Institut für Informatik
der Technischen Universität München

A GPU Framework for Interactive Simulation and Rendering of Fluid Effects

Jens Harald Krüger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende: Univ.-Prof. Gudrun J. Klinker, Ph.D.

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Rüdiger Westermann
2. Hon.-Prof. Dr. Hans-Peter Seidel, Universität des Saarlandes

Die Dissertation wurde am 12.10.2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.11.2006 angenommen.

To my wife, my family, and friends.

Abstract

In the evolution of virtual reality environments and computer games we observe an ever-growing demand for realistic effects such as swashing water, rising smoke, dancing fire, explosions or caustics. To be used in such environments, interactive simulation and rendering techniques have to be realized on consumer class PCs.

This thesis aims at exploiting graphics processing units (GPUs) for this purpose. We target the GPU as a numeric coprocessor for a couple of reasons. First, aside from their current utility, GPUs present the first commercially successful examples of a class of future computing architectures key to high-performance, cost-effective supercomputing. Mainly driven by computer games, GPUs have now evolved to a fully programmable and highly parallel vector processor. Second, as the simulation is carried out on the GPU, simulation results are already where they are needed for display—in local GPU memory. This eliminates CPU-GPU data transfer, which is likely to become the bottleneck otherwise.

To be able to implement general techniques of numerical computing on the GPU, a GPU abstraction for this kind of application is required. Therefore, we have developed a linear algebra framework. This framework allows the programmer to abstract from the underlying GPU data structures and algorithms, and to focus on the application itself rather than the GPU implementation. Based on this framework we have implemented efficient algorithms for solving large systems of linear equations, and we have used these algorithms to solve partial differential equations such as the wave equation or the Navier-Stokes equations on GPUs.

The proposed framework facilitates the use of numerical simulation techniques to drive real-time visual effects. By using simulation results to advect geometric primitives on the GPU, saving displaced positions in graphics memory, and then sending these positions through the GPU again to obtain images in the frame buffer, a variety of different effects can be generated. In combination with particle primitives and 3D textures, for the first time ever is it now possible to simulate and render dynamic 3D effects in real time on consumer class PCs.

In addition to physics-based simulation of fluid phenomena, in this thesis we also present new techniques to simulate optical effects caused by such phenomena; i.e., caustics that can appear whenever light impinges upon reflecting or transmitting material. The proposed techniques neither require any preprocessing nor an intermediate radiance representation, and they can thus deal efficiently with dynamic scenery and scenery that is modified, or even created on the GPU.

Zusammenfassung

In der Entwicklung von Computerspielen und Virtual-Reality-Umgebungen läßt sich seit einiger Zeit ein wachsender Bedarf an realistischen Effekten, wie z.B. Wasser, Rauch, Feuer, Explosionen oder Kaustiken, beobachten. Da derartige Softwareumgebungen auf handelsüblichen PCs betrieben werden, müssen zunehmend Algorithmen zur interaktiven Simulation und Darstellung auf genau dieser Architektur entwickelt werden.

Ziel dieser Arbeit ist es, aktuelle Grafikkarten – sog. Graphics Processing Units (GPUs) – für diesen Zweck zu verwenden. Für den Einsatz von GPUs als numerische Koprozessoren gibt es mehrere Gründe. Zum Einen sind GPUs die erste kommerziell erfolgreiche Hardware einer zukünftigen Klasse von günstigen, leistungsstarken Supercomputern. Angetrieben durch die ständig steigenden Ansprüche der Spieleindustrie, haben sich diese GPUs inzwischen zu frei programmierbaren, hochgradig parallel arbeitenden Vektorprozessoren entwickelt. Zum anderen führt die Verwendung von GPUs zur numerischen Simulation dazu, dass die Simulationsergebnisse direkt im Grafikkartenspeicher verfügbar sind und ohne Umwege angezeigt werden können. Somit wird der zeitaufwändige CPU-GPU Datentransfer vermieden, der sonst zum Flaschenhals vieler Applikationen werden würde.

Um allgemeine numerische Techniken auf einer derart spezialisierten Hardware einfach implementieren zu können, muss eine GPU-Abstraktionsschicht zur Verfügung gestellt werden. Aus diesem Grund wurde im Rahmen dieser Arbeit zunächst eine Bibliothek für lineare Algebra auf GPUs entwickelt. Dieses Framework abstrahiert von den grundlegenden GPU-Datenstrukturen und Algorithmen und erlaubt dem Programmierer, sich auf die Entwicklung seiner Applikation zu konzentrieren, ohne sich mit der GPU-Programmierung auseinander setzen zu müssen. Aufbauend auf diesem Framework, bestehend aus elementaren Operationen der linearen Algebra, wurden im Verlauf der Arbeit komplexere Algorithmen entwickelt, z.B. effiziente Löser für große Systeme linearer Gleichungen. Diese Algorithmen erlauben partielle Differentialgleichungen wie z.B. die Wellengleichung oder die Navier-Stokes-Gleichungen auf der GPU numerisch zu lösen.

Das in dieser Arbeit vorgestellte Framework erleichtert die Verwendung numerischer Simulationstechniken zur Generierung echtzeitfähiger visueller Effekte. Durch die Verwendung der Simulationsergebnisse zur Steuerung geometrischer Primitive direkt auf der Grafikkarte und der Möglichkeit, diese Resultate direkt wieder auf dem Bildschirm anzuzeigen, kann eine Vielzahl verschiedener Effekte erzeugt werden. Beispielsweise wird durch die Kombination von 3D-Texturen mit Partikeln zum ersten Mal die Simulation und Darstellung dynamischer dreidimensionaler Effekte in Echtzeit überhaupt ermöglicht.

Zusätzlich zur reinen Simulation von Stömungsphänomenen werden in dieser Arbeit auch neue Techniken zur Darstellung optischer Effekte, welche durch die Anwesenheit des Fluids auftreten, präsentiert. Ein Beispiel für solche Effekte sind Kaustiken. Diese entstehen, wenn Licht von gewölbten Oberflächen reflektiert oder refraktiert und somit gebündelt oder gestreut wird. Da alle hier vorgestellten Techniken keinerlei Vorverarbeitung bedürfen, können sie ohne Einschränkungen für dynamische oder erst auf der Grafikkarte erzeugte Szenen eingesetzt werden.

Acknowledgements

First and foremost, I want to thank my advisor, Rüdiger Westermann. Although it was certainly not always easy, I think that I have evolved in the last four years, both as a scientist and as a person. Rüdiger, more than any other person, was instrumental in this evolution. It was mainly through interaction with Rüdiger that I shaped my understanding of my subject, of Computer Graphics at large, and of being a scientist. (Although, of course, any errors in the end result are my responsibility...) I feel very lucky to have had Rüdiger as an advisor, and look forward to more opportunities to learn from and work with him in the future.

I would also like to thank Mark Segal and Alpana Kaulgud of ATI and John Owens, Nick Triantos, and Mark Harris of NVIDIA not only for supporting the final steps of writing this thesis by providing me with vital historical information but also for the ongoing support with hardware and software as well as for the many answers to all of my questions.

Like all the students I know, I am very happy I have had the chance to study Computer Science at the Technische Universität München. Our department has a unique atmosphere of academic excellence combined with friendliness and openness that makes it a very special place to learn in. I greatly enjoyed the interaction with all of my fellow Ph.D. students and post docs, Thomas Schiwietz, Kai Bürger, Jens Schneider, Joachim Georgii, Matrin Kraus, and Polina Kondratieva. A big thanks for finding all those typos and other errors in this thesis. A particular thank you goes to my former diploma thesis students Thomas and Kai who tremendously supported this work. My thank you includes all people I wrote papers with, namely David Luebke, Naga Govindaraju, Aaron Lefohn, Tim Purcell, as well as the many fellow researches I "just" talked with.

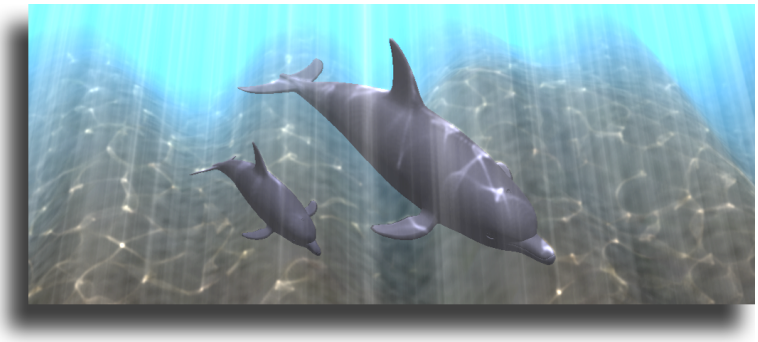
In particular I would like to say a big thank you to ATI for funding my research with the ATI fellowship award.

Finally, I would like to thank my family, my parents Harald and Birgitt, and in particular my beautiful wife Sabine. They made completing this project possible and worthwhile.

Contents

List of Figures

List of Tables



Chapter 1

My Thesis

This dissertation demonstrates how today's graphics processing units can be efficiently used for interactive simulation and rendering of fluid effects in computer games and virtual environments.

1.1 Introduction

In recent years there has been an increasing demand for realistic natural effects in virtual environments, computer games, "infotainment" and scientific applications. Many of these applications require real-time generation, interaction, and display of such effects, not only on supercomputers but on commodity PC hardware. On such commodity PCs most of the rendering workload—especially for 3D graphics—is performed on the graphics card or the graphics processing unit (GPU). To keep up with the ever-growing demands of computer games, the computational power of these GPUs has increased dramatically in the last couple of years, more than tripling Moore's Law. Current off-the-shelf GPUs outperform commodity CPUs including recent multi-core models by an order of magnitude. This alone lets them appear ideal platforms for numerical simulation. Even more important is the fact that the bus performance of today's PC systems can not keep up with the ever increasing amount of data being required, and the processing speed of both the CPU and the GPU. Therefore, an increasing demand exists to transfer not only the pure rendering workload to the GPU but also to perform the simulation on this fast and highly parallel coprocessor thus avoiding unnecessary bus transfer.

In this thesis a multi-layer framework is developed that allows for efficient simulation and rendering

of fluid and fluid-related effects on commodity PCs at interactive frame rates (see Figure 1.1). The foundation for the numerical simulation is formed by a linear algebra framework on the GPU. Similar to the BLAS/LAPACK partition more complex algorithms for solving very large sparse systems of linear equations are implemented on top of the basic framework. These algorithms in turn are used to solve ODEs and PDEs such as the shallow water or the Navier-Stokes equations. Since the simulation is directly executed on the graphics card it is ready for display and requires no time-consuming data conversion or bus transfer.

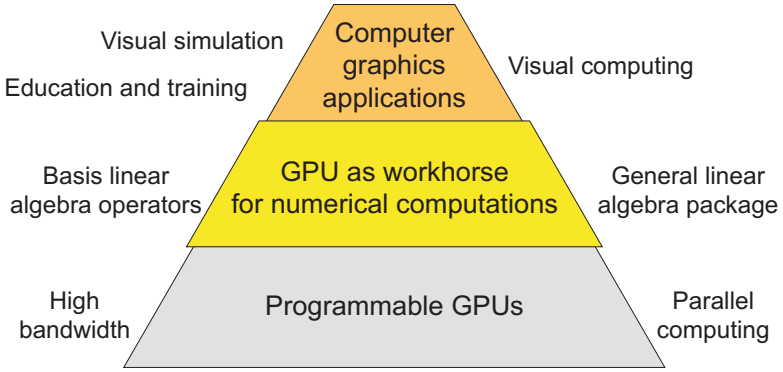


Figure 1.1: Using commodity GPUs with their enormous bandwidth and parallel computing capabilities as the foundation, a general linear algebra package is built. This package in turn, is used for computer graphics applications such as virtual environments, computer games, or education and training systems.

In addition to the actual simulation of fluid phenomena, this thesis also presents new ways to simulate and render effects that originate from the presence of the fluid. Such effects include the interaction of light with water surfaces such as caustics or light rays as well as the simulation and rendering of other matter being influenced by the simulation such as smoke, fire or other particle-based phenomena. By using the techniques described in this thesis, all of these effects can be simulated and rendered on today's off-the-shelf PC-hardware. By offloading the computation entirely to the GPU, the CPU is free to perform other tasks. In virtual environments these tasks can be tracking, collision-detection, etc. while in computer games the CPU can be more efficiently used to compute player AI, pathfinding, and other operations.

1.2 My Contribution

Before I started to write this dissertation I read a few others to find out how to do it best. I realized that many authors focus their theses strongly around their publications thus making clear where one of their papers ends and the next one begins. However, to make the dissertation more interesting for the interested reader I decided to write it in a different way, more like a book about the topic "Interactive Simulation and Rendering of Fluid Effects on GPUs" rather than a collection of my papers. That said, I will use the rest of this section to briefly summarize my papers that led to this thesis.

Figure 1.2 shows the chronological order and the connections among selected papers of mine. A very

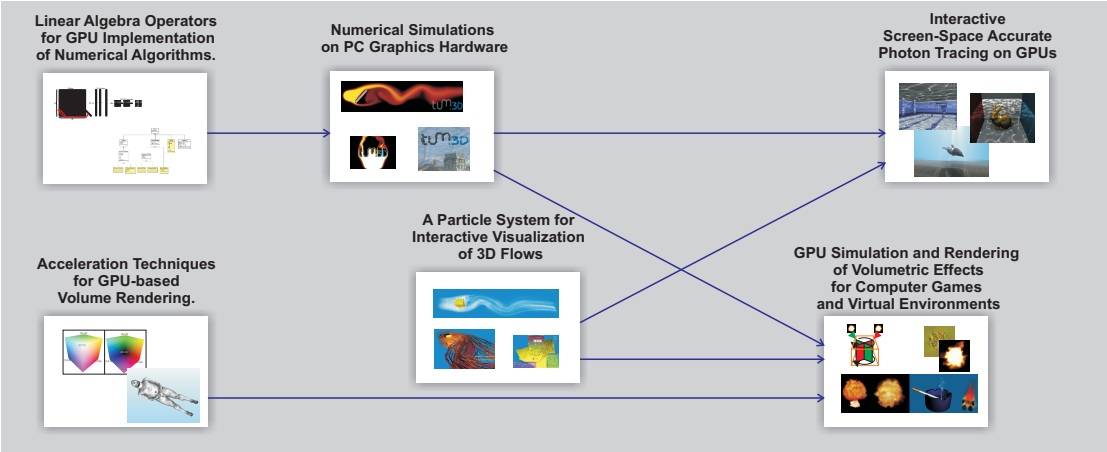


Figure 1.2: Important papers in my research and their dependencies.

important building block of my work was the paper "Linear Algebra Operators for GPU Implementation of Numerical Algorithms" [69] that lays the foundation of all of the numerical simulations presented later [64, 66, 71]. In the paper "GPU Simulation and Rendering of Volumetric Effects for Computer Games and Virtual Environments" [70] fast extensions to pseudo-3D simulation and a novel approach to control the fluid-based effects is presented. A second line of research consists of rendering-oriented papers, namely "Acceleration Techniques for GPU-based Volume Rendering" [68] and "A Particle System for Interactive Visualization of 3D Flows" [65] that form the basis of most of the rendering algorithms used in the later papers to display the simulation results. Finally, the paper "Interactive Screen-Space Accurate Photon Tracing on GPUs" [64] demonstrates how to integrate the effects into our target applications such as games and virtual environments.

1.2.1 My Thesis, Our Thesis?

Since it seems to me that there exists no "gold standard" whether to write a thesis in singular or plural, I decided to stick with the "academic plural" as used in most scientific publications. In my opinion this reflects best the way the results presented here were achieved. Certainly this work would not be as it is without the help of my colleagues, students, and last but least my supervisor Rüdiger Westermann.

1.3 How to Read this Thesis

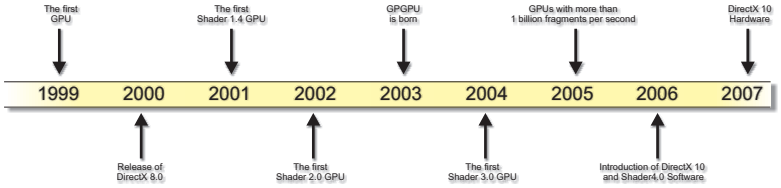
The structure of this thesis is similar to my research path over the last four years as it is depicted in Figure 1.2. In principle, the dissertation is designed to be sequentially read, from cover to cover especially by readers with no or only very little knowledge about GPU programming. The more experienced GPU programmer may want to skip the introduction and history sections and start directly with reading Chapters 3-6. Then again, someone familiar with the principles of volume visualization may want to

skip Chapter 5. In the final chapter we will recapitulate the presented work and take a look at the "big picture," the applications that use the proposed algorithms, and finally elaborate about future work. The "big picture" is presented at the end of this thesis rather than at the beginning since many of the concepts needed to understand this work are explained in the preceding text. Nonetheless the reader is welcome to peek ahead and start reading with Section 7.1.

1.3.1 Dependencies Between Chapters

The following information is of particular interest for readers who want to read only individual chapters.

The chapter dependencies are similar to the contributions in Figure 1.2. Since all of the chapters built on top of each other, there is always a certain dependency on the previous chapter. This is most obvious in Chapters 3 and 4, which should be read together. Chapter 5 describes how to display the results of the simulations from the previous chapters. Thus, it uses a lot of material presented before. The description of the integration of fluid effects into real-world scenes in Chapter 6 is designed to give enough information to be self-contained. The methods described in the final chapter depend on the entire previous text as it summarizes the whole thesis. However, to quickly answer the question "What has been done in this dissertation?" it is self-contained as well.



Chapter 2

"A Short History of GPUs"

The work presented in this thesis is motivated by the tremendous advances in GPU technology and performance over the last decade. Graphics hardware has quickly advanced from supercomputer-sized rendering servers and workstations to small and inexpensive PC add-on cards. In particular, the graphics chips on these PC boards have evolved from simple analogue/digital converters into highly parallel, almost arbitrarily programmable stream processors. Due to the rapid change in hardware capabilities there exists a close connection between the graphics APIs, the GPU features, and the work presented here. Therefore, this chapter gives a brief overview of the past, present, and near-future features and limitations of the software and the hardware used in GPU programming. Important related publications in the sense that they present applications of GPUs for purposes other than graphics applications* are referenced in this history recap. This allows for a chronological classification. For a thorough review of all the GPGPU work from the early days of GPU programming until recent advances the reader is referred to Owen et al.'s state of the art report on GPGPU [82].

2.1 Today's GPU Pipeline

Before we start with the GPU history we first make ourselves familiar with the basic structure of a GPU as it is used in today's PCs (see Figure 2.1). Technically a GPU is a parallel SIMD† stream processor. As can be seen in Figure 3.1, this processor takes a stream of vertex data and pushes it through a predefined

*These applications are often called GPGPU applications, with GPGPU standing for **G**eneral **P**urpose Computation on **GPU**s.
†SIMD = Single Instruction Multiple Data

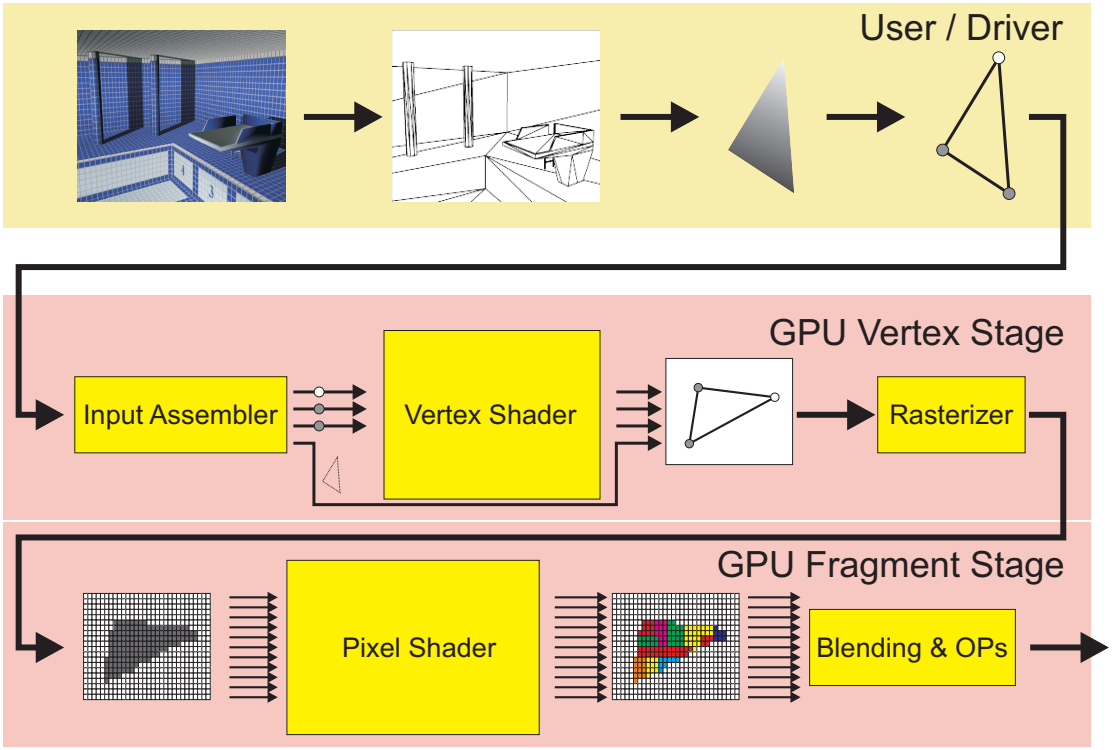


Figure 2.1: The image depicts the data-flow in a computer graphics application. First the scene is decomposed into single triangles, which are sent in a stream to the GPU. On the GPU, the input assembler schedules the triangles into the vertex shaders. These units transform each vertex and output the new coordinates and attributes. These values are passed to the rasterizer and reassembled with the information provided by the input assembler. The rasterizer computes the coverage of each primitive on the screen and outputs a new stream of covered fragments. Each of these fragments carries the linearly interpolated vertex values. This stream is passed to the pixel shaders. These units perform per-fragment operations such as texture-lookups. Next the fragment stream is handed to the blending and fragment operations stage. In this final stage fragment operations, such as the depth-test to resolve visibility, and blending are performed.

pipeline, until the output stream—a set of fragments—finally reaches either the screen or is stored in a memory array—a texture. The following list gives an overview of the stages in such a graphics pipeline:

- **Input Assembler**: This pipeline stage combines data from multiple input streams; e.g., normals, positions, texture coordinates, and schedules this data into the vertex shaders.

- **Vertex Shader**: This *programmable* stage takes all per-vertex information and usually performs transform and lighting; e.g., conversion from model-space into view-space, projection, and per-vertex Phong illumination.

- **Rasterizer**: The rasterizer converts the continuous triangle coordinates into discrete fragment information. It computes the screen coverage of every triangle and interpolates per-vertex attributes over it. For this purpose, the rasterizer takes the output of three vertex shader calls, combined with adjacency information from the input assembler.

- **Pixel Shader**: This *programmable* pipeline stage is invoked by the rasterizer for every fragment covered by a triangle. It takes as input the interpolated vertex values. All values that vary non-linearly over the triangle have to be computed in this stage. Such computations include texture mapping and Phong shading.

- **Blending and Fragment OPs**: The final stage of the pipeline computes how the color output of the pixel shader is combined with the values already present in the frame-buffer. Such operations include the depth-test, to resolve the visibility on a per-fragment basis, or alpha-blending if semitransparent objects are to be rendered.

2.2 APIs Overview

Since the introduction of standardized graphics APIs such as OpenGL [81] or DirectX [78] no real-world application that makes use of GPU features tries to communicate with the hardware directly anymore. All programs rely on a graphics API to abstract from the actual hardware implementation. This API layer in turn relies on the graphics driver to efficiently communicate with the GPU. Just as any graphics application we too will use APIs such as OpenGL or DirectX in this work. Although it requires some imagination to map linear algebra constructs such as vectors and matrices to textures and triangles, the use of a standard API is the most efficient and often the only possible way to use the horsepower of today's GPUs. Therefore, in the following section a short introduction to the most popular APIs is given.

2.2.1 OpenGL & Direct3D

Today, OpenGL and DirectX or more specifically the Direct3D subsystem of the DirectX API [‡] are the de-facto-standards that are used for GPU programming. OpenGL was initially specified by Silicon Graphics Incorporated in 1992 as the successor of their proprietary IrisGL library. The first version of Direct3D was released about tree years later by Microsoft. In an effort to create a single graphics API there even existed a joint project between many vendors in 1997 including Microsoft and SGI. This new API, called Fahrenheit, should also include scene-graph functionality. However, it was abandoned for "various reasons"[§] and thus OpenGL and DirectX still coexist, at least on Microsoft Windows systems. Today,

[‡] The term DirectX is used throughout to address Direct3D, since the graphics community—for no apparent reason—is used to call the graphics library by the name of the superordinate API name.

[§] [After the project was abandoned] The engineers involved at SGI held a beach party in celebration—complete with bonfires on which they burned piles of Fahrenheit documentation. [113]

the choice between these APIs rather depends on the experience and preferences of the programmer than on the APIs themselves. However, there exist some differences that a new user should consider before starting a project.

In general, both APIs map to the same hardware; thus, most of the time there exists a one to one mapping from DirectX code to OpenGL code and vice versa. One of the fundamental design differences lies in the way new functionality is added to these APIs. While OpenGL has the notion of “extensions” that allow hardware vendors to add arbitrary functionality to OpenGL, DirectX is under closed supervision of Microsoft. DirectX usually requires a feature set which is beyond the currently available hardware generation and only if the hardware reaches this level of functionality and grows beyond, an entirely new DirectX version is released. For the programmer, the OpenGL advantage is instant access to new and sometimes experimental hardware functionality but on the other hand it makes the code vendor-dependent and sometimes even unusable if an experimental extension is discontinued. The DirectX developers on the other hand can be certain that a well-defined set of GPUs will support their applications; however, they may have to rewrite parts of their programs every four to five years when a new DirectX version is released and access to the new features is required. Last but not least OpenGL has the advantage of being platform-independent over DirectX, which is only supported by Windows, the XBox and the new XBox 360.

2.3 Graphics Hardware and Software

2.3.1 Workstation Graphics

The basic principles of the graphics hardware that we are seeing in today’s PCs is derived from the huge workstation graphics systems such as SGI’s RealityEngine [1] and the InfiniteReality [79]. As early as 1993 SGI introduced the RealityEngine, which featured the SIMD graphics pipeline with parallel vertex and pixel engines. Although the limited possibilities in 1993 hindered a tight integration on a single chip making it hard to see the similarities to a GPU today, the general idea of the graphics pipeline has remained almost the same since then. This should not be too surprising considering that even today’s GPUs still use the same API, namely OpenGL, and still the same people that developed graphics hardware for SGI, design GPUs for today’s leading graphics companies.

2.3.2 PC Graphics

For commodity PCs the GPU history is even shorter as the chapter’s title already suggests. To recapitulate all relevant hardware developments, it is sufficient to start in the year 1999. From the many vendors in the mid-90s today practically only NVIDIA and ATI survived the rapid evolution and the hard competition amongst GPU vendors. Since the software and hardware progress in the GPU evolution is closely coupled, I will use both NVIDIA and ATI cards to depict the state of the art in GPU hardware and software. As can be seen in Figure 2.2, NVIDIA and ATI were always more or less on a par in terms of both, fill rate and vertex processing performance, as well as feature set.

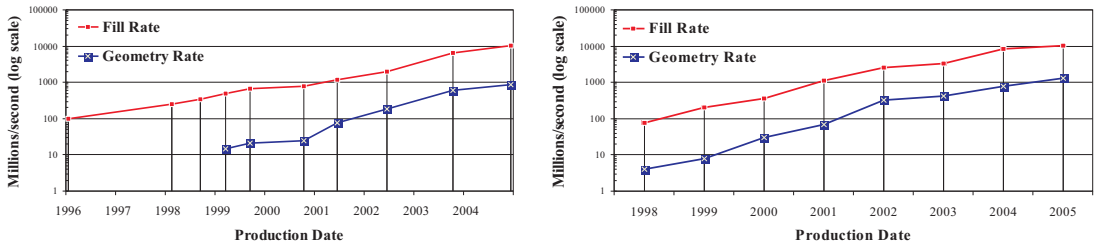


Figure 2.2: Pixel and Vertex performance of NVIDIA (left) and ATI GPUs (right).

	released	transistors	vertices/sec	fragments/sec
TNT2	1999	7 M	N/A	350 M
GeForce 256	1999	23 M	15 M	480 M
GeForce 2	2000	25 M	21 M	664 G
GeForce 3	2001	57 M	25 M	800 G
GeForce 4	2002	63 M	75 M	1.2 G
GeForce FX	2003	130 M	187 M	2.0 G
GeForce 6800	2004	222 M	600 M	6.4 G
GeForce 7800	2005	302 M	1.1 G	13.2 G

Table 2.1: NVIDIA's GPU History.

	released	transistors	vertices/sec	fragments/sec
Rage 128	1999	13 M	8 M	200 M
Radeon	2000	30 M	30 M	366 M
Radeon 8500	2001	60 M	69 M	1.1 G
Radeon 9700	2002	110 M	325 M	2.6 G
Radeon 9800 XT	2003	110 M	412 M	3.3 G
Radeon X800 XT PE	2004	160 M	780 M	8.32 G
Radeon 1900 XTX	2005	384 M	1.3 G	10.4 G

Table 2.2: ATI's GPU History.

2.3.3 Early PC Graphics Hardware

Our little GPU history starts in 1999 with the last PC graphics hardware generation without complete vertex processing capabilities, such as ATI's Rage 128 or NVIDIA's TNT2 (see Tables 2.2 and 2.1). These GPUs were capable of performing most of the fragment processing, such as texturing or blending but lacked the complete geometry transformation, and per-vertex lighting stage as specified by the OpenGL pipeline. This hardware generation was still too restricted and incomplete for serious GPGPU computation; thus, to our knowledge no GPGPU algorithms made use of PC hardware those days.

2.3.4 From Fixed-Function Hardware to Register Combiners

Later in the year 1999 the first consumer graphics card with vertex processing capabilities—called T&L (transform and lighting)—became available, the GeForce 256. This card not only implemented the entire

OpenGL pipeline in hardware but also featured another important novelty, the programmable *register combiner* (see Figure 2.3). These register combiners, although very restricted in functionality and extremely cumbersome to configure, began to change the idea of a fixed-function pipeline in favor of a new programmable stream architecture. Although earlier generations featured a combiner a stage as well, the GeForce 256 was NVIDIA's first card with a customizable combiner.

This was the first important step towards almost arbitrarily programmable GPUs as we see them today. With the GeForce 256 NVIDIA also introduced the term GPU:

> *"A GPU is a single-chip processor with integrated transform, lighting, triangle setup/clipping and rendering engines that is capable of producing a minimum of 10 million polygons per second."* [21]

which happened to mach exactly the GeForce 256 specifications.

Although exclusively designed for graphics-only applications, graphics processors of that time with non-programmable ("fixed-function") pipeline inspired scientists to use them for "non-graphics" applications. Possibly the first to do so were Lengyel et al. who used rasterization hardware for robot motion planning [73]. Hoff et al. described the use of z-buffer techniques for the computation of Voronoi diagrams [48] and extended their method to proximity detection [49]. Bohn et al. [12] used fixed-function graphics hardware in the computation of artificial neural networks. Convolution and wavelet transforms with the fixed-function pipeline were realized by Hopf and Ertl [50, 51]. Rumpf and Strzodka [91] analyzed the applicability of graphics hardware to nonlinear diffusion operations.

2.3.5 Shader 1.x

With Microsoft's DirectX API becoming more and more popular and version 7.0 supporting 3D graphics hardware for the first time, the vendors began to tune their hardware and drivers for DirectX's specifications. When Microsoft released version 8.0 with programmable vertex and pixel shaders in late 2000, hardware could be rated by which version of these shaders were supported. In the process of introducing updated versions of DirectX 8, Microsoft created a sequence of shader specifications. Most of these versions were designed in close collaboration with the GPU vendors to match a particular hardware such as the Pixel Shader 1.4 specifications being introduced for ATI's Radeon 8500 card. While every 1.x version became more powerful than its predecessor, still many seemingly odd restrictions remained, such as a certain order of calls had to be obeyed or variables were only allowed for certain purposes. These restrictions were imposed due to limitations of the underlying hardware, such as a lack of dependent fetch capabilities etc. While the Shader 1.x improvements were a big win for game effects, the limitations imposed massive restrictions for non-graphics applications on GPUs. In particular, the limitation to 8-bit fixed point precision in the pixel shader proved to be a serious barrier for GPGPU algorithms.

Regardless of these restrictions researchers became aware of the possibilities of commodity PC GPUs. Trendall and Steward gave a detailed summary of the types of computation available on these GPUs [105], Thompson et al. [104] used the vertex processor to solve the 3-satisfiability problem and to perform matrix multiplication. To overcome the 8-bit limitations in the much faster pixel shader, Strzodka showed

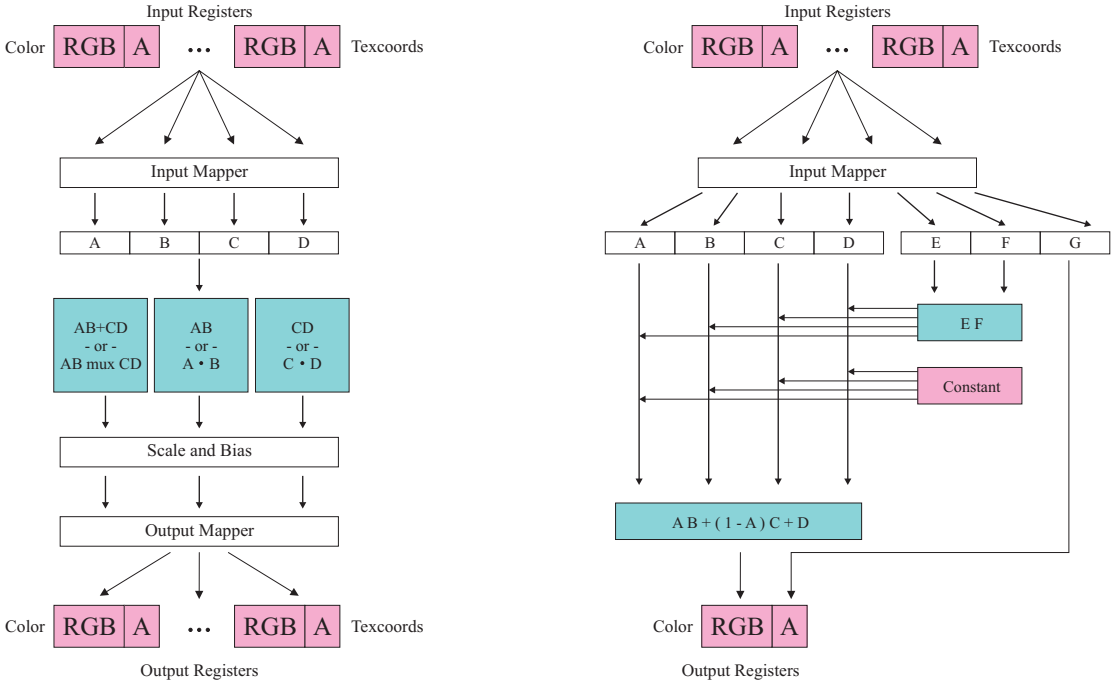


Figure 2.3: NVIDIA Register Combiners (simplified). The left image shows a general combiner while the right is a final combiner with restricted functionality. Depending on the GPU generation up to eight general combiners could be configured to work in a pipeline.

how to combine multiple 8-bit texture channels to create virtual 16-bit precise operations [101], and Harris analyzed the accumulated error in boiling simulation operations caused by the lack of precision [42].

While most this early work, due to the restrictions of the hardware, never made it into real-world applications, the ideas presented in these papers inspired many scientists to take a closer look at the upcoming Shader 2.0 graphics hardware. This new standard eliminated many of the drawbacks of the previous generation.

2.3.6 Shader 2.0 and Floating Point Precision

As mentioned before the next big step in GPU evolution was the introduction of the Shader 2.0 API and its implementation by the graphics card vendors in 2002 with ATI's Radeon 9700 and NVIDIA's GeForce FX. The hardware supporting this API had practically none of the semantic restrictions of the earlier shader versions left. In Shader 2.0 the only restrictions that were left were the length of the shader program and the number of texture fetches as well as texture dependencies. However, even these restrictions were soon removed with the vendor specific Shader versions 2.0a (NVIDIA) and 2.0b (ATI). Apart from the removal of the semantic restrictions in the vertex and pixel shader the vendors introduced floating point textures as well as floating point arithmetic in the vertex *and* pixel stage.

In the scope of this thesis and for GPGPU programming in general this was probably the most im-

portant improvement to previous hardware generations. The fact that Shader 2.0 functionality together with the availability of floating point textures boosted GPU programming can be seen by looking at the number of new publication that came out shortly after the release of the new hardware generation. To give just one example: SIGGRAPH 2003 alone, dedicated an entire session with papers from Hillesland et al. [47], Bolz et al. [13], and Krüger and Westermann [69] to this topic. In the months and years after, many researches published new ideas to program GPUs including entirely new languages to abstract even more from the hardware implementations such as Sh [76] and Brook [16]. While still most of the researchers prefer the better control and performance of native shader languages such as GLSL and HLSL these approaches reflect the general trend in GPU programming, to abstract from the hardware and to concentrate on higher-level algorithms. For a complete survey we refer again to Owen et al.’s state of the art report on GPGPU [82].

2.3.7 Shader 3.0

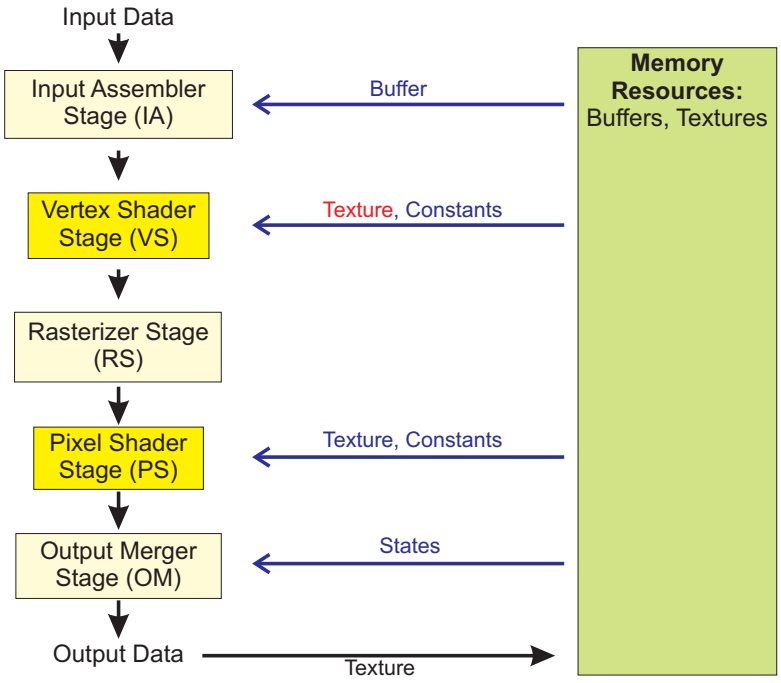


Figure 2.4: The Direct3D 9c Pipeline. Yellow boxes denote the two programmable stages. Note the red “texture” on the arrow to the vertex shader, this is one of the important additions to the pipeline by the Shader 3.0 API.

Although the Shader 3.0 feature set was specified together with Shader 2.0 as part of DirectX 9.0 it took until 2004 to built Shader 3.0 capable cards. Apart from practically removing the last remaining shader length restrictions and adding new inputs as well as “pseudo” dynamic branching, Shader 3.0 allowed for the first time to access texture maps in the vertex stage (see Figure 2.4). For this work the vertex texture fetch capability marks another dramatic change in GPU evolution since it allows dynamic

geometry modification on chip without the need to download and upload data to the CPU. In general, the availability of geometry generation and modification features on GPUs caused a massive speed up for many applications making them interactive for the first time. Particular examples of applications that depend on this functionality are GPU-based particle engines. With the geometry generation and modification features being available on current GPUs, many new applications were proposed. These systems made use of either the copy to vertex buffer [62, 61], the render to vertex buffer [58, 65] or the vertex texture fetch [70, 63] functionality to displace primitives.

2.3.8 Beyond Shader 3.0

The next shader release, Shader 4.0 is part of the upcoming DirectX 10 (DX10) [10]. DirectX 10 in turn will require an entirely new generation of graphics hardware. As can be seen in Figure 2.5 compared to Figure 2.4 new stages have been added to the pipeline that require a redesign of DirectX 9.0 class hardware.

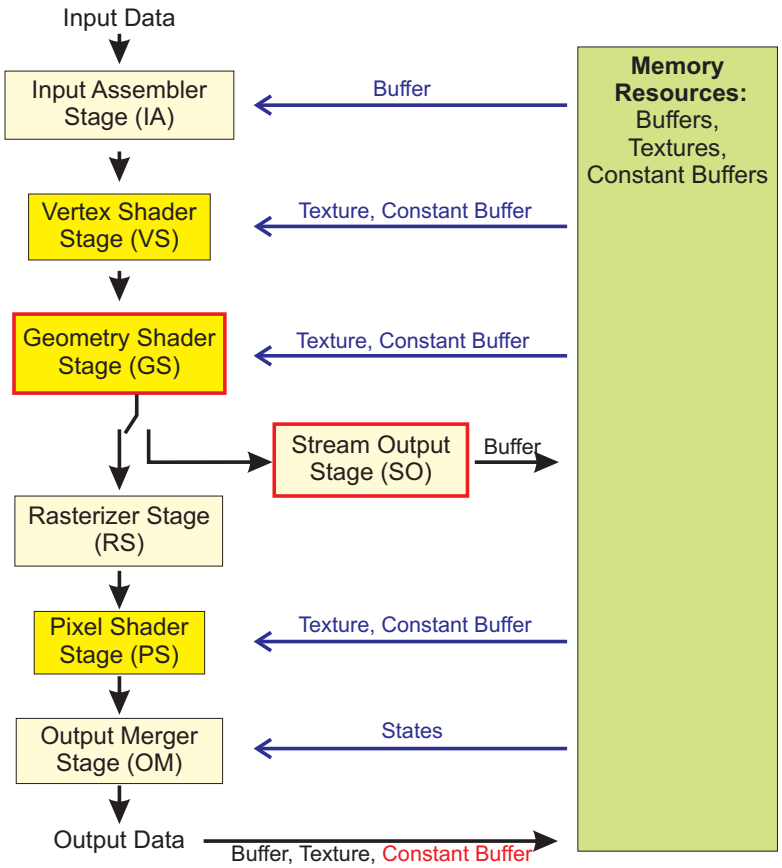
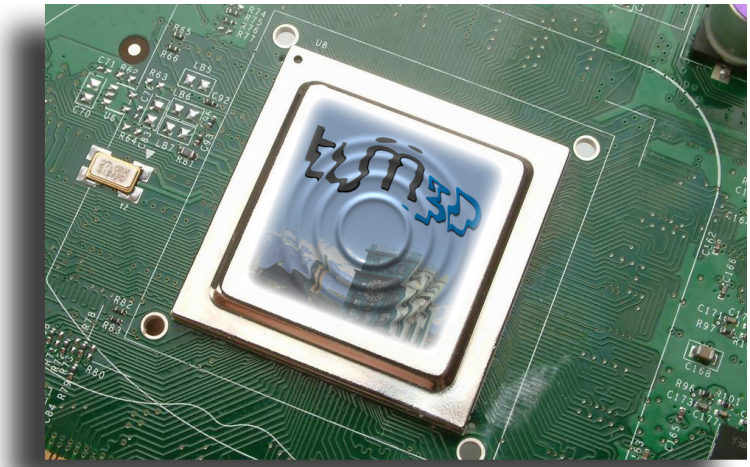


Figure 2.5: The Direct3D 10 Pipeline. Yellow boxes denote the three programmable stages. Important additions to the pipeline are marked in red.

The most obvious difference in the pipeline is a new geometry shader stage that allows the program-

mer to execute a shader on the primitive level, such as a triangle. Most interestingly the geometry shader allows for amplification and reduction of the vertex stream. As an example the geometry shader may take as input a single point and output hundreds of triangles. However, there is more to DX10 then just the geometry shader. New ideas include the unified shaders. Unified shader architecture does not necessarily mean that vertex, geometry, and pixel shaders are executed on the very same piece of silicon but it means there exist essentially no difference in their syntax and capabilities anymore. Therefore, shader length, texture indirections, intrinsic functions etc. are common to all programmable stages. Furthermore, Shader 4.0 requires integer support and bit-wise operations both of which will simplify many GPGPU programs. Apart from the differences in shader functionality DirectX 10 in general is optimized to relieve the CPU even more during rendering. In OpenGL and DirectX 9 the driver and the DirectX layer sometimes require a lot of CPU power even though the real computation is done on the GPU; this is to change with the entire design of DirectX 10.

Unfortunately, no DirectX 10 class hardware exists at present that supports this new functionality; thus, no meaningful speculations about the impact of the new features can be made.



Chapter 3

Linear Algebra on GPUs

One of the key challenges in many virtual environments and computer games is interactive simulation and rendering of fluid effects. In this dissertation this challenge has been addressed by leveraging the power of today's GPUs. For this purpose we first need to "teach" the GPU, which is intentionally built for the rendering of triangles, how perform basic linear algebra operations. Therefore, we have developed a GPU-based linear algebra framework perform more complex operations, such as solving systems of linear equations and to perform fluid simulation directly on the GPU.

Before the LA framework is described in detail, we give a short introduction to the basic concepts and algorithms of GPGPU programming. To keep this introduction brief, we will focus solely on the tools and algorithms required for the numerical simulation of fluids. For a more thorough discussion on other GPGPU projects the reader is referred to Owen et al.'s state of the art report on GPGPU [82].

