UNIVERSITY OF OSLO
Department of Informatics

Real-time simulation of the incompressible Navier-Stokes equations on the GPU

Master's thesis

Lars Moastuen

July 2007

Abstract

This thesis explores the possibility of using graphics processing units (GPUs) to compute approximations of the Navier-Stokes equations for usage in real-time simulation. The Navier-Stokes equations describe the change in mass and momentum of fluids, e.g., liquids or gases. Typical applications for such a simulator can be computer games to increase realism by simulating, e.g., clouds or smoke that interact with the environment. A new, efficient solver that supports arbitrary boundaries is presented. Support for objects following the flow whilst acting as obstacles themselves is also discussed. The simulation is performed solely on the GPU, and offers high quality simulations at a modest cost.

Contents

Abstract **i**

Contents **iii**

1 Introduction **1**

1.1 Research questions 1

1.2 Organization of the thesis 3

2 The GPU architecture **5**

2.1 The graphics pipeline 5

2.2 Technical introduction 9

2.2.1 Memory model 9

2.2.2 Performance 11

2.2.3 Branching 13

2.2.4 IEEE 754 floating-point arithmetic 14

2.3 The OpenGL shading language 15

2.4 NVIDIA GeForce 8800 17

3 Computation fluid dynamics on the GPU **21**

3.1 Real-time simulators 22

3.2 Offline rendering algorithms 25

3.3 The future of GPU physics 26

4 Incompressible Navier-Stokes **29**
4.1 Derivation 29
4.2 The incompressible Navier-Stokes equations 33
4.3 Semi-Lagrange discretization 35
4.4 Vorticity confinement 39
4.5 Transport of quantities 44
4.6 Implementation 46

5 Solving the Poisson equations **49**
5.1 The Jacobi iterative solver 51
5.2 The successive overrelaxation solver 53
5.3 Convergence of SOR and Jacobi 55
5.4 A GPU-optimized SOR solver 58
5.5 Results 65
5.5.1 Comparison between Jacobi and SOR 66
5.5.2 Analysis of the optimized SOR solver 68
5.6 Other solvers 69

6 Boundary conditions **71**
6.1 Boundaries of arbitrary shape 71
6.2 Types of boundary conditions 77
6.3 Discrete boundary conditions 80
6.3.1 Discrete no-slip boundary conditions 80
6.3.2 Discrete freeslip boundary conditions 81
6.3.3 Discrete outflow boundary conditions 81
6.3.4 Discrete inflow boundary conditions 82
6.3.5 Pure Neumann boundary conditions 83
6.4 Non-stationary obstacles 86
6.4.1 Representation 86
6.4.2 Simulation of movement 87

6.4.3 Avoiding stationary obstacles 90
6.4.4 Visualization and updating the boundaries . . 92
6.4.5 Results 93

7 Summary and results **95**
7.1 Discussion 97
7.2 Future work 99

Bibliography **103**

Preface

This thesis has been written at SINTEF ICT under the GPGPU project. It has been a lovely place to work, not many students get 24 inch flatscreens and an office view a view.

This thesis could not have been written without the assistance from many people. My first thanks goes to my supervisors Knut-Andreas Lie and Trond Runar Hagen for giving me the opportunity to write my thesis at SINTEF ICT and for valuable assistance, tips and opinions. Apologies to Knut-Andreas for all the red pens he has depleted during the draft reading.

The days would have been very long if there was no fellow students around. Thanks to Trygve Fladby, Thomas Lunde, Hanne Moen and Martin Lilleeng Sætra at SINTEF. Especially thanks to André Brodtkorb for (over-) excessive draft reading, valuable opinions and coffee. Also thanks to former fellow students, including Gaute Jørgensen, Guo Wei Ma, Eirik Munthe and Kristoffer "Litjkarn" Skaret.

If if had not been for my friends, I would probably been crazy by now from applying Navier-Stokes to the everyday things, such as skies and smoke. Thanks to everyone that have not contributed to the thesis in any way, you have kept me sane while working on it. Last, but not least, thanks to my family.

