cell-centered.

$$h^{1,1}_{\text{midpoint}} \overset{\text{def}}{=} \frac{1}{4}\begin{bmatrix}0&0&0\\0&1&1\\0&1&1\end{bmatrix}, \qquad h^{1,0}_{\text{midpoint}} \overset{\text{def}}{=} \frac{1}{2}\begin{bmatrix}0&0&0\\0&1&0\\0&1&0\end{bmatrix}$$

$$h^{0,1}_{\text{midpoint}} \overset{\text{def}}{=} \left(h^{1,0}_{\text{midpoint}}\right)^{\mathsf{T}}, \qquad h^{0,0}_{\text{midpoint}} \overset{\text{def}}{=} \left(h^{0,0}_{\text{nearest}}\right)$$

All four filter operations map again to a single bi-linear texture fetch each—in detail one nearest-neighbor, two linear, and one bi-linear interpolation is required in total—and form a simple uniform sampling pattern of data fetches in the input image (see Figure 4.5).

C^1-Continuous Filter Based on the biquadratic B-spline subdivision scheme introduced by Catmull and Clark [CC78], also known as *Doo-Sabin* subdivision [DS78] for regular quadrilaterals, an approximating C^1-continuous filter $h_{\text{Doo-Sabin}}$ can be derived by:

$$h^{1,1}_{\text{Doo-Sabin}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix}0&0&0\\0&9&3\\0&3&1\end{bmatrix}, \qquad h^{1,0}_{\text{Doo-Sabin}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix}0&0&0\\3&9&0\\1&3&0\end{bmatrix}$$

$$h^{0,1}_{\text{Doo-Sabin}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix}0&3&1\\0&9&3\\0&0&0\end{bmatrix}, \qquad h^{0,0}_{\text{Doo-Sabin}} \overset{\text{def}}{=} \frac{1}{16}\begin{bmatrix}1&3&0\\3&9&0\\0&0&0\end{bmatrix}$$

Interestingly, also these filter operations map to a single bi-linear texture lookup per output element—resulting in an equally efficient evaluation on graphics hardware than the two previously mentioned synthesis filters. An equivalent, but less efficient variant with two-texture interpolations instead of only one, is given as:

$$h_{\text{Doo-Sabin}} \otimes_{\uparrow} g \equiv h^{\nwarrow}_{\text{box}} \otimes \left(h_{\text{midpoint}} \otimes_{\uparrow} g\right)$$

As indicated by this notion, the more efficient version with a single texture fetch is actually a shifted version of the h_{midpoint} filter operation—shifted by half a pixel diagonal with respect to the output image (see Figure 4.6).

$\mathbf{C^2}$-Continuous Filter An C^2-continuous synthesis filter corresponding to the bicubic B-spline subdivision scheme can be constructed by introducing an additional convolution with a 2×2 box filter:

$$h_{\text{cubic}} \otimes_{\uparrow} g \equiv \overset{\nwarrow}{h}_{\text{box}} \otimes (h_{\text{Doo-Sabin}} \otimes_{\uparrow} g)$$

Permutations of the components of this construction results in less symmetric and/or less smooth synthesis filters. However, higher-order B-spline filters can be analogously be constructed by adding further box filter convolutions—continuing with alternating shift directions as already discussed in the context of analysis filters.

4.2 Applications to Image Processing

Pyramidal methods are an integral part for a broad set of highly efficient image processing techniques ever since. Notwithstanding the theoretical advantages, hardware limitations—specifically missing support for efficient render to texture capabilities—inhibited the application of such methods in the context of GPU-based image processing. A noteworthy exception is the specialised fixed functionality for hardware supported texture mipmapping, which supplement this lack of flexibility for the specific task of texture filtering. In fact, mipmaping or rather the underlying mean pyramid is the core of various early approaches to efficiently adopt graphics hardware for image processing. In contrast, the efficient implementation of the pyramidal operators discussed in Section 4.1 nowadays enables extended flexibility for the design of graphics hardware based image operations beyond the mean filtering provided by mipmapping.

The remainder of this section discusses the usage of these pyramid methods for efficient GPU-based image processing by means of the following four example applications: Image blurring (Section 4.2.1), scattered data interpolation (Section 4.2.2), image zooming (Section 4.2.3), and depth-of-field reconstruction (Section 4.2.4). These applications compass work originally published in cooperation with Martin Kraus and Mike Eissele [SKE06, KS07b, KS07a].

4.2.1 Image Blurring

Pyramid methods for image blurring with a filter support of 2^l pixel with $l \in \mathbb{N}^*$, execute l analysis operations followed by the same number of synthesis operations.

For an implementation on graphics hardware, this directly relates to l ping-pong render passes for each direction of processing. If $\Theta_\downarrow$ denotes the analysis filter the number of texture lookups required per output element for a single application of the analysis filter and $\Theta_\uparrow$ is defined accordingly, the overall costs for blurring a $n \times n$ input image is given as:

$$\underbrace{\sum_{i=1}^{l} \frac{\Theta_\downarrow n^2}{4^i}}_{\text{analysis}} + \underbrace{\sum_{i=1}^{l} \frac{\Theta_\uparrow n^2}{4^{i-1}}}_{\text{synthesis}} \quad = \quad n^2 \left(\Theta_\uparrow + \sum_{i=1}^{l-1} \frac{\Theta_\downarrow + \Theta_\uparrow}{4^i} + \frac{\Theta_\downarrow}{4^l} \right)$$

Supposed that $\Theta_\downarrow$, and $\Theta_\uparrow$ respectively, do not depend on n or l—as it is the case for all filters discussed in Section 4.1—the crucial characteristics of pyramidal image burring can be deduced from the upper cost definition (in fact, these properties hold for most of the discussed algorithms in the following): First, the overall costs are linear in the number of input elements. Second, the number of render passes are logarithmic in the width 2^l of the filter. And third, the computational complexity in respect to the rasterized fragments, texture fetches, and buffer writes is bound by a constant—actually costs converge to this limit hyperbolically; for image blurring this cost limit is given by

$$\Theta_\uparrow + \frac{1}{3} (\Theta_\downarrow + \Theta_\uparrow) \,. \tag{4.1}$$

Consequently, for reasonable sized filter masks render cost become effectively uncoupled from the actual filter support.

A great variety of combinations of the analysis filter and the synthesis filter introduced in Section 4.1 result in well-defined pyramidal blur filters: For the analysis the construction of a typical mean pyramid via the $h_{\text{box}}^{2\times2}$ filter ($\Theta_\downarrow = 1$) may be used. However, aliasing effects are in general introduced due to the discontinuous filter operation and likely become visible in the blurred image; especially animated input series get severely affecting by flickering artefacts. Hence, this option clearly favors speed over quality—automatic mipmap generation is the most efficient implementation approach. Alternatively, applying the $h_{\text{box}}^{4\times4}$ filter ($\Theta_\downarrow = 2$) solves these issues in exchange of doubled analysis costs. In both cases, for the synthesis operation the $h_{\text{Doo-Sabin}}$ filter ($\Theta_\uparrow = 1$) is favorable over less smooth alternatives to avoid the typical diamond shaped artifacts of bilinear interpolation. Figure 4.7 provides a comparative series of the obtained blur results against the non-interactive application of an *infinite impulse response* (IIR) Gaussian blur. The latter is of specific interest as Green [Gre05] proposed an efficient GPU based implementation of IIR filters. In contrast to the pyramidal method this method requires a constant number of four render passes, however, each element of the input image needs to be fetched at least twice per render pass. Thus, the texture

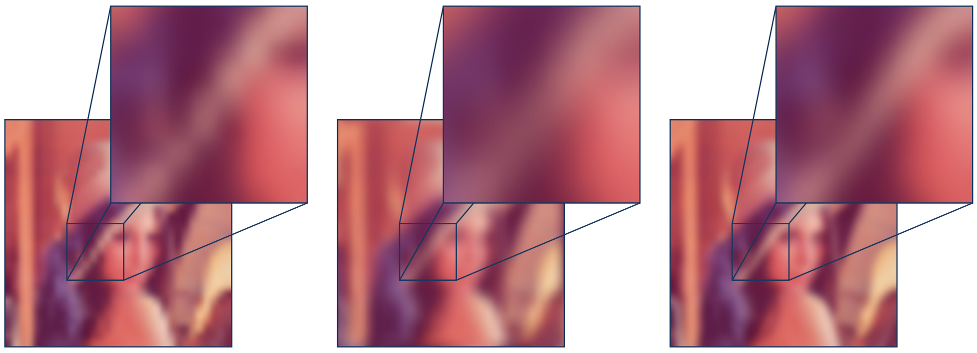

analysis: $h_{\text{box}}^{2\times 2}$
synthesis: $h_{\text{Doo-Sabin}}^{0,0}$
analysis: $h_{\text{box}}^{4\times 4}$
synthesis: $h_{\text{Doo-Sabin}}^{0,0}$
Gauss IIR

Figure 4.7: Comparison of blur results using a 16 pixel wide filter kernel.

fetch costs per input pixel for the implementing the IIR Gauss filter on graphics hardware as proposed by Green is four times higher than the pyramidal variant using the $h_{\text{box}}^{4\times 4}$ filter, which is bounded by two texture fetches per component only (cf. Equation (4.1)). While exposing better runtime characteristics, obviously, the blur results are not equivalent and the pyramidal approach have the disadvantage of only allowing for fixed step blur kernels of size 2^l. Both drawbacks were addressed by Kraus with the introduction of *quasi-convolution pyramidal blurring* [Kra09a], which improves the blur quality even further and also extends the filter concept to allow for a continuous variation of the blur width.

From a performance perspective, pyramidal blurring on a GeForce 7800 GTX requires less than 0.8 milliseconds to blur 1 megapixel using the $h_{\text{box}}^{2\times 2}$ filter with an arbitrary blur width. In accordance to the theoretical cost model, improving the analysis using the $h_{\text{box}}^{4\times 4}$ filter increases render times to 1.1 milliseconds for a million input elements. While providing suitable performance for real-time image post processing, e.g. for computer games and video processing, image blurring can also efficiently be used as basis or building block for more complex interactive algorithms. The next section extends the mechanism to scattered data interpolation, whereas Section 4.2.4 utilizes pyramidal blur filter for depth-or-field rendering.

4.2.2 Scattered Data Interpolation

Being actually among the first applications of pyramidal techniques in image processing, Odgen [OABB85] and Burt [Bur81] discuss methods for filling missing pixels by extrapolating data in the analysis process and selectively filling in synthesized data during the synthesis process. This kind of pyramid methods are

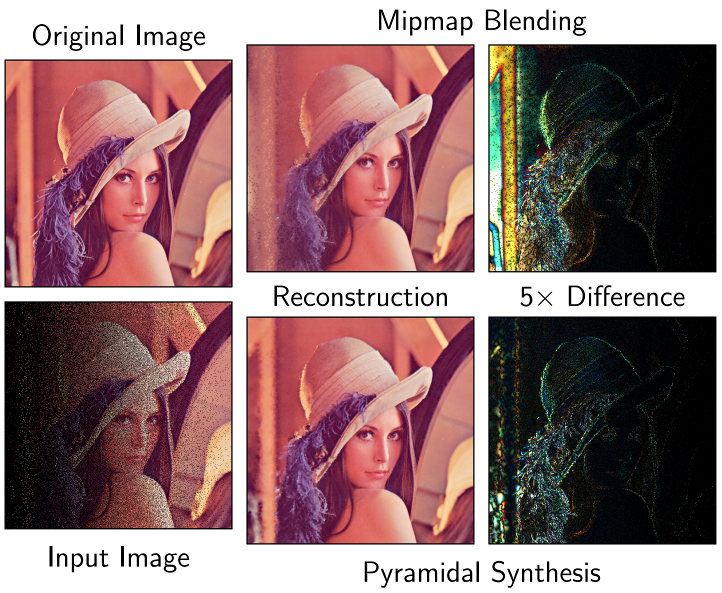


Figure 4.8: Comparison of reconstruction results from scattered data interpolation.

also well-known as *pull-push* methods in computer graphics, which were originally introduced by Gortler et al. [GGSC96] for multi-field data interpolation of the Lumigraph function. Sander and colleagues [SSGH01] applied a similar pull-push algorithm for avoiding artefacts at the boundaries of distinct charts in a texture atlas by extrapolating the boundary colors into the empty atlas regions.

For the same purpose, Lefebvre et al. [LHN05] were the first to propose an efficient GPU based algorithm based on automatic mipmap generation and blending. The input RGB image is extended by the alpha component, to provide a binary mask, called the *support image* by Burt [Bur81], specifying a pixel to be either valid or unset—the alpha component is defined to be 1, or 0 respectively. Consequently, during the mipmap generation, the analysis step tracks for each pixel the ratio between the number of valid input pixels and the total number of combined pixels in the alpha component of each output element. In a subsequent step, the mipmap levels are blended onto each other according to their alpha component. And finally, the normalized output color is derived by a "homogeneous division", i.e. by dividing the composed colors by their resulting alpha component.

While employing the identical analysis operation, an alternative approach for scattered data interpolation can be constructed via the pyramidal $h_{\text{box}}^{4\times4}$ synthesis operation: Prior to the actual synthesis, the current output pixels alpha component—that is the alpha component of the data location written to in the actual synthesis operation—is tested whether it already contains valid or already

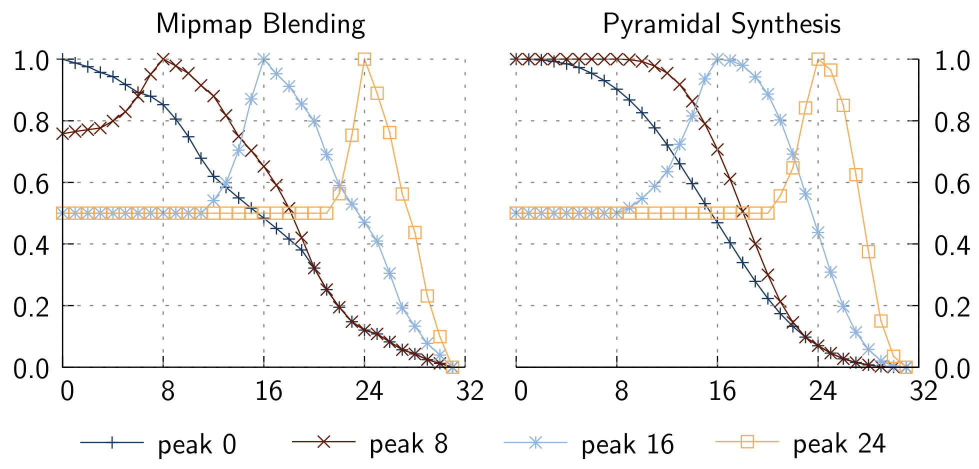


Figure 4.9: Series of reconstruction functions from scattered data interpolation of a 32 pixel wide data field. The four input fields all feature a single high/low pixel pair (data values 1.0/0.0), while all other data is left unspecified: The position of the high peak varies as specified; the rightmost pixel is always defined to be low counterpart.

interpolated color information generated during the analysis. Only if no information is available, i.e. the alpha component equals 0, the synthesised data according to the upsampling operation using the $h_{\text{box}}^{4\times4}$ filter is written; otherwise the current value remains untouched, i.e. the corresponding fragment is killed. As a consequence, the costs $\Theta_{\uparrow}$ per output element of the changed synthesis filter result in two texture lookups in the former case, and remains a single lookup if no synthesis is required.

Figure 4.8 shows a visual comparison of the results of the two reconstructions methods. As it can be seen in those images, the synthesis approach via the pyramidal reduce operation provides more smooth image reconstruction, especially in regions with low input sample frequencies. This characteristic become more obvious by comparing the actual reconstruction functions for a black/white pixel pair with varying numbers of unspecified pixels in between (see Figure 4.9).

In terms of the performance, the approach using pyramidal synthesis features all of the aforementioned beneficial properties; in particular, the costs per output element for the synthesis stage for the worst case of $\Theta_{\uparrow} = 2$ are bound by the constant limit of 2.67 texture fetches (cf. only synthesis in Equation (4.1))—independent of the size of the input image. In comparison, blending all mipmap levels is logarithmic to the input image size. However, several optimization strategies applied in volume rendering, i.e. early ray termination, are applicable to the approach by Lefebvre, capable of levelling the actual render costs for common texture image sizes. Without such optimization strategies, benchmarks using a NVIDIA GeForce 7800 GTX result in a total render time of 0.6, 2.5, 10.0 milliseconds for scattered data interpolation of an 512^2, 1012^2, and 2048^2 sized version of the input image shown in Figure 4.8; in contrast, the pyramidal technique requires 0.41, 1.4, 5.3 milliseconds.

4.2.3 Image Zooming

The process of image magnification is already fully described by the synthesis operation of a pyramid methods—with the overall restriction of the zoom factor being limited to a power of two. In this sense, image zooming corresponds to a multistage interpolation with a constant upsampling ratio of two for each iteration stage. However, the selection of the synthesis filter is crucial to take advantage and justify the additional costs, of a multi-pass zoom algorithm: While both, h_{nearest} and h_{midpoint}, do not benefit from a multistage approach—direct zoom evaluation leads to the identical zoom result without overhead—utilizing the $h_{\text{Doo-Sabin}}$ reconstruction filter enables the already mentioned efficient GPU implementation of higher-order B-spline interpolation—with an upper limit of $m + \frac{1}{3}m$ texture lookups per output element for a filter kernel of order m.

Based also on the efficiency of bilinear texture fetches on graphics hardware, Sigg and Hadwiger [SH05] introduced a sophisticated approach for implementing cubic B-spline interpolation that allows for random data accesses. Utilizing the filter's separability, the overall costs per data fetch for a n-dimensional texture is reduced to $3 \cdot 2^{n-1}$ bilinear texture lookups—improving the performance significantly compared to fetching all input elements directly. Most importantly, the technique can be implemented as a single shader function, i.e. allowing for a drop in replacement of the hardware provides texture fetch functionalities in a shader, and consequently does not pose any additional overhead to the graphics pipeline; thus, the approach is applicable to any image processing or visualization technique—and was applied by the authors to high-quality implicit surface rendering for volume visualization. However, for the special purpose of image zooming—limited to power of two zoom factors—this flexibility comes at costs of a more than doubled data fetch overhead compared to the pyramid method via the h_{cubic} filter.

Edge-Directed Image Zooming Being originally targeted for image magnification in the context of large, high resolution display walls—with a focus on remote visualization or remote desktop applications—the introduced pyramidal image zoom offers suitable performance; however, interpolation—even if done theoretically optimal—in general fails to maintain the sharpness of high frequency details, which leads to a more blurred perception of the input on such display systems.

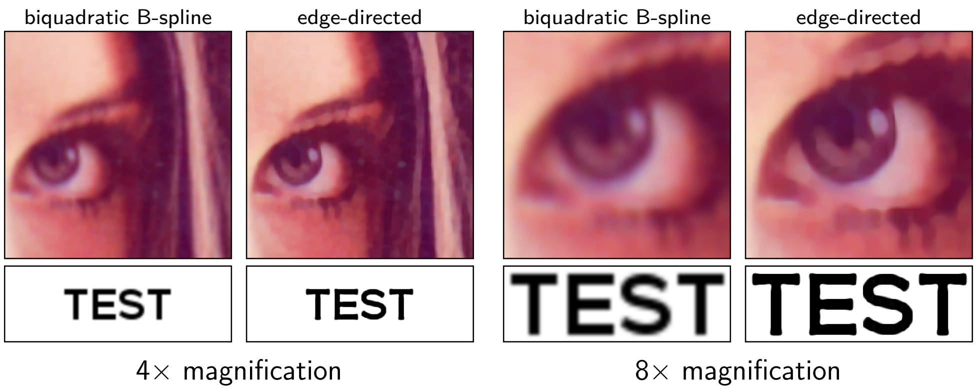


Figure 4.10: Side-by-side comparison of zoom results using the $h_{\text{Doo-Sabin}}$ synthesis filter with and without adaptive edge-directed sample positions.

An attempt to address this issue, was proposed in cooperation with Mike Eissele and Martin Kraus [KES07] by introducing an efficient GPU-based pyramid method for edge-directed image magnification. It utilizes the ideal edge model proposed by Kindlmann and Durkin [KD98] in order to shift the sample locations of the $h_{\text{Doo-Sabin}}$ sampling scheme in the vicinity of an identified edge towards this edge. In doing so, the high frequency features of an image are sampled with an increased frequency, thus, retaining such details in the upsampled image (see Figure 4.10). On an implementation side, the calculation of the sample offset vectors requires 7 texture lookups—making extensive use of the basic filter operations discussed in Section 4.1.1. Hence, synthesis costs per iteration are $\Theta_\uparrow = 8$ and overall costs, independent of the zoom factor, are bound to less than 11 texture lookups per output pixel—still allowing real-time render performance: For zoom factors of 2, 4, 8, and 64 and a target image size of one megapixel, the adaptive zoom method requires 0.70, 0.95, 1.04, and 1.14 milliseconds on a NVIDIA GeForce 8800 GTX.

For a display wall setup with a full HD output resolution for each graphics card—as it is common for 4k-projectors with four input streams—the technique, hence, processes arbitrary sized input content above 400 frames per second using the aforementioned GPUs; thus, zooming performance is sufficiently fast to be integrated into the post-processing pipeline of such a system. In particular in the context of remote desktop applications, the edge-directed interpolation method can improve capturing high frequency details, e.g. improve sharpness and readability of text in the magnified representation.

4.2.4 Depth-Of-Field Rendering

More complex image processing techniques targeted for an efficient evaluation on graphics hardware can be constructed based on the previously introduced concepts. As an example, a pyramid based method for approximating the effect of depth-of-field from a single pinhole image pair, i.e. color and corresponding depth data, was proposed in cooperation with Martin Kraus [KS07a]. While a great variety of interactive algorithms were proposed to visually approximate this effect as a post-process on graphics hardware [Rok93, MvL00, Dem04], most of these interactive methods ignore the problem of partially occluded samples due to computational complexity, or render time restrictions respectively. In contrast, Barsky et al. [BTCH05] presented an non-interactive system, that avoids color bleeding by decomposing the scene into sub-images before blurring and applying color extrapolation to these sub-images to address the issue of partially occluded pixels. Following those ideas, an interactive post-processing technique suitable for efficient evaluation on graphics hardware can be constructed by utilizing the pyramidal techniques described in Sections 4.2.1 and 4.2.2 as building blocks. A short outline of the rendering process is given in the following paragraph:

The input image pair (see upper row in Figure 4.11)is decomposed into several sub-images, whereas each sub-image contains only pixel's of a defined depth range, i.e. initially only the closer bound of these depth ranges is enforced; as a result, each sub-image represents the input image with the foreground culled according to the depth bounds (Figure 4.11c). Culled regions correspond to image areas hidden by closer objects in the input image, however, such areas do partially contribute to the final image assuming a thin lens camera—opposed to a pinhole system. To account for this lack of input data, a common approach is extrapolating the missing data from the unculled sub-image regions. From an implementation point of view, this directly relates to scattered data interpolation, which can be implemented efficiently on graphics hardware via pyramidal methods as described in Section 4.2.2. Interpolating color and depth at the same time, each processed sub-images consequently approximates the scene without the occluding objects (Figure 4.11d). Based on those estimates, each pixel of an sub-image is weighted according to the elements depth in respect to the sub-image's depth range. In this process, the ranges of consecutive sub-images overlap each other to allow for seamless reconstruction of the final rendering result from this set of images. In other words, the true depth location, and thus the corresponding circle of confusion, of an screen element is approximately expressed as linear combination of multiple sub-image elements of discrete depth, and discrete blur radii respectively. Using a trapezium wighting function, only sub-image element inside the corresponding depth range are assigned with non-zero weights (Figure 4.11e). With each sub-image relating to a fixed blur radius—designed to be of form 2^i—application of the scene blur simply implements the pyramidal blur technique introduced in Section 4.2.1. Processing colors plus the attached weights, the fully processed sub-images contain each layer's weighted contribution to the final rendering result (Figure 4.11f). In turn, the sub-images are blended "over" each other to generate the final rendering output (Figure 4.11h).

The analysis of the upper bound of expected data fetches for the multi-stage algorithm with an input image embedded in a $n \times n$ sized buffer and m processed sub-image, relates to (cf. Equation 4.1):

$$n^2 m \Big[\underbrace{1}_{\text{culling}} + \underbrace{2\left(\Theta^{\mathrm{s}}_{\uparrow} + \frac{1}{3}\left(\Theta^{\mathrm{s}}_{\downarrow} + \Theta^{\mathrm{s}}_{\uparrow}\right)\right)}_{\text{scatter}} + \underbrace{1}_{\text{matting}} + \underbrace{\Theta^{\mathrm{b}}_{\uparrow} + \frac{1}{3}\left(\Theta^{\mathrm{b}}_{\downarrow} + \Theta^{\mathrm{b}}_{\uparrow}\right)}_{\text{blur}} + \underbrace{1}_{\text{blending}} \Big]$$

Please note, that the scatter data interpolation is accounted for with doubled costs, as interpolation of color and depth plus the storage of the interpolation weights, requires five scalars to be stored throughout the pyramidal processing; thus, requiring two off-screen buffers and twice the number of texture lookups for current hardware generations. In total, the overall costs are linear to the input elements, linear to the number of sub-images, and limited to an upper bound of 12.67 data fetches per processed sub-image element—assuming the $h^{4\times4}_{\mathrm{box}}$ filter to be used for analysis operations and $h_{\text{Doo-Sabin}}$ for synthesis.

Compared to the reference images rendered using the distributed ray tracer `pbrt` by Pharr and Humphreys [PH04] the pyramidal approach tends to result in more blurred images (best seen by comparing the lower third of Figure 4.11g and Figure 4.11h). This effect is due to the weighted combination of differently blurred versions of the "identical" scene element—being part of multiple overlapping sub-images. As a consequence, the blended result leads to the visual perception of a noticeably stronger blur. Nevertheless, the pyramid method still features high visible similarity to the reference images without introducing severe artefacts—likely to be of sufficient quality, in particular in the context of interactive or real-time applications. Noticeable artefacts common to mipmap-based reverse-mapped z-buffer techniques as stated by Demers [Dem04], e.g. artifacts due to depth discontinuities, due to tri-linear interpolation, or due to pixel bleeding like darkened silhouettes, hardly occur with the described render technique on account of the disocclusion process via pyramidal scattered data interpolation and the biquadratic B-spline interpolation during image blurring.

Despite the multiple render stages, the complete algorithm is executed in 70.4 milliseconds (14.2 fps) for the scene shown in Figure 4.11 on a NVIDIA GeForce 7900 GTX graphics card—processing a total of twelve 1024×1024 sub-images using four half-float RGBA off-screen buffers. Obviously, the definition of the lens radius, i.e. the intensity of the depth-of-field effect, directly relates to the number

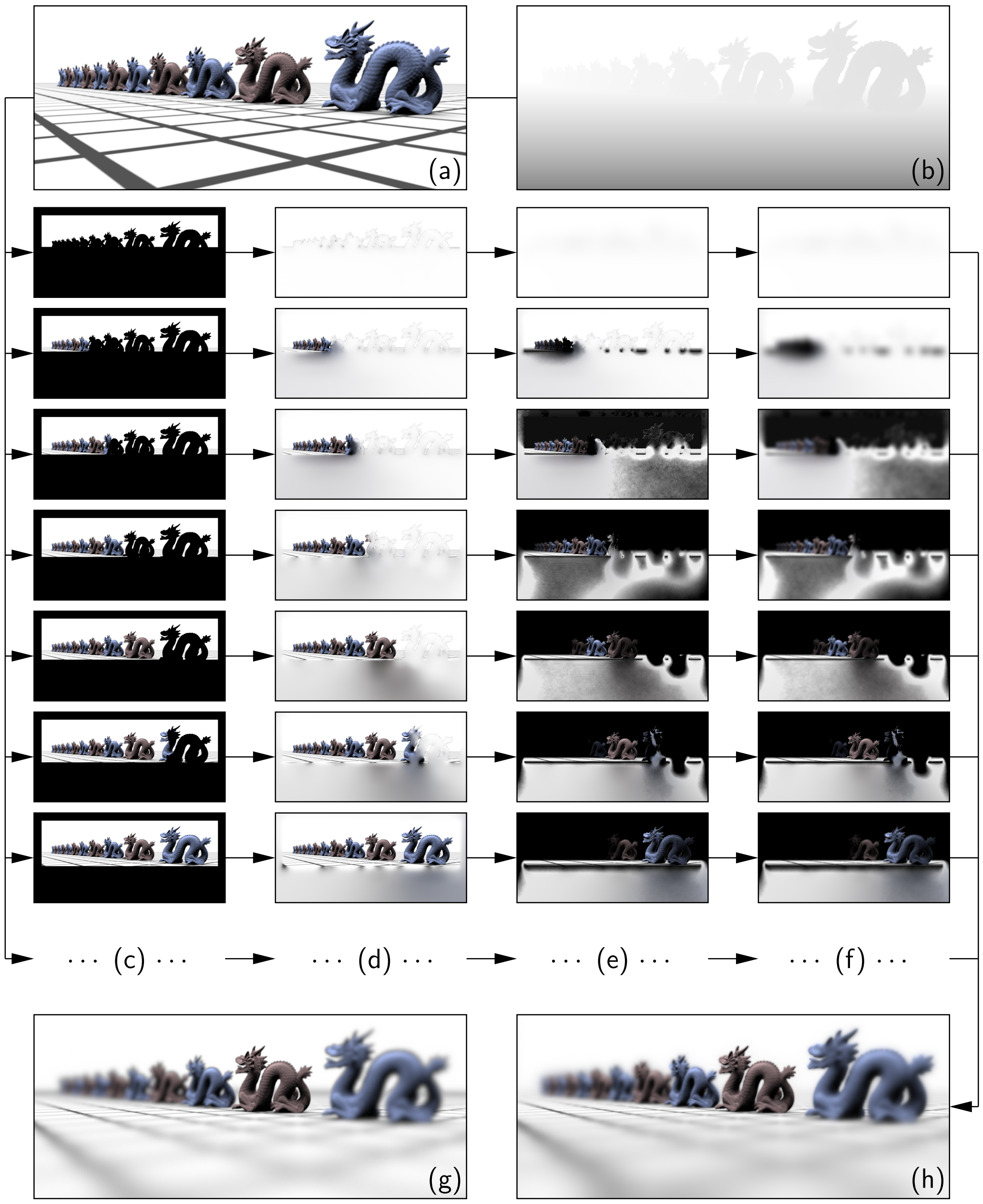


Figure 4.11: Data flow: (a) input pinhole image, (b) input depth map, (c) sub-images after foreground culling, (d) sub-images after disocclusion via scattered data interpolation, (e) sub-images after matting, (f) sub-images after blurring, (g) ray traced reference, (h) blended result of pyramid method.

of sub-images and the required blur radii. Hence, a variant of the same scene with a more subtle application of the effect—requiring only six sub-images—can be rendered in about 35.5 milliseconds (28.2 fps).

While being still out of reach for real-time post processing of graphics intense applications, i.e. games or visualization, performance is already suitable for high-quality rendering of depth-of-field effects for real-time video feeds or for fast pre-visualization of off-line rendering. Being based on the introduced pyramidal methods, the technique scales well with target image size; however, maximum output sizes become rapidly bound by current hardware limitations, like maximum off-screen buffer sizes or texture memory restrictions.

4.3 Applications to Visualization

Pyramidal methods can significantly improve run-time characteristics of image processing techniques on graphics hardware, i.e. effectively uncoupling memory bandwidth requirements from kernel sizes. Likewise GPU-based visualization techniques may equally draw benefit from pyramidal data processing: While some of the presented techniques inherently relate to the filtering stage of visualization in general, e.g. enabling higher-order B-spline data reconstruction or efficient on-the-fly scattered data interpolation, pyramid methods lately gained increased attention beyond filtering, being more directly geared towards efficient design of GPU-based visualization techniques—a trend also observed in the context of general purpose computation on graphics hardware (cf. stream reduce operations of Brook [BFH$^+$04b], or parallel prefix sums in CUDA [Har07]). Applications in visualization span areas like efficient data compaction for point-cloud generation [ZTTS06], reconstruction of soft shadows with low a number of sampling points [GBP07], extraction of silhouettes edge from a geometry model [DRS08], point-based rendering using image reconstruction [MKC08], or iso-surface extraction from volume data [DZTS07].

In the context of of large, high-resolution display devices, a pyramidal method for adaptive sampling in three dimensions for volume rendering was published in cooperation with Martin Kraus and Thomas Klein [KSKE07]. Serving as application example for pyramidal techniques in the design visualization algorithms, this technique is further detailed in the next section.

4.3.1 Adaptive Volume Sampling in Three Dimensions

Volume rendering on most render systems—based on CPUs, GPUs, or special purpose hardware alike—is commonly bound by the system's performance of fetching the interpolated volume data. Consequently, the render costs directly relate to the number of required data samples to generate the final image. To address this common bottleneck, a broad class of optimization strategies is based on the idea to skip samples in volume areas, that either do not contribute to the final image or may be sampled less frequent without significant loss of quality. These adaptive sampling techniques can be coarsely divided into three categories:

Based on the paradigm of casting rays through the volume, the first group varies the sample distance along each ray independently. Primary examples are empty space skipping and early-ray termination as discussed in Section 3.3.2, but also more elaborate strategies that adapt sampling in respect to the local quality of volume reconstruction [Kra03, BMWM06]. While offering great potential for savings in the direction of the ray, such techniques, however, still scale linear to output resolution, i.e. miss to exploit possible savings in the in the dimensions of the image plane.

In contrast, the second group of techniques applies adaptive image-space sampling by progressively refining the shooting of rays in image space according to visual errors or variation between the evaluation of the volume rendering integral of spatially adjacent rays [AAN05, Lju06]. In combination with the aforementioned class of techniques, adaptive sampling in all three dimensions can effectively be achieved, but since refinement in screen-space is based on final render results after the volume integration missed details and/or less optimal adaptivity can not be avoided completely.

Finally, the third group builds upon hierarchical data structures representing the input scalar field at multiple resolutions. During rendering coarser approximations are sampled instead of the original data where appropriate, i.e. based on error metrics in the hierarchy and/or view parameters [LHJ99, GWGS02]. To this end, texture memory is traded for more efficient volume sampling.

In an attempt to address the drawbacks of all three types of optimization strategies, a new adaptive sampling mechanism was introduced that allows for data local refinement in all three spatial dimensions. In a nutshell, the primary goal was to significantly reduce the number data fetches for GPU-based volume rendering—thus enabling better scaling characteristics for high resolution displays.

Prerequisites The method is based on three assumptions: First, data sampling for all rays in the image space is executed in lock step mode in parallel; as a consequence, the technique is integrated into a front-to-back slice based volume renderer using view-aligned slices as proxy geometry. Second, it is possible to specify an oracle that returns the preferable sampling distance for any point in the volume, which is accessible from a fragment shader. The oracle may or may not be view-dependent and it is optimally computed on the fly; however, it might also be precomputed and stored in graphics memory—breaking the aim of not introducing additional memory overhead. And third, sampling distances are bound from below by the distance between view rays corresponding to adjacent pixels of the image plane. In other words, sampling distances given by the pixel distance in the image plane limit the maximum sampling distance along the ray—actually higher sampling distances, even if commonly used, may introduce sampling artefacts during rotation. For slice base volume rendering, this results in having the uniform slice distance to correspond to the distance between neighboring view rays; in the following all distances are, thus, specified as in units of this smallest sampling distance.

Adaptively Skipping Sampling Points Adaptive sampling is achieved by querying the oracle for the sampling distance d at each sampling point starting on the 0th slice, i.e., the front most slice. The result is stored alongside the data sample in the frame buffer and updated with new values of each sampling point on the same view ray unless the sampling point is skipped. Sampling points on the next slices that are projected to the same pixel (i.e., that are on the same view ray) are skipped in dependency of the stored distance d; however, at most $d-1$ sampling points are skipped; thus, the sampling rate does not drop below the sampling rate required by the oracle if d is not less than 1 in the units described above. The actual condition for skipping a sampling point on the i_{slice}-th slice is $d \geq 2^{\text{slice}+1}$ where d is the distance stored in the frame buffer and

$$l_{\text{slice}} = \max\{l \in \mathbb{N}_0 \mid i_{\text{slice}} \bmod 2^l = 0 \;\wedge\; l \leq l_{\max}\} \tag{4.2}$$

with a positive integer $l_{\max}$ that bounds the maximum actual sampling distance from above by $d = 2^{\text{lmax}}$. The index of the last slice should be divisible by $2^{l_{\max}}$ to ensure sufficient sampling at boundaries. Note that l_{slice} is just the number of trailing zeros (bounded from above by $l_{\max}$) in a binary representation of the index i_{slice}. One important feature of this scheme is that the skipping condition relies exclusively on previously stored results of the oracle and the index of the slice. This is crucial for an implementation that relies on early pixel-depth tests.

Adaptive Sampling of the Zeroth Slice While the previous paragraph addresses adaptive sampling along view rays, i.e. corresponds to the first group of techniques in the above classification, this section discusses the adaptive sampling of one slice—in fact, the 0th slice—while the combination with the adaptive sampling on view rays is described in the following paragraph.

The data required for the sampling of all view rays (of the finest level) is stored in an image of $w \times h$ pixels. For the adaptive sampling of this image on the 0th slice, the algorithm employs pyramidal synthesis on $l_{\max}$+1 levels. All pixels of the topmost level $l_{\max}$ are sampled while pixels on lower levels are either synthesized or also sampled if the oracle requires a smaller sampling distance. The first row

```
PROJECTSLICE( $i_{\text{slice}}$ ):
  $l_{\text{slice}} \leftarrow \max\{l \in \mathbb{N}_0 \mid i_{\text{slice}} \bmod 2^l = 0 \wedge l \leq l_{\max}\}$
  for $l \leftarrow l_{\text{slice}}$ to 0 do
    for $x \leftarrow 0$ to $w/2^l - 1$ do
      for $y \leftarrow 0$ to $h/2^l - 1$ do
        if ($l < l_{\text{slice}}$ and ($l_{\text{slice}} = l_{\max}$ or $d_l(x,y) < 2^{l_{\text{slice}}+1}$)) {
          $d_l(x,y) \leftarrow$ SYNTHESIZE( d , l , x , y )
          $s_l(x,y) \leftarrow$ SYNTHESIZE( s , l , x , y )
        }
        if ($l = l_{\max}$ or $d_l(x,y) < 2^{l+1}$) {
          $d_l(x,y) \leftarrow$ ORACLE( l , x , y )
          $s_l(x,y) \leftarrow$ SAMPLE( l , x , y )
        }
        if ($l = 0$ and ($l_{\text{slice}} = l_{\max}$ or $d_l(x,y) < 2^{l_{\text{slice}}+1}$)) {
          ACCUMULATE( $x, y, i_{\text{slice}}$ )
        }
```

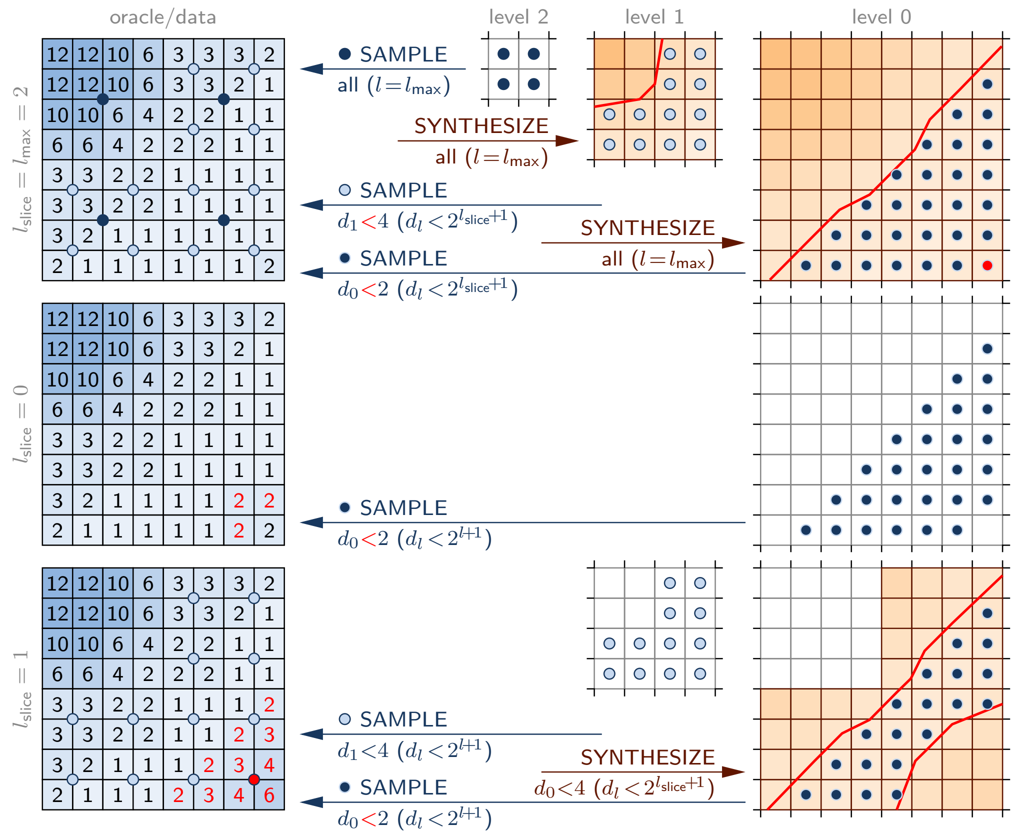

oracle/data
level 2
level 1
level 0
$l_{\text{slice}} = l_{\max} = 2$
$l_{\text{slice}} = 0$
$l_{\text{slice}} = 1$
● SAMPLE
all ($l = l_{\max}$)
SYNTHESIZE
all ($l = l_{\max}$)
○ SAMPLE
$d_1 < 4$ ($d_l < 2^{l_{\text{slice}}+1}$)
SYNTHESIZE
all ($l = l_{\max}$)
● SAMPLE
$d_0 < 2$ ($d_l < 2^{l_{\text{slice}}+1}$)
● SAMPLE
$d_0 < 2$ ($d_l < 2^{l+1}$)
○ SAMPLE
$d_1 < 4$ ($d_l < 2^{l+1}$)
SYNTHESIZE
$d_0 < 4$ ($d_l < 2^{l_{\text{slice}}+1}$)
● SAMPLE
$d_0 < 2$ ($d_l < 2^{l+1}$)

Figure 4.12: Pseudo code with example. The oracle (left) controls the pyramidal sampling/synthesis (blue/red arrows). For each stage the triggering condition is given below the arrow. Due to the oracle, the upper left area—later also the lower right—is excluded from synthesis (white elements) and sampling (elements without dot).

in the example of Figure 4.12 illustrates the adaptive sampling of the 0th slice for $l_{\max}=2$; pseudo code for the algorithms is given side-by-side.