3.1 GPGPU Stream Programming

In the previous chapter, GPUs were introduced as highly parallel stream processors, optimized for the rendering of triangles via OpenGL and DirectX. But from a different point of view the exact same hard-

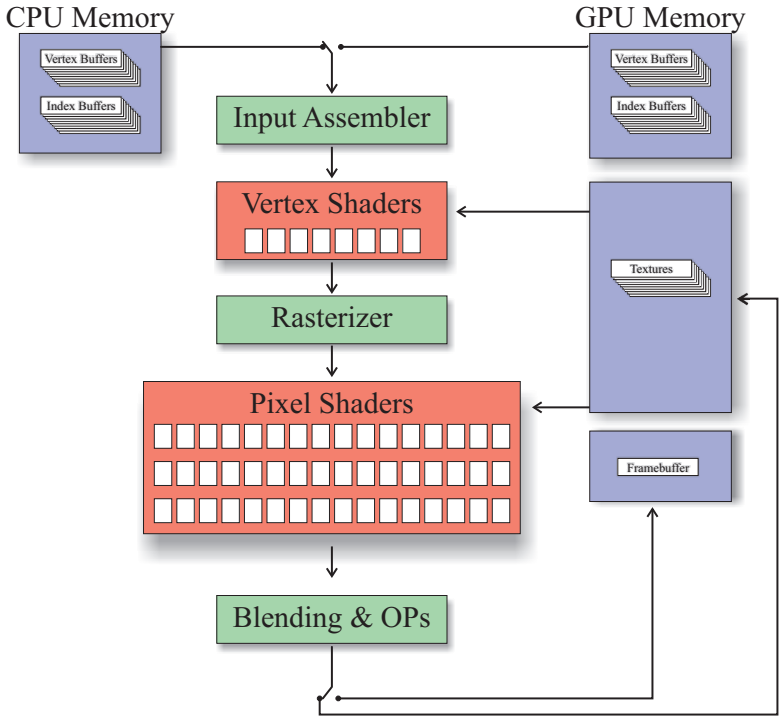


Figure 3.1: The figure shows the pipeline of a DirectX 9.0c GPU. Red boxes indicate programmable pipeline stages. Within these the white boxes show the number of pixel and vertex shader units for current GPUs, in this case an ATI Radeon X1900 [7].

ware can be described in a less graphics-specific manner. This view makes it much easier to understand how GPUs can be used for numerical computations such as linear algebra.

- **Input Assembler**: The input assembler (IA) is a stream muxer that can combine multiple input streams into a single output stream.

- **Vertex Shader**: The vertex shader is invoked for every element in the combined stream output by the IA. Since it is executed before the rasterizer is used to perform arithmetic operations on the vertex stream to control the pixel coverage of the geometry sent to the rasterizer.

- **Rasterizer**: In the rasterizer pipeline stage, the vertex stream is converted into a fragment stream that in turn is used for the GPGPU computation in the pixel shader. Typically only very few geometry is required, often just a single quad or triangle, to initiate the processing in the pixel shader.

- **Pixel Shader**: In terms of its capabilities the pixel shader is very similar to the vertex shader, it also takes one stream element at a time and outputs one element. However, current GPUs have

much more pixel shader units than vertex shaders making this pipeline stage much more efficient (see Figure 3.1). Additionally, these pixel shader units can access textures much more efficiently; therefore, GPGPU computations focus on the pixel shaders.

- **Blending and OPs**: The blending and OPs stage works as a stream filter that can be used to remove unnecessary elements from the output stream before it is stored in memory.
- **Render Targets**: Finally, the render target setup controls into which memory area the output stream is written.

Planing a strategy on how to use the GPU as a general purpose processor means to first decide how to represent data efficiently on the GPU and how to operate on it. Before we answer these two questions it is worth remembering that the hardware we are dealing with was built for computer graphics, or more precisely for computer games. These computer games usually render richly textured low-polygon models into a high-resolution viewport. Thus, the amount of triangles rendered is usually much smaller than the number of fragments generated. Keeping this in mind makes many design decisions much easier—even without knowing the hardware exactly.

Back to the considerations on data representation and processing. With the hardware optimizations towards computer games in mind the later consideration is straightforward to answer. As can be seen in Figure 3.1 the highest level of parallelism on current GPUs can be achieved in the pixel shader stage making this stage the premier choice for our computation*.

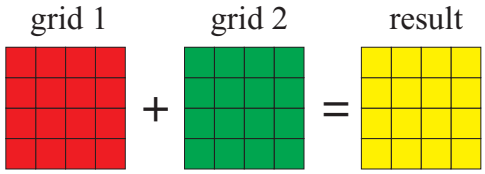
On the GPU large data is either stored as vertex or index buffers, or textures. However, access to vertex and index buffers is restricted to read-only in a fixed order†, as opposed to textures where dependent fetches; i.e., using the result of one texture fetch as coordinates for a second, allow for truly arbitrary access. Furthermore, the contents of textures can be changed by binding them as render targets. For these reasons textures become the canonical choice in most cases and almost all of the computations presented later will be executed in the pixel shader units, with the data residing in 2D texture maps. The following Section 3.1.1 will demonstrate a pipeline setup for such a pixel shader based computation. We consider the following example: the computation of a cell-wise sum of two large grids as shown in Figure 3.2.

3.1.1 The Power of a Single Quad

Although being a very simple example, this cell-wise sum operation requires the same setup as many other GPGPU algorithms; therefore, the understanding of this basic idea is crucial for the understanding of many GPGPU approaches.

*It is worth noting that future generations of GPUs are expected to have unified shaders (see Section 2.3.8) not only in syntax but also in terms of hardware. If this is the case, the vertex and pixel shader stages would be equally parallelized and it would not matter which stage to use.

†Although an index buffer can define an arbitrary order for reading out of the vertex buffer, yet this order is still fixed once the buffer is created and it can not be changed by the shaders.




```
for (int x = 0; x < N; x++) {
   for (int y = 0; y < N; y++) {
     result[x][y] = grid1[x][y] + grid2[x][y];
   }
}
```

Figure 3.2: Cell by cell summation of two grids as it could be implemented on the CPU in C++

Once again, remember that the GPU—with its parallel vertex and pixel shader units executing the same code on different stream elements—is a SIMD processor, which has to be configured in two ways. Firstly we set up its pipeline to be able to apply the computation kernel to every of the N elements in the target array, secondly we configure the kernel itself to execute an add operation.

As mentioned earlier we always store data in 2D textures, thus the two input grids *grid*1 and *grid*2 the output grid *result*—all of size $N \times N$—are stored in three textures of the same size. To execute the sum computation we first bind the target texture *result* as current render target. Then a stream of four vertices, representing a quadrilateral, is sent to the GPU. With the vertex shader configured to do nothing but simply pass the stream through, the four coordinates of the quadrilateral were chosen such that they cover the entire viewport. The viewport in turn is configured to the size of $N \times N$. With this setup the rasterizer generates N^2 fragments and thus invokes the pixel shader N^2 times—once for every fragment. The pixel shader is setup to fetch values from both textures, and output the sum (see Figure 3.3).

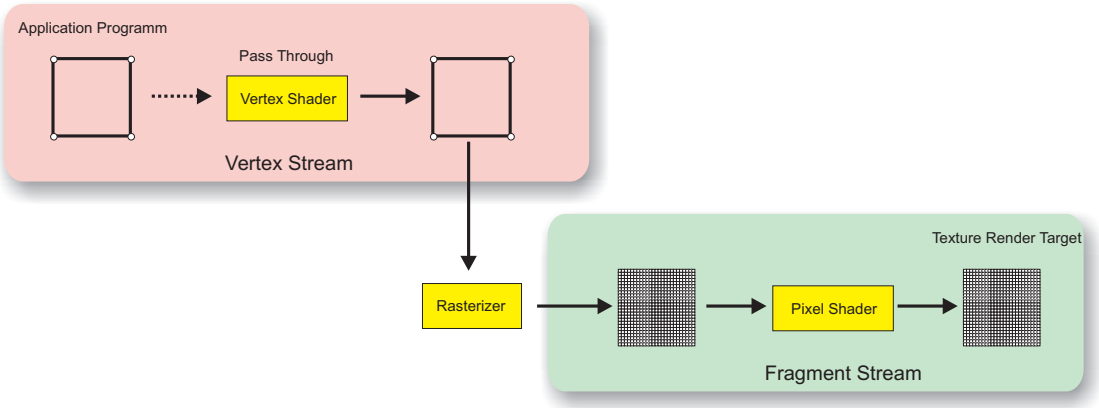
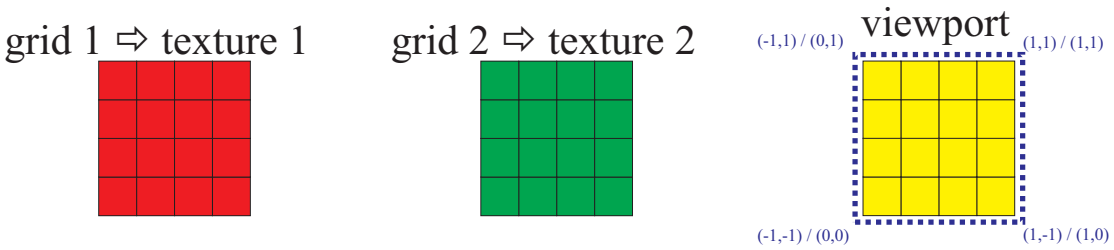


Figure 3.3: The basic setup of the rendering pipeline for a GPGPU computation. The computation is initiated by a single viewport-covering quadrilateral. This quadrilateral is converted by the rasterizer into a fragment stream. This stream in turn initiates the computation in the pixel shader.

Figure 3.4 depicts the texture setup as well as the vertex and pixel shader for the sum operation of two grids. Both shaders are written in DirectX's High Level Shading Language (HLSL). Even without knowledge about the exact syntax of this language the elegance and simplicity of GPU programming should become obvious. Essentially the GPU stream processing model simply removes the "for each element"-loop from Figure 3.2 by parallel SIMD execution, leaving only the loop's body in the pixel

shader.




```
void VertexShader( in float4  p : POSITION,  in float2  c : TEXCOORD0
                   out float4 pO : POSITION, out float2 cO : TEXCOORD0) {
  pO = p;
  cO = c;
}

void PixelShader(in  float2  coord  : TEXCOORD0,
                 out float4  result : COLOR0) {

  result = tex2D(texture1, coord) + tex2D(texture2, coord);

}
```

Figure 3.4: This Figure shows the setup of the rendering pipeline for the sum computation of two grids. The grids are stored in two textures, with a third texture bound to the viewport as render target. The dotted blue square around the viewport depicts the position and texture coordinates of the full-screen quad. Below the textures are the vertex and and pixel shaders in HLSL code.

With this pipeline setup any componentwise operation on streams is possible, including operations that use arbitrarily many input arrays. Since the pixel shader can fetch textures randomly, more complex kernels can be implemented using arbitrary kernels.

3.1.2 Reduce Operations

So far we have presented the basic approach that takes two input data sets—the grids—and computes a single output data set—the sum grid—of the same size. While this idea is already powerful enough to implement a wide variety of algorithms, we still lack the functionality to reduce a data set to a single element or only a few values. Imagine the following scenario: After a set of componentwise operations on a data grid, the algorithm needs to find the maximum value within the grid. This means that all the data in the grid needs to be reduced to only a single value, the maximum in this example.

To do this on the GPU, the reduce operation combines the grid entries in multiple rendering passes by recursively combining the result of the previous pass. Starting with the initial 2D texture containing the grid, and the quadrilateral lined up with the texture in screen space, in each step a quadrilateral covering only a quarter of the pixels is rendered. This can be achieved by either reducing the render target and keeping the full-screen quadrilateral or by reducing the size of the quadrilateral itself and keeping the render target. In our implementation we use the second alternative since it requires only two intermediate textures and thus has a smaller memory footprint when compared to storing the entire reduction pyramid.

Therefore, the application sets a shader constant such that in every reduction pass the vertex shader scales the quadrilateral down by a factor of 0.5. In the pixel shader program, each fragment that is covered by the shrunken quadrilateral combines the texel that is mapped to it and the three adjacent texels in positive (u,v) texture space direction. The distance between texels in texture space is issued as a constant in the shader program. The result is written into a new texture, which is now by a factor of two smaller in each dimension than the previous one. The entire process is illustrated in Figure 3.5. This technique is a standard approach to combine vector elements on parallel computer architectures, and is used in this scenario to keep the memory footprint for each fragment as low as possible. Thus, for a grid of size N, $\lceil \log(N) \rceil$ rendering passes have to be performed until the result value is obtained in one single pixel.

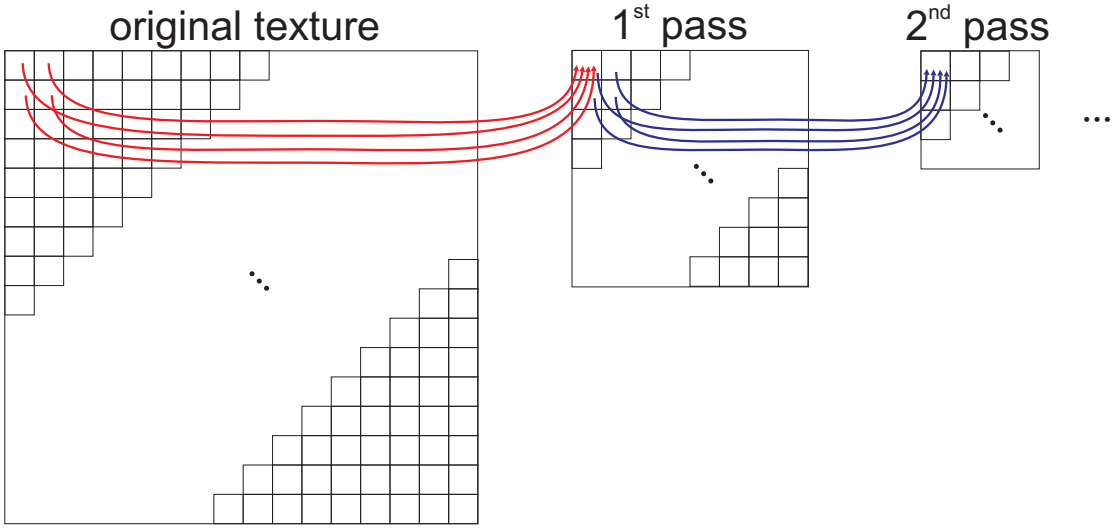


Figure 3.5: Schematic illustration of the reduce operation.

Equipped with the two basic principles, the componentwise operations and the reduce, we are ready to step from general considerations to the implementation of linear algebra on the GPU.

3.2 The Linear Algebra Framework

In contrast to previous approaches, which were specifically designed with regard to the solution of particular problems, our goal is to develop a generic framework that enables the implementation of general numerical techniques for the solution of differential equations on graphics hardware (see Figure 3.6). Therefore, we provide the basic building block routines that are used by standard numerical solvers. Built upon a flexible and efficient internal representation, these functional units perform arithmetic operations on vectors and matrices. In the same way as linear algebra libraries employ encapsulated basic vector and matrix operations, many techniques of numerical computing can be implemented by executing GPU implementations of these operations in a particular order. One of our goals is to replace software implementations of basic linear algebra operators as available in widespread linear algebra libraries; e.g.,

the BLAS (Basic Linear Algebra Subprogram) library [27, 28], by GPU implementations, thus enabling more general linear algebra packages to be implemented on top of these implementations; e.g., the LAPACK (Linear Algebra Package)[3].

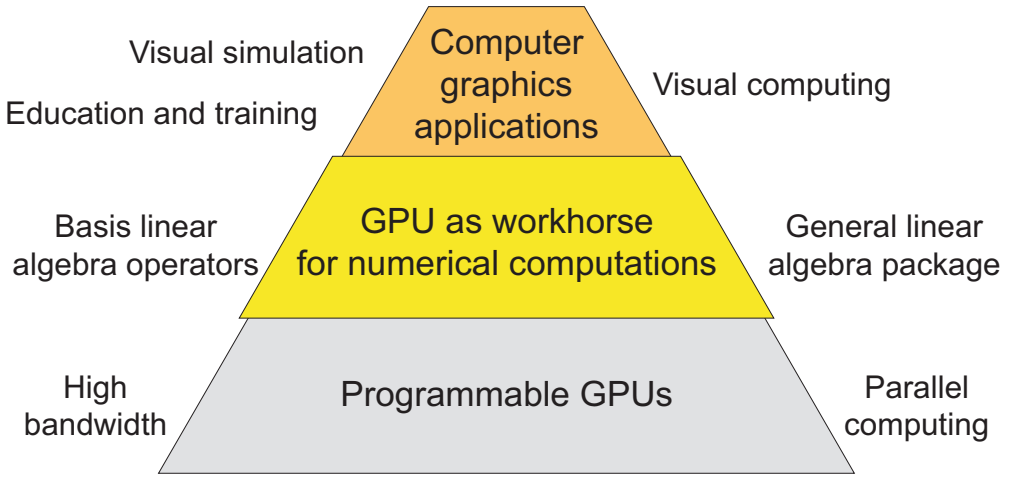


Figure 3.6: Using commodity GPUs with their enormous bandwidth and parallel computing capabilities as the foundation, a general linear algebra package is built. This package in turn, is used for computer graphics applications such as virtual environments, computer games, or education and training systems.

3.2.1 Data Structures

In this section we describe the internal representation of vectors and matrices on graphics hardware. The proposed representation enables the efficient computation of basic algebraic operations frequently used in numerical simulation techniques. The general idea is again to store matrices as texture maps and to exploit pixel shader programs to implement arithmetic operations. For the sake of simplicity only square matrices; i.e., vectors and square $N \times N$ matrices, are discussed. General matrices, however, can be organized in exactly the same way.

Vectors, Matrices, and Scalars

Most straight forwardly, vectors could be represented as 1D textures on the GPU. This kind of representation, however, has certain drawbacks: Firstly, limitations on the maximum size of 1D textures considerably reduce the number of vector elements that can be stored in one texture map. Secondly, rendering into a 1D texture is significantly slower than rendering into 2D textures with the same number of texture elements. On current graphics cards, the use of 2D textures yields a performance gain of about a factor of two. The reason is a more efficient use of the rasterizing and pixel shading hardware when rendering quads rather than lines. Thirdly, if a 1D vector contains the result of a numerical simulation on a computational grid, the data finally has to be rearranged in a 2D texture for rendering purposes. Fourthly, this representation prohibits the efficient computation of matrix-vector products. Although,

both the matrix and the vector can be rendered simultaneously and combined on a per-fragment basis by means of multi-textures (see Figure 3.7), the computed values are not in place and have to be summed along the rows of the matrix to obtain the result vector.

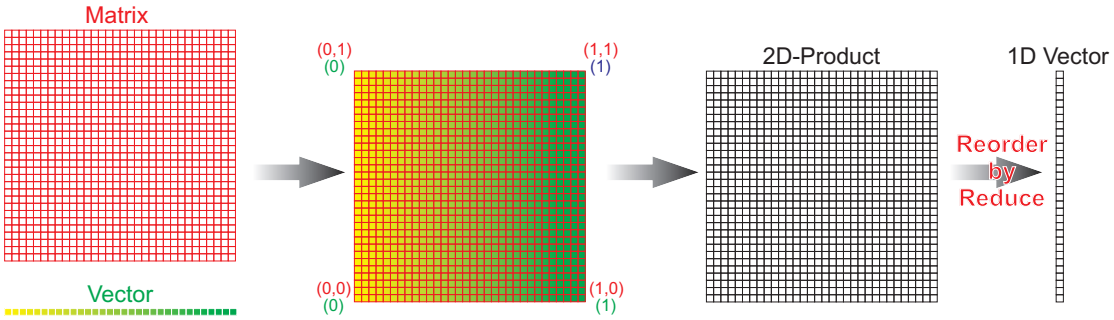


Figure 3.7: This illustration demonstrates how a matrix-vector product using 1D and 2D textures to represent vectors and matrices could be computed. The 1D texture is continued periodically across the rendered quadrilateral. This can be achieved by specifying two sets of texture coordinates as shown in red and blue in the center image. In the pixel shader the red coordinates are used to fetch from the matrix texture while the blue coordinates are used to fetch from the vector texture. This operation results in an intermediate 2D texture that needs to be reduced vertically in each line to get to the final 1D vector structure.

To circumvent the mentioned drawbacks, matrices are represented as a set of diagonal vectors and we store vectors in 2D texture maps (see Figure 3.8). To every diagonal starting at the i-th entry of the first row in the upper right part of the matrix, its counterpart starting at the $(N - i)$-th entry of the first column in the lower left part of the matrix is appended. In this way, no texture memory is wasted. Each vector is stored in a square 2D texture by the application program. Vectors are padded with zero entries if they do not entirely fill the texture. This representation has several advantages:

- Much larger vectors can be stored in one single texture object.
- Arithmetic operations on vectors perform significantly faster because square 2D textures are rendered.
- Vectors that represent data on a 2D grid can directly be rendered to visualize the data.
- Matrix-vector multiplication can be mapped efficiently to vector-vector multiplications.
- The result of a matrix-vector multiplication is already in place and does not need to be rearranged.

Most notably, however, the particular layout of matrices as a set of diagonal vectors allows for the efficient representation and processing of banded diagonal matrices, which typically occur in numerical simulation techniques. In a pre-process the application program inspects every diagonal and discards those diagonals that do not carry any information. If no counterpart exists for one part of a diagonal, it is filled with zero entries.

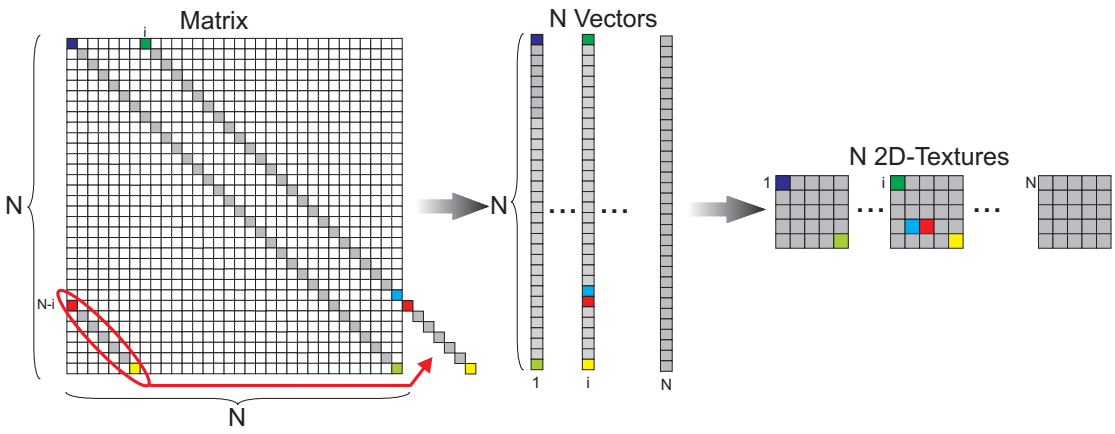


Figure 3.8: The representation of a 2D matrix as a set of diagonal vectors, and finally as a set of 2D textures is shown.

A nice feature of this representation is that the transpose of a matrix is generated by simply ordering the diagonals in a different way. Off-diagonals numbered i, which start at the i-th entry of the first row, now become off-diagonals numbered $N - i$. Entries located in the former upper right part of the matrix are swapped with those entries located in the lower left part. Swapping does not have to be performed explicitly, but it can be realized by shifting indices used to access these elements. Each index has to be shifted by the number of entries in the vector that come from the lower left part of the matrix. This can easily be accomplished in the pixel shader program, where the indexing during matrix-vector operations is performed (see Section 3.2.2).

After considering vectors and matrices we finalize the representations section with scalar values. Scalars are represented either as a program-specified shader constant or as a one-element vector. A shader constant is used when the scalar is specified by the application while the one-element vector is used when the scalar is the result of a previous computation on the GPU. With the current hardware being able to render solely into texture maps, we have to make this internal distinction. Nonetheless, note that the upcoming DirectX 10 adds functionality to render into shader constants—so called constant buffers. In that case only one scalar representation is required. To hide this internal and the possible change in behavior with DX10, scalar values are stored in a *clFloat* class that uses whatever GPU representation is needed and most efficient.

3.2.2 Operations

In the following the implementation of basic algebraic operations on vectors, matrices, and scalars based on the proposed internal representation is described. In each operation, rendering is directed to a specific render target that can be directly bound to a 2D texture. This texture in turn can be used in subsequent rendering passes to communicate intermediate results between operations.

Vector and matrix containers are defined as C++ classes *clVector* and *clMatrix*, respectively. Both containers assemble C++ arrays in that the array is decomposed into one or multiple vectors. Vectors

composed of zero entries neither have to be stored nor to be processed.

Upon initialization, for each vector a texture is created and bound to a texture handle. Internally, each class instance stores the resolution of the respective vector or matrix and of all the textures that are handled by that instance. Texture handles and the size of each texture can be accessed via public functions.

Vector Arithmetic

Arithmetic operations on two vectors can be realized in a simple pixel shader program following the ideas from the beginning of this chapter: The application issues both operands as multi-textures, which are accessed and combined in the shader program. On current graphics cards supporting at least the Pixel Shader 2.0 instruction set, arithmetic operations like arbitrary addition, subtraction and multiplication are available. The product of a scalar times a vector is realized by issuing the scalar as a constant value in the shader program or via a texture fetch into a 1×1 texture.

The method header for implementing standard arithmetic operations on vector elements looks like this:

```
void clVector:VectorOp(
  const CL_enum op,
  const clFloat& a, const clFloat& b,
  const clVector& x,
  clVector& res
);
```

The enumerator *op* can be one of **CL_ADD**, **CL_MULT** or **CL_SUB**. The scalars a and b are multiplied with the vector itself and x, respectively. At the beginning of each routine a consistency check is performed to verify that both vectors have the same internal representation. Then, the respective shader program is activated and the two vectors are issued as multi-textures. Finally, a square quadrilateral is rendered, which is lined up in screen space with the 2D textures used to represent the active vectors. The result is kept as a 2D texture to be used in consecutive computations. The fact that this operation can be expressed very efficiently in DirectX can be seen in the following example, where the **CL_ADD** branch in the **clVector:VectorOp** operator is shown. As can be seen it only requires a few lines of C++ code:

```
1  // setup rendering into target ‘‘res’’
2  res.BeginScene();
3    // setup input textures
4    ms_pShaderClVector->SetTexture("g_TexVector", this->m_pVectorTexture);
5    ms_pShaderClVector->SetTexture("g_TexVectorX", x->m_pVectorTexture);
6    // select operation
7    switch (op) {
8      case CL_ADD : ms_pShaderClVector->SetTechnique("techVectorAdd");
9                    break;
10     [...]
11   }
12
13   // setup scalar values, either as constants or as 1 × 1 textures
14   a.SetShaderValue("a", ms_pShaderClVector);
15   b.SetShaderValue("b", ms_pShaderClVector);
16
17   // send viewport covering quadrilateral
18   RenderViewPortCover(ms_pShaderClVector);
19 // unbind render target
20 res.EndScene();
```

On the GPU the following HLSL pixel shader is executed, for the sake of simplicity we only consider the case where both a and b are stored as shader constants by the `clVector` class[‡].

```
1  float4 psVectorAdd(float2 texCoord : TEXCOORD0) : COLOR0 {
2    return tex2D(g_SamplerVector,texCoord) * a +
3           tex2D(g_SamplerVectorX,texCoord) * b;
4  }
```

Matrix-Vector Product

In the following we will consider, the product of a matrix A times a vector x. A second vector y might be specified to allow for the computation of $Ax\ op\ y$. The constant op is one of **CL_ADD**, **CL_MULT**, **CL_SUB**. By allowing multiple operations to be combined in one function, semantic optimizations can be performed to reduce the number of render targets to be used. To compute $Ax\ op\ y$, the result of Ax is first written into the render target. Then, the result is bound as an additional texture, and in a final rendering pass it is combined with the vector y and rendered into the destination target.

[‡]The other cases are stored in different shader programs. HLSL allows shader programs to be split up into passes whereas the application program can select which passes are executed. The call **SetShaderValue** of the *clFloat* specifies the pass depending on whether the value is stored as a shader constant or as a 1×1 texture.

The header of the function performing matrix-vector operations looks like this (if one of A, x, or y is NULL, then only the respective component not equal to NULL is considered in the operation):

```
void clMatix:MatrixVectorOp(
  const CL_enum op,
  const clVector& x, const clVector& y,
  clVector& res
);
```

As matrices are represented as a set of diagonal vectors, matrix-vector multiplication becomes a multiplication of every diagonal with the vector. Therefore, for a matrix with N non-zero diagonals, N rendering passes are performed. In each of these passes one diagonal and the vector are issued as separate textures in the shader program. Finally the corresponding entries in both textures are multiplied. However, the j-th element of a diagonal that starts at the i-th entry of the first row of the matrix corresponds to the $((i + j) \mod N)$-th entry of the vector. First this index has to be computed in the shader program before it can be used to access the vector.

Values i and N are simply issued as constant values in the shader program. Index j, however, is directly derived from the fragment's texture coordinates and N. Finally, the destination index is calculated via $((i + j) \mod N)$ and converted to 2D texture coordinates.

After the first diagonal has been processed as described, the current render target is used as a texture and rendering is performed to a second intermediate render target.[§]. Thus, fragments in consecutive passes have access to the intermediate result, and they can update this result in each iteration.[¶] After rendering the last diagonal, the result vector is already in place, and the current render target can be used to internally represent this vector.

A considerable speed-up is achieved by specifying multiple diagonals as multi-textures, and by processing these diagonals at once in every pass. Parameters i and N only have to be issued once in the shader program. A particular fragment has the same index j in all diagonals, and as a matter of fact it only has to be computed once. Starting with the first destination index, this index is successively incremented by one for consecutive diagonals. The number of diagonals that can be processed simultaneously depends on the number of available texture units; i.e. sixteen on the employed GPU.

Note that with regard to the described implementation of a matrix-vector product, there is no particular need to organize matrices into sets of diagonal vectors. For instance, dense matrices might be represented as sets of column vectors, giving rise to even more efficient matrix-vector multiplication. In this case every column has to be multiplied with the respective vector element, resulting in a much smaller memory footprint.

[§]To avoid the re-creation of this intermediate target, for every operation, all framework classes use a common memory pool that is administered by the *clMemMan* class. The functionality of this class is further explained in Section 3.2.5

[¶]If 32 bit floating point blending should become available in an upcoming generation of GPUs the switching between intermediate render targets can be replaced by simply setting the corresponding blend equation.

Vector Reduce

Quite often it is necessary to combine all entries of one vector into one single element; e.g., computing a vector norm, finding the maximum/minimum element. Therefore, a special operation that outputs the result of such a reduce operation to the application program is provided:

```
clVector clVector::Reduce(
  const CL_enum cmb,
  const clVector& x
);
```

The enumerator *cmb* can be one of **CL_ADD**, **CL_MULT**, **CL_MAX**, **CL_MIN**, **CL_ABS**. If the second parameter *x* is not equal to NULL, the operation is carried out on the product of the current vector times *x* rather than only on the vector.

For the implementation of this operation we use the reduce operation described in Section 3.1.2, which combines the vector entries in multiple rendering passes by recursively combining the result of the previous pass. For a diagonal vector that is represented by a square texture with resolution N, $\lceil \log(N) \rceil$ rendering passes have to be performed until the result value is obtained in one single pixel. The respective pixel value is finally returned to the application program or stored in a 1×1 texture in the *clFloat*-container. If the value is required by the application program, the *clFloat* can be cast into a regular float, which triggers a read-back from the CPU. More often however, the scalar is needed on the GPU for further operations and remains in the 1×1 texture to avoid the bus transfer latency.

3.2.3 Matrix Update

Some numerical techniques require matrices to be modified during the computation. Single elements or entire diagonals[‖] have to be replaced, or two matrix diagonals have to be swapped. The latter operation can be performed simply by swapping the texture handles used to represent the two diagonals. To replace a diagonal, however, the set of 2D textures it is composed of has to be reloaded. If a diagonal should be replaced by another vector, the contents of this vector is rendered into the diagonal vector of the matrix. In exactly the same way the function that copies one vector into another one is implemented. Below is the function header for vector modification:

```
void clVector::CopyFrom(
  const clVector& source
);
```

Replacing a single element at position (i, j) of a matrix is realized by rendering a single point exactly at the position of the element in the respective diagonal. In this way, reloading the entire texture map

[‖]As described above, for full matrices a row based representation is more advantageous. In this representation the operations described here would become row swapping and row replacement.

can be avoided. Furthermore, with a function at hand that allows one to immediately replace particular values there is no need to store a duplicate of a matrix in main memory. The header for this function is specified here:

```
void clVector::ReplaceElementInplace(
  const int index, const clFloat& value
);
```

3.2.4 Sparse Matrices

The operations discussed so far execute very efficiently on current commodity graphics hardware. On the other hand, storing all entries of a matrix is not advantageous to process sparse matrices as they typically arise in numerical simulations. A matrix is called sparse if only a relatively small number of entries are non-zero. Obviously, storing a sparse matrix as a full matrix is quite inefficient both in terms of memory requirements and numerical operations. In order to effectively represent and process sparse $N \times N$ matrices, in which only $O(N)$ entries are supposed to be non-zero, alternative representations on the GPU need to be developed. In such cases the representation strongly depends on the patterns of non-zero entries. In our framework we consider two types of sparse matrices, the banded sparse and the random sparse matrices.

Banded Matrices

Due to the internal representation of matrices as a set of diagonal vectors we can effectively represent a banded matrix containing only a few non-zero diagonals. Zero diagonals are simply removed from the internal representation, and off-diagonals that do not have a counterpart on either side of the main diagonal are padded with zero entries. Thus, a matrix-vector product of a banded sparse matrix with N non-zero diagonals is realized by N vector-vector products. In contrast to banded matrices other sparse matrices exist, where non-zero entries are positioned randomly in the matrix, the diagonal layout of vectors does not allow for the exploitation of the sparseness in a straightforward way. To account for this kind of matrices, a particular representation has to been developed. This representation will be described in the following.

Random Sparse Matrices

To represent a random sparse matrix, we use a vertex-based approach instead of a texture-based approach. The principal layout results directly from the definition of a vector-matrix product $Ab = c$. For the sake of simplicity we consider only a single matrix entry $a_{i,j}$ in the matrix A at once. The point of influence of this entry $a_{i,j}$ with regard to the result-vector c is determined only by its row-index i. The column-index j of the entry $a_{i,j}$ determines which element in the vector b is to be multiplied with a. In our vertex-based approach we encode the row-index i in the vertex position and the column-index j in the texture coordinates of the vertex. The value $a_{i,j}$ itself is stored as the color component of the vertex.

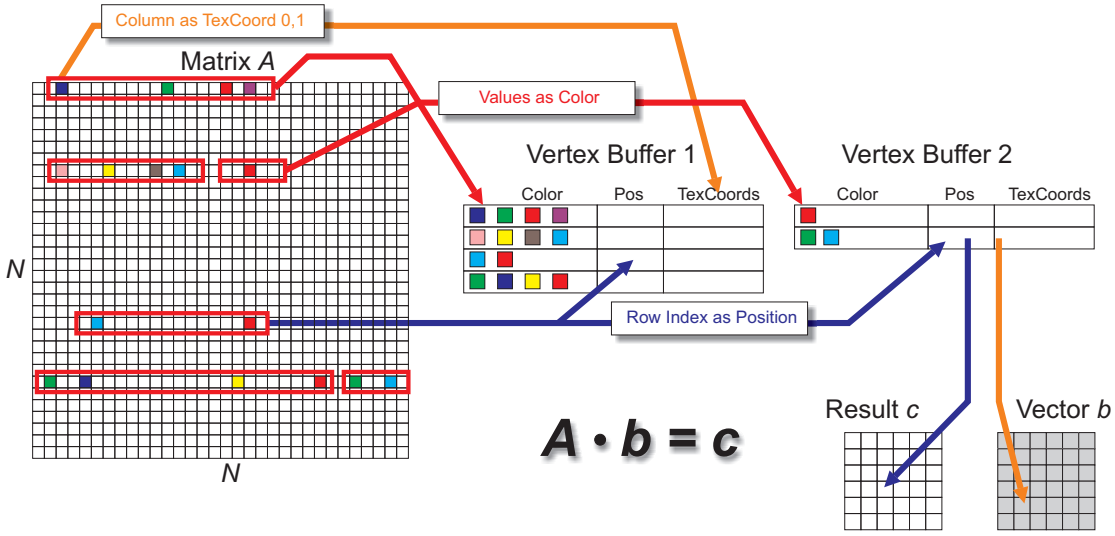


Figure 3.9: This image illustrates the representation of a vertex-based matrix and the computation matrix-vector product. The matrix vertex buffers are built such that every four entries in a row are stored in the color components of a vertex. The row index is stored as a 2D position and up to four column indices are stored in two 2D texture coordinates 0 and 1. To perform the matrix vector product the texture coordinates are used to fetch into the vector b and the position is used to instruct the rasterizer where to place the result in the target vector c.

For the multiplication of a matrix and a vector $Ab = c$ with the aforementioned matrix representation, we render the vertex that represents $a_{i,j}$ as a single pixel into the target vector c (see Figure 3.9). The position of this pixel is determined by the row-index i that was stored as vertex-position. In the pixel shader we use the texture coordinates of the vertex to fetch the respective values from the input vector b. This value is multiplied with the value of $a_{i,j}$, which is encoded as vertex color. Finally, the product is output to the target vector c.

It is also worth noting that, since we store the vectors in 2D textures, the coordinates i and j are two-dimensional. As with the previous representations we can combine four non-zero entries in a row into a single vertex for performance considerations. Finally, all of the vertices of a matrix are combined in a vertex buffer, which is stored in local GPU memory. As long as the matrix is not going to be modified, the internal representation does not have to be changed and a matrix-vector product can be executed by simply rendering a vertex buffer as a set of points.

It is a nice feature of the described scheme that the realization of matrix-vector operations on the GPU as it was proposed in Section 3.2.2 is not affected by the graphical primitives we use to internally represent and render matrices. The difference between sparse and full matrices just manifests itself in that we render the matrix entries as a set of vertices instead of a set of 2D textures. With this additional representation, a significant amount of texture memory, rasterization operations and texture fetch operations can be saved in techniques where random sparse matrices are involved.

3.2.5 Memory Management

In the following we will consider the memory management subsystem of the linear algebra framework. In particular the ability to pool and reuse texture resources efficiently is described.

As mentioned earlier, some of the framework operations require intermediate texture targets. One of these operations is the vector reduce operation, which renders sequence of quadrilaterals each with only a quarter of the size of its predecessor. In this operation two intermediate texture targets are needed, one for reading the four values and a new target to write the combined values into (see Section 3.1.2). After rendering one quadrilateral in the sequence these two textures are swapped and the process is repeated until the reduction is completed with rendering a quadrilateral of size 1×1. After the final value is either read back to the CPU or copied into a 1×1 texture on the GPU, the two texture targets are not needed anymore and could be released. However, this would require to entirely recreate the textures once the next reduce operation on this vector is performed. One way to avoid this recreation is to have every vector reserve the maximum number of temporary render targets it could possibly need at vector creation time and then reuse these textures every time a reduce operation on the vector is performed. This approach would require every vector to reserve about three times its size[**], which is too memory-intensive with the limited GPU resources in mind.

A much more flexible and less memory-intensive way of handling this issue is to introduce a memory manager that is shared by all framework classes. Whenever a vector, a matrix, or any other class needs a temporary texture or any other GPU resource, it requests the resource from the memory manager. When the resource is not needed anymore it is returned to the manager. The manager in turn does not necessarily destroy the returned texture but keeps it in a texture pool for later use. This allows the system to supply all GPU classes with temporary resources while keeping the memory overhead at a minimum. For an applications such as the Navier-Stokes simulation described later, the memory overhead for temporary textures is only about 1% of the program's overall memory consumption.

Another nice feature of this flexible memory management approach is that many of the GPU restrictions can be hidden from the user. Consider the following vector-vector summation $a := a + b$ with the data of the vectors stored in textures A and B. When implemented naively, one would bind the textures A and B as texture and also bind a's texture as render target. Binding the same texture as render target and as input texture is not allowed on the GPU and such an operation would lead to undefined results. By using the memory manager we simply acquire a temporary texture C from the global texture pool, compute $C := A + B$ and return the texture A back to the memory manager's pool. Finally texture C becomes the new data carrier of vector a. From the users point of view the initial operation $a := a + b$ was carried out without any loss in performance, since we can avoid any copy operation, while we still hide the GPUs restrictions.

The memory management class completes the set of basic LA classes. For an overview of the whole framework and the classes it provides see Figure 3.10.

[**] Every vector would have to hold a temporary texture for any componentwise operation, due to the GPU restriction that a render target can not be used for reading an writing at the same time. This texture, plus the two textures required for the reduce operation, sum up to about three times the space required to simply store the vector values.

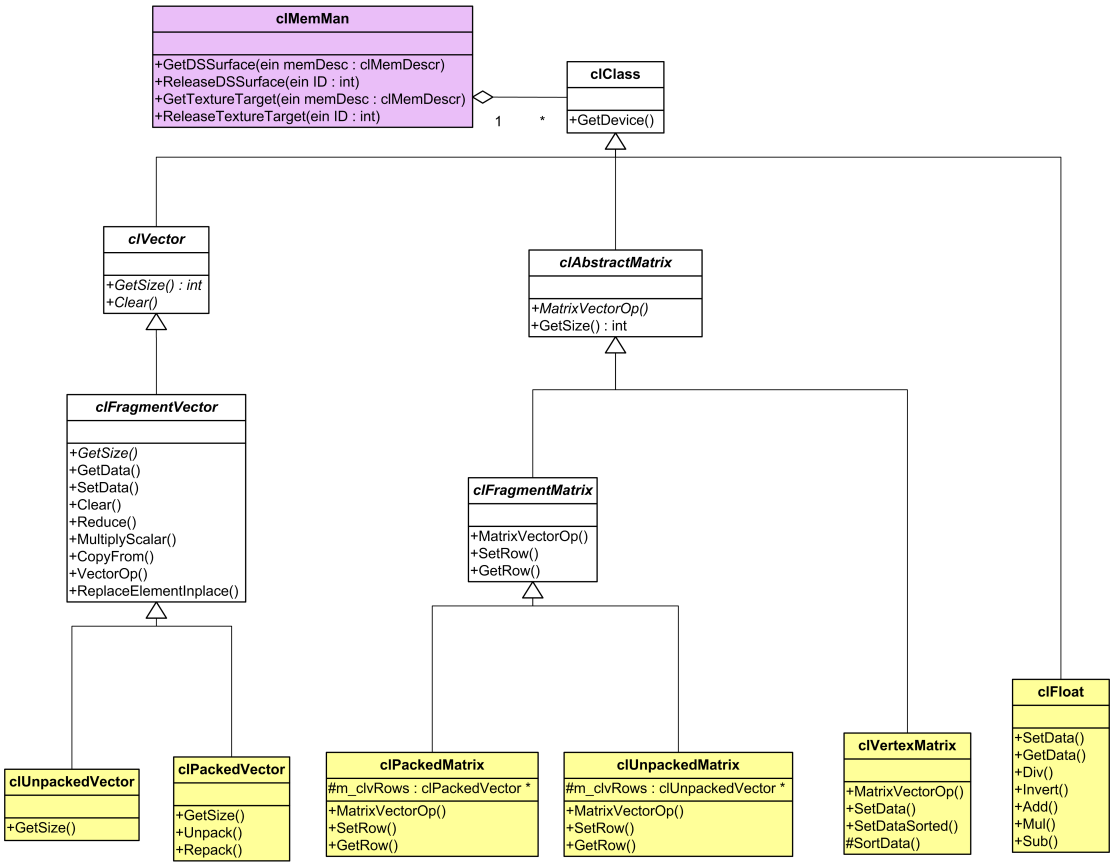


Figure 3.10: This figure shows a simplified UML diagram of the clFramework classes. The yellow classes represent the data types the user interferes with. Note the purple clMemMan, which takes care of the centralized memory management within the framework making sure that none of the precious GPU resources are wasted.

3.2.6 Performance Measures

To verify the efficiency of the proposed linear algebra framework for the implementation of numerical simulation techniques we have timed the performance of all the basic operations such as vector-vector and vector-matrix multiply as well as the reduce operation. All our experiments were run under Windows XP on an AMD Athlon 64 X2 4400+ processor [2] equipped with an NVIDIA Geforce 7900 GTX graphics card [22].

With regard to the realization of methods of numerical computing on graphics hardware, limited accuracy in both the internal texture stages and the shader stages is certainly the Achilles heel of any approach. In many cases, numerical methods have to be performed in double precision to allow for accurate and stable computations. Current generations of graphics cards, on the other hand, provide only single precision IEEE floating-point arithmetic in both the shader and texture stages. Other researchers have investigated methods to circumvent these limitations at least for a limited set of applications [35].

As our linear algebra framework is designed to be a building block for the interactive simulation of fluid effects in virtual environments and computer games, the limited precision is of lesser importance than interactivity and control over the effects.

Let us now investigate the performance of our approach as well as the differences to CPU implementations of some of the described basic operations. In our experiments the resolution of vectors and matrices was chosen such as to avoid paging between texture memory and main memory, and between main memory and disk. All our operations were run on vectors and matrices of size 512^2 to 2048^2. We have also not considered the constant time to initially load textures from main memory to texture memory. The reason is that fluid effects within virtual environments and computer games require data upload in form of vectors and matrices to the GPU only during the initial startup phase of the application. During the entire lifetime of the program the results of the simulations are visualized directly, or used on the GPU for further computation but never read back to the CPU. In these applications—on which this thesis focuses—the initial time required to setup the hardware is insignificant compared to the time required to constantly update the simulations. In other scenarios; e.g., if frequent updates of a matrix occur, this assumption may not be justifiable anymore. In this case, also the time needed to transfer data between different units has to be considered; however, none of our applications exhibit this property.