Chapter 1

Introduction

This thesis explores the use of graphics processing units (GPUs) to numerically solve the incompressible Navier-Stokes equations. Visual appearance and efficiency are the two main areas of focus, with physical accuracy as a secondary target. Such simulations could potentially be interesting in games to simulate phenomena such as smoke, clouds, etc. The demand for realistic physics has increased as graphics has become more and more photorealistic. The highly parallel nature, the potential floating-point performance and the recent shader programmability makes the GPU an interesting architecture for such applications.

As GPUs are evolving very rapidly, it is important to state that the work done in this thesis is based on the NVIDIA NV70 architecture. More specifically, the simulator has been implemented and tested on a Dell Optiplex GX620. The machine is equipped with a Pentium 4 3.00GHz processor, 2 GB of RAM and an NVIDIA 7800GT graphics card.

1.1 Research questions

There exists a lot of research within the field of computation fluid dynamics (CFD). Since the mid 1990s there has been an increase in focus on visual appearance from parts of the CFD community. Methods for simulations with high visual quality for use in graphics applications have emerged. Recently, graphics processing units have been used to accelerate the CFD simulations, in contrast to traditional approaches where the CPU is the main computational engine.

Lately, GPUs have been utilized for CFD applications, resulting in high performance simulations. This thesis is written with the following questions in mind:

1. Is it plausible that GPUs will be used to handle physics computations in the future, and more specifically, fluid dynamics simulations?

2. How can solutions of the Poisson equations that arise from the semi-Lagrange discretization of the Navier-Stokes equations be computed efficiently on the GPU?

3. Can arbitrary stationary and non-stationary obstacles be incorporated in a real-time simulator for the incompressible Navier-Stokes equations?

The first question asks if GPUs will be used for other purposes than rendering in the future, especially physics. GPUs have been used to solve a large number of problems already. However, the use of GPUs for non-rendering tasks in games remains a relatively unexplored territory. Lately, there has been an increased focus on physically plausible effects in games and dedicated physics hardware has emerged. A common approach to fluid dynamics in games has been to use pre-rendered animations that allow little or no interaction with the environment. Fluid dynamics simulation can be used to improve interaction with phenomena such as smoke and clouds. The question if such simulations can be performed on the GPU while allowing high quality rendering remains open.

By using operator splitting, Lagrangian description of one of the terms in the Navier-Stokes equations and implicit methods, an unconditionally stable solver with respect to the size of the time steps can be developed. This discretization is called a semi-Lagrange description of the Navier-Stokes equations. When discretizing the incompressible Navier-Stokes equations, two Poisson equations appear. Many methods exist to efficiently compute solutions of these equations. These terms are bottlenecks in the simulation and it is crucial that the equations are solved in a very efficient manner for the simulator to be used in real-time simulations.

For the simulator to support interaction with the environment, it is important that arbitrary obstacles are supported. The mechanism for handling boundaries needs to be very efficient, as boundaries must be applied several times per time step when using an operator splitting technique to solve the incompressible Navier-Stokes equations. It is also desirable that several types of boundaries are supported. For many applications, support for obstacles that advect through the flow is also desirable. This can typically be flakes of ashes in a smoke simulation, or floating objects that follows a liquid flow.

1.2 Organization of the thesis

The thesis is organized around the questions introduced in the previous section. In the next chapter, some background material about the technical properties of the GPU, and how they can be utilized for general purposes, is presented. Chapter 3 examines some of the problems that have been solved using GPUs, and introduces some of the most important research presented within the CFD research field. Especially is research concerning techniques to simulate fluids for visual purposes mentioned here.

In Chapter 4 the incompressible Navier-Stokes equations are reviewed. The equations are derived from conservation of momentum and mass. The result is discretized using the semi-Lagrange scheme presented. A technique for reinserting lost energy as a result of numerical dissipation is also reviewed.