The pyramidal synthesis for the 0th slice starts from the coarsest level $l_{\max}$ of size $w/2^{l_{\max}} \times h/2^{l_{\max}}$ (assuming that w and h are divisible by $2^{l_{\max}}$). On this topmost level $l_{\max}$ no sampling points may be skipped; therefore, data values $s_{l_{\max}}(x, y)$ are sampled from the volumetric data and sampling distances $d_{l_{\max}}(x, y)$ are requested from the oracle for all pixels (x, y) of level $l_{\max}$. The coordinates of the sampling point for a pixel (x, y) on level l is determined by a mapping to pixel coordinates on level 0 and thereby to a sampling point on the corresponding view ray. The subroutine SAMPLE(l, x, y) fetches the input data at the three-dimensional point corresponding to the pixel (x, y) of level l of the current slice, while ORACLE(l, x, y) returns the sampling distance for this point given by the oracle.

For each pixel (x, y) of the next level $l=l_{\max}-1$, the algorithm synthesizes scalar data $s_l(x, y)$ and a sampling distance $d_l(x, y)$ from the values on level $l+1=l_{\max}$ (denoted as orange components in Figure 4.12). Since the actual distance between the sampling points corresponding to two adjacent pixels of level l is 2^l, sampling on this level l is necessary if the synthesized sampling distance suggested by the oracle is less than 2^{l+1} (condition $d=2^l$ is marked in red in Figure 4.12). In this case the volumetric data is sampled and the oracle's sampling distance is computed and stored in $s_l(x, y)$ and $d_l(x, y)$, respectively.

The synthesis of data and distances on finer levels is specified by the function SYNTHESIZE(s, l, x, y) and SYNTHESIZE(d, l, x, y), respectively, which implements the $h_{\text{Doo-Sabin}}$ synthesis filter; resulting in a biquadratic B-spline interpolation for the reconstruction of coarser sampled areas.

Putting It Together The pseudo code in Figure 4.12 already covers the adaptive sampling of all slices. The main difference between the sampling of the 0th slice discussed previously and the sampling of further slices is the computation of l_{slice}. While l_{slice} is equal to $l_{\max}$ for the 0th slice, it is computed according to Equation (4.2) for a general slice index i_{slice}.

l_{slice} determines the starting level for the loop over l in the pyramidal synthesis. For example, the 0th slice processes all levels, while the 1st slice processes only the bottommost level 0 (second row in example of Figure 4.12). After the first sampling on the 0th slice, level 1 is updated for the first time on the 2nd slice (third row in the example) and level 2 on the 4th slice, etc. In general, samples on level l are updated for the first time on the 2^l-th slice. More samples for level l are unnecessary since the distance between sampling points of adjacent pixels on level l is 2^l in units of the minimum distance between sampling points. Note that data that has been synthesized from data of level l, is always replaced by new samples on level $l-1$ taken at smaller sampling distances if required by the oracle.

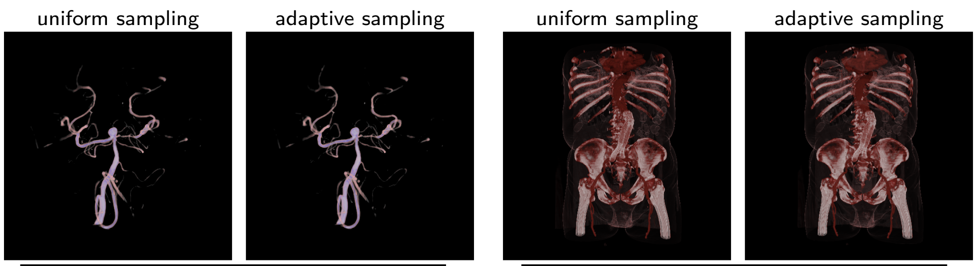
The condition $d < 2^{l_{\text{slice}}+1}$ for actually performing the sampling becomes $l < l_{\text{slice}}$ *and* $d_l(x,y) < 2^{l_{\text{slice}}+1}$ in Figure 4.12. The former ensures that no new values from data of level $l_{\text{slice}}+1$ is synthesized, which is not updated for this slice since the loop over l starts with l_{slice}. The remaining algorithm for sampling slices is exactly as in the case of the 0th slice.

GPU Implementation Implementation of the algorithm on graphics hardware maps to a multi-pass render algorithm: Each processing stage, i.e. sampling, synthesis, and blending, are realized as an independent render pass; intermediate results are stored in and subsequently sampled from 16-bit floating-point off-screen buffers.

Since all conditional statement guarding each processing stage, i.e. sampling, synthesis, and blending, are solely dependent on per-slice constants (l, l_{slice}, and $l_{\max}$) the straightforward implementation variant would interpret the oracle data as depth information; by doing so, the conditions could be implemented via the `EXT_depth_bounds_test` extension[†]. However, at the time of writing no graphics hardware supports an early depth bound test in combination with changing bounds. In detail, the tests is only performed prior to fragment shader execution as long as the depth bound remain constant. Thus, an alternative—less optimal—implementation approach linearizes the use of the depth buffer by assigning each volume slice a fixed depth range. This way, it can be assured that all conditional tests can be implemented by the traditional depth test with a constant compare operation `GL_LESS`—fulfilling the requirements for an early rejection of fragments via the depth test (please see [KSKE07] for more details).

Conclusion For evaluating the adaptive sampling technique in comparison to traditional slice-based volume rendering, two comparative studies were performed on a NVIDIA GeForce 7800 GTX graphics card (see Figure 4.13). Most importantly, those benchmarks reveal significant savings of data accesses to the original input scalar field—the main motivation of this technique. For both data sets, visual comparable results can be obtained from less than 15% of sampling steps compared to the uniform case in both setups. Such benefits only comes at costs of an increased fragment processing load—a consequence of the pyramidal synthesis process per slice—however, most of the fragments are efficiently culled due to early depth comparison prior to shader execution.

[†]The depth bound tests relates to the current depth value stored in the depth buffer at a fragment's raster position prior to shader execution—in contrast to the depth test, that relates to depth of the currently processed fragment. Thus, the depth bound test may be evaluated "early" even if the depth is written in the fragment shader.



Aneurysm	uniform	adaptive
no. fragments	49 M	61 M
no. samples	96 M	1 M
time in secs	0.87	0.23

Abdomen	uniform	adaptive
no. fragments	37 M	61 M
no. samples	74 M	9 M
time in secs	0.76	0.38

Figure 4.13: Comparison of traditional slice based volume rendering versus pyramidal adaptive sampling; both 512^3 data sets[‡] were rendered in a 512×512 viewport using pre-integration. The ad-hoc oracle returned the precomputed distance to $|\nabla\alpha| > \epsilon$.

While the render performance slightly increases for both evaluated setups, the currently possible implementation approach requires high computational as well as implementation overhead compared to traditional slice based volume rendering—hindering performance benefits to be linear to the number of saved data lookups. Nevertheless, the algorithm may be of particular interest in combination with complex and/or compressed volume data structures, which pose very high costs for data sampling. For example, volume rendering based on adaptive texture maps [KE02], adaptive mesh refinement [VSE06], radial basis functions [JWH$^+$04], or on-the-fly resampling of unstructured grids [CDM06] would be well suited candidates for further research in the context of adaptive sampling. In all such cases the higher fragment processing load may be used to avoid the intrinsic data fetch bottlenecks.

In a summary, the presented technique is only an initial step towards efficient and practical application of three-dimensional adaptive volume sampling. Several aspects of the algorithms need further research, e.g. suitable oracle functions, on-the-fly evaluation of oracle data, reduction of the fragment processing overhead; nevertheless, the presented algorithm indicates the high potential of pyramidal techniques in the context of volume visualization.

[‡]The aneurysm data set is courtesy of Michael Meißner, Viatronix Inc.

Chapter 5

Parallel Visualization Techniques for Graphics Clusters

The previous two chapters introduced diverse approaches and methodologies for porting critical building blocks of the visualization pipeline to match the execution model of modern graphics processors. Providing interactivity for complex visualization tasks by utilizing the GPU's inherent parallelism, is the common motivation for all these techniques.

This chapter continues in leveraging parallelism for interactive visualization, however, at larger scale. On the one hand, such "*larger-scale*" relates to the targeted visualization challenge—typically by far exceeding the capabilities of a single visualization workstation in both, manageable data sizes as well as required visualization performance. On the other hand, "*larger-scale*" relates to the level of applied parallelism that shifts from the hardware architecture of a single chip to the system architecture of complete cluster environments. In particular, this thesis focuses on GPU cluster environments—a interlinked group of visualization nodes, each equipped with at least a single or multiple graphic processing units.

While this chapter discusses the concepts of parallelism for distributed visualization primarily in the context of volume visualization, the underlying techniques are applicable to other visualization algorithms as well. As an example for this generality may serve a distributed visualization system for vector field visualization, which was developed as part of this thesis in cooperation with Sven Bachthaler and Daniel Weiskopf [BSWE06]. However, apart from the shared, underlying parallelization strategies, this specific system is not further discussed here.

5.1 Distribution Paradigms

Efficient decomposition of computational workload to utilize the capabilities of parallel compute environments has multitudinous forms in the field of parallel computing—spanning across nearly all levels of processing architectures: From a single processor core, to multi-core systems, to processor arrays, up to large scale cluster systems. Not limited to the field of graphics rendering, but of particular interest to the execution model of graphics hardware, general partition paradigms include: *Instruction Parallelism*, i.e. the simultaneous execution of multiple independent operations; for example, implemented as dual-issue of one *multiply-and-add* (*MAD*) plus one *multiply* (*MUL*) instruction per clock cycle on a NVIDIA GT200 architecture. *Data Parallelism*, i.e. the distribution of data across processors for parallelized execution; heavily exploited in a *single instruction multiple data* (*SIMD*) fashion by processing multiple elements concurrently in each shader stage. As well as *Task Parallelism*, i.e. the simultaneous execution of various different tasks in parallel; for example, manifested conceptually by the pipeline paradigm of vertex, primitive, and fragment processing in the rendering pipeline as well as its actual realization in hardware.

Specifically focused on the partitioning of workload for distributed rendering systems, Whitman [Whi92] as well as Molnar et al. [MCEF94] introduced taxonomies based on a fully parallelized graphics pipeline—that is a pipeline defined to execute the render stages of geometry processing and rasterization in parallel. While Whitman categorizes distributed rendering primarily by means of the conceptual space of the graphics pipeline in which the decomposition is realized, Molnar and colleagues understand parallel rendering as a sorting problem of input primitives to output pixels. In detail, this sorting process links the arbitrary input set of primitives, of which each can contribute to any portion of the screen, to the fixed grid of output pixels in the final image. The classification is then based on the location in rendering pipeline at which this sorting is performed. In other words, the two render stages of primitive transform and rasterization may each work either in relation to the uncorrelated input primitives or in relation to a distinct spatial reference in the output image. Hence, the sorting step equals a redistribution of both, graphical elements and the responsibility to process them, between the two pipeline stages. This way, the three categories of *sort-first*, *sort-middle*, and *sort-last* systems can be defined.

Following the taxonomy of Molnar et al. the next sections briefly discuss those three fundamental parallelization strategies with a special focus on multi-GPU systems and graphics clusters.

5.1.1 Sort-First Parallel Architectures

As indicated by the naming scheme the redistribution process for sort-first partitioning is performed at *first*, that is prior to geometry processing. More precisely, primitives are pre-assigned to a distinct processor that is able to perform the complete render process for its input. Conceptually, such a distribution paradigm mimics multiple discrete implementations of a full rendering pipeline, each responsible for a disjunct region of the final rendering—therefore sort-first architectures are often interchangeably referred to as parallel *image-space partitioning* techniques, according to the taxonomy by Whitman; the sorting process, thus, boils down to assign each input primitive to the responsible pipeline with respect to the actual screen-space position, which in turn requires each primitive to be at least partially transformed before sorting. However, this pre-transform may be performed in an approximately manner, e.g. by projecting bounding volumes only, and therefore is supposed to be computationally cheap compared to the full evaluation of transform and lighting. It is important to note, that input primitives crossing the boundaries of the disjunct image-space regions need to be split and/or assigned to multiple processors, which may hinder scaling behaviour for large scale systems.

While not gained much attention in the parallel community early on, sort-first architectures became increasingly popular in the context of large output devices, like tiled displays or large projection walls, due to the excellent scaling behaviour in respect to output resolution. For cluster environments commonly the input data are either replicated or communicated over then network interconnect, however, such fully parallelized sort-first systems are still rare. Instead, in the context of parallel GPU-based rendering systems, sort-first approaches are often focused on scaling a system's graphics pipeline throughput, while the actual host application is commonly executed monolithically. Hence, sorting is primarily concerned to send the input primitives to the responsible GPU.

Humphreys et al. [HEB$^+$01] devised OpenGL call interception to exchange the render back end of an arbitrary OpenGL application transparently with a distributed render pipeline based on static sort-first partitioning across cluster nodes. This work also built the basis for the "tilesort" stream processing unit—enabling dynamic tiled-based rendering—in the more flexible *Chromium* system [HHN$^+$02]. For both approaches, the input primitives are sorted according to their approximated pre-computed screen footprint and redistributed across a network interconnect to the responsible render node. In contrast the *Equalizer Framework* presented by Eilemann et al. [EMP09] executes parts of the application in parallel on the render nodes. If configured for sort-first rendering the sorting stage can then be performed locally on each node and costly network communication can be significantly reduced.

Multi-GPU systems nowadays provide similar driver-level parallelization across multiple graphics processors, i.e. AMD Crossfire or NVIDIA SLI platform. However, it is likely that such techniques actually replicate all input data, i.e. broadcast the graphics command stream to all GPUs, and geometry processing is effectively executed for all primitives on all cards. The actual partitioning, thus, is performed not until frustum culling, i.e. after the complete geometry processing is finished. Hence, such a procedure does not fully conform to a strict definition of a sort-first architecture—or may only be seen as a degenerated case with the complete transform and lighting calculation performed as pre-transformation; in addition, the constructed parallel pipeline is actually not a fully parallelized one, since geometry processing is performed for all input primitives on each processor—effectively impeding scaling of vertex and geometry workload across multiple graphics processors. Yet, as untransformed input primitives are communicated across the parallel processors, such systems are in general classified as sort-first methods.

For the special application of distributed GPU-based volume rendering in a cluster environment further details are given in Section 5.2 with a special emphasis on image collection (Section 5.2.2) and load balancing strategies (Section 5.2.3).

5.1.2 Sort-Middle Parallel Architectures

In sort-middle architectures, the redistribution of data is placed in the *middle* of geometry processing and rasterization; consequently, fully transformed screen-space primitives, i.e. in general points or triangles only, are distributed and sorted in respect to the raster processors' disjunct areas in the final image. In effect, any additional overhead for pre-transformation, as required by sort-first architectures, is avoided.

Redistributing data in between the render pipeline effectively unlinks geometry processing from fragment processing, breaking the pipeline into two parts. Opposed to the conceptual model of multiple distinct parallel pipelines in case of sort-first parallelization, sort-middle systems naturally map to two separate sets of parallel stages connected via the redistribution phase. Consequently, both stages become fully independent of each other, allowing for any diverse level of parallelism between the pipeline stages, whilst sort-first architectures commonly feature a multiple of fixed geometry/raster pairs.

Originally the sort-middle paradigm was the most common approach for parallel render pipelines in both, specialized hardware and software architectures. Specific hardware examples include the UNC Pixel-Planes 5 [FPE$^+$89] or the SGI's Reality Engine [Ake93]. The latter utilizes a custom triangle bus crossbar to evade the likely bottleneck of data communication in between the pipeline tasks. While such systems were successfully applied to parallel rendering systems, the sort-middle paradigm was not applicable to GPU cluster systems based on commodity hardware, as the traditional rendering pipeline of graphics hardware originally did not provide access to the projected vertex elements prior to rasterization.

Interestingly, such a capability became available with DirectX10 compatible graphics hardware via the transform feedback functionality, also named stream out mechanism. This way, transformed vertices may be retrieved directly from the GPU, may be sorted on the CPU or the GPU*, and can be feed back into

*Sorting can be performed in multiple passes in a geometry shader currently, but can also be mapped to a single DirectX11/OpenCL compute shader working on the vertex buffer.

the hardware pipeline, ultimately providing all requirements to implement a distributed GPU-based sort-middle architecture. At the time of writing, however, no such system was so far published or realized to my knowledge.

5.1.3 Sort-Last Parallel Architectures

Postponing the sorting stage to be executed *last*, i.e. after fragment processing, results in the paradigm of sort-last partitioning. Work distribution across geometry and rasterization units thus becomes independent of the spatial location of a primitive or fragment. The conceptual model of sort-last consequently resembles the idea of sort-first, having multiple distinct render pipelines, i.e. geometry/raster processor pairs, with the difference of performing the sorting after the pipeline execution. In general each of the parallel pipelines outputs pixels at arbitrary locations in image-space, hence rendering a sub-section of the input primitives in a full sized output image. The sorting phase then redistributes those pixel data to reconstruct a final rendering result by resolving the visibility, more precisely evaluating the contribution, of spatially collocated sub-image element.

As indicated each distinct rendering pipeline processes only a subset of the original input primitive stream, making sort-last architectures perfectly suitable to scale with the size of the rendered data—therefrom sort-last systems are also commonly referred to as *object-space* partitioning techniques as introduced by the taxonomy of Whitman. With respect to the limited amount of local memory per compute processor—particularly in the context of on-board memory for graphics hardware—this property is key to large data visualization. Among others, sort-last parallel render systems include applications to distributed ray tracing, polygonal rendering, as well as volume rendering. Of particular interest to all such systems is the efficient implementation of the final compositing stage, which tend to become the limiting factor for many sort-last implementations. Several efficient parallelized compositing schemes as well as specialized hardware solutions were proposed, which are discussed in more detail in Section 5.3.4.

Further algorithmic details and specific implementation concepts, especially for sort-last parallel volume rendering on GPU clusters, are given in Section 5.3. In particular, data partitioning (Section 5.3.2), efficient rendering methods (Section 5.3.3), as well as concepts for load workload balancing (Section 5.3.6) are further discussed.

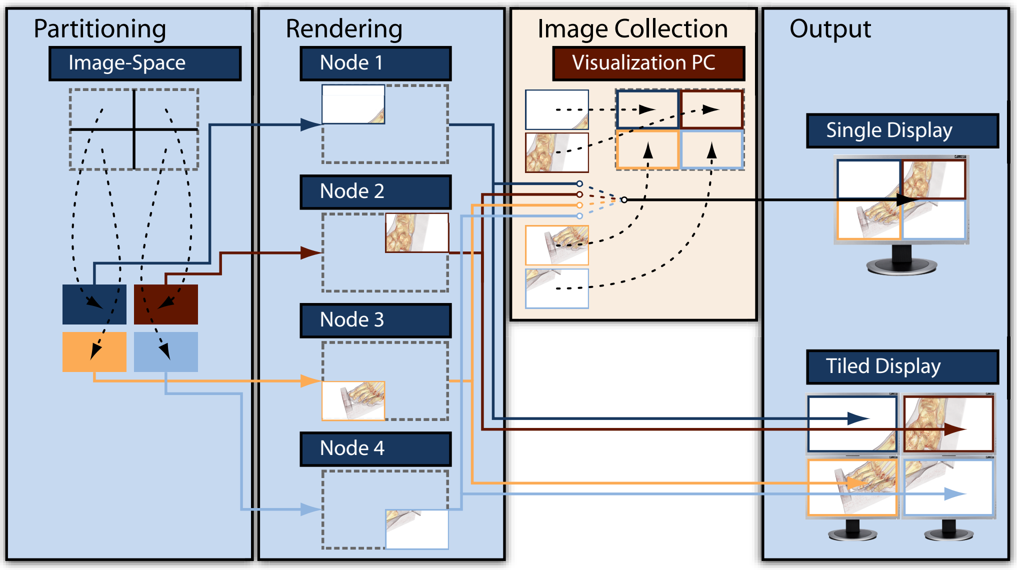


Figure 5.1: Simplified process layout for sort-first parallel volume rendering using four render nodes. Depending on the target output device the stage of image collection may be unneeded.

5.2 Parallel Sort-First Volume Visualization

5.2.1 Architecture Overview

Assuming direct access to the complete volumetric scalar field from each render node, i.e. data replication, the render pipeline for sort-first parallel volume rendering is straightforward (see Figure 5.1): Triggered upon user interaction, the scene parameters, like camera position or classification properties, are distributed across the cluster environment. Independently of each other, every node generates its designated sub-region of the final image by rendering the complete volume using an accordingly adapted view frustum—strictly, applying the partitioning not until completion of the transform and lighting stage (cf. discussion of multi-GPU systems in Section 5.1.1). Finally, the combination of all intermediate results form the actual rending result in the style of a mosaic. However, this final stage of image collection is often superfluous, thus marked optional in Figure 5.1, since actual screen partitioning often originates from the setup of the output device, e.g. the grid of a tiled display or a multi-projector wall. In such cases, the render nodes commonly drive a single element of the tiled display and actual image collection—including costly network communication—is avoided.

In contrast to those system setups, the focus of this thesis is directed towards cluster environments that do not feature a one-to-one correspondence of render

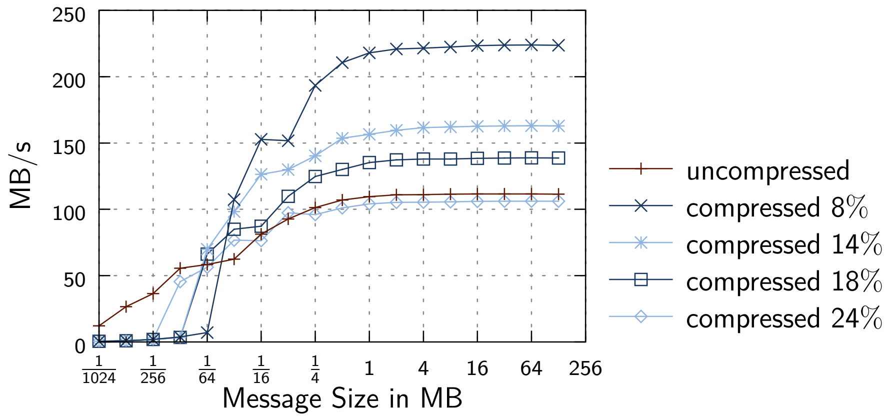


Figure 5.2: Analysis of communication performance transmitting over a Fast Ethernet interconnect with and without on-the-fly LZO compression; the percentages indicate the compression ratio—that is compressed versus uncompressed data.

nodes and output tiles. In particular, distributed volume rendering for visualization on a single remote desktop, i.e. an n-to-1 relation, as well as m-to-n setups with $m > n$, that is the number of render nodes exceed the number of output tiles. For both scenarios, communication of image data is mandatory and likely bounds the performance of a sort-first render architecture.

5.2.2 Image Collection

Especially in the context of remote visualization, image data—either the subimages or the already combined final render result—often need to be transferred via narrow-bandwidth interconnections. To increase total data throughput, a common approach is to encapsulate communication with real-time data compression, and decompression respectively. While on-the-fly data compression for network transmission is an own active field of research, with various software and specialized hardware solutions targeted for all layers of a network protocol stack, such generic approaches exceed the scope of this thesis; thus, the focus is limited to application-driven compression of image data for remote rendering only.

Motivated by the work of Ahrens and Painter [AP98], Ma and Camp [MC00] as well as Stegmaier et al. [SDWE03]—all studying various compression techniques especially for remote rendering in the context of cluster systems or hand-held devices—the distributed render systems described in the remainder of this chapter utilize the lossless *Lempel-Ziv-Oberhumer* (*LZO*) compression, more precisely variant *LZO1X-1*, for the transfer of image data over narrow-band interconnects. However, as indicated by the performance benchmarks in Figure 5.2, data reduction is only beneficial for reasonable large message sizes as well as data enabling sufficiently high compression ratios. Hence, compression is limited to a minimal message size of 64KB. In addition, compression may be disabled dynamically as soon as the current compression ratio falls beyond a given threshold; on such occasions, compression would be postponed after image collection and performed in parallel to the actual GPU-based rendering of the next frame to dynamically re-enable compression for coming frames if applicable. Even though such an approach was implemented experimentally in the context of the distributed volume render system described in [SMW$^+$05], it is not included in any benchmarks to avoid diluting measurements due to temporal effects of user input.

5.2.3 Load Balancing

Beside the need for image collection, the targeted cluster environments with an output correspondence other than a one-to-one relation between render nodes and output tiles allow for the distribution of workload in respect to the actual render complexity. Such optimal balancing of the render costs among all render nodes is crucial for efficient cluster utilization and is requisite for scaling to large scale cluster systems. In the following the most important techniques in the context of GPU-based volume rendering are discussed (see for example Whitman [Whi92] chapter 2.2.2 or Mueller [Mue95] for further discussions of sort-first load balancing focused towards polygon based rendering):

Static As initially proposed by Hu and Foley [HF85] even a static distribution of the screen space may be used for load balancing, if the fixed partitioning facilitate costly portions of the rendered scene to be assigned across many render nodes. Hence, a scan-line based distribution of workload to render processors in a round robin fashion was proposed; in the context of multi-GPU systems, such an approach was successfully adopted by 3Dfx's *scan line interleave* (*SLI*) mechanism, introduced with the *Voodoo 2* graphics accelerator—splitting workload by even and odd scan lines across two GPUs. However, scan line interleaving breaks the SIMD execution model of recent graphics architectures that process fragments at minimum in a 2×2 quad pattern to allow for operations being based on automatic gradient evaluations.

However, in the context of GPU cluster based volume rendering using ray casting a similar variant may be applied that was analyzed as part of this thesis during the development of a sort-first variant of the system described in [MSE06]. Instead of distribution of scan lines, the partitioning may be performed by splitting the volume's front faces in object space, on the basis of the polygonal distribution proposed by Chang and Jain [CJ81]. In the end, the front faces of the volume are split uniformly among all render nodes, whereas the constructed pattern may

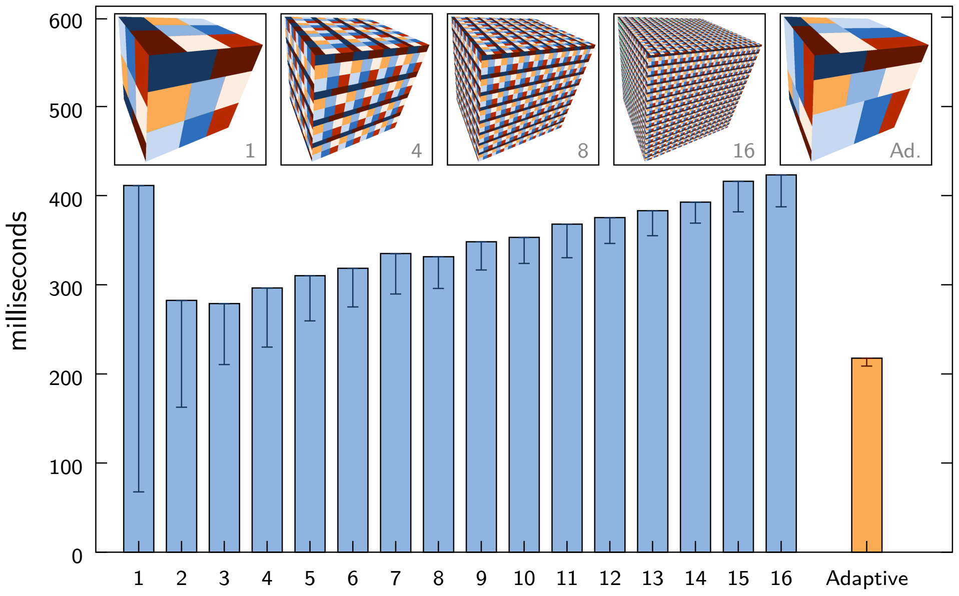


Figure 5.3: Comparison of static (blue) to dynamic (orange) load balancing mechanisms for sort-first volume visualization on 8 render nodes; timings denote the slowest/fastest cluster node (bars/error-bars). For the static cases, the x-axis states the number of pattern repetitions per front face. Image partitions for selected cases are shown in the inlets on the top (each color corresponds to a single cluster node).

be replicated repeatedly to increase evenly distribution among the cluster nodes. A benchmark series of such a static load balancing approach is shown in Figure 5.3. As expected the balancing becomes more optimal with increasing number of pattern repetitions, however, this only comes as costs of higher overall render costs due to the parallel evaluation of large pixel groups on the used NVIDIA GeForce 6800 Ultra graphics boards. Nevertheless, the static partitioning with three repetitions in each dimension offers significantly better workload balancing and consequently higher overall performance without introducing additional rendering or communications costs.

Scheduling Kaplan and Greenberg [KG79a] introduced an alternative technique for load balancing for sort-first render systems. They devise a static mesh to split the image space into tiles that are distributed to idle render nodes by a central scheduler. AMD's Crossfire technology enables the direct application of this approach to multi-GPU systems via the *SuperTiling* rendering mode. Screen-space tiles of 32×32 pixels are distributed between multiple GPUs. While offering very fine grained load balancing for arbitrary render applications, including volume rendering, the central scheduling process introduces high inter-frame communications costs. Opposed to multi-GPU systems with very low inter-GPU communication latencies, application to cluster environments are in general limited to significantly higher network interconnect latencies. Hence, such balancing approaches are only rarely used for parallel sort-first volume rendering in cluster environments.

Temporal Adaption To avoid costly inter-frame communication among render nodes, communication of render costs may be combined with the image collection stage—piggybacking the necessary data for load balancing to the transfered sub-images. Based on this information an optimized partitioning for subsequent frames—assuming temporal coherence between successive frames—may be derived by adapting the screen space distribution accordingly.

For multi-GPU setups, such a dynamic partitioning of workload is adopted by NVIDIA SLI systems running in *split frame rendering* (SFR) mode. The screen space it split into two horizontal stripes, whereas the actual size of the two subregions is dynamically determined with respect to the actual render costs. For GPU cluster systems, with applications to distributed volume rendering, Abraham et al. [ACCC04] presented an extended version of such a load balancing mechanism working with screen space tiles. Again, for the special case of volume ray casting, a new object space variant of such a system can be constructed according to the approach discussed for the static distribution paradigm—avoiding the more costly adaption of view frustums. Benchmarks results for such a technique are given in Figure 5.3 in comparison to the static workload partitioning.

Cost Estimation In a joint project with the Brendan Moloney, Daniel Weiskopf, and Torsten Möller another approach for achieving load balancing of sort-first volume rendering systems was analyzed [MWMS07]. Being based on an inexpensive cost estimate for the expected rendering complexity, the system allows to maintain load balancing without the assumption of temporal coherence and, hence, may adapt more rapidly to severe scene changes.

A separate render pass is added prior to volume rendering, which approximates or accurately evaluates the required sampling steps per pixel. Based on the collected information—estimate generation is executed in parallel—the optimal partition for the current frame can be constructed. While not assuming temporal coherence, the overhead for the additional render pass and, more importantly, the additional communication may hinder scalability; nevertheless, it does allow for more optimal load balancing compared to techniques based on heuristics computed from data from previous frames and, thus, may be favorable for some system setups.

An alternative of this approach would be to evaluate the cost estimation for the complete scene on all nodes in parallel to avoid any additional communication

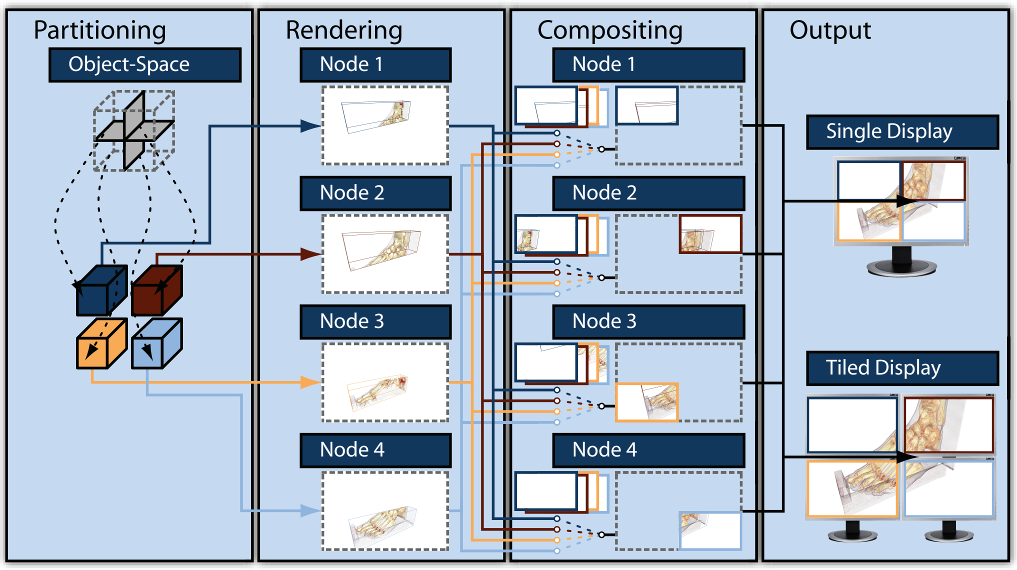


Figure 5.4: Simplified process layout for sort-last parallel volume rendering using four render nodes. For compositing purposes *Direct Send* is used as an example.

overhead. This might result in improved performances, especially in cluster setups with high latency interconnects and high render capabilities per cluster node.

5.3 Parallel Sort-Last Volume Visualization

5.3.1 Architecture Overview

Opposed to the commonly applied data replication of sort-first architectures, each node of a sort-last volume rendering system holds and processes only a subset of the input data field. This way the overall data size may by far exceed the capabilities of a single render node.

The image generation process operates effectively independent on each render node, producing a final rendering of the assigned data subset per contributing node (cf. Figure 5.4). As no static spatial relation between the assigned data block and the corresponding footprint of rendered pixel data in image space is given for an interactive visualization, each node's data may come to lie anywhere in the target view port. In particular, any arbitrary number of render nodes may contribute to a single output pixel element. In the context of volume rendering, this relates to splitting the ray integration into disjunct segments according to the data distribution. The overall rendering result per pixel, hence, is a combination of the concurrently evaluated ray segments.

However, the actual merge operation is tightly coupled to the designated visualization technique. For rendering opaque objects only—like non-transparent isosurface representations—simple depth comparisons suffice to reconstruct the final image. As such an operation is associative, the evaluation can be performed in any order. In contrast, dealing with transparent objects requires a depth-sorted combination of the transparent objects, according to the compositing equations (3.15) or (3.16) discussed in Section 3.1.1. More precisely, to comply with the prerequisite of an associative blending operation the alternative definition of the compositing equations using the "pre-multiplied" color notation [PD84] is required to obtain correct results.

Apart from the visualization technique's actual rendering costs, image compositing as well as data distribution play a major role for performance and imaging quality alike. The latter is of particular interest in combination with load-balancing strategies that redistribute workload, i.e. input data responsibilities, across the cluster setup and, thus, adapts the initial data partitioning on-the-fly. In the following the topics of data distribution, brick-based rendering, image compositing, and load-balancing are discussed in close relation to distributed GPU-based volume visualization.

5.3.2 Data Distribution

Subdividing a volume data set into a set of smaller data bricks is a widely used technique for rendering large data sets that cannot be processed as a whole at once. A technique, often also referred to as *out-of-core* rendering. Particularly for GPU-based volume visualization on a single GPU system, such an approach is commonly used to bypass texture memory limitations by processing bricks sequentially. For parallel environments, the partitioning serves the same purpose with the difference of processing each block concurrently on multiple render nodes to speed up the overall image generation process.

An ad-hoc data distribution may split the input data field into n equally sized bricks—assuming any data-driven metric, e.g. data size—for parallel execution on n render nodes. However, in general such a simple approach leads to severe load imbalances due to several observations: First, rendering cost may largely differ in object-space, i.e. parts of the data set may be evaluated quickly while others require extensive computational costs. Adding to this issue, optimization strategies, like early ray termination or empty-space-skipping, strongly add to such object-space imbalances. Second, rendering parameters such as view port settings and camera settings introduce unequal distribution of rendering costs in image-space. For example, render cost diminish completely for volume regions that are culled or fully occluded—again considering early ray termination.

In contrast, applying a very fine grained grid with the number of blocks by far exceeding the number of render nodes enables to tackle most of the aforementioned issues. By assigning spatially adjacent blocks to different rendering nodes render times on each node tend to become less affected by data characteristics as well as viewing parameters. However, as a multitude of bricks need to be rendered per cluster node the costs for brick setup, brick rendering, and brick border interpolation quickly become the limiting factors for the rendering stage. To alleviate the impact of decreasing brick sizes on the rendering performance the following Section 5.3.3 details on efficient rendering techniques for brick-based volume visualization. Apart from the actual volume rendering, the second drawback of such an approach is a severely increased workload for the compositing stage, as each convex group of blocks rendered on the same node requires a distinct compositing operation. Assuming optimal brick distribution across all nodes—that is, all *von Neumann neighbors* of any given brick b are assigned to different render nodes than b—a separate compositing operation per brick is required.

As pointed out by the two aforementioned extreme cases, static data distribution requires to find an optimum between the ability to enable fine-grained load balancing versus increased rendering/compositing costs for smaller bricks. In general, however, no such optimum can be specified prior to the actual rendering process, as view parameters highly influence the spatial rendering costs. To address this issue, dynamic load-balancing strategies adapt the distribution or workload on-the-fly during the visualization process. Optimized data distribution techniques supporting such adaptive mechanisms are discussed in more detail in the context of load balancing strategies for sort-last rendering in Section 5.3.6.

5.3.3 Brick-Based Rendering

Rendering a set of volume bricks on a single render node basically results in sequentially processing each block as a separate volume by the visualization techniques described in Section 3.2.3. To assure correct volume integration, the bricks are required to be processed in a depth-sorted order, which can mostly be directly derived from the grid data structure without significant computational overhead.