On vectors and full matrices the implementation of standard arithmetic operations i.e., vector-vector arithmetic and matrix-vector multiplication, was about 20-30 times faster compared to an optimized software implementation on the same multi-core target architecture (see Table 3.1). A considerable speed-up was achieved by internally storing vectors and matrices as RGBA textures. Sets of four consecutive entries from the same vector are stored in one RGBA texel. Thus, up to four times as many entries can be processed simultaneously. We should note here that operations on vectors and matrices built upon this particular internal format perform in exactly the same way as outlined. Just at the very end of the computation need the vector elements be stored in separate color components to be rearranged for rendering purposes. We can easily realize this task by means of a simple shader program, which fetches the result image for each pixel in the respective color component.

Vector - Vector Product	
Vector Size	Computation Time
512×512	0.025 ms
1024×1024	0.13 ms
2048×2048	0.56 ms

Vector - Matrix Product	
Vector Size	Computation Time
512×512	0.2 ms
1024×1024	1.2 ms
2048×2048	5.2 ms

Table 3.1: The left table gives timings for a vector-vector multiplication, while the right table shows timings for a multiplication of a vector with a banded sparse matrix, which was composed of ten non-zero diagonals.

Obviously, only one vector element can be stored in a single RGBA texel if numerical operations on vector-valued data, i.e., color, have to be performed. On the other hand, in this case also the performance of software implementations drops down due to enlarged memory footprints. Our current software implementation is highly optimized with regard to the exploitation of cache coherence on the CPU. In practical applications, a less effective internal representation might be used, so that we rather expect a

relative improvement of the GPU-based solution.

The least efficient operation compared to its software counterpart is the reduce operation. It is only about a factor of ten faster even though we store four elements in one RGBA texture (see Table 3.2). The relative loss in performance is due to the fact that the pixel shader program to be used for this kind of operation is a lot more complex than the one that is used for vector-vector multiplication. Furthermore, more passes are required to obtain the final result[††]. On the other hand, even a performance gain of a factor of ten seems to be worth an implementation on the GPU.

Vector Reduction	
Vector Size	Reduction Time
512×512	0.09 ms
1024×1024	0.28 ms
2048×2048	1.08 ms

Table 3.2: Timings for a vector-reduce operation with vectors of different sizes.

3.2.7 GPU Solvers for Systems of Linear Equations

We will now exemplify the implementation of general techniques of numerical computing by means of the proposed basis operations for matrix-vector and vector-vector arithmetic. Similar to the BLAS LAPACK partition we implement more complex algorithms for solving very large sparse systems of linear equations on top of the basic linear algebra framework.

Conjugate-Gradient Method

The conjugate-gradient (CG) method is an iterative matrix algorithm for solving large, sparse linear system of equations $Ax = b$, where $A \in R^{n \times n}$. Such systems arise in many practical applications, such as mechanical engineering or in computational fluid dynamics, which is of particular interest for this thesis. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals r corresponding to the iterates, and search directions used in updating the iterates and residuals. The CG algorithm remedies the shortcomings of steepest descent by forcing the search directions $p^{(i)}$ to be A-conjugate, which is ${p^{(i)}}^T A p^{(j)} = 0$, and the residuals to be orthogonal. Particularly in numerical simulation techniques, where large but sparse finite difference equations have to be solved, the CG-algorithm is a powerful and widely used technique.

In the following, pseudo code for the unpreconditioned version of the CG algorithm is given. Lower and upper subscripts indicate the values of scalar and vector variables, respectively, in the specified iteration. For an introduction to the CG method as well as to other solvers for linear system equations we refer the reader to Press et al. [85].

[††]It is worth noting that it is possible to combine more than four values per pass, thus reducing the number of passes until the reduction result is achieved. Our tests have shown that the optimum number of combining fetches per pass is strongly GPU dependent. For maximum performance it would be possible to benchmark the GPU during runtime and select the optimal reduction strategy. However, we did not consider this optimization in our current implementation since the time for the reduction operation is negligible compared to the entire simulation process as presented later.

1 $p^{(0)} = r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2 **for** $i \leftarrow 0$ **to** *#itr*
3 $\rho_i = r^{(i)^T} r^{(i)}$
4 $q^{(i)} = Ap^{(i)}$
5 $\alpha_i = \rho_i / p^{(i)^T} q^{(i)}$
6 $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
7 $r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$
8 $\beta_{i+1} = r^{(i+1)^T} r^{(i+1)} / \rho_i$
9 $p^{(i+1)} = r^{(i+1)} + \beta_{i+1} p^{(i)}$
10 convergence check

The CG method can effectively be stated in terms of the described building blocks for GPU implementation of numerical techniques[‡‡]:

1 A.**MatrixVectorOp**(CL_SUB, $x^{(0)}, b, r^{(0)}$) initial guess $x^{(0)}$
2 $r^{(0)}$.**VectorOp**(CL_ADD, $-1, 0, NULL, r^{(0)}$)
3 $r^{(0)}$.**VectorOp**(CL_ADD, $1, 0, NULL, p^{(0)}$)
4 **for** $i \leftarrow 0$ **to** *#itr*
5 $\rho_i = r^{(i)}$.**Reduce**(CL_ADD, $r^{(i)}$)
6 A.**MatrixVectorOp**(CL_ADD, $p^{(i)}, NULL, q^{(i)}$)
7 $\alpha_i = p^{(i)}$.**Reduce**(CL_ADD, $q^{(i)}$)
8 $\alpha_i = \rho_i$ / α_i
9 $x^{(i)}$.**VectorOp**(CL_ADD, $1, \alpha_i, p^{(i)}, x^{(i+1)}$)
10 $r^{(i)}$.**VectorOp**(CL_SUB, $1, \alpha_i, q^{(i)}, r^{(i+1)}$)
11 $\beta_i = r^{(i+1)}$.**Reduce**(CL_ADD, $r^{(i+1)}$)
12 $\beta_i = \beta_i$ / ρ_i
13 $r^{(i+1)}$.**VectorOp**(CL_ADD, $1, \beta_i, p^{(i)}, p^{(i+1)}$)
14 convergence check

In the GPU implementation, the application program only needs to read back a single pixel value from the GPU thus minimizing bus transfer. All necessary numerical computations can be directly performed on the GPU. Moreover, the final result is already in place and can be rendered as a 2D texture map.

Gauss-Seidel Solver

As a final example in this chapter, let us describe the GPU implementation of a Gauss-Seidel solver. For this example we used a column based matrix representation. Denoting with L and U the strict lower

[‡‡]Note that using a preconditioner matrix, for instance the diagonal part of A stored in the first diagonal vector in our internal representation, only involves solving one more linear system in each iteration.

and upper triangular sub-matrices, and with D the main diagonal of the matrix A, we can rewrite A as $L + D + U$. In one iteration, the Gauss-Seidel method essentially solves for the following matrix-vector equation:

$$x^{(i)} = Lx^{(i)} + (D + U)x^{(i-1)}$$

where $x^{(k)}$ is the solution vector at the k-th iteration.

$r^{(i)} = (D + U)x^{(i-1)}$ can be derived from the previous time step by one matrix-vector product. To compute $Lx^{(i)}$, however, updates of $x^{(i)}$ have to be done in place. Based on the representation of matrices as a set of column vectors, we sweep through the matrix in a column-wise order, using the result vector $x^{(i)}$ as the current render target as well as a currently bound texture. Initially, the content of $r^{(i)}$ is copied into $x^{(i)}$. When the j-th column of L is rendered, each element is multiplied with the j-th element in $x^{(i)}$, and the result is added to $x^{(i)}$. We thus always multiply every column with the most recently updated value of $x^{(i)}$.

Chapter 4

Simulation

In the previous chapter, the basic techniques to map linear algebra operations to the GPU were introduced. In this chapter it is demonstrated how these building blocks are used for the physics-based simulation of a variety of fluid effects.

The evolution of a variety of fluid effects can be expressed by the means of Partial Differential Equations (PDEs), which in turn can in most practical cases only be solved numerically. This chapter explains in detail how fluid differential equations are solved numerically on the GPU. This is done with the support of the previously introduced linear algebra framework. It is worth noting that the proposed linear algebra framework is not limited to the application of fluid simulation. On the contrary, it has been successfully applied to other simulations as well, such as the simulation of deformable bodies or image registration.

4.1 2D Wave Equation

We will first consider the numerical solution to the 2D wave equation. We describe the numerical solution on the GPU in much detail, from the numerical ideas down to the C++ and HLSL source code to get the reader more familiar with the idea of solving fluid PDEs on the GPU and with the linear algebra framework itself. For all further PDEs we will focus less on the code itself but describe the reasons and challenges of the GPU realization in more detail.

From mechanics we know that the temporal evolution of a homogeneous 2D membrane can be de-

scribed by Equation 4.1*. We use this membrane model as an approximation of a hydrostatic homogeneous, incompressible flow on the surface of a plane.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{4.1}$$

In this equation u represents the height of the surface over time t for every point x, y on the plane. The parameter c denotes the wave speed. To solve this PDE numerically we first discretize the water surface with a regular lattice and the time in regular intervals. On this regular lattice of interval spacing h we replace the partial derivatives by the finite differences

$$F'(x) = \frac{F(x+h) - F(x)}{h} + O(h) \tag{4.2}$$

$$F'(x) = \frac{F(x) - F(x-h)}{h} + O(h) \tag{4.3}$$

$$F'(x) = \frac{F(x+h) - F(x-h)}{2h} + O(h^2) \tag{4.4}$$

$$F''(x) = \frac{F(x+h) - 2F(x) + F(x-h)}{h^2} + O(h^2) \tag{4.5}$$

which can be derived from the Taylor expansion of F at $x + h$ and $x - h$. Substituting the second order derivatives in the wave Equation 4.1 we get the finite difference equations for each grid point (i,j):

$$\frac{u_{i,j}^{t+1} - 2u_{i,j}^{t} + u_{i,j}^{t-1}}{\Delta t^2} = c^2 \cdot \left(\frac{u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t}}{h^2} \right) \tag{4.6}$$

Solving for the unknown $u_{i,j}^{t+1}$ we get:

$$u_{i,j}^{t+1} = \alpha \cdot \left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} \right) + (2 - 4\alpha) \cdot u_{i,j}^{t} - u_{i,j}^{t-1} \quad \text{with } \alpha = \frac{\Delta t^2 \cdot c^2}{h^2} \tag{4.7}$$

This explicit solution can be rewritten as two basic LA operations. A matrix-vector product, involving a sparse matrix with five non-zero diagonals and a vector-vector subtraction. Equation 4.8 depicts these operations for a grid of size 3×3.

*For a derivation we refer the interested reader to Joel Feldman's notes on the Wave Equation [33].

$$\begin{pmatrix}
2-4\alpha & \alpha & & \alpha & & & & & \\
\alpha & 2-4\alpha & \alpha & & \alpha & & & & \\
 & \alpha & 2-4\alpha & & & \alpha & & & \\
\alpha & & & 2-4\alpha & \alpha & & \alpha & & \\
 & \alpha & & \alpha & 2-4\alpha & \alpha & & \alpha & \\
 & & \alpha & & \alpha & 2-4 & \alpha & & \alpha \\
 & & & \alpha & & & 2-4\alpha & \alpha & \\
 & & & & \alpha & & \alpha & 2-4\alpha & \alpha \\
 & & & & & \alpha & & \alpha & 2-4\alpha
\end{pmatrix} \cdot \begin{pmatrix} u^t_{0,0} \\ u^t_{1,0} \\ u^t_{2,0} \\ u^t_{0,1} \\ u^t_{1,1} \\ u^t_{2,1} \\ u^t_{0,2} \\ u^t_{1,2} \\ u^t_{2,2} \end{pmatrix} - \begin{pmatrix} u^{t-1}_{0,0} \\ u^{t-1}_{1,0} \\ u^{t-1}_{2,0} \\ u^{t-1}_{0,1} \\ u^{t-1}_{1,1} \\ u^{t-1}_{2,1} \\ u^{t-1}_{0,2} \\ u^{t-1}_{1,2} \\ u^{t-1}_{2,2} \end{pmatrix} = \begin{pmatrix} u^{t+1}_{0,0} \\ u^{t+1}_{1,0} \\ u^{t+1}_{2,0} \\ u^{t+1}_{0,1} \\ u^{t+1}_{1,1} \\ u^{t+1}_{2,1} \\ u^{t+1}_{0,2} \\ u^{t+1}_{1,2} \\ u^{t+1}_{2,2} \end{pmatrix} \tag{4.8}$$

By using the GPU linear algebra framework these operations can be carried out with a single call to the `m->MatrixVectorOp(CL_SUB,currentSurface,lastSurface,nextSurface)` method of the matrix class. In this call `currentSurface` and `lastSurface` are GPU vectors containing the grid values at time-step t and time-step $t-1$. The result is written into the vector `nextSurface`. To initialize the matrix `m` with its five non-zero-diagonals C++ code in Listing 4.1 is used.

```
1  int i;
2  float alpha = (dT*dT*C*C) / (h*h), mainDiag = 2-4*alpha;
3  float *data = new float[X*Y];
4  // setup main diagonal-Y
5  for (i=0;i<X*Y;i++) data[i] = (i>=Y) ? alpha :  0;
6  m->getDiag(X*(Y-1))->setData(data);
7  // setup main diagonal-1
8  for (i=0;i<X*Y;i++) data[i] = (i%X) ? alpha :  0;
9  m->getDiag(X*Y-1)->setData(data);
10 // setup main diagonal
11 for (i=0;i<X*Y;i++) data[i] = mainDiag;
12 m->getDiag(X*Y)->setData(data);
13 // setup main diagonal+1
14 for (i=0;i<X*Y;i++) data[i] = ((i+1)%X) ? alpha :  0;
15 m->getDiag(X*Y+1)->setData(data);
16 // setup main diagonal+Y
17 for (i=0;i<X*(Y-1);i++) data[i] = (i<X*(Y-1)) ?  alpha :  0;
18 m->getDiag(X*(Y+1))->setData(data);
```

Figure 4.1: C++ code for initialization of the matrix in Equation 4.8. The code contains five blocks that fill the five non-zero diagonals of the matrix.

Thinking about this operation in a more grid-centric way, the explicit solution described so far is nothing more than a filter kernel. This kernel takes two grids as input and writes the result into a new grid. At this point one may wonder: “Why not simply write a custom shader to compute the water

surface?" In fact, a customized shader program for this explicit solution is possibly slightly faster than the solution using the linear algebra framework. On the other hand, the advantage of the linear algebra framework approach lies in its flexibility and extendibility. This can be best demonstrated by using an implicit solution to the wave equation. Applying the Crank-Nicholson scheme, which specifies $u^t_{i,j} := 0.5 \cdot \left(u^{t+1}_{i,j} + u^t_{i,j}\right)$ to Equation 4.7 results in Equation 4.9. Now, the difference equation contains unknowns on both sides of the equality and the system of algebraic equations has to be solved simultaneously.

$$\begin{gathered}(4\alpha+1)\cdot u^{t+1}_{ij} - \alpha\cdot\left(u^{t+1}_{i+1\,j} + u^{t+1}_{i-1\,j} + u^{t+1}_{ij+1} + u^{t+1}_{ij-1}\right) = c^t_{ij} \\ \textbf{with} \\ c^t_{ij} = \alpha\cdot\left(u^t_{i+1\,j} + u^t_{i-1\,j} + u^t_{ij+1} + u^t_{ij-1}\right) + (2-4\alpha)\cdot u^t_{ij} - u^{t-1}_{ij} \textbf{ and } \alpha = \tfrac{\Delta t^2\cdot c^2}{h^2}\end{gathered} \tag{4.9}$$

This set of algebraic equations can be rewritten as a system of linear equations as follows:

$$\begin{pmatrix}
4\alpha+1 & -\alpha & & -\alpha & & & & & & \\
-\alpha & 4\alpha+1 & -\alpha & & -\alpha & & & & & \\
 & -\alpha & 4\alpha+1 & & & -\alpha & & & & \\
-\alpha & & & 4\alpha+1 & -\alpha & & -\alpha & & & \\
 & -\alpha & & -\alpha & 4\alpha+1 & -\alpha & & -\alpha & & \\
 & & -\alpha & & -\alpha & 2-4 & -\alpha & & -\alpha & \\
 & & & -\alpha & & & 4\alpha+1 & -\alpha & & \\
 & & & & -\alpha & & -\alpha & 4\alpha+1 & -\alpha & \\
 & & & & & -\alpha & & -\alpha & 4\alpha+1
\end{pmatrix} \cdot \begin{pmatrix} u^{t+1}_{0,0} \\ u^{t+1}_{1,0} \\ u^{t+1}_{2,0} \\ u^{t+1}_{0,1} \\ u^{t+1}_{1,1} \\ u^{t+1}_{2,1} \\ u^{t+1}_{0,2} \\ u^{t+1}_{1,2} \\ u^{t+1}_{2,2} \end{pmatrix} = \begin{pmatrix} c^t_{0,0} \\ c^t_{1,0} \\ c^t_{2,0} \\ c^t_{0,1} \\ c^t_{1,1} \\ c^t_{2,1} \\ c^t_{0,2} \\ c^t_{1,2} \\ c^t_{2,2} \end{pmatrix} \tag{4.10}$$

To solve this system of linear equations it takes more than a simple filter kernel. Using the linear algebra framework with the CG solver built-in ,however, only a few lines of code need to be changed. First the matrix has to be initialized differently. Therefore, it is sufficient to change the second line in Listing 4.1 to "`float alpha = -(dT*dT*C*C)/(h*h), mainDiag=-4*alpha+1;`." Finally we replace the per fame iteration by the following code:

```
1  // solve system
2  c->computeRHS(lastSurface, currentSurface);
3  iSteps = cgSolver->solve();
4  lastSurface->copyVector(currentSurface);
5  currentSurface->copyVector(nextSurface);
6
7  // display result
8  RenderTexturedQuad(currentSurface->m_pVectorTexture);
```

In this code fragment the right-hand-side vector is computed first, then the conjugate-gradient solver is called. During program initialization this solver class *C* is initialized with the matrix *m*, and the vectors *c* and *nextSurface*. After *iSteps* iterations the solver stores the result in the *nextSurface* vector. The next two lines swap the *last*, *current*, and *next* vectors. Finally, the texture associated with the current time-step is used to texture a quadrilateral (see Figure 4.2 for an example). Note that the framework effectively hides any GPU programming and allows the programmer to focus on the algorithm itself. By means of this simple example it should be clear that using a general linear algebra framework on the GPU is much more flexible, and less error prone than rewriting custom shaders for every new application.

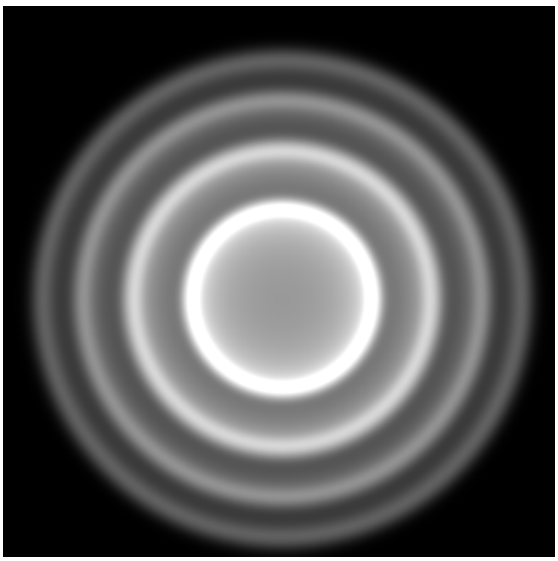


Figure 4.2: The image shows a quadrilateral textured with the solution vector of the water surface PDE. In this image the values in the vector texture are interpreted as grayscales.

The beauty of the wave equation is its simplicity both in terms of implementational and numerical complexity. The entire program to compute the images in Figure 4.2 requires only about 200 lines of code not counting the clFramework files and some DirectX initialization. The application can render the implicit water surface on an ATI X1900 XT graphics card [7] at more than 50 fps on a 1024×1024 grid. This means that a sparse linear system with more than a million unknowns is solved and the result is rendered to the screen in less than 20ms, all this on a commodity PC.

Regardless of the performance and simplicity, a clear disadvantage of this simple formulation of the wave equation is that it models only a 2D membrane. Therefore, in the next sections we take a look at the Navier-Stokes equations (NSE) for incompressible laminar fluids.

4.2 2D Fluid Simulation

The Navier-Stokes equations, named after the French engineer-physicist Claude Louis Marie Henri Navier and Irish mathematician-physicist George Gabriel Stokes, are a set of equations that describe

the motion of fluid substances. Fluids, in turn, are liquids, gasses or plasma (see Figure 4.3) or—as defined by Griebel et al. [37]—"Substances which, in contrast with solid materials, cannot resist shear stress when at rest." For these fluids, the Navier-Stokes equations constitute that changes in momentum are simply the product of changes in pressure and dissipative viscous forces acting inside the fluid.

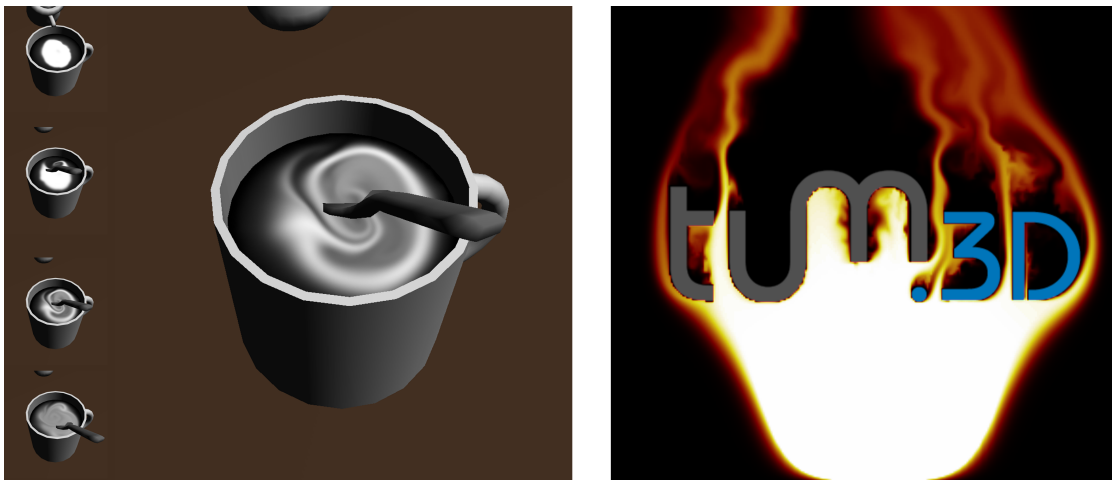


Figure 4.3: Images of an incompressible fluid are shown. On a 128×128 grid the simulation using five CG iterations is performed at about 600 fps on our target architecture.

Unfortunately so far no closed-form solution for the Navier-Stokes equation exists (see [32]). Therefore, the only known way to solve the equations is via means of computational fluid dynamics. In the following we focus on the simulation of incompressible, laminar flows as these have been shown to allow for the simulation of many fluid effects such as fire, water, or smoke (see Figure 4.3). To efficiently control vortex structures in flow we will later introduce the concept of pressure- and flow-templates (see Section 4.2.1).

In the following paragraph we will discuss the two-dimensional, unsteady, laminar flow as can be described by the following three equations: the two impulse equations for the two dimensions and one continuity equation, which enforces the flow to be divergence free[†].

$$\frac{\partial u}{\partial t} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) + f_x - \frac{\partial p}{\partial x} \tag{4.11}$$

$$\frac{\partial v}{\partial t} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) + f_y - \frac{\partial p}{\partial y} \tag{4.12}$$

$$\mathrm{div}(V) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{4.13}$$

The impulse equations are composed of the following four components, the diffusion $\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$,

[†]For a more thorough discussion of the Navier-Stokes equations and in particular the derivation of the equations themselves, we refer the reader to introductory books on CFD; e.g., Griebel et al.[37], Anderson [4].

the advection $u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}$, the external forces f_x, and a pressure gradient $\frac{\partial p}{\partial x}$.

As mentioned earlier, for the general case no closed-form solution for the Navier-Stokes equations exists, thus we solve the equations numerically. We therefore partition the domain into a regular lattice. To improve numerical stability we discretize the domain on a staggered grid shown in Figure 4.4. In this way, centered space derivatives use successive points of the same variable, and the dispersion characteristics are improved because the effective grid length is halved and pressure oscillation is avoided. Furthermore, we can now handle arbitrarily positioned obstacles exhibiting special boundary conditions such as free-slip and no-slip conditions very effectively.

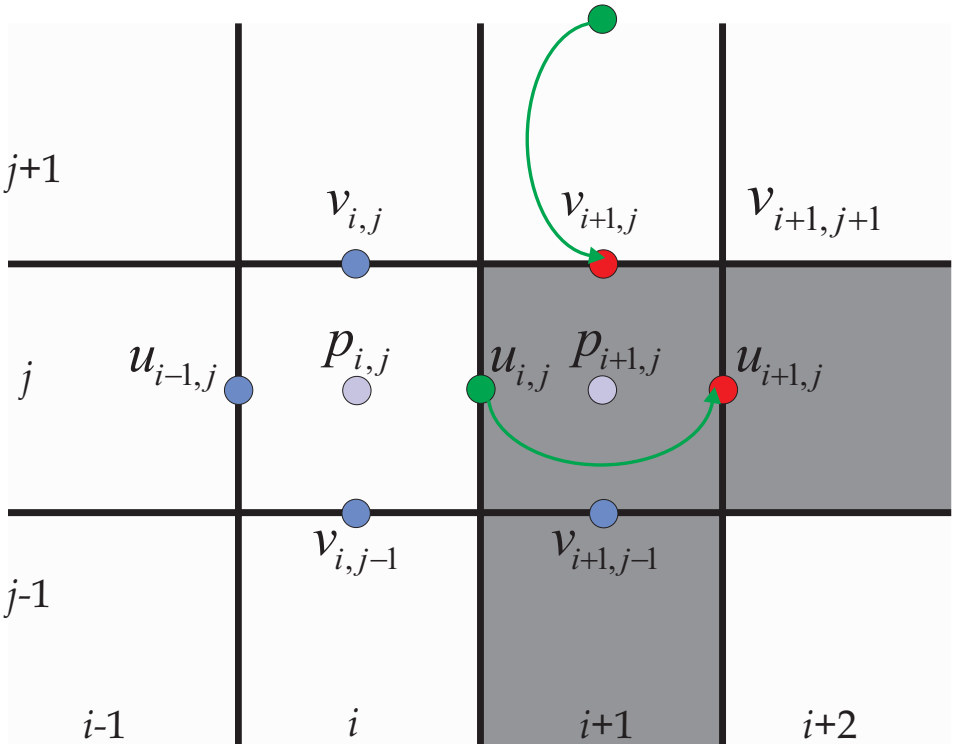


Figure 4.4: The image shows a staggered grid with the pressure values stored in the center of each cell, and the u and v components stored on the cell borders. This image also shows how the velocity values at obstacle (dark cells) borders are mirrored to match the border conditions.

By using a discretization on a regular lattice we can approximate $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$ first-order accurate with forward differences and get:

$$
\begin{aligned}
u^{t+1} &= F^t - \delta t \frac{\partial p}{\partial x} \\
v^{t+1} &= G^t - \delta t \frac{\partial p}{\partial y} \\
&\textbf{with} \\
F^t &= u^t + \delta t \left(\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \left(u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) + f_x \right) \\
G^t &= v^t + \delta t \left(\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \left(u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) + f_y \right)
\end{aligned}
\tag{4.14}
$$

We can now solve for the velocity $V = (u, v)^T$ in two passes: First, by ignoring the pressure term $\frac{\partial p}{\partial x}$ intermediate values F and G are computed. The diffusion operator is discretized by the means of central differences, and, as proposed by Stam [99], we solve for the advection part by tracing the velocity field backward in time. Let us pause a moment to consider the ramifications of the previous statements. By neglecting the pressure and computing F and G with the aforementioned methods we reduce the computation to an explicit filter kernel which again could be computed by either a set of custom shaders or the proposed GPU linear algebra framework. However, to compute the flow field itself we need to consider the third equation (4.13) in our solution. To solve for the pressure, we derive from $\mathrm{div}(V) = 0$:

$$
\mathrm{div}(V) = \frac{\partial u^{t+1}}{\partial x} + \frac{\partial v^{t+1}}{\partial y} = \frac{\partial F^t}{\partial x} - \delta t \frac{\partial^2 p}{\partial x^2} + \frac{\partial G^t}{\partial y} - \delta t \frac{\partial^2 p}{\partial y^2} = 0
$$

Discretizing the pressure using central differences and solving for p^{t+1} we option the following Poisson equation:

$$
\frac{p^{n+1}_{i-1,j} - 2p^{n+1}_{i,j} + p^{n+1}_{i+1,j}}{(\delta x)^2} + \frac{p^{n+1}_{i,j-1} - 2p^{n+1}_{i,j} + p^{n+1}_{i,j+1}}{(\delta y)^2} = \frac{1}{\delta t} \left(\frac{F^n_{i,j} - F^n_{i-1,j}}{\delta x} + \frac{G^n_{i,j} - G^n_{i,j-1}}{\delta y} \right) \tag{4.15}
$$

With the known right-hand side, the solution of the Poisson equation requires the linear system of equations to be solved. This operation can not be expressed by a filter kernel anymore, therefore we employ the conjugate gradient method of our linear algebra framework on the GPU. Finally, with the solution of the Poisson equation the F and G terms are updated according to Equation 4.14 and a new flow field is computed. Figure 4.5 depicts the entire 2D Navier-Stokes pipeline.

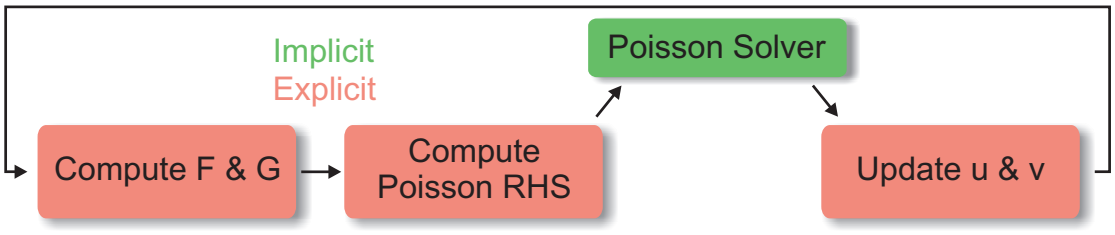


Figure 4.5: The principal 2D Navier-Stokes pipeline

In addition to the solution to the Navier-Stokes equation as described above we have integrated the following extensions (see Figure 4.6).

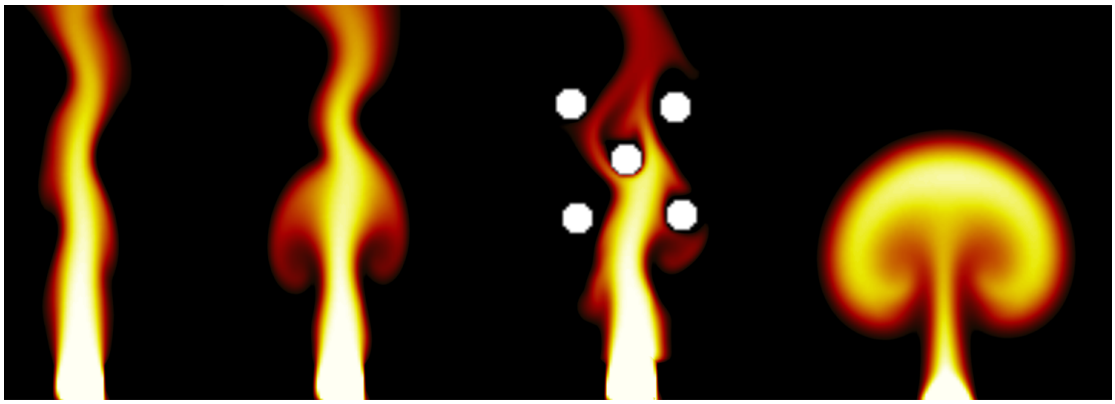


Figure 4.6: From left to right we show a simple inflow, the same flow with vorticity confinement enabled. The next image depicts a few obstacles placed in the flow while the rightmost picture shows a temperature driven inflow

- We have integrated a mechanism that allows one to arbitrarily specify inflow regions and characteristics. Therefore, the current inflow settings are stored in the color components of a 2D texture map, which is interpreted as external forces acting on the flow in every iteration of the simulation process.

- To preserve vorticity on the regular staggered grid we have integrated vorticity confinement [23] into our simulation code. At each grid point, a pixel shader computes

$$v_c = \tilde{n} \times (\nabla \times \omega) \tag{4.16}$$

  where $\tilde{n} = \nabla|\omega|/|\nabla|\omega||$ is a unit vector pointing towards the centroid of the vortical region, and ω is the vorticity vector. This vector is added to the velocity to convect vorticity towards the centroid (see Figure 4.7).

- We have integrated a dynamic time step control that makes sure that the Courant-Friedrichs-Levi (CFL) condition (cf., e.g., [37])

$$\Delta t = \tau \cdot \min\left(\frac{Re}{2}\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u_{mx}|}, \frac{\delta y}{|v_{mx}|}\right) \tag{4.17}$$

  is satisfied. This condition ensures that the velocity magnitude is not larger than the discretization step size, the additional safety factor $\tau \in]0, 1]$ compensates for numerical error due to limited precision.

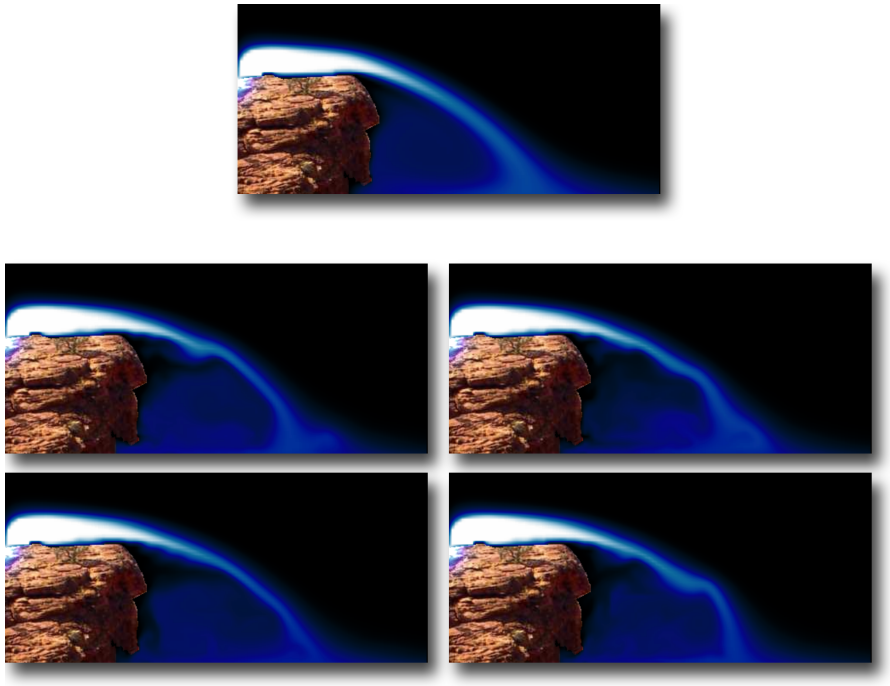


Figure 4.7: The images show a step flow. On top the static solution without vorticity confinement can be seen. In the four bottom images of the dynamic simulation with vorticity confinement is shown.

In the following listing we give pseudo-code for the GPU-fluid-solver.

```
1  t = 0;                              // initialize simulation time
2  InitializeDomain();
3  while (bSimulationIsRunning) {
4    if (bDynamicTimestep) dT = Compute_CFL_Timstep();
5    SetupDomainBorderUV();
6    SetupObstacleBorderUV();
7    Compute_F_and_G();                // LA Framework (diff., adv., ext. forces)
8    SetupDomainBorderFG();
9    SetupObstacleBorderFG();
10   ComputePressureRighHandSide();    // LA framework
11   SolvePoissonEquation();           // LA framework
12   UpdateUV();
13   t += dT;
14 }
```

Analogously to the previous chapter, we encapsulated the 2D Navier-Stokes solver in C++ classes (see UML Diagram 4.8). These classes include an extensible file format that allows us to share simulation

settings amongst all our applications that use the Navier-Stokes framework. This is of particular interest for the effects modeling tool presented in the next section and the re-usability of the project files created with this tool for all rendering applications presented in Chapter 5.

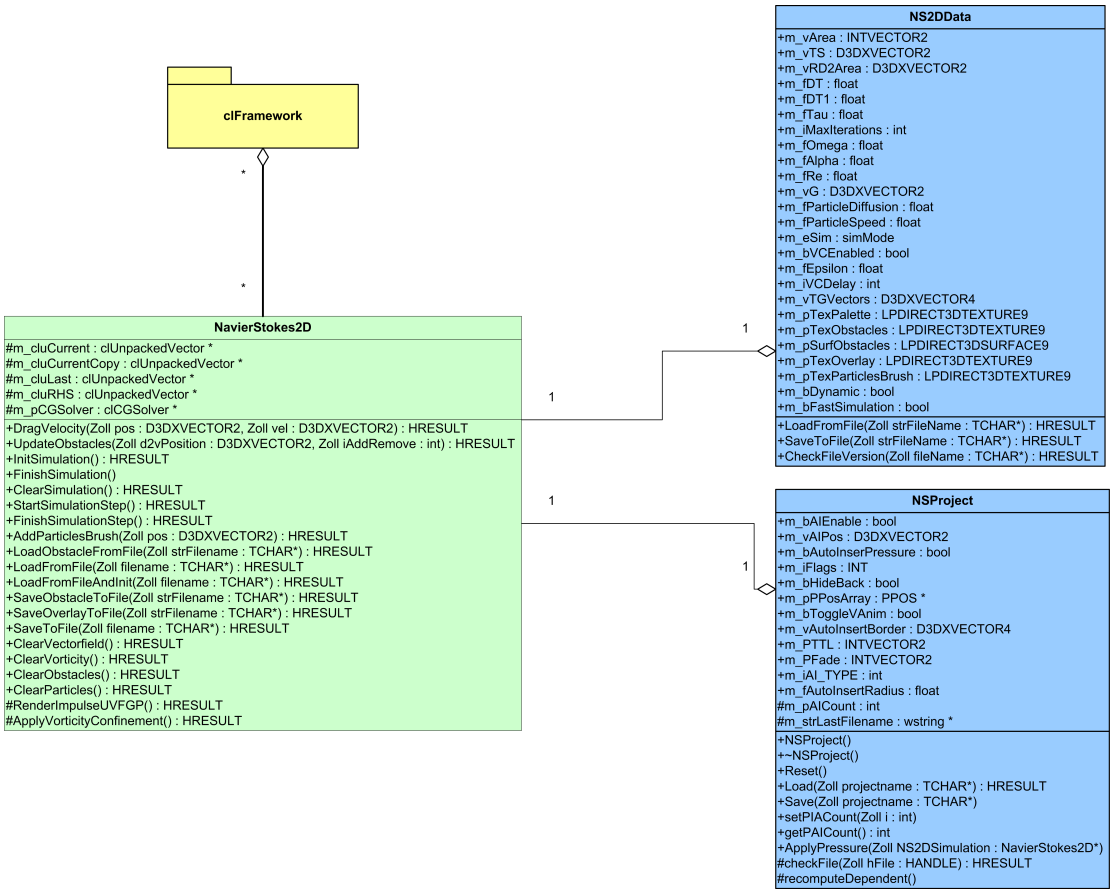


Figure 4.8: This figure shows a simplified UML diagram of the 2D Navier-Stokes classes. Note the connection to the clFramework (see Figure 3.10 for details) and the data storage classes colored blue.

Performance Measurements

In Table 4.1 we give timings for the presented 2D Navier-Stokes implementation for different grid sizes. A comparison with the latest GPU-based work on real-time fluid simulation using reduced models by Treuille [106] shows that our GPU approach is still a factor of three faster, even though we simulate the entire domain and thus avoid expensive preprocessing. This gives use the advantage to be able to handle arbitrary obstacle motion. Furthermore, all of our timings include not only the simulation but also dye advection and rendering. Finally, it is worth noting that the fact that our simulation results are computed and stored on the GPU gives us an advantage over any CPU simulation: Even if a CPU simulation algorithm should outperform our GPU-based solver, the GPU-based simulation would still

have to transfer the results over the CPU-GPU interconnection, which is very likely to remain a major bottleneck. Our tests show that on an entirely idle PC the CPU to GPU texture upload alone consumes about the same time as the entire simulation plus rendering on the GPU.

Grid Size	FPS
128×128	650
256×256	440
512×512	140
1024×1024	35
2048×2048	7.5

Table 4.1: The table gives performance measures for the 2D Navier-Stokes simulation including dye advection and rendering. The timings were conducted on a P4 3.0 GHz PC equipped with a GeForce 7900 GTX card [22] in every iteration, five CG steps were performed.

4.2.1 Effect Modeling with Flow Patterns

Ideally, the simulation of incompressible non-turbulent fluids using the Navier-Stokes equation produces physically accurate results. On the other hand, adjusting fluid parameters to achieve a particular effect can be complicated or even impossible due to restrictions of the underlying physical model. In particular, expansion and contraction effects are not covered by these equations. Additionally, due to the limited size of grids that can be used in real-time scenarios, highly detailed features can hardly be generated.

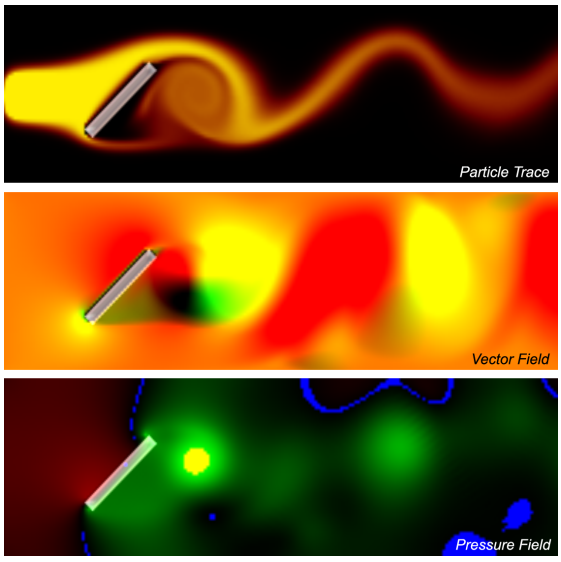


Figure 4.9: A vortex in the flow field, the vector field and the corresponding pressure field (red and green indicate high and low pressure regions, respectively).

To overcome the described limitations one could either perturb the resulting velocity field to introduce small-scale features [100, 89], or one could try to modify the simulation process in such a way

as to produce the desired features automatically. For this purpose we will now introduce parameterized templates that can be blended into the respective fields, which are then considered by the simulation. Internally the solver distinguishes between pressure templates—pre-computed pressure masks that are continually inserted into the pressure field *during* the solution of the Poisson equation—and velocity templates —vector field masks that are inserted as external forces into the velocity field.

The key observation that lends itself directly to pressure templates is that characteristic features in the vector field are related to characteristic features in the corresponding pressure field (see Figure 4.9 for an example). While pressure templates provide an intuitive way to model divergence phenomena, i.e., sinks and sources, direct modification of the vector field enables the user to intuitively add large-scale structures like vortices of particular shape and size.

The need for pressure templates in addition to impulse templates is motivated by the following observations: First, pressure is a comprehensible feature of space, and it is thus easy to imagine what would happen if the pressure at a certain point in space is changed—increasing pressure leads to a source in the field, whereas a decrease creates a sink. By modifying the pressure distribution within a region, a wide range of different velocity structures can be modeled (see Figure 4.10). Second, pressure templates allow for the creation of divergence effects that can not be efficiently and intuitively modeled using velocity templates. As pressure acts as a global correction term for the velocity field, divergence effects created by velocity templates will be destroyed by the Poisson solver. If we do not want the effect to be "corrected," direct modification of the pressure term turns out to be an effective mechanism.

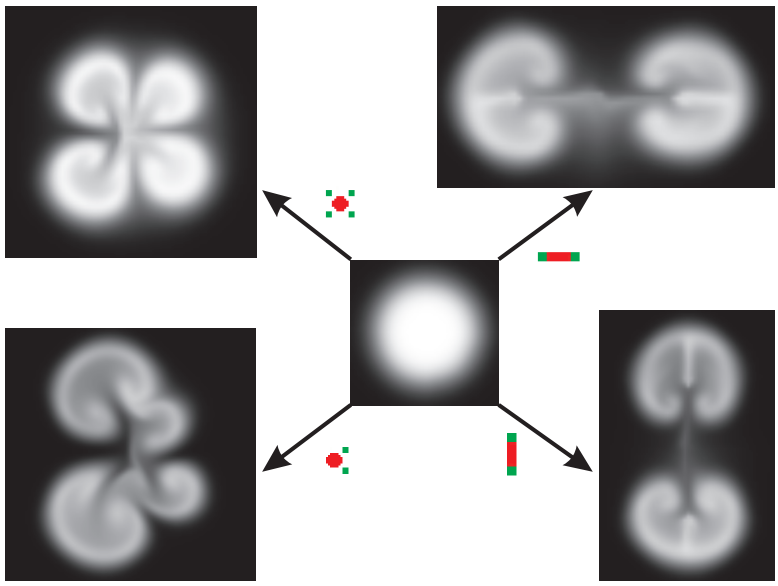


Figure 4.10: Velocity structures created by different pressure templates are shown. Pressure templates are shown in red/green indicating negative/positive pressure.

The system presented in this thesis provides the user with a set of effect templates that have been pre-computed, or which have been captured either from the pressure or the velocity field. To the user,

only iconic representations of the effects generated by a particular template are presented. The way these effects have been generated is hidden. The template processing is implemented as an extension to the 2D Navier-Stokes class hierarchy presented above. Note that we store all template parameters as well as the templates themselves in our extensible NS2D file format, thus allowing all other rendering applications to use the Navier-Stokes framework for the render of the designed flow features (see UML Diagram 4.11).