Chapter 5 discusses how the Poisson equations that emerge when discretizing the incompressible Navier-Stokes equations can be solved efficiently. A Jacobi iterative solver and an optimized successive overrelaxation iterative solver are presented. Their performance and convergence properties are also discussed.

Boundaries are discussed in Chapter 6. Here a technique for approximating obstacles of arbitrary shape using piecewise linear segments is discussed. This approximation is used in an efficient algorithm for applying arbitrary boundaries. The algorithm supports several types of boundary conditions. Last, but not least, the chapter introduces techniques for simulation of a large amount of obstacles that follows the flow.

In Chapter 7 some important results are reviewed. The research questions are discussed in light of the results reviewed. Possible extensions and improvements for future work are presented.

Chapter 2

The GPU architecture

GPUs are specialized hardware designed to offer high render performance at a low cost. The architecture is highly parallel and offers extremely high floating-point performance. However, as the hardware is highly specialized for rendering 3D graphics, new methods must be developed in order to utilize the full potential. Since the architecture is very different from regular CPUs, it is important to have good knowledge about how the hardware is designed and what its strengths and weaknesses are.

This chapter reviews how GPUs work and discusses how the architecture can be utilized for general purpose computation, a field commonly called GPGPU. The first section gives a basic introduction to the OpenGL pipeline. Then some of the technical features of the GPU, and the impact these features has for general purpose utilization of GPUs is discussed. At the end of the chapter, the new NVIDIA GeForce 8800 series is mentioned briefly.

This chapter is written mainly with the NVIDIA GeForce 7800 architecture in mind. Most of the discussion also applies to the GeForce 6800 and the ATI X1900-series, but there may be some minor differences.

2.1 The graphics pipeline

Today there are two major graphics APIs in use, OpenGL [SWND05] and Direct3D [Mic07]. The APIs are specialized in rendering real-

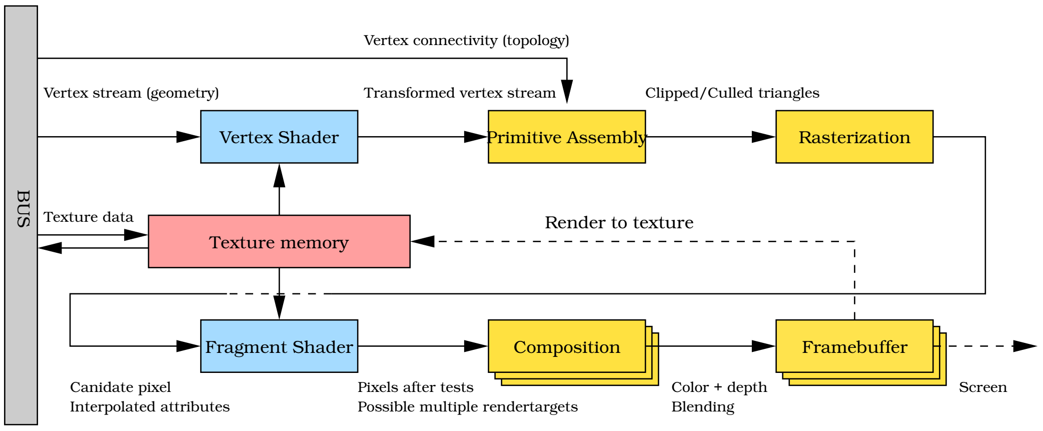


Figure 2.1: The graphics pipeline. Rendering is initiated by specifying geometry and topology, textures and transformations (called "the state"). The vertex transformation step and fragment transformation (blue) step are programmable on current hardware.

time 3D graphics and utilize native hardware in doing so. OpenGL is used in this thesis, and the discussion will therefore not necessarily apply to Direct3D, even though the two APIs are similar in most cases.