In a nutshell, bricking in general leads to a trade-off between the granularity of applied grid and the increasing processing costs for larger sets of bricks. For the latter, the involved overhead can be split into two categories: On the one hand, splitting the data set requires special attention at the brick borders to avoid rendering artefacts. On the other hand, bricking may shift the ratio of actual GPU render costs versus CPU time for feeding the render pipeline; consequently, such systems are susceptible to become CPU or CPU-GPU-bus limited quite easily. Both concerns are further discussed in the following two paragraphs.

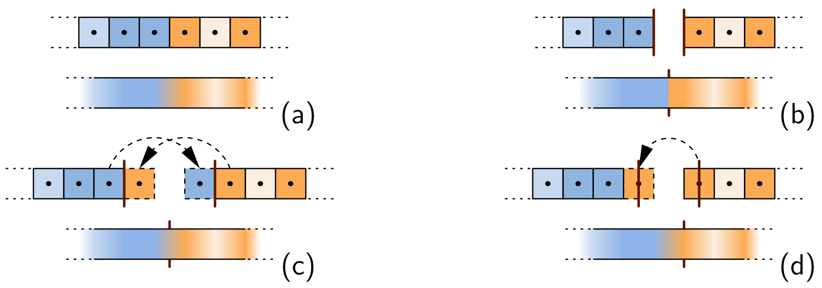


Figure 5.5: Comparison of interpolation techniques at brick boundaries showing for each case the input texels (top) as well as the linearly interpolated result (bottom): original input data (a), uncorrected bricking (b), two-sided overlapping (c), and single-sided overlapping (d).

Seamless Inter-Brick Sampling In order to guarantee artefact free rendering using a bricked volume representation the two criteria of a continuous interpolation as well as constant sampling distances across brick borders must be ensured. The latter is commonly achieved by maintaining a global sampling scheme and adapting the slice generation for each brick to this global pattern, i.e. assuring continuous slice geometry across brick boundaries.

By simply splitting the input data set into distinct bricks, the former requisite is unaccounted for—at least for any higher order interpolation scheme than point sampling (cf. Figure 5.5b). As an interpolation scheme of order n in 1D features an $n+1$ sized support, interpolation at brick boundaries miss up to n voxels from the neighboring block. Consequently, for the common case of tri-linear interpolation a single voxel overlap suffices to guarantee continuous interpolation. In a straightforward realization the overlap needs to be applied on all brick faces for every brick (cf. Figure 5.5c). However, by exploiting the fact that adding overlap to a brick guarantees the correct evaluation of samples up to the center of the first overlap voxel—that is the overlap voxel adjacent to the original brick data—the memory overhead can be reduced by a factor of two. In other words, by shifting the actual brick border by half a voxel size in the direction of the overlap, allows to obtain continuous interpolation by only adding the overlap on one side of the brick in each dimension (cf. Figure 5.5d). This model can also be extended to multi-resolution representations of bricked volumes as shown by Weiler et al. [WWH$^+$00], and was also utilized during the work of this thesis in the context of the parallel GPU-based visualization of hierarchically compressed volume data [SMW$^+$04].

Pre-integration, however, complicates such a simple model as both, the front and the back sample of each slab, are required for evaluation. As a consequence, an n-

voxel overlap does not suffice to correctly integrate across brick borders, since not all required data is available for slabs enclosing the center of first overlapping voxel. However, for any given upper bound of the maximum sampling distance, a minimal sufficient size of the overlap can be specified. Hence, assuming a reasonable upper bound of at least a single sample per voxel, an overlap of $n+1$ voxels is sufficient for an interpolation scheme of order n. For pre-integration, a single sided overlap is sufficient as long as the ray traversal direction in respect to the overlap location in a brick is known and taken into account for the rendering process. Nevertheless, such an approach at least doubles the required memory overhead for duplicated voxels and introduced an upper sampling limitation.

Aimed at solving the two mentioned drawbacks—thus, allowing better scaling in terms of memory overhead for large scale parallel volume visualization—an alternative approach was introduced in cooperation with Christoph Müller for a parallel GPU-based ray casting system [MSE06]. The idea is based on splitting each slab leaving the domain of a brick into the segment inside and outside this specific block. During processing this sub-volume, only the segment inside is accounted for, while the remaining portion of the segment is sequentially added while rendering the corresponding neighboring block—more precisely, the remaining segment may cross another boundary, but sequential decomposition always assures correct accumulation of the contributing segments.

Since both, the front and back sample of each slab, are thus defined to be at most aligned with the block boundary, no additional overlap compared to the simple non-pre-integrated case is required. From an implementation point of view, this alternative approach equals to start and stop volume integration exactly at the volume bounding faces—instead of the more commonly used approach of simply ignoring sampling segments smaller than the specified sampling distance. While the ray casting approach discussed in Section 3.3 inherently starts integration on the front faces, correct evaluation of the last sampling segment requires additional computation. In short, the actual intersection of each ray with the bounding volume needs to be evaluated, the volume sampled at that location, and the contribution of the corresponding ray segment added to the accumulated color[†]. As shown in [MSE06], the critical task of ray-volume intersection can be implemented efficiently on graphics hardware based on the Sutherland-Hodgman polygon clipping algorithm, requiring a total of 14 shader operations[‡]. Since this test needs to be performed only once after the actual volume rendering, the introduced com-

[†] As such segments vary in length in the range of $[0 \ldots d]$ with d being the sampling distance, an extended "three-dimensional" pre-integration table is required to account for different segment sizes [WKME03]. However, memory overhead for this table is independent of the granularity of the bricking and in general insignificant compared to the overhead introduced for bricking.

[‡] In the meantime, Klein [Kle08] further improved this code fragment to only 10 shader operations.

	CPU-based			GPU-based	
	BF	I	MCT	VS	VT
CPU Overhead	⊖⊖	⊖	⊕	⊕⊕	⊕⊕
plane-edge inters.	12	0	n	–	–
ADD/SUB ops	48/12	12/0	24+4n/8+n	–	–
MUL/DIV ops	72/12	0/0	24+6n/n	–	–
further processing	validate, sort	validate, sort	–	front vertex	template offset
CPU-GPU Transfer	⊖⊖	⊖⊖	⊖⊖	⊖	⊕⊕
render mode	interm.	interm.	interm.	interm.	vertex array
#vertices	3n float	3n float	3n float	12 int	–
GPU Overhead	⊕⊕	⊕⊕	⊕⊕	⊖	⊖
vertex overhead	–	–	–	9–15 inters.	–
fragment overhead	–	–	–	–	6 clip planes
Numerical stability	⊖	⊖⊖	⊕	⊕	⊕

Table 5.1: Comparison of various slicing techniques for volume rendering using view-aligned slices (evaluation costs are given in respect to a slice's number of vertices $n \in [3 \ldots 6]$): brute force (BF), iterative (I), based on a Marching Cubes table (MCT), vertex shader evaluation (VS), and via a vertex array template (VT).

putational overhead is negligible in respect to the actual volume sampling. Please note, that this stands in contrast to slice-based techniques that do not allow for such a simple and efficient implementation of this technique.

Lowering CPU Overhead for Volume Slicing Beside the growing memory overhead, increasing granularity of brick-based rendering techniques may also pose a severe overhead for the selected volume visualization technique. While the actual rendering costs for a single brick on the GPU scales down with the size of the brick, tasks that expose a constant overhead per brick, i.e. texture uploads or OpenGL state changes, and tasks that do not exhibit equal scaling behaviour, e.g. proxy geometry generation, emerge as limiting performance factors. For the technique of view-aligned slicing, the latter process of evaluating the slice geometry and sending the data down to the graphics card quickly becomes a critical performance issue, as observed in the distributed sort-last parallel rendering system described in [SMW$^+$04] or [MWMS07]. Thus, as part of this thesis several approaches to reduce the overhead for volume slicing in the context of parallel brick-based volume rendering were proposed. A brief comparison of the techniques described in the following is given in Table 5.1.

The *Brute Force* technique for generating a single view-aligned volume slice is a three-stage algorithm: First, each of the twelve lines defined by the volume edges is intersected with the plane equation of the designated slice. In general this results in twelve—in case the slice is parallel to a volume's edge only eight—intersection position. Second, these locations are validated, i.e. only those intersections are further processed that come to lie on an actual volume edge. Finally, the remaining intersections needs to be sorted in a way that they form a valid, none self-intersecting polygon or triangle fan that can be issued for rendering. While many implementation variants for this last step exist, the most commonly used approach is based on the property that each volume face features exactly zero or two intersections on its edges. Consequently, a valid sorting can be constructed by randomly selecting the first intersection and iteratively appending the intersection sharing the same face than the lastly added vertex.

Analyzed from a performance perspective, however, this brute force method poses a very high computational complexity on the CPU, since all twelve intersections need to be evaluated always and the sorting additionally introduces severe computational costs (cf. column 2 in Table 5.1). Beside that, the sorting approach may only result in correct slice if the stated assumption is fulfilled; this prerequisite might, however, fail rarely due to the inherent rounding of floating point operations, especially in cases of a slice being very close to a volume corner—effectively generating no valid or an incomplete slice.

To address at least the high computational costs for this approach, an often applied *iterative* variant avoids costly evaluation of plane-edge intersections during slicing. Assuming a constant slice distance for all slices in a brick, consecutive intersections on each of the twelve straight lines feature a distinct constant interval. Consequently, all intersections of a new slice can simply be obtained by adding these pre-calculated distances to the intersections of the previously rendered slice. The remaining stages of validation and sorting remain unchanged. Even though the costs for evaluation of the intersections are significantly reduced (cf. column 3 in Table 5.1), the costly sorting step may still hinder render scalability; more importantly, the benefits only come at costs of increased numerical instability—as errors are accumulated from slice to slice.

In order to account for speed and accuracy, another CPU-based variant for slice generation is derived from the Marching Cubes algorithm [LC87]. At first, only the corners of the volume are classified to lie in front or behind the designated slice. Based on a pre-calculated lookup table—actually a simplified *Marching Cubes table*, since only planar geometry needs to be taken into account—the intersected edges as well as their correct ordering can directly be obtained. Consequently, only the edges that truly contribute a vertex to the slice need to be intersected and costly validation as well as sorting is omitted completely (cf. column 4 in Table 5.1). As validation is performed implicitly during the evaluation of the marching cubes index, i.e. being based on volume's corners rather than the volume's edges, valid generation of slices is guaranteed.

Variants of this technique were developed independently by multiple research groups and were first published by Benassarou et al. [BBJL05]. In the context of this work a similar approach was firstly realized in the parallel render system described in [SMW$^+$05] to reduce CPU-based rendering overhead, after being designed in cooperation with Manfred Weiler and Frank Enders for improving the original slicing code for the volume node of the scenegraph system *OpenSG*[§].

Common to all aforementioned CPU-based approaches is the costly transfer of vertex data of each slice to the GPU. While GPU-based approaches to volume slicing can help to solve this bottleneck, they introduce additional workload for the GPU that may conflict with the actual volume rendering. However, depending on the used cluster environment, the computational power of the CPU versus the GPU, as well as the load on each processor posed by other tasks, graphics hardware based slicing solutions may offer a beneficial alternative in some cases.

Initially published in [EHK$^+$06], volume slicing may be evaluated completely in a vertex shader. The idea is based on issuing six dummy vertices per slice that become transformed to the designated slice vertices in the vertex shader, forming a valid, closed—albeit possibly degenerated—slice polygon. Each of the six input vertices process a fixed pre-defined order of edges for possible intersections—likly recalculating some edges twice as communication between the vertices is not possible due to the parallel execution model of the vertex shader. In detail, the scheduled tests—assuming the vertex v_0 to be closest to the camera—for the input vertices p_0 to p_5 are defined as follows (see Figure 5.6 and chapter 3.5.2 in [EHK$^+$06]):

$$\begin{aligned} p_0 &= e_{01} \text{ or } e_{14} \text{ or } e_{47} & \qquad p_1 &= e_{15} \text{ otherwise } p_0 \\ p_2 &= e_{02} \text{ or } e_{25} \text{ or } e_{57} & \qquad p_3 &= e_{26} \text{ otherwise } p_2 \\ p_4 &= e_{03} \text{ or } e_{36} \text{ or } e_{67} & \qquad p_5 &= e_{34} \text{ otherwise } p_4 \end{aligned}$$

For a single input vertex this leads to a worst case of calculating up to four plane-edge intersections; in total, for a triangular-shaped slice cutting edges e_{47},e_{57},e_{67} a worst case of 21 intersections need to be evaluated. However, the original scheme proposed in [EHK$^+$06] was further optimized in the context of this thesis to reduce computational complexity. While the specification of even input vertices remain

[§]The root cause of redesigning the slicing code fragment was a reported error causing incomplete slice representations while rendering a volume via a single slice—the error was actually caused by the numerical instability of the brute force algorithm.

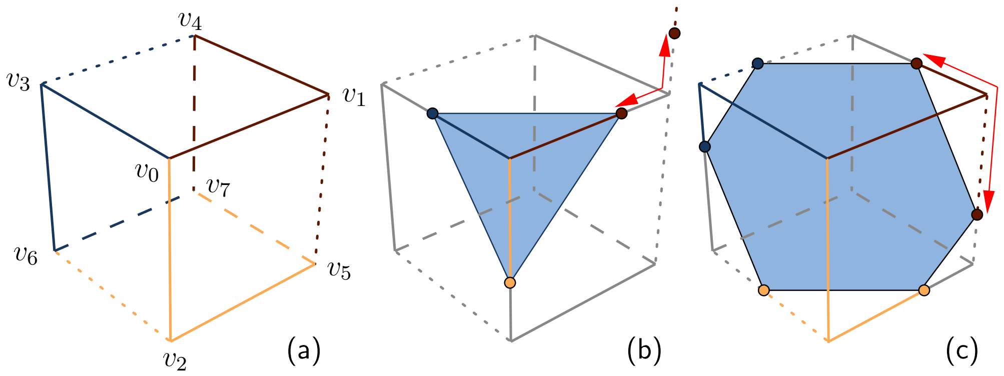


Figure 5.6: Edge scheduling for slicing evaluated in a vertex shader (a). Taking λ_{15} into account, the originally proposed scheduling can be further optimized: The two possible cases $\lambda_{15} \leq 0$ (b) and $\lambda_{15} > 0$ (c) are shown. Intersection locations λ_{26} and λ_{34} can be used analogously.

unchanged, the evaluation order of the inputs is adapted in the following way:

$$\begin{aligned} p_1 &= e_{15} \text{ otherwise } \lambda_{15} \leq 0 \ ? \ e_{01} : e_{47} \text{ or } e_{14} \\ p_3 &= e_{26} \text{ otherwise } \lambda_{26} \leq 0 \ ? \ e_{02} : e_{57} \text{ or } e_{25} \\ p_5 &= e_{34} \text{ otherwise } \lambda_{34} \leq 0 \ ? \ e_{03} : e_{67} \text{ or } e_{36} \end{aligned}$$

The λ_{ij} term denotes the location of intersection in the parameter space of the straight line formed by edge e_{ij}. The conditional test, thus, relates to distinguishing in which segment of the line, split by v_i, the intersection is located. By reusing this information from the first edge test, the remaining three candidate edges can be split into two subgroups (cf. Figure5.6(b) and (c)). Please note, that the conditional tests do not introduce additional overhead, as it can be implemented as part of the necessary intersection validation process (cf. code fragment provided in [EHK$^+$06]). With this adaption of the algorithm the worst case scenario is limited to a total of 15 intersection tests, which a maximum of 3 intersections per input vertex (cf. column 5 in Table 5.1). It is important to add, that the theoretical maximum of 3×6 intersections is efficiently avoided by the mirrored evaluation order in the false branch.

While the vertex shader approach was primarily designed for non-uniform shader architectures, i.e. the vertex stage is likely idle during the actual volume rendering and the additional overhead thus does not impact the overall performance, this assumption does not hold any longer for recent unified hardware implementations. Nevertheless, such an approach is still feasible in case CPU to GPU bandwidth mark the bottleneck, since this overhead can be reduced to two indices only per vertex—in comparison to sending three-dimensional positions. It also seems possible to utilize geometry instancing and transfer only the depth component via a shader uniform variable; however, such a mechanism was not further investigated so far. On a side note, it is also interesting to add, that the core idea behind this GPU slicing approach, i.e. the intersection test order, may also be used to improve the CPU-based brute force method.

Finally, in the context of bricked sort-first distributed render system proposed in cooperation with Brendan Moloney et al. [MWMS07] another GPU-based slicing variant for uniform data partitioning was introduced with the primary goal of minimizing the CPU to GPU data transfer per brick. Prior to the actual rendering, a single extended slice template is uploaded as vertex array. In detail, this vertex array corresponds to the generated slices for a single brick that is extended by one sampling distance along the view direction. The slightly enlarged template is necessary to allow for shifting the slice set about one sampling distance to enable constant sampling distances across brick borders. During the actual rendering of a brick via the template, six user-defined clip planes are enabled to crop the actual geometry of the brick from the enlarged template. Beside the additional overhead for the six clip panes, which only need to be adapted once per brick, data transfer is limited to issue the single draw call for the vertex array already residing in GPU memory. Consequently, such an approach proves to be quite optimal in terms of overall GPU-CPU transfer requirements (cf. column 6 in Table 5.1).

In comparison to the slice based rendering techniques, ray casting provides the benefit of very low setup costs per brick, i.e. only the bounding geometry needs to be rendered—posing not only low computation overhead to the CPU, but equally important also to the complete vertex processing stage of the GPU. In case of uniform bricks, thus, geometry instancing of a single bounding volume—with fixed texture coordinates attached—currently seems to be by far the fastest rendering method for large sets of bricked volumes. In the context of uniform as well as non-uniformly bricked sub-volumes, the low processing overhead for ray casting was successfully exploited for improving bricking scalability in [MSE06] and [MSE07] as part of this thesis.

5.3.4 Image Compositing

Reconstruction of the final image from the generated sub-images on each render node is a global process. That is, pixel data from every node need to be collected and merged, i.e. the contribution of all pixels of each partial image to the final image must be evaluated. The latter task of pixel merging effectively poses the difference in sort-first image collection versus sort-last image compositing, as the sub-images are not spatially disjunct and therefore multiple sub-images may contribute color information for a single pixel location.

Considering at first the naïve compositing of the full-sized sub-images, referred to as *sl-full* techniques [MCEF94], the total amount of input data scales linearly with the number of participating render nodes, i.e. a total of pn pixel elements needs to be processed, with p being the size of the viewport and n the number of render nodes. For object space partitioning, however, the actual area covered by rendering a node's portion of the input data in general shrinks with increasing number of render nodes, i.e. with decreasing amount of data assigned to each node. Only taking those areas into account leads to the more efficient class of *sl-sparse* techniques, that approximately requires to process a total of $pn^{1/3}$ pixel elements only, assuming a block data distribution according to Neumann [Neu93]. In either case, however, the overall costs increase with the number of nodes. As a consequence, scalability of image compositing can only be achieved, if this task benefits from parallelization itself. Hence, there was and is an ongoing research interest in efficient, parallel image compositing algorithms/implementations that improve scalability of sort-last architectures to large cluster environments.

While the next paragraph briefly introduces the most important image compositing schemes, the subsequent paragraph focuses on highly efficient implementations of the compositing operator itself. Both components are necessary to enable high performance distributed render setups.

Processing and Communication Schemes

An optimal image compositing algorithm can be coarsely defined as fulfilling the two primary criteria of distributing compositing workload optimally among all available processors while concurrently reducing the network communication costs to a minimum. For the latter criteria, a cost minimum is highly dependent on the actual network interconnect, like network bandwidth, communication latencies, and per-message overhead, but important key features to define a cost metric include the overall amount of data to be transferred, the number of sent messages, per-message sizes, and the used communication layout. For various system environments, hence, different image compositing algorithms were proposed.

One of the firstly applied parallel compositing scheme arrange the available processors in a chain: Pixel data is received from the previous node, merged with the local image portion, and passed to the successive compositing processor, effectively processing data in a *sequential pipeline* (cf. Figure 5.7). Parallelism of such methods is solely derived from processing the pixel data in a pipelined fashion, i.e. overall workload is split into a series of batches like scanlines and the compositing node process consecutive elements of the series in parallel. In an optimal case, rendering and compositing is tightly coupled and executed concurrently. Several specialized hardware implementations for high-performance image compositing were proposed based on this communication layout: The *PixelFlow* render architecture utilizes a static pipelined image compositing network with

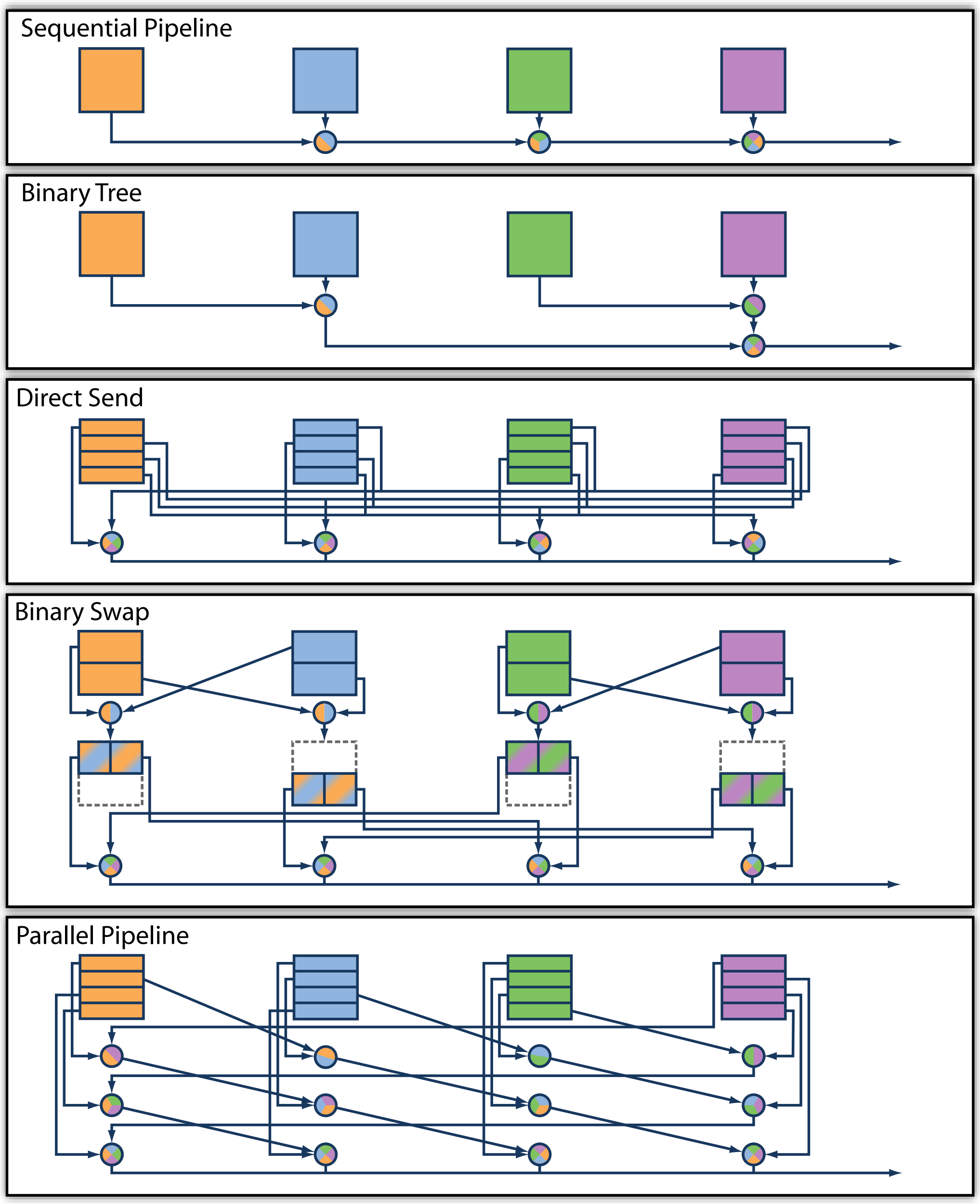


Figure 5.7: Comparison of communication layout and processing schemes for various image compositing techniques on the exemplary basis of four render nodes. Colors of the compositing stages (circles) denote contributing source nodes.

specialized compositing processors attached directly to the output frame buffer of each rendering/shading/frame buffer board [MEP92]. The *Lightning-2* [SEP+01] and *Metabuffer* [BBFZ00] compositing systems both work directly on the *Digital Visual Interface* (*DVI*) output of graphics hardware to avoid any costly transfer of image data from the GPU to the CPU after rendering—depth information are either encoded spatially or temporally to the digital image output. While such systems—driven by a fixed pipeline setup—are limited to depth compositing, the *Sepia* PCI adapter cards [MSH99, LMS+01] employ a dedicated network bus for image transfer that enables on-the-fly reconfiguration of the virtual compositing pipeline; hence also allowing for alpha compositing.

The *binary tree* image compositing scheme requires the same number of compositing stages, however, the tree layout inherently enables parallelism for all compositing processor on the same tree level [SGS91]. This way, the final results is obtained after $\log(n)$ compositing operations per input image (cf. Figure 5.7). Muraki et al. [MOM+01] built a custom compositing back-end for GPU-based cluster systems composed of PCI interface boards connected to specialized compositing hardware. The latter consists of $n-1$ image merge units to efficiently process image data in a binary tree layout.

However, for a cluster based implementation—with each render node acting also as compositing processor—utilization of parallelism is not optimal for the binary tree layout. More precisely, in the first stage only half of the processors are actually utilized in parallel. Subsequent stages cut the active number in halves, with only one processing node remaining in the $\log(n)$-th stage.

Direct send [Hsu93, Neu93] distributes compositing workload across all available processors by splitting each input image into multiple distinct regions. Each processor is assigned with a separate set of those regions. At first, image data is communicated between all nodes in a way that each node has local access to all data of its assigned image region (cf. Figure 5.7). In total, $n(n-1)$ messages need to be send with an overall data transfer of $(1-\frac{1}{n})pn^{1/3}$ pixel—again assuming the volume rendering is using a brick-based data decomposition. As all communication happens at once, i.e. each node sends/receives data from all others, network congestion might likely occur for most network environments. However, in case each node has either local or shared knowledge about the data distribution as well as the image decomposition for compositing, the n-to-n communication pattern can be reduced to only communicate data from rendering nodes that truly contribute to a target's node image region—as a rendered volume subset is likely not to contribute to all image regions for most view conditions. Such a technique was for example utilized by Peterka et al. [PYRM08] for large scale volume rendering on an IBM Blue Gene architecture.

Binary Swap [MPHK94] is an improved algorithm based on the binary tree layout—thus limited to power of two render nodes. Instead of combining the input from the two child nodes on a single compositing node, responsibility is divided among the two processors. Half of the input data is swapped between one another. As a result, each node generates half of the merged output image. Recursive application yields in each node holding $1/n$ of the final render result, while on each tree level all n nodes participate in the computation (cf. Figure 5.7). In comparison to the direct send paradigm, only $n\log(n)$ messages need to be send, but transfered data volume is slightly higher at approximately $2.43pn^{1/3}$. However, communication on each stage occurs in pairs, which might perform favorable in respect to the previously mentioned network congestion. To make the underlying compositing scheme applicable to arbitrary node configurations Yu et al. [YWM08] introduced the *2-3 Swap* compositing method, which starts operating on pairs of two or three nodes and continues to utilize all available nodes throughout the compositing stages. For the special case of power of two render nodes both techniques result in identical compositing sequences, making *2-3 Swap* a powerful generalization of *Binary Swap*.

Working with any arbitrary number of render nodes, the *parallel pipeline* [LRN96] concept employs n virtual compositing chains executed concurrently. Each pipeline processes $1/n$ of the final image by merging the local portion with the data already in the pipeline. The latter is transferred after each stage from node to node resembling a global shift operation—avoiding the occurrence of network link contention (cf. Figure 5.7). In total, $n(n-1)$ messages need to be send in $n-1$ compositing stages—higher in the number of stages opposed to direct send and higher in both, messages and stages, in comparison to binary swap. Nevertheless, based on the used network, i.e. message overhead vs. network congestion, parallel pipeline might perform favorably in some settings.

Combining fine-grained image space partitioning with an on-the-fly generated compositing schedule derived from the brick-based object space partitioning, *Scheduled Linear Image Compositing* (*SLIC*) [SML+03] was designed to further improve workload balancing for large cluster environments. The image space is decomposed into spans, i.e. segments of scanlines split by edges of the bricks' convex hulls that feature discontinuities in the number of overlaps. Using a global heuristic—locally evaluated on each render node—a consistent redistribution schedule is derived on a per frame basis. Following the direct send paradigm, the spans are transfered to the responsible render nodes for final compositing. Overall, about $n^{4/3}$ messages with an data volume of $(1-n^{-1/3})pn^{1/3}$ pixel need to be transmitted. Benchmark results indicate SLIC performs favorably especially for bandwidth-limited interconnects and/or high image resolutions.

Optimized Compositing Operators

Apart from an optimal compositing scheme for the targeted cluster environment, software compositing, i.e. image blending without specialized compositing hardware, requires efficient computation of the compositing operator on the available hardware resources. This is in particular the case in connection with low-latency, high-bandwidth network interconnects, like Myrinet or InfiniBand, since the impact of communication to the overall compositing execution times is significantly lessened. As a consequence, actual evaluation of the pixel compositing may ultimately limit performance—even more though, if targeting very high resolutions output devices.

While graphics hardware features specialized hardware for pixel compositing as part of the raster operations (see Section 2.2.1) or the GPU's parallel processor array may be easily utilized to evaluate even more complex compositing operators—exceeding the functionality of the fixed function blending stage—all such methods include the transfer of the pixel data from the CPU to the GPU and vice versa. More precisely, for each compositing task in Figure 5.7 the data communicated over the network interconnect needs to be transferred to the GPU, merged there with the local image data, and subsequently the result needs to be read back to the CPU for further processing. For an exemplary image resolution of 1920×1600 pixels solely the combined CPU↔GPU image transfer accounts for a total of 14.7ms[¶].

To avoid such costly transfer times and to allow for improved compositing performance, efficient CPU-based implementations of the alpha compositing operator used for parallel volume rendering were explored. For 8bits per color channel, the "pre-multiplied" over-operator is given as

$$A_i + \frac{(255 - A_\alpha)B_i}{255} \tag{5.1}$$

with index $i \in \{RGB\alpha\}$ denoting the color plus alpha components. An naïve combination of a pixel pair $\{A, B\}$ according to this equation is primarily dominated by the four integer divisions. For the above example of full HD input content, evaluation of this equation requires extensive 107.2ms on an AMD Q6600 processor—clearly being more expensive than GPU-based blending including the data transfer.

Firstly introduced by Magallón [Mag04], the costly division in equation (5.1) may be reformulated—assuming 8bit color components—in a more efficient way as

$$\frac{x}{255} = \frac{x + (x+128)/256 + 128}{256}, \tag{5.2}$$

[¶]Transfer times are given for a PCIe×16 slot using CUDA's `cudaMemcpy` from/to pageable memory, since such a method offers best performance at the time of writing; thus, timings may be considered a feasible lower bound for any GPU-based compositing method. Even though, page-locked memory can offer superior performance, the usage is inhibited as memory needs to be directly accessible to the network adapter to avoid further CPU-sided *memcpy* operations.

```
rex64 movd                    (%1), %%xmm0  // $x0_{[63,0]} = a1_{[31,0]}\ a0_{[31,0]}$
rex64 movd                    (%2), %%xmm1  // $x1_{[63,0]} = b1_{[31,0]}\ b0_{[31,0]}$

      mov        $0x00800080 , %%rax
      movd             %%rax , %%xmm2
      pshufd      $0 , %%xmm2, %%xmm2  // $x2_{[127,0]} = 00\,80 \times 8$

      pxor             %%xmm3, %%xmm3  // clear $x3_{[127,0]}$
      pxor             %%xmm4, %%xmm4  // clear $x4_{[127,0]}$

      punpcklbw        %%xmm3, %%xmm0  // unpack 8bit → 16bit: $x0_{[63,0]} \rightarrow x0_{[127,0]}$
      punpcklbw        %%xmm3, %%xmm1  // unpack 8bit → 16bit: $x1_{[63,0]} \rightarrow x1_{[127,0]}$

      pcmpeqd          %%xmm3, %%xmm3  // set $x3_{[127,0]}$
      punpcklbw        %%xmm4, %%xmm3  // unpack 8bit→16bit: $x3_{[127,0]} = 00\,FF \times 8$
      pxor             %%xmm0, %%xmm3  // $x3_{[127,0]} = 00\,FF$ - $x0_{[127,0]}$
      pshufhw     $0 , %%xmm3, %%xmm3  // $x3_{[127,64]} = 00\,FF$ - $a1_{[7,0]} \times 4$
      pshuflw     $0 , %%xmm3, %%xmm3  // $x3_{[63,0]} = 00\,FF$ - $a0_{[7,0]} \times 4$

      pmullw           %%xmm3, %%xmm1  // $x1_{[127,0]} = (00\,FF - a_{[7,0]}) * x1_{[127,0]}$
      paddusw          %%xmm2, %%xmm1  // $x1_{[127,0]} += 00\,80$
      movdqa           %%xmm1, %%xmm2  // $x2_{[127,0]} = x1_{[127,0]}$
      psrlw               $8 , %%xmm2  // $x2_{[127,0]} \gg 8$ (divide by 256)
      paddusw          %%xmm2, %%xmm1  // $x1_{[127,0]} += x1_{[127,0]}/256$
      psrlw               $8 , %%xmm1  // $x1_{[127,0]} \gg 8$ (divide by 256)
      packuswb         %%xmm1, %%xmm1  // pack 16bit → 8bit: $x1_{[127,0]} \rightarrow x1_{[63,0]}$

      packuswb         %%xmm0, %%xmm0  // pack 16bit → 8bit: $x0_{[127,0]} \rightarrow x0_{[63,0]}$

      paddusb          %%xmm1, %%xmm0  // $x0_{[63,0]} += x1_{[63,0]}$
rex64 movd             %%xmm0, (%0)    // copy result to memory
```

Listing 5.1: Blending operator implemented on the bases of SSE2 instructions to efficiently composite two 32bit RGBA color pairs in parallel. Code is given for GNU inline assembly in AT&T syntax.

which can be evaluated using `add` and `shift` operations only. This way timings significantly decrease to 25.9ms—already being competitive to GPU-based compositing techniques plus pixel data transfers. Further optimizations are possible by exploiting the available parallelism on recent processors offering both, data parallelism as well as task parallelism.

With the introduction of the *MMX* technology and its successors, modern CPUs became capable of executing instructions in a SIMD fashion. Technically, *MMX* extended instruction set enables operands of size 64bit, efficiently allowing for operating in parallel on a packed byte, packed word, or packed doubleword—representing eight 8bit characters, four 16bit words, and two 32bit double words

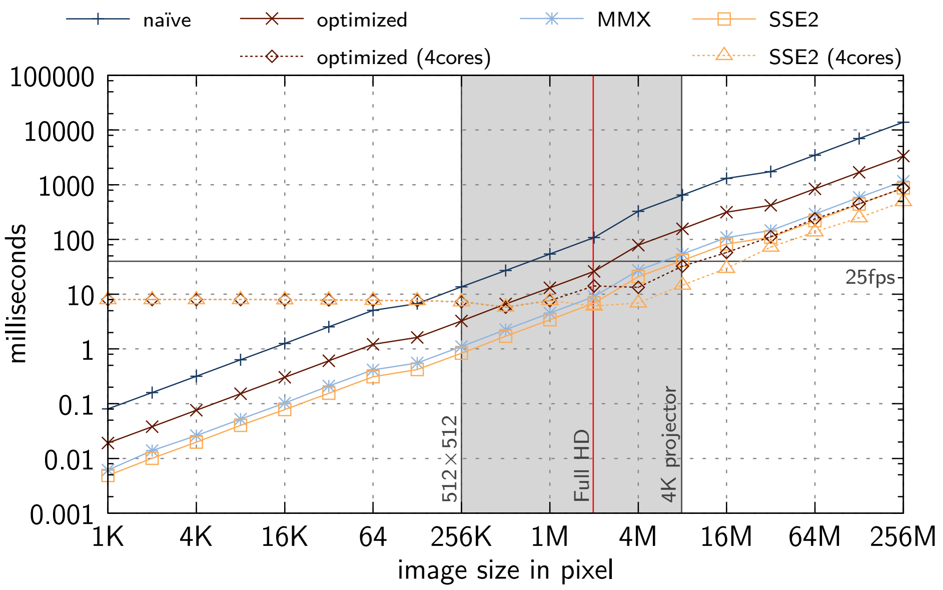


Figure 5.8: Compositing benchmarks for various implementation variants; compiled and tested on an AMD Q6600 processor using ICC version 10.1.012. The optimized SIMD variants offer interactive image compositing across the whole target range (highlighted in gray). The "4core" versions utilize a thread pool to distribute work across the quad-core processor.

respectively. Operations on packed types are executed in parallel on each of the basic typed data fields. Instead of executing each of the pixels' components sequentially, image compositing according to equation (5.2)—each pixel represented as packed word—can be evaluated for the complete RGBα tuple in parallel via MMX instructions. This approach was initially published in cooperation with Marcelo Magallón [SMW$^+$04] in the context of improving compositing overhead for sort-last volume rendering on a Myrinet connected cluster environment. Overall compositing times for the example case reduces to 9.1ms, being already faster than CPU$\leftrightarrow$GPU communication only.

With the introduction of the second version of the *Streaming SIMD Extensions* (*SSE2*) a full superset of the MMX integer instruction working on 128bit registers became available—effectively doubling the level of SIMD parallelism. Relating to image compositing, the MMX version can be extended to SSE2 to evaluate two pixel pairs $\{A0, B0\}$, $\{A1, B1\}$ concurrently. Compared to the 23 instructions of the MMX code (see Appendix A in [SMW$^+$04]), the extended implementation based on SSE2 instructions requires only a single instruction more—though blending double the pixel pairs (cf. Listing 5.1). By doing so, the overall computation time for composing two full HD input images further reduces to 7.3ms—about half of the time required for CPU↔GPU only. This SSE2 implementation was first published in cooperation with Marcelo Magallón, Stefan Guthe, and Daniel Weiskopf [SMW$^+$05] to effectively avoid blending computation limiting compositing speed on a 16-node cluster environment connected via Myrinet. A similar approach was later used by Cavin et al. [CMF05] to speed up compositing performance on a 32-node cluster system using a Gigabit interconnection.

In conclusion, Figure 5.8 summarizes compositing performances for the discussed implementation variants across a wide range of input image sizes. For the complete target range of input sizes, highlighted in gray, the optimized SSE2 implementation provides real-time performance, i.e. at least 25fps, up to blending two images suitable for 4K projectors. In addition, two variants are evaluated that utilize a thread pool to distribute the workload equally among the available cores of a multi-core processor. That way, slightly higher performances can be obtained for large data sizes. However, CPU threads introduce a significant performance overhead, which effectively lead to uncompetitive results for images smaller than full HD resolution. In either case, threaded and even single thread CPU blending enables high performance software image compositing for distributed render environments. In particular for GPU clusters, shifting compositing to the CPU allows to downsize data transfer pressure to the GPU and reduce resource conflicts between the actual rendering and the compositing stage.

5.3.5 Data Compression

Particularly targeted for good scaling with data set sizes, *sort-last* parallel rendering is commonly utilized to deal with visualization challenges by far exceeding the capabilities of a single desktop system, especially in terms of memory requirements. However, modern visualization tasks driven by advanced sensor/imaging technologies or derived from numerical simulation—likewise stand to benefit highly from parallelization—may even exceed the available hardware resources of a cluster environment. In the context of GPU-driven parallel volume visualization this is particular the case for texture memory, as this resource (used to) scale less rapidly compared to other key resources.