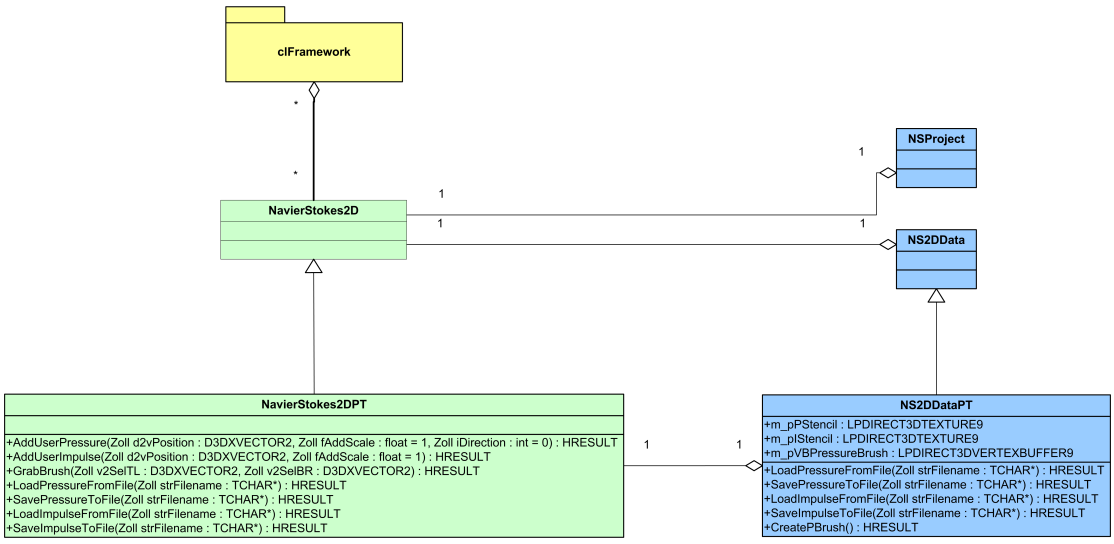


Figure 4.11: This figure shows a simplified UML diagram of the 2D Navier-Stokes classes supporting the flow templates.

User Interaction

Our approach enables the user to design different effects by capturing a template from the respective fields. By adding external forces via mouse drags or by applying templates from a library the desired flow structures are produced. It is possible to switch between different views showing either the vector field visualized by dye-advection or the pressure field. If a desired structure has been created, the simulation can be stopped allowing the template to be grabbed with a selection box. Depending on the current view, the system either captures a pressure- or an impulse-template. The current template becomes active and the user can blend it into the current field to analyze the effect. Selected templates are stored as RGB images in the template library, and they can be post-processed using imaging tools.

To model a particular phenomenon, the user has two possibilities: First, templates from the library can be pasted interactively into the current simulation (see Figure 4.12). Second, templates can be inserted automatically by the program into user-defined regions (see Figure 4.13 left). While the first approach is more adequate for prototyping and experimenting, the second approach enables the design of complex phenomena requiring only simple user interaction. In either case, templates can be arbitrarily transformed to an appropriate size and orientation, and values stored in a template can be scaled at run-time to produce more or less dominant effects. In particular, templates can be automatically aligned along pre-defined

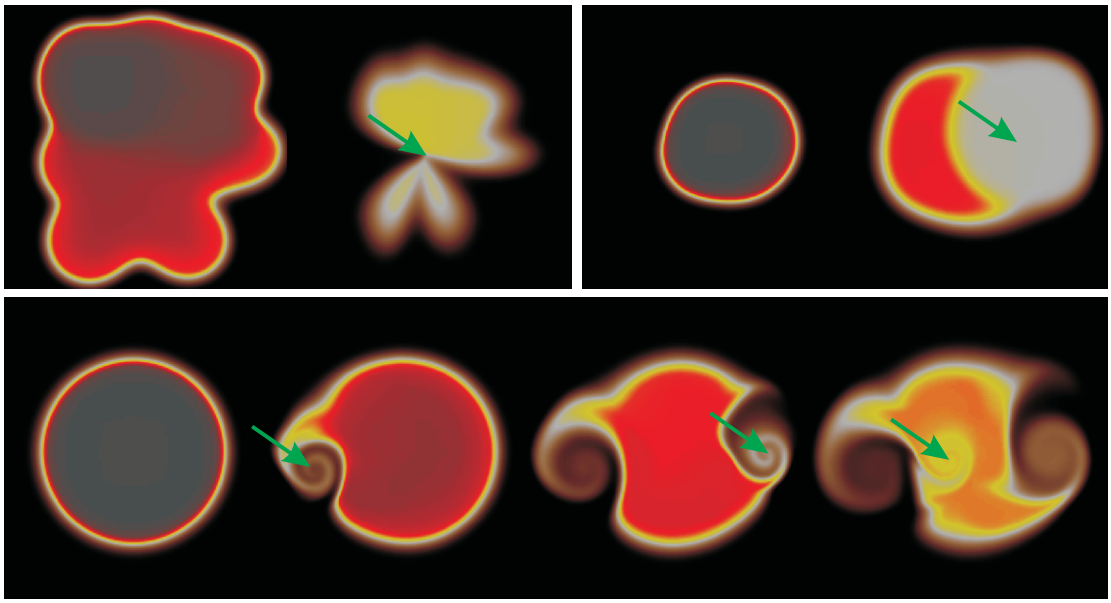


Figure 4.12: Effects of interactive template insertion are shown. The upper left image shows the effect of a sink template, in the right image a source template was applied. In the lower row an impulse template producing a vortex was inserted at different positions and orientations.

contours or to the shape of the insertion regions, e.g., to produce radially symmetric flow.

One important effect that needs to be paid attention to is the modification of inserted templates in each simulation step. If a template is only inserted once, it will be destroyed by the pressure-correction-term (see Equation 4.13) in the upcoming iteration. Consequently, templates need to be injected constantly into the field over a number of iterations. Only then will the respective flow structures be formed.

From this observation an additional parameter that controls the effect caused by a template is obtained. Templates can be kept in a certain region for a certain number of time steps. The longer the templates stay in the flow, the more dominant the resulting structures will appear and the longer their lifetime will be after they have been taken out of the flow. If multiple templates are applied simultaneously and their lifetime is stochastically varied between a minimum and a maximum lifetime, the frequency characteristics of the resulting structures in the velocity field can be varied. These effects are illustrated in Figure 4.13 on the right.

On the GPU, the insertion of templates can be handled very efficiently. An appropriately scaled and positioned quadrilateral, texture mapped with the desired template, is rendered into the respective field. In the pixel shader stage, scalar or vector values are fetched from the texture, and these values are then alpha blended with the simulated results. To generate continuous transitions at template boundaries, a Gaussian distribution of alpha values ranging from one in the center of the template to almost zero at the boundaries is used.

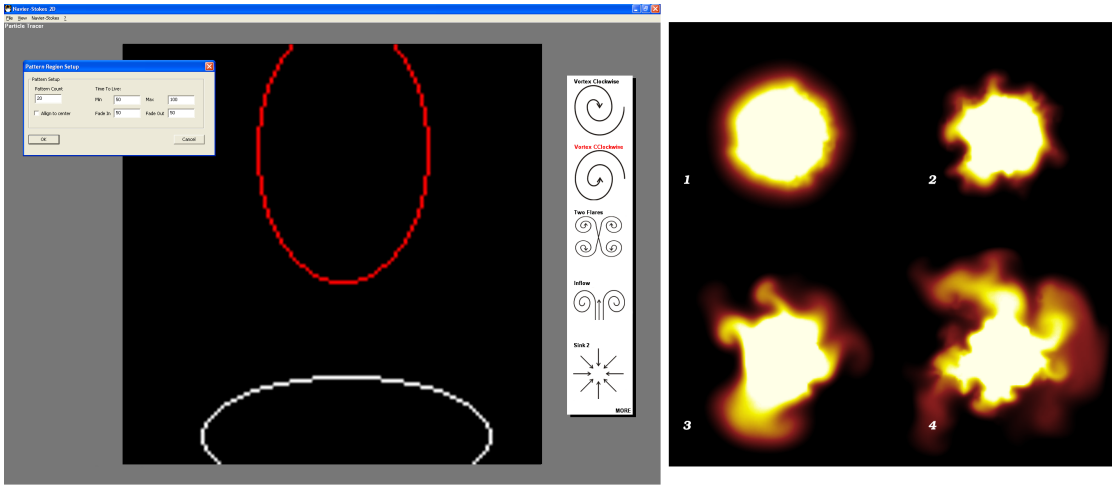


Figure 4.13: The left image shows the user interface to define regions for automatic insertion of templates. In the four right images, templates of equal shape and number are inserted. In images 1 to 3 we see high, medium, and low frequency structures. In image 4 all frequencies are contained. These effects are achieved by using different lifetimes and scale factors.

4.3 3D Fluid Simulation

Numerically the extension of the 2D Navier-Stokes equations to the 3D equations is canonical, by adding the third dimension as a fourth equation to the set of Equations 4.11-4.13, and by extending the other equations to three dimensions. Thus, for three dimensions we get the following equations:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)-\left(u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}\right)+f_x-\frac{\partial p}{\partial x}\\
\frac{\partial v}{\partial t} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}+\frac{\partial^2 v}{\partial z^2}\right)-\left(u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}\right)+f_y-\frac{\partial p}{\partial y}\\
\frac{\partial w}{\partial t} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}+\frac{\partial^2 w}{\partial z^2}\right)-\left(u\frac{\partial w}{\partial x}+v\frac{\partial w}{\partial y}+w\frac{\partial w}{\partial z}\right)+f_z-\frac{\partial p}{\partial z}\\
\mathrm{div}(V) &= \frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}=0
\end{aligned}$$

Applying the same approach as for the 2D Navier-Stokes equations in the previous section we now get the three F, G, and H terms, which can be computed in exactly the same way as described in Section 4.2.

$$\begin{aligned}
u^{t+1} &= F^t-\delta t\frac{\partial p}{\partial x}\\
v^{t+1} &= G^t-\delta t\frac{\partial p}{\partial y}\\
w^{t+1} &= H^t-\delta t\frac{\partial p}{\partial z}
\end{aligned}$$

with

$$F^t = u^t + \delta t\left(\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) - \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\right) + f_x\right)$$
$$G^t = v^t + \delta t\left(\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) - \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}\right) + f_y\right)$$
$$H^t = w^t + \delta t\left(\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) - \left(u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}\right) + f_z\right)$$

Finally we need to solve the 3D Poisson equation :

$$\frac{p^{n+1}_{i-1,j,k} - 2p^{n+1}_{i,j,k} + p^{n+1}_{i+1,j,k}}{(\delta x)^2} + \frac{p^{n+1}_{i,j-1,k} - 2p^{n+1}_{i,j,k} + p^{n+1}_{i,j+1,k}}{(\delta y)^2} + \frac{p^{n+1}_{i,j,k-1} - 2p^{n+1}_{i,j,k} + p^{n+1}_{i,j,k+1}}{(\delta z)^2} = \frac{1}{\delta t}\left(\frac{F^n_{i,j,k} - F^n_{i-1,j,k}}{\delta x} + \frac{G^n_{i,j,k} - G^n_{i,j-1,k}}{\delta y} + \frac{H^n_{i,j,k} - H^n_{i,j,k-1}}{\delta z}\right) \tag{4.18}$$

While theoretically the extension from two to three dimensions should not cause any problems, our target architecture imposes some restrictions on the 3D simulation. The first issue is imposed by the sheer size of the domain. Based on the observation that we compute a 128×128 Navier-Stokes simulation in 2D at about 650 fps on our target architecture, even theoretically a 128^3 simulation can not be expected to run faster than 5 fps. This is in contradiction to the interactivity requirements of our target application. The second issue, particularly on current GPUs, is the data structure required to represent a 3D grid. While current GPUs do support reading from 3D textures, no reliable and standardized way to write into a 3D texture exists yet[‡]. This leads to severe restrictions. Even the explicit computations of the terms F, G, and H can not be performed on a native 3D grid structure, let alone the writing into a 3D texture for rendering. Thus, a read/write 3D texture needs to be emulated by a large 2D texture atlas, which contains all the slices of the 3D grid (see Figure 4.14). While using a texture atlas is common in many GPU applications it imposes a few restrictions. Firstly, the texture coordinate conversion from 3D coordinates to 2D coordinates introduces some numerical overhead to every texture fetch into this texture. Secondly, trilinear interpolation has to by emulated by two 2D texture fetches and special care has to be taken to make sure that the borders are handled correctly. Thirdly, the limitation in size of 2D float textures, which is currently 2048×4096 restricts the 3D domain to 128^3.

Despite the aforementioned issues, we implemented the 3D Navier-Stokes equations on the GPU with an application that uses the described 2D texture atlas to represent the 3D domain. Therefore, we extended code for the computation of the F and G terms by adding a new pass to compute the H term. For the solution of the Poisson Equation 4.18 we use our linear algebra framework again. As the 3D Poisson equation differs from Equation 4.15 only in the computation of the right hand-side and in the fact that the resulting matrix has seven instead of only five bands, most of the computation code for 2D can be reused. Note that due to our 3D domain being stored in a 2D atlas, no data conversion is required for the linear algebra framework. The whole domain is simply treated as one long serialized vector stored in a 2D texture, which is exactly what the atlas does.

[‡]With the introduction of the Frame Buffer Objects (FBO) extension in OpenGL [5] a functionality to write into 2D slices of a 3D texture has been added. However, this extension is only rudimentary supported by the current GPUs. This will change in the near future with the upcoming DirectX 10 API that requires any DX10 capable GPU to allow writing to a 2D surface of a 3D texture; however, no DX10 capable GPU is currently available.

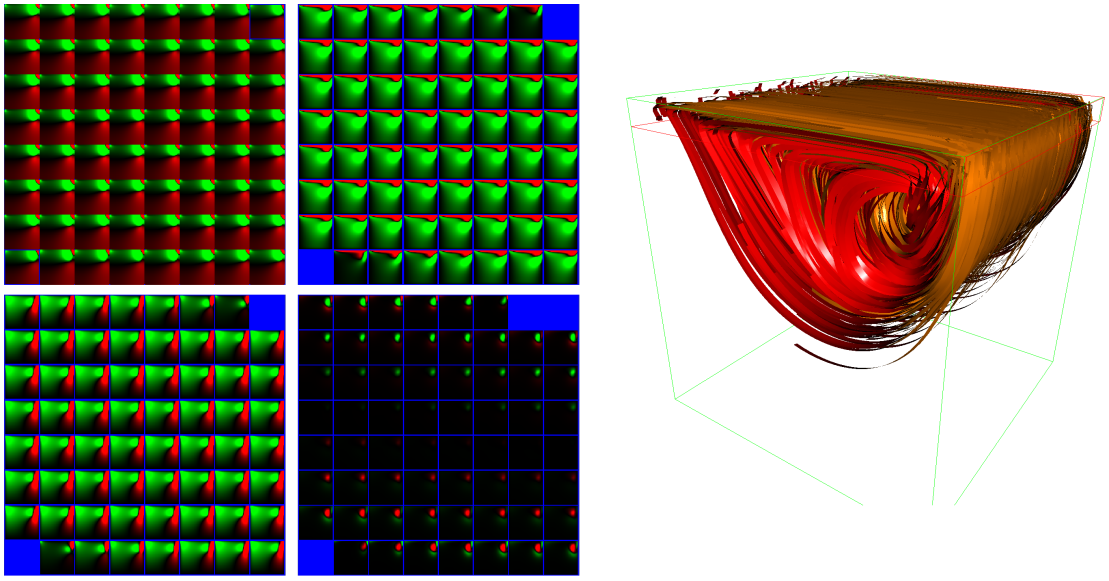


Figure 4.14: A flow inside a container with the lid moving at constant velocity. On the left the four atlases for pressure, x-, y-, and z-flow are shown. Positive values are color-coded in red, negative values are coded green. The right image shows a stream-ribbon visualization of the same simulation.

Performance Measurements

In Table 4.2 we give timings for the presented 3D implementation of the Navier-Stokes fluid simulation for different grid sizes. These timings indicate that by means of GPU-based numerical techniques it is now possible to interactively simulate the evolution of 3D flows on commodity PCs. A comparison with the latest GPU-based work on real-time fluid simulation using reduced models as proposed by Treuille [106] reveals that our GPU approach achieves competitive performance—although claimed otherwise in that paper. As with the 2D simulation it is worth mentioning, however, that in contrast to that work our method does neither rely on a reduced model formulation nor does it require extensive preprocessing. Thus, our method allows for arbitrary real-time interactions with the flow.

Grid Size	FPS
32^3	35
64^3	20
128^3	4

Table 4.2: The table gives performance timings for the 3D Navier-Stokes simulation. The timings were conducted on a P4 3.0 GHz PC equipped with a GeForce 7900 GTX card. For all sizes we ran the driven cavity simulation while performing nine CG steps per frame to solve the Poisson equation.

Despite the impressive overall performance our timings indicate that a full 3D simulation with the current generation of GPUs does not meet the extremely strict performance requirements of a computer game or a virtual environment. In these domains update rates above 60 fps are required. However, with

the rapid increase of performance and the DX10 capabilities to efficiently update 3D textures in mind our 3D simulations may be well suited for the use in computer games and interactive environments on next generation graphics hardware, which can be expected as soon as late 2006 to early 2007. For the meantime—until a full 3D simulation can be executed fast enough for a computer game or interactive environment—we focus our attention on methods to create pseudo-3D phenomena by means of less computational intense techniques. We achieve this with a small set of 2D simulations, combined into a $2\frac{1}{2}$D simulation, allowing for the simulation and rendering of 3D effects in real-time on current generation of GPUs.

4.4 $2\frac{1}{2}$D Fluid Simulation

While conceptually neither the pressure-templates method nor the GPU-Navier-Stokes Equation-framework is restricted to two dimensions, a full direct simulation in 3D is not practical for grid sizes larger than 32^3 in real-time environments due to computational and memory requirements (see Section 4.3 for detailed performance measures). Thus, we restrict ourselves to the simulation of effects that exhibit a rotationally symmetric appearance. In this case, the simulation can be restricted to a few two-dimensional slices, from which the volume is extruded. This idea is similar to the work proposed by Rasmussen et al. [89].

The simulation loop starts by computing multiple independent two-dimensional fluid simulations; to produce similar structures in each slice, only slightly different initial conditions are specified. Volumetric effects can now be generated by spherical linear interpolation (called revolution) between these two-dimensional simulations (see Figure 4.15).

For every data point in the 3D domain the projection along the circular arc onto the closest two slices is computed. At these projection points the 2D simulation slices are sampled. Reconstructed values are interpolated with regard to their circular arc distance from the original point in 3D space. Since this interpolation does not introduce new features along the circular arc, it results in a symmetric result. For the spherical interpolation of dye these interpolation artifacts are avoided by perturbing the initial point coordinate by a stochastic fractal distribution. This leads to the following compute kernel for every point in the domain:

- The initial point position is disturbed by a small turbulence offset, which is fetched from a 3D hypertexture
- The point is radially projected onto the closest simulation slices
- Respective values are fetched from both slices
- Spherical linear interpolation is performed

This basic idea of this revolution is depicted in Figure 4.15 for two slices.

This spherical interpolation can be computed surprisingly simple in a pixel shader. The following HLSL code demonstrates this for two axis aligned slices (aligned with the xy and the yz plane). The

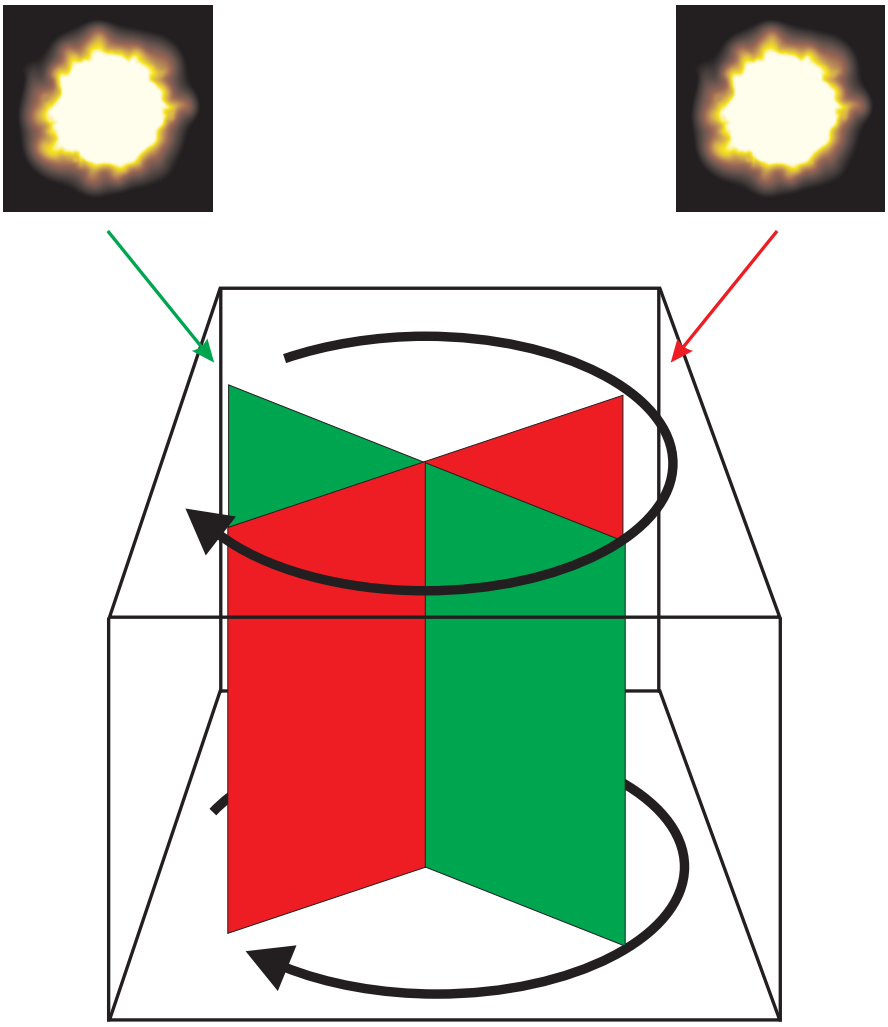


Figure 4.15: The revolution of 2D simulation results to 3D is illustrated.

shader snippet takes the perturbed 3D position `volPos` in the range $[-1..1]$ as input and outputs two 2D vectors `texA` and `texB` in $[0..1]$ for a texture-lookup into the two slices.

```
1  // compute distance to center on (x,z)-plane
2  float l = length(VolCoords.xz);
3
4  // compute texture coordinates and re-scale to [0..1]
5  float2 texA = (float2(l*sign(VolCoords.x), y) + 1) * 0.5;
6  float2 texB = (float2(l*sign(VolCoords.z), y) + 1) * 0.5;
```

With lines 5 and 6 being executed simultaneously due to the vector processing nature of the GPUs the

entire revolution code requires only two multiplications, one addition, a `sign` and a `length` call. Thus, it is not surprising that this pixel shader introduces no measurable overhead during the rendering of the volumetric data as described later in Section 5.2.

4.4.1 Example Effects

To generate effects like explosions, fire and smoke, the GPU engine simulates flow dynamics using different flow and template parameters. Figure 4.16 shows the generated phenomena after 2D simulation results have been revolved to 3D. In contrast to low speed events such as fire, flames and smoke, for the realistic simulation of explosions compressible events like expansion and contraction are important. Nevertheless, in all scenarios the incompressible Navier-Stokes equations in combination with pressure templates are employed.

Explosions: For modeling explosions, we simulate a vast amount of energy that is instantaneously liberated by initially setting the temperature in near-by regions of the center of the explosion to be very high. In subsequent simulation steps the temperature of injected heat is decreased. The buoyancy force is computed and added as an external force. By using appropriate pressure templates, bigger structures are created in the very heart of the explosion while they slowly fade out towards the outer regions and finally disappear outside the explosion. As we continually inject dye at the source of the explosion, the dye density is used as scaling parameter for the pressure templates.

Fire Ball: To realistically simulate a corona, templates are positioned at the surface. In addition, they are oriented such as to generate flow structures moving away from the center of the corona. Buoyancy is not considered in this case. To simulate high frequency structures giving rise to a pumping like behavior, only for a short period rather small templates are inserted. These templates, however, are scaled by a large factor to amplify their effect.

Smoke: Smoke rising from the cigarette exhibits all frequency structures in the upper part, where cooling has already taken place. The rising hot air in the lower part is not modified. In order to prevent regular vortex-like structures as in the explosion, much smaller templates with a short lifetime are specified. In this example, heat is continuously injected and templates are scaled by temperature. Additionally, an outflow condition is specified at the upper boundary of the domain, and the Reynolds number is decreased to simulate less viscous material. In this way we achieve a much straighter and faster upward flow.

Camp Fire: To simulate a camp fire, we use many small templates at the base of the fire to disturb the otherwise too homogenous flow. Larger templates are used in the upper part of the simulation grid where the fire turns into smoke. Like in the cigarette smoke example, heat is constantly injected and templates are scaled by temperature.

4.5 Particle Tracing

In this section we describe the principal techniques of GPU-based particle tracing. Although particle tracing in general is a visualization technique we put this section in the simulation chapter since the process of tracking an element through a given flow requires the solution to an Ordinary Differential

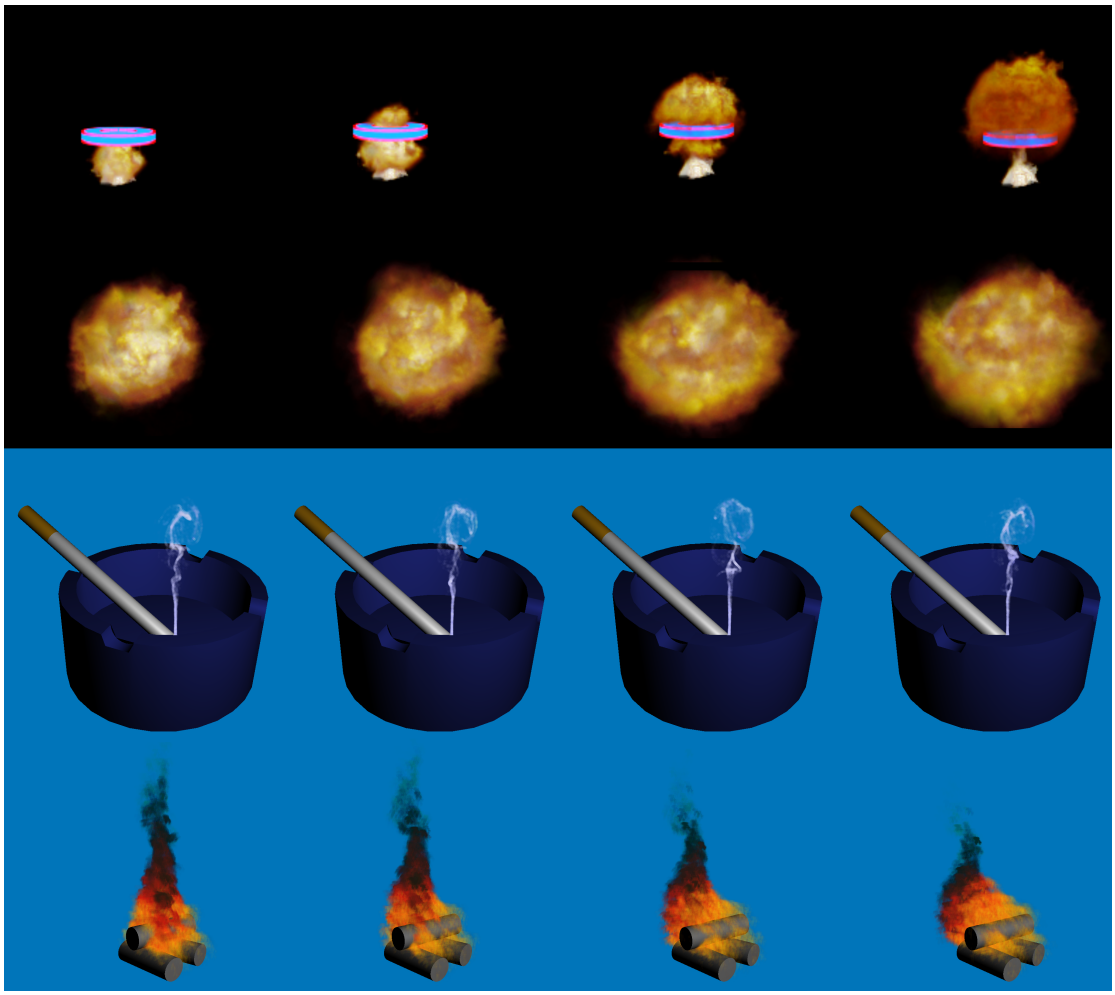


Figure 4.16: From left to right we see the time evolution of different phenomena as they are generated by the GPU simulation engine.

Equation (ODE), thus we consider the tracing of the particle rather simulation than rendering. The rendering of primitives, however, is described in detail later in Section 5.3. In that section we will use point primitives to visualize fire, smoke, and other phenomena. Before we discuss the rendering of the particles themselves, we need to compute the motion of particles within the flow field. This is where particle-tracing ODEs come into play.

In general, particle tracing is a technique for computing the trajectory of massless particles in a flow field over time. In classical particle tracing the ordinary differential equation

$$\frac{\partial \tilde{x}}{\partial t} = \tilde{v}(\tilde{x}(t), t) \tag{4.19}$$

equipped with an appropriate initial condition $\tilde{x}(0) = x_0$ is solved numerically. Here, $\tilde{x}(t)$ is the time-varying particle position, $\frac{\partial \tilde{x}}{\partial t}$ is the tangent to the particle trajectory, and $\tilde{v}$ is an approximation to the real vector field v. As v is sampled on a discrete lattice, interpolation must be performed to reconstruct particle velocities along their characteristic lines. The higher the approximation order of the integration scheme, the more often the interpolation function has to be evaluated. Consequently, the number of both numerical operations and memory access operations increases in a higher order setting. Additionally, in a single integration step the memory footprint is enlarged, letting higher order schemes become less efficient in terms of memory cache coherence.

Besides fixed step size integration schemes like classical Euler or Runge-Kutta (see Equations 4.20 and 4.21), embedded schemes are known to yield superior results both with respect to accuracy and speed. In embedded schemes, the local integration error is used to refine or to enlarge the integration step size. In particular, RK3(2) [11] (see Equation 4.22) computes a third and a second order solution and estimates a fourth order accurate local truncation error from them. Using the truncation error, the optimal step size can be computed. If it is larger than the current step size, the third order solution is taken and the next integration step will use the larger step size. If it is smaller, a third order solution using the reduced step size is recomputed.

Euler:

$$x_{t+1} = x_t + \Delta t \cdot f'(t, x_j) \tag{4.20}$$

Classical Runge-Kutta:

$$x_{t+1} = x_t + \frac{\Delta t}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4) \tag{4.21}$$

where

$$\begin{aligned} k_1 &= f'(t, x_j) \\ k_2 &= f'(t + \tfrac{\Delta t}{2}, x_j + \tfrac{\Delta t}{2} k_1) \\ k_3 &= f'(t + \tfrac{\Delta t}{2}, x_j + \tfrac{\Delta t}{2} k_2) \\ k_4 &= f'(t + \Delta t, x_j + \Delta t\, k_3) \end{aligned}$$

Bogacki-Shampine RK3(2):

$$\begin{aligned} x_{t+1} &= x_t + \tfrac{\Delta t}{9} \cdot (2k_1 + 3k_2 + 4k_3) \\ \tilde{x}_{t+1} &= x_t + \tfrac{\Delta t}{24} \cdot (7k_1 + 6k_2 + 8k_3 + 3x_{t+1}) \\ \text{Error} &= |x_{t+1} - \tilde{x}_{t+1}| \end{aligned} \tag{4.22}$$

where

$$\begin{aligned} k_1 &= f'(t, x_j) \\ k_2 &= f'(t + \tfrac{\Delta t}{2}, x_j + \tfrac{\Delta t}{2} k_1) \\ k_3 &= f'(t + \tfrac{3\Delta t}{4}, x_j + \tfrac{3\Delta t}{4} k_2) \end{aligned}$$

For particle visualization in general and in particular for the rendering of fluid effects, one is interested in continuous animation of particle sets within constant time intervals. Consequently adaptive schemes are only of limited relevance in this application. Consecutive particle positions have to be displayed at equally spaced simulation time steps. Therefore, adaptive schemes require varying numbers of operations to be carried out per particle. In a single animation frame, some particles perform many small integration steps while others have to repeat the integration step with reduced step size. This approach, however, does not map well to the SIMD compute model of the GPU. Therefore, we implemented both simple Euler and an embedded RK3(2) scheme for numerical particle integration. For the RK3(2) schema we use the local integration error as an additional visual cue. The proposed particle system provides a visualization mode that enables the user to visualize the local per particle truncation error using the third and second order solution to the current particle position, respectively. This visualization allows the effect designer to reduce the global integration time step to make sure the particle motion reproduces all the fluid flow effects.

4.5.1 GPU Particle Tracing

To understand the reasons why particle tracing maps well to a SIMD architecture and in particular to the GPU let us first take a look at the pseudo-code for a basic particle engine.

```
1  Initialize(particles);
2  while (SimulationIsRunning()){
3     for each (Particle p in particles){
4        // particle incarnation
5        if (p.lifetime == 0) Restart(p);
6        p.lifetime- -;
7        // particle advection
8        Vector v = SampleFlowfield(Field, p.Pos);
9        p.pos += v * deltaT;
10       // particle rendering
11       Draw(p);
12    }
13 }
```

This short pseudo-code snippet forms the core of practically any particle engine and contains prototypes for the three major components of a particle tracing algorithm.

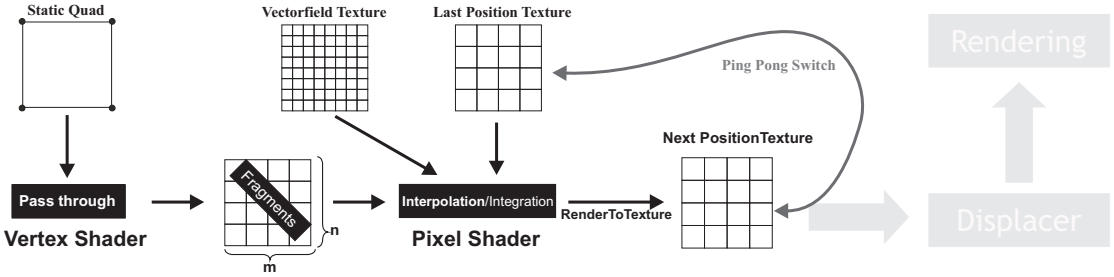


Figure 4.17: GPU Particle Advection. The black part of the image depicts the general GPGPU setup used for particle advection. The pixel shader takes as input the vector field and the current positions of the particles and computes the new advected positions. These positions can be used to drive a particle rendering engine (gray blocks, explained later in Section 5.3). In the next iteration the position textures are swapped and the advection process is repeated.

- *Particle Incarnation*: Any particle needs to be initialized to a ceratin state; usually this state needs to contain the position of the particle and a particle lifetime counter. If this counter reaches zero the particle is re-spawned at a new position.

- *Particle Advection*: In the particle advection state, the particles are advected through the flow by first interpolating the flow field and performing the numerical integration to the new position (in the pseudo-code above Euler integration was used).

- *Particle Rendering*: Finally the particles need to be displayed to visualize the flow dynamics.

These three steps are executed for every particle in the simulation independently, thus making this algorithm a perfect match for any SIMD architecture such as the GPU. Figure 4.17 gives an overview of our GPU particle tracing approach.

Throughout the tracing algorithm we use the same GPGPU full-screen quad setup as presented in Section 3.1. The computation is started by rendering a view-port covering geometry and having the rasterizer generate a fragment for every particle. Thus, the body of the *foreach* loop in the above listing is executed in the pixel shader for all particles simultaneously.

Particle Initialization

In our application initial particle positions are stored in the RGB color components of a floating point texture. These positions can be arbitrarily distributed in the flow domain and it is up to the effects designer to select a distribution. The visual effect of different distributions is demonstrated in Figure 4.18. In the alpha component, each particle carries a random floating point value that is uniformly distributed in the range of $(1.0 - v, 1.0 + v)$ where v is a user-defined variance value. This alpha-component is multiplied by another user-defined global lifetime l to give each particle an individual lifetime. With these two values l and v the effect designer can control the lifetime behavior of the particles in a sophisticated way. By letting particles die—and thus reincarnate—after different numbers of time steps controlled by v, particle

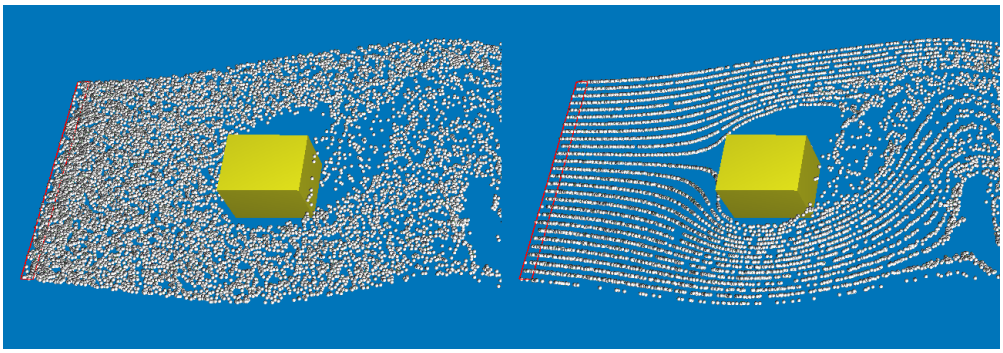


Figure 4.18: Random and Regular Distribution of Starting Positions Within the Particle Probe.

distributions very similar to those generated by real-world phenomena such as a wind tunnel can be simulated.

Finally the effect designer can specify the number m of particles continuously released into the flow. Once this number is changed, a particle texture of size $m = M \times N$ is generated on the CPU and it is uploaded to the GPU. Particle integration now consists of two steps: incarnation and advection.

Particle Incarnation

In every simulation pass, a stream of $M \times N$ fragments corresponding to a contiguous block of pixels in screen space is generated. Figure 4.17 illustrates this procedure. The pixel shader output is rendered to an equally sized texture render target, which becomes the particle container in the next pass. The initial particle container is bound to the first texture unit; thus, enabling access to initial particle attributes in a pixel shader. In the first simulation pass, the shader performs the following operations:

- *Birth*: Each particle is born by initializing its lifetime as described. The maximum lifetime is presented to the shader via a uniform parameter.
- *Update*: Updated particle coordinates and the initial lifetime are output to the RGB and alpha components of the render target, respectively.

The 2D render target now contains for every particle both its current local object-space coordinate and lifetime. In the advection step these values are changed subsequently.

Particle Advection

Particle advection is again performed in the pixel shader units. The above procedure is employed to generate $M \times N$ fragments, which read current particle positions, advect these positions and store the results in an additional texture. Besides the initial particle container, the current container and the texture storing the velocity field are bound to a second and third texture unit. This velocity field can be anything for a 2D Navier-Stokes simulation or multiple 2D slices from the $2\frac{1}{2}$D simulation, a texture atlas from

the 3D simulation, or a single 3D texture from a pre-computed flow field. In all cases vector valued information is stored in the RGB color components of the textures. The only difference is the interpolation kernel used to reconstruct values from a particular position from these containers. In the pixel shader the following operations are carried out:

- *Texture Access*: Current particle positions and lifetimes are read from the texture unit.
- *Death Test*: The pixel shader checks for positions outside the domain or lifetimes equal to zero. If one of these conditions is true, the particle is reincarnated. Otherwise, it is advected through the flow.
- *Advection*: Particles are advected using either the Euler or Runge-Kutta integration scheme, which involves multiple fetches into the flow field textures. In addition, the lifetime of each particle is decremented by one. Updated positions are written to a 2D texture render target, which becomes the particle container in the next pass.
- *Reincarnation*: The same operations as described above for particle incarnation are carried out.

In HLSL these operation can be efficiently expressed within the pixel shader. The following listing shows the `EulerAdvection` shader code, which calls `SampleField3D` to fetch from a 3D texture map and uses `OutOfGrid` to check if the particle has left the domain. Note how `EulerAdvection` closely matches the loop in the listing of the principal particle engine above.

```
1  // GPU Euler Advection
2  float4 EulerAdvection(float2 texCoords) :  COLOR0 {
3    float4 pos = tex2D(sPositions,texCoords);
4    float3 field = SampleField3D(pos.xyz);
5    pos += float4(field*deltaT,-1);
6    if (pos.w <= 0 || OutOfGrid(pos.xyz))
7      pos = tex2D(sStartPositions,texCoords)*float4(1,1,1,lifetime);
8    return pos;
9  }
10 // fetch from 3D flow field
11 float3 SampleField3D(float3 pos){
12   return tex3D (flowFieldTex, pos);
13 }
14 // check if position is within [0..1]^3
15 bool OutOfGrid(float3 position) {
16   float p1 = dot(float3(1,1,1), position);
17   float p2 = dot(float3(1,1,1),saturate(position));
18   return ( p1 != p2 );
19 }
```

Chapter 5

Rendering

In the previous chapters we have introduced the linear algebra framework to efficiently handle vectors and matrices on the GPU. Furthermore, we used this framework to interactively simulate a variety of fluid effects. Finally, we discussed methods to trace massive amounts of particles at interactive rates through flow fields. Yet, we did not consider the question how to generate 3D imagery of these fluids. In this chapter we will explain three methods to visualize the simulation results.

5.1 Texture-Based Rendering

One of the key advantages of the proposed GPU-based simulation system is the fact that all simulation data is stored directly in GPU memory. To display intermediate values or the final simulation results no transfer over the GPU-CPU bus is necessary. Data sets that can directly be visualized are the height-field of the water surface simulation as well as the pressure distribution from the 2D Navier-Stokes simulation, the F and G terms*, and the Navier-Stokes flow field itself. In these cases, visualization simply means that only a textured quad has to be drawn (see Figures 5.1 and 5.2).

In addition to these quantities, which are directly available for visualization, many derived quantities

*see Chapter 4 for details on these terms

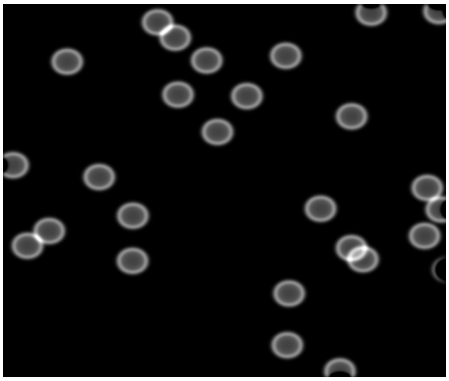


Figure 5.1: Rendering of the 2D height map that was computed by the water surface simulation.

in the 2D Navier-Stokes simulation, such as the rotation of the flow field, require only simple pixel shaders to be executed during the rendering of the quadrilateral (see Figure 5.2 left image).

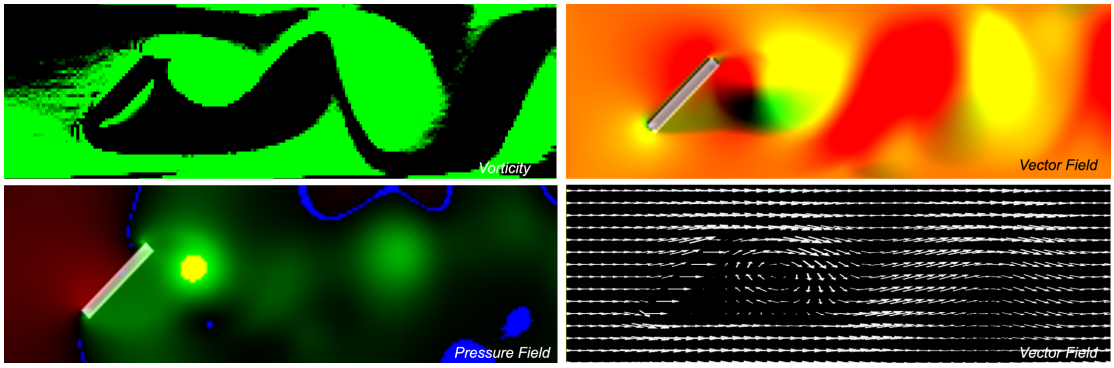


Figure 5.2: On the left side the vorticity, pressure distribution, and on the right side the flow field itself is shown. The bottom right image shows a particle-glyph based visualization of the same flow field, for comparison.

While all of these 2D images are helpful for both educational purposes and flow field design, their use is mostly restricted to these areas of application. For the rendering in computer games or virtual environments more realistic render modes are required. Such effects can be achieved by applying additional shaders on the 2D pixel data. For the water surface a refraction shader is added to the renderer. This shader computes—based on the view direction and the surface normal—the refracted ray. This ray is used in turn to compute the intersection with a second texture map located below the water surface (see Figure 5.3).

For flow fields a texture-based dye advection (see Figure 5.4) is the most intuitive mode. The implementation of the dye advection is similar to the diffusion and advection steps in the Navier-Stokes simulation itself. However, this time density values in a texture are advected by the flow. To compute the dye advection an additional 2D texture is used which stores density values added by the user. On this

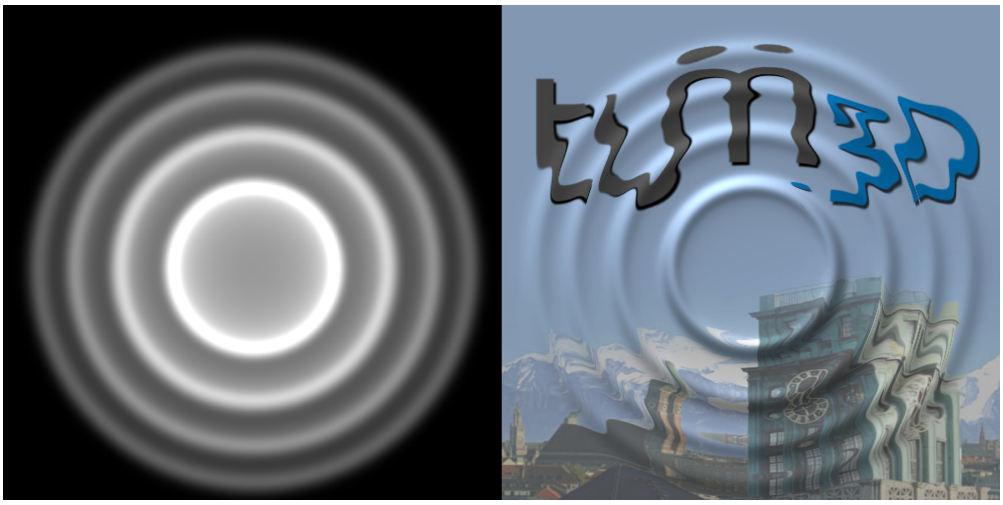


Figure 5.3: The left image shows the vector texture directly while on the right a refraction shader was used.

additional texture diffusion and advection using the 2D flow is performed. These density values, in turn, can be directly texture-mapped to a quad. Integrated into our 2D Navier-Stokes rendering system with an intuitive mouse interface that allows for dye insertion on a left mouse click and for the application of external forces with a right mouse drag we get a powerful 2D flow simulation and visualization program that can be used even by non-experts.