In OpenGL, 3D graphics consists of vertices (geometry) that form triangles, quads and lines (topology). To achieve a result of high visual quality, lights are placed, textures applied and colors, normals and material properties specified (among other things). Custom programs may be applied to each vertex and candidate output pixel to achieve custom effects. In order to render a scene to screen, a series of transformations must be applied to transform and clip the primitives from world space to screen space. The triangles are rasterized to form candidate pixels called fragments that must pass through a series of tests in order to be written to the framebuffer. A simplified presentation of the pipeline is presented in Figure 2.1.

Before discussing the pipeline in detail, some background information about transformations in OpenGL is helpful. Transformations are necessary in order to transform from world space to screen space, and are used to implement cameras, projection, etc. By adding a w-component to the spatial $\langle x, y, z\rangle$-coordinates, affine transformations can be used to perform the transformations required. The advantage of affine transformations over linear transformations is that translatation can be expressed as matrix multiplication:

$$\vec{x} \mapsto \mathbf{A}\vec{x} + \vec{b},$$

where A is a linear transformation and $\vec{b}$ is a translation vector. The affine transformation matrix for this transformation is

$$\mathbf{T} = \begin{bmatrix} \mathbf{A} & \vec{b} \\ 0 & 1 \end{bmatrix}.$$

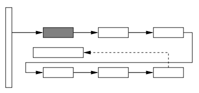
This enables transformations to be specified as matrices, and multiple transformations can be concatenated, or multiplied together:

$$\vec{x} \mapsto \mathbf{T}\vec{x} \quad = \quad \begin{bmatrix} \mathbf{A}\vec{x}_{xyz} + \vec{x}_w\vec{b} \\ \vec{x}_w \end{bmatrix}.$$

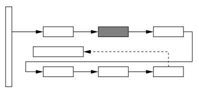
The 3D nature of graphics and the extra w-component, makes GPUs specialized in handling vectors of four components.

Now follows a description of each of the stages in the OpenGL pipeline (see Figure 2.1). The figures in the margin of the paragraphs below refers to the stages in this figure, and marks what stage of the pipeline the paragraph discusses.

Vertex shader. When vertices are sent to the GPU, they are processed by the vertex shader which outputs an intermediate set of coordinates, called clip coordinates. The vertex shader has access to vertex attributes, such as the color, texture coordinates and normals. Custom attributes can also be specified per-vertex and may be outputted for usage later in the pipeline. By default, OpenGL computes lighting in the vertex shader and interpolates the result between vertices in the rasterization stage. A more detailed introduction to OpenGL shaders is found in Section 2.3.



Primitive assembly. The vertex shader has no knowledge about the connectivity between the different vertices. The primitive assembly stage uses the transformed vertex stream as input and applies connectivity (topology) to form primitives, such as triangles, quads and lines. The primitives are clipped and culled to fit into screen space before the rasterizer processes the triangles. Culling is a process to remove back-facing triangles and clipping removes primitives outside the view.



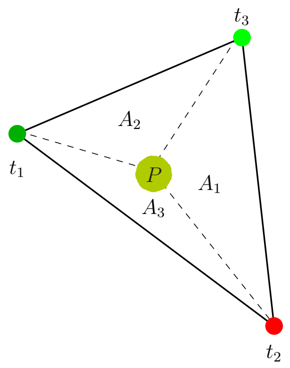
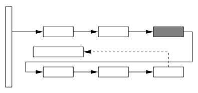
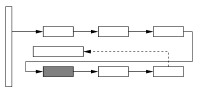


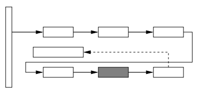
Figure 2.2: Interpolation using barycentric coordinates. Interpolated output is calculated using the homogenous barycentric coordinate $P =< A_1, A_2, A_3 >$. A_1, A_2 and A_3 are the normalized areas ($A_1 + A_2 + A_3 = 1$) to the opposite of the vertices t_1, t_2 and t_3 respectively.