To address this issue and to manage very large visualization input, even parallel architectures may resort to the classical approach of out-of-core rendering techniques and/or incorporate data compression to further increase the manageable data volumes. While the former approach intrinsically deals with efficient (distributed) data storage, high performance disk utilization, as well as advanced data caching algorithms, the work of this thesis focus more on the latter approach of lossless data compression. In this context, the term “lossless” is defined as allowing to reconstruct and render the original input data set on request—even

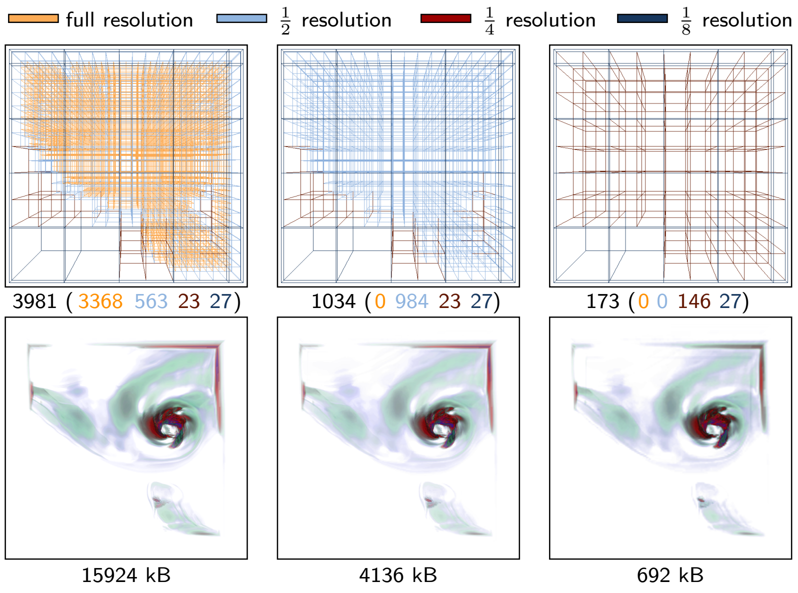


Figure 5.9: Comparison of different levels of data compression for the second timestep of the hurricane Isabel data set[||]. The uncompressed volume is of size $512\times512\times128$ and requires a total of 32M texture memory to be rendered. The leftmost setting is visually equivalent, but requires less than half the memory footprint.

though, rendering may selectively choose to use a more coarse, i.e. less accurate, representation of the original data in favor of faster image generation.

Following the approach introduced by Guthe et al. [GWGS02] for large volume visualization on a single desktop PC, a transformation of the input data set into a compressed hierarchical representation by the means of a wavelet pyramid is performed in a preprocessing step; this way, an octree structured data representation is constructed: First, the input data set is split into equally sized cubes of m^3 voxels, which serve as starting point for the pre-processing. Recursively, eight cubes sharing one corner are transformed at a time using linearly interpolating spline wavelets. After transformation, the resulting low-pass filtered portion is a combined representation of the eight input cubes with half the resolution of the original data—again the size of the low-pass data is m^3 voxels. Recursion continues as described with this data until the complete volume is represented in a single block. This cube forms the root node of the hierarchical data structure; except for the root node, all other nodes of the hierarchical structure hold high-pass filtered data only, which is compressed via an arithmetic encoder [GS01] to save on

[||]The Isabel data set is courtesy of Wei Wang, Cindy Bruyere, Bill Kuo, and others at the National Center for Atmospheric Research, US.

the disk/memory footprint of the constructed data representation (see Figure 5.9). While such an approach fails to guarantee a fixed compression ratio—in worst case there might even not be any compression at all—typical observed data reduction for volumetric input data is in the range of 1:4 to 1:8.

During rendering an adaptive decompression scheme is used that depends on the viewing position as well as the characteristics of the actual data set itself. Starting at the root node of the hierarchical data structure, a priority queue determines the order of sub-blocks in which the volume is scheduled for further decompression. Depending on the ratio between the resolution of a volume block and the actual display resolution, regions closer to the viewer are more likely decompressed than others. This view-dependent criterion is coupled with an error metric describing the discrepancy between two representations of different quality; that enables to identify regions that can be rendered in low quality without noticeable artifacts. A typical example for such areas are empty regions around a specimen in medical CT scans. The priority queue is processes as long as either a given time has elapsed (assuring a lower bound for the update rate), the front-most block satisfies a given quality threshold (assuring a defined level of image quality), or a maximum amount of uncompressed data per node is hit (in case in-core rendering must be assured).

After the data reconstruction is finished, the decompressed blocks are transferred to the graphics board's texture memory for rendering. Depending on the quality level of each reconstructed block, the number of slices used for rendering varies accordingly. With increasing reconstruction quality the number of slices increases as well, delivering higher quality for areas with high resolution data. Vice versa, rendering overhead is saved on regions with less high-frequency details in the reconstructed visualization data.

By incorporating such a data compression approach, the parallel volume rendering architecture built up in cooperation with Marcello Magallón, Daniel Weiskopf, and Stefan Guthe [SMW$^+$04, SMW$^+$05] allows for rendering the full Visible Human Project's 8bit cryosection data set—sized $2048 \times 1024 \times 1878$ for an overall data volume of about $3.67GB$—on an eight node cluster system with a total of $8 \times 128M$ texture memory. Please note, that rendering can be performed in-core for most view conditions using the adaptive decompression technique—without introducing noticeable reduction of image quality compared to the image of the uncompressed data set[**]. The system was benchmarked on cluster environments using Gigabit-Ethernet, Myrinet, and InfiniBand interconnects; in particular for the latter system, the bottleneck of image generation could be successfully shifted from the actual rendering stage to image compositing, allowing for interacting

[**]During animations slight notions of popping artefacts might become noticeable when blocks change their decompression level.

with the aforementioned data set with more than 11.4/6.5 fps on a $512^2/1024^2$ viewport.

While such an approach can help to significantly reduce the memory footprint required during rendering and, hence, still enable in-core rendering for very large data sets, the main draw back of the discussed approach is the likely congestion of the memory bus to the graphics hardware. As decompression is performed on the CPU, all changes in actual displayed data need to be communicated to the GPU memory—which is likely to become the system's bottleneck frames exposing only low coherence. Recent research utilizing the high flexibility of modern GPUs is directed towards on-the-fly decompression, for example based on Adaptive Texture Maps [KE02] or hierarchical vector quantization [SW03] in combination of deferred filtering [FAM+05]. Via such techniques the bus bottleneck can be reduced or even avoided, however, again at higher rendering costs.

The presented technique was also extended to time-varying data sets [SMW+05]: Each time step is compressed independently, while rendering might choose to keep blocks from previous time frames, in case they expose sufficient temporal coherence in order to save on transfer bandwidth. This system served as basis for combining the discussed data compression scheme with a *Time Space Partitioning* data structure [SCM99] in order to improve image quality, data compression, as well as rendering performance. Closely related to this thesis, this work was done by Felix Müller in the context of his diploma thesis [Mül05].

5.3.6 Load-Balancing Strategies

As already pointed out in Section 5.3.2 in the context of data distribution, load-balancing for the sort-last paradigm is primarily driven and/or limited by the underlying local data distribution, i.e. the local accessibility of data on each node. The following classification tries to identify and compare the primal load-balancing mechanism for sort-last volume rendering on the basis of different data distribution schemes. For the remainder of this section, let b be the number of sub-blocks the input data set is split to, while n denotes the number of render nodes.

Static – Coherent This category includes the special case of $b = n$ as well as any distribution for $b > n$ that satisfies for every render node that all assigned sub-blocks form a single convex compound. By definition, the latter case is hence interchangeable with the former one by simply merging all sub-blocks on each node to a single super-block. The example of the ad-hoc data distribution in Section 5.3.2 is a candidate of this class and variants of that approach were applied to the rendering systems described in [SMW+04, SMW+05], which both are based on the initial architecture introduced by Magallón et al. [MHE01].

Balancing strategies in this category are solely limited to the data-centric metrics, i.e. the uniform data distribution across all nodes. However, as actual render costs in general only poorly correlate to the data size—in particular in connection with adaptive rendering techniques or view parameters that exhibit extensive frustum culling—load-balancing is severely limited. By incorporating knowledge of view parameters to the metric, such as the transfer function, data distribution may be adapted in a way that allows for load-balanced rendering in connection with optimizations like empty-space skipping. However, this only holds true if the applied view parameters do not change or data distribution is required to be recalculated upon view setting changes. Hence, such approaches are only applicable to special visualization cases, but do not apply to the general paradigm of unconstrained data exploration.

Static – Interleaved The second category includes any arbitrary distribution of sub-blocks to render nodes for the case of $b > n$, whereas the assigned blocks to a single node do not build a single convex set. To improve on load-balancing, blocks get commonly distributed in an interleaved fashion—that is spatially co-located blocks are assigned to different render nodes. Under the assumption that rendering costs are spatially coherent, load-balancing can be improved for any adaptive visualization technique or for arbitrary view conditions[††]. The main drawback of such balancing approaches is the increased workload imposed on image-compositing as each convex sub-block needs to be handled in a separate compositing step. For GPU-based rendering retrieving the sub-images after rendering a single data brick might already pose the highest overhead as switching states from rendering to data retrieval and vice versa tends to be very costly.

Candidates for this category of load balancing strategies are, for example, described by Wang et al. [WGS04] and further extended by Gao et al. [GWLS05] for a multi-resolution distributed volume rendering architecture based on octree balancing via space-filling curves [CDF$^+$03].

Static – Replicated The final static load-balancing category differs from the previous ones as it allows for a single data sub-block to be assigned to multiple render nodes. This way, workload can be dynamically balanced by shifting render responsibilities across nodes that have local access to the same sub-block. Please note, even though there is a dynamic component to the load balancing mechanism in this case, the actual category is classified to be static as there is a fixed, static data distribution assigned to all nodes at the beginning of the rendering process.

[††]Load-Balancing is effective as long as the visualization is not restricted to show less than n sub-blocks. In this case, a more fine granularity of sub-blocks may by applied to improve even load distribution. This in contrast to the class of *static – coherent* methods, that only enables some extent of load-balancing if the complete data set is shown.

While such an approach can offer superior load balancing compared to both previous methods—as no assumption on spatial coherence is made—those techniques are only rarely used (at least for for volume rendering), as the duplicated data storage stands in direct contrast to the scaling behaviour of sort-last parallelism. However, if the combined available memory in a cluster system exceeds the requirements of the visualization task, such an approach may be used effectively to utilize on such extra resources.

An example of such a balancing strategy is given by Samanta et al. [SFL01]. Their system is capable of arbitrary k-way replication of polygonal data for load-balanced distributed rendering in a GPU cluster environment. It's interesting to note, that in case each block is replicated to all nodes, denoted as sub-category static–replicated–full in this classification, data set scaling is actually impeded and the sort-last system inherits the characteristics of a sort-first architecture.

Dynamic Apart from statically assigning volume sub-blocks to render nodes, the physical distribution of sub-blocks might also change over time in order to adapt to occurring load imbalances. The key characteristic of the class of dynamic techniques is that actual input data is communicated between render nodes or from a central data storage as block assignments change. In return of this communication overhead, the load-balancing is effectively unrestricted and is theoretically capable of always resulting in an optimal workload distribution for a given sub-brick granularity. However, even though algorithmically unrestricted, resource limitation, i.e. local storage capabilities, might still put a limit of the possible data relocation per node.

Independently of each other, Marchesin et al. [MMD06] as well as the architecture build up in cooperation with Christoph Müller [MSE06] utilized a *binary space partitioning* (BSP) tree to mange the distribution of sub-bricks to the render nodes. The leaves of the BSP-tree represent the portions of the data set assigned to a distinct render node, which also implies those bricks are locally present on the corresponding cluster node. Based on the render times of the previous frame or other render time estimates, the load-balancing stage may shift the BSP-tree boundaries to strive for a more optimal data distribution. On the data level, such a shift triggers the physical exchange of data.

For the system described by Marchesin et al. the receiving node requests the data from a central, multi-layer cached file server, while the sending node might simple discard the data. In our case, the payload data is transfered via a direct network communication between the two nodes. As in both cases data exchange is a very costly operation, the systems utilize a least-recently-used (LRU) caching mechanism for sub-blocks per cluster node to avoid the overhead of communicating the same bricks back and forth multiple times. To some extent such caching

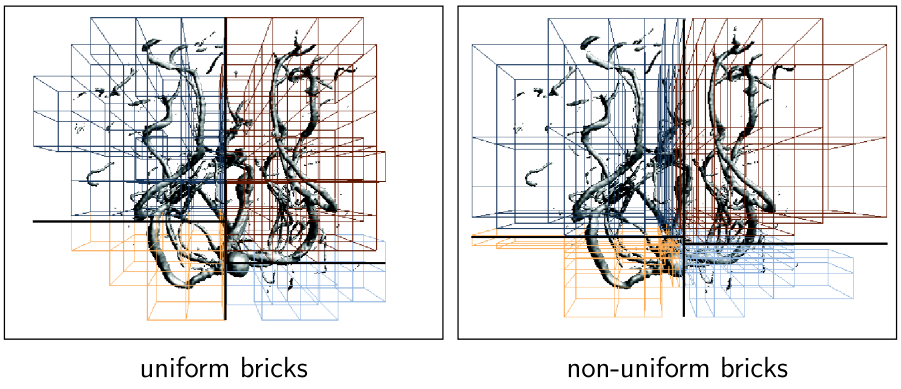


Figure 5.10: Comparison of load balancing results based on uniform versus non-uniform bricks. The differences can be best seen on the left part of the volume[‡‡], as the optimal split in this scenario lies close to the center of a uniform brick. Consequently, in case of non-uniform bricks this optimum can be approximated more precisely.

approaches may, hence, be seen as selective replication mechanism.

As shown by the evaluation of both systems, the discussed dynamic load balancing strategy is efficiently able to adapt to varying view parameters and to cope with highly varying distribution of rendering costs in the spatial domain. On the other hand, dynamic balancing introduces a high communication overhead and in some cases this can result in a temporary drop of performance for frames that actually execute a reallocation of bricks; however, multi-threaded communication of sub-bricks parallel to the actual render thread might help to address this issue, but was not available at that time for both described systems.

A second drawback for the TSP-based distribution is that always a complete front of bricks need to be transfered between two nodes, hence, the granularity of load balancing is quite large. In order to enhance granularity and to allow for improved workload distribution, the described system was further extended to non-uniform bricks. This follow-up work was again carried out in cooperation with Christoph Müller [MSE07].

The actual construction of the sub-bricks in the non-uniform scheme is changed to be a two phase process: First, a TSP-tree is generated that represents the estimated optimal distribution according to a data-centric metric, similar to the static–coherent approach—though without having a sub-block grid underneath. Second, based on this TSP-structure the data set is subsequently split into sub-bricks in a way, that bricks closer to a node transition are smaller than the ones in the center of a TSP-tree leave node or at the overall volume boundary. This way,

[‡‡]The Aneurysm data set is courtesy of Michael Meißner, Viatronix Inc., US.

the granularity of load balancing is increased at the most likely areas, while overhead for increased number of bricks is saved on areas unlikely to ever participate in the redistribution phase of the load balancing mechanism.

Performance benchmarks of this system shows that non-uniform bricks can effectively improve the discussed load-balancing technique. Visual comparison of uniform versus non-uniform balancing is given in Figure 5.10. However, it is important to note that rendering non-uniform bricks may introduce other side effects on rendering performance. In the described system, the actual overhead for rendering non-uniform bricks was negligible, however, efficiency of some rendering optimizations, i.e. empty space skipping on a sub-brick basis, is significantly decreased due to the enlarged grid cells in the center of the TSP-tree leave nodes. Thus, the overall benefit of the dynamic load balancing based on non-uniform bricks is highly dependent on the applied rendering technique.

Chapter 6

Distributed General Purpose Computations on Graphics Clusters

Continuing the previous chapter's efforts in leveraging graphics clusters to tackle large-scale visualization challenges, this chapter focuses on utilizing the very same GPU cluster architecture for large-scale, non-graphics computations; more specifically, non-graphics computations that do take advantage of the parallelism of modern graphics processors for general purpose computations.

6.1 GPGPU Programming Environments

Processing non-graphics tasks on GPUs has been the focus of research ever since graphics hardware evolved into programmable and, thus, more flexible parallel processors. Due to the pure graphics centric programming interfaces, however, general purpose computations had to be translated to the graphics paradigm of fragments, vertices, and triangles; prominent examples following this approach, include linear algebra operators [KW03b], cloud simulation [HBSL03], or photon mapping [PDC$^+$03]. Spurred by such developments, completely new programming models evolved, which are detached—at least from a programmers point of view—from the traditional rendering pipeline policy, and therefore help to address the existing *Programming Gap* [MZ08, MZD09] by providing new interfaces specifically tailored for parallel programming of general purpose computation on GPUs.

6.1.1 Low-Level Hardware Description Languages

Primarily triggered by the hardware vendor's focus towards high performance GPU computing, low-level interfaces to graphics processors—more precisely their Data Parallel Processor (DPP) arrays—were made publicly available. Opposed to the strict data-centric view of the graphics pipeline, such specialized programming environments directly expose access to the underlying parallel hardware architecture. Being closely coupled to the actual hardware abstraction layer, naturally all of these interfaces are highly vendor specific.

AMD provides a set of low-level programming interfaces, all combined in the *Stream Computing Software Stack*: Building upon ATI's hardware abstraction layer, named Close-To-Metal (*CTM*) [PSG06, AMD06], programming access starts as low as the GPU-specific Instruction Set Architecture [AMD07d]—primarily made public in the context of Open Source device drivers. The next higher level in the software stack is formed by the *Compute Abstraction Layer* (*CAL*) [AMD07a], that exposes a pseudo-assembly Intermediate Language (*IL*) [AMD07c]; being actually a generalization of the graphics hardware specific instruction set, *IL* is designed to provide compatibility to a variety of platforms, i.e. across graphics hardware generations but also multi-core CPUs. Conceptually, this programming interface, thus, mimics hardware access to AMD DPP arrays equivalent to the *x86* assembly language for the *x86* class of CPU processors.

NVIDIA's low-level programming interface is based on *PTX* [NVI08], a combination of a *parallel thread execution* virtual machine and a virtual Instruction Set Architecture. Aimed to span multiple GPU generations and to provide a generic data-parallel compute interface for graphics processors, the virtual machine provides a low-level abstraction layer of the native machine-specific GPU architecture; specifically designed for general purpose parallel programming the virtual instruction set, thus, is completely machine-independent. In comparison, *PTX* and *CAL* expose conceptually quite similar programming environments for general purpose computations on their respective instance of a Data Parallel Processor.

Compared to both, graphics APIs and high-level GPGPU languages, the low-level interfaces currently enable an extended feature set and are in particular of special interest in the context of code tuning for high performance computing.

6.1.2 High-Level Programming Languages

Obviously, for application development high-level programming environments come with the clear benefits of fast learning curves, good code readability, and often an adequate abstraction from the actual hardware architecture. Depending on the hardware access layer on which the high level of abstraction is built upon, the available environments can be split into two classes.

Graphics Pipeline based Initially driven by academic research focused on leveraging the GPU's compute capabilities for non-graphics purposes, the first class of GPGPU programming environments was apparently restricted to the only interface for accessing graphics hardware, which was provided via graphics APIs.

Published as the first of such systems, the *Sh toolkit* [MQP02] basically consists of a *C++* library that includes a run time mapping of special *C++* user code to graphics interfaces. Originally designed for the *SMASH* interface [McC00], additional back-ends for OpenGL ARB shaders and OpenGL GLSL shaders were made publicly available. Further development lead to the commercialized RapidMind *Multi-core Development Platform* that additionally supports multi-core CPUs as well as the *Cell Broadband Engine*. Analog to *Sh*'s paradigm of translating data-parallel operations to GPU shaders at run time, Microsoft Research's *Accelerator* project [TPO06] maps a *C#* host language to DirectX pixel shaders to provide access to general purpose computations on GPUs.

Based on the *Brook* language specification [Buc03]—originally designed for the Merrimac streaming supercomputer [DLD+03]—*Brook for GPUs* [BFH+04b] exposes an *ANSI C* programming environment with streaming extensions. In the initial version of the toolkit the extended *C* syntax was compiled to *Cg* fragment shaders to map to the graphics hardware; additional compiler back-ends were added subsequently, including OpenGL/DirectX shaders as well as ATI's low-level *CTM* interface. The system continued to be developed by PeakStream, which was later on acquired by Google. Additional key features of the commercialized product include a highly optimized back-end targeting multi-core CPUs as well as run time compilation and run time optimization akin to the concept introduced by *Sh*.

Low-Level Interface based Closely coupled with the release of the low-level hardware interfaces, each hardware vendor also introduced its own high-level programming environment specifically designed for GPGPU application development; ultimately, turning away from the paradigm of the graphics pipeline and, thus, eliminating the intrinsic limitations of the prior approaches, e.g. scattered writes or direct memory management instead of data storage solely via textures.

With the *Compute Unified Device Architecture* (*CUDA*) [NVI07] NVIDIA provides a *C* environment that works on the NVIDIA G8x and all currently available newer NVIDIA hardware generations. It is based on top of *PTX* and is designed to maintain access to most of the underlying hardware concepts. In particular, the parallel thread execution hierarchy—organized in *threads*, *warps*, *blocks*, and *grids*—as well as the accessible memory hierarchy are closely tied to the underlying hardware architecture; ultimately, handing over the complete control on how an algorithm maps to the actual hardware to the user—being a complex task on one hand, but also enabling highly efficient hardware utilization on the other hand.

In contrast, AMD devised the concept of *Brook for GPUs* and released an adapted and slightly enhanced version, called *Brook+* [AMD07b], as part of its *Stream Computing Software Stack*. As with the original *Brook* language, code is written on a highly abstract level, effectively hiding any hardware specific characteristics; as a consequence, the automatic mapping of the user code to *IL* at compile time plays a major role for high performance.

In order to establish a more general, vendor-independent compute interface and to enhance interoperability to existing graphics APIs both, Microsoft and the Khronos Group, recently defined their own GPU computing interfaces—which also indicates the rising importance of having compute interfaces to modern graphics hardware. In case of DirectX, this is exposed through *compute shaders* that are an integral part of DirectX 11. In case of OpenGL, the Khronos Compute Working Group has defined an open-standard compute programming specification named Open Computing Language (*OpenCL*) [Khr09]. Designed with the idea of being generally applicable to any parallel compute device—including CPUs, GPUs, and FPGA-based accelerator boards—publicly available OpenCL implementations currently exist for AMD and NVIDIA GPUs as well as for multi-core CPUs.

6.2 Compute Unified Device and Systems Architecture

All programing paradigms for GPU computing implicitly feature a specific language design targeted for efficiently utilizing the highly parallel execution architectures immanent to modern graphics hardware. In line with that, a suitable algorithm for efficient GPU processing is required to feature a very high level of parallelism, exposing in the best case hundreds or thousands of independent threads. This characteristics is key to high-performance in GPU computing and inherently enables an excellent scaling behavior with the number of execution units. In fact, scaling is already a crucial issue in targeting a single GPU, as various instances of even the same hardware generation already expose a highly variant number of processing units; for example, instances of NVIDIA's G8x architecture scale from the low end of 16 stream processors (G86) up to a maximum of 128 stream processors (G80). As a consequence, this implicit property strongly suggest scaling even beyond a single GPU's processing power; even more so, as extending parallelism may also be used to scale other limited hardware resources—most importantly the available memory pool.

However, extending GPU compute applications to higher levels of parallelism, i.e. multi-GPU systems and GPU cluster environments, usually involves changing development paradigms and also comes along with introducing additional, often highly different, programming interfaces for each new level of parallelism. Besides the peculiarities of GPU computing, the developer is additionally confronted with CPU thread management, network communication, process synchronization, and so forth.

As part of this thesis, an alternative approach was studied that extends the existing paradigms of GPU compute environments to additionally cover these higher levels of parallelism. The primary goal was to provide a consistent development interface—based upon an already existing and prevalent programming paradigm—that transparently handles the parallelism intrinsic to GPUs, among multiple GPUs in a single system, and across GPU cluster environments. The implemented system is based on the *CUDA* programming environment, however, the basic paradigms presented could also be adapted to match other GPU compute interfaces, e.g. *Brook+*. This work was first published in cooperation with Christoph Müller and Carsten Dachsbacher [SMDE08] and was later on extended in further cooperation with Steffen Frey [MFS$^+$09].

6.2.1 System Overview

The *Compute Unified Device and Systems Architecture* (*CUDASA*) programming environment consists of four abstraction layers as depicted in Fig. 6.1 from left (top layer) to right (bottom layer). Each of the three lower levels addresses one specific type of parallelism: The lowest utilizes the highly parallel architecture of a single graphics processor, while the next higher level builds upon the parallelism of multiple GPUs within a single system. The third layer adds support for distributing program execution in cluster environments and enables parallelism scaling with the number of participating cluster nodes. Finally, the topmost layer represents the sequential portion of an application which issues function calls executed by exploiting the parallelism of all underlying abstraction levels. Each layer exposes its functionality to the next higher level via specific user-defined functions which are declared using the extended set of function type qualifiers implemented in CUDASA. These functions are called using a consistent interface across all layers whereas each call includes the specification of an execution environment, i.e. the grid sizes, of the next lower level.

GPU Layer The lowest layer (see Fig. 6.1, right) simply represents the unmodified CUDA interface for programming GPUs as a highly parallel, multi-threaded processor via an extended C language. One of the primary design goals for CUDASA was to maintain full compatibility to code written in plain CUDA. That is to allow for existing CUDA code to be used as part of a CUDASA project without any modifications—even more, this code is intended to serve as a building block for higher levels of parallelism. Especially, in the context of extending an exiting CUDA application to support multi-GPU systems or GPU cluster environments, development can solely be focused on the higher levels of parallelism without the need to further adapt the existing code base. It also proved to be beneficial to support a common build setup no matter if the application currently is targeted for a single GPU using CUDA mechanisms only or if the application utilizes the extended feature set of the CUDASA environment.

Bus Layer The second layer (Fig. 6.1) abstracts from multiple GPUs within a single system, for example *SLI*, *Crossfire*, *Quad-SLI* configurations, or single-box

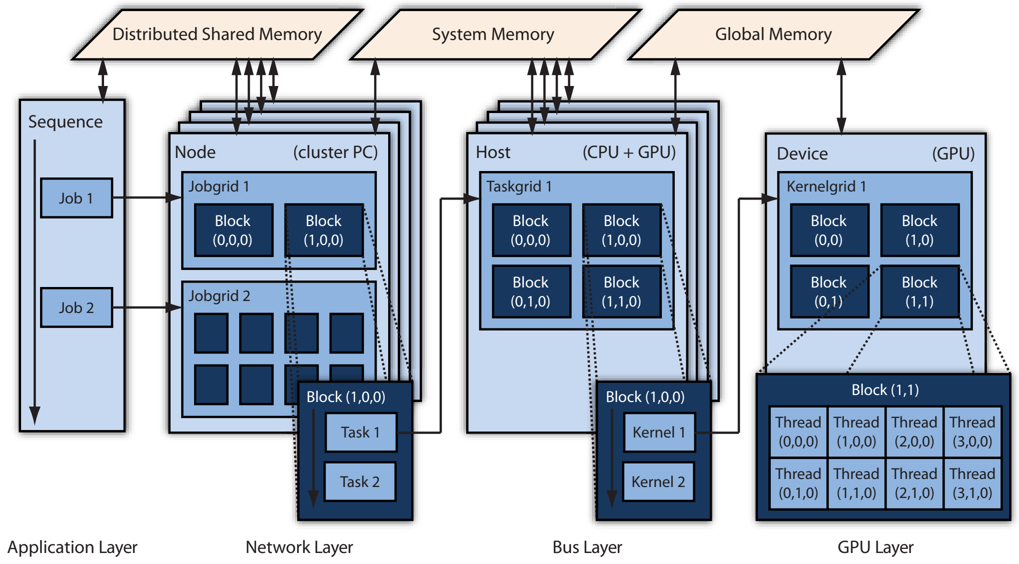


Figure 6.1: Schematic overview of all four abstraction layers of the CUDASA programming environment. The topmost layer is placed left, with decreasing level of abstraction from left to right.

setups based on the *QuadroPlex* platform, or *Tesla* computing systems respectively. A CPU thread together with a GPU forms a basic execution unit (BEU), denoted *host* on the bus layer. The programmability of these BEUs is exposed to the programmer through *task functions*, which are the pendants to *kernel functions* of the GPU layer. We follow the execution model of CUDA and define that a single call to a *task* consists of a grid of distinctive blocks. A scheduler distributes the pending workloads to participating *hosts* and also handles inhomogeneous system configurations, e.g. systems with two different GPUs or different number of physical PCIe lanes to the GPUs. The scheduling process works transparently to the user and the desirable consequence is that the application design is completely independent of the underlying hardware. For example, a once compiled CUDASA program is able to fully exploit the parallelism of a *Tesla* S1070 computing system system by executing four kernel blocks in parallel, while it processes the same blocks sequentially on a single-GPU system.

The specification of CUDA's *kernel grid* and the corresponding *kernel blocks*—in particular, the assured independence of a block execution with respect to all others blocks of a grid—would also allow for an alternative distribution of workload on the bus level. Instead of introducing a separate grid/block structure for the *task functions*, one could think of a *kernel grid* that automatically distributes its *kernel blocks* to all available GPUs in the compute system. The main benefit of such an approach is that it can be fully embedded in the existing CUDA language semantics; thus, no change of code would be required at all—allowing the parallelism to be applied to all existing CUDA code completely automatically on the CUDA driver level or upon recompilation as an additional compile option. However, this elegance only comes as costs of replicating all CUDA memory to all participating graphics cards, as any block can possibly access arbitrary data from global memory. In addition, atomic functions, as introduced with CUDA *compute capability* 1.1, would either require costly communication between all participating GPUs, or could only be guaranteed to work for the subset of blocks executed on the same graphics processor—which might be non-deterministic to enable load-balancing. Therefore, CUDASA introduces the overhead of an additional grid/block structure for various reasons: First, it allows to execute completely inhomogeneous *kernel functions* with independent grid and thread granularity in parallel on a multi-GPU system. Second, such an approach offers the flexibility to design algorithms that also scale in terms of the available texture memory. And finally, it gives the user control over the subsets of blocks assured to be executed on a distinct card; thus enabling to utilize atomic functions separately for such distinct sets. Even though not supported yet, it would be desirable to emulate the described alternative approach within CUDASA in order to directly support bus level parallelism out of the box in case none of the additional CUDASA functionality is required.

While the main focus of CUDASA is to provide easy access to multiple GPUs, the *bus layer* is also able to delegate its tasks to CPU cores. This allows to use CPU cores—in parallel if available—for tasks of a CUDASA program which cannot be executed on GPUs or for which the user prefers the execution to happen on CPUs. Tasks, both with and without GPU support, can be used together in arbitrary combinations. It is also possible to use CPU cores to emulate a system with multiple GPUs by using the built-in device emulation provided by CUDA. The user specifies the operation mode (CPU only or CPU+GPU) for each task at compile time and optionally defines a maximum number of parallel devices to be used.

Network Layer The third layer adds support for clusters of multiple interconnected computers. Its design is very similar to the underlying *bus layer*: A single computer, called *node*, acts as the BEU of the network layer and all nodes process blocks of the *job* grid (issued through a *job function*) in parallel. Again, the scheduling mechanism takes care of distributing the workload, in both homogeneous and inhomogeneous environments.

The difference to all underlying layers is that the *network layer* has to provide its own implementation of a distributed shared memory model in software. The distributed memory provides means to transfer data between blocks of a *jobgrid* and successive *jobs*. It can be considered as the pendant to the *global memory* in CUDA, which is used to transfer data between blocks and kernels, or the shared system memory that allows communication between *task blocks*. However, in contrast to GPUs this memory does not exist as an ”on-board component”, nor does a system-wide shared memory necessarily exist. Thus, each node makes a portion of its system memory available to the distributed shared memory pool. Having the global address space that the shared memory exposes to the programmer scattered over all nodes, access to this memory can be more or less costly depending on whether the requested data is already resident on the requesting node or not. Therefore, the CUDASA scheduler is capable of data locality-aware block scheduling of a *jobgrid* in order to minimize the required network load for costly shared memory accesses.

Application Layer The topmost layer describes a sequential process which issues calls to *job functions*. It also takes care of the (de-)allocation of distributed shared memory which holds input and output data and is processed by the *nodes*. The distributed shared memory enables the processing of computations which exceed the available system memory of a single *node*. It also provides the means for communication between the *sequence* and the *jobs* as well as between multiple *jobs*. A single *job* in turn can communicate with the *tasks* it spawns simply via system memory and each *task* communicates with the GPU by acting as a normal CUDA *host* and transferring data via *global memory*.

6.2.2 Programming Model

In order to seamlessly integrate the described extensions to the CUDA development environment the CUDASA system consists of three main components: A runtime library, a minimal set of extensions to the CUDA language, and a self-contained CUDASA compiler.

Runtime Library The runtime library splits into two segments providing the CUDASA internal system functionalities as well as a user-space interface for CUDA capabilities that were extended to the higher levels of abstraction. The former include mechanisms like the automatic scheduling of *job* and *task* distribution as well as the distributed shared memory management for *the network layer*. Such functionalities are solely intended to be used internally by the CUDASA system, i.e. the functions are only accessible from the context of the generated code of the CUDASA compiler. In contrast, the latter part of the runtime library exposes directly callable functions exposing CUDA concepts common to the *bus* or *network layer*. This includes atomic functions and synchronization mechanisms that are defined analogously to the corresponding CUDA counterparts; adding a `cudasa` prefix, but otherwise trying to completely mimic the original CUDA function definitions. In order to avoid introducing any dependencies to network libraries, i.e. the necessity of having a MPI implementation installed on a single-box environment, the CUDASA library is built with and without network layer support. While this mainly affects library dependencies it is important to note, that for both cases the linked application is capable to run completely sequentially on a single node setup as well as utilizing the available parallel architecture on each level without recompilation.

Language Extensions The extensions to the original CUDA language solely introduce additional programmability for the higher levels of parallelism while the syntax and semantics of the *GPU* layer remain unchanged. Hence existing CUDA code does not require any manual modifications and can be used with CUDASA directly.

For each new layer (*bus*, *network*, and *application layer*) CUDASA defines a set of function type qualifiers to specify a function's target BEU and its corresponding scope visibility. This is in line with the existing CUDA qualifiers `__device__` and `__global__` for functions executed on a graphics device and `__host__` functions acting as the front-end for CUDA device functions. Table 6.1 lists corresponding CUDA as well as the newly added CUDASA keywords. As indicated there, each layer also introduces specific built-in variables holding block indices and dimensions (Table 6.1, right column), each accessible to functions of the corresponding and the underlying layers.

In addition, CUDASA needs a way to link the abstraction layers and define function calls to the respective next-lower layer. Again, CUDASA follows the original CUDA interface and generalize its concepts to the higher levels of abstraction: A CUDA function call requires the *host* level to specify an *execution configuration* which includes the requested grid and block sizes for the parallel execution on the GPU. In an analogous manner, functions of each layer are allowed to call the exposed functions (see Table 6.1) of the next underlying layer. In order to maintain a consistent interface, the CUDA-specific parenthesized parameter list (denoted with `<<< ... >>>`) is also used for the specification of the *execution configuration* of all newly introduced abstraction layers.

The newly added extensions to the CUDA language are limited to a minimal set of new keywords. However, they provide powerful control over all levels of additional parallelism and enable the tackling of much more complex computations while keeping the additional programming and learning overhead for the user very low. Specifically, programming CUDASA *job* and *task functions* is very

Abstraction	Exposed	Internal	Built-ins
application layer		**`__sequence__`**	
network layer	**`__job__`**	**`__node__`**	**`jobIdx`**, **`jobDim`**
bus layer	**`__task__`**	`__host__`	**`taskIdx`**, **`taskDim`**
GPU layer	`__global__`	`__device__`	`gridDim`, `blockIdx` `blockDim`, `threadIdx`

Table 6.1: The extended set of function type qualifiers of CUDASA; new keywords are printed bold-face. Internal functions are only callable from functions within the same layer, while exposed functions are accessible from the next higher abstraction layer. Built-ins are automatically propagated to all underlying layers.

similar to CUDA *kernel functions* with respect to distributing the workload. All communication-related tasks are completely hidden from the user and are covered by the CUDASA run time library and the compiler.

Optionally to the described extensions so far, a developer might provide additional information to the CUDASA system in order to allow the network scheduler to automatically optimize the network communication overhead. This optimization process in targeted to minimize the necessary data transfers with respect to the distributed shared memory usage by assigning *job blocks* by virtue of data locality. In more detail, a *job function* should optimally be assigned to the *node*, which already holds most of the block's required data in its local segment of shared distributed memory; enabling to access the data from the *node's* local memory opposed to requiring remote memory accesses paired with costly network communication. While the CUDASA system intrinsically keeps track of the data distribution of the distributed shared memory at run time, it is impossible to automatically predict a block's memory accesses to the distributed memory; neither by code analysis during compile time nor at run time prior to executing the block, since memory access locations and sizes can be calculated dynamically. Therefore, an additional set of CUDASA language extensions provide the optional possibility to specify the required data for an efficient run time evaluation. In case the programmer knows the access pattern of all *jobs* to the distributed shared memory before the start of the *job grid* execution—which is by design common for many distributed algorithms—the newly introduced scheduler mechanism can use this information to improve scheduling in terms of remote network communication costs; and consequently improve overall execution times. Table 6.2 summarizes the complete set of additional keywords for this optimized scheduling mechanism. Even though, this functionality goes beyond the original semantics offered by CUDA, the concept can be added with a small set of changes to the original programming language; Sec. 6.2.4 explains in detail how those keywords

Abstraction	Function Qualifier	Parameter Qualifier	Keywords
application layer			
network layer	`__descriptor__`	`__nomap__`	`using`
bus layer			
GPU layer			

Table 6.2: Additional set of keywords introduced to CUDASA in order to enable optimized network scheduling in respect to communication costs for the distributed shared memory.

are integrated into the syntax and semantics of CUDA.

CUDASA Compiler The last component of the CUDASA programming environment is the self-contained compiler which processes CUDASA programs and outputs code which is then compiled with the standard CUDA tools (Fig. 6.2). Although regular expressions could possibly handle the new set of keywords, such an approach is not suitable for the translation of CUDASA code to the underlying parallelization mechanisms. This requires detailed knowledge of variables types and function scopes and can only be obtained from a full grammatical analysis. The code translation process is described in detail in section 6.2.3 for the bus layer and in section 6.2.4 for the network layer.

CUDA itself exposes the C subset of C++ to the programmer, while some language-specific elements rely on C++ functionality, such as templated texture classes. CUDASA needs to act as a pre-compiler to CUDA including the ability to parse the header files of CUDA. Consequently, the CUDASA compiler needs to cope with the full C++ standard to translate the new extensions into plain CUDA code. We opted for building our compiler using *Elkhound* [MN04], a powerful parser generator capable of handling the full C++ grammar, and *Elsa*, a freely available C/C++ parser based on *Elkhound*. We extended the compiler to support all CUDA-specific extensions to the C language as well as the extensions described in the previous paragraphs. The compiler takes pre-compiled CUDASA code as input and outputs source code, which is strictly based on CUDA syntax without any additional extensions. This means that the additional functionality exposed by CUDASA is translated into plain C code with additional references to the CUDASA runtime library.

6.2.3 Bus Parallelism

The goal of bus parallelism is to scale processing power and the total available memory with the number of GPUs within a single system. For this, a *task* needs

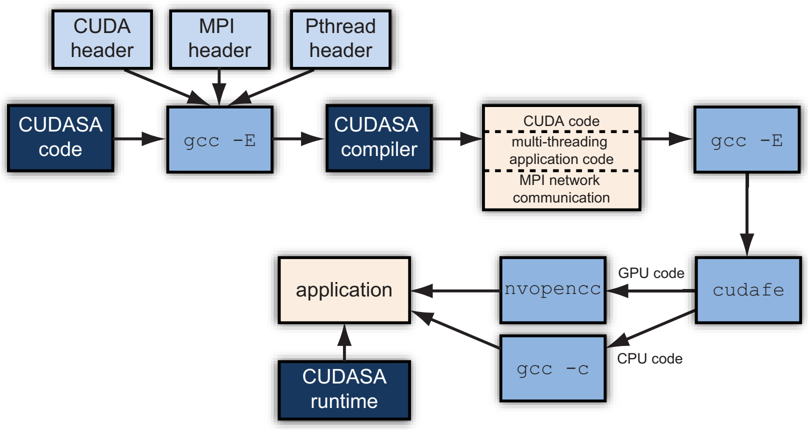


Figure 6.2: The CUDASA compiler processes a program and outputs standard CUDA code but also generates multi-threading and network code providing higher levels of parallelism. The CUDA compiler separates GPU and CPU code and hands it over to the corresponding compilers (nvopencc and any standard C/C++ compiler). The generated application executable, together with the CUDASA runtime library, is able to distribute workload across clusters and GPUs.

to be executed in parallel on multiple graphics devices, i.e. blocks of a *taskgrid* are assigned to different GPUs. CUDA demands a one-to-one ratio of processes or CPU threads to GPUs by design. Thus, each BEU of the bus layer has to be executed as a detached thread. Practically speaking, a *host* corresponds to a single CPU thread with a specific GPU device assigned to it.