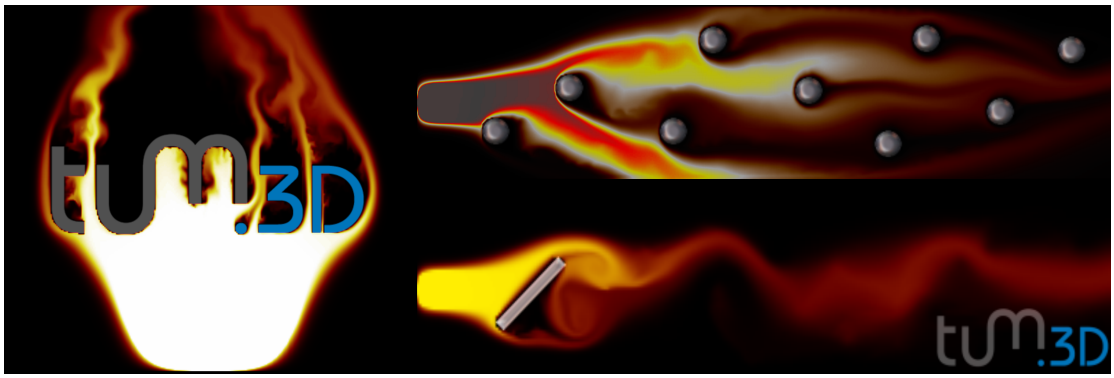


Figure 5.4: 2D texture-based rendering using dye advection

Although simple in nature, direct texture rendering in combination with texture-based dye advection is already sufficient for a wide range of applications. Examples of real-live demos are given in Figure 5.5 and 5.6. The first image shows the integration of the dye advection into a teapot scene. This effect integrated in a computer game or virtual environment allows for a rich user/scene interaction and can thus increase the level of realism in the environment.

The second example shows a photo of an user interacting with the modified version of the 2D flow

Figure 5.5: In this demo the user can insert milk into a cup of coffee and stir it with a spoon. In this case a plausible fluid behavior can be conveyed by only using a 2D simulation, and by mapping the images of the advected dye onto the coffee's surface.

advection program. In this version the application of external forces is done by camera based motion tracking.

By using this program the user can interact directly with the flow. By waving his arms or by simply "walking through" the domain external forces are applied to the simulation. The system works as follows: In an initial calibration phase the program renders a checkerboard pattern and determines where in the camera image the projected dye advection is located. For recognition of the pattern we employed Intel's OpenCV toolkit [19]. After this phase we track the motion in front of the dye-advection quadrilateral by means the GPU-PIV framework provided by Schiwietz and Westermann [93]. This framework, which runs entirely on the GPU, takes two images as input, computes the cross-correlation and outputs a 2D motion vector field into a texture. This vector field in turn is used as external force added to the simulation to advect the flow. To make sure that the camera captures only the motion of the objects in front of the dye advection and not the dye motion itself a color mask inverse to the dye color is used, i.e., for motion estimation only the blue channel of the camera image is used while we render the dye in red, green, or yellow.

Despite its advantages in terms of performance and simplicity direct 2D texture-based rendering is restricted to 2D flow fields. Although 3D flow phenomena can sometimes be augmented by 2D surfaces or 2D imposter rendering, these techniques often suffer from a loss in realism in animated scenes and are often not applicable to environments with stereo rendering capabilities such as virtual environments. Therefore, we propose more powerful techniques for the true 3D rendering of 3D flows in the following sections.

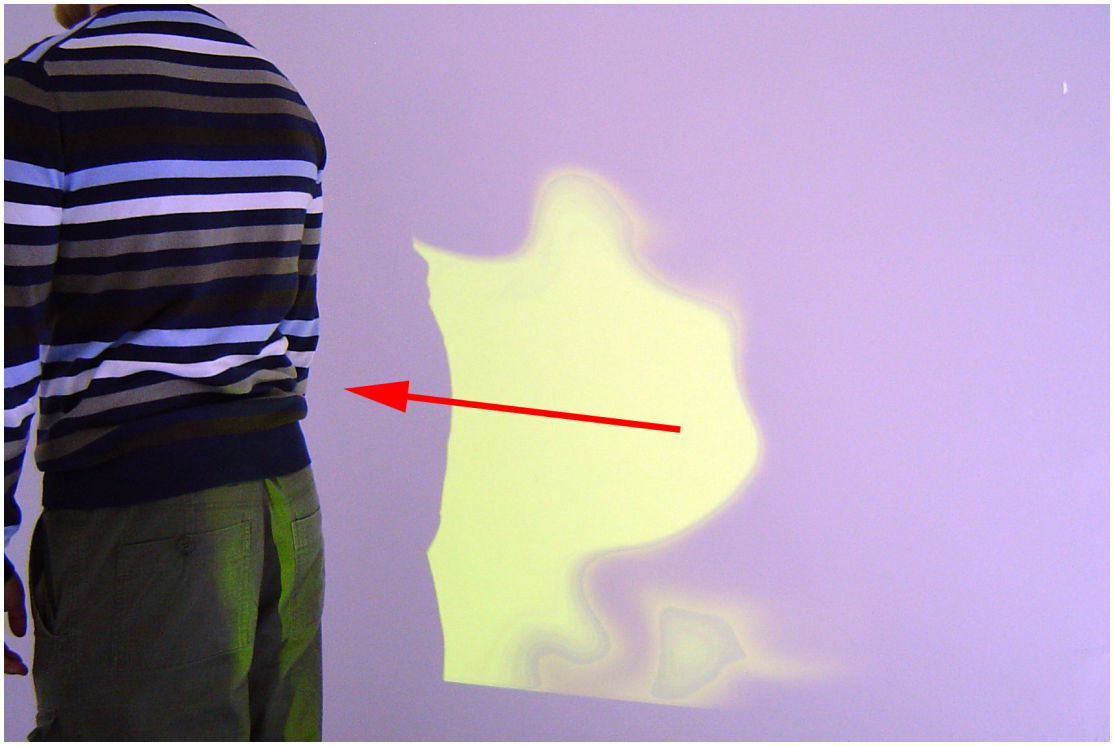


Figure 5.6: This image shows a person walking “through” a flow field interacting with the dye particles. As can be seen the initially round particle splat was advected to the left following the user. The red arrow indicates the motion of the user.

5.2 Direct Volume Rendering

For high-quality rendering of 3D flow, and for 3D volumetric data sets in general, there exists no native GPU primitive like the quadrilateral in 2D. Therefore, specialized render functionality has to be implemented for 3D volume rendering. In this thesis we propose a fast and high-quality render system for volumetric data. But before we describe our approach, we will briefly recapture the history of volume rendering.

In general, real-time methods to directly convey the information contents of large volumetric scalar fields are still a challenge to the computer graphics community. Based on the observation that the capability of a single general-purpose CPU is not sufficient to achieve interactivity or even real-time performance for large data sets in general, considerable effort has been spent on the development of acceleration techniques for CPU-based volume rendering and on the design and exploitation of dedicated graphics hardware.

Among others, one research direction has lead to volume rendering techniques that exploit hardware-assisted texture mapping. Fundamentally, these systems re-sample volume data, represented as a stack of 2D textures or as a 3D texture, onto a sampling surface or so-called proxy geometry. The most common surface is a plane that can be aligned with the data, aligned orthogonal to the viewing direction,

or aligned in other configurations (such as spherical shells). The ability to leverage the embedded trilinear interpolation hardware is at the core of this acceleration technique.

This capability was first described by Cullip and Neumann [25]. They discussed the necessary sampling schemes as well as axis-aligned and viewpoint-aligned sampling planes. Further development of this idea, as well as the extension to more advanced medical imaging, was described by Cabral et al. [17]. They demonstrated that both interactive volume reconstruction and interactive volume rendering was possible with hardware providing 3D texture acceleration. Today, texture-based approaches have positioned themselves as efficient tools for the direct rendering of volumetric scalar fields on graphics workstations or even consumer class hardware [112, 77, 90, 29, 40, 59]. Furthermore, it is commonly accepted that for reasonably sized data sets appropriate quality at interactive rates can be achieved by means of these techniques.

Nevertheless, despite the benefits of texture-based volume rendering, one important drawback has not been addressed sufficiently throughout the ongoing discussion. For a significant number of fragments that do not contribute to the final image, texture fetch operations, numerical operations, i.e., lighting calculations, and per-pixel blending operations are performed. It is thus important to integrate standard acceleration techniques for volume rendering, like early ray termination and empty-space skipping [74, 26, 98, 118, 34], into texture-based approaches. This is the key contribution of this section, and we will demonstrate the effectiveness of the proposed algorithms for effects rendering $2\frac{1}{2}$D simulation results from the previous chapter as well as scientific visualization on a variety of real-world data sets.

The essential mechanism that allows us to effectively integrate early ray termination and empty-space skipping into the raycasting process is the early z-test. This test is applied before the pixel shader program is executed for a particular fragment; thus, the early z-test can be used to avoid a pixel shader execution enabling a fragment processor to process the next incoming fragment. The early z-test, however, is only enabled if all other per-fragment tests are disabled and the fragments' depth value is not modified in the pixel shader. The effect of the early z-test can be demonstrated quite easily by simply drawing an opaque quadrilateral occluding parts of a volume prior to rendering the volume. If the depth test is enabled and depth values have been written in the first pass, a considerable speed up can be perceived that is directly proportional to the area of the occluded parts of the volume.

In the following we first review texture-based volume rendering techniques and we put emphasis on some intrinsic drawbacks and limitations of current implementations. Next, we present volume raycasting on programmable graphics hardware as an alternative to traditional object-order approaches. Then, we outline new techniques to integrate early ray termination and empty-space skipping into the ray traversal process.

5.2.1 3D Texture-Based Volume Rendering

Volume rendering via a 3D texture or revolved 2D textures is usually performed by slicing the texture block in back-to-front order with planes oriented parallel to the view plane, see Figure 5.7 for an example.

For each fragment the texture is sampled by trilinear interpolation, and the resulting color sample is blended with the pixel color in the color buffer. If slicing is performed in front-to-back order the blending

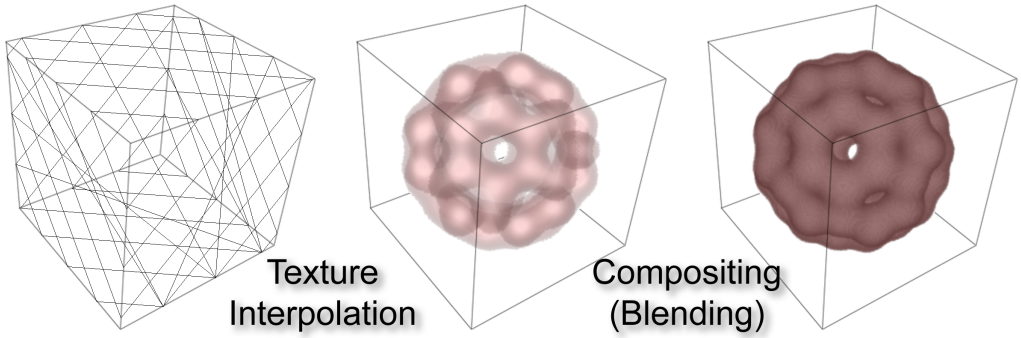


Figure 5.7: Volume rendering via texture slicing.

equation changes from

$$C_{\text{dst}} = (1 - \alpha_{\text{src}})C_{\text{dst}} + \alpha_{\text{src}}C_{\text{src}} \qquad \text{(over, back-to-front)}$$

to

$$\begin{aligned} C_{\text{dst}} &= C_{\text{dst}} + (1 - \alpha_{\text{dst}})\alpha_{\text{src}}C_{\text{src}} \qquad \text{(under, front-to-back)} \\ \alpha_{\text{dst}} &= \alpha_{\text{dst}} + (1 - \alpha_{\text{dst}})\alpha_{\text{src}} \end{aligned}$$

Here, $C_{\text{dst}}, \alpha_{\text{dst}}$ and $C_{\text{src}}, \alpha_{\text{src}}$ are the color and opacity values of the color buffer and the incoming fragment, respectively.

In front-to-back order an additional α-buffer needs to be acquired to store the accumulated opacity. Pre-multiplication of source color with source alpha is realized in a pixel shader as proposed later in this thesis.

As a direct implication of the selected proxy geometry in texture-based volume rendering, from pixel to pixel the data is sampled at varying rates. In Figure 5.8 the sampling patterns are shown for 2D and 3D texture-based volume rendering. As can be seen, the sampling on spherical shells around the view point mimics at best the kind of sampling that is performed in volume raycasting. Here, if no acceleration technique is employed, the rays of sight are traversed with a constant step size. Spherical clip geometries, however, have turned out to be rather impractical due to the huge number of geometry that has to be generated, transferred and finally rendered on the GPU.

Probably the most obstructive limitation of texture-based volume rendering, however, is the huge amount of fragment and pixel operations, like texture access, lighting calculation, and blending that are performed, but which do not contribute to the final image. In order to verify our hypothesis, typical volumetric effects and scientific volume data sets are shown in Figure 5.9. Most commonly in volume rendering applications the focus is on emphasizing boundary regions or selected material values. Usually

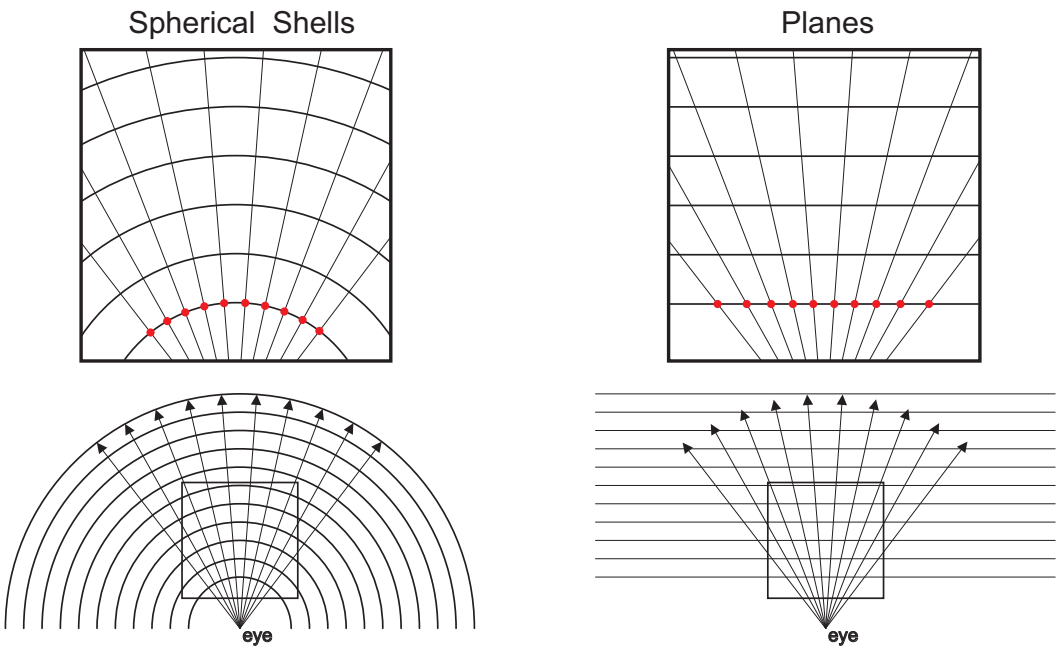


Figure 5.8: Different proxy geometries used in texture-based volume rendering. Note the non-uniform sampling (red) when rendering planes instead of spherical shells.

this is done by locally adjusting the color and opacity in order to highlight these structures with respect to others. As a matter of fact, due to the inherent occlusion effects, the majority of fragments that are generated during volume rendering never contribute to the image. In addition, because non-relevant structures are often suppressed by setting their opacity to zero, many fragments contributing to empty space are generated and have to be processed. For instance, in the images shown in Figure 5.9 only between 0.2% and 4% of all generated fragments contribute to the final image.

If the volume rendering technique should also provide realistic simulation of lighting effects, the waste of per-fragment operations is even more dramatically. Lighting effects are usually simulated by means of a gradient texture, which is comprised of the material gradients and the scalar material values in the RGB and α color components, respectively. The illumination model is evaluated on a per-fragment basis in the shader program. Consequently, even for non-visible fragments the gradient texture has to be sampled and numerical computations have to be performed.

5.2.2 Volume Raycasting on Graphics Hardware

The key to volume raycasting on the GPU is to find an effective stream model that allows one to continuously feed multiple, data-parallel fragment units. In addition, the number of fragments to be processed and the number of operations to be performed for each fragment should be minimized.

The proposed algorithm is a multi-pass approach. For each fragment, it casts rays of sight through the volume until an opacity threshold is reached or a selected iso-value is hit. In the latter case, the

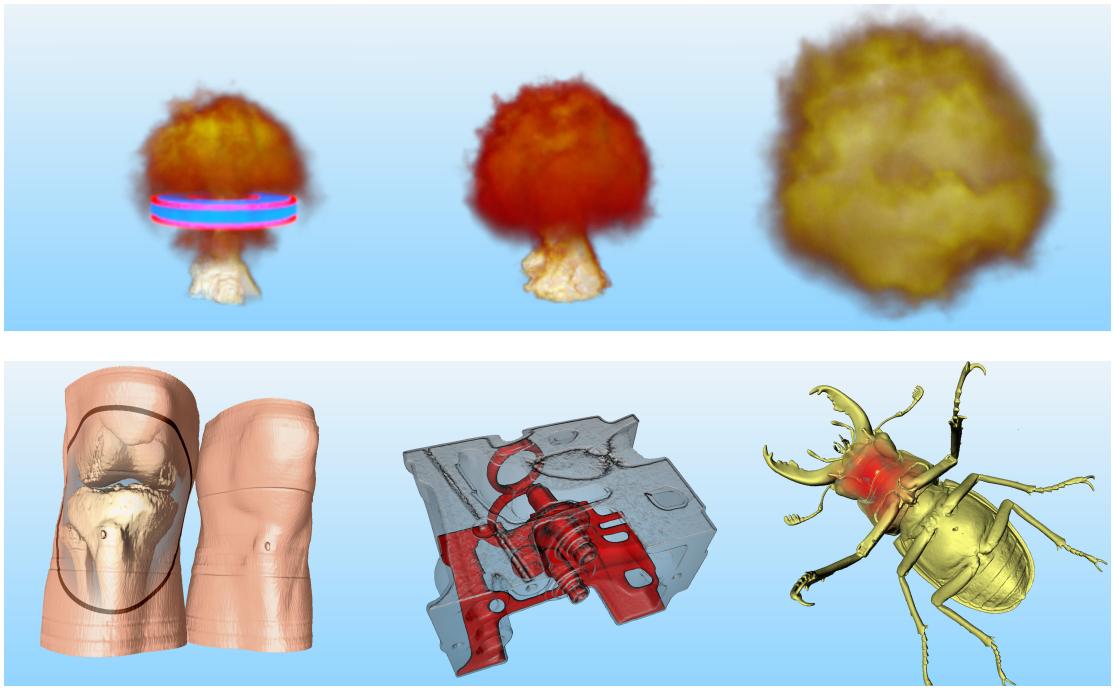


Figure 5.9: Volume rendering examples. The upper images show $2\frac{1}{2}$D simulation extrusions. The images in the lower row show typical example of volumes used in scientific visualization [67].

coordinates in local texture space of the intersection points with the surface are written to a 2D texture. This texture is used in a final pass to restrict upcoming computations, i.e., access to a gradient texture and shading computations, to these points.

Prior to ray traversal, for each pixel the direction in local texture coordinates of the ray through that pixel is computed. As this direction is stored in two 2D textures, it can be retrieved directly in all upcoming rendering passes. Ray traversal is done in a fixed number of rendering passes, each performing a constant number of steps along the rays. The render target in each pass is a 2D texture that is accessed in consecutive passes to access accumulated color and opacity values.

Between any two main passes, an additional pass is performed that simply tests whether the current opacity value has already exceeded a specified threshold or an iso-surface is hit. Depending on the result of this test, the pixel shader modifies the z-value. If the test succeeds the z-value is set to the maximum value otherwise it is set to zero. As a consequence, if the z-test is set to GREATER, all consecutive main passes will be discarded due to the early z-test.

In essence, the following steps are performed (the depth test is always set to GREATER):

- **Pass** 1 *(Entry point determination)*: The front faces of the volume bounding box are rendered to a 2D texture. 3D texture coordinates of each vertex are issued as per-vertex color, COL, (see left of Figure 5.10). The result is a 2D texture (TMP) having the same resolution as the current viewport. The color components in the texture correspond to the first intersection point between the rays of

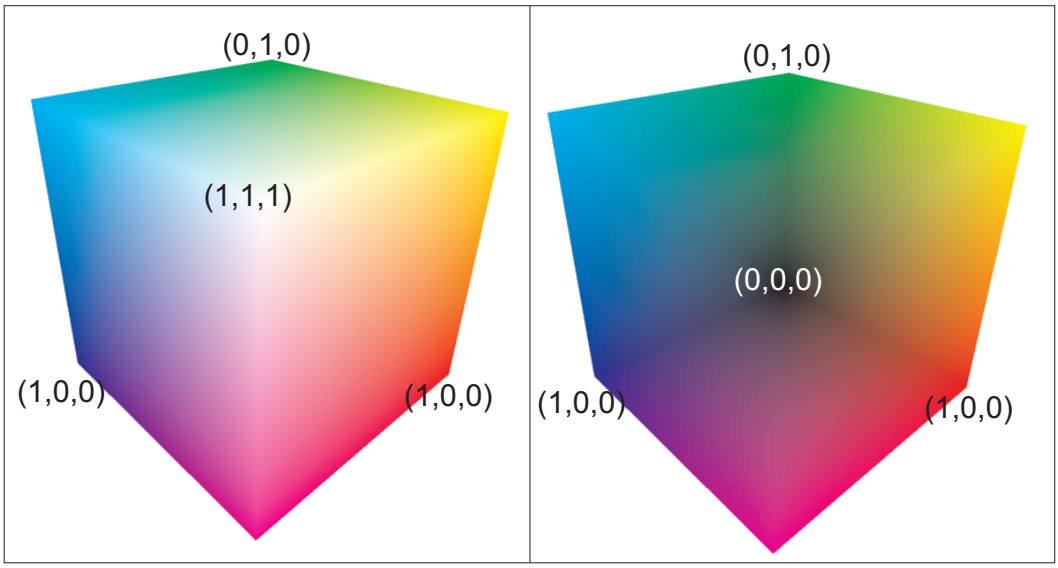


Figure 5.10: Rendering front faces (left) and back faces (right) of the volume bounding box in order to generate ray directions and texture coordinates of the first intersection points.

sight and the volume. Coordinates of the intersection points are given with respect to texture space.

- **Pass** 2 *(Ray direction determination)*: The same steps as in Pass 1 are performed, but now back faces of the volume bounding box are rendered to a 2D texture (DIR) (see right of Figure 5.10). In this pass, a fragment shader is issued that fetches for each fragment the respective value from TMP and computes the normalize ray direction as *normalize*(COL - TMP) to be rendered into the color components of the render target. In addition, the length of the non-normalized direction is rendered to the opacity component. The 2D floating point render target DIR now holds for every fragment the normalized ray direction in local texture coordinates and the length of each ray passing through the volume.

- **Main passes** 3 **to** *N (Ray traversal and early ray termination)*: In each pass, M steps along the rays are performed, and rendering is directed to a 2D texture, RES, that can be accessed in the consecutive passes. The front faces of the volume bounding box are rendered. Multiple parameters are issued at the vertices of these faces: Texture coordinate 1 gets assigned the normalized device coordinate (x,y) of each vertex. It is used to index textures DIR and RES. 3D texture coordinates of each vertex are issued as per-vertex color (C). Thus, the parametric ray equation $r(t) = C + t \cdot DIR[x][y]$ in local texture coordinates is directly available by performing one texture lookup. In constant color 1 (C1) the step size Δ to be performed along the rays of sight is specified. In constant color 2 (C2) the product of Δ and the number of steps already performed ($\Delta \cdot M \cdot (N-3)$) is issued. By initially setting t to C2, ray traversal now involves incrementing t by C1 and fetching the respective value from the 3D texture at $r(t)$. This is done M times, at each sample blending the

current contribution with the contribution that has been accumulated up to this position. Finally, the result is blended with RES[x][y] and it is written back to the 2D render target RES. If by means of the opacity value read from DIR, which stores the length of each ray within the volume, it turns out that the current ray has already left the volume, an opacity value of 1 is written to RES.

- **Intermediate passes** 3 **to** *N (Stopping criterion)*: In an intermediate pass, the front faces are rendered again, and a shader is issued that accesses the current opacity value and compares it with a constant threshold T (Note that in case the ray has left the volume, opacity is always larger than T). Essentially, it performs the following conditional statement:

```
1  if (RES[x,y] > T)
2    THEN z = MAX;
3    ELSE z = 0;
```

If the ray has to be terminated, the respective value in the z-buffer is set to the maximum value and the color buffer is kept by drawing zero color and opacity. Otherwise, the shader has no effect.

Iso-Surface Raycasting

If only a single illuminated iso-surface should be rendered by means of GPU-based raycasting, the shader setup becomes even more simple. The first two rendering passes are performed as described above. In passes 3 to N, however, considerably less arithmetic has to be performed. Within each pass, we also perform M steps along the ray, but now we perform the traversal in back-to-front order. At each sample we fetch the respective value from the 3D texture, and we test the scalar value whether it is larger or equal to the selected threshold. If this condition is true, we store the current position along the ray, $r(t)$, in a temporary variable. By going back-to-front within one main pass, only the intersection point closest to the viewpoint is kept. At the very end of the shader, the temporary variable is checked. If it is not equal to an initial value, the content is used to look up a 3D gradient texture and to perform surface lighting. In this case, the final color value that is rendered to the texture render target contains the illumination of the surface point, and an opacity value equal to one to guarantee for early ray termination in consecutive passes. The intermediate pass is the same as proposed above.

Empty-Space Skipping

Apart from early ray termination, one important issue needs to be addressed to speed up volumetric raycasting. Empty-space skipping essentially relies on an additional data structure, which encodes empty regions in the data. For instance, an octree hierarchy can be employed, which stores at every inner node statistical information about child nodes, e.g., min/max-bounds for the region that is covered by the node. If this information is available to the raycaster, it can effectively increase the sampling distance along the ray when entering empty regions.

In our current implementation, only one particular octree level is computed to speed up rendering. We always encode disjoint blocks of size 8^3 within the original data set, and for every block we store the minimum and the maximum scalar value contained in this block. The data is stored in a 3D RGB texture map with $\frac{1}{8}$ the size of the original volume in every dimension. The minimum and maximum values are encoded in the R- and G-components, respectively. In addition to this texture we generate a 2D texture, CT that indicates for every pair (min/max) whether there is at least one non-zero color component in the range from min to max after shading via a color table has been performed. This texture is updated on the CPU and loaded on the graphics subsystem whenever the color table is modified.

In order to test for empty space we extend the intermediate pass that is executed right after each of the passes 3 to N described above. Therefore, the front faces are rendered in the same way as described, but the ray is traversed with eight times the step size. The number of steps is decreased accordingly. Instead of the original data set, the coarse resolution copy is accessed at every sample point $r(t)$. The R- and G-components at every sample are used to index CT, and thus to test whether empty space is present or not. Whenever non-empty space is found at one of the sample points, the entire segment processed in this pass is considered to be non-empty. In this case, the z-value is set to zero and the color buffer is left unchanged. Otherwise, i.e., for empty space, the z-value is set to the maximum, thus forcing the next main shader pass to be skipped.

Altogether, the following conditional statement is now executed in the intermediate shader:

```
1  if ((RES[x,y] > T) or (EmptySpace == true))
2    THEN z = MAX;
3    ELSE z = 0;
```

Note that the z-value is reset to zero as soon as no empty space was found, thus re-enabling ray-traversal.

5.2.3 Performance Measurements

The proposed stream programming model for volume raycasting has several benefits over object-order projection methods. Besides equal sampling density along the rays of sight, the most important one is the reduction of the number of fragment operations to be executed if opaque structures or empty space are contained in the data. As the traversal procedure is split into multiple passes with at most M steps along the ray, at most M texture fetch operations have to be performed to no purpose. This happens if the first sample within a segment already saturates the opacity, generating $M - 1$ void samples, or if the first sample is already outside the volume.

Therefore, M should be rather small compared to the maximum number of samples that have to be performed for the longest ray through the volume. On the other hand, the number of rendering passes to be executed also depends on M. Assuming the longest ray through the volume performs K steps, then $\lceil K/M \rceil$ passes need to be issued. As with each pass some overhead comes along, i.e., rendering the front faces and accessing various textures, supposedly M should be as large as K. Then, however, all rays

shorter than the longest one have to compute the same number of sample points along the ray because a shader program cannot be discarded during execution.

For shaded iso-surfaces, the gradient needs to be reconstructed only once in the rendering passes 3 to N to illuminate the surface point. Note that, due to the fact that fragment shaders do not support conditional execution of expressions efficiently, gradient reconstruction and illumination is always computed even if no surface was found. The fact that this is only performed once every M steps, significantly accelerates the rendering of opaque iso-surfaces.

Moreover, the algorithm can effectively take advantage of texture cache coherence. A particular fragment traverses the ray along a certain distance thereby accessing the volume texture multiple times. If texture memory is organized in a block-wise layout, successive texture fetches are likely to access data already in texture cache. This is different from typical slicing approaches, where each slice has to be processed entirely before the next slice is going to be projected. Cache coherence between particular regions within successive slices, however, are destroyed by this approach. Our tests have shown that on GPUs that exhibit a strong view-direction/performance dependency the raycaster's sampling pattern reduces this dependency resulting in a more constant frame-rate when rotating the volume.

It is of course worth noting that for highly transparent and dense data sets, where each ray needs to be traversed until it exits the volume, no gain in performance can be expected. In this case, the intermediate pass to check for early ray termination and empty space introduces some overhead.

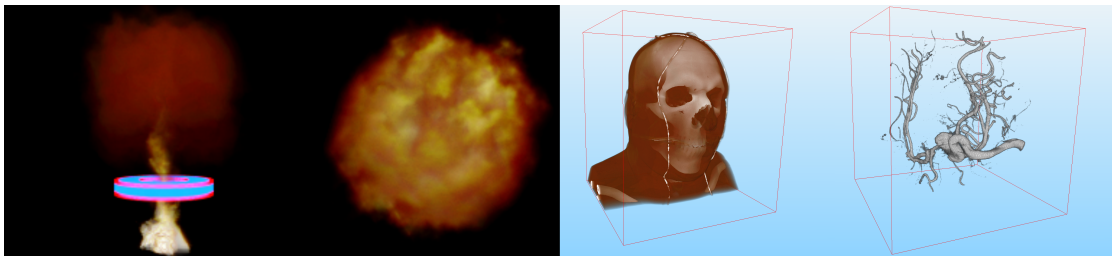


Figure 5.11: These images show the example data sets we have used to test the performance of our acceleration techniques. The simulation was executed on two revolved 128^2 slices. The visible human data set has size 512^3 and the aneurism is of size 256^3.

Let us now demonstrate the effectiveness of the proposed acceleration schemes for 3D texture-based raycasting. We use the corona and the explosion demo and stop the simulation to only measure the rendering performance. Additionally, to demonstrate the applicability to scientific volume visualization, two scientific data sets are shown, the Philips aneurism data set from the volren web-page [8] and the head data set from the visible human project [107] (see Figure 5.11). Rendering was directed to a 800×600 viewport. In the table below we give exact timings for our algorithm, which are compared to the times achieved by the slice-based volume rendering (SBVR) that uses planar proxy geometry (ref. Figure 5.8) as opposed to the spherical sampling used by the raycaster. Furthermore, we compare ourselves to a highly optimized SIMD software raycaster implementation. *RC* is the time consumed by the raycaster without any acceleration. *RC-*α and *RC-*β account for early ray termination and early ray termination combined with empty-space skipping, respectively.

	SBVR	RC	RC-α	RC-β
Corona	131	61	155	n/a
Explosion	131	61	160	n/a
VH-Head	10	6.5	14.4	27
Aneurism	32	14	14	46.6
Aneurism Software				8.74

Table 5.1: Timings (fps) for 3D texture-based volume rendering on an ATI X1900 XT GPU. For the simulation data sets no RC-β are given since we did not compute an empty space volume dynamically. In the last row we compare the timings of an highly optimized SIMD software raycaster as recently presented by Marmitt et al. [75] for the aneurism data set.

Obviously, with regard to performance, SBVR is clearly superior to volume raycasting without early ray termination and empty-space skipping. The reason is that in SBVR only those fragments have to be processed that are within the volume bounding box. On the other hand, the raycaster achieves superior image quality compared to SBVR (see Figure 5.12). In RC, depending on the choice of M many fragments outside the volume have to be processed. If early ray termination and empty-space skipping is enabled, however, we can see a performance gain of RC in typical real-world examples exhibiting opaque structures and empty regions. Unfortunately, the speed up is not as dramatic as we would have expected. This is due to the fact that we need to perform the intermediate shader pass to check for the stopping criterion and to replace z-value in order to exploit the early z-test. The intermediate pass itself, however, cannot be discarded because it modifies the z-values. Thus, empty-space detection is always performed, even if early ray termination is already in use. We could easily integrate empty-space detection into the main shader passes; however, we would need one additional flag that can be written from the shader to identify whether empty-space has been detected or not. For this purpose we can not use the alpha channel, because it already serves as a flag indicating early ray termination.

For semi-transparent volumetric data sets, where neither early ray termination nor empty-space skipping can be applied, the overhead to perform the intermediate pass to check the stopping criterion manifests itself in a loss in performance of about 30%. This, however, is always the price that has to be payed for the integration of acceleration techniques if the data set does not provide appropriate stopping cues. For the target data within the scope of this thesis we usually use a very hard transfer function that classifies most of the volume as empty. In non-empty places—where the explosions are located—only very few raycasting steps need to be performed since the effect volumes created by the simulation are usually very opaque (see Figure 5.13).

With direct volume rendering we have presented a powerful and versatile tool to visualize volumetric effects. However, direct volume rendering requires a scalar or color/opacity data set. To generate this scalar data from the $2\frac{1}{2}$D simulations dye advection in each 2D slice is performed. During volume rendering the revolution of these dye slices is computed as elaborated in Section 4.4. To avoid artifacts from rotational symmetry, texture coordinates are perturbed with an additional noise volume (for details see Section 4.4). While this already gives visually compelling results (see Figure 5.13) yet it is restricted

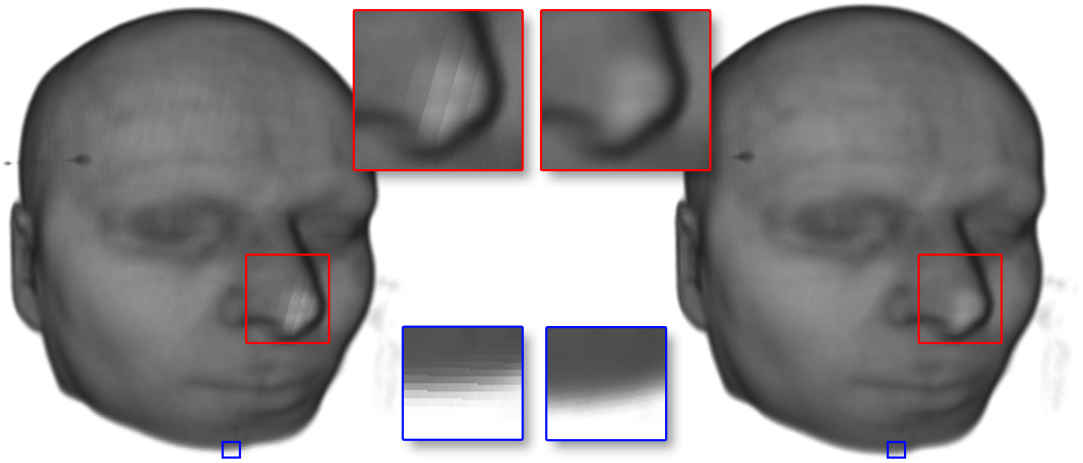


Figure 5.12: An MR data set visualized with SBVR (left) and the raycaster (right). Although the same step size was used, one can clearly see the superior image quality of the raycasted image. The red and blue squares magnify areas where the slice artifacts are particularly disturbing.

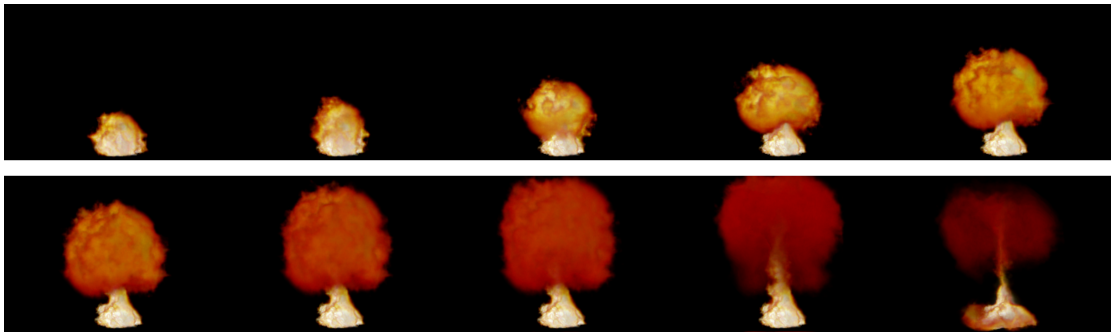


Figure 5.13: A sequence of images of a large scale explosion is shown. This demonstration is simulated and rendered at highly interactive frame rates of over 150 fps on our target architecture.

to rotational symmetric external influences, such as ring obstacles or external forces coming from all directions (see Figure 5.14).

For a 3D simulation we can perform the dye advection on a texture atlas. This advection is similar to the one performed in 2D, however we need to perform more texture fetches to do the trilinear interpolation by means of two interpolated bilinear fetches and we are dealing with a much larger domain. Thus, a rendering mode that operates directly on the vector field is desirable. In the next sections we will therefore explain how to render particles efficiently on the GPU to avoid traversing the entire domain for rendering.

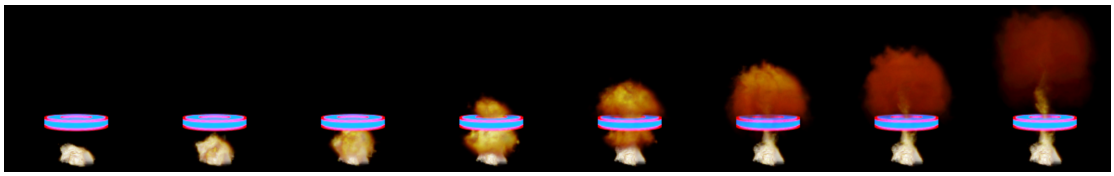


Figure 5.14: Rotational symmetric obstacles in the $2\frac{1}{2}$D Simulation.

5.3 Primitive Displacement

As we explained in Section 4.5 the sole advection of particles in a flow field is well suited for the GPU and maps perfectly to its SIMD processing model. However, the list of new particle positions that is output at the very end of the particle tracing algorithm is only a 2D texture not suitable for direct display. To render particles, the position information needs to be mapped onto a renderable primitive. This representation is fed into the pipeline as a vertex buffer, which is used to render particles in an upcoming pass. Therefore, we need some sort of displacement functionality on the GPU to avoid the time consuming download of the particle position texture to the CPU and the construction of a vertex buffer, which afterwards would have to be uploaded to the GPU again. But if we have the functionality to displace geometry directly on the GPU, the complete particle tracing and rendering can be performed entirely on the GPU avoiding time consuming bus transfers.

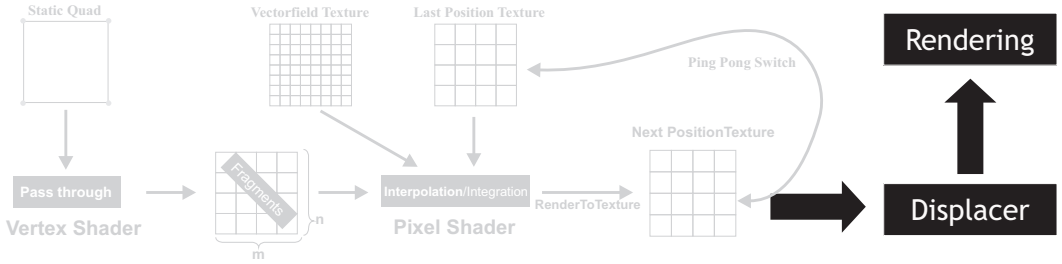


Figure 5.15: The entire particle tracing loop, with the advection subsystem feeding positions into the displacer, which in turn drives the particle rendering subsystem.

5.3.1 The Displacer Black Box

Today, three different ways to efficiently realize a vertex displacement on the GPU exist. These are the vertex texture fetch, the render-to-vertex array, and the PBO-VBO-Copy functionality. Since these abilities of GPUs play a very important role in the particle displacement, we will use the following section to explain the three possible displacement techniques in detail. The result of all these approaches is essentially the same: A stream of vertices with coordinates modified by the values that were stored in a texture map. Therefore, after we made ourselves familiar with the techniques themselves, we can abstract from the implementation and just use the algorithms as a black box calling it the *Displacer Black Box* (see Figure 5.15).

PBO-VBO-Copy

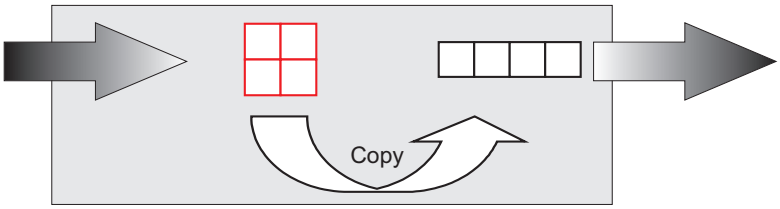


Figure 5.16: The PBO-VPO-Copy displacement black box

The oldest and least efficient way to convert the output of the GPUs pipeline into vertex information is the PBO-VBO-Copy (Figure 5.16). VBO stands for vertex buffer object and denotes a buffer that can be used as a source for vertex information. A PBO—a pixel buffer object—is a special extension of the VBO to allow a VBO to become the target of a copy operation. To perform vertex displacements using this mechanism, a floating point pixel buffer (PBuffer) or a framebuffer object (FBO) needs to be created and used as a texture render target to store the positional information. In addition to this a VBO needs to be created with the `glGenBuffersARB` call.

After the application has rendered the position into the texture, a `glReadPixels` call is issued, and the driver takes care that instead of reading the pixels back to the CPU the data is stored in the VBO. After the copy is complete, the VBO can be bound as vertex buffer via the `glBindBufferARB` call, and geometry can be rendered via `glDrawArrays`.

The advantage of this techniques is compatibility with older graphics cards: essentially any GPU with floating point texture support also supports this technique. On the downside this algorithm is only usable in OpenGL applications since no render target to vertex buffer functionality exists in DirectX and most importantly this approach requires a copy operation from the texture into the vertex buffer. This operation is in the best case executed directly in fast GPU memory. This copy operation is the reason why the PBO-VBO copy should be avoided if possible since it makes the algorithm inherently slower than the other approaches.

Vertex Texture Fetch (VTF)

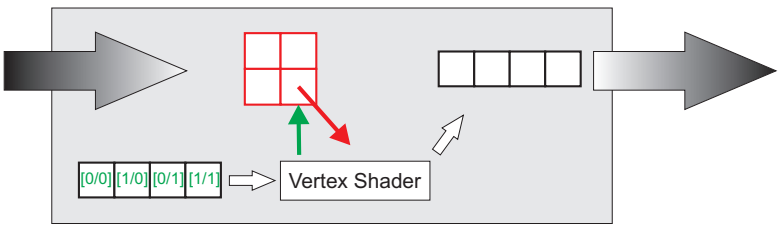


Figure 5.17: The vertex texture fetch displacement black box

The most flexible technique to displace vertices with data stored in a texture is the vertex texture fetch

(Figure 5.17). Hardware supporting the VTF allows the shader programmer to fetch from floating point textures in the vertex shader stage. Thus, to displace a vertex by a coordinate stored in a texture map, the programmer sends a static vertex buffer, which is stored in local GPU memory, into the pipeline. For every vertex in this buffer one or more texture fetches are executed and the final position of the vertex can be computed by arbitrary arithmetic operations using these texture values. This makes the VTF very versatile. On the downside it requires a Shader Model 3.0 capable hardware with VTF functionality. Currently only NVIDIA GeForce 6 series and newer NVIDIA cards support this feature. However, the VTF capability is required for DirectX 10 level hardware, thus broader support can be expected in the near future. Finally, the random access into the texture makes this algorithm potentially slower than the R2VB explained next, which enforces a sequential access through the texture.