Rasterization. Rasterization is the processes of creating candidate pixels, called fragments, from the primitives. The rasterizer also interpolates vertex attributes (normal, color, texture coordinates and custom attributes among others) and passes them along with the fragments. The OpenGL specification requires that interpolation behaves as if interpolated using barycentric coordinates [MB05], where the resulting value is a weighted average of the three vertices of the primitive (see Figure 2.2).

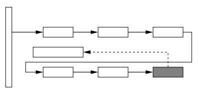


Fragment shader. The fragment shader uses the interpolated values as input and discards the fragment or outputs at minimum a vector containing the color of the fragment, but can also output several vectors using so-called multiple render targets. Both the NVIDIA GeForce 6- and 7-series support four render targets [NVI05], making it possible to output up to a total of 16 scalars in addition to fragment depth, which is a special buffer used for depth-tests.

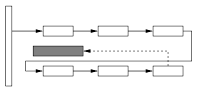


Composition. If the fragment shader does not discard the fragment, it will pass through a series of tests including depth-, stencil- and alpha-test (when the tests are enabled). Fragments that pass all tests are processed by the composition-step. Here fog is added and blending between new and current framebuffer values is applied if enabled.

Framebuffer. The result of the composition is stored in the framebuffer. In addition to the color buffer, the framebuffer consists of several other buffers including depth, stencil and an accumulation buffer. The depth and stencil buffers are used as reference when performing depth and stencil tests, respectively. The accumulation buffer is a special buffer used to accumulate intermediate rendering results.



Render to texture. Instead of outputting the result directly to the screen it is possible to store the result as a texture. For graphic purposes this can typically be used to render scenes with mirrors. For this purpose, the scene is rendered from the mirrors point-of-view and stored to texture. The stored texture is then used as a texture when rendering the mirror in the next pass. For GPGPU purposes this serves as storage of the result and intermediate storage in multi-pass algorithms.



2.2 Technical introduction

The GPU architecture differs radically from the CPU architecture. Different programming techniques are used, and care must be taken to exploit the full potential of the GPU. This section aims to inform about some of the pitfalls related to GPGPU and to mention some of the most important restrictions of the GPU architecture.

2.2.1 Memory model

The memory model of the GPU differs quite a lot from the memory model of the CPU. NVIDIA 7800 GPUs uses a 256 bit memory interface [NVIb] to achieve high memory bandwidth. Cache layout differs drastically between the two architectures, as GPU cache is based on a 2D layout (see Figure 2.3) while the CPU cache is based on a 1D layout. This is done to suit the common usage of the GPUs to apply 2D textures to primitives. The vendors keep the size and layout of the GPU cache secret, but measurements have indicated that the cache of each fragment processor may be of size 8x8 [GLGM06]. The restricted size is a result of the high cost of

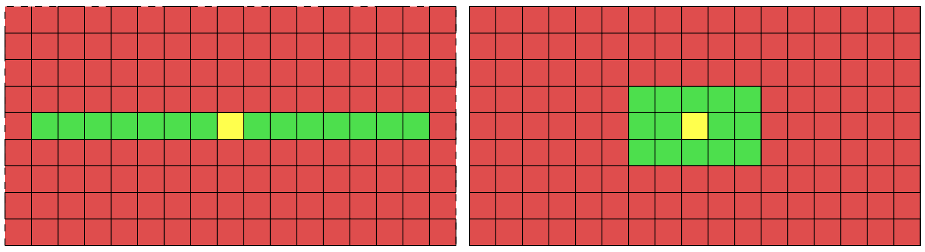


Figure 2.3: Cache layout for CPUs (left) and GPUs (right). Memory is cached row-wise on CPUs (1D) and in the neighbouring area on GPUs (2D) when using two dimensional arrays.

cache, and the common usage of textures, e.g., to do texture filtering.