Task Scheduling Calling a *task* triggers the execution of the *host* threads and initializes the scheduling of the *taskgrid* blocks. A queue of all blocks waiting for execution is held in system memory. Idle *host* threads process pending blocks until the queue is empty, i.e. the execution of the complete *taskgrid* is finished. Mutex locking ensures a synchronized access to the block queue, provides the necessary thread-safety, and also avoids a repeated processing of blocks on multiple BEUs. The block-threads are organized using a thread pool in order to keep the overhead for calling a *task* at a minimum. In particular, this is also important to avoid the costly initialization of CUDA for every function call.

A polling mechanism achieves load balancing on the block level across *hosts* as the actual execution time for each block implicitly controls how many of them are assigned to each BEU. This does not guarantee deterministic block assignment, but it does guarantee parallel execution, even for inhomogeneous setups, as long as enough pending blocks are left in the queue.

Compile Process The automatic translation of code using the CUDASA interface into code which can be executed in parallel by multiple CPU threads needs to map the CUDASA syntax and semantics to the underlying concept of parallelism using POSIX threads. For each *task function* a corresponding wrapper function is generated that translates the function parameters into a POSIX compatible wrapper structure and spawns the desired number of parallel threads; all issued functions calls to the original *task function* are modified to call the wrapper function in a standard C compatible way—instead of using the CUDA-specific *execution environment*. In more detail, these wrapper functions perform the following steps:

- Copy the original function parameters into the wrapper structure.
- Populate the queue of the scheduler with all blocks of the *taskgrid* as specified in the *execution environment* of the corresponding function call.
- Annotate each block with its distinct set of built-in variables.
- Wake up BEU worker threads from the pool.
- Wait for all blocks to be processed as executing a *taskgrid* is defined to be a blocking call.

Additionally, the signature of the original *task function* is modified internally to accept the wrapper structure as single parameter. The actual function parameters as well as the necessary built-ins are then reconstructed from the data structure prior to executing the function body.

The following example demonstrates the code transformation of a CUDASA *task function*. From the function definition, written in CUDASA as

```
__task__ void tfunc(int i, float *f) { ... }
```

the compiler first generates a corresponding internal wrapper structure holding the necessary built-in variables for the *bus layer* as well as the function's original parameters in a single POSIX compatible data structure:

```
typedef struct {
    int i; float *f;                          // user-defined
    dim3 taskDim;                             // built-ins
} wrapper_struct_tfunc;
```

Subsequently, the original function's signature is adapted in accordance to the new concept of parameter passing; thus, matching the POSIX requirements for the start routine of the thread creation process. Reconstruction of these parameters is automatically added to the start of the function. The original function body is further enclosed by the mechanism to poll the block queue for the next pending block—provided by the CUDASA runtime library

```
void tfunc(void *params) {
    wrapper_struct_tfunc *tfunc_params;
    tfunc_params = (wrapper_struct_tfunc*) params;
    int     i = tfunc_params->i;            // reconstruct parameters
    float *f = tfunc_params->f;
    dim3 taskIdx;
    dim3 taskDim = tfunc_params->taskDim;   // reconstruct built-ins
    while (_cuaKernelGridNext(&taskIdx)) {  // query next task block
        { ... }
    }
}
```

The resulting code after the transformation is plain CUDA code, and thus, can be passed into the standard CUDA compile tool chain.

Atomic Functions CUDASA also adds support for atomic functions on the bus parallelism level to enable thread-safe communication between multiple *task* invocations. The implementation of those atomic functions is primarily based on the locked instruction `xaddl` and `cmpxchgl` in *x86* assembler code. Built from these basic instructions the atomic functions introduced with CUDA *compute capability* 1.1 are provided with matching syntax and semantics by the CUDASA runtime library.

Scalability and Performance To compare the performance, the efficiency, and the overhead of CUDASA generated code to other parallel execution environments, i.e. multiple CPU cores and the intrinsic parallelism of a single GPU, single precision general matrix multiply (*SGEMM*), a subroutine of the level 3 BLAS library standard, is used for benchmarking. This particular test was selected to assure meaningful comparability across different hardware platforms, as vendor specific performance-optimized implementation are available for all targeted hardware platforms; namely, Intel's Math Kernel Library 10.0 (MKL), AMD's Core Math Library 4.0 (ACML), and NVIDIA's CUBLAS Library 1.1 were analyzed in comparison to a CUDASA *SGEMM* implementation.

The CUDASA version of *SGEMM* is based on the CUBLAS library function, that servers as building block for the implementation on the *GPU layer*. In fact, the *kernel function* does not differ at all from the single-GPU variant solely based on CUBLAS. For the higher level of abstraction, the *bus layer* distributes the workload among the *task blocks* by splitting the input matrices into sub-regions: Each *task block* is responsible to compute a distinct, non-overlapping sub-matrix of the output. Conceptually, this parallelization scheme is analogous to the mechanism already used by CUBLAS to map the SGEMM calculation to the *GPU layer* [NVI07].

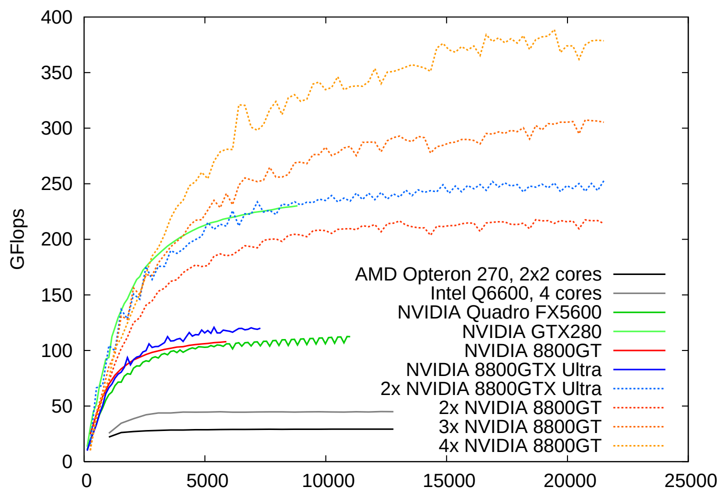


Figure 6.3: Benchmark results providing insight to the SGEMM scaling behavior for various system configurations and matrix sizes (x-axis). The multi-GPU setups (dotted lines) are based on the CUDASA software stack for distributing workload on the *bus layer*.

Figure 6.3 summarizes the results for the CUDASA *SGEMM bus* parallelization on multiple GPUs (dotted colored lines) compared with the above-mentioned CPU implementations (solid gray lines) as well as single-GPU benchmarks (solid colored lines). The result for the multi-GPU systems demonstrate excellent scaling behavior for both test setups: two 8800GTX Ultra (blue lines) and up to four 8800GTs (red lines) cards, especially for large problem sizes. In the former setup, a speedup of 1.95 is achieved when comparing the pure CUBLAS variant running on a single GPU to the CUDASA implementation using two cards. In the latter case, distributing the work over all four cards results in a speedup of 3.60. Further investigation showed that even better scaling for this setup is primarily hindered by the motherboard's physical layout of the PCIe lanes, which offers a 16/4/4/4 connection only; subsequently, leading to a non-linear scaling of the communication overhead across the GPUs.

6.2.4 Network Parallelism

The network layer is very similar to the bus layer, not only conceptually, but also regarding its implementation. *Jobs* are the pendants of the *tasks* on the bus layer; with the main difference that *jobs* are not executed by local operating system threads, but across different cluster nodes. For that, the parallelization on the network level, i.e. all network communication, is based upon the MPI-2

Standard [Mes96]. One node of the communication group acts as head node to the system, executing the sequential part of the application—including the issue of *job functions*—whereas all other nodes operate as the BEUs of the *network layer*.

Compile Process Analogous to the *bus layer*, each CUDASA *job function* definition as well as the corresponding function calls need to be transformed into CUDA compatible output code. In terms of function parameters, the concept of a single wrapper structure, including the required built-ins, is again applied in order to serialize the data; thus, enabling function parameter transmission via MPI communication. However, the remaining challenge is to issue the function call from the head node on the remote BEUs. Therefore, during the compile process each user-defined *job function* is assigned with a unique function identifier. In parallel, the compiler generates an event loop for the BEUs that is immediately executed at run time on all cluster nodes, except the head node. Basically, the event loop waits for broadcast messages from the head node, i.e. notifications to execute a *jobgrid* or memory operations on the distributed shared memory. In case of a *jobgrid* execution, the broadcast messages contain the unique function identifier enabling to remotely execute the issued function call. Thus, with respect to executing a *job function*, the necessary code transformation is limited to replacing the original CUDASA function call with sending a broadcast message to all BEUs carrying the corresponding function identifier. Similarly, all shared distributed memory operations in the sequential user code are also translated by the CUDASA compiler to corresponding broadcast messages.

Distributed Shared Memory The network layer of CUDASA allows for computations that exceed the main memory of a single node. Therefore, each BEU makes a portion of its memory available to the distributed shared memory pool that is exposed as a continuous virtual address range to the application. This is in line with CUDA's global memory, or the system memory on the *bus layer*, that both allow for global storage of data, accessible to all subsequent *kernel* or *task* functions.

Similar to the CUDA interface to control the global memory space from the *host function*, CUDASA provides `malloc`- and `memcpy`-style functions for managing the shared memory from the head node. Allocations in the shared memory are evenly split across all cluster nodes that participate in the memory pool. This is to ensure a homogeneous data distribution for each allocation to the global memory; without prior knowledge of the access pattern to the data, such a simple distribution concept can still offer a good balancing of the workload for the shared distributed memory in the common case that all data fields are accessed equally often. In contrast, from a BEU's point of view access to distributed shared memory is not directly accessible via a paging mechanism. Instead, the programmer needs to explicitly request specific memory ranges to be cloned to the local memory of the executing node. At any time, the local copy of data can be used to update the global memory. However, such update operations may conflict each other if executed in parallel on an overlapping memory segment. In accordance with the shared memory handling of CUDA, it is however guaranteed that exactly one of the operations succeed and the result of this instance is completely written to the global memory. Ultimately, the approach to the distributed shared memory is similar to the concept of Global Arrays [NHL94], even though, the mechanism was primarily derived directly from the CUDA architecture in order to best match the graphics hardware concepts.

The shared memory manager is implemented using MPI-2 Remote Memory Access (RMA). It distinguishes between two classes of operations: Collective and single-sided memory operations. Allocations and deallocations, which may only be invoked from the head node, but affect all nodes due to the memory distribution scheme, are collective function calls. On the implementation side, both operations are thus addressed by the event loop of the BEUs. Issuing a collective memory operation consequently maps to the previously mentioned broadcast messages with the necessary data of the allocation sizes and data pointers attached. As a side effect, this also allows to keep a consistent view of the total global memory on all BEUs without any additional communication overhead, as each node is able to determine the exact distribution of each allocated memory segment. On the other hand, accessing such allocations is fully single-sided on both, the head and the compute nodes. This important, as opposed to allocations and deallocations such operations are necessary to operate in parallel to—or more precisely, as part of—a *jobgrid* execution. With the coherent view on the global allocation state, all nodes can access, lock, and read from/write to distributed shared memory through `MPI_Get` and `MPI_Put` operations. As one global allocation is distributed across all nodes, a single memory request from a BEU may require to access the local segments of more than one node; in consequence, such a memory access is not guaranteed to be atomic. However, this could be achieved by applying a two-phase locking protocol at the costs of introducing a substantial communication overhead.

Besides explicitly mapping shared memory segments using the corresponding API functions inside a *job function*, CUDASA also provides the alternative possibility of declaring the required memory segments for each *job block* separate from the actual *job* implementation. The primary benefit of this approach is to enable the system to optimize block scheduling in terms of shared distributed data accesses. From a programmer's point of view, this solely affects the direct calls to the CUDASA library functions, controlling the BEUs shared memory usage. Instead of calling these functions, the CUDASA language extensions provide means to declare the necessary information via *memory descriptor functions* by using the `__descriptor__` qualifier—analog to the standard CUDA function qualifiers. All such functions need to comply to a common function signature; that is, they accept a *job* index and the grid dimension as input and are expected to return the requested memory segments for the given *job* index. In other words, a *memory descriptor function* allows to query the location and size of the necessary shared memory segments for each *job* block at run time, independent of the actual *job* execution. For an implementation, this often relates to simply regrouping the relevant parameters of `cudasaMemmap`—CUDASA's library function for mapping global memory into local memory—into a separate function, i.e. a *memory descriptor function*. For example, instead of explicitly implementing the remapping of global memory inside a *job function*

```
__job__ void jfunc(float *f) {
    ...
    cudasaMemcpy(&handle,                    // memory handle
                 (void**) &localMem,         // local memory destination
                 f + jobIdx.x*JOB_SIZE,      // DSM source location
                 JOB_SIZE);                  // bytes to map
    ...
}
```

an equivalent *memory descriptor function* for the exemplary mapping operation can be defined in the following way:

```
__descriptor__ void memDescF(unsigned int *outStart,
                             unsigned int *outSize,
                             dim3 idx, dim3 grid) {
    *outStart = jobIdx.x*JOB_SIZE;
    *outSize  = JOB_SIZE;
}
```

To connect the descriptor with the corresponding function parameter, the original parameter declaration of the *job function* is adapted accordingly:

```
__job__ void jfunc(using memDescF float *f) {
    ...
}
```

In the same way, any other parameter can be defined to use different *memory descriptor functions* or these functions may also be reused for multiple parameters at the same time. By applying this mapping paradigm, CUDASA automatically maps and unmaps the requested segments prior and after to the execution of each *job*; thus, no direct call to *cudasaMemmap* is required anymore. The pointer to the distributed shared memory, that is passed as original parameter to the *job function*, is replaced at compile time with the pointer to the locally mapped memory on the executing BEU. In doing so, the access to distributed shared memory becomes completely transparent within the *job function*.

However, is some cases it might not be possible to solely utilize this automatic mapping mechanism; for example, a *job function* might map more global data than is locally available in a sequential fashion. For such cases, it is possible to explicitly disable the automatic mapping per parameter by adding the `__nomap__` qualifier to the parameter declaration. On the one hand, this revokes the elegance of transparent distributed shared memory usage and the user is back in charge to control the local mapping via the CUDASA library functions. On the other hand, this still maintains the possibility to optimize scheduling of the block of the *jobgrid* to the BEUs according to their memory usage, i.e. the expected communication overhead in terms of shared memory accesses.

Job Scheduling In the general case, the assignment of the blocks of a *jobgrid* to the BEUs is similar to the scheduling mechanism of the *bus layer*. Again, a single node, i.e. the head node, maintains a list of all pending blocks that is queried by the BEUs upon being idle. Thus, block distribution is completely non-deterministic, but load balancing is assured as long as enough blocks are waiting to be processed. However, scaling to very large cluster systems is very likely to be hindered by the bottleneck of a single node being in charge to handle all requests to the *job* queue; a large number of idle nodes asking for new jobs simultaneously may congest the network communication with the job queue and hamper parallel *jobgrid* execution. Multiple job queues, and multiple *job* distribution nodes respectively, within a single cluster setup or the assignment of more than one *job* per query request could be used to bypass this bottleneck. In particular, the head node can reasonably estimate the number of blocks to be processed on each node based on the number of compute nodes and the size of the grid. However, such mechanisms only come at costs of less optimal load balancing.

In contrast to the general case of *job* distribution, providing *memory descriptor functions* for the function parameters can aid the scheduling process. The combination of the knowledge about the shared memory allocations, i.e. their local distribution across the BEUs, and the possibility to evaluate the data requirements for each block prior to its execution, allows to optimally distribute the blocks to the cluster nodes. An optimal scheduling in this context refers to assigning each block to the BEU that features the largest overlap between the required global data and the local portion of the shared distributed memory residing on this BEU. In doing so, costly remote memory accesses can be severely reduced and are replaced by fast local memory operations within the executing node.

To derive such an optimal distribution in CUDASA, each node generates a sorted list of all blocks in the order of prioritized execution before querying any *job* from the head node. In detail, each BEU estimates the costs for executing each *job* in terms of communication costs for shared distributed memory usage.

By evaluating the *memory descriptor functions* mdf_p for all function parameters p, that are linked to use such a descriptive function, the overall cost estimate d is given as

$$d_n(\boldsymbol{j}) = \frac{\sum_p size(locallyAvailable_n(mdf_p(\boldsymbol{j})))}{\sum_p size(mdf_p(\boldsymbol{j}))}$$

for a specific node n and the block with *job* index $\boldsymbol{j}$. Thereby, the portion of the locally available memory segments is determined based on the allocation tables for shared distributed memory, that is local knowledge one all nodes. As a result each *job* is assigned with an indicator of the suitability to be executed on node n in the range of $[0 \ldots 1]$, with a value of 1 indicating a perfect overlap of the requested global data with the locally stored memory segment available on this BEU. Subsequent sorting of the list of blocks according to their suitability indicator, leads to the *job* priority list on each node. Based on this list, each node sends a small set of preferred *jobs* as request for the next block to process. If the head node cannot fulfill the request, e.g. because all the requested *jobs* have already been assigned to other BEUs, it orders the requesting node to send another packet of work items to choose from until an appropriate one was found. By choosing the request sets to be of a reasonable size, e.g. the number of nodes in the cluster, such message round trips can be effectively reduced to a minimum in most cases.

For very large *jobgrids* the sequential computation of the preferred *jobs* can take a prohibitively long time to be computed, as it requires the *memory descriptor functions* being evaluated for each possible *job* on all BEUs prior to the first *job* execution. Thus, a single call to a *job function* could introduce an severe computational overhead; leading to a high latency between the issue of the execution of a *jobgrid* on the head node and the actual start of the parallel processing on the BEUs. In order to minimize the effect of this issue, the *memory descriptor functions* are translated at compile time to CUDA code, that can be executed directly on the GPU. Hence, the evaluation of the block priorities can severely benefit from the parallel execution model of the graphics processor. By design the *memory descriptor functions* offer a very high arithmetic intensity—the ratio of arithmetic operations to memory operations—and the evaluation of each block of a *jobgrid* is completely independent of all others; thus making these functions perfectly suitable for fast execution on graphics processors. It is worth to mention, that this evaluation does not conflict with any other processing on the BEUs, as executing a *job function* is a blocking call for the host node, and thus, the graphics processors are defined to be in an idle state at the moment of issuing a new *job function*.

Atomics CUDASA also extends the concept of atomic functions to distributed shared memory. They are implemented based on the memory window locking

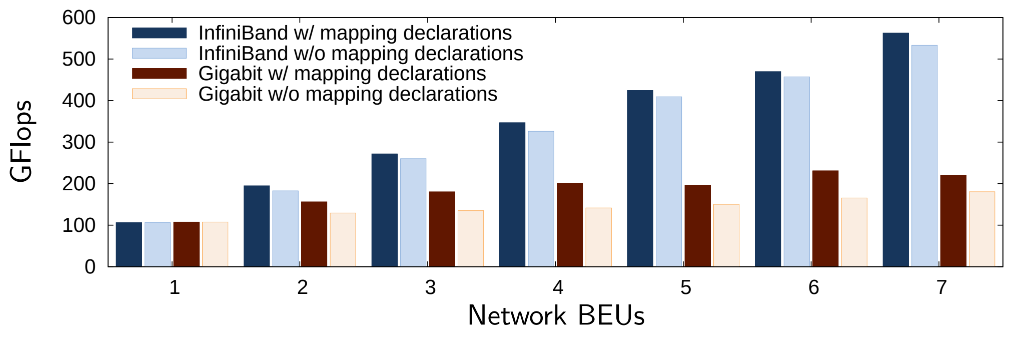


Figure 6.4: SGEMM benchmark series using the CUDASA *network layer* on both, InfiniBand and Gigabit Ethernet. In the latter case, scaling is severely hindered by communication costs.

mechanism of MPI-2. Several preconditions for atomic functions must be met to avoid a two-phase locking protocol for multiple segments: First, atomic functions are only allowed for 32-bit words, which must be aligned on a word boundary within the allocation. This is a reasonable constraint which normally also applies to atomic operations in main memory. And second, each aligned word must not span across segment boundaries. This precondition can be easily enforced using a segment size of multiple of the word length.

Apart from these limitations, all CUDA atomic functions are provided by the CUDASA runtime also for the *network layer*. On the implementation side, the communication overhead for such operations is limited to a single `MPI_Accumulate` call with the corresponding operation code inside a locked exposure epoch.

Scalability and Performance In correspondence to the evaluation of the *bus layer*, the BLAS library function *SGEMM* also serves as testbed for extending parallelism to a cluster environment. The basic workload distribution on the *network layer* follows the same sub-matrix distribution scheme as already applied previously for the multi-GPU evaluation. In fact, the distribution mechanism is simply shifted up one level of abstraction, whereas the new *bus layer* solely passes through the requests, i.e. there is a one-to-one correspondence of *kernels* and *tasks*. Two variants of the network version of *SGEMM* exist: The first one manually manages access to the shared distributed memory via CUDASA's library functions; thus, featuring a random distribution of *jobs* to the network BEUs. In contrast, the second variant provides *memory descriptor functions* for all *job function* parameters in order to enable the scheduler to assign the *jobs* even more optimally in terms of communication overhead.

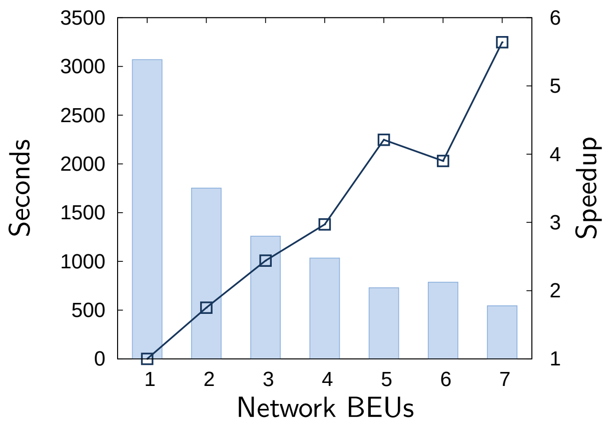
Both implementation variants were benchmarked using a size of $16,000^2$ elements for all matrices, of which each *job* has a constant sub-matrix of size $3,200^2$ assigned. The series of tests, aimed to analyze network scalability, were conducted on an eight node cluster system; each node equipped with an Intel Core2 Quad CPU, 4 GB of RAM and an NVIDIA GeForce 8800 GTX GPU.

Figure 6.4 shows the scaling behavior of CUDASA's cluster level for both, an InfiniBand interconnect (blue bars) and Gigabit Ethernet (red bars). The diagram also highlights the benefit of the locality-aware scheduling mechanism (dark colored bars) in comparison to the random *job* distribution (light colored bars). Except for using only a single compute node—minimizing network traffic between the compute nodes is irrelevant in this case—using *memory descriptor functions* proved to be always faster than arbitrarily assigning the *jobs* to the BEUs.

6.2.5 Discussion

Extending the available programming languages for GPGPU development to expose those higher levels of parallelism offers several benefits: First, a common interface for controlling *GPU*, *bus*, and *network level* parallelism keeps the programming and learning overhead for the programmer very low. Second, as shown exemplary with the CUDASA system, only few additional language elements suffice to extend the available language concepts from single GPUs to cluster environments. And third, even though porting applications to GPU computing is still a challenging task, the overhead for extending the once ported algorithm to multi-GPU setups or GPU clusters is comparatively small with the help of such extended language specifications.

However, the extended levels of abstraction also add a new challenge to GPU processing for the programmer: the distribution of workload is now required to be distributed across a complete hierarchy of parallel executed BEUs; with each level exposing different underlying communication mechanism and, thus, very different costs for identical operations. For instance, overhead for shared memory accesses range from global memory lookups on the *GPU layer*, to system memory lookups for the *bus layer*, up to costly remote memory accesses involving inter-node communication on the *network layer*. As a result, finding the optimal distribution strategy for a given algorithm can be quite difficult and may sometimes only be based on experience or even trial-and-error. To automatically derive such an optimized distribution of workloads across a full hierarchy of inhomogeneous compute devices Frey and Ertl [FE10] recently introduced the *PaTraCo* framework, which employs a weighted directed acyclic graph to model the workload dependencies as well as performance estimates. Their scheduler operates on this graph structure and assigns workload with the target to optimize the expected longest critical path in order to successfully derive the workload distribution with the best execution time based on the initial estimates.



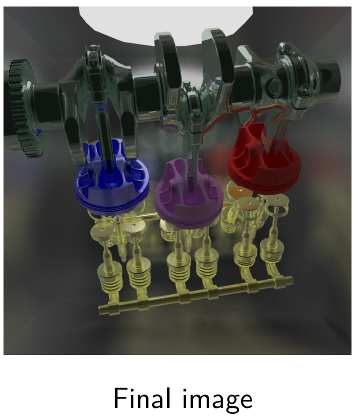


Figure 6.5: Benchmark series of an path tracer based on the CUDASA system. The left diagram shows overall rendering times as well as the scaling behavior in respect to the number of active render nodes. The final rendering of the scene with approximately 310,000 triangles is depicted on the right. Data set courtesy Madness GmbH.

For the special case of replicating all input data across all introduced layers, i.e. scaling is solely targeted for processing power and not for the amount of available memory, the blocks of the *kernelgrid* may be arbitrarily assigned across all other layers. As already indicated for the *bus layer* in Section 6.2.1, in this special setup an automatic distribution of the workload based upon the systems knowledge of the available BEUs on each layer becomes possible.

6.2.6 Applications

To conclude this chapter, two real-world applications based on the CUDASA architecture or its concepts are briefly presented and evaluated. Furthermore, these examples are meant to indicate the applicability of the extended programming paradigm inherent to CUDASA to larger, more complex software projects.

Global Illumination The first application is an iterative path tracer suitable to be executed on graphics hardware. In short, path tracing is a Monte Carlo Markov chain method that converges to the solution of the *rendering equation* (cp. Section 3.1). Actually, Kajiya [Kaj86] already introduced path tracing in the course of defining the *rendering equation*.

In this implementation the distribution of workload across *jobs*, *tasks* and *kernels* is devised in image space. A *job* processes a fixed number of consecutive image lines, whereas each *task* is responsible for 64 pixels in a consecutive row (or two rows if wrapping occurs). Finally, each *kernel* computes one pixel tracing all necessary rays through the scene; for evaluation, a total number of 150 rays were used per pixel with a maximum of 6 ray bounces. The complete scene description—packed in a uniform grid acceleration structure—is replicated across all BEUs on all three abstraction levels.

Results for scalability using the full stack of the CUDASA system on the previously described cluster system utilizing the InfiniBand interconnection are given in the diagram on the right of Figure 6.5. The final rendering for this series of tests is shown on the left. Overall scaling of performance using this rather simple workload distribution is mostly hindered by load imbalances between the generated *jobs*. Especially as the rendered scene exposes very large differences in overall rendering costs per pixel. That is also the reason for the drop in performance when moving from five to six cluster nodes. Employing an improved distribution scheme that assigns the individual rows in an interleaved fashion is expected to further improve the scalability of this application.

CT-Reconstruction In the context of this thesis efficient parallelization of reconstructing volumetric data from the projection images of computer tomographic scans was explored. The work was carried out in collaboration with Steffen Frey [Fre08]; the project was part of a cooperation with the Daimler AG targeted for supporting quality insurance in the context of production and material testing.

Various approaches to parallelize CT reconstruction based on the *cone-beam algorithm* by Feldkamp et. al. [FDK84] were developed for both, OpenGL and CUDA. Even though, the final application is actually not based on the extended CUDASA language—development of the CT reconstruction and the CUDASA system were simultaneous—the distribution concepts on each abstraction layer fully resemble the parallelization paradigms inherent to CUDASA; thus, the findings can be well translated to CUDASA.

In line with all other test series, the scaling behavior for a multi-GPU system is nearly linear up to four GPUs in one system. Effectively, reconstructing a 1024^3 volume from 720 input images of size 1024^2 is achieved in slightly over six minutes for a full 32-bit float reconstruction pipeline. Further scaling to the *network layer* was analyzed for small system setups; distributing the workload to two single GPU machines improved the overall reconstruction time by a factor of 1.7, including all necessary communication overhead as well as the final combination of the output data. However, due to the high interest of the industrial partner on utilizing only a single multi-GPU machine, scaling to larger cluster environments has not yet been evaluated.

Chapter 7

Development Tools for Parallel Graphics Processors

The previous chapters covered a variety of visualization techniques specifically designed to utilize the parallelism of graphics processors in single GPU systems as well as multi-GPU environments. This chapter shifts focus from the algorithm design to the actual development process for graphics units. Motivation for directing research to this specific area is given by answering the following question: What makes programming a graphics processor so different from the development process for CPU-based applications? In a nutshell, three key differences can be identified:

First, the fast evolution of the rendering pipeline gets reflected in a constant change of both, the graphics processors as well as the programming interfaces. As a consequence, typical design patterns, common development practices, and programming experiences tend to be quickly invalidated as the underlying programming environments advance.

Second, achieving efficiency is often tightly coupled to detailed knowledge of the underlying target hardware architecture. While this applies to any type of hardware oriented programming in general, the aforementioned fast-paced evolution paired with largely graphics-centric, fairly high-level documentation additionally complicate this challenge.

Third and likely most important, graphics programming is by design a massively parallel task, demanding thousands of independent threads in order to efficiently saturate the available hardware resources. No matter if the programming interface tries to hide most of this parallelism, like common for most graphics-centric APIs, or explicitly hands control of the parallelism to the user, as typical for GPGPU programming interfaces, the developer is inescapably confronted with the challenges of parallel programming, such as thread synchronization, thread coherence, or workload balancing.

Initially motivated by own experiences made during GPU programming in the course of this thesis, this chapter's contribution focuses on improving the development process for graphics processors via customized developer tools. In detail, a new approach for debugging all programmable shader stages of a graphic application is introduced that is specifically designed with the GPU's parallelism in mind (Section 7.1). On the one hand, the introduced methods aim at exposing traditional debugging support known from the CPU, like variable inspection or single-step execution, in the context of graphics processors. On the other hand, the goal was to extend those traditional debugging features—mostly known from inspecting a single, sequential process only—to maintain the specific parallel execution environment of GPUs to additionally provide insight to key properties for parallel shader execution, such as thread coherence or critical path analysis. While specifically designed for handling shader execution on graphic processors, the derived way of debugging parallel threads also applies to the broader range of parallel processors based on a *SIMD* execution model, as for example the *Synergistic Processing Units* (SPEs) of a Cell processor, the vector processing units of Intel's Larrabee processor, or the *poly execution units* of Clearspeed's CSX700 processor.

Building upon the function set exposed by the debugger, a first step towards profiling graphics shaders is discussed subsequently in Section 7.2. The combination of both, customized graphics debugging and shader profiling, is targeted at addressing all three aforementioned critical characteristics of programming for graphics processors.

7.1 GLSL Shader Debugging

Debugging is, in general, a tedious, time consuming, and often highly inefficient task that is, however, a crucial component of every software development process; and shader programming is no exception to that. Especially, with the transition from fixed-function or barely configurable shading units to fully programmable shaders, graphics programming languages almost match their traditional, general-purpose counterparts in feature set as well as complexity. Naturally, alongside the enriched programmability and increased processing power of modern GPUs corresponding graphics shaders tend to expose severely larger number of lines of code as well as increased complexity—with the most significant event being the introduction of fully dynamic flow control to graphics shaders starting with shader model 3 in 2004.

However, programmability of graphics processors—and the resulting difficulties in developing and debugging shader code—were not met adequately by developer tools support, typically available for CPU development. On the one hand, that is a direct consequence of the hardware's limited support for low-level debugging mechanism, like interrupt handling, trap signals, or direct memory manipulation. On the other hand, the GPU's unique execution model with thousands of independent threads executing the very same shader does neither map efficiently to the classical concept of debugging of a sequential process, nor does it match the communication-centric debugging schemes common to highly multi-threaded or distributed cluster applications.

Therefore, most prevalent debugging techniques for graphics shaders still include printf-style debugging—remapping visible output attributes like pixel colors to carry debug information—or software emulation in combination with traditional CPU-side debugging. Both techniques, however, have major disadvantages. Printf debugging requires manual code instrumentation, code recompilation, and is practically limited to fragment shaders only, as vertex programs and geometry shaders offer no suitable output attribute for simple visual feedback. While software emulation of the graphics pipeline, as for example available with Microsoft's PIX debugger for Direct 3D shaders [Mic07], offers all the ease of conventional debugging, the debug focus is typically limited to a single shader thread only, both for performance reasons and to reduce the highly parallel debugging environment to a simple sequential process. Consequently, the notion of parallelism and the ability to easily compare thread execution across the complete screen is lost.

Several concepts were to address the aforementioned limitations of shader debugging, which also greatly influenced the work presented in this chapter. Purcell and Sen were the first to propose automatic shader instrumentation to aid debugging of shader execution [PS03]. While the system still required host code instrumentation, single-step shader execution was transparently exposed via the concept of *interactive deepening*, a technique for automatically generating a sequence of truncated debug shaders that emulate single-step execution of ARB assembly fragment programs. However, *interactive deepening* is intrinsically limited to shaders without flow control and is applicable to assembly fragment shaders only. Following this idea for shader debugging, Duca et al. [DNB+05] introduced a debugging environment for the complete graphics pipeline based on a custom debug language rooted in relational database queries. Such queries could include *Cg* shader variables, which were automatically extracted by the system via code instrumentation. However, the concept of debugging heavily diverged from traditional debugging concepts: Most notably, the system does not allow to step through shader execution, but rather requires the user to specify watch variables prior to the debug execution. Hence, multiple iterations of the complete debugging process might become necessary to identify the root cause of a shader bug.

Inspired by both systems, a debugging system for vertex, geometry, and fragment shaders written in the OpenGL shading language was introduced in cooperation with Thomas Klein [SKE07], which allows completely transparent debugging of shader code on the GPU via single-stepping without the necessity to manually

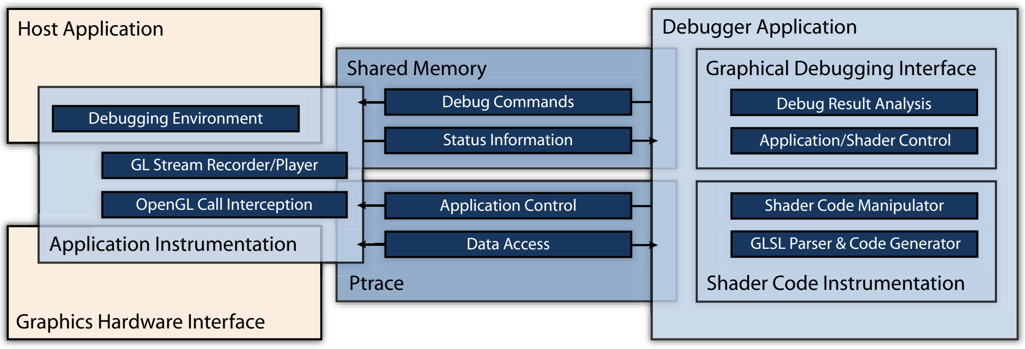


Figure 7.1: System architecture of the GLSL debugging environment.

introduce changes to both, the host and the shader code[*]. That way, it can be easily included to any development cycle of graphics applications.

7.1.1 Architecture Layout

The introduced architecture for shader debugging splits into two main components that communicate by means of exchanging essential debug information, e.g. debug commands, status information, and debug data.

The first component is a library running in the process space of the debugged host application, which enables the instrumentation of the host application for debugging GLSL shaders in arbitrary OpenGL programs without the need to recompile or even having the source code of the host program available. The concept is based on interactively intercepting all OpenGL calls evoked by the application during program execution with full access to all function parameters, thus enabling among other things the retrieval of shader source code. Depending on the current state of the debugger this allows for either running the host program unaltered with only little performance overhead or stopping and stepping through the execution of the target program on a per OpenGL function call level. The later is used to identify a single draw call as target operation for shader debugging. In addition, the instrumenting library establishes a debugging environment, i.e. float buffer objects, in the graphical context of the host application that permits not only the retrieval of the requested debug results but also assures that subsequent program execution is not affected by the debugging process.

The second component is the actual debugger application that can be further divided into two large modules. First, the GLSL shader code instrumentation

[*]In fact, the system is designed to be applicable to any OpenGL application using GLSL shaders without the need of having the source code at hand.

performs automatic code manipulation. An OpenGL shading language parser is used to create an intermediate representation for a given shader that serves as basis for identifying the program execution order, variable scope determination, and the manipulation of shader source code in a syntactically and semantically correct manner. The compiler back-end is a GLSL code generator that reconstructs valid shader programs from the intermediate language. Second, a graphical debugging interface allows to control program execution of both the host application and the target shader, to select the draw call of interest from the OpenGL command stream, to specify debug requests for shader variable data, and to provide capable analysis and interaction methods for the generated debug results. Figure 7.1 gives a brief overview of the main components of the system and their interaction.

Based on these components the general control flow for debugging a GLSL shader program is as follows. For the draw call of interest the source code and execution environment of the currently bound GLSL shader is read back from the OpenGL state and passed to the shader code analysis module. An intermediate representation is built that serves as basis for scope determination, debug code insertion, and program control flow determination. Then, for each debug step in the shader program, an augmented debug shader is generated from the intermediate representation and inserted into the OpenGL state of the host application. Now the draw call is replayed and the debug result is read back and transfered to the debugger application.

7.1.2 Shader Code Analysis

In order to establish a basis for shader code instrumentation for a high-level shader language with dynamic flow control, it is necessary to fully analyze the syntactic structure of a shader program. Thus, an intermediate representation of the shader, i.e. a parse tree, is created. This tree structure fully replicates the syntax of the program string and provides the basis for any automated code manipulation or debug execution. In the context of Cg shaders such an approach was successfully applied by Duca et al. [DNB$^+$05] for debugging purposes. In this context, a heavily updated and extended version of 3DLabs' OpenGL Shading Language Compiler Front-end[3Dl05] is used.

The main focus of 3DLabs' original code is the rapid development of cross-platform compilers for low-level machine specific code generation. As the intended back-ends are highly vendor and hardware specific no actual implementation of any code generator is supplied. Applying this to shader debugging, the back-end is supposed to reconstruct valid GLSL shader code, which requires additional program information to be stored in the intermediate parse tree. Most importantly, this affects preprocessor directives, variable declarations as well as user defined `struct` data types. The former needs to be preserved since preprocessor state-

```
#extension GL_EXT_gpu_shader4 : enable
#define SQRT5F 2.236068f

uniform isampler1D tex;

int fibonacci(int n) {
  const float goldenRatio = (1.f + SQRT5F) / 2.f;
  return int((pow(goldenRatio, n) - pow(-goldenRatio, -n)) / SQRT5F);
}

void main() {
  int n;
  n = texture1D(tex, gl_FragCoord.x).x;
  gl_FragColor = vec4(fibonacci(n));
}
```

Listing 7.1: Exemplary input fragment shader for code analysis; further possible optimizations, i.e. additional constant folding, were omitted for simplicity. For demonstration purposes the assignment statement highlighted in red is selected as current debug target.

ments expose direct compiler control that should affect not only the debug compilation process, but also succeeding compilation of the generated instrumented code by the driver-level compiler. The later requires additional data to be stored, e.g. structure names, which are usually not necessary in the context of machine level code generation. To this end, the overall goal for extending the intermediate representation was to allow for syntactically valid and semantically equivalent reconstruction of a given shader in the compiler back-end. Thus, all compiler level optimizations in the original code, especially concepts like dead code removal or constant folding, were completely removed from the original code, to assure the best possible match between the input shader code and its corresponding generated output from the compiler. In particular, besides the absolute need for semantic equivalence for debugging purposes, a close syntactical similarity between an input shader and its un-instrumented output code is crucial to minimize the possible side effects of the shader code analysis on the driver-level compiler. An example of the code analysis is given for the input example of Listing 7.1, with the abstract syntax tree depicted in Listing 7.2. Code manipulation as well as shader reconstruction is performed solely on this intermediate representation.