Render To Vertex Buffer (R2VB)

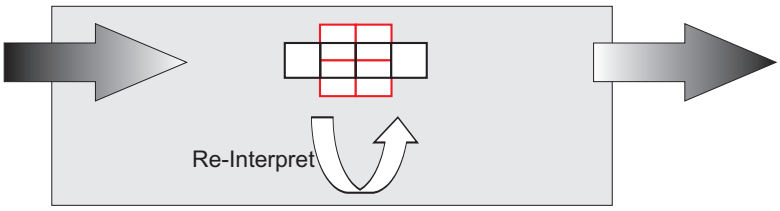


Figure 5.18: The render-to-vertex-array displacement black box

At present the fastest technique for vertex displacement is the render to vertex buffer (R2VB) approach (Figure 5.18). This technique should better be called “cast to vertex buffer” since this term better describes what happens on the GPU. Basically, a 2D texture is reinterpreted as a vertex array and fed back into the pipeline without a copy operation. The R2VB approach proceeds as follows: After the application program has rendered the positional data into a floating point texture, it is unbound from the render context and directly rebound as a vertex buffer. After this the application can use the vertex buffer as a regular source of input for the rendering pipeline. From the description of the algorithm it should be clear that this approach is both the simplest one to implement and conceptually the fastest. However, it is not as flexible as the vertex texture fetch and has the drawback that it is only supported by ATIs hardware and only under DirectX [92]. As with the VTF the R2VB functionality is part of the DirectX 10 functionality and will be supported by all vendors in the near future.

Displacement Summary

As we mentioned at the beginning all three techniques take as input a floating point texture storing the vertex positions and output a stream of displaced vertices into the primitive assembly stage. The only difference is the way this functionality is achieved internally by the GPU. Therefore, in the remainder of this thesis whenever vertex positions are to be displaced we simply use a displacer black box, which is to be replaced by one of the three algorithm that best matches the GPU and API requirements of choice. Table 5.2 once again summarizes the advantages and disadvantages of all three techniques.

Algorithm	Pro	Contra
PBO-VBO-Copy	well supported by hardware	only OpenGL, usually slowest solution
R2VB	fastest method supported by DX10 core feature set	only supported by ATI and in DirectX
VTF	supported by well defined API and DX10 core feature set	still slower than R2VB currently only supported by NVIDIA GPUs

Table 5.2: Comparison of the three GPU displacement techniques.

5.3.2 Particle Rendering

Equipped with basic functionality to displace geometry based on the GPU we can now use the displacer back box to render the results of the particle tracing algorithm from Section 4.5.1. The particle tracing algorithm in turn is fed with the result of either the 3D simulation or the $2\frac{1}{2}$D simulation. By putting together the simulation and the particle advection we get the data flow as depicted in Figure 5.19. So far this pipeline generates a stream of vertices displaced in the flow. The coordinates of each vertex corresponds to the positions of particles numerically integrated through the flow. These vertices can then be rendered using an appropriate representation. The only component that is left to implement is the final particle rendering step. This component is explained in the following section.

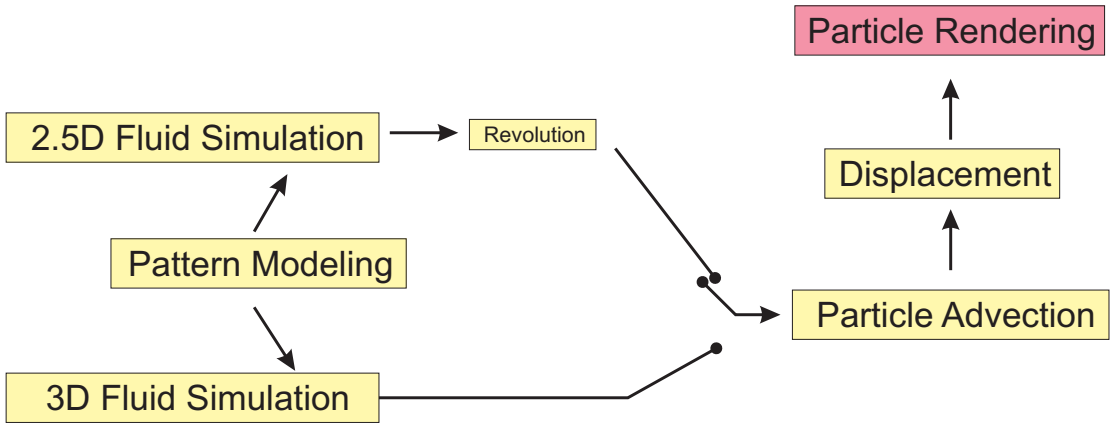


Figure 5.19: This image shows the principal interplay of some of the software components proposed so far. The combined components allow for the rendering of particle-based fluid effects such as the smoke in Figure 5.20.

Once particle positions have been used to generate a data stream, the vertex shader transforms these positions according to the viewing parameters. In general these vertices can be rendered as particles with different modes.

5.3.3 Points

Rendering of point primitives does not necessarily require any special fragment shader computations, and in particular no texture fetch has to be performed. By using this mode, millions of particles can

be displayed at interactive rates. Although every particle is displayed by a single pixel in screen space, the massive amount of primitives enables the simulation of real-world effects where tiny but numerous particles like small ash pieces, water vapor, or gas droplets are released into the flow. On our current target architecture, more than one million particles, stored in local video memory, can be sent through the vertex unit and rendered as colored point primitives at about 250 fps (see Figure 5.20).

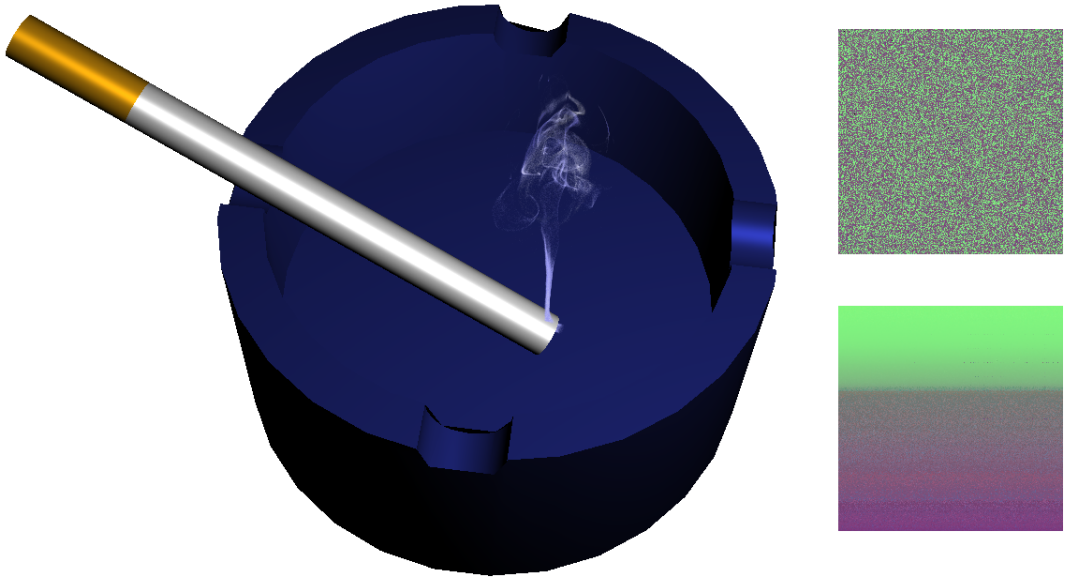


Figure 5.20: Particle positions are randomly stored in a 2D texture map (upper right quad), which is then used as input for the displacer black box. The lower right quad shows the particle positions sorted with respect to the viewer.

5.3.4 Point Sprites

Rendering the displaced vertices as simple pixels to screen creates believable impressions of homogenous media. However, in some cases a less homogenous structure within the medium is required to convey a certain effect. One example of such a case is the campfire demo (see Figure 5.21). In this case the chaotic high frequency structures within the fire and smoke are emphasized by textured points, so called point sprites. Conceptually, a point sprite is a textured quadrilateral centered at the points' screen space projection. Only a single vertex is transformed in the vertex units and the rasterizer generates a contiguous block of n by n fragments around this projection. 2D texture coordinates ranging from (0,0) to (1,1) are automatically generated and used to map a given texture image. Figure 5.21 shows particles that are rendered as point sprites with a small fire texture.

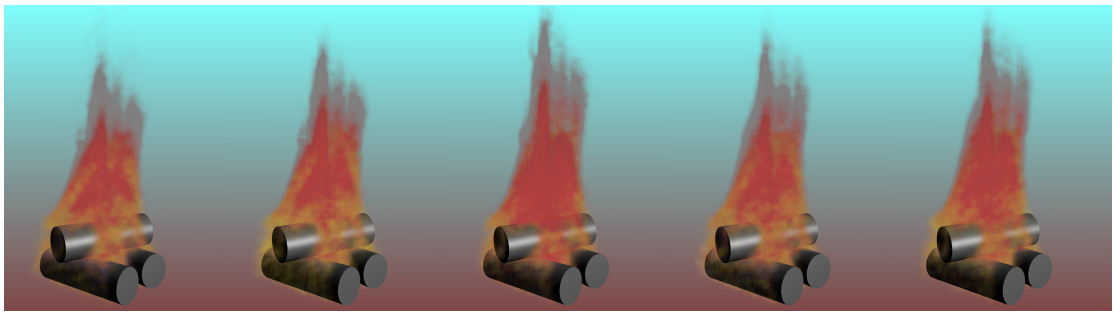


Figure 5.21: The images show a sequence of screen shots from the campfire demo, giving an impression of the evolving fire. In this demo large textured point sprites were used to render the fire and smoke.

5.3.5 Oriented Point Sprites

For a scientist observing simulated or measured flow fields and for the effect designer an in-depth analysis of the flow characteristics can prove helpful in understanding the reasons of certain effects. In the scope of effect rendering, non photo-realistic render modes are of particular interest during the flow-design phase. Simple point primitives, for instance, can not easily reveal flow direction, in particular not in still images, but even in an animation it is interesting to observe that oriented iconic particle representations like arrows, vector glyphs or ellipsoids provide a much more effective means for showing flow direction. Such geometric representations, however, put the burden of the visualization almost entirely on the geometry subsystem hence limiting the number of particles that can be rendered.

While point sprites can effectively render rotationally symmetric particle primitives, they produce incorrect results if used to display arbitrarily shaped geometry. This is mainly due to the loss of degrees of freedom if object transformations are performed after the projection into screen space. To overcome this drawback, we employ a texture atlas similar to the one proposed by Guthe et al. [39], but we use a parametrization that is more suitable for a GPU implementation.

The texture atlas contains a 2D array of different views of the 3D particle primitive. Views are parameterized with respect to scaling factor and rotation angle around the y-axis. The parameter domain ranges from 0 to 1 and from 0 to π for scaling factor and rotation angle, respectively (see Figure 5.22) . Here, we assume the particle primitive is aligned with the x-axis, the local direction of the vector field. To get all rotations from 0 to 2π we use the texture wrap mode *mirror*.

For rendering point sprites, a fragment shader transforms the uniform texture coordinates (u, v), which are generated for every fragment covered by this sprite, in such a way as to map into the appropriate atlas sub-image. Therefore, the magnitude of the local velocity vector is used as v-offset and the arc sine of the z-component of the normalized vector is used as u-offset. To rotate the selected sub-image around the z-axis, we built a rotation matrix $(x, -y)^T$, $(y, x)^T$, where x and y are the first components of the normalized velocity vector. As all four parameters—texture coordinate offsets and velocity components—are constant for each sprite, these values can be computed in the vertex shader and passed to the fragment shader as a parameter. Since this approach requires the vertex shader to access the flow field, it can only

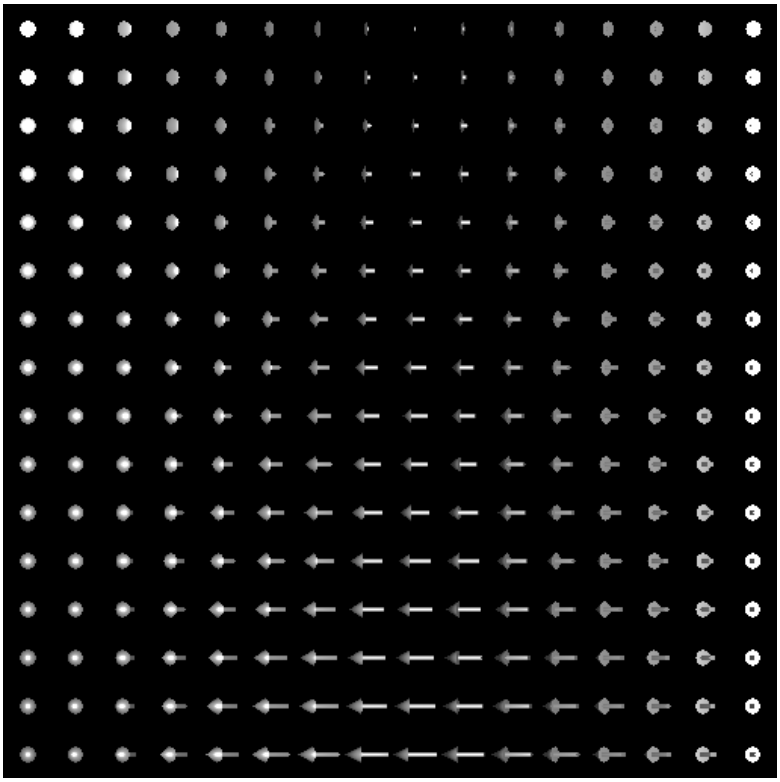


Figure 5.22: Low resolution version of the texture atlas, the scale of the model changes from bottom to top while from left to right different rotations are applied.

be realized using Shader 3.0. Another alternative is to create an additional particle container, into which these parameters are written by a fragment shader. Then, this container can be accessed by all fragments that are covered by a point sprite.

By using this method, the virtual geometry can be rotated around two axes and scaled correctly before the rotation takes place. Hence, primitives can point into any spatial direction while the overhead that is introduced by additional fragment computations is negligible.

5.3.6 GPU-Based Depth Sorting

Rendering semi-transparent objects—such as the point sprites—using blending modes other than additive blending (i.e., using the under or over operator [84]) requires the objects to be sorted in strictly monotonic front-to-back or back-to-front order with respect to the viewer (see Figure 5.23). Since this ordering is viewer dependent and since the particle motion in a vector field can be arbitrarily complex, it is impossible to sustain such a strict order without reordering the particles after every advection step. To avoid the transfer of the particle positions to the CPU for sorting and the upload of them back to the GPU for rendering, a GPU-based sorting network has been integrated into our particle rendering framework.

The sorting routine accounts for the architecture of today's graphics processors. Recent GPUs can

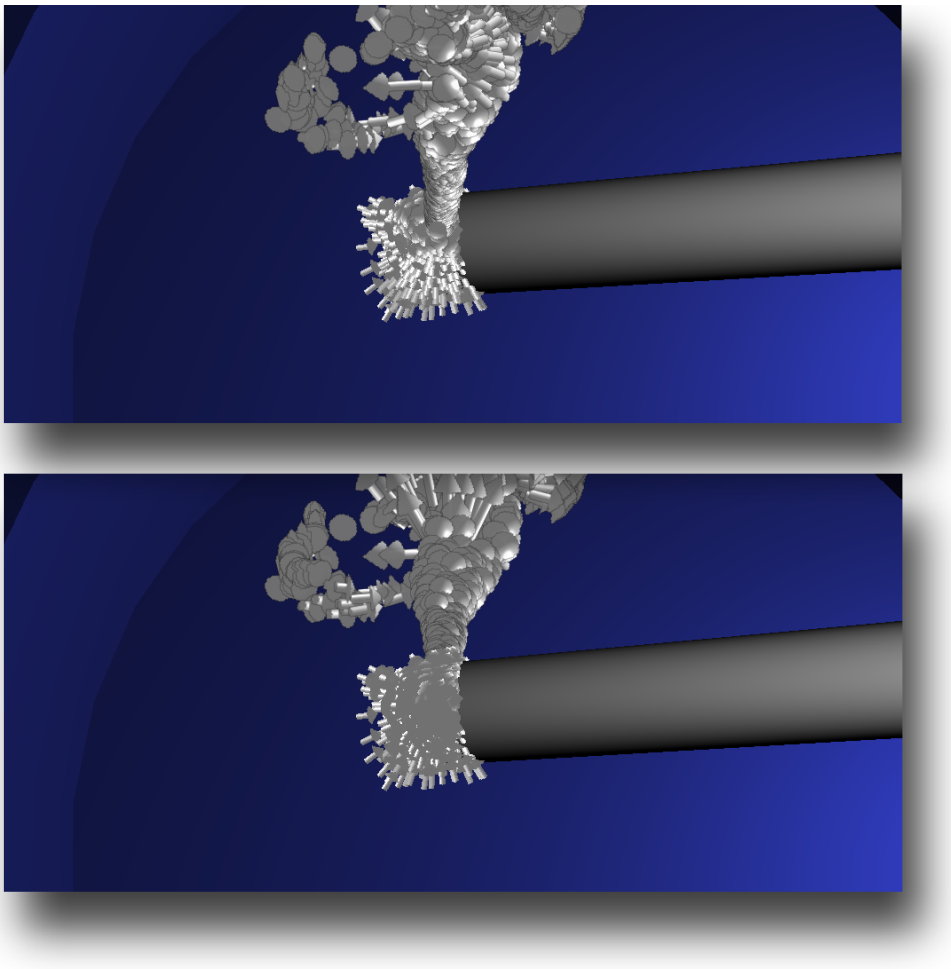


Figure 5.23: Flow direction analysis in the cigar demo. First, particles are rendered in correct visibility order. Next, particles are rendered in the order of their incarnation.

be thought of as SIMD computers, in which a number of processing units simultaneously execute the same instruction on their own data. Considerable effort has been spent on the design of sorting algorithms amenable to the data parallel nature of such architectures. Bitonic merge sort [9] is one of these algorithms. Compared to other sorting algorithms like Quicksort or Heapsort, it is well suited for such architectures because its sequence of operations is fixed and not dependent on the data to be sorted.

Purcell et al. [86], Govindaraju et al. [36], and Kipfer and Westermann [58] propose GPU implementations of the bitonic merge sort. The latter approach minimizes both the number of instructions and texture operations to be executed. On the GPU, a texture is built that contains for every particle a floating point sorting key—the distance to the viewer in the current scenario—as well as an integer floating point identifier—a reference into the appropriate position in the current particle container. Both values are stored in the R and G color components, respectively. Since the graphics pipeline as imple-

mented on recent cards is highly optimized for the processing of RGBA samples, two consecutive entries in each row—including sorting key and identifier—are packed into one single RGBA texture sample. Thus, coherence between adjacent entries with respect to memory access and arithmetic operations can be exploited.

Table 5.3 compares the GPU bitonic merge sort with an optimized data-dependent sorting routine of the C++ STL. Both algorithms were run on key/index pairs of equal bit width including the final reorder pass to exchange particle positions according to index permutations. As can be seen, the GPU solution has the potential to outperform the CPU solution or at least achieve similar performance. Sorting can now be relocated freely between the central processor and the graphics card without performance penalty.

sorter	#keys	megakeys/sec
Bitonic merge sort	256^2	7.2
ATI X800 XT	512^2	6.4
	1024^2	5.1
`std::sort`	256^2	5.4
P4 3.0 GHz	512^2	5.4
	1024^2	5.0

Table 5.3: Sorting Performance

Since sorting becomes the major performance bottleneck in the particle engine, alternative strategies have to be considered. One alternative is to lay out a full sort of particles over multiple rendering passes. Therefore, a sorting routine is required that yields "smoother" intermediate results than the bitonic merge sort. The odd-even merge sort is such an algorithm, and it has been shown to be well suited for this purpose [60]. It has the same number of stages and therefore the same asymptotic complexity as the bitonic merge sort. Moreover we can utilize similar coding optimizations as for the bitonic merge sort. In particular for the rendering of transparent point sprites, the odd-even merge sort gives visually pleasant results even in case the particles set is incompletely sorted. It allows us to spend as much time of one frame as we want for sorting, thus keeping the overall simulation time step within a fixed time limit.

In this scenario, another advantage of GPU sorting becomes apparent. The layout of sorting steps over multiple frames on the CPU still requires the entire particle set to be downloaded and uploaded from and to the GPU. Due to bandwidth limitations no more than 5 fps for a million particles can be achieved. Using GPU sorting, on the other hand, we can exactly determine the number of sorting steps per simulation pass until sorting becomes the performance bottleneck.

In our particle system, both sorting strategies are integrated and can be activated as additional rendering passes subsequent to the advection step, which can be switched on or off. A particular shader performs the sort and reorganizes particle positions and attributes in the current container accordingly. None of the other stages in the particle system are affected by the sort.

5.3.7 Grid Displacement

A different avenue of applications for the displacer black box is the displacement of entire triangle meshes. In the scope of this thesis this is of particularly interest for the 2D wave equation simulation. Instead of using the simulation texture that stores the height values to directly texture a quadrilateral, the texture can also be used to displace a triangle grid (see Figure 5.24).

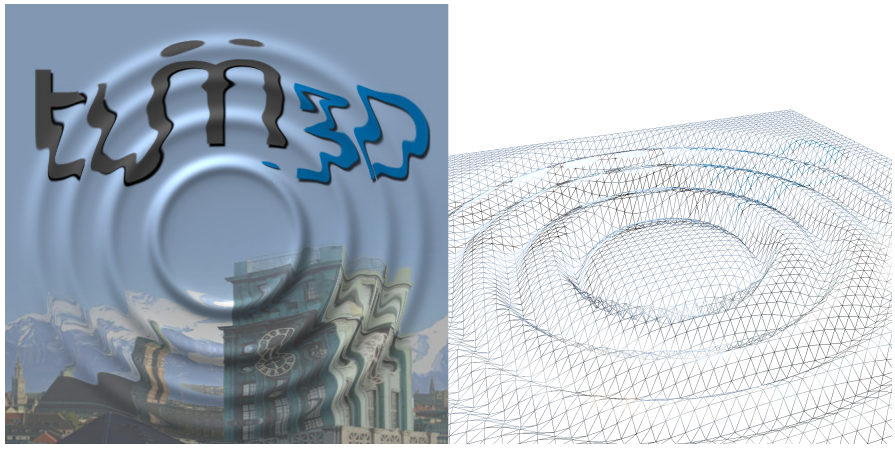
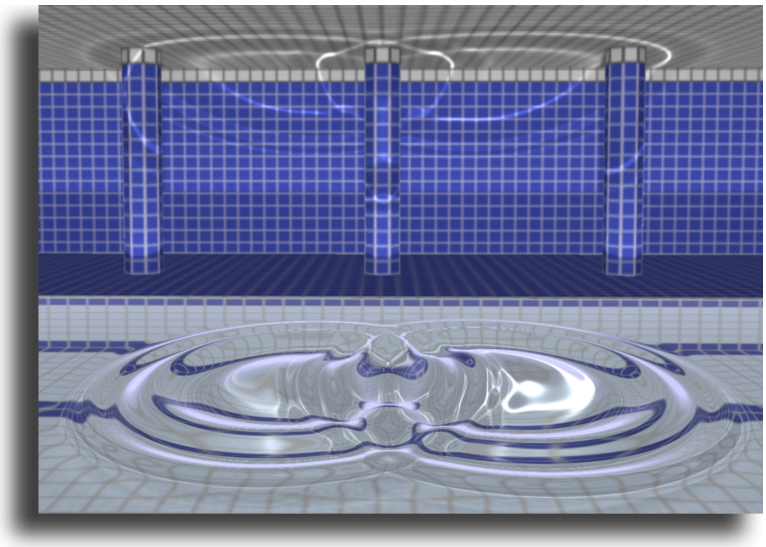


Figure 5.24: Rendering the height field of the shallow water equation in different modes. The left image shows the water surface with the refraction shader. In the right image the height field was used to displace a triangle grid.



Chapter 6

Effects

In the previous chapters we have presented a variety of ways to interactively simulate and render fluid effects. In this chapter we will demonstrate how these effects can be integrated into an interactive environment.

To successfully convey the illusion of real world phenomena in computer games or virtual environments, it is not sufficient to focus on a stand-alone effect. It is rather necessary to faithfully capture the scene/effect interplay. In this context, indirect light interactions between the scene and the fluid effect is of particular interest. An example is an animated water surface causing refractions, reflections, and caustics on the scene. However, due to the global nature of such phenomena they are difficult to simulate, traditionally prohibiting their use in real-time environments. To overcome the performance restrictions of traditional CPU-based techniques we propose an algorithm that makes use of the GPU to dramatically speed up the computation. We adapt the photon mapping process [56] to GPUs to allow its use in real-time environments. Our implementation focuses on the caustic light effect; thus, demonstrating the applicability for the integration of the water surface PDE into computer-generated environments. At this point it is important to note that the algorithm presented in the following is not limited to caustics but can also be used to simulate any global illumination effect that can be handled with traditional photon-maps.

Furthermore, at the end of this chapter we describe the extension of the algorithm to trace view rays to allow for the efficient and correct simulation of refractions on dynamic and arbitrarily complex geometries (see Figure 6.1).

Figure 6.1: GPU photon tracing can be used to simulate a variety of different light effects at interactive frame rates.

6.1 Caustics

Caustics are a light phenomenon that is caused by converging light, and they can appear whenever light impinges upon reflecting or transmitting material. A caustic is observed as an increase of brightness or radiance due to many light paths hitting a surface at the same position. These paths may have been reflected or refracted one or several times before impinging upon the receiver. In this work, we restrict the discussion to specular-to-diffuse light transport where reflected or refracted light hits a diffuse receiver causing light rays to terminate. In Figure 6.1, a number of different caustics and refractions are illustrated.

As the receiver, in general, does not know from which directions the light converges towards it, caustics cannot easily be simulated using conventional ray-tracing or scanline algorithms. As these methods backtrace the light paths from the eye point, all possible directions the light could arrive from have to be sampled at a caustics receiver. While it is impossible in general to directly render caustic effects using single pass scanline algorithms, indirect lighting effects involving specular reflections can be simulated via Monte Carlo ray-tracing [57, 108]. As these techniques come at the expense of tracing many rays to sample the incoming energy at every receiver point, only parallel implementations [109, 83] have the potential to run at interactive rates.

Instead of sampling the incoming energy for every receiver during the rendering pass, Arvo [6] proposed a two pass method that first creates an illumination map containing the indirect illumination received by an object and then samples this map in the final rendering pass. The method was named backward ray-tracing because it creates the illumination map by tracing rays from the light source instead of the eye point as in conventional ray-tracing. A different strategy to create the illumination map based on the projection of area splats was later proposed by Collins [18]. Inspired by the work of Heckbert and Hanrahan [46] on beam tracing, a variation of backward ray-tracing that creates caustics polygons by projecting transmitted light beams onto diffuse receivers was suggested by Watt [111]. A similar

approach has been utilized by Nishita and Nakamae [80] for the rendering of underwater caustics.

Following the early work on two-pass ray-tracing, Jensen and Christensen [56] introduced the more general concept of photon maps. The key idea is to first emit energy samples into surface-aligned maps via backward ray-tracing and then reconstructing radiance estimates from these maps in the final rendering pass. It is in particular due to the computational cost for building accurate photon maps via ray-tracing that even GPU implementations [88, 87] cannot achieve interactive rates for dynamic scenery of reasonable size. Only highly parallel implementations on multi-processor architectures [38, 117] have shown this potential so far.

As caustics—in particular if caused by dynamic objects, such as a water surface—are a light phenomenon that adds increasing realism and diversity to any 3D scene and rendering, much effort has been spent recently on developing techniques that can be used in interactive applications like computer games or virtual reality environments. In the following we will discuss the most fundamental concepts behind them.

6.1.1 GPU Caustics

Before summarizing previous work on interactive caustics rendering we will first briefly describe how to employ functionality of GPUs to efficiently construct illumination maps. This preliminary discussion is restricted to planar surfaces that receive energy via one single specular indirection. As refractions at a water surface above planar grounds satisfy these assumptions, this particular example is used in the following. The purpose of this discussion is to introduce some of the basic GPU concepts used to render caustics at interactive frame rates and to demonstrate the state of the art in this field. Later in this chapter we will present efficient techniques that are far less limited in the kind of transmitting objects and receivers they can render.

With the functionality presented in Section 5.3.1 we introduced a general technique to displace a vertex stream using texture values as positions. With this functionality caustics can be rendered into an illumination map in two passes. In the first pass, a water surface with color-coded normals is rendered from the light source position. This normal map can be dynamically generated by our shallow water surface PDE solver. Next, a simple pixel shader calculates for each fragment the refracted line of sight. If the equation of the planar receiver is known the intersection points between these lines and the receiver can be computed analytically, too. Assuming the mapping of the illumination map to the receiver is known, the intersection points can be transformed into the local parameter space before they are output to a vertex array. In the second pass, this array is rendered as a point set into an illumination texture map, where contributions to the same texel are summed by accumulative blending [102]. Figure 6.2 shows a typical scene that can be rendered using this method.

As pointed out by Wyman and Davis [116], the rendering of the vertex array using one pixel sized point primitives results in speckled caustics. In addition it was observed that the rendering of points with the same intensity does not account for the spread of transferred energy over an area receiver. To overcome this limitation, two different approaches were suggested. Either the intersection points of reflected or refracted light rays with the planar receiver are connected and rendered as caustics polygons

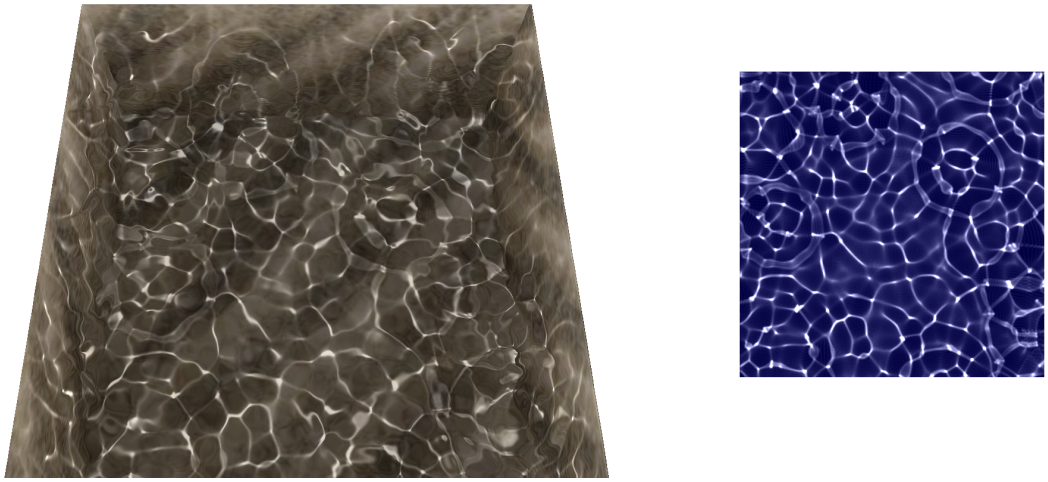


Figure 6.2: Caustics on planar receivers: 256^2 Photons are rendered as point primitives into a cube-map. On a $1K \times 1K$ viewport the scene is rendered (including the construction of a 5×128^2 cube-map) at over 130 fps on current GPUs. The illumination map for the bottom plane is shown on the right.

with area-dependent intensity, or the intensity at the receiver is computed by adding the contributions from all photons hitting the object and by spreading these intensities in image space. Both alternatives assume adjacent primitives—either vertices in the vertex array or pixels in image space—belonging to the same object and being close to each other in world space.

With respect to the aforementioned GPU implementation of caustic effects two questions remain to be answered. First, how can several reflections or refractions be handled. Second, how can intersections between light rays and arbitrary, i.e., non-planar, diffuse receivers be computed. Wyman [114] proposed to approximate the exit point of a light ray refracted several times in the interior of a focal object by a linear interpolation between two distances: The distances from the entry point to the convex hull of the object into the direction of the inverse normal and into the direction of the undistorted light ray. While the former distance is pre-computed the latter is obtained from the depth buffer at run-time. The refracted image of the surrounding scene is then obtained using a final lookup into an environment map. An extension to this approach that also accounts for geometry close to the exit point was later proposed in [115]. Although the approximation of the refracted ray is valid only if the object is convex and both intersection points along the inverse normal and the undistorted ray fall within the same polygon, in particular if the refraction is low, the method gives realistic results at impressive frame rates. On the other hand, correct intersections between light rays and non-planar diffuse receivers have not been addressed in this work.

Shah and Pattanaik [97] estimate the intersection point between a single refracted light ray and a receiver using the undistorted image of the receiver as seen from the light source. Starting with an initial guess for the intersection point, its position is corrected by iteratively moving this point along the refracted ray in screen space. As a consequence, the quality of this method strongly depends on the initial guess as well as on the size and the shape of the receiver.

Nishita and Nakamae [80] employed frame buffer hardware for blending polygonal caustics beams. This work was later extended by Iwasaki et al. [54] to efficiently utilize the potential of programmable graphics hardware. Wand and Strasser [110] suggested to gather the radiance at sample points on a receiver by rendering the scene as seen from these points. The method effectively accounts for caustics shadowing, but it requires the scene to be rendered several times and restricts the gathering to a rather small solid angle. Larsen and Christensen [72] proposed to compute photon maps on the CPU, to render photons as point primitives and to filter the photon distribution in image space on the GPU. Distance impostors were introduced in [103] to efficiently approximate the intersection points between light rays and reflections stored in environment maps. Ernst et al. [30] built on the concepts of polygonal caustics beams and shadow volumes [24]. For every patch of the focal object a caustics volume is created, and it is used as a shadow volume that illuminates points of the scene inside that volume. The method is well suited for the rendering of underwater scenery including shafts of light, but it introduces a significant geometry load and does not account for the blocking of caustics rays by a receiver. Iwasaki et al. [53] suggested an acceleration technique for caustics simulation based on the voxelization of polygonal objects into a stack of 2D slices. Approximate caustics can then be rendered by computing the intensities on these slices.

From the above discussion it becomes clear that it is still a challenge to develop techniques that enable real-time *and* accurate rendering of caustics on and caused by complex objects. The method proposed in this work addresses these requirements in that it provides an effective means to resolve inter-object light transfer as well as several refractions at complex and dynamic polygonal objects. This is achieved by the following strategy:

- Photon rays are traced in screen space on programmable graphics hardware. This is realized by rendering line primitives with respect to the current view. To be able to resolve intersection events at arbitrary positions along these lines they are rasterized into texture maps.
- Intersections between photon rays and objects in the scene can now be detected using layered depth maps and simple pixel shader operations.
- By rendering oriented point sprites at the receiver pixels in screen space we account for the energy transport through beams of light. The use of energy splats significantly minimizes the number of photons that have to be traced.

Thus, an interactive method at high visual quality is proposed. As this method does neither require any pre-processing nor an intermediate radiance representation it can efficiently deal with dynamic scenery and scenery that is modified or even created on the GPU. Caustics shadowing is implicitly accounted for by terminating photon rays at the diffuse objects being hit. By only slight modifications the method can be used to render shafts of light as well.

Besides the advancements our method achieves, the following limitations are introduced by its special layout: First, as intersection events are resolved in screen space, intersections with triangles not covering any pixel will be missed, i.e. triangles outside the view frustum, triangles parallel to the view direction

and triangles below the screen resolution. Second, the accuracy of the method depends on the accuracy of the scan-conversion algorithm implemented by the rasterizer as well as the floating point precision in the GPU shader units. It is thus clear that the proposed method can produce images that are different from an exact solution. It is, on the other hand, worth noting that the proposed method significantly accelerates photon tracing and makes it amenable to interactive and dynamic scenarios. In addition, as will be shown in a number of examples throughout this chapter, it achieves very plausible results without notable artifacts.

6.1.2 Screen-Space Photon Tracing

In the following we assume that for a set of photons emitted from the light source all specular-to-specular bounces have been resolved and a final specular-to-diffuse energy transfer is going to take place. For each photon a position along with the direction of this transfer is stored in the *transfer map*. For single refractions, for instance at a water surface, the GPU-based approach as described above can be directly used to generate this information. More complex bounces can be simulated using either the method proposed by Wyman [114] or the one we present at the end of this chapter.

The problem is to find for all photons the intersections with the receiver objects in the direction of light transfer. This can be done on a per-fragment basis by rendering caustics rays as line primitives clipped appropriately at the scene boundaries. The idea then is to process the fragments being generated during the rasterization of these lines by a pixel shader, and to only illuminate those pixels that correspond to an intersection point. Assuming that the depth image of the scene is available in a texture map, intersections can be detected by depth comparison in the pixel shader. Every fragment reads the corresponding value from the depth map and compares this value to its own depth. If both values differ less than a specified threshold the fragment color is set. Otherwise the fragment is discarded.

The envisioned algorithm has two basic problems: First, as it finds and illuminates all intersection points along the photon paths it is not able to simulate the termination of these paths at the first intersection points. While a shadow algorithm could be utilized to attenuate this effect, caustics focusing in the interior of the shadow, and especially shadowing effects due to dynamic focal objects cannot be simulated in this way. Second, intersections with occluded geometry, which is not present in the depth map, cannot be found (see Figure 6.3).

6.1.3 Line Rasterization

To overcome the aforementioned limitations we suggest to rasterize photon rays into a texture map and to perform intersection testing for every texel in this map. The first intersection along each ray can then be determined using a log-step texture reduce min operation as described in Section 3.1.2. The coordinates of the detected receiver points—now stored in the RGB components of a texture map—can finally be rendered in turn on the GPU without any read-back to the CPU.

Every photon ray is rendered into a distinct row of a 2D off-screen buffer (see Figure 6.5). This is accomplished by rendering for each photon a line primitive. Due to performance issues the set of all these

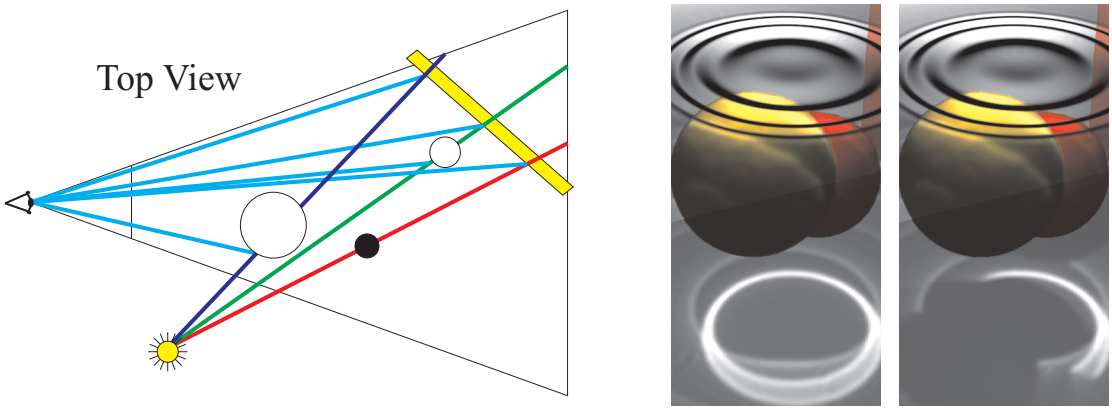


Figure 6.3: If a pixel shader illuminates all line-object intersections in view-space incorrect results are produced. In the example, five hits instead of three would be illuminated due to the black sphere being occluded in view space. In the two right images a scene is rendered without (left) and with (right) caustics shadowing. The artifacts can be clearly observed.

lines is stored in a vertex array in GPU memory. For every line being rendered a vertex shader fetches the respective photon position $\vec{o}$ and direction $\vec{d}$ from the transfer map, and it computes the number of fragments, n_f, that would have been generated for this line by the rasterizer. This number can simply be obtained from the projection of $\vec{o}$ and $\vec{o} + t \cdot \vec{d}$ into screen space. The shader then displaces the initial vertex coordinates to generate a line starting at the left column of the off-screen buffer and covering n_f fragments (see Figure 6.4). In addition, the screen-space position of $\vec{o}$ and $\vec{o} + t \cdot \vec{d}$ is assigned as texture coordinates to the start and the end vertex, respectively, of every line.

During scan-conversion the rasterizer interpolates the texture coordinates along the horizontal lines. It thus generates for every fragment the screen-space position it would have been mapped to if the initial caustics ray was rendered as a line primitive. A pixel shader operating on these fragments can use this position to fetch the respective value from the depth map and to compare the interpolated screen-space depth to this value. If an intersection is determined, the screen-space position of the fragment in the off-screen buffer is written to that buffer. Otherwise the fragment is discarded (see Figure 6.4).

After all lines have been rasterized, a texture is generated that stores in each row all the intersections between a particular photon ray and the visible objects in the scene. By applying a horizontal texture reduce operation the first intersection points are rendered into a single-column texture map. For photon rays that are almost parallel to the view direction and thus only cover very few fragments a minimum line length is enforced. If the number of rows in the render target is less than the number of photons, the process is repeated for the remaining photons until all of them have been processed.

Similar to our volume raycasting approach as described in Section 5.2, photon tracing can also be implemented as a single pixel shader being executed for each photon. In theory, the advantage of such an implementation is that the traversal process can stop as soon as an intersection with any of the objects in the scene is found. However, a comparison with the approach suggested here revealed a loss in performance of about 50%. We attribute this observation to the fact that early-out mechanisms like “dis-

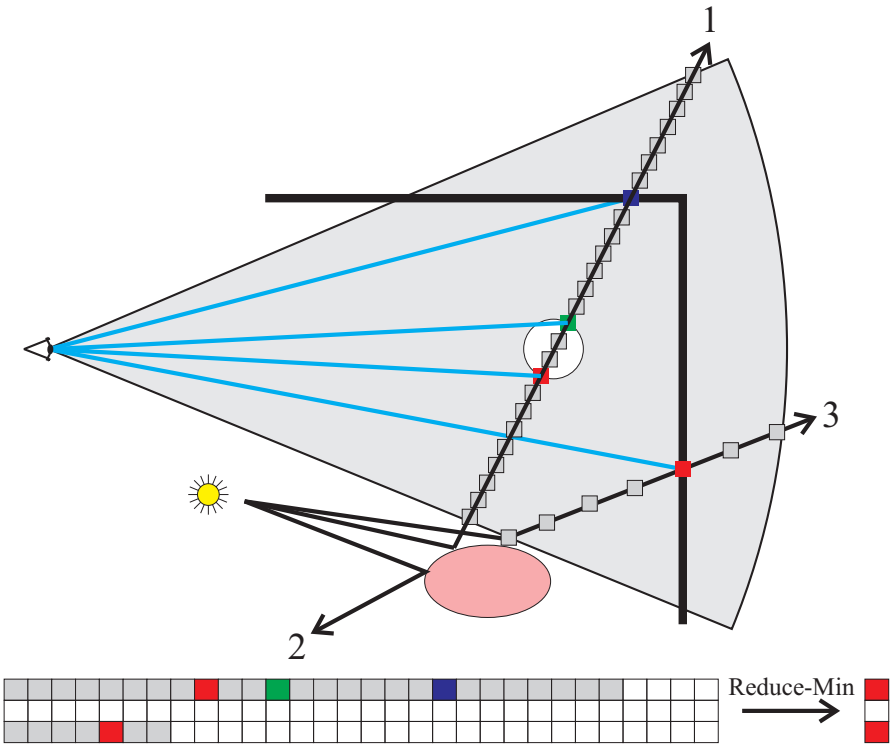


Figure 6.4: The pink object at the bottom reflects the light from the light source and causes three caustics rays. These are processed by the vertex shader, which computes the length of each ray and transforms them to horizontal lines of equal length. The rendering of these lines generates the image shown at the bottom. Red, green, and blue pixels indicate hits with an object, and grey cells indicate fragments that have been discarded in the pixel shader. Into white cells a fragment has never been rendered.

card" statements in a pixel shader introduce some overhead in form of an expensive branch instruction. Moreover, since the pixel shader hardware runs in lock-step, a performance gain can only be achieved if all fragments in a contiguous array exit the program early. These observations are backed up by latest GPUBench [15] results*. These results attest current GPUs a rather bad branching performance even if all fragments in a 4×4 block exit the program simultaneously. However, if in future hardware this ray-casting approach will be more efficient, it should be clear that such functionality can easily be integrated into our approach without introducing any significant modifications.

6.1.4 Depth Peeling

In the presentation so far intersections with occluded geometry have been entirely ignored. The reason for this is that only one single depth map containing the depth values of the visible fragments has been considered. As can be imagined easily, this limitation introduces false results and noticeable artifacts.

To be able to compute the intersection between photon rays and all objects in the scene we generate a

*Results for our target architecture, the GeForce 7800 GTX, are available at http://graphics.stanford.edu/projects/gpubench/results/)

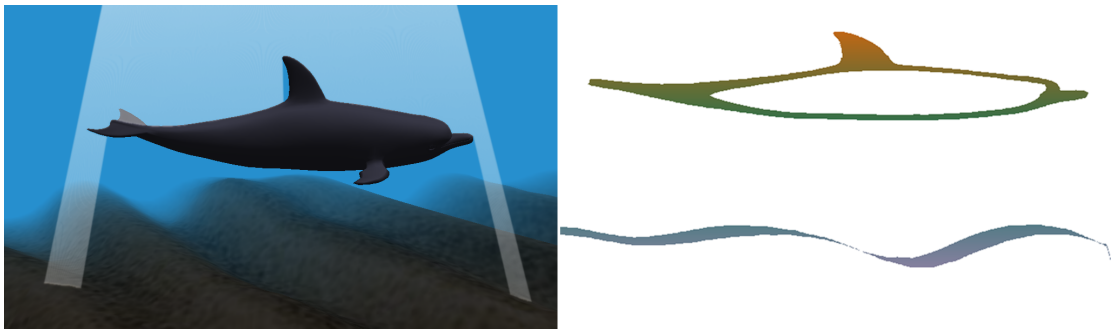


Figure 6.5: The right image shows an except of the ray intersection texture. In the left image the corresponding rays can be seen. For this image a very large epsilon was chosen.

layered depth image [96] from the current viewing position. Therefore, we employ depth-peeling [31]. In the current implementation we use depth-peeling not only to generate multiple depth maps but also to generate multiple normal maps of the layered fragments (see Figure 6.6). In the caustics rendering approach with lines, instead of comparing the fragment's depth to only the values in the first depth layer we now compare it to the values in the other layers as well. An intersection is indicated by at least one entry in the layered depth map that is close to the fragments depth.