The processing units of GPUs differ radically from CPUs. Both CPUs and GPUs utilize a technique called instruction pipelining to increase instruction throughput by dividing the execution of an instruction into several stages. Intel Pentium 4 has an instruction pipeline that consists of about 20-30 stages [Sto04, HSU$^+$01]. GPUs have even more instruction pipeline stages. The GeForce 7 series has more than 200 stages [NVIc]. This results in very high theoretical performance, but the penalty of flushing the pipeline (e.g., at branches) is high. Even though memory read latency is hidden through pipelining, cache efficiency is important to avoid stalls when waiting for memory reads [GLGM06]. Since the memory bandwidth is higher on GPUs than on CPUs, the penalty of a cache miss is less on the GPU than on the CPU. However, due to the limited cache size, the cache miss ratio will often be higher on the GPU.

There are two conceptual ways to work with memory. Gather corresponds to the operation `c = a[i]`, or in other words, that the entire memory can be read. Memory scatter corresponds to the operation `a[i] = c` that makes it possible to write to the entire memory area. Regular CPUs are capable of both memory gather and scatter, while GPUs are less flexible. Fragment shaders are capable of fetching data anywhere from a texture, thereby making it capable of memory gather. However, the position of the output is determined before the fragment is processed, making the fragment shader incapable of memory scatter. Vertex shaders have texture read capabilities and are capable of changing the position of the vertices. The vertex shader is therefore both capable of memory gather and scatter. However, vertex scatter can lead to memory and rasterization coherence issues further down in the pipeline. In order to achieve scatter, programmers instead have to utilize various tricks. These include rewriting the program in terms of gather, or by tagging the output with some output address and later sorting the data based on this tag (address sorting) [Buc05b].

2.2.2 Performance

Modern GPUs offer very high floating-point performance compared to CPUs. NVIDIA 7800 offers 165 GFLOPS performance and the new NVIDIA 8800 offers performance up to 520 GFLOPS. The Intel Pentium 4 Extreme Edition offers up to 24.6 GFLOPS. The theoretical maximum performance of GPUs is rarely experienced in practice as cache is very limited and very high arithmetic intensity is required to keep the processors busy at all times. Arithmetic intensity is a measurement on the number of operations necessary to perform per memory fetch in order to keep the processor busy. Table 2.1 lists a summary of floating point performance and memory bandwidth for different architectures. All GPU bandwidth figures are collected from GPUBench results [GPU]. GFLOPS figures for GPUs are collected from Owens et al. [OLG$^+$05, OLG$^+$07], and NVIDIA [NVIc] (note that 520 GFLOPS for NVIDIA 8800 is a theoretical peak). Intel Pentium 4 Extreme is an dual-core 3 GHz processor, and figures are collected from Geer [Gee05] and Hinton et al. [HSU$^+$01].

Modern GPUs are connected through a PCI Express slot. This standard comes in different variants and the transfer rate is determined by the number of lanes available. PCIe x16, supported by modern GPUs, can transfer up to 4.0 GB/s in each direction simultaneously [Mic04]. Even though the transfer rate has improved tremendously, it is still a bottleneck when utilizing the GPU for general purposes. Reading and writing to the GPU memory should be kept to a minimum and problems that are to be solved on the GPU should be of sufficient complexity such that gain in speed is not lost by the overhead of data transfer. Texture formats must also be taken into consideration as some formats require conversion from/to native format by the graphics driver. The native formats

Table 2.1: Single-point floating performance versus memory bandwidth. Note that these figures are disputed and that many different figures appear in different articles. However, the essence is that GPUs offer higher theoretical performance than CPUs.

Architecture	GFLOPS	Cache BW (GB/s)	Sequential BW (GB/s)
ATI X800 XT	63.7	31	17
ATI X1900 XTX	240	39	27
NVIDIA 6800 Ultra	53	21	9.5
NVIDIA 7800 GTX	165	48	19
NVIDIA 8800 GTX	520	140	55
Pentium 4 Extreme	24.6	48.0	6.4

vary with vendors and models [Buc05b].