The compiler was also updated to add support for recent graphics hardware. Unfortunately, public development for the original compiler front-end stopped with support for GLSL 1.10. Therefore, all functionalities exposed by GLSL version 1.20, e.g. non-square matrices, handling of arrays as first class objects, etc., had to be integrated following the language specification [Kes06]. At last, all

```
Sequence                                          {818 819 821 783} <Path>
 Declaration                                                     {} <Path>
  Preprocessor GL_EXT_gpu_shader4 enabled
  declare 'tex' (uniform isampler1D)                             {} <None>
 Function Definition: fibonacci(i1; (int)                 {818 819} <Path>
  Function Parameters:                                        {818} <None>
   'n' (in int) [id:818]                                      {818} <None>
  Sequence                                                    {819} <Path>
   Declaration                                                {819} <Path>
    declare 'goldenRatio' (const float) [id:819]              {819} <Path>
     move second child to first child (float)                 {819} <Target>
      'goldenRatio' (const float)                             {819} <None>
      1.618034 (const float)                                     {} <None>
   Branch: Return with expression                                {} <None>
    convert float to int (int)                                   {} <None>
     divide (float)                                              {} <None>
      subtract (float)                                           {} <None>
       pow (float)                                               {} <None>
        1.618034 (const float)                                   {} <None>
        'n' (in int)                                             {} <None>
       pow (float)                                               {} <None>
        -1.618034 (const float)                                  {} <None>
        negate value (in int)                                    {} <None>
         'n' (in int)                                            {} <None>
      2.236068 (const float)                                     {} <None>
 Function Definition: main( (void)                {821 783 818 819} <Path>
  Function Parameters:                                           {} <None>
  Sequence                                        {821 783 818 819} <Path>
   Declaration                                                {821} <None>
    declare 'n' (int) [id:821]                                {821} <None>
   move second child to first child (int)                     {821} <None>
    'n' (int)                                                 {821} <None>
    direct index (int)                                           {} <None>
     Function Call: texture1D(sI11;f1; (ivec4)                   {} <None>
      'tex' (uniform isampler1D)                                 {} <None>
      direct index (float)                                       {} <None>
       'gl_FragCoord' ([id:783] vec4)                            {} <None>
       x (swizzle)                                               {} <None>
      x (swizzle)                                                {} <None>
   move second child to first child (vec4)            {783 818 819} <Path>
    'gl_FragColor' ([id:783] vec4)                            {783} <None>
    construct vec4 (vec4)                                 {818 819} <Path>
     convert int to float (float)                         {818 819} <Path>
      Function Call: fibonacci(i1; (int)                  {818 819} <Target>
       'n' (int)                                                 {} <None>
```

Listing 7.2: Resulting parse tree from the analysis of the example shader (Listing 7.1), with the corresponding debug state per node on the right. The indentation depicts the tree structure; whereas the leafs are built-in or user defined variables/functions as well as user defined constants (blue). In addition, each node keeps track of all identifiers changes by its child nodes or directly by itself (green).

extensions coming along with the introduction of NVIDIA's G80 hardware, especially the `EXT_gpu_shader4` and `NV_geometry_shader4` specifications [NVI06b], were added. To achieve compatibility for shader code that relies on vendor specific enhancements to GLSL, such as additional implicit type casts support was added where changes were sufficiently documented or obviously perceivable.

Besides the information capturing the syntax of the input shader code, additional semantic data is attached to each tree node especially for debugging purposes. In short, each node is additionally augmented with a list of all variables in scope at the location of the corresponding statement, a second list of possible side effects this specific statement may cause to any variable content, and finally with its current state of debugging.

Scope Even though the scope is completely tracked by a stacked symbol table during the compile process, the list of user-defined variables in scope for each node is additionally stored inside the parse tree; this was mainly for avoiding repeated invocation of the costly scope level determination for every change of the debugged statement. Thus, every node is attached with a list of unique identifiers that represent the corresponding user-level scope—also identifying the complete set of user variables that can be queried at the source code location of the corresponding statement.

State Changes Debugging is primarily about keeping track of the state of variables while stepping through the source code. One valid option to provide a consistent up to date view on all variable states for a given debug statement is to always read back all variables in scope as soon as the debug target changes. However, this approach has two major drawbacks: First, especially in the context of debugging large amounts of threads in parallel, retrieval of the data for all variables in scope from the GPU may impose a very high processing overhead. And second, it does not allow for identifying possible side effects a statement may have outside of its original scope, e.g. when jumping over a function call. To address both issues, each node in the tree structure is augmented with a second list of variable identifiers, containing all variables that may have changed in at least one thread due to the execution of the corresponding statement. In contrast to the scope list, the list of possible state changes may contain variables not in scope of the corresponding statement.

The construction of those lists of state changes is done by traversing once through the syntax tree in a pre-/post-order fashion. Whenever the visited statement indicates that a child node may possibly change, e.g. the left child of an assignment operator, traversing of this child continues with adding every encountered variable operand to their corresponding list of changes. During the post-order visit of every node, the lists of all direct children are concatenated in order to obtain the final result per node. An example of the result of this process is also given in Listing 7.2.

Debug State In contrast to all prior data, the debug state is not a direct result of the analysis of the syntax and semantics of the shader code itself. In fact, it reflects the current state of code debugging with respect to a specific user-defined *debug target*; in detail, this target specifies the statement, i.e. a specific tree node, in front of which debugging is actually performed. This is analogous to the common process for debugging a sequential process on a CPU in which the debugging results represent the variables' state prior to executing the statement marked as target. Compared to all other data stored in the tree, this is the only information that needs to be adapted upon stepping through the shader during a debug session.

The debug state is defined to be a tri-state attachment to every node (see the rightmost column in Listing 7.2) that expresses the relation of a statement to the currently selected debug target to be of the three classes: *Target*, *Path*, or *None*.

A node is classified to be of type *Target*, if the corresponding statement is either the user-defined debug target at firsthand or the node marks a winding operation, i.e. pushing a sub-routine on top of the call stack, which represents the chosen flow control to the debug target. In other words, the latter condition tags a function call to be of class *Target* only if the control flow is interrupted on this stack level and continues on the next higher level of the called function on its way to the debug target.

As debugging is not only dependent on the call stack, but may also be affected on enclosing language constructs, e.g. conditionals or loops as further detailed in Section 7.1.4, the second class of type *Path* identifies all statements that possibly enclose the already marked target statements. In the tree structure this equals all nodes that lie on the direct path from a target node up the hierarchy to its enclosing function definition; in the following this sequence of nodes is called the *debug path*. For the special case of function calls, the difference of classification between type *Target* and type *Path* denotes whether the flow control needs to continue inside the called function or the target is enclosed by the function call on the current call level. More precisely, the latter case implies that the target is part of a function parameter.

Any node not marked as either *Target* or *Path* is classified as type *None* and is reconstructed in the instrumented debug shader without any code changes.

7.1.3 Shader Output Analysis

In comparison to the traditional process of debugging a sequential program on a CPU, debugging a shader compiled to run on a graphics processor imposes several challenges specific to the hardware environment as well as its execution model.

First, graphics processors so far do not expose a true machine-level language—like the x86 assembly language for x86 processors—which could serve as low-level access for debugging; in particular, no shader language without the necessity and constrain to be processed by any stage of the graphics driver exists up to date. Even though GPU compute languages like ATI's CTM [PSG06] or NVIDIA's PTX [NVI08] provide an instruction set architecture close to the actual hardware, they do not necessarily avoid translation to native GPU code nor do their instruction sets fully match with the features expose by the graphics pipeline and its programmable stages.

Second, opposed to CPUs the current generation of graphics processors does not feature any publicly accessible hardware functionality to support single stepping through a shader code, e.g. the concept of trap flags for x86 microprocessors, in order to interrupt the processing of shaders at a specific target instruction. Again, such a functionality has already been defined in the context of GPU compute languages, i.e. PTX's `trap` and `brkpt` instructions, but so far these instructions are actually not publicly supported by the available hardware.

Third, there is no direct access to the register file or memory used during shader execution on a graphics processor. The only access model is based on using the graphics pipeline and its programmable shaders, which requires that the retrieval of debug information must be performed in the context of the debugged process.

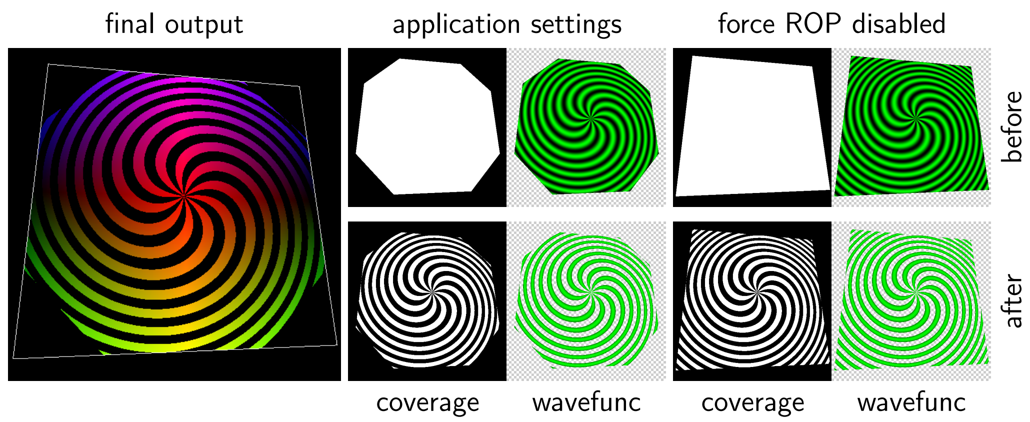
And finally, the fourth challenge concerns the massively parallel execution model of a shader program, no matter of which type (vertex, geometry, or fragment); thus, a debug process for graphics processors covering all executed threads at once needs to handle different control flows and different variable states for a very large amount of parallel threads.

In particular the two latter characteristics require any requested debug information, queried during the execution of a shader program, to be packed into and transfered as standard shader output. As defined by the graphics pipeline, such output options are highly dependent on the actual debugged shader stage. In addition, from an application's point of view and its access options, any result from a programmable shader stage is logically passed down the pipeline only after all elements finished execution for that stage—opposed to the actual streaming model of the underlying hardware architecture. Thus, collection of the debug results is not selective for a specific thread, but can only be retrieved once for all executed threads. With respect to this parallel execution model, for the purpose of debugging it is therefore crucial to always maintain a one-to-one mapping between a shader invocation and the location of its result in the accessible shader output. For each of the three different shader stages this mapping is obtained in a different way, which is discussed in the following in the order of increasing complexity.

Fragment Shader For every fragment created during rasterization one distinct invocation of a fragment shader is scheduled and its result is written to its predefined screen position into the output buffer. Thus, the straightforward mapping between a single thread and its location in the output data stream are simply given by its screen position. To be more exact, this mapping is only unambiguous in case of a single input triangle or for multiple input primitives that do not overlap each other, i.e. no two fragments are written to an identical screen coordinate. For debugging purposes, this natural mapping is used with the additional constraint to consider only those fragments, that actually contribute to the final rendering after all fragments of a debugged input primitive stream is fully processed. This way, the mapping is unambiguous for any given group of input primitives. For a user this model maps to the natural concept of debugging only fragments that actually become visible on the screen and ignoring those that are hidden due to raster operations, i.e. that fail to pass the per-fragment tests. Alternatively, Duca et al. [DNB$^+$05] propose to apply depth-peeling to resolve the general case of arbitrarily overlapping input and to derive a correct mapping function. As one of the global design goals was to make the least possible assumptions on the target application, the described system is restrained to the somewhat simpler, but less restrictive model in terms of expected usage of the graphics pipeline.

Having a valid mapping still leaves the question, whether an output value in the debug buffer was actually written by the corresponding shader thread or execution was aborted due to a fragment `discard` operation, i.e. the value reflects the unchanged state prior to the shader execution. To resolve this issue, a specialized debug render pass is used, that simply sets a flag for every written fragment in a previously cleared buffer. In doing so, the output buffer can be classified in active and passive fragment shader threads in terms of their output. The active fragments corresponds to all threads that pass the given debug statement and result in a written, visible pixel in the output buffer (see Figure 7.2). Please note, that this render pass is only required to run once for every change of the debug target. In order to allow debugging of fragment shader threads that may be terminated due to a `discard` after the current debug target, debug code instrumentation does not reconstruct such operations from the syntax tree. Several alternatives to this approach exist, such as using the stencil buffer, but again the selected implementation option guarantees to not introduce any assumption on the usage pattern of such auxiliary buffers by the target application.

Reading back the actual results from a fragment shader is realized using the



```
vec4 t = gl_TexCoord[0];
float wavefunc = sin(80.0*cos(length(t.xy)) + 10.0*atan(t.x, t.y));
if (wavefunc < 0.0) discard;
gl_FragColor = vec4(1.0 - abs(t.x), -t.y, t.y, 1.0);
```

Figure 7.2: Comparison of debugging variable `wavefunc` prior and after the `discard` operation in the 3rd line of given code. The coverage depicts the corresponding classification in active (white) and passive shader threads (black); debugging results are only valid—and thus, only presented—for active threads. The results shown in the middle and on the right side differ in the configuration of the raster operations (*ROP*). The former keeps the application settings, while the latter disables all fragment tests to reveal the otherwise discarded fragments in the corners of the quad.

`EXT_framebuffer_object` extension. A single 32bit floating-point RGBA color attachment is used as debug environment that captures the debug output of the fragment shader stage. Additionally a depth render buffer and if necessary a stencil buffer are attached to the frame buffer object in order to fully mimic the original render environment of the undebugged application. To assure retrieving correct debug results—as they are actually emitted by the fragment shader—it is crucial to setup the raster operations accordingly and to disable all parts of the imaging pipeline that may affect debug values. In order to correctly rebuild the target applications' behaviour, per-fragment tests should operate identically regardless whether they work on the original target or debug render buffer. For debugging purposes, however, direct control of these tests is desirable, e.g. disabling the depth/alpha test may be beneficial in certain cases. Therefore the user can choose whether to copy alpha, depth, and stencil from the currently bound frame buffer or to clear them to user defined values for each shader step. Furthermore, the possibility for enabling or disabling individual fragment tests and blending is provided (see Figure 7.2 for an example).

Vertex Shader In case of vertex shaders the actual debug vertex data, i.e. the values of varyings emitted by the respective GLSL shader, have to be captured. This is accomplished by using the `NV_transform_feedback` extension that allows to capture vertex data prior to the clipping stage of the rendering pipeline and to store it into vertex buffer objects that can be subsequently mapped into main memory. In order to capture correct data from a vertex shader, a potentially active geometry shader is disabled during debugging. The current primitive mode for transform feedback is given by the primitive mode specified for the draw call in question in case of a vertex shader.

Besides the more complex retrieval of the debug data, the actual mapping between an input vertex and its corresponding data location in the output stream is intrinsically given by the input order; this is due to the warrant that the stream out, captured after the execution of the vertex shader stage, maintains the order of vertices with respect to the sequence in which they were issued. As vertex shaders feature a static *one-to-one* relation between input and output vertices, i.e. no vertex shader thread can be aborted, no additional render pass is required for establishing an unambiguous mapping between vertex shader threads and the debug data.

Geometry Shader Data retrieval for debugging geometry shaders equals the approach taken for vertex processing. The data output is captured again by the stream out option provided by the `NV_transform_feedback` extension, with the differences that the vertex shader stage remains unchanged in this setup and the primitive mode is given by the output primitive mode of the geometry shader being currently debugged.

In comparison to the two previously discussed shader stages, the geometry shader exposes a completely different execution model. Vertex and fragment shaders feature an *one-to-one* correspondence between the input elements and their corresponding output. Even though the fragment shader stage logically exposes an *one-to-zero* relation due to the ability to discard fragments, the position of the debug data in the output buffer still allows to clearly identify the corresponding shader thread for every output element. In other words, for both shader stages the required mapping can be fully described as a bijective function—considering the output of discarded fragments to be the unchanged pixel in the output buffer. In contrast, geometry processing allows for any arbitrary relation of type *one-to-many*, where the number of output elements may range from zero up to a hardware given maximum of output vertices. The output stream of a geometry shader is the simple concatenated list of all generated output vertices for all executed threads; without any additional hints relating to the underlying tessellation or to the originating shader thread. Thus, the original shader output is not suited for defining a valid mapping between the output elements and the distinct states of all shader instances. This is most obvious when considering shader threads, that do not emit any vertex; these threads can neither be located in the output stream nor would it be possible to output any debug information for those instances.

However, there is still a *one-to-one* mapping between the input sub-primitives of a geometry shader—compound input primitives, e.g. polygons, quads, or strips, are decomposed into their basic sub-primitives, i.e. points or triangles—and the executed shader threads. In order to utilize this correspondence for debugging purposes, it is necessary to guarantee that the output of a debugged shader matches the given input primitives—again restoring a bijective correspondence. In more detail, for every sub-primitive, and thus every shader thread, a single output primitive with a single vertex holding the debug data needs to be generated. Similar to the process of vertex shader debugging, the mapping is then given by the issuing order of the primitives, as the `NV_transform_feedback` extension also maintains the primitive order.

To comply with the above condition, debug code generation for geometry shaders does not reconstruct any `emitVertex` or `endPrimitive` statements from the syntax tree. By definition of the `EXT_geometry_shader4` extension, this approach does not alter the semantics of the shader execution, with the only difference that no output is actually written. In addition, a single vertex forming a point primitive is emitted wherever the control flow may reach the termination of the shader execution, i.e. at the end of the *main* function and before any `return` statement issued inside the *main* function. As a side effect, this approach also provides an intrinsic solution to the issue of debugging a shader thread, that does not emit any vertex, and therefore would not be accounted for in the original output data. On the application side, the output primitive mode is forced to point primitives for all debug render passes. Analogous to the debug process of the two other shader stages, the debug data is attached to the output element and each generated debug primitive reports data from it corresponding shader thread. In this way, debugging fully resembles the concept of vertex shader debugging with one debug element per sub-primitive.

While the approach described so far fully supports step-through debugging for geometry shaders, bugs related to the structure of the generated output primitives or problems best identified by comparing the data of the emitted vertices may be quite challenging to find. The reasons for this are twofold: On on hand, this is a result of anchoring the debug data solely to the input primitives and completely ignoring the actual output structure of the original shader. Thus, it remains the task of the user to follow and retrace the generation of output primitives and output vertices, while stepping through the code. On the other hand, comparing data of emitted vertices is tedious due to the semantics of the emit mechanism. For each emitted vertex the current state of the shader thread is attached at the time the corresponding `emitVertex` statement was called. Thus, the state reported by the debug process only matches the data valid for the next vertex to be emitted, but does not provide any reliable information about previously emitted vertices as soon as the shader's state was changed after those vertices were emitted.

In order to capture both, each thread's current variable state as well as the status of the already emitted vertices for a given debug target, it is necessary to correlate the actual written output of the original shader to the executed shader threads. In the above used terminology, this is equivalent to reconstruct the *one-to-many* relation, but only for the task of attaching the written output to their corresponding shader thread and not for recovering the variable state of the shader threads; thus not requiring a bijective mapping. In fact, as each shader thread may emit an arbitrary number of output primitives, each made up from an arbitrary number of vertices, the actual correlation to be found is a concatenation of two *one-to-many* relations. The sought-after mapping, thus, assigns each output vertex its distinct output primitive as well as its well-defined input primitive.

To derive this mapping from shader output only—as this is the only data the debugger is able to intercept and retrieve from the graphics pipeline—a two-pass algorithm is used: The first pass utilizes the shader state debugging mechanism, as previously described, to collect data on a per sub-primitive basis, or a per shader thread basis respectively. For each shader instance an unique identifier of the input primitive as well as the number of actually emitted vertices is collected in a single debug render pass. From an implementation point of view, the former is simply achieved by passing the `gl_PrimitiveIDIn` built-in to the written debug output. The latter, is accomplished by generating specialized debug code; instead of ignoring all `emitVertex` statements during code generation, a counter mechanism is generated in place. Thus, the number of actually written output vertices—which may differ from the number of issued `emitVertex` calls due to ill-defined primitives—is tracked during shader execution. The second pass of the reconstruction algorithm collects the original output vertex stream of the debugged geometry shader, i.e. operates on a per output vertex basis, and collects the number of emitted primitives prior to the corresponding `emitVertex` call for each vertex. Again, such a counting mechanism is added during the debug code generation in place of `endPrimitive` statements.

The mapping can then be fully reconstructed from the three derived debug data fields. From coarse to fine granularity, the input identifier is used to decompose the concatenated vertex stream into segments generated from a distinct input primitive. The number of actually emitted vertices allows to further separate those segments into vertex lists, that are generated by a distinct sub-primitive. In

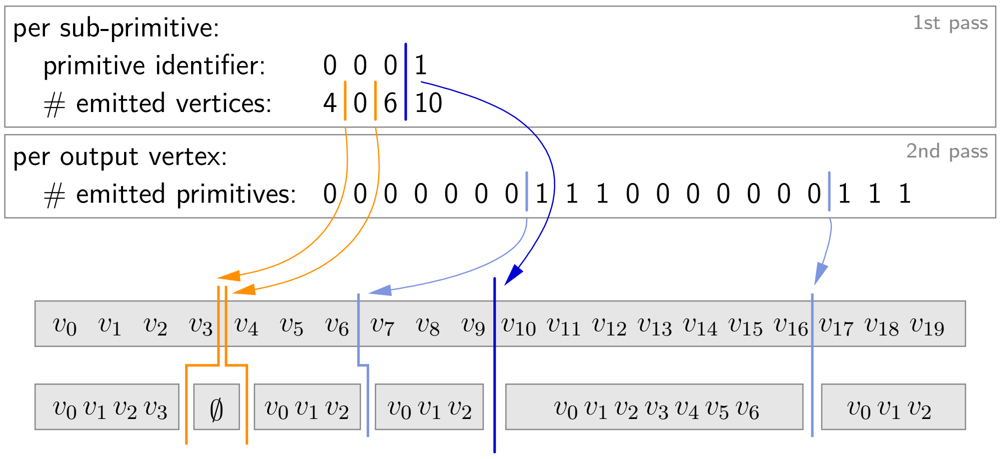

per sub-primitive:
1st pass
primitive identifier: 0 0 0 1
# emitted vertices: 4 0 6 10
per output vertex:
2nd pass
# emitted primitives: 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16} v_{17} v_{18} v_{19}
$v_0\,v_1\,v_2\,v_3$
$\emptyset$
$v_0\,v_1\,v_2$
$v_0\,v_1\,v_2$
$v_0\,v_1\,v_2\,v_3\,v_4\,v_5\,v_6$
$v_0\,v_1\,v_2$

Figure 7.3: Decomposition of a geometry shader's vertex output stream on the basis of collected data from two debug passes. The combination of the obtained data allows to split the concatenated stream of output vertices according to their input primitive (dark blue), input sub-primitive (orange), and output primitive (light blue).

particular, it allows for clearly identifying the location of the shader threads, that do not emit any output primitive. On the lowest level, the number of emitted primitives is then used to further split the lists into sections that define a single output primitive. An example of this process is given in Figure 7.3.

The final data structure, representing the relation between input primitives, sub-primitives, output primitives, and their vertices, is then given as a list of four-level trees. For debugging, the current state of a shader thread is attached to the level of the sub-primitives, as each of these primitives equals a distinct shader instance. For the given example case, the final debug structure is depicted in Figure 7.4. Please note, that the original debug output—being identical for both input primitives in the given example (see repetitive pattern in the output data of the second pass in Figure 7.3)—is not sufficient to reconstruct the actual tree layout.

The process of reconstructing the relationship between input and output elements for a geometry shader is required only once per debugged shader. It can be thought of as preprocessing to the actual debugging and is performed automatically prior to enabling the set through functionality of the geometry shader debugging. In this way, the unpopulated output relation structure is already available at the beginning of shader debugging—even though from a debug point of view no vertices have been emitted for the initial debug target. During the debug process, the initially empty vertex structure is then populated with debug data whenever the control flow passes an `emitVertex` statement. In this way, bugs due

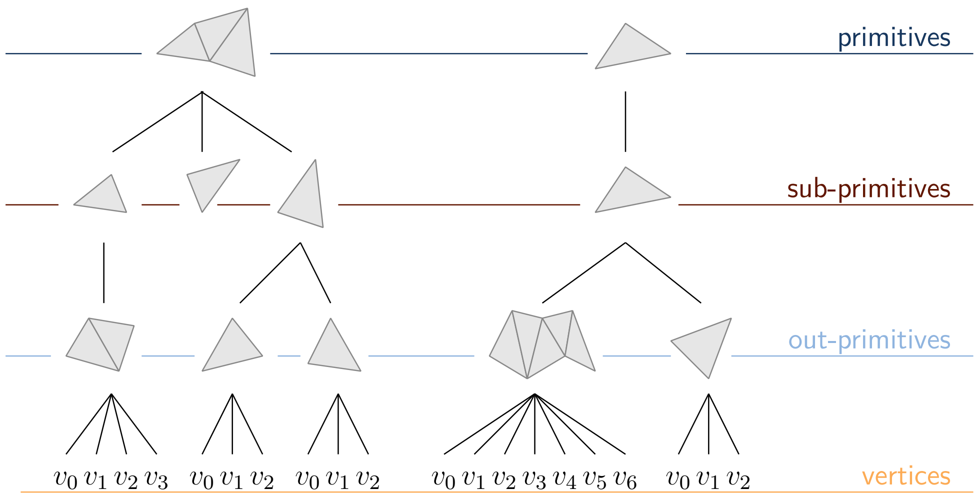


Figure 7.4: Data structure capturing the reconstructed relation between an geometry shader's input and output for debugging. Debug data representing the variable state of distinct shader threads are attaches to the sub-primitives; as each sub-primitive corresponds to a unique thread instance.

to erroneous output generation can already be identified without tedious manual tracking of the shader state during debugging.

7.1.4 Shader Code Instrumentation

The requirements for shader code instrumentation in the context of automated state debugging are threefold. First of all, manipulations to valid input code, i.e. shaders that comply with the GLSL specification and its extensions, must result in syntactically correct output, preserving the semantic structure of the input program in all parts that are not directly affected by the debugging process. In detail, required code additions must not induce any side effects to output registers, except for what is necessary to pass the target data values to the debugging environment. Particularly, with regard to fragment shaders, manipulation of the alpha and depth output are not permissible, as they influence the subsequent raster operations. Furthermore, code manipulation should be minimal to assure a maximum degree of similarity to the input program, in particular with respect to hardware limitations such as nesting limits. Finally, the code instrumentation is required to allow debugging of any variable in scope at arbitrary code positions on an expression level.

In contrast to *interactive deepening*, the method described here does not terminate the program reconstruction and execution directly after the debug target, but always restores the complete input shader. This ensures that subsequent calculations that may affect per element tests, e.g. depth or alpha test, are still performed by the instrumented code. In order to output debug values, a newly inserted varying is used in case of vertex and geometry shaders, while for fragment programs it is necessary to reuse at least one color channel of a bound render target. As GLSL features read-write access to output registers, early manipulation of an output register at the requested debug position may introduce side effects to the execution of succeeding parts of the code. Therefore, a global variable is defined to buffer the debug result until it is safe to write its content to a result register, i.e. until the program control flow terminates. For all newly added variables unique names are assured to avoid scope collisions by appending random suffixes, if necessary.

The additional code introduced to assign a requested variable to the debug register is in most cases directly added in front of the target statement by using the sequence (,) operator. This method proved to be the most flexible and generic. As the sequence operator can be used in place for any single expression, its operation order from left to right is well-defined, and the return type and value are defined by the right-most operand, which will all become relevant later on. An example for basic shader code instrumentation, with automatically added code marked in boldface, is shown here:

```
float dbgResult;
void main() {
  dbgResult = gl_Color.x, gl_FragColor = gl_Color * 2.0;
  gl_FragDepth = gl_FragColor.x;
  gl_FragColor.x = dbgResult;
}
```

Declarations One exception to the applicability of the sequence operator for debug instrumentation is the declaration statement. This statement is defined as a type name followed by a comma separated list of variables with optional initializers. Thus, the sequence operator cannot be used to debug the variable state in between variables of the same declaration statement; as the comma literal would be interpreted as separator of the variable list, and ultimately code compilation would result in a syntax error due to the redeclaration of the debug result register:

```
int i = 1, j = i++, dbgResult = float(j), k = j++;
```

Instead, declaration statements that lie on the *debug path*, i.e. that are marked with the debug state *Path*, are reconstructed in their semantically equivalent form of a sequence of declaration statements with only a single element per declaration list. Please, note that this is a valid transformation for all analyzed implementations of the OpenGL shading language and such a behavior is also indicated by the original GLSL 1.00 compiler front-end provided by 3DLabs; however, the order in which side effects occur among the initialization list expressions is is not fully defined in the OpenGL shader language specification—and explicitly defined to be of unspecified order in the ANSI C standard [Int99]. By using this transformation anyway, the code instrumentation can consequently be added in between the variable declarations as additional sequence expression:

```
int i = 1;
int j = i++;
dbgResult = float(j); int k = j++;
```

Conditionals In contrast to traditional debuggers for serial applications, not all threads of a shader necessarily need to follow the same program path. For inhomogeneous cases the user has to decide which branch should be followed for debugging purposes. To be able to base this decision on current variable content, debugging needs to be performed after the evaluation of the conditional test itself, given that side effects of the test possibly cause the target variable to change. On the one hand, adding the required debug expression after the conditional statement using the sequence operator would result in a wrong evaluation of the test and thus breaks the semantic equivalence with respect to the original program. This is due to the fact that the right-most operand determines the result value, which would be the newly added expression. On the other hand, code insertion can not be moved inside the branch body for conditionals that do not affect all elements, as it is the case for conditionals without an `else` branch.

As the usual way of inserting the debug code cannot be applied in this special case, the proposed solution uses a temporary, locally defined variable to buffer the result of the conditional test. Again, by utilizing the characteristics of the sequence operator, the conditional test is evaluated before the shader instrumentation while the resulting value of the used sequence is assured to equal the original conditional statement. This is illustrated in the following code fragment.

```
bool dbgCond;
if (dbgCond = (i++ < 10), dbgResult = float(i), dbgCond) {
  i = 20;
}
```

Please note, this approach is only required in case debugging needs to be performed after the evaluation of the condition test, but before flow control actually continues inside the corresponding branch, i.e. still for all threads that evaluate the condition. Debugging prior to the condition statement or after the condition test—only for the threads fulfilling the condition—is covered by the standard mechanism of code instrumentation. However, both possibly differ either in the reported state

```
dbgResult = float(i), if (i++ < 10) {
  i = 20;
}
```

or in the number of covered threads:

```
if (i++ < 10) {
  dbgResult = float(i), i = 20;
}
```

Loops Since GLSL exposes no built-in loop counters, debugging statements at a specific user defined iteration inside loops, i.e. statements for which the corresponding call stack holds elements enclosed by a loop structure, requires the definition of a debug loop counter per nesting level. Having those counters defined at global scope allows for debugging a specific iteration even if the target is not directly an element of the loop body anymore, as it occurs by stepping into user-defined functions.

There are two alternatives for adding loop-aware debug code. One is inserting an `if` block that restricts its body to be evaluated only if the loop counter matches the requested iteration. However, the disadvantages of this approach are twofold: The `if` statement cannot be used in place for arbitrary expressions and insertion of this structure as debug code would increase the nesting level and may exceed the hardware given limit. Instead the logical-AND (`&&`) operator is utilized that evaluates the right-hand operand only if the left-hand operand evaluates to true. An additional `true` expression is appended to the debug assignment to assure a scalar boolean return type. An equivalent construct is possible using the ternary (`?:`) conditional operator. Both do not increase the nesting level and are very likely to map to conditional write masks on graphics hardware, as indicated by NVIDIA's cgc compiler.

```
int dbgIter0;
...
dbgIter0 = 0;
for (i = 10; i > 0; i--, dbgIter0++) {
  (dbgIter0 == 5 && (dbgResult = float(i), true)), f += f;
}
```

Function Calls To debug a statement inside a user-defined function, it is crucial to guarantee that the instrumented code is only executed for the single distinctive function invocation in question; but all other calls to this function remain unchanged. According to the terminology of the statements' *debug state*, as introduced in the context of the syntax analysis in Section 7.1.2, those function calls lie on the *debug path*; more specific, they are marked as *Target* and their corresponding function definition are of type *Path*. By definition of the OpenGL shading language, a single function may only occur once one the debug stack, as recursive function calls are not supported. Nevertheless, a debug target may still be nested inside multiple user-defined functions, that all need to be handled correctly. Therefore, those functions are duplicated and renamed, so that only the function call statement in question calls the relabeled debug instance. Code instrumentation is only performed in the copied function body, while the original function is reconstructed unmodified. This approach was chosen in favor of adding conditional code to the function body, as changes are thus limited to the debug call stack and evaluation of non-debugged calls remain completely unaffected by additionally generated code fragments.

```
void F(inout int p1, int p3, out int p4) { ... }
void F_debug(inout int p1, int p3, out int p4) {
  dbgResult = float(p1), ...
}
...
F(i, j, k);
F_debug(m, n, o);
```

Function Parameters Besides debugging statements inside user-defined function, it may be required to debug the state of a shader before a function call is actually issued, but after all parameters are already fully evaluated. In particular, this is of interest in cases in which the function parameters itself introduce side effects to the shader state, e.g. when using parameters that are again function calls or arbitrary expressions. The problem is similar to debugging conditionals after their test expression, as the comma literal in this context is already defined to separate the function parameters. The solution again uses a temporary register to buffer a function parameter in order to allow for in-place code instrumentation with an unchanged return value for the instrumented code fragment in respect to the original function parameter. The key element is to buffer the rightmost `in` parameter and insert code after its evaluation. This is sufficient, as `inout` and `out` parameters need to be l-values in GLSL and thus cannot feature any side effects to the variable state of the shader. An example for the retrieval of the debug state after the evaluation of all parameters, but prior to issuing the actual function call, is given in the following:

```
void F(inout int p1, int p3, out int p4);
...
int dbgParam;
F(i, (dbgParam = (k += j), dbgResult = k, dbgParam), k);
```

7.1.5 Host Application Instrumentation

As already mentioned, instrumenting the shader code is only one part of debug architecture. Since graphics applications often employ a number of different shader programs, it has to be possible for the user to select the one of interest. Furthermore, since the execution of a shader program is triggered solely by rendering geometry, a shader debugger must also provide a method for selecting the draw call of interest. This means that at least part of the functionality of conventional OpenGL state machine debuggers is required. Most importantly, a possibility for interactively stepping through OpenGL calls as they are invoked by the application.

Acting as an interface between the host and the debugged shader, the application instrumentation must provide the means for replacing the original shader program by an instrumented one, to execute it, and to read back the debug results, thereby ensuring that the host application is oblivious of those changes. In short, the general process of debugging a shader for a given draw call is as follows:

```
1) Setup a debug environment capturing the shader output
2) Record the target draw call
3) For each shader debugging request
       Inject instrumented shader in OpenGL state
       Replay draw call and read back debug result
4) Restore the original OpenGL state
5) Replay draw call to ensure correct continuation
```

For instrumenting the host application a combination of OpenGL command stream interception [BHH00] and the native `ptrace` debugger interface [Ins90] available on Linux systems is used. However, the presented approach is not limited to the Linux operating system. Other Unix systems as well as the MS Windows API provide equivalent functionality for monitoring and controlling the execution of processes from within a debugger. *Ptrace* is also used for transferring data between the address spaces of the host application and the graphical debugging environment. Additional communication between the debugger and the host application is handled through a shared memory segment (see Figure 7.1).

OpenGL command stream interception is realized by a shared library that provides function hooks, i.e. function definitions with the same signature, for all possible calls to the OpenGL and GLX library. Those wrapper functions are automatically generated from the interface declarations provided by the OpenGL and GLX C header files. Thus, the system trivially supports the full OpenGL standard and all vendor-specific extensions known at compile time. Mapping the debug library into the process space of the host application through the *preloading* mechanism provided by the dynamic linking facility of the operating system causes its exported symbols to take precedence over symbols of libraries occurring later in symbol lookup scope and therefore allows us to intercept all calls to the original OpenGL implementation. See for example [Dre06] for an in-depth discussion of dynamic-linking concepts. Again, a similar mechanism exists for MS Windows, named *DLL hooking* [HB99]. While *preloading* is a straight forward solution for intercepting function calls to dynamically loaded libraries it has its limitations. If an application obtains pointers to OpenGL functions by explicitly loading symbols from the OpenGL library using the programming interface of the dynamic linking loader, there is no way of intercepting those calls by *preloading*. Therefore, a wrapper function for the `dlsym` part of the programming interface of the dynamic linking loader is provided that exploits the `DEEP_BIND` functionality of recent GNU C library versions. This allows debugging of applications that access OpenGL functions through explicitly obtained function pointers instead of those provided by the OpenGL library or the `glXGetProcAddress` interface.

Basically, each wrapper function is responsible for the following three tasks: First, it must provide the debugger tool with information about the function that is called, i.e. the function name and its parameters. Second, it has to be possible to call the original function and, eventually, communicate the potential result of the call or an error that might have occurred to the debugger. And last, it has to provide the means of performing an arbitrary number of additional debug operations. This functionality is realized by suspending the normal execution of the debugged program immediately after entering a wrapper function and switching to a special debug command execution mode. First, the name of the called function as well as the addresses of its function parameters and their respective types are stored in the common shared memory segment. Then, a simple event loop is entered. The debugger can now access the information provided by the wrapper function and issue debug commands. Among others this includes functionality for recording the current call for later playback, replaying a previously recorded OpenGL stream, retrieving the currently active GLSL shader program, injecting a new debug shader, or reading back the results of a shader debug step. Most importantly, this also includes calling the original OpenGL function to guarantee proper operation of the host application or evoking the function call with modified parameter values in order to facilitate debugging.

Besides the described functionality for OpenGL call stepping, the preload library offers an additional immediate execution mode that allows running the program without actually interrupting program execution until a stop command from the debugger is received. This provides the possibility of user interaction with the traced program.

OpenGL Stream Recording and Playback Debugging shader programs requires to repeatedly render the geometry that triggers their execution. Although it possible to record and replay an arbitrary stream of OpenGL commands [DNB$^+$05], the described system is restricted to track single draw calls only. This means that either a single OpenGL function call, e.g. `glDrawArrays`, or an immediate mode stream of OpenGL commands delimited by `glBegin` and the corresponding `glEnd` call is recorded. Since an immediate mode stream may alter GL state, it is also necessary to save those parts of the state that may change inside such a block. Working on a per draw call level has several advantages. In comparison to relying on automatic or forced frame redraws, record and replay is much more fine-scaled and flexible, allowing for debugging animations and multi-pass algorithms.

However, the draw call record and playback method has its own pitfalls that have to be avoided. Rendering the same geometry multiple times invalidates OpenGL query objects and produces incorrect results if the application uses transform feedback mode, which results in incorrect behavior if program execution is continued after debugging a shader. However, those problems can be solved by taking special care of active query objects. In case of timer queries the solution is simple: For debugging purposes they can be just ignored, since the significance of timing results using an instrumented application program is highly questionable. On the other hand, the result of occlusion or primitive queries might be crucial for the correct operation of an algorithm. Therefore, the system keeps track of all active query objects during the debug process. When entering the shader debug stage, active queries are terminated and their current values are stored. The queries are restarted using the same query object names at the moment the user is leaving the shader debug stage. Subsequent requests for `QUERY_RESULT` are adapted to return the sum of saved and current query result. Dealing with active transform feedback mode is similar. Every time the host application calls `BeginTransformFeedbackNV`, it has to be checked whether a `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV` query is already active. If this is not the case, a separate query is initiated. Thus, the system can keep track of the number of primitives actually written to the buffer. Of course, queries started by the host have to be addressed as described before. An active transform feedback can now be terminated when entering the shader debug stage and it can be restarted using appropriate buffer offsets obtained from the primitive queries on exit.

7.1.6 Practical Considerations

Using this system for debugging a shader program is quite similar to using a traditional source level debugger. However, there are some differences due to the special characteristics of graphics hardware.