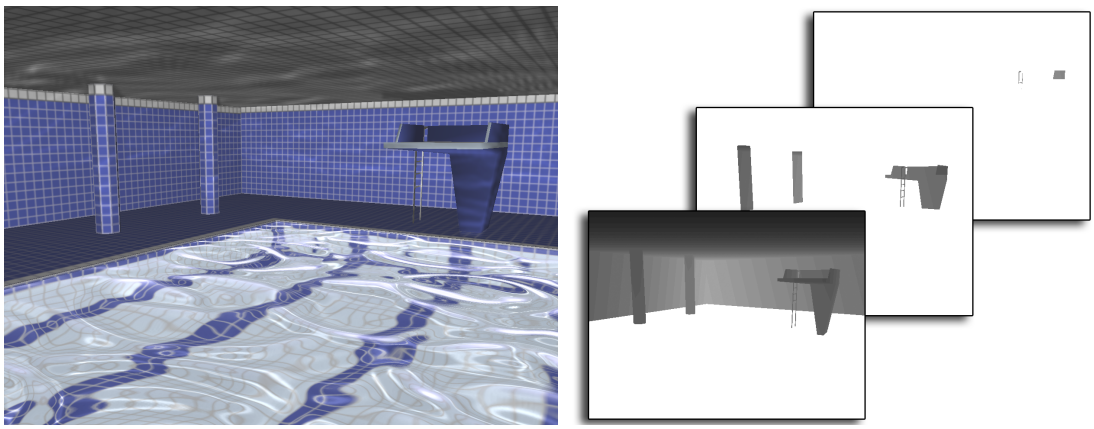


Figure 6.6: The caustics pool demo is shown on the left. In the right image the first three depth layers can be seen.

To generate the layered depth image we need to know the depth complexity of the scene for the current view. The depth complexity can be determined by rendering the object once and by counting at each pixel the number of fragments falling into it during scan-conversion. The maximum over all pixels is then collected by a texture reduce-max operation. As the depth complexity of a scene can vary significantly depending on the view-point and the viewing direction, depth-peeling has to be restricted to a maximum number or passes to maintain a constant frame rate. Interestingly, our experiments have shown that it is usually sufficient to only consider up to eight depth layers. This is due to the fact that

deeper layers do in general not contain a significant number of fragments, and only a few of them block any photon ray that would otherwise be seen by the viewer. In scenarios where the depth complexity is high because the scene consists of many objects, separate depth images for each object should be favored to reduce geometry load and memory overhead.

6.1.5 Recursive Specular-to-Specular Transfer

The proposed method enables the tracing of photons through the scene until the first intersection with any object in this scene is found. For every photon the algorithm outputs the position of the intersection point, the normal at this point and the direction of the incoming photon ray. It is thus easy to simulate subsequent specular-to-specular light transfer by reflecting or refracting the incoming ray at the intersection point, and by writing both the point coordinate and the modified direction vector into the transfer map. The algorithm is then restarted with the updated map.

6.1.6 Rendering

When light hits a diffuse surface where it is emitted equally in all directions, the scene is illuminated at the point of intersection. As we have described above, for all photons these *receiver points* are encoded in the transfer map. To minimize the number of photons to be traced through the scene we assume that a given photon deposits a certain amount of energy in the surrounding of the intersection point. Similar in spirit to the distance weighted radiance estimate proposed by Jensen [55] we spread the photon energy over a finite area centered around the intersection point. Surface points within this area receive an energy contribution that is inversely proportional to their distance to the center. In contrast, however, in our approach we do not gather the energy from nearby photons but we accumulate the contributions by rendering area illumination splats.

6.1.7 Sprite Rendering

To simulate the spread of energy at a particular point in the scene we employ point sprites as explained in Section 5.3.4. Unfortunately, point sprites come with the restriction that they are screen-aligned; i.e., they resemble quadrilaterals centered at the point position and kept perpendicular to the viewing direction. As the alignment of point sprites according to the orientation of the receiver is not supported by current GPUs, a new method that overcomes this limitation is required.

At the core of our technique we have developed a GPU-based ray-tracer similar to the one proposed in [14] for high-quality point rendering. The idea is to compute for every pixel covered by a sprite the point seen under this pixel in the tangent plane at the receiver point. This plane is defined by the normal stored for every receiver point in the transfer map. The distance of the ray-plane intersection point to the receiver point can directly be used to estimate the amount of energy received from the photon. It also has to be considered that only *visible* surface points should be illuminated. This is accounted for by comparing the screen-space depth of the ray-plane intersection point to the value in the first layer of the

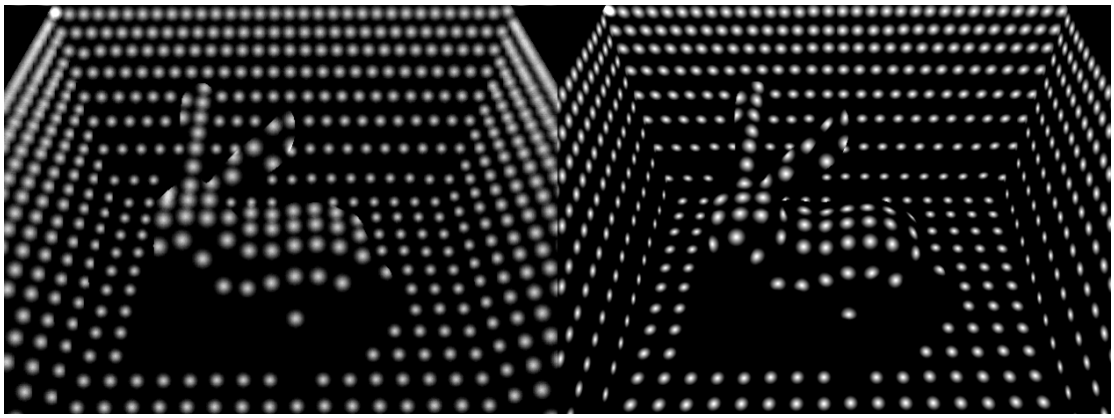


Figure 6.7: Sprite rendering: The images show light patterns generated by a few photons. The difference between screen-aligned textured point sprites (left) and the sprite-based raycaster (right) is shown.

layered depth image at the current pixel position. Only if both values are close together does the pixel receive an intensity contribution. In Figure 6.7 the quality of this approach is demonstrated.

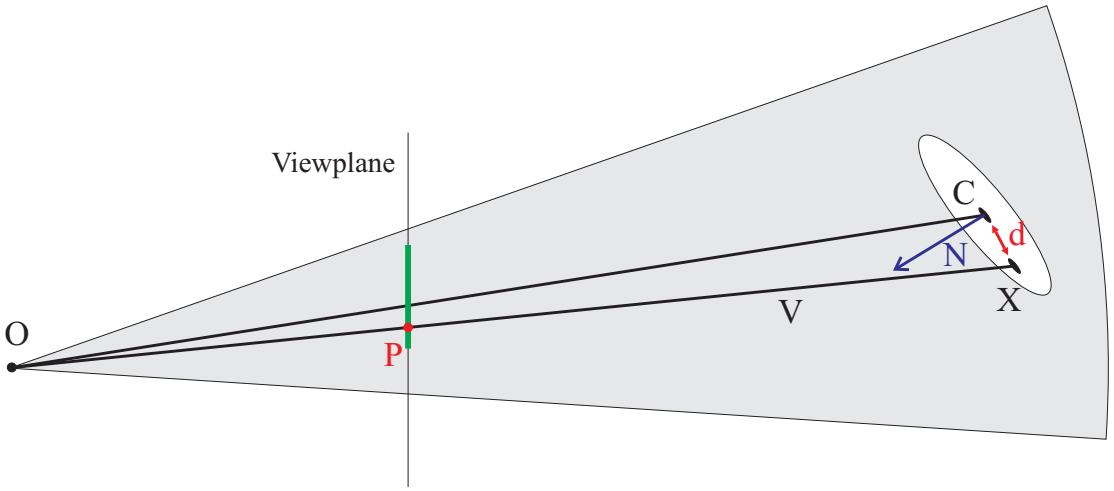


Figure 6.8: Schematic view of sprite-based raycasting. P is a pixel in the image plane, O denotes the viewing position and d is the distance of the intersection point to the receiver point. The distance is used to calculate the intensity. For a spherical intensity splat with size s and linear attenuation this is $s - d$.

To simulate the illumination caused by a photon ray, for each of the receiver points in the transfer map a point sprite of constant size s is rendered. For every fragment generated during rasterization a pixel shader computes the intensity I as follows (based on the setup in Figure 6.8):

$$I = \max\left(0, \frac{s - \left|\frac{\vec{N}\cdot\vec{C}}{\vec{N}\cdot\vec{V}} \cdot \vec{V} - \vec{C}\right|}{s}\right) \tag{6.1}$$

Here, $\vec{N}$, $\vec{C}$, and thus $\vec{N} \cdot \vec{C}$ are constant over the sprite and can be computed in a vertex shader and passed as constant parameters to the fragment stage. The view direction $\vec{V}$ is also computed in a vertex shader by multiplying the pixels screen-space coordinate with the inverse projection matrix. Equation 6.1 is easily derived from the following observation. Given the splat normal $\vec{N}$ and its center $\vec{C}$, then for any point $\vec{X}$ on the plane orthogonal to $\vec{N}$ the following condition holds:

$$\vec{N} \cdot \vec{X} = \vec{N} \cdot \vec{C} \tag{6.2}$$

Furthermore, the intersection point $\vec{X}$ between the ray starting at $\vec{O}$ into the direction $\vec{V}$ can be expressed as:

$$\vec{O} + t \cdot \vec{V} = \vec{X} \tag{6.3}$$

Using 6.3 in 6.2 and solving for t results in

$$\begin{aligned} \vec{N}\left(\vec{O} + t \cdot \vec{V}\right) &= \vec{N} \cdot \vec{C} \\ t &= \frac{\vec{N} \cdot \vec{C} - \vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{V}} \\ &= \frac{\vec{N} \cdot \vec{C}}{\vec{N} \cdot \vec{V}} \text{ with } \vec{O} = \vec{0} \text{ in eye coordinates} \end{aligned}$$

Therefore, we can compute the distance between the intersection point $\vec{X}$ and the center $\vec{C}$ as

$$d = \left| \underbrace{\frac{\vec{N} \cdot \vec{C}}{\vec{N} \cdot \vec{V}} \vec{V}}_{=\vec{X}} - \vec{C} \right|$$

and we can use this value to modulate the intensity of the fragment with Equation 6.1.

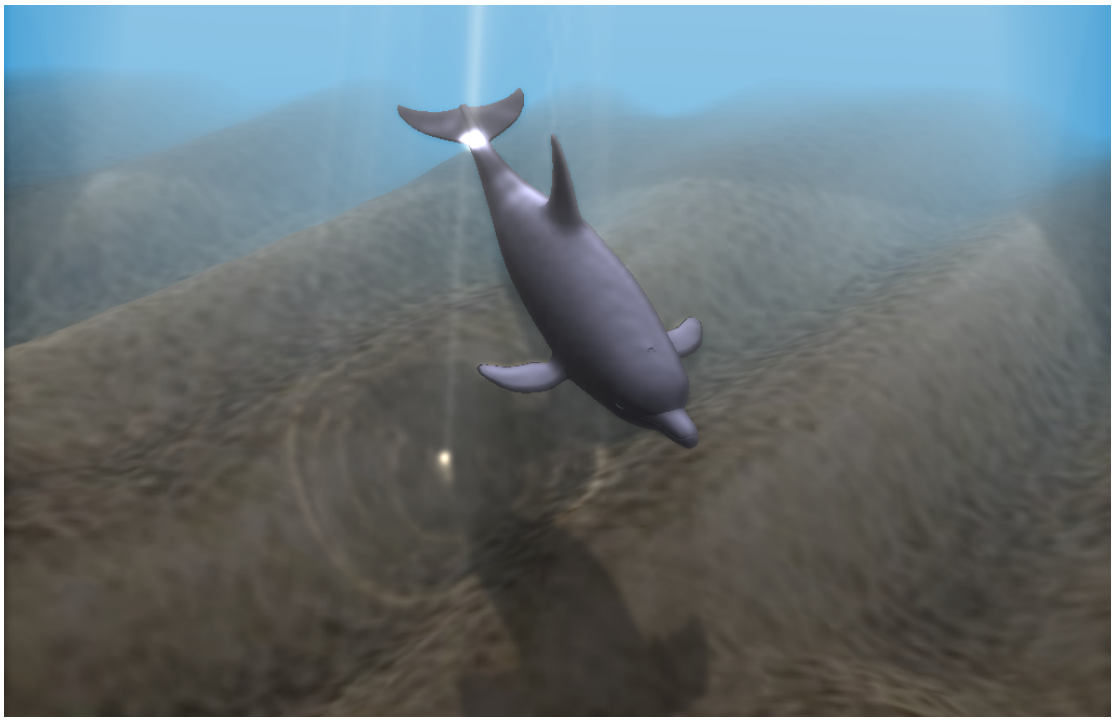


Figure 6.9: Underwater scene illuminated by sunlight. In this scene the light rays converge towards the flukes of the dolphin in a bright spot.

6.2 Rendering in Homogeneous Participating Media

Besides simply illuminating surfaces, light causes another fascinating effect when passing through homogeneous participating media: Shafts of light or god-rays (see Figure 6.9). Adding these shafts of light to our approach is fairly simple as we know for every photon ray its start position at the focal object and the position where it terminates. Consequently these rays can be rendered as line primitives, which are combined in the frame buffer. As we assume the media to be homogeneous, light intensity decreases exponentially with distance. This kind of attenuation can be computed in a pixel shader for every fragment that is rasterized. Contributions from multiple god-rays are accumulated in the frame buffer.

To account for the scattering of light along the god-rays, they are rendered into a separate color buffer and blurred with a Gaussian filter. As this filter has a constant screen space support it emulates a filter with increasing support in object space; i.e., rays further away from the viewer are blurred more than those closer to the viewer. The filtered image is finally blended over the image of the scene. As the images demonstrate, the results look very realistic and convincing although only a very coarse approximation of the real scattering of light is computed. The overhead that is introduced by filtering the god-ray image is insignificant compared to the other stages of the caustics rendering algorithm.

6.3 Screen-Space Refraction Tracing

The idea of screen-space ray-tracing can also be applied to render refractions through transparent objects as seen by the viewer. Compared to the method suggested by Wyman [114], this method can efficiently deal with dynamic and concave geometry and it simulates refractions more accurately. On the other hand it comes at the expense of more complex fragment computations thus limiting the performance that can be achieved.

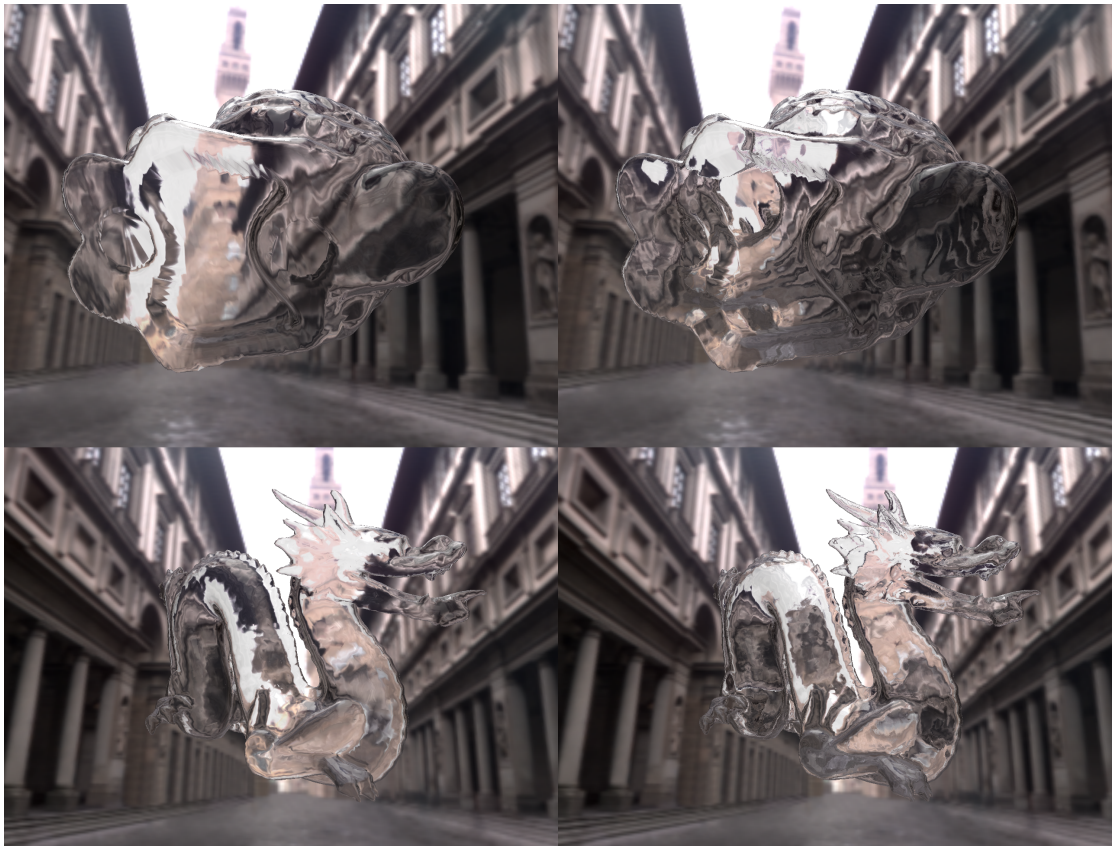


Figure 6.10: Comparison between refractions through only one interface (left) and multiple refractions generated via screen-space photon tracing (right).

For the tracing of refracted lines of sight in screen space we use the same approach as for the tracing of photons in screen space. The main difference is that now the projected lines usually cover only a few pixels on the screen. This is due to the fact that at least the rays entering at close to perpendicular angles stay almost parallel to the view ray after the first refraction. Then the same ideas suggested in Section 6.1.2 can be utilized. In contrast, however, we built the tracing of refractions on the simplification that along any ray the n-th intersection point with the refracting object is found in the $(n + 1)$-th depth layer (see Figure 6.11). We thus assume that ray-object intersection are "monotonic" with respect to these layers. In this way the number of texture access operations to be performed at every pixel under

the view-ray is equal to one. It is clear that testing against all levels of the layered depth image causes the performance to drop linearly in the depth complexity of the object. As the possibility of violating this simplification increases with the strength of the ray deflection, in such scenarios there is a higher chance to miss interior object structures.

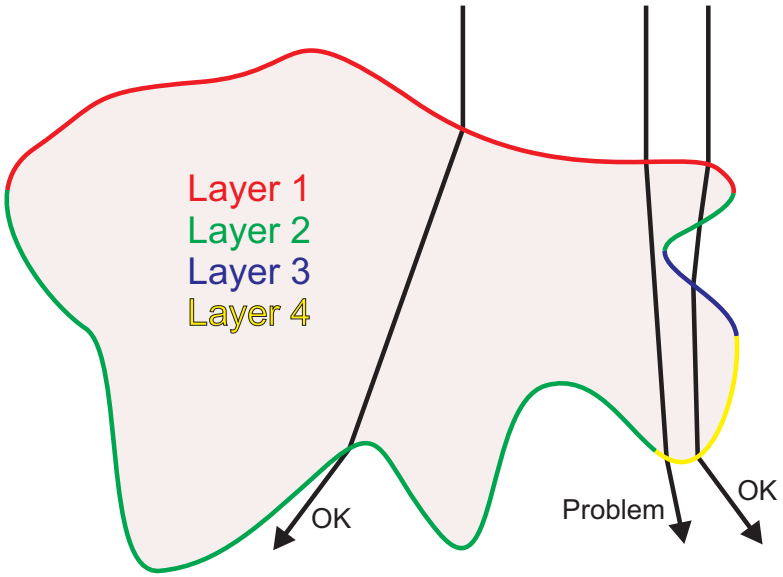


Figure 6.11: This image depicts refraction on an object that was peeled in four depth layers from top to bottom. Two rays that hit the layers in monotonic order and one problematic ray that "skips" a depth layer are shown.

The algorithm starts by constructing the layered depth image of the transmitting object. Visible object points along with their normals are rendered into the transfer map. We then rasterize refracted view-rays into a texture map and perform intersection testing for every texel in this map. These rays, just like the caustics rays, are then traced through the depth layers until the depth along the ray is either greater than the depth in the next layer or less than the depth in the current layer. In either case we can assume an intersection with the surface. The point in the interior of the object is considered the exit point and the normal at this point is fetched from the depth image. From this normal and the incoming ray direction the refracted exit direction is computed. This information in turn can either be used to look up an environment map or to retrace the refracted view-rays. Figure 6.10 shows a comparison between refractions through only one interface and refractions generated by our method.

If a layered depth image of the surrounding scene exists, intersections between the exiting ray and the scene can be detected as well. This approach has been applied in Figure 6.12 to simulate caustics seen through the water surface. It should be clear that this is an approximation that can only simulate the refraction of light from objects that are visible to the viewer in the absence of the refracting surface. Otherwise, the refracted object points have not been rendered and they are thus not available in the image of the scene. It is interesting to note that especially for low refractions most of the refracted rays are

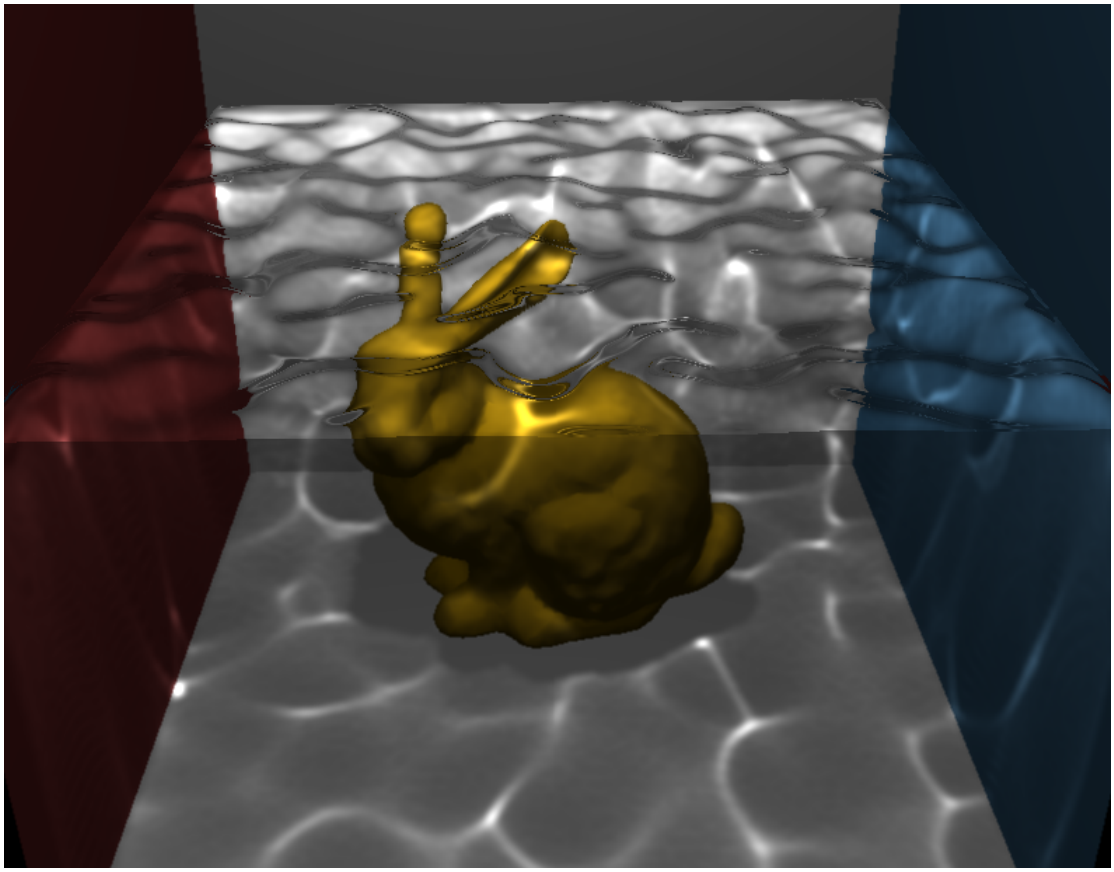


Figure 6.12: A Stanford Bunny in the Cornell Box. Note how the refraction on the water surface displaces the bunnies ears.

almost parallel to the view ray. In this case we only need to trace over a few pixels to find the exit point. Yet this exit point is still the optimal position with respect to the limited resolution of the depth buffer, e.g., even for refracted rays that are perfectly parallel to the view direction and thus generate only a single tracing step, we still find the correct exit position namely the position stored the layered depth/normal layer exactly behind the entry point. Thus, since the performance of our algorithm is dominated by the length of the refracted rays in screen space we achieve high frame rates.

6.4 Performance Measurements

Table 6.1 lists timings for the rendering of refracting objects using the proposed method. In these tests only the first exit point has been considered for casting view-rays into the surrounding scene. As it can be seen, on recent graphics hardware the algorithm is only slightly slower than a highly optimized version of the one proposed in [114]. It is, on the other hand, worth noting that our technique does not require any pre-processing and is thus suitable for dynamic geometry.

	triangles	Viewport resolution 800×600	1200×1024
Teapot	2257	600 (870)	220 (300)
Stanford Bunny	69665	220 (270)	110 (180)
Stanford Dragon Low	47794	280 (320)	120 (160)
Stanford Dragon High	202520	150 (156)	95 (105)

Table 6.1: Timing statistic in fps for screen-space refraction tracing to allow a comparison to Wyman's approach [114] (fps in parenthesis). Our algorithm was set to compute only two refractions.

Representative timings in milliseconds (ms) for caustics simulation in three example scenes are listed in Table 6.2: In the bunny scene ("A", Figure 6.12) the refraction of underwater objects at the water surface was simulated as described in Section 6.3. God-rays were added to the dolphin scene ("B", Figure 6.1). In the pool scene ("C", Figure 6.13) refractions as well as reflections at the water surface were simulated. All scenes have a depth complexity of eight.

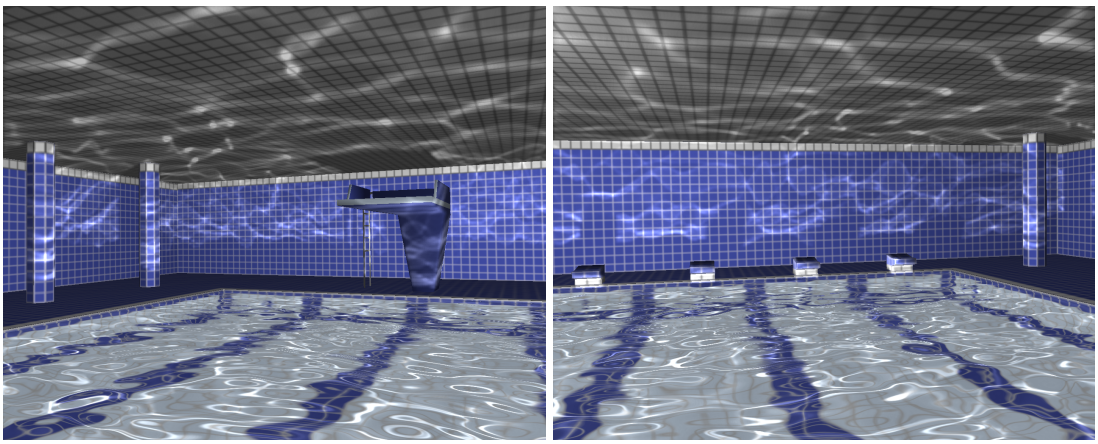


Figure 6.13: Refractions and reflections on a water surface.

The first column "C" shows the amount of time spent by the GPU for caustics simulation including depth-peeling, line rasterization, intersection testing and god-ray rendering. Of all these parts intersection testing consumed over 80% of the time. In the second column "P" the time spent rendering the point sprites is given. In the last column "O" the overall performance is shown. All measurements have been repeated for different amounts of photons traced from the light source. Photon grids used in the pool demo are 1024×512, 512×256, 256×128, and 128×64. Additional results using 1024×1024 photons are shown in Figure 6.14.

As the timings show, even very complex light pattern can be simulated at interactive rates and convincing quality. In particular such effects as shafts of light and caustics shadowing, for instance below the bunnies and the dolphin, and behind the diving platform, add increasing realism and diversity to the 3D scenery. The images also show that even with a rather small number of photons traced through the scene

	Number of photons								
	512×512			256×256			128×128		
	C	P	O	C	P	O	C	P	O
A	243	26	285	69	8	89	17	3	29
B	232	17	555	62	6	103	17	2	45
C	*434*	*10*	*454*	*119*	*7*	*135*	*32*	*2*	*33*

Table 6.2: Timing statistics (in ms) for different scenes.

the caustic effects look very realistic and do not exhibit intensity speckles. This is due to the sprite-based approach that realistically accounts for the orientation of the receiver geometry and the spread of photon energy.

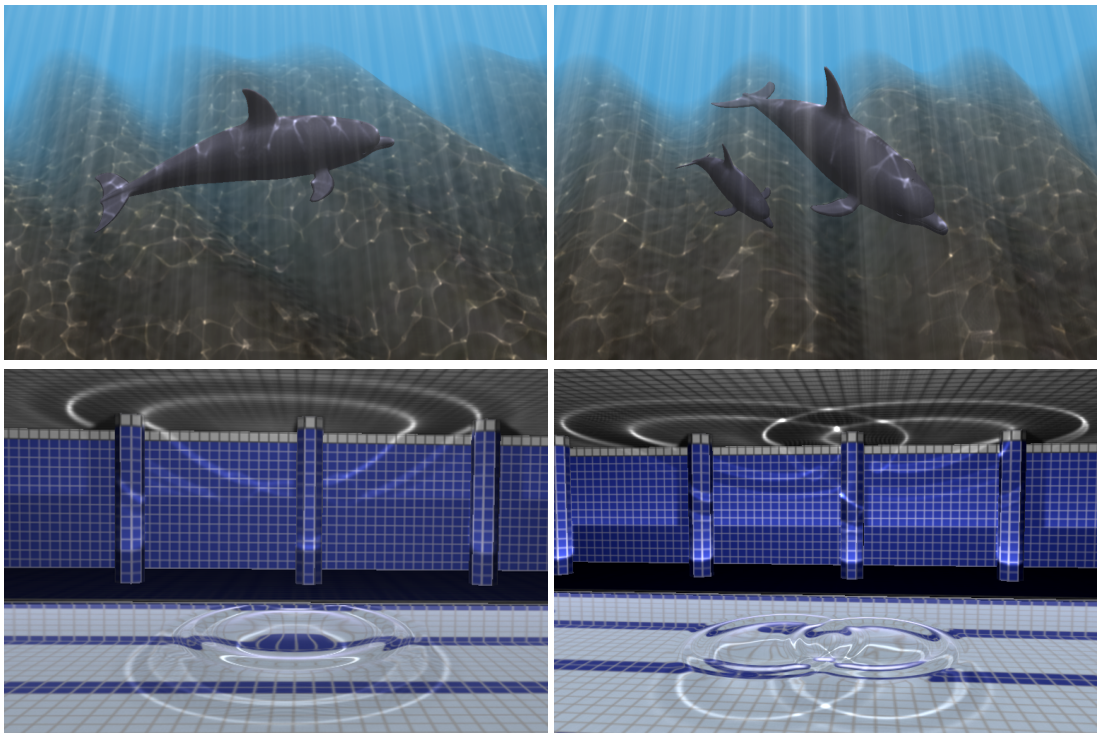


Figure 6.14: Screen-space photon tracing using 1024×1024 (Dolphin) and 512×256 (Pool) photons.

Chapter 7

Results and Future Work

7.1 Results

As promised at the very beginning of this dissertation I will summarize what has been done in the last four years of my Ph.D. work in this final chapter. First, all the software components that implement the ideas and algorithms proposed so far are presented, and their interplay is pictured. Next, a detailed description of the features and the user interface of the applications is given, depicting the scenario of a flow field designer working for a game or virtual environment to create a specific effect. Finally the thesis is concluded with some ideas about ongoing and future work.

7.1.1 Software Components

In Figure 7.1 we give an overview of the software components developed in the course of this thesis. As can be seen, the components corresponding to the algorithms presented here are all encapsulated in a layered architecture. This stack resides on top of the graphics API, colored in grayscales. On top of these blocks reside two stacks of software components: First the blue low-level blocks that allow for the efficient numerical simulation of fluids and second the rendering subsystems, colored in red. Target applications, such as those presented in the remainder of this chapter, use these components as simulation and rendering subsystems. Furthermore, these applications are free to add new functionality on top, such as caustics presented in Chapter 6. To support the application programmers, well defined interfaces exist even to the lower-level functions; e.g., the memory manager of the LA framework (see Section 3.2.5) that allows higher-level functionality to benefit from its efficient memory pooling and reuse capabilities.

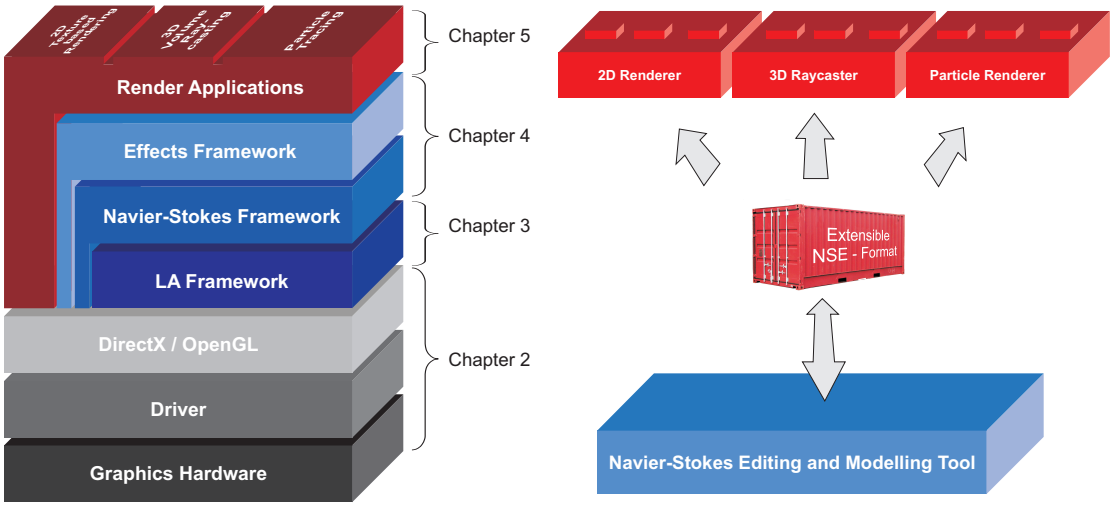


Figure 7.1: The blue and red blocks in left image show the software stack of modules developed in chapters 2 to 5 in this thesis. The gray blocks demote system and GPU vendor layers, blue blocks indicate the LA and simulation layers, and the red blocks finally contain the rendering applications. The right image depicts the horizontal data flow via the Navier-Stokes Equation container, which allows all applications to work on common data.

.

As can be seen in the right image of Figure 7.1, the software components developed in this thesis are not only connected vertically but also horizontally. This means that all applications, demos, and design tools use a common file format, which allows them to communicate. This interconnection is a very important feature for the usability of the programs in a streamlined environment, from the design sketch to the final frame. As we will see in the next section, end users only need to familiarize themselves with a single WYSIWYG modeling tool to create effects for all subsystems. Thus, the user has a true design-once-run-everywhere experience.

From the developers point of view the software components behind the applications exhibit the same level of abstraction as the applications convey. Thus, programmers who decide to implement their own modeling applications or want to integrate our functionality into their existing systems can very easily integrate our software stack into their new projects. Multiple new projects that made use of the proposed software stack have proven this statement, including image registration or flow field reconstruction.

7.1.2 Applications

In this section we take a closer look at the applications that have been developed in the course of this thesis. The theoretical concepts presented in the preceding chapters, the frameworks, the file formats, and the software components, become manifest to the user in these applications. In general, all applications are structured in main components. Firstly, the simulation engine that runs a numerical simulation on the GPU. Secondly, the visualization subsystem that visualizes the result of the simulation. While often the simulation core is shared among the applications, the visualization subsystems differ significantly.

2D Navier-Stokes Simulation

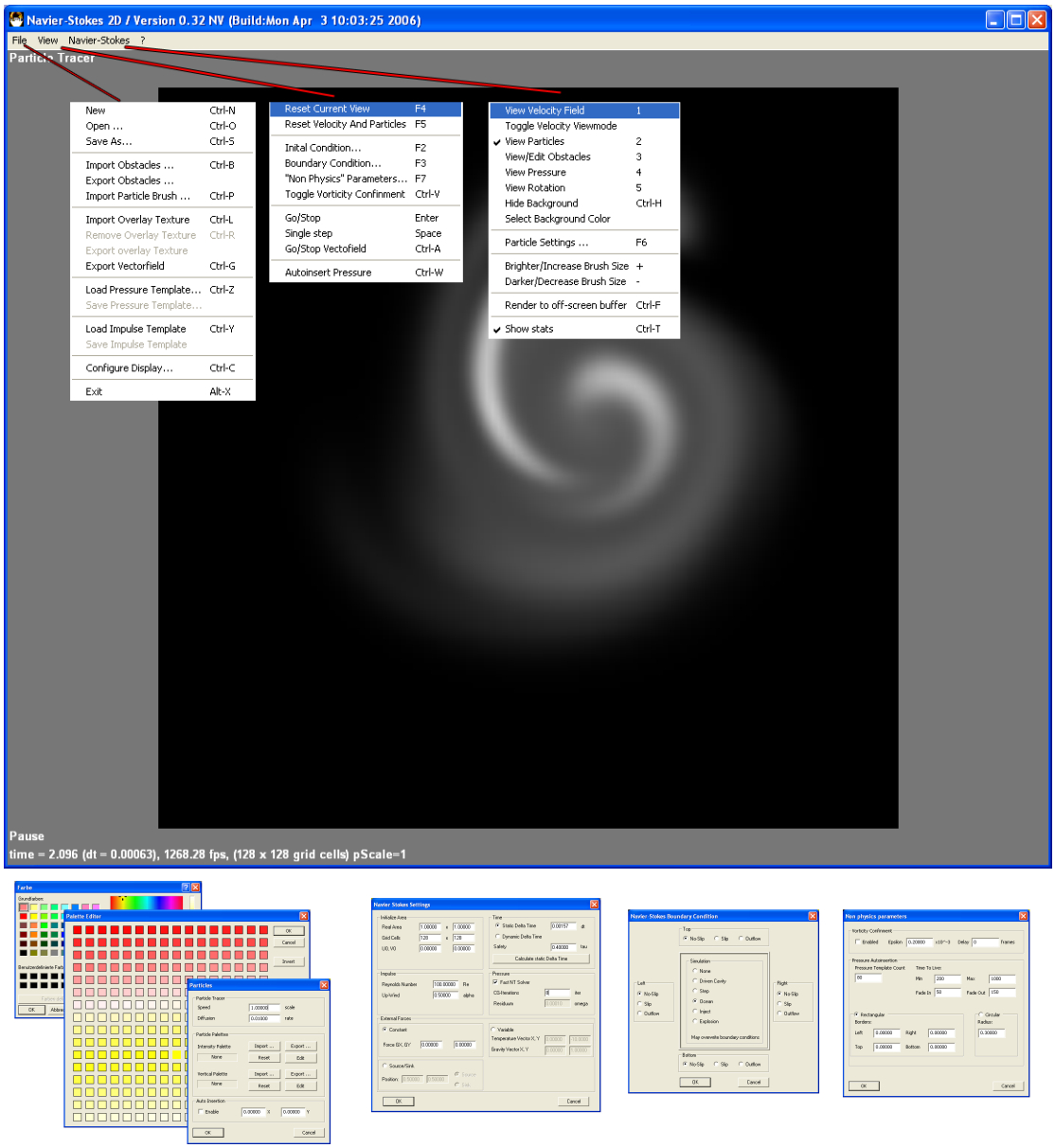


Figure 7.2: The basic UI of the "NavierStokes2D" application. The top image shows the screenshot of the program after startup together with the menu structure. The application allows the user to instantaneously interact with the flow, by inserting dye and adding external forces with the mouse.

The "NavierStokes2D" application uses the 2D Navier-Stokes simulation subsystem to generate a 2D flow field; on top of this multiple 2D texture based visualization modes are implemented. The application incorporates the 2D fluid renderer, the fluid VR demo, and the editing and modeling tool shown in

Figure 7.1 into one application. This allows a flow designer to use the various visualization modes during the design phase. Following the work-flow required to create real-time 3D animation, such as the "Explosions" demo shown in Figure 7.7, the "NavierStokes2D" program is used to create a new flow configuration. Therefore, a large set of parameter settings to control the flow properties, obstacles, and boundary conditions as well as specific visualization parameters (see Figure 7.2) are available. Note that all of the parameters have meaningful default values so that creating a new project requires the designer only to change a few of these values leaving most of them unchanged.

Through a set of views, including flow, dye, pressure, obstacle and rotation, the program allows the user to analyze the flow. Images of these modes can be seen in Figure 7.3. In all modes—except the obstacle view—the user can use the left and right mouse buttons to add dye and forces to the simulation. In the obstacle-editor mode ("blue mode" in Figure 7.3) the right mouse key is used as a pencil to draw obstacles while the left button serves as an eraser. Since the simulation requires absolutely no preprocessing, obstacles, dye, external forces, pressure and velocity templates can be applied while the simulation is running. This provides the users with instantaneous feedback about the influence of their actions.

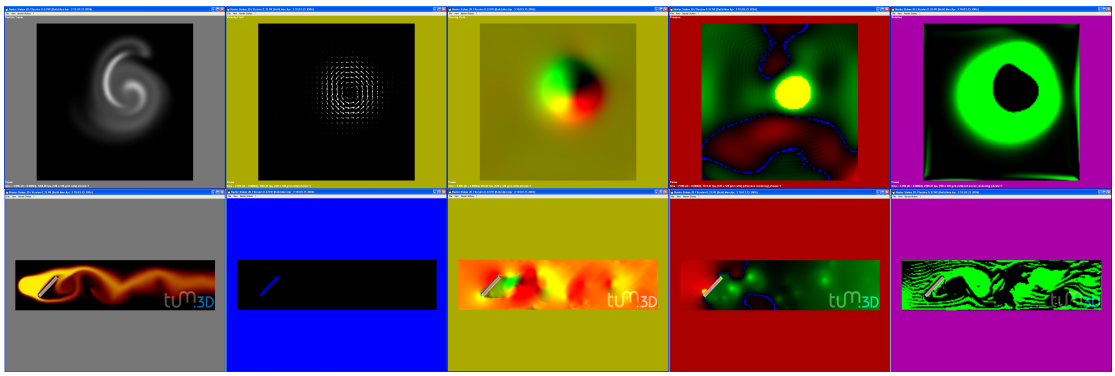


Figure 7.3: The screenshots show the five different render modes of the "NavierStokes2D" application. In the lower row from left to right, dye advection, the obstacle editor, the flow field view, the pressure distribution, and the rotation view are shown. The top row depicts the same modes for a different flow field. Instead of the obstacle editor, the second mode for the flow field view is shown; in this mode the vectors are drawn as little arrow glyphs instead of a color code.

It is worth noting that all of these interactions (i.e., the obstacles, pressure templates, and external forces) are represented on the GPU as regular texture maps. Therefore, a programmer can access these features of our framework in two ways, either via a well defined C++ interface; i.e., via methods like:

```
AddForce(const FLOATVECTOR2& Position, const FLOATVECTOR2& direction)
```

or

```
SetObstacle(const FLOATVECTOR2& ULCorner, const BITMAP& obstacle)
```

or by updating the textures directly.

As explained in Section 4.2.1 the “NavierStoks2D” system can also be used to capture templates for effect modeling (see Figure 7.4). To do so the user pauses the simulation and draws a rectangle with the mouse into the domain enclosing the desired effect; e.g., a vortex or a low pressure region. Internally the system stores a velocity template when viewing the flow field (“yellow”-mode in Figure 7.3) and a pressure template in the “red” pressure mode.

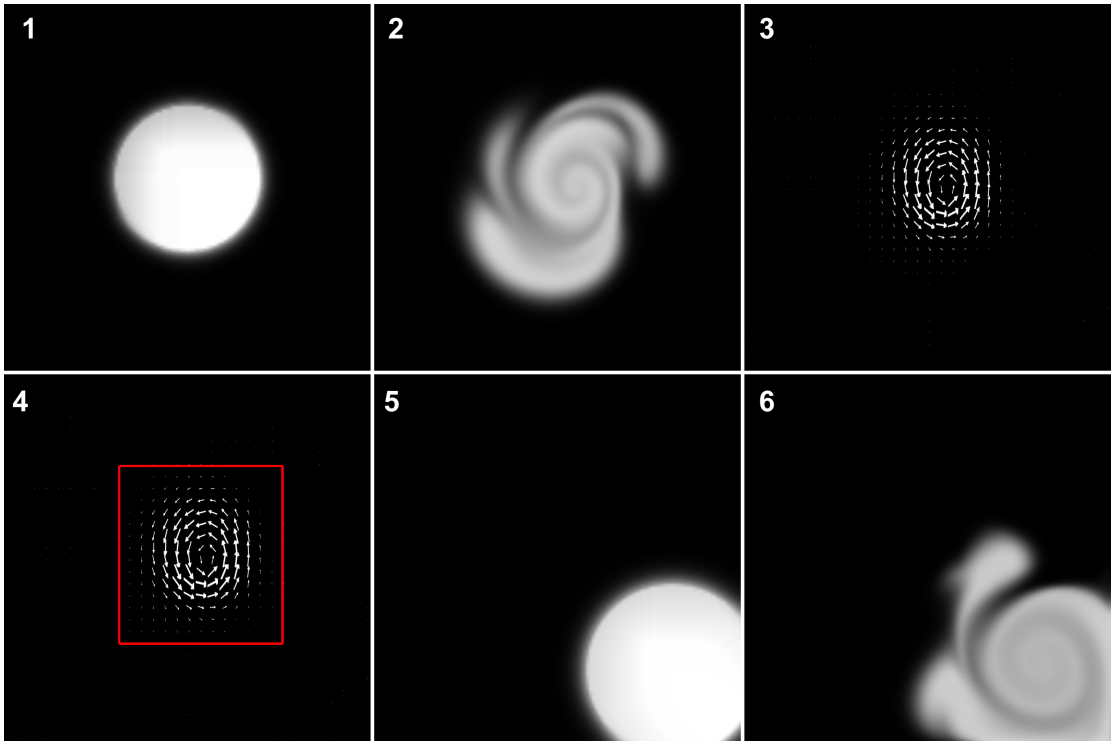


Figure 7.4: The images depict the capture and insert process of a flow template. The user starts by inserting dye (1) and applies an external force to the domain (2) then the simulation is paused. In the flow view (3) the flow field can be observed via glyphs. In this view the user draws a red quadrilateral over the desired region (4). This completes the capture process. Since the template was captured in flow visualization mode, a flow template is now stored in the database. To reinsert the template the user creates a new dye spot (5) and applies the template by hand to generate a similar vortex as the one captured before (6).