GPUs are highly parallel stream processors, typically with 16, 24 or even more fragment-shading pipelines and six or more vertex pipelines, suitable for processing large data streams. Each pipeline is a powerful floating-point arithmetic processor, and on most hardware each of them can process up to four scalars simultaneously as they are designed to work with vectors of length four. This requires care when implementing algorithms as various tricks must be utilized to pack data in a manner that allows for maximum utilization of the hardware. The advantage of a parallel processor is that when the number of pipelines is doubled, the performance is doubled (in theory). Regular CPUs are based on the von-Neumann architecture, where processing is driven by the instruction sequence. This architecture is very flexible, but yields bad performance for large data blocks. In the data-stream processing-architecture used by GPUs, the processor is first configured using the instructions that should be performed. These data-stream operators are called kernels and can be written in assembly or C-like languages. A number of parallel units then process the data-stream [LBM+05].

Due to the nature of computer graphics, there is typically a lot more pixels in a scene than vertices, so the number of fragment processors exceeds the number of vertex processors. Therefore, most of the computational potential resides in the fragment processors [OLG+05]. In a GPGPU sense, vertex shaders are usually used to compute texture coordinates and other simple computations.

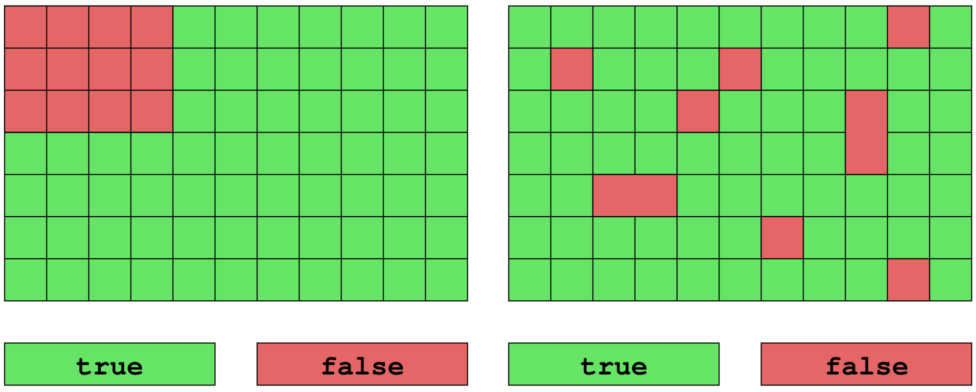


Figure 2.4: Spatially coherent branches (left) execute efficiently, while spatially incoherent branches (right) execute less efficiently as all the fragment processors must execute the same instruction simultaneously.

2.2.3 Branching

The highly parallel nature of GPUs makes their support for branching very limited and demands care in how branching is used. There are two common branch-control mechanisms in parallel architectures. All processors in a single instruction, multiple data (SIMD) architecture execute the same instruction at the same time, while processors in a multiple instructions, multiple data (MIMD) architecture may execute different instructions simultaneously. MIMD branching is the ideal case, while SIMD branching is cost-effective in cases where the branch conditions are spatially coherent (see Figure 2.4). Incoherent branching is very expensive with SIMD. A single processor entering a branch is sufficient to keep the remaining processors idle while waiting for the one processor entering the branch to complete. In older architectures, condition codes (extra bits set to indicate status of various mathematical operations [Wik07c]) were used to emulate branching. With this architecture, both the taken and not taken branch must be evaluated, but only the branch taken is written to registry [HB05]. NVIDIA GeForce 6 and 7 series support MIMD branching in their vertex processors and SIMD in their fragment processors [KF05, NVId].