Interactive Shader Debugging From a user's point of view the typical work flow of a debug session is split into two major tasks. First, by using a combination of interactively executing the target application and OpenGL call stepping, the

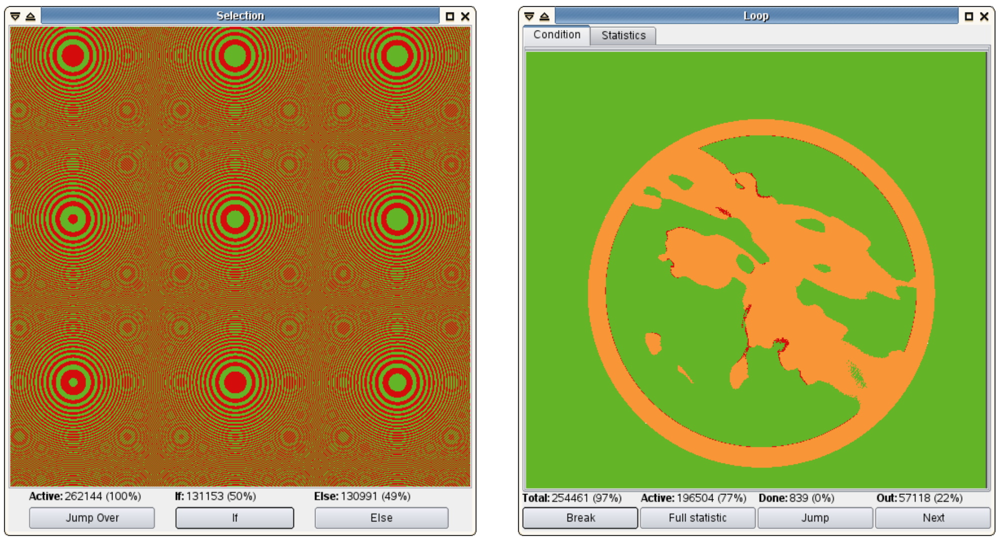


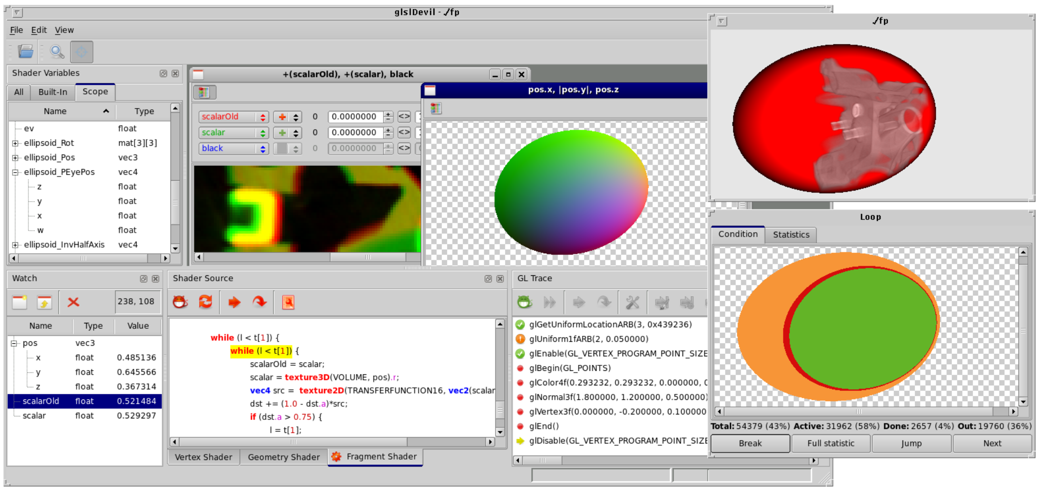
Figure 7.5: Graphical representation of flow control decisions for fragment shaders. Threads are classified in terms of the evaluation of the current conditional test; `true`/`false` test are marked green, and red respectively. For loops, orange colored threads represent shader instances that stopped processing the loop in a previous iteration. In addition to the visual representation statistical data are provided.

draw call of interest for shader debugging needs to be selected. For improved usability there is support for jumping to the next draw call, shader switch, or any user-specified OpenGL function as well as for optionally stopping program execution upon OpenGL errors. In addition to pure program flow inspection, it is possible to directly manipulate the OpenGL state machine by editing function call parameters, e.g. changing shader program uniforms, which proved to be very useful in finding issues related to an erroneous pipeline setup. In the next step, the user selects a target shader and starts debugging by single stepping through the code. Due to the parallel nature of the GPU, additional user interaction is required at program flow decisions, i.e. conditionals and loops. At each flow control instruction it is necessary for the user to specify which execution path of the control flow the debugging process should follow. In case of an *if-else* conditional statement the decision is to either continue debugging in the true branch or stepping into the false branch. For loop statements the options are to either break the loop execution or to proceed with the next loop iteration.

Depending on the user's choice debugging continues only in the selected branch and consequently only those threads matching this control flow are available for state inspection; the data of all other threads are masked to avoid inspection of invalid variable states as well as to visually indicate the sub-group of shader threads sharing the same program flow. As soon as the control flow of different branches merge again, e.g. after an `if-else` conditional, debugging continues with the merges sets of threads, i.e. debugging always covers all valid threads for the given debug target and debug stack. These decisions can be based on current debug data and are additionally supported by a per-element evaluation and visual representation of the corresponding condition (see Figure 7.5). Especially, a visual representation of flow control decisions of all threads executed in parallel helps to quickly identify outliers, e.g. threads with a disproportionate or even an infinite number of loop iterations. In particular for debugging fragment shaders, such visualizations of the flow control also provide hints to possible thread coherence, and thus, can provide references to overhead caused by the dynamic control flow. For example, the conditional statement debugged in Figure 7.5 on the left-hand side clearly shows a highly incoherent evaluation of the conditional test; ultimately forcing both conditional branches to be fully evaluated for the bigger part of the executed threads.

An alternative concept for solving the conflict of diverging program flow in the context of tracing parallel processes, is to restrict debugging to only a single thread or handle multiple threads completely sequentially. This way, state debugging is limited to the current thread in focus only, but on the other hand flow control decisions are implicitly given and debugging mimics the more usual concept of debugging a sequential process. Microsoft *PIX* [Mic07], originating from Microsoft's Performance Investigator for the *Xbox*, uses this paradigm for debugging DirectX vertex or pixel shaders in a completely sequential manner. In contrast, the proposed system is explicitly focused on preserving the underlying parallelism also for the debugging process. In particular, such an approach allows to identify coherent thread groups, to find correlations between threads with unexpected results, and enables to iteratively explore and compare the debug results of all executed shader instances. Especially the latter, provides detailed insight to differences between shader threads in respect to register content as well as flow control, which proved to be very useful for quickly investigating and review code statements and their effects on the shaders' output.

Similar to the common paradigm of sequential process debugging, at any time during the debug process the user can select variables from the current scope and add them to the list of active watch variables. The actual numerical values of watch variables can then be inspected on a per element level (vertex, primitive, or fragment) or by using specialized visual debugging tools. For fragment data this includes an image viewer that maps floating-point fragment data to a color image. While this is the canonical solution for fragments, in case of vertex data such an inherent geometrical analogy does not necessarily exists. Thus, two possibilities for inspection of vertex data is offered: table views and scatter plots. Figure 7.6



Name	Type
ev	float
ellipsoid_Rot	mat[3][3]
ellipsoid_Pos	vec3
ellipsoid_PEyePos	vec4
z	float
y	float
x	float
w	float
ellipsoid_InvHalfAxis	vec4

238, 108

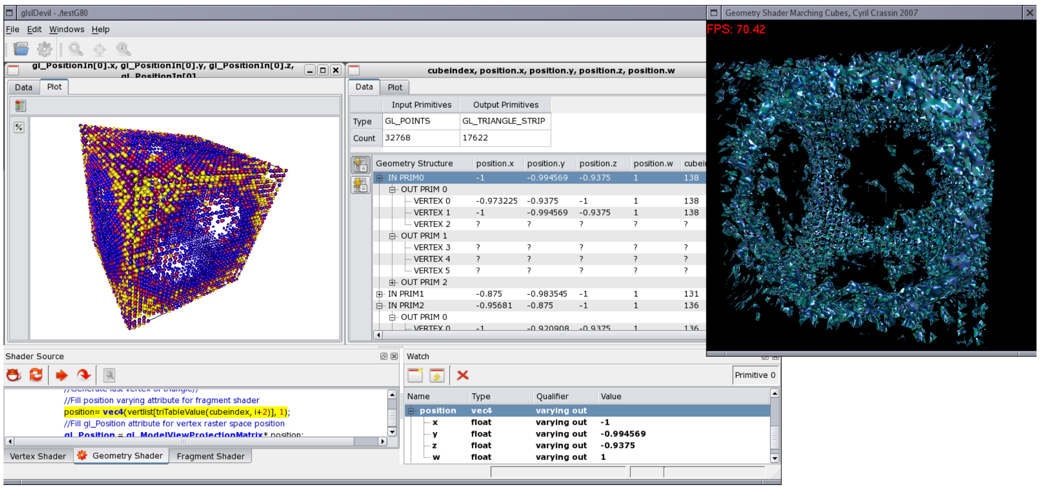
Name	Type	Value
pos	vec3	
x	float	0.485136
y	float	0.645566
z	float	0.367314
scalarOld	float	0.521484
scalar	float	0.529297

```
while (l < t[1]) {
    while (l < t[1]) {
        scalarOld = scalar;
        scalar = texture3D(VOLUME, pos).r;
        vec4 src =  texture2D(TRANSFERFUNCTION16, vec2(scalar
        dst += (1.0 - dst.a)*src;
        if (dst.a > 0.75) {
            l = t[1];
```

```
glGetUniformLocationARB(3, 0x439236)
glUniform1fARB(2, 0.050000)
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE
glBegin(GL_POINTS)
glColor4f(0.293232, 0.293232, 0.000000, 0
glNormal3f(1.800000, 1.200000, 0.500000)
glVertex3f(0.000000, -0.200000, 0.100000
glEnd()
glDisable(GL_VERTEX_PROGRAM_POINT_SIZE
```

Total: 54379 (43%) **Active:** 31962 (58%) **Done:** 2657 (4%) **Out:** 19760 (36%)

Figure 7.6: A typical debugging session: From the original application's (top right) OpenGL stream a single draw call is selected and the active fragment shader is debugged having shader source code, scope lists, variable data content, and graphical analysis tools at hand.


glslDevil - ./testG80
File Edit Windows Help
gl_PositionIn[0].x, gl_PositionIn[0].y, gl_PositionIn[0].z, gl_PositionIn[0]...
Data Plot
cubeindex, position.x, position.y, position.z, position.w
Geometry Shader Marching Cubes, Cyril Crassin 2007
FPS: 70.42
Shader Source
Watch
Primitive 0
Vertex Shader Geometry Shader Fragment Shader

	Input Primitives	Output Primitives
Type	GL_POINTS	GL_TRIANGLE_STRIP
Count	32768	17622

Geometry Structure	position.x	position.y	position.z	position.w	cubei
IN PRIM0	-1	-0.994569	-0.9375	1	138
OUT PRIM 0					
VERTEX 0	-0.973225	-0.9375	-1	1	138
VERTEX 1	-1	-0.994569	-0.9375	1	138
VERTEX 2	?	?	?	?	?
OUT PRIM 1					
VERTEX 3	?	?	?	?	?
VERTEX 4	?	?	?	?	?
VERTEX 5	?	?	?	?	?
OUT PRIM 2					
IN PRIM1	-0.875	-0.983545	-1	1	131
IN PRIM2	-0.95681	-0.875	-1	1	136
OUT PRIM 0					
VERTEX 0	-1	-0.920908	-0.9375	1	136

```
//Generate last vertex of triangle//
//Fill position varying attribute for fragment shader
position= vec4(vertlist[triTableValue(cubeindex, i+2)], 1);
//Fill gl_Position attribute for vertex raster space position
gl_Position = gl_ModelViewProjectionMatrix* position;
```

Name	Type	Qualifier	Value
position	vec4	varying out	
x	float	varying out	-1
y	float	varying out	-0.994569
z	float	varying out	-0.9375
w	float	varying out	1

Figure 7.7: Debugging a geometry shader: Data inspection in the tree view is performed on a per input primitive level, showing for each input primitive the resulting output primitives and their corresponding vertices. For already emitted vertices the variable state are shown. The scatter plot allows to visually inspect the debug data; the color depicts the Marching Cubes index for each input cell generating at least one output primitive.

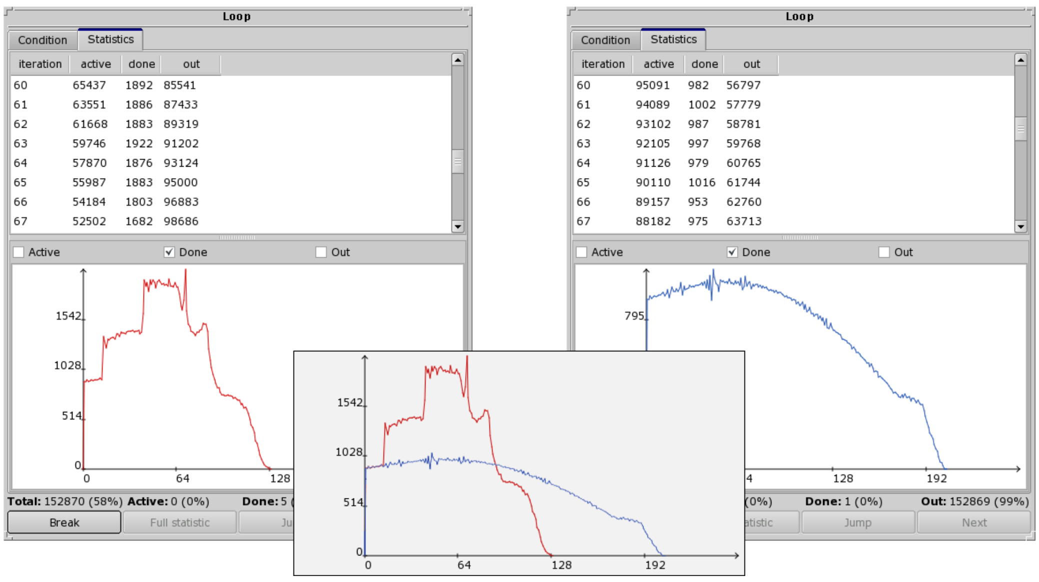
and 7.7 show screen shots of typical debug sessions for fragment and geometry shaders, depicting also the various methods for debug data inspection.

Limitations Although the described approach was designed to be very general, there are also some limitations. Most importantly, since shader instrumentation is done in a high-level language, it depends strongly on the correctness and reliability of the vendor-specific GLSL compiler and on the assumption that code manipulation, if performed in a semantically correct way, does not influence the program execution. In case of a driver or hardware bug, results are totally unpredictable, i.e. the bug may not manifest for the instrumented shader or, even worse, it will be triggered by the instrumentation in the first place. However, based on the practical experience current GLSL compilers seem to be quite mature.

Other limitations concern the proposed solution for debugging vertex and geometry shaders: The necessary OpenGL extensions are currently only supported on NVIDIA GPUs, starting with the G80 architecture. Furthermore, the described approach does not allow to debug vertex programs that operate on primitives generated by the OpenGL display lists mechanism. This is due to missing possibility to determine what kind of primitives will be submitted by the execution of a display list. However, transform feedback mode requires specification of the correct output primitive type.

Advanced Analysis Combined instrumentation of the host application and the shader enables analysis functionality that goes beyond traditional debugging. Even though, the presented code instrumentation does not allow to retrieve any data from the graphics processor that could not also be retrieved by manually adding *printf*-style statements, the automated retrieval directly inside the target application during runtime offers additional benefit for shader debugging, that would be tedious and cumbersome to obtain manually.

As an example for such advanced debug features, an in-depth loop profiling tool was implemented that summarizes statistical data for all iterations of a loop. Until the loop iteration terminates, i.e. all shader instances dropped out of the loop due to a `break` statement or the loop conditional, the loop profiler automatically collects the number of iterating threads per loop iteration. Based on this data, diagrams for the number of active threads and the number of threads dropping out of the loop for each iteration are generated. This facilitates, for example, identifying the critical path of a shader or allows to quickly detect an unintentional high amount of loop iterations as cause for a low shader performance. A different application example is given in Figure 7.8. Here the collected data is used to analyze and compare the efficiency of early ray termination in a volume rendering application.



iteration	active	done	out
60	65437	1892	85541
61	63551	1886	87433
62	61668	1883	89319
63	59746	1922	91202
64	57870	1876	93124
65	55987	1883	95000
66	54184	1803	96883
67	52502	1682	98686

Total: 152870 (58%) **Active:** 0 (0%) **Done:** 5

iteration	active	done	out
60	95091	982	56797
61	94089	1002	57779
62	93102	987	58781
63	92105	997	59768
64	91126	979	60765
65	90110	1016	61744
66	89157	953	62760
67	88182	975	63713

(0%) **Done:** 1 (0%) **Out:** 152869 (99%)

Figure 7.8: Using the loop analysis tool for comparing the efficiency of a volume rendering example with (left) and without (right) early ray termination. The middle graph shows an overlay of the automatically collected debug data for both setups. Note the higher number of fragments (y-axis) breaking out of the loop per iteration (x-axis) in the first case as soon as the accumulated alpha exceeds the given threshold; the number of loop iterations substantially decrease.

Based on the debug access to arbitrary shader data, additional analysis tools could be easily integrated. Some of the ideas for future developments include the collection of the number of sample instructions to a given texture target per thread, or more general, the possibility to obtain the statistical distribution of how often a specific function, built-in or user-defined, is called per thread. The other way round, it would also be possible to obtain the local distribution of data accesses for a given texture target. That way, the debugger could also be used to identify access patterns, unused texture regions, or errors due to misplaced data fetches more easily.

7.2 GLSL Shader Performance Analysis

While high level shader languages provide means for lowering the initial learning curve and allow for more rapid development, making optimal use of the underlying hardware is an essential requirement, especially for interactive and real-time applications, e.g. games or scientific visualizations. However, writing truly efficient shader code and optimizing existing code is still a challenging task that requires long trained experience, detailed knowledge of the underlying hardware, or may even only be based on trial-and-error. On the one hand, this is due to the rapid evolution of graphics hardware, that requires a constant adaption of the shader design in order to to be most efficient and may even completely invalidate optimization strategies effective for prior hardware generations. On the other hand, until recently support for detailed performance analysis of the shader stages was completely missing. Even though, the support for performance analysis of the rendering pipeline, both in hardware and in the accompanying development tools, substantially improved in the last few years, especially shader profiling is still a challenge due to the underlying highly parallel execution model.

On way to provide insight to the performance of the shader stages are synthetic benchmarks, specifically designed to measure the performance on an instruction level or to reveal specific characteristics of the stream processors that execute the shader threads. For example, Diepstraten and Eissele [DE04, ED06] applied this concept to evaluate the performance of arithmetic as well as texture instructions for DirectX 9 pixel shaders. In a similar way, the benchmark tool *GPUBench* [BFH04a] uses synthetic shaders to evaluate texture fetch latencies in respect to arithmetic instructions. In the context of GPGPU computing Volkov and Demmel [VD08] and more recently Wong et al. [WPSAM10] utilized *microbenchmarks*—a suite of synthetic kernel programs in combination with a highly precise test bed for performance measurements—to derive low-level characteristics of modern graphics architectures, like instruction latencies, instruction throughput, as well as cache sizes. While all those tools are perfectly suitable to deduce universal development guideline for shader programming for each graphics processor, the application of this rules to the optimization process of a complex shader still requires lots of experience and expert knowledge; and the actual evaluation of the findings in the context of the target application still requires means for profiling as part of this application.

In contrast, cycle emulators allows to directly analyze the target shader and enable to draw detailed conclusions of the expected performance as well as the efficiency on the target hardware. In particular, such tools, like the AMD *GPU ShaderAnalyzer* or NVIDIA's *ShaderPerf*, prove to be of great value during shader optimization, as code changes can be directly mapped to necessary clock cycles—even for hardware not at hand or various graphics driver versions. Thus, cycle emulators are also an integral component of commercial development kits, e.g. Microsoft's *PIX* for the *Xbox 360* or *GCM Replay* for the Sony *Playstation 3*. Based on the freely available tools—only few information of the professional variants is publicly available—the downside of cycle emulation is that costs can only be determined for the longest or shortest path of the profiled shader. While this offers the possibility of determining the lower and upper bound of performance of a shader, it is not possible to judge the performance upon the actual input provided by the target application; nor does this technique allow to analyze the actual distribution of costs across all executed threads for the target frame in order to identify the fragments with the effective critical path.

A more recent interface for performance analyses of the complete rendering pipeline is provided by low-level performance counters, that provides access to hardware based profiling capabilities. With respect to shader profiling, such counters include the number of GPU clock ticks an experiment ran for high-performance time measurements or even the number of cycles a distinct shader unit was busy executing a fragment shader. Besides application interfaces also vendor specific development tools are available, e.g. AMD *GPU PerfStudio* or NVIDIA's *PerfKit*, to allow for application-transparent inspection of such counters at runtime. Unfortunately, up to now such hardware counters do not expose the possibility to analyze the shader executions on a lower level than for a distinct draw call, but not for a single shader instance. In addition, this functionality only comes at costs for being vendor specific and functionalities change with hardware generations, and may only be queried in combination with a specialized driver version that introduces additional overhead and is less often updated. Therefore, for this thesis the focus was on designing a profiling method that can provide insight to the performance of a distinct shader thread in the context of the target application, but still works without those low level counters; even though, the optional combination of the methods can be used to further increase timing preciseness.

Based on the functionality of the available developer tools and the requirements derived during shader development in the cause of this thesis, profiling in this context is primarily targeted on providing detailed performance data to enable the analysis of the partitioning of rendering costs per pixel in the environment of the host application. The goal is to measure the time spend for each processed pixel of a given target draw call individually to uncover the distribution of the shader execution costs in image space. Analogous to shader debugging, profiling is supposed to be executed by means of OpenGL command stream interception and shader code instrumentation; allowing the profiling tool to be used at run-time in an application-transparent way without the need for any code rewriting.

7.2.1 Profiling Environment

Shader profiling for a single, user-defined target pixel requires two building blocks: First, a method for isolating a single target shader thread for performance analysis, or in other words, a render setup that only evaluates the single target pixel. And second, an accurate way of measuring the corresponding performance. Based on those blocks, analyzing the rendering costs of a frame in image space can be performed by sequentially measuring all pixels in the requested region of interest.

A brute-force approach on implementing this sequence of operations could be based on *view frustum culling* and time measurement over multiple iterations of the same rendering call. In more detail, the selection of a single pixel can be achieved by adapting the viewport and the projection matrix to restrict fragment generation to a single pixel in image space. While this may be the easiest way to select a specific target pixel, this technique does not offer sufficient timing quality. The reason for this is that only the costs for fragment processing are reduced while the costs for all other stages of the rendering pipeline are kept completely unchanged. Most importantly, this affect the issue of the draw call and the driver overhead on the CPU as well as vertex and geometry processing on the GPU. In the worst case, fragment processing is completely hidden in the inaccuracy of timing all other stages; preventing to draw any conclusions from the collected data. Also repeated invocation of the draw call per timing measurement—a common way of increasing timing accuracy—does not offer any benefit in this case.

Profiling Render Pass To enable high accuracy time measurements for a CPU clock based approach, it is necessary to perform a single test multiple times and integrate measurement over the time span of all tests. But to avoid high CPU overhead the repetition of tests is optimally performed completely on the GPU without any CPU intervention. At the same time the overhead of all GPU stages except fragment processing needs to be reduced to a constant minimum for not influencing the overall timing. For the proposed profiling method, this is achieved by rendering an instance of a single target pixel multiple times in parallel. In more detail, all fragments of a full screen sized quad perform exactly the same calculation for a single pixel that is currently profiled. By doing so the number of repetitions is scaled with the viewport size and without any significant CPU overhead. In fact issuing only a single quad may even be more cost effective, i.e. enable a higher timing precision, than replaying the actual target rendering call. At the same time vertex and geometry overhead on the GPU is also cut down to process only a single quad; reducing the probability for the influencing execution times of the fragment shader to a minimum.

Shader Code Instrumentation In order to assure that every fragment performs the very same calculation for a single render pass, several changes to the profiled target shader are necessary. The crucial input to the fragment shader written in GLSL—that causes every fragment to perform different calculations—are *varying in* parameters. If the same input is given to those parameters for every fragment in the viewport, it is assured that all perform the identical calculation. Thus, to select a single target pixel for profiling, it is sufficient to assign the values originating from that pixel to the corresponding *varying in* parameters of every fragment. To collect this data, a preprocessing stage queries all varying variables used in the unmodified target shader for all executed shader threads. Given that this task is identical to debug those variables at the beginning of the shader, the concept of shader debugging as described in Section 7.1 can be applied; in one debug render pass per *varying in* parameter, all necessary data for all shader instances for an arbitrary target pixel are collected. In the second stage, the code of the profiled shader is instrumented, so that there are no dependency to any varying variable anymore. This is simply achieved by replacing those varying input variables by uniform registers that the original varying data of the single target pixel.

Implementation For shader profiling the same basic mechanism are necessary as they were already introduced for application-transparent shader debugging in Section 7.1.1. For the prototypical evaluation of the proposed method, the concept was embedded into the debugger project to allow for application-transparent fragment shader profiling. All necessary changes to the source code, i.e. *varying in* replacement, are carried out on the intermediate language level from which the compiler back-end reconstructs the actual instrumented target shader. The time measurement is simply based on the standard CPU clock interface for Linux systems, i.e. `gettimeofday`, in order to show the independence of the system to specialized CPU-based high-performance counters or GPU-based hardware performance counters.

Limitations While the proposed rendering technique offers great scalability with nearly no additional overhead, thus enabling very accurate timing measurements, it is important to note that the render method introduces several side effects that affect the expressiveness or even the correctness of the recorded profiling data. In general, all those side effects are the result of interrelation between distinct shader threads, either on an algorithmic level or on the hardware level or both. From an algorithmic point of view, such interactions between threads can be explicitly given in form of the GLSL built-in derivative operations, i.e. the *dFdx*, *dFdy*, and *fWidth* function. In the underlying execution model, such functions exchange data between unspecified direct neighbors with respect to the thread's raster position. Thus, the proposed profiling method cannot guarantee semantic equivalence to the original shader execution in case such a functionality is used—even in case no code instrumentation would be performed—as the render setup purposely changes the complete environment of the target pixel to be homogeneous. Ultimately, this may lead to a wrong evaluation of the target shader instance, and thus, may invalidate the obtained profiling data. On a hardware level, those interrelations are also heavily used for texture fetch operations, i.e. for the automatic determination of *mip-map* levels and for anisotropic texture lookups. Thus, also for those cases the correctness of the profiling cannot be fully guaranteed. However, this only affects shaders for which the actual shader workload is truly dependent on such neighborhood data; in case the sequence of operations completely remain the same, the resulting data may be negatively affected—which is in deed undesirable—but performance evaluation still maintains its expressiveness. In addition, measuring of the low level hardware characteristics, e.g. texture caching, latencies for data fetches, or thread coherence, are not possible in general, as the render method severely changes the data access pattern and the flow control of all fragments to fully match the selected target pixel. However, the proposed method still is able to provide quantitative performance statements, as the data access pattern of the profiling render pass for each target pixel corresponds to the lower bound of the overhead for all data fetch operations, i.e. the hardware can make optimal usage of inter thread data caching. Although the discussion as well as the actual implementation solely focus on fragment shaders, the proposed concept is fully extensible to vertex or even geometry shaders; even with the advantage, that those shader types intrinsically do not suffer from the mentioned limitations as they do not expose any correlation between distinct shader threads.

7.2.2 Sample Analysis

In order to show the potential of the described profiling approach, the single-pass ray casting method for volume rendering, as described in Section 3.3, served as a testbed for the performance analysis. The used fragment shader approximates the *volume rendering integral* by iteratively combining pre-integrated volume samples with the *over*-operator; for the benchmarks no optimization strategies, i.e. early-ray termination, were applied. The final rendering result of the analyzed frame is shown in Figure 7.9(a). The complete series of tests were all performed on a NVIDIA 8800GT based graphics board, and thus, the analysis of the obtained results is naturally limited to the underlying G92 chip architecture.

For each pixel in the viewport, covered by the volume's front faces, the evaluation of the shader performance was measured in a separate rendering pass in order to gain insight to the overall distribution of rendering costs in image space. The obtained raw data thus form a two-dimensional uniform grid, resembling the pixel positions in the output buffer, with the qualitative rendering time attached to each grid sample, i.e. fragment shader instance. A color-coded visualization with a one-to-one correspondence to the original render output of Figure 7.9(a) is given in Figure 7.9(b). An alternative visualization of this data is in addition provided in Figure 7.9(c) as hight-field representation. In the following the three most important features affecting the rendering costs of a single pixel—originating from both the render algorithm and the graphics hardware—are discusses on the basis of this profiling data.

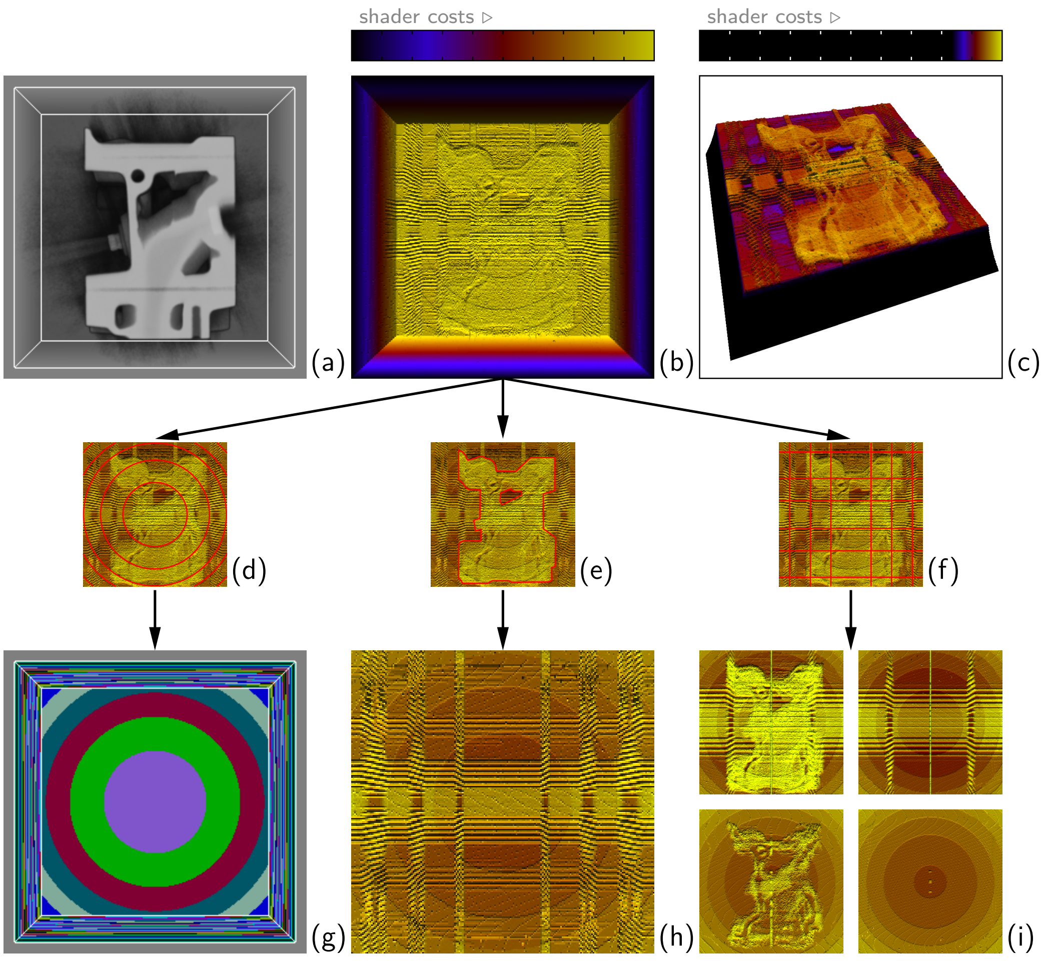


Figure 7.9: Results for profiling the sample application of volume rendering. Showing the original rendered image (a) and the obtained results as color coded, shaded two-dimensional view (b) and recolored three-dimensional view of the back face (c). The three most important effects on per-pixel rendering times, that become visible with profiling, are highlighted in the middle row (d-f). Evidence of the dependency of the rendering times to the number of sampling iterations (g), the higher overhead for pre-integrated classification in high-frequency volume regions (h), and 3D texture cache strategies (i) become clearly visible.

Sampling Iterations In case of volume ray casting, the number of sample iterations required using a constant sample distance is directly dependent to the length of the ray segment inside the volume. This length varies largely in image space due to the applied perspective projection, i.e. rays leaving the volume through a lateral surface on average require fewer sample steps that those hitting the back face. As each ray sampling step corresponds to a single loop iteration in the shader code, the overall render costs should be highly dependent to the required sampling iterations. The evidence of this interrelation is clearly visible in the obtained profiling data; in Figure 7.9(b) this results in the pyramidal increase of execution times at the volume border. For the same reason rays hitting the backplane of the volume also differ in the number of ray sampling steps, with rays directed towards the edges or corners of the backplane requiring more loop iterations. This effect is also visible in the profiling data, leading to concentric circles (highlighted in Figure 7.9(d)) with increasing overall rendering costs from inside to outside of the back plane. Comparing the profiling results to the true number of sample steps per pixel (see Figure 7.9(g))—obtained by debugging a manually added loop counter—reveals that the circle-shaped discontinuities of the profiler perfectly match to the actual loop iterations; thus, the method of profiling accurately captures and reflects the underlying characteristic of the render algorithm, and the fragment shader respectively.

Pre-Integrated Classification Unexpected at first, the obtained profiling data—the render times per pixel—show a clear notion of the engine data set (highlighted in Figure 7.9(e)). With respect to the shader code, this may seem quite inexpiable as the control flow and order of execution of all shader operations is completely independent of the sampled volume. Comparing this result to the profiling data of an identical test rendering a same sized, but completely homogeneous data set (see Figure 7.9(h)) indicates that the effect is caused by the classification process. Each texture samples' contribution to color accumulation along the ray is derived by a dependent lookup into a 2D texture that stores the pre-integrated approximation to the volume rendering integral for the sampling intervals (cp. Section 3.2.2). However, the performance of dependent texture lookup—most likely the latency of the corresponding texture fetch operation—is highly dependent on the texture access pattern. While a ray is traced through homogeneous regions the transfer function is repeatedly sampled at the same location, thus allowing for more efficient use of the texture cache. On the opposite, the access pattern becomes quite random in regions of high frequencies, i.e. inside the engine block, leading to the overall higher rendering costs for those pixels (best visible in Figure 7.9(c)).

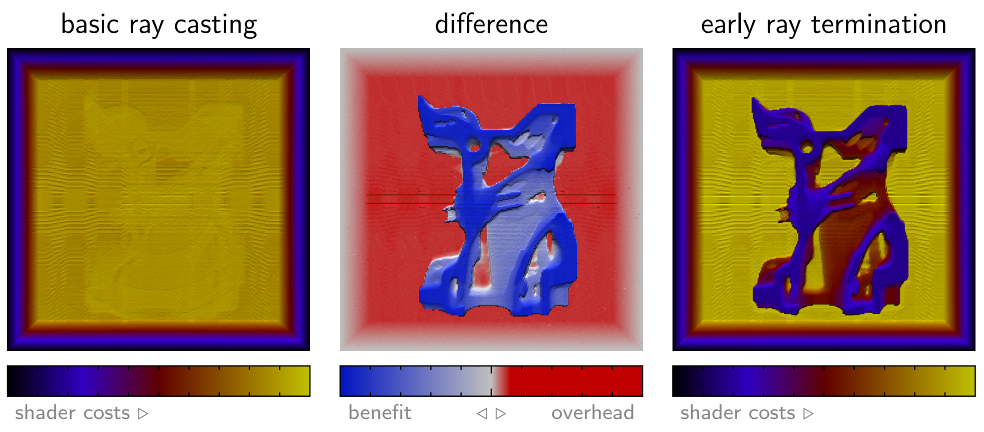


Figure 7.10: Analysis of the efficiency of early ray termination for single-pass ray casting in the test setup. The profiling data for rendering the data set with and without the acceleration technique are shown on the right and left side, respectively. Profiling also enables a detailed comparison per shader thread (middle image) that reveals the distinct benefits in the regions of high opacity (blue), but also shows the newly introduced overhead for the conditional ray termination (red).

Volume Sampling The profiling results also feature a prominent, regular typed pattern that covers the whole volume texture (highlighted in Figure 7.9(f)). Comparisons to down sampled versions of the original $256^2{\times}110$ sized data set indicate a direct relation to the volume's texture size. Figure 7.9(i) shows the results for data set down sampled to $128^2{\times}55$ and $64^2{\times}26$, both for the engine and a homogeneous data set side by side. Based on this comparison and the way these structures are changing with the actual data size, changes in rendering costs are very likely to be the result of the volume sampling, i.e. the texture cache strategy used for three-dimensional textures. For example, in the x-dimension it is notable that render costs tend to increase for rays that cross a regular grid pattern, repetitive every 32 voxels, independent of the used texture sizes.

Conclusion Interpreting the collected analysis data may still be quite difficult in some cases and may also require more detailed knowledge of the underlying hardware architecture—arguably even exceeding what is publicly documented right now. However, the introduced approach to shader profiling proved to be accurate enough to capture low-level performance characteristics and the obtained results provide detailed insight to the distribution of the overall rendering costs. Although only quantitative conclusions can be drawn from this method—for accurate qualitative conclusions for a single thread execution the actual scheduling of threads to the stream processors of the underlying hardware as well as their corresponding load would be required—the proposed profiling method offers a fast and application transparent way to analyze fragment shader performance on a per pixel basis. An exemplary use case for such a profiling method during shader development is given in the analysis of *early ray termination* for the previously described test scenario. Figure 7.10 depicts a side-by-side comparison of the profile results as well as an evaluation of the per-pixel benefit or overhead for volume rendering with and without this acceleration technique. In contrast to evaluating frames per seconds only, the more detailed performance analysis clearly reveals the balance between the newly introduced performance overhead, which is a direct consequence of extending the shader code, and the benefit drawn from potentially terminating the rays early on. In addition, the analysis of the shader performance on a per pixel basis also emphasizes the strong dependency of this optimization method on the rendered data set and the applied transfer function. In a more general sense, the developer is enabled to relate the effects of shader code changes more directly to the changes in execution performance. And what is likely even more important, such an analysis also provides support in understanding the overall performance characteristics of an optimization approach—including spatial dependencies—early on in the development cycle. Thus, even within the discussed limitations this technique might still be very useful for evaluating algorithmic or plain code optimizations.

Chapter 8 Conclusion

GPU clusters quickly evolved from a pure prototype architecture—being the focus of basic research itself—into an important vehicle to drive research and discovery in a broad spectrum of domains. Predominantly being focused on large-scale data visualization only, graphics clusters are nowadays also employed to solve some of the most demanding challenges in bio-informatics, computational fluid dynamics, computational finance, molecular dynamics, or weather forecasting. This upward trend also comes along with a quickly increasing number of large-scale GPU cluster installations worldwide. Some recent examples include the compute cluster at the Max Planck Institute Göttingen (340 GPUs), the latest cluster computer of the Commonwealth Scientific and Industrial Research Organisation (256 GPUs), or the *Tianhe-I* cluster operated by China's National University of Defense Technology (5.120 GPUs)—currently being the world's 5th most powerful supercomputer. Those top spots for hybrid cluster systems, covering all GPU-enabled systems but also the second-ranked *PowerXCell*-accelerated *Roadrunner* supercomputer at the Los Alamos National Laboratory, clearly indicate the potential of such hybrid architectures to further scale the performance of distributed systems. This is of particular importance as some of the previously dominant driving forces behind compute performance, such as clock frequencies or chip fabrication processes, might hit their physical limitations, progress at a significantly slower pace, or come with immense costs for any further incremental development. Hence, hybrid cluster systems might provide the most feasible solution to maintain the progress in performance development in order to be able to solve tomorrow's most demanding research challenges.

Finding suitable algorithms and methods to efficiently utilize all the distributed processing power of those massively parallel processors as well as the hierarchical systems architectures built upon those units, hence, will continue to become increasingly important. In the context of GPU clusters this naturally includes efficient distributed visualization techniques—as discussed in Chapter 5 using the example of distributed interactive volume visualization—but nowadays also covers efficient parallel algorithms for general purpose computations. The latter is of particular interest for two reasons. First, with the most recent push into GPU computing, namely by the introduction of NVIDIA's Fermi architecture, graphics processors now are capable to provide all important key features necessary for high performance computing. Especially with the large improvement in double precision performance as well as the increased reliability due to error correction, GPUs are likely to become an interesting processing platform for more and more compute intense applications. And second, with all the available compute capabilities GPUs can finally serve as vehicle for both, data generation and data analysis—a flexibility that was long reserved for CPUs only. Consequently, new challenges arise to not only utilize all available processing power for optimally solving a single task at a time, but also to allow for efficiently sharing those processing resources of both, the CPU and the GPU, to simultaneously execute compute and visualization tasks. While such interplay of simulation and visualization was recently demonstrated and evaluated in the context of very large scale supercomputers [Ma09, YWG$^+$10], it is still open for investigation if such highly efficient cooperation is achievable on GPU cluster architectures as well.