The user can now re-insert the captured template into the simulation by either clicking with the mouse while holding the shift key or by defining automatic insertion. This automatic insertion in turn can either be a predefined type, selected and parameterized from the pressure templates menu (see Figure 7.2) or it can be a complex pattern. In the later case multiple different velocity and pressure templates can be defined to be inserted into particular regions of the domain. Therefore, circular or rectangular insertion regions can be specified with the mouse and templates are selected from the palette of pre-recorded effects. In this “free insertion mode,” shown in Figure 7.5, templates are accessed via their iconic representation. This iconic representation is of particular interest in an environment where the pressure capturing and

the flow design are executed by different persons, similar to the rigging process in traditional animation where a modeler prepares the characters for the animator by adding rigs to the model.

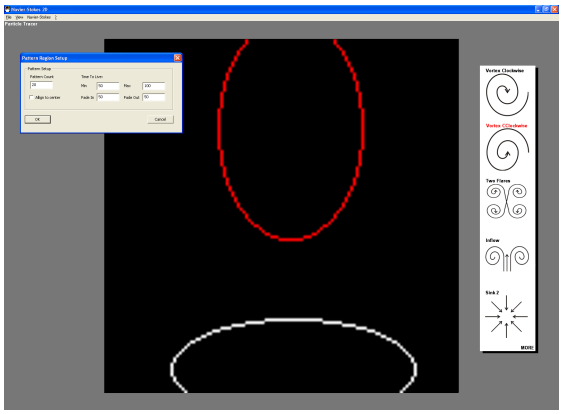


Figure 7.5: The WYSIWYG user interface to define regions for automatic insertion of templates. On the right, the iconic representations of the templates can be seen.

Besides its modeling capabilities the "NavierStokes2D" application, including the different visualization modes, also serves as the basis for other 2D applications such as the VR demo (see Section 5.1). In this demo program the mouse interaction capabilities are complemented by vision-based motion detection algorithms. Figure 7.6 shows a user interacting with the flow. The demo requires only a simple webcam and a projector connected to the PC (for details on the system see Section 5.1).

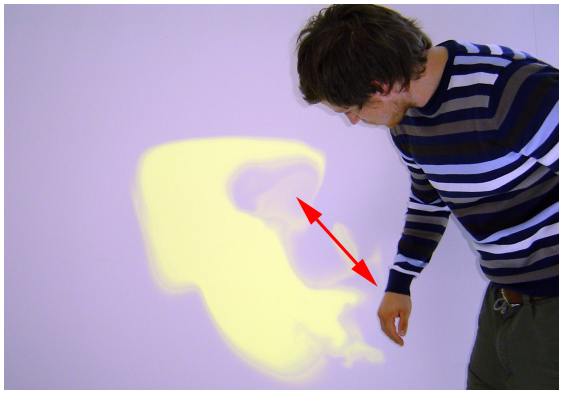


Figure 7.6: This image shows a person moving with his arm to "wave" dye particles away. The red arrow indicates the motion of the hand.

Explosions

The "Explosions" application uses multiple 2D Navier-Stokes simulation subsystems to generate a stack of 2D flow fields. On top of this, the texture revolution algorithm described in Section 4.4 together with volume raycasting is used for the 3D volume visualization. From the users point of view, a revolution

project is loaded that contains a list of 2D Navier-Stokes projects. The application opens the associated Navier-Stokes simulations and instantiates multiple Navier-Stokes simulations on one GPU. Note that due to the randomized pressure template insertion, all simulations even if loaded from the same Navier-Stokes project file will evolve in a slightly different way. To generate images—such as the ones shown in Figure 7.7—the renderer reconstructs the 3D volume from the 2D simulations by the means of the texture slice revolution. On this virtual 3D volume direct volume raycasting is performed.

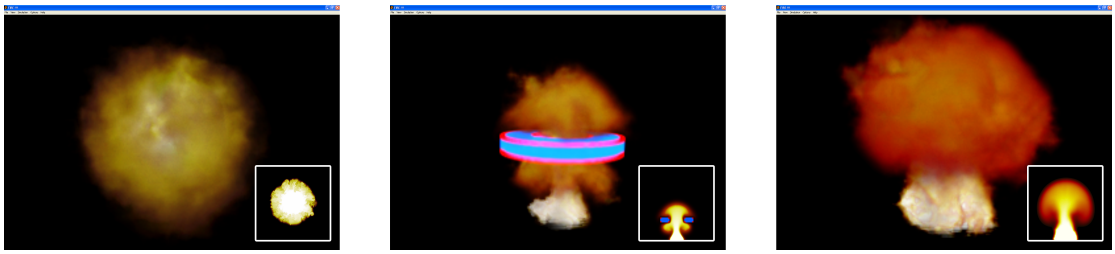


Figure 7.7: Example screenshots of three different revolution effects; the small images in the lower right show the 2D dye-advection as it was designed in the "NavierStokes2D" application.

The program itself requires only a limited user interface. Apart form the project dialogs for loading and saving, it allows the designer to set some final parameters of the projects, such as changing the dye's transfer function, volume rendering settings, or the template parameters.

Particle Engine

Particle tracing in general has proven to be a useful tool not only for effect rendering but also for scientific visualization. Since both application domains share the same core functionality, the particle engine was developed with a modular design in mind. The program consists of an engine core shared among all modules and plug-in space for different modules (see Figure 7.8). At this point it is worth noting that other modules such as pre-computed 2D, 3D, and time-dependent flow fields, as well as PIV-reconstruction, tensor field, or height field modules have been developed as plug-ins for this particle engine, emphasizing the flexible design of the engine core.

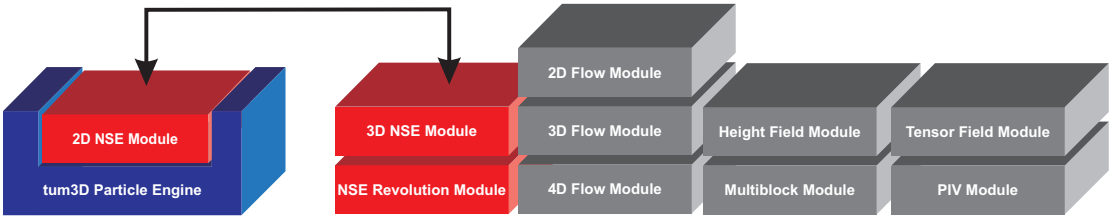


Figure 7.8: This image depicts the modular design of the particle engine. The basis particle engine takes plug-ins depending on what project type is loaded; in the scope of this thesis only the two Navier-Stokes modules for linear and spherical extrusion (both red) are of direct interest. Nonetheless it is worth noting that due to the flexible design of the basis engine many other plug-ins have been developed so far (grey boxes).

The strength of the particle engine is on the one hand its massive GPU acceleration that allows for the interactive visualization of flow fields with a huge amount of particles. Such massive amounts of primitives usually require seconds to minutes to compute a single frame on the CPU. On the other hand, the core module of the particle engine offers a vast amount of visualization features such as lines, ribbons, oriented particles, etc. (see Figure 7.9 and 7.13), which are applicable to any plug-in written to run inside the engine.

Figure 7.9: Particle Engine UI Overview. The images depict the controls for a wide variety of visualization modes.

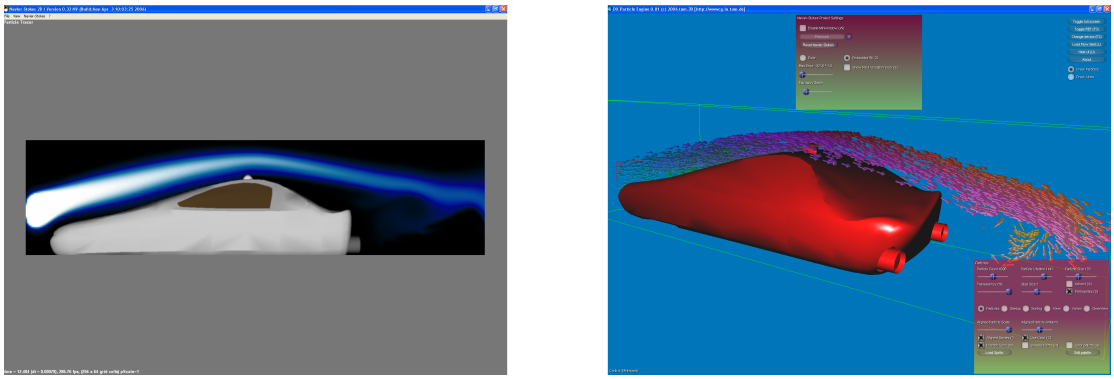


Figure 7.10: The left screenshot shows the design view in the "NavierStokes2D". The right screenshot depicts the same project in the particle engine; the flow is visualized using oriented point sprites (see Section 5.3.5).

The fluid simulation algorithms are integrated into the engine as such plug-ins. In the scope of this thesis three modules are of particular interest. The linear extrusion *2D Navier-Stokes Simulation Module*, the *NSE Revolution Module* and finally the full 3D Navier-Stokes Simulation. The first module takes a single Navier-Stokes simulation project, as designed in the "NavierStoke2D" tool. This simulation,

which results in a two-dimensional flow field is extended into the third dimension by linear extrusion. In this pseudo 3D domain the particle engine performs the tracing of particles. This results in planar motion of every particles as the flow values fetched from the single simulation have no z-component. Nonetheless, for certain effects this may be an acceptable restriction. In particular in simulations where the obstacle hardly changes in one dimension, e.g., cars in computer games, this simplification still leads to convincing results (see left on Figure 7.10). The same Figure 7.10 depicts the work-flow to create a 3D effect. At first the obstacle-grid (left) is drawn or imported from an image of the obstacle. Next the flow parameters such as viscosity and external forces are set and finally the project is loaded into the particle engine. The engine in turn activates the same Navier-Stokes plug-in to simulate the flow field as it was used in the "NavierStokes2D" design tool. With the 2D simulation, which is extended to 3D, the user has access to all the visualization modes of the basis particle engine, including oriented point sprites, stream lines, stream balls etc.

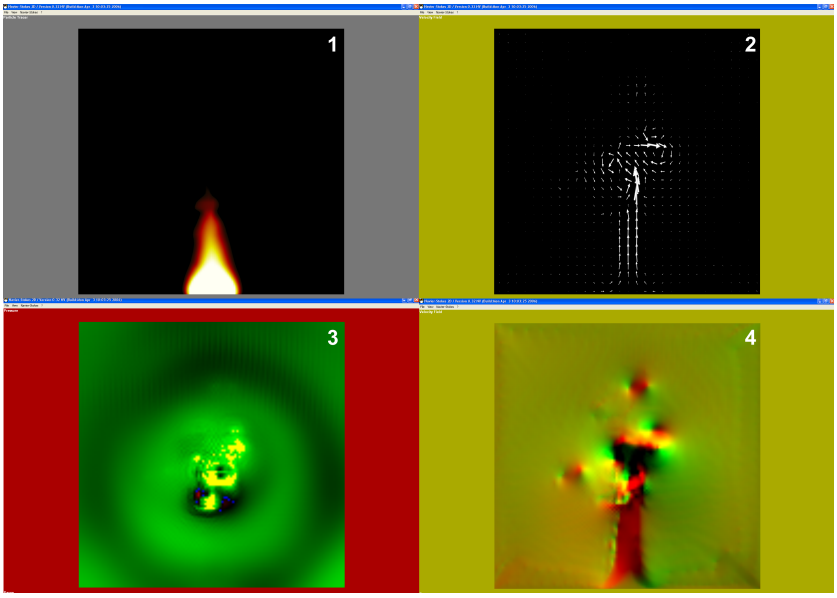


Figure 7.11: Flow design for the cigarette smoke demo. With a heat source on the bottom and pressure templates above the flame, a "pseudo-turbulent" fire flow is modeled. The four screenshots show the dye-advection view (1), the glyph-based flow view (2), the pressure view (3), and the color coded flow view (4).

The second Navier-Stokes module—the NSE revolution project—implements the concepts presented in Sections 4.4 and 5.3.2. For such a revolution one or multiple 2D simulations are designed, and stored as NSE files together with a revolution file containing the list of associated Navier-Stokes projects. Figure 7.11 shows one of these Navier-Stokes projects designed for the smoke simulation, running in the "NavierStokes2D" application. This particular simulation is set to use the "heat-dye" inserted in the bottom of the domain as a source of buoyancy. This creates a strong rising force right above the flame (see image 2 in Figure 7.11). Since a laminar simulation hardly captures the small scale vortices of a hot gas,

a set of pressure templates was inserted to disturb the flow above the flame to create the vortex effects of a shaking flame and swirling smoke. The result can be nicely seen in the left two images, showing the pressure view and the color-coded flow field. In the flow field view the vortices created by the pressure templates are clearly visible. These vortices are later responsible for the fine detail smoke structures in the final frame animation in Figure 7.12.

After the design phase the NSE files are loaded into the particle engine as revolution projects. The particle engine—similar to the Explosion demo—starts multiple instances of the Navier-Stokes fluid solver for the different projects. However, as opposed to the Explosions demo now the vector fields rather then the particle traces are used for the revolution. In this way a pseudo 3D flow field is generated. Note that we never actually convert the 2D slices into a 3D field but rather compute the revolution only at those positions required by the advection engine.

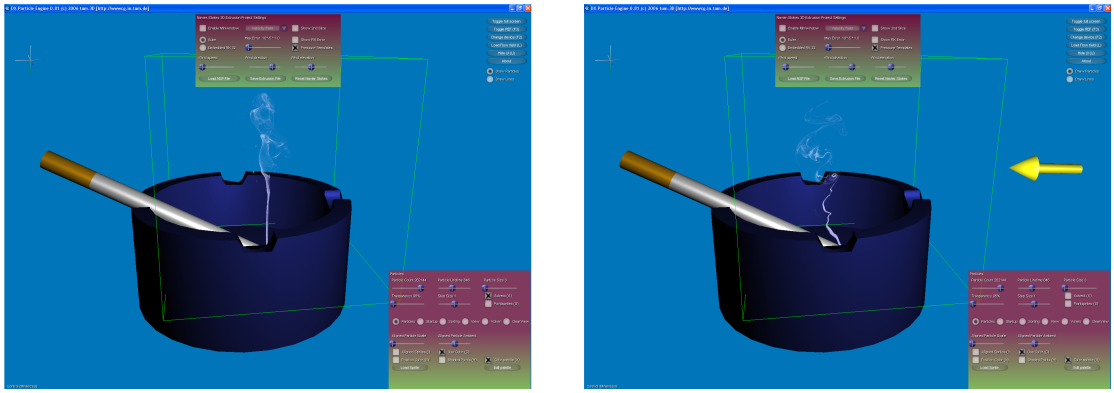


Figure 7.12: Spherical Extrusion in the Particle Engine. The left image shows the extruded data set from Figure 7.11 combined with a custom mesh file for the cigarette. On the right side the influence of an external force is depicted. Such external effects can by arbitrary; e.g., an avatar in a game or a virtual environment, interacting with the smoke.

As with any other particle engine project the user can now use all visualization options provided by the basic module to investigate the flow field. These modules include the rendering of millions of simple point primitives, textured particles (see Figure 7.12 left), different types of streamlines, or oriented sprites. At this point it is important to remember that although the Navier-Stokes simulations are executed only on multiple 2D slices, the particle tracing is done in full 3D allowing the computer game or virtual environment to influence the particle arbitrarily. In the left image in Figure 7.12 such an interaction is shown when the user applies an external wind force to blow the particles away. It is certainly clear that this simulation is not physically correct anymore. Nonetheless, as can be seen in Figures 7.12, plausible visual results are achieved at update rates well beyond 100 fps on current GPUs. Finally, due to the simple design of only one or a few slices the simulation is well controllable, which is much more important than physical accuracy in a computer game or virtual environment.

The final module is the 3D Navier-Stokes simulation. It allows for the interactive simulation and visualization of true 3D flows as described in Section 4.3. Analogously to the “NavierStokes2D” program

for 2D flows, the engine allows the user to control the simulation parameters in real-time, such as border conditions, inflow, viscosity, etc. In the true 3D simulation the many visualization primitives supported by the particle engine are of particular importance to understand the complex flow. Figure 7.13 shows the interactive exploration of the flow with stream ribbons combined with oriented sprites.

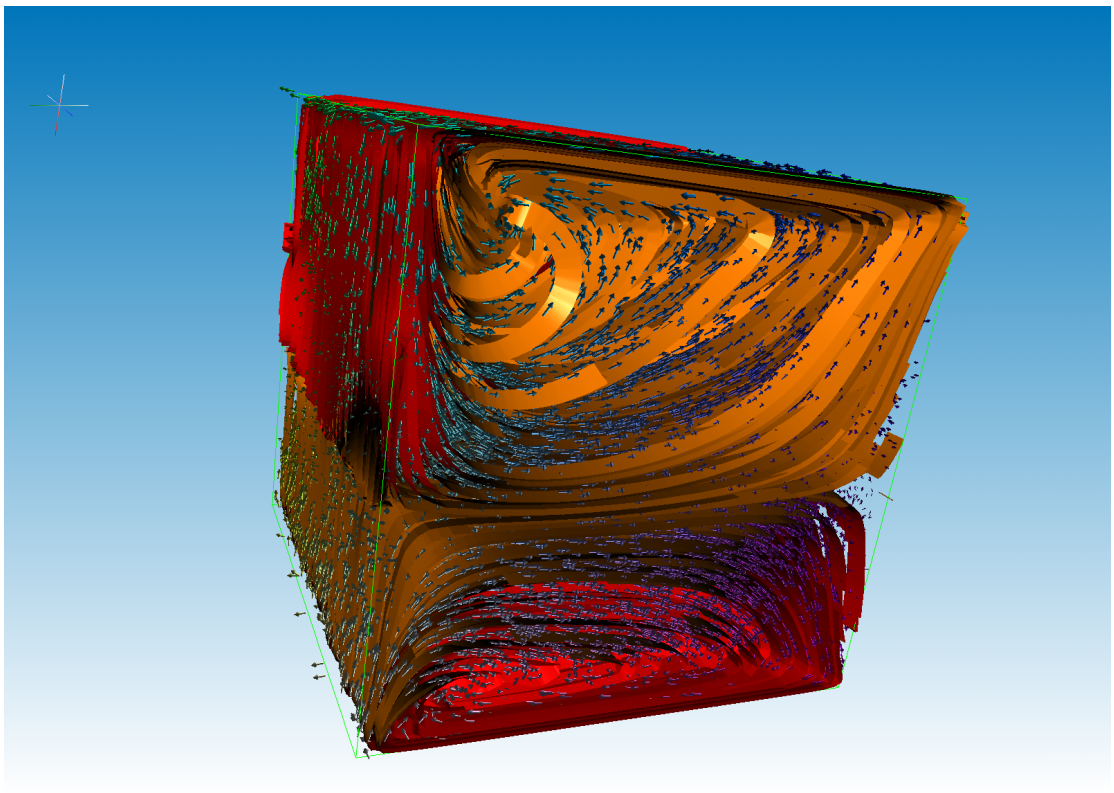


Figure 7.13: A 3D flow simulation with different forces driving on the top and on the bottom of the domain. The flow is visualized with a combination of stream ribbons and oriented particle splats.

Water

The 2D Water demo was initially written as a first test tool for the linear algebra framework and the PDE solver. Figure 7.14 on the left shows the first mode that was implemented. The image depicts the texture containing the height field of the simulation. In this image no additional post-processing was applied. This program is a good example how simple the proposed framework can be utilized in a computer graphics application. Only about ten lines of code are necessary to render the water surface to screen (see Figure 7.14 left). With just one additional cube-map and a slightly more complex pixel shader, realistically looking water surface effects such as shown in Figure 7.14 on the right are possible.

With the availability of vertex texture fetch capabilities it was again the simple water demo that was used to test and perform timings on the vertex displacement features of the new hardware (see

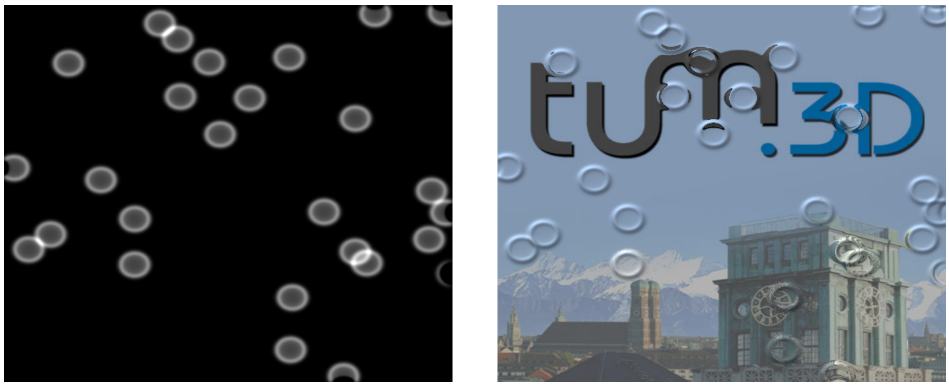


Figure 7.14: The 2D water demo. In the left image the result vector was rendered directly; thus, positive height values appear brighter. In the right image these height values were used to compute the refraction vector and this vector was used for a second texture fetch into a static environment map

Section 5.3.7). The application was extended towards a full 3D demo that allowed to change the point of view or to render the water surface as a surface grid to reveal the underling rendering structure. Again, only a few dozen lines of code were necessary to extend the program to render the water surface as a 3D grid (see Figure 7.15).

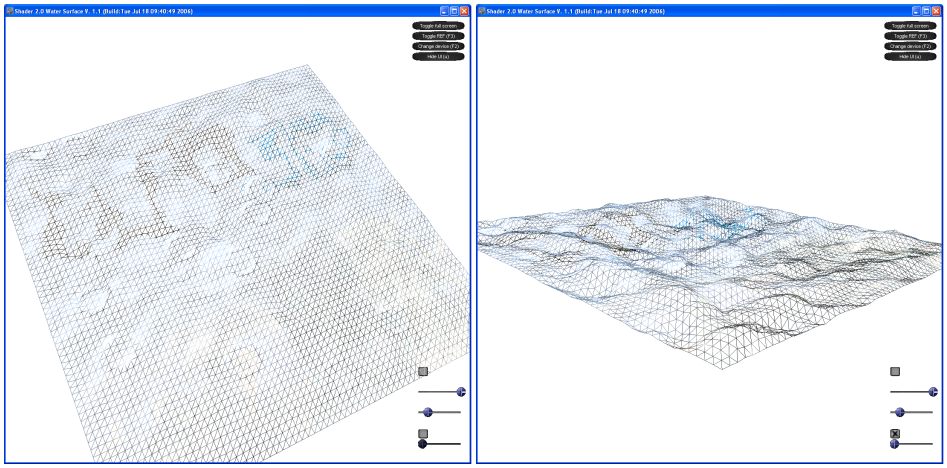


Figure 7.15: The 3D Water Demo. This program extends the functionality of the previous demo in such a way that a static triangle grid is displaced by the simulated height values.

Caustics

The Caustics application is designed to serve two purposes. First, it shows the capabilities of our screen-space accurate photon tracing algorithm. Second, it is a demonstration how easy the GPU-based fluid simulations can be integrated into real-world applications. This integration includes the complex global two-way interactions between the simulation and the surrounding scene. Figure 7.16 shows the caus-

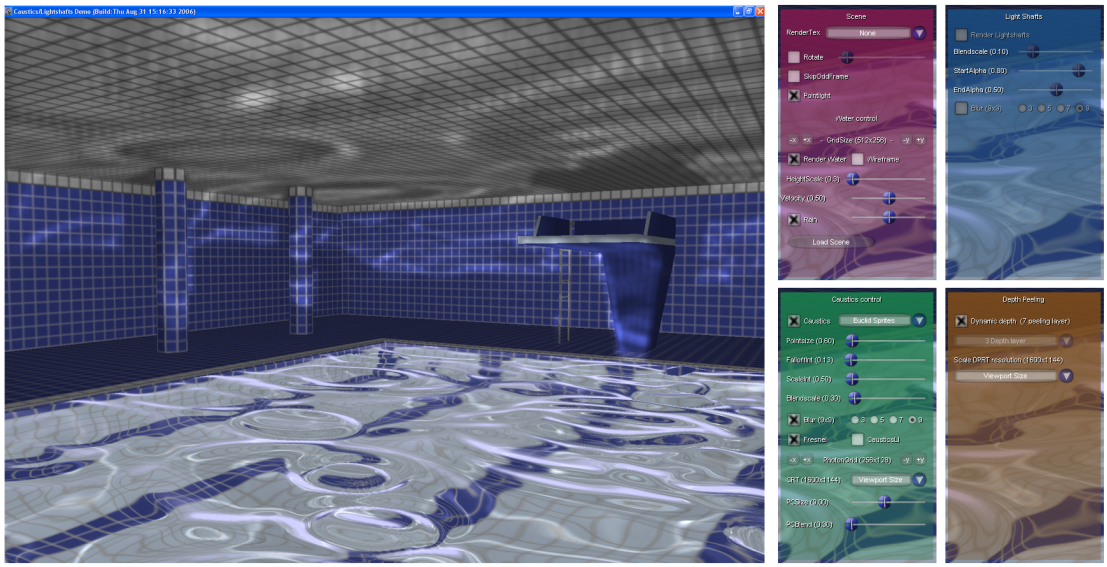


Figure 7.16: The Photon Tracing demo application with the pool project being loaded. On the right screenshots of the parameter UI are shown.

tics application together with its extensive UI to demonstrate the effects of changing the simulation and photon-tracing parameters. For the simulation and water surface rendering, the program uses the displacement mapped water surface core as shown above. The caustic subsystem takes the height-field generated by the water simulation and generates a normal-map which in turn is used to compute the refracted and reflected light rays. These rays are used to generate the photon/scene intersections.

7.2 Conclusion

In this work, we have described a general framework for the interactive simulation and rendering of fluid effects. For this purpose, we have first developed a framework for the implementation of numerical simulation techniques on graphics hardware. Therefore, we have presented efficient internal layouts for vectors and matrices. By considering matrices as a set of diagonal or column vectors and by representing vectors as 2D texture maps, matrix-vector and vector-vector operations could be accelerated considerably compared to software-based approaches.

Our emphasis was on providing the building blocks for the design of general techniques of numerical computing. This is in contrast to existing approaches, where dedicated, mainly explicit solution methods have been proposed. In this respect, for the simulation of particular phenomena some of these approaches might be superior to ours in terms of performance. On the other hand, our framework offers the flexibility to implement arbitrary explicit or implicit schemes, and it can thus be used in applications where larger step sizes and stability are of particular interest. Furthermore, because our internal matrix layout can benefit from the sparsity of columns quite efficiently, we do not expect our approach to be significantly

slower compared to customized explicit schemes.

On top of this framework, in a BLAS/LAPACK like manner, we have described a GPU implementation of the conjugate-gradient method. This functionality in turn was used to numerically solve the 2D wave equation and the incompressible Navier-Stokes equations in 2D and 3D. In all examples, implicit schemes were employed to allow for stable computations, yet providing interactive frame rates. As our timings clearly indicate, the approach developed in this thesis achieves considerably better performance compared to software realizations. With the data remaining on the GPU during the entire simulation and rendering pipeline, for the first time ever it is now possible to compute 2D and 3D simulations at reasonable accuracy and resolution interactively on a single processor PC.

Using the simulation framework we have described techniques for modeling *and* rendering volumetric real-world phenomena. By using our GPU simulation engine in combination with a novel approach to control the shape and appearance of simulated structures, interactive visual simulation of fluid effects is now possible. Again the timings show that the proposed techniques have the potential to be integrated into real-time scenarios like computer games or virtual environments.

We did not restrict ourselves to the sole simulation and rendering of the described effects in real-time but did also propose methods to give the designer more control over the effects. We have introduced pressure templates as a new modeling paradigm for fluid flow. Such templates provide an effective means for modeling small-scale features that can be added at almost arbitrary resolution. To give the designer control over the shape, the size and the dynamic behavior of evolving structures, the proposed flow templates can be instantiated with a variety of different parameters. In this way, custom modeling of fluid flow is made possible. Although the flow phenomena created by these methods are not necessarily physically correct, they can realistically convey real-world phenomena.

To render the volumetric effects efficiently on the GPU we have proposed novel techniques both for high-quality, and high-performance visualization of scalar volumes as well as vector-valued volumes. This has been achieved by means of GPU-based raycasting and GPU-based particle tracing. In both contributions we have presented novel techniques that resulted in ground-breaking performance gains. On the one hand, our raycasting implementation is still about an order of magnitude faster compared even to the most optimized single CPU raycasting implementations. On the other hand, the particle engine has proven to be even two orders of magnitude faster than CPU-based SIMD solutions. With its unmatched performance and render possibilities the particle engine won both the 2005 and 2006 IEEE Visualization contests [94, 95] which are part of the “IEEE Visualization”, the worlds most renowned conference on visualization. Thus, we have shown that GPU-based solutions are capable not only of outperforming well established commercial solutions but at the same time also offer competitive visual quality.

Finally we considered the integration of our simulations into real-world applications. We have prosed GPU photon tracing in screen-space to enable interactive simulation and rendering of complex light patterns caused by refracting or reflecting objects. The ability to trace large photon sets by rasterizing lines into texture maps in combination with a novel rendering method to account for area energy splats enables visual simulation of caustic effects at high frame rates. In a number of different examples these

statements have been verified. The possibility to integrate caustics shadowing, one or several refractions in the interior of complex objects and intensity splats on curved objects distinguishes the proposed GPU technique from previous approaches and presents an alternative to CPU based ray-tracing

In retrospective, it is interesting to see the overall impact of this work on research and practical applications. When we began our work 2003, the enormous potential of graphics processors for numerical computations were widely underestimated. Today, we see ATI releasing a special API, called CTM*, together with a special line-up of GPUs "FireStream 2U" to particularly support numerical stream processing on GPUs. Havok [44], the premier provider of interactive software and services for digital media creators in the games and movie industries, recently released Havoc FX, a special effects SDK and tool chain that leverages Shader Model 3.0 class GPUs [43]. In terms of academic impact, a whole new field of research—called GPGPU—has emerged from the baseline contributions. Today, numerous scientists (ref. GPGPU.org [41]) focus their attention on the numerical capabilities of GPUs.

In the introduction chapter we promised that "*This dissertation demonstrates how today's graphics processing units can be efficiently used for interactive simulation and rendering of fluid effects in computer games and virtual environments.*" Now, after the explanation of the techniques and the presentation of the results, reassured by the new level of visual quality at interactive frame rates we are convinced that this goal has been achieved.

7.3 Future Work

The work in this dissertation may take several directions. Firstly, we plan to continue the ongoing work on integrating the proposed effects into a game engine and a virtual reality application. Particularly interesting will be the integration of Navier-Stokes-based smoke, fire, and fluid effects in these scenarios. Besides further performance optimizations, one of the most interesting questions is how these effects can be controlled by external forces and modeling paradigms in a given environment. Furthermore, we will extend the concept of flow templates to 3D by integrating volumetric templates into 3D simulations.

Secondly, we will investigate a new avenue of applications for our algorithms—the mobile platform. Although initially designed for the desktop, our software components can prove extremely valuable in these environments. This is due to the fact that handheld GPUs are currently undergoing the same dramatic evolution as the desktop GPUs did during the last ten years. Currently mobile GPUs possess the horsepower of about four year older Shader 2.0 desktop GPUs—our target platform at the beginning of this thesis in 2003. Another interesting fact about mobile hardware is that handheld CPUs have considerably less horsepower than their PC counterparts, in particular these CPUs do not have a numerical coprocessor, thus they entirely lack the ability to efficiently allow for floating point computations making the use of GPUs even more important.

Thirdly, we plan to adopt our work to multi-GPU environments. This includes single PCs with multiple GPUs using technologies such as ATI's CrossFire [52] or NVIDIA's SLI [20], as well as PC

*CTM stands for Close To Metal. In terms of syntax it is based on Microsoft's High Level Shading Language (HLSL). However, CTM allows the developer to use the GPU without initializing DirectX thus making pure GPGPU code, without graphics output, simpler and more portable.

clusters equipped with one ore more GPUs per node. Of particular interest in these environments are data exchange requirements. It may not be possible to keep the data in local GPU memory during the entire pipeline. Therefore, efficient GPU-based selection and compression schemes have to be developed to reduce the bus and network transfer to a minimum.

Fourthly, we plan to integrate other simulation models into our fluid framework such as the Lattice-Boltzmann method (LBM) [45]. Due to our layered architecture a GPU implementation of the LBM will integrate well into our multi-level simulation and rendering framework. With both the Navier-Stokes and the LBM simulation at hand we will be able to perform a detailed comparison of the two methods with the focus on the effects generation for computer games and virtual environments. To our best knowledge such an analysis has not been performed before.

Finally, we are planing to integrate our visualization subsystems, such as the raycaster and the particle engine, into other existing supercomputer-based simulation environments. This includes local image generation, where our rendering software has direct access to the simulation data, as well as remote visualization, where the researcher is connected to the supercomputer via only a narrowband connection and parts of the visualization are to be executed on the server while other parts run on the client side. In this domain the scenario of a handheld client device with a mobile GPU could be of particular interest as well.

Bibliography

[1] Kurt Akeley. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA, 1993. ACM Press.

[2] AMD. AMD Athlon 64 X2 Dual-Core Processor for Desktop. http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html, 2006.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] John D. Anderson. *Computational Fluid Dynamics, The Basics with Applications*. McGraw-Hill, Inc, 1995.

[5] The Architecture Review Board (ARB). The EXT_framebuffer_object extension. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt, 2005.

[6] James Arvo. Backward ray tracing. Developments in Ray Tracing, SIGGRAPH 1986 Course Notes, 1986.

[7] ATI. The Radeon X1900 Specifications. http://www.ati.com/products/RadeonX1900/.

[8] Dirk Bartz. volren.org. http://www.volren.org/, 2005.

[9] K.E. Batcher. Sorting networks and their applications. In *Proceedings AFIPS 1968*, 1968.

[10] David Blythe. The direct3d 10 system. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 25(3):724–734, 2006.

[11] P. Bogacki and L. F. Shampine. A 3(2) pair of runge-kutta formulas. *Appl. Math. Lett.*, 2(4):331–325, 1989.

[12] Christian A. Bohn. Kohonen feature mapping through graphics hardware. In *Proceedings of the Joint Conference on Information Sciences*, volume II, pages 64–67, 1998.

[13] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):917–924, July 2003.

[14] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings of Pacific Graphics 2003*, pages 335–343, 2003.

[15] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. GPUBench: Evaluating GPU performance for numerical and scientifc application. In *Proceedings of the ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[16] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[17] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, pages 91–98, 1994.

[18] S. Collins. Adaptive splatting for specular to diffuse light transport. In *Proceedings of the Fifth Eurographics Workshop on Rendering*, pages 119–135, 1994.

[19] Intel Corporation. Open source computer vision library. http://www.intel.com/technology/computing/opencv/index.htm, 2005.

[20] NVIDIA Corporation. What is NVIDIA SLI technology? http://www.slizone.com/page/slizone_learn.html.

[21] NVIDIA Corporation. What is a GPU? http://www.nvidia.com/object/IO_20010528_6691.html, 1999.

[22] NVIDIA Corporation. The GeForce 7900 GTX. http://www.nvidia.com/page/geforce_7900.html, 2006.

[23] Richard Courant, Eugene Isaacson, and Mina Rees. On the solution of nonlinear hyperbolic differential equations by finite differences. *Communications on Pure and Applied Mathematics*, 5:243–255, 1952.

[24] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, 1977.

[25] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[26] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, pages 91–98, 1992.

[27] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[28] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. A set of level 3 basic linear algebra subprograms,. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[29] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001.

[30] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 87–96, 2005.

[31] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[32] Charles L. Feffermann. Existence and smoothness of the navier-stokes equation. http://www.claymath.org/millennium/Navier-Stokes_Equations/, 2000.

[33] Joel Feldman. Derivation of the wave equation. http://www.math.ubc.ca/ feldman/apps/wave.pdf.

[34] J. Freund and K Sloan. Accelerated volume rendering using homogeneous region encoding. In *Proceedings IEEE Visualization '97*, pages 191–197, 1997.

[35] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In *Proceedings of the 18th Symposium on Simulation Technique (ASIM 2005)*, pages 139–144, September 2005.

[36] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, Chicago, United States, June 2006.

[37] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[38] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 111–121, 2004.

[39] S. Guthe, S. Gumhold, and W. Strasser. Interactive visualization of volumetric vector fields using texture based particles. In *Proceedings of WSCG*, volume 10, pages 33–41, 2002.

[40] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and T. Ertl. High-quality unstructured volume rendering on the PC platform. In *ACM Siggraph/Eurographics Hardware Workshop*, 2002.

[41] Mark Harris. General-purpose computation using graphics hardware. http//www.GPGPU.org.

[42] Mark J. Harris. Analysis of error in a CML diffusion operation. Technical Report TR02-015, University of North Carolina, 2002.

[43] Havok. Havok product family. http://www.havok.com/content/view/72/57/.

[44] Havok. Who we are. http://www.havok.com/content/blogcategory/20/37/.

[45] Xiaoyi He and Li-Shi Luo. Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation. *Phys. Rev. E*, 56(6):6811–6817, Dec 1997.

[46] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 119–127, 1984.

[47] Karl E. Hillesland, Sergey Molinov, and Radek Grzeszczuk. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Transactions on Graphics*, 22(3):925–934, July 2003.

[48] Kenneth Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, August 1999.

[49] Kenneth E. Hoff III, Andrew Zaferakis, Ming C. Lin, and Dinesh Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 145–148, March 2001.

[50] Matthias Hopf and Thomas Ertl. Accelerating 3D convolution using graphics hardware. In *IEEE Visualization '99*, pages 471–474, October 1999.

[51] Matthias Hopf and Thomas Ertl. Hardware based wavelet transformations. In *Proceedings of Vision, Modeling, and Visualization*, pages 317–328, 1999.

[52] ATI Technologies Inc. Crossfire FAQ. http://www.ati.com/technology/crossfire/faq.html.

[53] K. Iwasaki, Y. Dobashi, and T. Nishita. A fast rendering method for refractive and reflective caustics due to water surfaces. In *Eurographics*, pages 283–291, 2003.

[54] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. An efficient method for rendering underwater optical effects using graphics hardware. *Computer Graphics Forum*, 21(4):701–711, 2002.

[55] Henrik Wann Jensen. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum*, pages 57–64, 1997.

[56] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, 1995.

[57] James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.

[58] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: A GPU-based particle engine. In *Graphics Hardware 2004*, pages 115–122, August 2004.

[59] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, pages 168–176, 2002.

[60] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In T. Akenine-Möller and M. McCool, editors, *Proceedings Eurographics Graphics Hardware Conference*, pages 123–131. IEEE, 2004.

[61] Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *Proceedings of the 18th Symposium on Simulation Technique*, pages 722–727, September 2005.

[62] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware 2004*, pages 123–132, August 2004.

[63] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. The application of GPU particle tracing to diffusion tensor field visualization. In *IEEE Visualization 2005*, pages 73–78, 2005.

[64] Jens Krüger, Kai Bürger, and Rüdiger Westermann. Interactive screen-space accurate photon tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering—EGSR)*, pages 319–329, June 2006.

[65] Jens Krüger, Peter Kipfer, Polina Kondratieva, and Rüdiger Westermann. A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6), 11 2005.

[66] Jens Krüger, Thomas Schiwietz, Peter Kipfer, and Rüdiger Westermann. Numerical simulations on PC graphics hardware. In *ParSim 2004 (Special Session of EuroPVM/MPI 2004)*, 2004.

[67] Jens Krüger, Jens Schneider, and Rüdiger Westermann. ClearView: An interactive context preserving hotspot visualization technique. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5), September-October 2006.

[68] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[69] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH 2003)*, 22(3):908–916, 2003.

[70] Jens Krüger and Rüdiger Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3):685–693, 2005.

[71] Jens Krüger and Rüdiger Westermann. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 44 A GPU Framework for Solving Systems of Linear Equations, pages 703–718. Addison-Wesley, 2005.

[72] B. D. Larsen and N. Christensen. Simulating photon mapping for real-time applications. In Alexander Keller Henrik Wann Jensen, editor, *Eurographics Symposium on Rendering*, 2004.

[73] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, volume 24, pages 327–335, August 1990.

[74] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[75] Gerd Marmitt, Andreas Kleer, Heiko Friedrich, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In Bernd Girod, Marcus Magnor, and Hans-Peter Seidel, editors, *Vision, modeling, and visualization 2004 (VMV-04)*, pages 429–435, Stanford, USA, 2004. Akademische Verlagsgesellschaft Aka.

[76] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, August 2004.

[77] M. Meiner, U. Hoffmann, and W. Straer. Enabling classification and shading for 3d texture mapping based volume rendering using OpenGL and extensions. In *IEEE Visualization '99*, pages 110–119, 1999.

[78] Microsoft. DirectX developer center. http://msdn.microsoft.com/directx/.

[79] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinitereality: a real-time graphics system. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 293–302, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[80] Tomoyuki Nishita and Eihachiro Nakamae. Method of displaying optical effects within water using accumulation buffer. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 373–379, 1994.

[81] OpenGL.org. OpenGL & OpenGL utility specifications. http://www.opengl.org/documentation/specs/.

[82] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[83] S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics: Interactive 3D*, pages 119–126, 1999.

[84] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[85] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C++ : The Art of Scientific Computing*. Cambridge University Press, 2002.

[86] T. Purcell, C. Donner, M. Cammarano, H.W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 41–50, 2003.

[87] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, 2002.

[88] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50, 2003.

[89] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. *ACM Transactions on Graphics*, 22(3):703–707, 2003.

[90] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and Ertl. T. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–119, 2000.

[91] Martin Rumpf and Robert Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84, 2001.

[92] Thorsten Scheuermann. Render to Vertex Buffer with D3D9, SIGGRPAH 2006 Course Notes. http://www.ati.com/developer/siggraph06/SIGGRAPH06_ShadingCourse_Scheuermann.pdf, 2006.

[93] Thomas Schiwietz and Rüdiger Westermann. GPU-PIV. In *Vision, Modeling and Visualization 2004*, 2004.

[94] Jens Schneider, Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. All you need is ... – Particles!, IEEE Visualization contest 2005 first price winning contest entry. http://wwwcg.in.tum.de/Research/Projects/VisContest05.

[95] Jens Schneider, Jens Krüger, Kai Bürger, and Rüdiger Westermann. California Streaming, IEEE Visualization contest 2006 first price winning contest entry. http://wwwcg.in.tum.de/Research/Projects/VisContest06.

[96] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998.

[97] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *Transactions on Visualization and Computer Graphics (TVCG)*, to appear.

[98] L.M. Sobierajski. *Global Illumination Models for Volume Rendering*. PhD thesis, The State University of New York at Stony Brook, August 1994. Disertation.

[99] Jos Stam. Stable fluids. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 121–128, Los Angeles, 1999. Addison Wesley Longman.

[100] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. In *ACM Computer Graphics (Proceedings SIGGRAPH '93)*, pages 369–376, 1993.

[101] Robert Strzodka. Virtual 16 bit precise operations on RGBA8 textures. In *Proceedings of Vision, Modeling, and Visualization*, pages 171–178, 2002.

[102] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 93–102, 1997.

[103] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3), 2005.

[104] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, 2002.

[105] Chris Trendall and A. James Stewart. General calculations using graphics hardware, with applications to interactive caustics. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 287–298, June 2000.

[106] Adrien Treuille, Andrew Lewis, and Zoran Popovic. Model reduction for real-time fluids. *ACM Trans. Graph.*, 25(3):826–834, 2006.

[107] United States National Library of Medicine. The visible human project. http://www.nlm.nih.gov/research/visible/visible_human.html, 1986.

[108] Eric Veach and Leonidas J. Guibas. Metropolis light transport. *Computer Graphics*, 31(Annual Conference Series):65–76, 1997.

[109] I. Wald, C. Benthin, P. Slusalek, T. Kollig, and A. Keller. Interactive global illumination using fast ray tracing. In *Eurographics Rendering Workshop*, pages 15–24, 2002.

[110] Michael Wand and Wolfgang Straßer. Real-time caustics. In P. Brunet and D. Fellner, editors, *Computer Graphics Forum*, volume 22(3), pages 611–620, 2003.

[111] Mark Watt. Light-water interaction using backward beam tracing. *SIGGRAPH Comput. Graph.*, 24(4):377–385, 1990.

[112] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 291–294, 1998.

[113] Wikipedia, the free encyclopedia. OpenGL. http://en.wikipedia.org/wiki/OpenGL.

[114] Chris Wyman. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053, 2005.

[115] Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE*, pages 205–211, 2005.

[116] Chris Wyman and Scott Davis. Interactive image-space techniques for approximating caustics. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 153–160, New York, NY, USA, 2006. ACM Press.

[117] Chris Wyman, Charles Hansen, and Peter Shirley. Interactive caustics using local precomputed irradiance. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 143–151, 2004.

[118] R. Yagel and Z. Shi. Accelerated Volume Animation by Space-Leaping. In *Proceedings IEEE Visualization '93*, pages 62–69, 1993.

Index