Alongside the efforts to increase the efficiency of GPU cluster systems for various types of application an equally important key challenge is to improve the effectiveness in working with such a complex system architecture. In particular, as graphics clusters become more widespread and get applied to a broader field of application—far outside the GPU's classical core domains, such as interactive rendering and visualization.

High level abstraction layers are well suited to hide some of this complexity from the developer and also allow for utilizing various levels of parallelism in a fully user-transparent way—enabling developers to employ graphics processors or cluster environments without necessarily requiring detailed knowledge about programming GPUs or about communicating over a network. In the context of GPU development a variety of such libraries were introduced over the last couple of years. For example, AMD as well as NVIDIA provide their own GPU-enabled implementation of a *Basic Linear Algebra Subprograms* (BLAS) library allowing to speed up existing applications and algorithms by offloading suitable math computations to the GPU. While some minor code modifications are necessary, those libraries transparently expose highly efficient GPU algorithms via an already established, well known programming interface. In the very same way *Thrust* [HB09] provides an open source GPU-accelerated C++ template library mimicking the interface of the *C++ Standard Template Library* (STL) or *OpenVIDIA* [FMA05] combines various algorithms for computer vision especially designed for the parallel execution model of modern graphics processors.

In line with this approach of hiding some of the complexity from the user in order to enable more effective development, the compute programming environment introduced in Chapter 6 represents one example of such a high level abstraction layer in the context of GPU clusters. Based on an existing compute programming interface for graphics processors additional levels of abstraction for multiple GPUs in a single system as well as interconnected GPU cluster systems are introduced. That way, a developer accustomed to program a single GPU with CUDA is enabled to scale the computation into a graphics cluster environment, without having to learn any new programming interface and without having to leave the single methodology for parallel programming exposed by the graphics processor.

Two most recent technological advancements of the underlying CUDA programming interface further boost the interoperability of the graphics processor with all other system components of a GPU cluster—further improving the efficiency and practicability of the discussed compute environment. *Zero-copy* allows to directly access pinned system memory from the graphics processor, which eliminates the absolute need to always upload all data to the graphics memory first. Especially, for data that is only accessed sparsely—an information CUDASA could evaluate via the memory descriptor functions—costly data transfers can be avoided to further improve kernel execution latencies. In addition, CUDA-enabled graphics processors will shortly be able to communicate more efficiently with InfiniBand network adapters by sharing common pinned system memory, which is expected to significantly lessen the overall communication bottleneck for the network layer of the CUDASA environment—and any GPU cluster application, that heavily relies on network communication, in general.

Another way to ease programming—and thus to increase effectiveness—is to provide support through developer tools, such as highly optimizing compilers, integrated development environments, and debuggers. This support for the developer is well established for programming the central processor and even in the context of massively parallel CPU cluster applications several tool sets are available that allow to debug and analyze parallel program execution and network communication. However, for programming graphics processors such developer support was largely lagging behind the exposed programmability of modern GPUs.

That gap motivated the work presented in Chapter 7, which ultimately led to a publicly available GPU-based debugger for a high level graphics shader programming language that allows to inspect true GPU data from all programmable shader stages of the OpenGL render pipeline. Having picked up more and more interest from academia as well as from hardware vendors, several related debugging concepts and tools were recently presented or announced. NVIDIA's Shader Debugger plug-in for FX Composer—an integrated development environment for shader authoring—enables debugging of OpenGL fragment shaders by using a similar approach as discussed in the context of this thesis. Likewise AMD's recently released GPU PerfStudio 2.2 exposes a comparable set of features for DirectX pixel shaders. Sharif and Lee [SL08] introduced a capture and replay technique that allows to selectively emulate shader pipeline stages on the CPU. By using input data captured directly from the GPU, specific shader invocations of interest for debugging can thus be analysed on the CPU. In the context of general purpose computations on graphics hardware Hou et al. [HZG09] introduced a debugging framework that uses clever code instrumentation to enable system callbacks from within GPU code. That way, arbitrary CPU code can be triggered from any location inside the GPU kernel, which is shown to enable debugging of massively parallel compute kernels. Unfortunately, the latter two systems are not released to the public at the time of writing.

Just recently NVIDIA introduced *CUDA-GDB* and publicly announced *Parallel Nsight*, which both provide true native GPU debugging support by making use of low level hardware functionality for debugging. Being by design much closer to the debugging concepts available for CPUs, this new class of developer tools clearly mark the next big step towards highly improved developer support for GPU programming.

Being one of the few publicly available GPU debuggers—and the only one supporting OpenGL shader debugging of the vertex, geometry, and fragment stages up to date—the discussed debugging application still enjoys high interest with lots of positive user feedback, advanced feature request, as well as a still constant download rate. Exceeding more than 9,000 downloads since its first public release, it also testifies the strong request for such developer tools for graphics hardware and likewise shows the urgent need to further advance those tools to open up effective GPU programming to an even broader audience.

Bibliography

[3Dl05] 3DLABS CORPORATION. OpenGL Shading Language Compiler Front-End, 2005.

[AAN05] A. ADAMSON, M. ALEXA, AND A. NEALEN. Adaptive Sampling of Intersectable Models Exploiting Image and Object-Space Coherence. In *Proceedings of Symposium on Interactive 3D Graphics and Games 2005*, pages 171–178. ACM, 2005.

[ACCC04] F. ABRAHAM, W. CELES, R. CERQUEIRA, AND J. L. CAMPOS. A Load-Balancing Strategy for Sort-First Distributed Rendering. In *Proceedings of Symposium on Computer Graphics and Image Processing 2004*, pages 292–299. IEEE Computer Society, 2004.

[Ake93] K. AKELEY. Reality Engine Graphics. In J. T. KAJIYA, editor, *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 109–116. ACM Press / ACM SIGGRAPH, 1993.

[AMD06] AMD: ADVANCED MICRO DEVICES. ATI CTM Guide, Technical Reference Manual, 2006.

[AMD07a] AMD: ADVANCED MICRO DEVICES. AMD Compute Abstraction Layer Programming Guide, 2007.

[AMD07b] AMD: ADVANCED MICRO DEVICES. Brook+ Programming Guide, 2007.

[AMD07c] AMD: ADVANCED MICRO DEVICES. Compute Abstraction Layer (CAL), Technology Intermediate Language (IL) Reference Manual, 2007.

[AMD07d] AMD: ADVANCED MICRO DEVICES. R600 Technology, R600-Family Instruction Set Architecture, 2007.

[AP98] J. AHRENS AND J. PAINTER. Efficient Sort-Last Rendering Using Compression-Based Image Compositing. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 98*, pages 145–151. Eurographics Association, 1998.

[ARC05] A. Abdul-Rahman and M. Chen. Spectral Volume Rendering Based on the Kubelka-Munk Theory. *Computer Graphics Forum*, 24(3):413–422, 2005.

[BBFZ00] W. Blanke, R. Bajaj, D. Fussell, and X. Zhang. The Metabuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System Using Commodity Rendering Engines. Technical Report Tr2000-16, University of Texas at Austin, 2000.

[BBJL05] A. Benassarou, E. Bittar, N. W. John, and L. Lucas. MC Slicing for Volume Rendering Applications. In *Computational Science*, volume 3515 of *Lecture Notes in Computer Science*, pages 314–321. Springer, 2005.

[BFH04a] I. Buck, K. Fatahalian, and P. Hanrahan. GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications. ACM Workshop on General Purpose Computing on Graphics Processors 2004 Poster Session, 2004.

[BFH$^+$04b] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[BHH00] I. Buck, G. Humphreys, and P. Hanrahan. Tracking Graphics State for Networked Rendering. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 2000*, pages 87–95. ACM, 2000.

[Bli94] J. F. Blinn. Compositing, Part 1: Theory. *IEEE Computer Graphics and Applications*, 14(5):83–87, 1994.

[BMDF02] S. Bergner, T. Möller, M. S. Drew, and G. D. Finlayson. Interactive Spectral Volume Rendering. In *Proceedings of IEEE Visualization 2002*, pages 101–108. IEEE Computer Society, 2002.

[BMTD05] S. Bergner, T. Möller, M. Tory, and M.S. Drew. A Practical Approach to Spectral Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(2):207–216, 2005.

[BMWM06] S. Bergner, T. Möller, D. Weiskopf, and D. J. Muraki. A Spectral Analysis of Function Composition and its Implications for Sampling in Direct Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1353–1360, 2006.

[BPS97] C. L. Bajaj, V. Pascucci, and D. R. Schikore. The Contour Spectrum. In *Proceedings of IEEE Visualization 97*, pages 167–175. IEEE Computer Society, 1997.

[BS04] J. Braun and M. Sambridge. A Numerical Method for Solving Partial Differential Equations on Highly Irregular Evolving Grids. *Nature*, 376:655–660, 2004.

[BSWE06] S. Bachthaler, M. Strengert, D. Weiskopf, and T. Ertl. Parallel Texture-Based Vector Field Visualization on Curved Surfaces Using GPU Cluster Computers. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2006*, pages 75–82. Eurographics Association, 2006.

[BTCH05] B. A. Barsky, M. J. Tobias, D. P. Chu, and D. R. Horn. Elimination of Artifacts due to Occlusion and Discretization Problems in Image Space Blurring Techniques. *Graphical Models*, 67(6):584–599, 2005.

[Buc03] I. Buck. Brook Language Specification, 2003.

[Bur81] P. J. Burt. Fast Filter Transforms for Image Processing. *Computer Graphics and Image Processing*, 16(1):20–51, 1981.

[Car93] I. Carlbom. Optimal Filter Design for Volume Reconstruction and Visualization. In *Proceedings of IEEE Visualization 93*, pages 54–61. IEEE Computer Society, 1993.

[CC78] E. Catmull and J. Clark. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer-Aided Design*, 10(6):350–355, 1978.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of IEEE Symposium on Volume Visualization 94*, pages 91–98. ACM, 1994.

[CDF$^+$03] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic Octree Load Balancing Using Space-Filling Curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[CDM06] H. Childs, M. Duchaineau, and K.-L. Ma. A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Proceedings of*

EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2006, pages 153–162. Eurographics Association, 2006.

[Cha50] S. CHANDRASEKHAR. *Radiative Transfer.* Clarendon Press, Oxford, 1950.

[CJ81] P. CHANG AND R. JAIN. A Multi-Processor System for Hidden-Surface-Removal. *Computer Graphics (Proceedings of SIGGRAPH 81)*, 15(4):405–436, 1981.

[CMF05] X. CAVIN, C. MION, AND A. FILBOIS. COTS Cluster-Based Sort-Last Rendering: Performance Evaluation and Pipelined Implementation. In *Proceedings of IEEE Visualization 2005*, pages 111–118. IEEE Computer Society, 2005.

[CN93] T. J. CULLIP AND U. NEUMANN. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993.

[CS94] D. COHEN AND Z. SHEFFER. Proximity Clouds—An Acceleration Technique for 3D Grid Traversal. *The Visual Computer*, 11(1):27–38, 1994.

[DE04] J. DIEPSTRATEN AND M. EISSELE. *Shader X3*, chapter 7.1, *In-Depth Performance Analyses of DirectX9 Shading Hardware concerning Pixel Shader and Texture Performance*, pages 523–544. Charles River Media, 2004.

[Del34] B. N. DELAUNAY. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, 6:793–800, 1934.

[Dem04] J. DEMERS. *GPU Gems*, chapter 23, *Depth of Field: A Survey of Techniques*, pages 375–390. Addison Wesley, 2004.

[DH92] J. DANSKIN AND P. HANRAHAN. Fast Algorithms for Volume Ray Tracing. In *Proceedings of IEEE Symposium on Volume Visualization 92*, pages 91–98. ACM, 1992.

[Dir50] G. L. DIRICHLET. Über die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen. *Journal für die reine und angewandte Mathematik*, 40:209–227, 1850.

[DLD$^+$03] W. J. DALLY, F. LABONTE, A. DAS, P. HANRAHAN, J.-H. AHN, J. GUMMARAJU, M. EREZ, N. JAYASENA, I. BUCK, T. J. KNIGHT, AND U. J. KAPASI. Merrimac: Supercomputing with Streams. In

Proceedings of ACM/IEEE Conference on Supercomputing 2000, pages 35–43. IEEE Computer Society, 2003.

[DNB$^+$05] N. Duca, K. Niski, J. Bilodeau, M. Bolitho, Y. Chen, and J. Cohen. A Relational Debugging Engine for the Graphics Pipeline. *ACM Transactions on Graphics*, 24(3):453–463, 2005.

[Dre06] U. Drepper. How to Write Shared Libraries. Technical report, Red Hat Inc., 2006.

[DRS08] C. Dyken, M. Reimers, and J. Seland. Real-Time GPU Silhouette Refinement Using Adaptively Blended Bézier Patches. *Computer Graphics Forum*, 27(1):1–12, 2008.

[DS78] D. Doo and M. Sabin. Behaviour of Recursive Division Surfaces Near Extraordinary Points. *Computer-Aided Design*, 10(6):356–360, 1978.

[DZTS07] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. GPU Marching Cubes on Shader Model 3.0 and 4.0. Technical Report MPI-I-2007-4-006, Max-Planck-Institut für Informatik, 2007.

[ED06] M. Eissele and J. Diepstraten. *Shader X4*, chapter 8.6, *GPU Performance of DirectX 9 Per-Fragment Operations Revisited*, pages 541–560. Charles River Media, 2006.

[EHK$^+$06] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics.* A. K. Peters Ltd., 2006.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 2000*, pages 9–16. ACM, 2001.

[EMP09] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, 2009.

[FAM$^+$05] N. Fout, H. Akiba, K.-L. Ma, A. Lefohn, and J. Kniss. High Quality Rendering of Compressed Volume Data Formats. In *Proceedings of Eurographics/IEEE VGTC Symposium on Visualization 2005*, pages 77–84, 2005.

[FDK84] L. A. Feldkamp, L. C. Davis, and K. W. Kress. Practical Cone-Beam Algorithm. *Journal of the Optical Society of America*, 1(6):612–619, 1984.

[FE10] S. FREY AND T. ERTL. PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2010*. Eurographics Association, 2010. accepted for publication.

[FMA05] J. FUNG, S. MANN, AND C. AIMON. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of ACM Conference on Multimedia 2005*, pages 6–11. ACM, 2005.

[FPE$^+$89] H. FUCHS, J. POULTON, J. EYLES, T. GREER, J. GOLDFEATHER, D. ELLSWORTH, S. MOLNAR, G. TURK, B. TEBBS, AND L. ISRAEL. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23:79–88, 1989.

[Fre08] S. FREY. GPU-basierte Kegelstrahlrekonstruktion großer CT-Datensätze. *Diploma thesis*, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, 2008.

[GBP07] G. GUENNEBAUD, L. BARTHE, AND M. PAULIN. High-Quality Adaptive Soft Shadow Mapping. *Computer Graphics Forum*, 26(3):525–534, 2007.

[GGSC96] S. J. GORTLER, R. GRZESZCZUK, R. SZELISKI, AND M. F. COHEN. The Lumigraph. In *Computer Graphics (Proceedings of SIGGRAPH 96)*, volume 30, pages 43–54. ACM, 1996.

[GH03] S. GUTHE AND P. HECKBERT. Non-Power-of-Two Mipmap Creation. Technical Report TR-01838-001, NVIDIA Corporation, 2003.

[Gla89] A. S. GLASSNER. How to Derive a Spectrum from an RGB Triplet. *IEEE Computer Graphics and Applications*, 9(4):95–99, 1989.

[Gla94] A. S. GLASSNER. *Principles of Digital Image Synthesis.* Morgan Kaufmann Publishers Inc., 1994.

[GR90] B. GUDMUNDSSON AND M. RANDÉN. Incremental Generation of Projections of CT-Volumes. In *Proceedings of Visualization in Biomedical Computing 90*, pages 27–34, 1990.

[Gre05] S. GREEN. Image Processing Tricks in OpenGL. Presentation at Games Developer Convention 2005, NVIDIA Corporation, 2005.

[GS01] S. Guthe and W. Strasser. Real-Time Decompression and Visualization of Animated Volume Data. In *Proceedings of IEEE Visualization 2001*, pages 349–356. IEEE Computer Society, 2001.

[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization 2002*, pages 53–60. IEEE Computer Society, 2002.

[GWLS05] J. Gao, C. Wang, L. Li, and H.-W. Shen. A Parallel Multiresolution Volume Rendering Algorithm for Large Data Visualization. *Parallel Computing*, 31(2):185–204, 2005.

[Haa10] A. Haar. Zur Theorie der orthogonalen Funktionensysteme (Erste Mitteilung). *Mathematische Annalen*, 69:331–371, 1910.

[Har71] R. L. Hardy. Multiquadric Equations of Topography and Other Irregular Surfaces. *Journal of Geophysical Research*, 76:1906–1915, 1971.

[Har07] M. Harris. Parallel Prefix Sum (Scan) with CUDA. Technical report, NVIDIA Corporation, 2007.

[HB99] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 Functions. In *Proceedings of USENIX Windows NT Symposium 99*, pages 135–143, 1999.

[HB09] J. Hoberock and N. Bell. Thrust: A Parallel Template Library, 2009.

[HBSL03] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2003*, pages 92–101. Eurographics Association, 2003.

[HEB$^+$01] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In E. Fiume, editor, *Proceedings of SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 129–140. ACM Press / ACM SIGGRAPH, 2001.

[HF85] M.-C. Hu and J. D. Foley. Parallel Processing Approaches To Hidden Surface Removal in Image Space. *Computers & Graphics*, 9(3):303–217, 1985.

[HHKP96] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of Transfer Functions with Stochastic Search Techniques. In *Proceedings of IEEE Visualization 96*, pages 227–235. IEEE Computer Society, 1996.

[HHN+02] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In T. Appolloni, editor, *Proceedings of SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 693–702. ACM Press / ACM SIGGRAPH, 2002.

[HM90] R. B. Haber and D. A. McNabb. *Visualization in Scientific Computing*, chapter 2.2 *Visualization Idioms: A Conceptual Model for Scientific Visualization Systems*, pages 74–93. IEEE, 1990.

[HSS+05] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[Hsu93] W. M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of IEEE Symposium on Parallel Rendering 1993*, pages 7–14. IEEE Computer Society, 1993.

[HZG09] Q. Hou, K. Zhou, and B. Guo. Debugging GPU Stream Programs through Automatic Dataflow Recording and Visualization. *ACM Transactions on Graphics*, 28(5):1–11, 2009.

[ICG86] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A Radiosity Method for Non-Diffuse Environments. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 133–142. ACM, 1986.

[Ins90] Institute of Electrical and Electronics Engineers, Inc. IEEE Std 1003.1-1990, Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API), 1990.

[Int99] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C.* International Organization for Standardization, 1999.

[JKM01] T. J. Jankun-Kelly and K.-L. Ma. Visualization Exploration and Encapsulation via a Spreadsheet-Like Interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[JWH$^+$04] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proceedings of Eurographics/IEEE VGTC Symposium on Visualization 2004*, pages 35–44, 2004.

[Kaj86] J. T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150. ACM, 1986.

[KD98] G. Kindlmann and J. W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *Proceedings of IEEE Visualization 98*, pages 79–86. IEEE Computer Society, 1998.

[KE02] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2002*, pages 7–15. Eurographics Association, 2002.

[Kes06] J. Kessenich. The OpenGL Shading Language, 2006. Language Version: 1.20.

[KES07] M. Kraus, M. Eissele, and M. Strengert. GPU-Based Edge Directed Image Interpolation. In *Image Analysis*, volume 4522 of *Lecture Notes in Computer Science*, pages 532–541. Springer, 2007.

[KG79a] M. Kaplan and D. P. Greenberg. Parallel Processing Techniques for Hidden Surface Removal. *Computer Graphics (Proceedings of SIGGRAPH 79)*, 13(2):300–307, 1979.

[KG79b] D. Scott Kay and D. Greenberg. Transparency for Computer Synthesized Images. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, volume 13, pages 158–164. ACM, 1979.

[KG02] A. König and E. Gröller. Mastering Transfer Function Specification by Using VolumePro Technology. In *Proceedings of Spring Conference on Computer Graphics 2002*, volume 17, pages 279–286, 2002.

[Khr09] Khronos OpenCL Working Group. The OpenCL Specification, 2009. Version: 1.0.

[KKH01] J. Kniss, G. Kindlmann, and C. Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proceedings of IEEE Visualization 2001*, pages 255–262. IEEE Computer Society, 2001.

[Kle08] T. Klein. *Exploiting Programmable Graphics Hardware for Interactive Visualization of 3D Data Fields.* PhD thesis, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, 2008.

[KM31] P. Kubelka and F. Munk. Ein Beitrag zur Optik der Farbanstriche. *Zeitschrift für technische Physik*, 12:593–601, 1931.

[KPHE02] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive Translucent Volume Rendering and Procedural Modeling. In *Proceedings of IEEE Visualization 2002*, pages 109–116. IEEE Computer Society, 2002.

[KPI$^+$03] J. Kniss, S. Premoze, M. Ikits, A. Lefohn, C. Hansen, and E. Praun. Gaussian Transfer Functions for Multi-Field Volume Visualization. In *Proceedings of IEEE Visualization 2003*, pages 497–504. IEEE Computer Society, 2003.

[Kra03] M. Kraus. *Direct Volume Visualization of Geometrically Unpleasant Meshes.* PhD thesis, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, 2003.

[Kra09a] M. Kraus. Quasi-Convolution Pyramidal Blurring. *Virtual Reality and Broadcasting*, 6(6):155–162, 2009.

[Kra09b] M. Kraus. The Pull-Push Algorithm Revisited - Improvements, Computation of Point Densities, and GPU Implementation. In *Proceedings of Conference on Computer Graphics Theory 2009*, pages 179–184. INSTICC Press, 2009.

[KS07a] M. Kraus and M. Strengert. Depth-of-Field Rendering by Pyramidal Image Processing. *Computer Graphics Forum*, 26(3):645–654, 2007.

[KS07b] M. Kraus and M. Strengert. Pyramid Filters Based on Bilinear Interpolation. In *Proceedings of Conference on Computer Graphics Theory 2007*, pages 21–28. INSTICC Press, 2007.

[KSKE07] M. Kraus, M. Strengert, T. Klein, and T. Ertl. Adaptive Sampling in Three Dimensions for Volume Rendering on GPUs. In *Proceedings of Asia Pacific Symposium on Visualization 2007*, pages 113–120, 2007.

[KSSE05] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume

Raycasting on Graphics Hardware. In *Proceedings of IEEE Visualization 2005*, pages 223–230. IEEE Computer Society, 2005.

[Kub48] P. Kubelka. New Contributions to the Optics of Intensely Light-Scattering Materials. Part I. *Journal of the Optical Society of America*, 38(5):448–457, 1948.

[Kub54] P. Kubelka. New Contributions to the Optics of Intensely Light-Scattering Materials. Part II: Nonhomogeneous Layers. *Journal of the Optical Society of America*, 44(4):330–335, 1954.

[KW03a] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization 2003*, pages 287–292. IEEE Computer Society, 2003.

[KW03b] J. Krüger and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[LC87] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21:163–169, 1987.

[Lev88] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[Lev90] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[LH91] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 285–288. ACM, 1991.

[LHJ99] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proceedings of IEEE Visualization 99*, pages 355–361. IEEE Computer Society, 1999.

[LHN05] S. Lefebvre, S. Hornus, and F. Neyret. *GPU Gems 2*, chapter 37, *Octree Textures on the GPU*, pages 595–613. Addison Wesley, 2005.

[Lju06] P. Ljung. Adaptive Sampling in Single Pass, GPU-based Raycasting of Multiresolution Volumes. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2006*, pages 39–46. IEEE Computer Society, 2006.

[LMS$^+$01] S. LOMBEYDA, L. MOLL, M. SHAND, D. BREEN, AND A. HEIRICH. Scalable Interactive Volume Rendering Using Off-the-Shelf Components. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2001*, pages 115–121. IEEE Computer Society, 2001.

[LRN96] T.-Y. LEE, C. S. RAGHAVENDRA, AND J. B. NICHOLAS. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[LWM04] E. LUM, B. WILSON, AND K.-L. MA. High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. In *Proceedings of Eurographics/IEEE VGTC Symposium on Visualization 2004*, pages 25–34, 2004.

[Ma09] K.-L. MA. In Situ Visualization at Extreme Scale: Challenges and Opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[MAB$^+$97] J. MARKS, B. ANDALMAN, P. A. BEARDSLEY, W. FREEMAN, S. GIBSON, J. HODGINS, T. KANG, B. MIRTICH, H. PFISTER, W. RUML, K. RYALL, J. SEIMS, AND S. SHIEBER. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In T. WHITTED, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 389–400. ACM Press / ACM SIGGRAPH, 1997.

[Mag04] M. MAGALLÓN. *Hardware Accelerated Volume Visualization on PC Clusters.* PhD thesis, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, 2004.

[Max95] N. MAX. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[MC00] K.-L. MA AND D. M. CAMP. High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network. In *Proceedings of ACM/IEEE Conference on Supercomputing 2000*, volume 29. IEEE Computer Society, 2000.

[McC00] M. D. MCCOOL. Smash: A Next-Generation API for Programmable Graphics Accelerators. Technical Report CS-2000-14, University of Waterloo, 2000.

[MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26:231–240, 1992.

[Mes96] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, 1996.

[MFS+09] C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, 2009.

[MHC90] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 27–33. ACM, 1990.

[MHE01] M. Magallón, M. Hopf, and T. Ertl. Parallel Volume Rendering Using PC Graphics Hardware. In *Proceedings of IEEE Conference on Computer Graphics and Applications 2001*, pages 384–389. IEEE Computer Society, 2001.

[Mic07] Microsoft Corporation. PIX: Direct3D Debugging and Analysis Tool, DirectX 10 SDK, 2007.

[Mit87] D. P. Mitchell. Generating Antialiased Images at Low Sampling Densities. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 65–72. ACM, 1987.

[MKC08] R. Marroquim, M. Kraus, and P. R. Cavalcanti. Efficient Image Reconstruction for Point-Based and Line-Based Rendering. *Computers & Graphics*, 32(2):189–203, 2008.

[MKS98] M. Meissner, U. Kanus, and W. Strassner. Vizard II: A PCI-Card for Real-Time Volume Rendering. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 98*, pages 61–67. ACM, 1998.

[ML94] S. R. Marschner and R. J. Lobb. An Evaluation of Reconstruction Filters for Volume Rendering. In *Proceedings of IEEE Visualization 94*, pages 100–107. IEEE Computer Society, 1994.

[MMD06] S. Marchesin, C. Mongenet, and J.-M. Dischler. Dynamic Load Balancing for Parallel Volume Rendering. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2006*, pages 51–58. Eurographics Association, 2006.

[MMMY97] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and Design of Filters Using a Taylor Series Expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.

[MN04] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Proceedings of Conference on Compiler Construction 2004*, pages 73–88, 2004.

[Möb27] A. F. Möbius. *Der barycentrische Calcul*. Johann Ambrosius Barth, 1827.

[MOM+01] S. Muraki, M. Ogata, K.-L. Ma, K. Koshizuka, K. Kajihara, X. Liu, Y. Nagano, and K. Shimokawa. Next-Generation Visual Supercomputing Using PC Clusters with Volume Graphics Hardware Devices. In *Proceedings of ACM/IEEE Conference on Supercomputing 2001*, volume 51. IEEE Computer Society, 2001.

[MPHK94] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa. Shader Metaprogramming. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2002*, pages 57–68. Eurographics Association, 2002.

[MSE06] C. Müller, M. Strengert, and T. Ertl. Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2006*, pages 59–66. Eurographics Association, 2006.

[MSE07] C. Müller, M. Strengert, and T. Ertl. Adaptive Load Balancing for Raycasting of Non-Uniformly Bricked Volumes. *Parallel Computing*, 33(6):406–419, 2007.

[MSH99] L. Moll, M. Shand, and A. Heirich. Sepia: Scalable 3D Compositing Using PCI Pamette. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines 99*, pages 146–155. IEEE Computer Society, 1999.

[Mue95] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics 95*, pages 75–84. ACM, 1995.

[Mül05] F. Müller. Verteilte Visualisierung und Kompression zeitabhängiger Volumendatensätze. *Diploma thesis*, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, 2005.

[MvL00] J. D. Mulder and R. van Liere. Fast Perception-Based Depth of Field Rendering. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology 2000*, pages 129–133. ACM, 2000.

[MWMS07] B. Moloney, D. Weiskopf, T. Möller, and M. Strengert. Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2007*, pages 45–52. Eurographics Association, 2007.

[MZ08] B. S. Michel and H. Zima. Bridging Multicore's Programmability Gap. ACM/IEEE Conference on Supercomputing 2008 Workshop, 2008.

[MZD09] B. S. Michel, H. Zima, and N. Desai. User Experience and Advances in Bridging Multicore's Programmability Gap. ACM/IEEE Conference on Supercomputing 2009 Workshop, 2009.

[Neu93] U. Neumann. Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *Proceedings of IEEE Symposium on Parallel Rendering 93*, pages 97–104. IEEE Computer Society, 1993.

[NHL94] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. In *Proceedings of ACM/IEEE Conference on Supercomputing 1994*, pages 340–349. IEEE Computer Society, 1994.

[NvdVS00] H. J. Noordmans, H. T. van der Voort, and A. W. Smeulders. Spectral Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):196–207, 2000.

[NVI06a] NVIDIA CORPORATION. NVIDIA GeForce 8800 GPU Architecture Overview. Technical Report TB-02787-001, 2006.

[NVI06b] NVIDIA CORPORATION. NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x), 2006.

[NVI07] NVIDIA CORPORATION. CUDA Programming Guide, 2007.

[NVI08] NVIDIA CORPORATION. PTX: Parallel Thread Execution, 2008. ISA Version: 1.2.

[OABB85] J. M. OGDEN, E. H. ADELSON, J. R. BERGEN, AND P. J. BURT. Pyramid-based Computer Graphics. *RCA Engineer*, 30(5):4–15, 1985.

[PD84] T. PORTER AND T. DUFF. Compositing Digital Images. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 253–259. ACM, 1984.

[PDC+03] T. J. PURCELL, C. DONNER, M. CAMMARANO, H. WANN JENSEN, AND P. HANRAHAN. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2003*, pages 41–50. Eurographics Association, 2003.

[PH04] M. PHARR AND G. HUMPHREYS. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., 2004.

[PHK+99] H. PFISTER, J. HARDENBERGH, J. KNITTEL, H. LAUER, AND L. SEILER. The VolumePro Real-Time Ray-Casting System. In A. ROCKWOOD, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 251–260. ACM Press / ACM SIGGRAPH, 1999.

[PS03] T. J. PURCELL AND P. SEN. Shadesmith, 2003.

[PSG06] M. PEERCY, M. SEGAL, AND D. GERSTMANN. A Performance-Oriented Data Parallel Virtual Machine for GPUs. *ACM Transactions on Graphics*, 25(3):184–192, 2006.

[PYRM08] T. PETERKA, H. YU, R. ROSS, AND K.-L. MA. Parallel Volume Rendering on the IBM Blue Gene/P. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2008*, pages 73–80. Eurographics Association, 2008.

[RC01] D. Rodgman and M. Chen. Refraction in Discrete Raytracing. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2001*, pages 3–17. IEEE Computer Society, 2001.

[RC06] D. Rodgman and M. Chen. Refraction in Volume Graphics. *Graphical Models*, 68(5):432–450, 2006.

[RE02] S. Röttger and T. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2002*, pages 23–28. IEEE Computer Society, 2002.

[RGW$^+$03] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of Eurographics/IEEE VGTC Symposium on Visualization 2003*, pages 231–238, 2003.

[RKE00] S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection. In *Proceedings of IEEE Visualization 2000*, pages 109–116. IEEE Computer Society, 2000.

[Rok93] P. Rokita. Fast Generation of Depth of Field Effects in Computer Graphics. *Computers & Graphics*, 17(5):593–595, 1993.

[RS00] H. Ray and D. Silver. The RACE II Engine for Real-Time Volume Rendering. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 2000*, pages 129–136. ACM, 2000.

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 51–58. ACM, 1988.

[SBM94] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proceedings of IEEE Symposium on Volume Visualization 94*, pages 83–89. ACM, 1994.

[ŠBSG06] P. Šereda, A. Vilanova Bartroli, I. W. O. Serlie, and F. A. Gerritsen. Visualization of Boundaries in Volumetric Data Sets Using LH Histograms. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):208–218, 2006.

[Sch05] A. Schuster. Radiation Through a Foggy Atmosphere. *Astrophysical Journal*, 21:1–22, 1905.

[SCM99] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. In *Proceedings of IEEE Visualization 99*, pages 371âĂŞ–377. IEEE Computer Society, 1999.

[SCS$^+$08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core X86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[SDWE03] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. Widening the Remote Visualization Bottleneck. In *Proceedings of Symposium on Image and Signal Processing and Analysis 2003*, pages 174–179, 2003.

[SEP$^+$01] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In J. F. Hughes, editor, *Proceedings of SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 141–148. ACM Press / ACM SIGGRAPH, 2001.

[SFL01] R. Samanta, T. Funkhouser, and K. Li. Parallel Rendering with k-Way Replication. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2001*, pages 75–84. IEEE Computer Society, 2001.

[SGS91] C. D. Shaw, M. Green, and J. Schaeffer. *Advances in Computer Graphics Hardware III*, chapter 12, *A VLSI Architecture for Image Composition*, pages 183–199. Springer, 1991.

[SH05] C. Sigg and M. Hadwiger. *GPU Gems 2*, chapter 20, *Fast Third-Order Texture Filtering*, pages 313–329. Addison Wesley, 2005.

[She68] D. Shepard. A Two-Dimensional Interpolation Function for Irregularly-Spaced Data. In *Proceedings of ACM National Conference 68*, pages 517–524. ACM, 1968.

[Sib81] R. Sibson. *Interpreting Multivariate Data*, chapter 2, *A Brief Description of Natural Neighbor Interpolation*, pages 21–36. John Wiley and Sons Ltd., 1981.

[SKE06] M. Strengert, M. Kraus, and T. Ertl. Pyramid Methods in GPU-Based Image Processing. In *Proceedings of Workshop on Vision, Modelling, and Visualization 2006*, pages 169–176, 2006.

[SKE07] M. Strengert, T. Klein, and T. Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007*, pages 81–88. Eurographics Association, 2007.

[SL08] A. Sharif and H.-H. S. Lee. Total Recall: A Debugging Framework for GPUs. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2008*, pages 13–20. Eurographics Association, 2008.

[SMDE08] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2008*, pages 49–56. Eurographics Association, 2008.

[SML$^+$03] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003*, pages 6–12. IEEE Computer Society, 2003.

[SMW$^+$04] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2004*, pages 41–48. Eurographics Association, 2004.

[SMW$^+$05] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Large Volume Visualization of Compressed Time-Dependent Datasets on GPU Clusters. *Parallel Computing*, 31(2):205–219, 2005.

[SSGH01] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture Mapping Progressive Meshes. In E. Fiume, editor, *Proceedings of SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 409–416. ACM Press / ACM SIGGRAPH, 2001.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-Based Raycasting. In *Proceedings of Eurographics/IEEE VGTC Workshop on Volume Graphics 2005*, pages 187–195, 2005.

[SSN$^+$98] Y. Sato, N. Shiraga, S. Nakajima, S. Tamura, and R. Kikinis. Local Maximum Intensity Projection: A New Rendering Method for Vascular Visualization. *Computer Assisted Tomography*, 22(6):912–917, 1998.

[ST90] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24:63–70, 1990.

[SW03] J. Schneider and R. Westermann. Compression Domain Volume Rendering. In *Proceedings of IEEE Visualization 2003*, pages 293–300. IEEE Computer Society, 2003.

[TBU00] P. Thèvenaz, T. Blu, and M. Unser. Interpolation Revisited. *IEEE Transactions on Medical Imaging*, 19(7):739–758, 2000.

[TLM05] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma. An Intelligent System Approach to Higher-Dimensional Classification of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, 2005.

[TPO06] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Technical Report MSR-TR-2005-184, Microsoft Corporation, 2006.

[VD08] V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of ACM/IEEE Conference on Supercomputing 2008*, pages 1–11. IEEE Computer Society, 2008.

[Vor08] G. Voronoi. Nouvelles Applications des Paramètres continus à la Théorie des formse Quadratiques. *Journal für die reine und angewandte Mathematik*, 134:198–287, 1908.

[VSE06] J. Vollrath, T. Schafhitzel, and T. Ertl. Employing Complex GPU Data Structures for the Interactive Visualization of Adaptive Mesh Refinement Data. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2006*, pages 55–58. IEEE Computer Society, 2006.

[Wes90] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 367–376. ACM, 1990.

[WF87] A. J. Worsey and G. Farin. An n-Dimensional Clough-Tocher Interpolant. *Constructive Approximation*, 3(1):99–110, 1987.

[WG91] J. Wilhelms and A. Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 275–284. ACM, 1991.

[WGS04] C. Wang, J. Gao, and H.-W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Proceedings of EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2004*, pages 23–30. Eurographics Association, 2004.

[Whi80] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[Whi92] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering.* A. K. Peters Ltd., 1992.

[WKME03] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization 2003*, pages 44–52. IEEE Computer Society, 2003.

[WM92] P. L. Williams and N. Max. A Volume Density Optical Model. In *Proceedings of the Workshop on Volume Visualization 92*, pages 61–68. ACM, 1992.

[WMFC02] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral Projection Using Vertex Shaders. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2002*, pages 7–12. IEEE Computer Society, 2002.

[WMG98] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation for Volume Sampling. In *Proceedings of IEEE Symposium on Volume Visualization 98*, pages 135–142. ACM, 1998.

[WMS98] P. L. Williams, N. L. Max, and C. M. Stein. A High Accuracy Volume Renderer for Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, 1998.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Proceedings of IEEE Symposium on Performance Analysis of Systems and Software 2010*, pages 28–40. IEEE Computer Society, 2010.

[WS82] G. WYSZECKI AND W. S. STILES. *Color Science Concepts and Methods, Quantitative Data and Formulae.* Wiley-Interscience Publication, 1982.

[WS01] R. WESTERMANN AND B. SEVENICH. Accelerated Volume Raycasting Using Texture Mapping. In *Proceedings of IEEE Visualization 2001*, pages 271–278. IEEE Computer Society, 2001.

[WSK02] M. WAN, A. SADIQ, AND A. KAUFMAN. Fast and Reliable Space Leaping for Interactive Volume Rendering. In *Proceedings of IEEE Visualization 2002*, pages 195–202. IEEE Computer Society, 2002.

[WWH+00] M. WEILER, R. WESTERMANN, C. HANSEN, K. ZIMMERMANN, AND T. ERTL. Level-of-Detail Volume Rendering via 3D Textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2000*, pages 7–13. IEEE Computer Society, 2000.

[YS93] R. YAGEL AND Z. SHI. Accelerating Volume Animation by Space-Leaping. In *Proceedings of IEEE Visualization 93*, pages 62–69. IEEE Computer Society, 1993.

[YWG+10] H. YU, C. WANG, R. W. GROUT, J. H. CHEN, AND K.-L. MA. In-Situ Visualization for Large-Scale Combustion Simulations. *IEEE Computer Graphics and Applications*, 2010.

[YWM08] H. YU, C. WANG, AND K.-L. MA. Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In *Proceedings of ACM/IEEE Conference on Supercomputing 2008*. IEEE Computer Society, 2008.

[ZSD+00] D. ZORIN, P. SCHRÖDER, T. DEROSE, L. KOBBELT, A. LEVIN, AND W. SWELDENS. Subdivision for Modeling and Animation. ACM SIGGRAPH 2000 Course Notes, 2000.

[ZTTS06] G. ZIEGLER, A. TEVS, C. THEOBALT, AND H.-P. SEIDEL. On-the-Fly Point Clouds through Histogram Pyramids. In *Proceedings of Workshop on Vision, Modelling, and Visualization 2006*, pages 137–144, 2006.