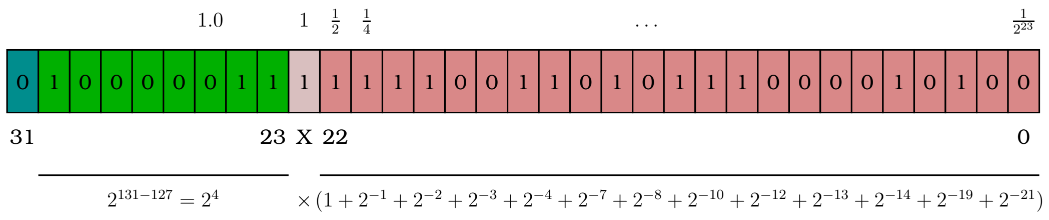


Figure 2.5: Binary 32-bit float representation of 31.21. The number cannot actually be represented exactly and the best approximation is 31.209999.

2.2.4 IEEE 754 floating-point arithmetic

The IEEE standard for binary floating-point arithmetic (IEEE-754) is the most used standard for real number computation. Floating-point representations differ from fixed-point representations in that the location of the decimal point is stored, in this case using an exponent. Fixed-point representation is used for integers, but is rarely used for rationals. In the IEEE-754 standard, floating-point numbers are represented by:

- one sign bit,
- e exponent bits,
- p significand bits

Figure 2.5) shows an illustration of the bits in a 32 bit floating-point number. The significand consists of a "hidden bit" (d_0) that represent $2^0 = 1$, and bits $d_1, \cdots, d_p$ that represents $2^{-1}, 2^{-2}, \cdots, 2^{-p}$. The hidden bit is not actually stored and is always 1 except for some special values such as zero, NaN, and Infinity. The significand is normalized, that is, it is always between 1 and 2 except for the special values zero, NaN, and Infinity. Zero is represented by reserving the value when all the exponent bits are zero [Gol91].

The exponent is biased by $2^{e-1} - 1$ such that exponents are signed values making it possible to represent very small numbers as well as very big ones. The distance between two adjacent floating-point numbers doubles when the exponent is increased by 1. This means that the relative precision is constant, but the absolute precision is highest near zero [Dar03]. The value of a binary floating-point number is computed by

$$\pm\left(d_0 + d_1 2^{-1} + d_2 2^{-2} + \cdots + d_p 2^{-p}\right) 2^{\Sigma}, \tag{2.1}$$

where Σ is the value of the unbiased exponent and the sign is $+$ if the sign bit is zero and $-$ else.

The IEEE 754 standard treats 32 bit (called float or single) and 64 bit (double) floating-point numbers, and is currently under revision. The revised standard, IEEE-754r, will also treat 16 and 128 bit floating point numbers [Wik07a]. GPUs have near complete IEEE-754 single-precision support with a goal to converge to exact behavior within a few years [Bly06]. NVIDIA GPUs also support 16 bit floating-point numbers (often called half). This format should be used over float whenever precision is not crucial as this can increase performance drastically [NVI05]. On NVIDIA, the 32 bit floating-point numbers are represented using a 23 bit significand and a 8 bit exponent in addition to a sign-bit, or in short, `s23e8`, as specified by the IEEE 754 standard. In this format, all integers in range $[-2^{24}, 2^{24}]$ are represented exactly. 16 bit floating-point numbers are represented using 5 bit exponent and 10 bit significand and a sign-bit (`s10e5` in short) [EBL05]. Integers in the range $[-2048, 2048]$ are represented exactly using the half format.

2.3 The OpenGL shading language

The programming languages used by the shaders have developed since the first revision, when assembly language was the only option. Today, several C-like languages have evolved, including C for graphics (often called Cg), by NVIDIA [FK03], HLSL by Microsoft [Mic] and GLSL (OpenGL Shading Language) [Ros06]. High-level languages and interfaces have also evolved, such as Brook [BH], RapidMind [Rap] and PeakStream which has been acquired by Google [Pop07]. For more information on high-level languages, I refer to Buck [Buc05a]. The rest of this section focuses on GLSL.

GLSL closely resembles C, and all C keywords are reserved words in GLSL (although not always in use). The built-in instruction set is very limited and tailored for graphics needs. It contains some functions to access texture memory and some common functions such as `clamp()`, `min()` and `max()`. The last group of functions are trigonometric and other mathematical functions such as `cos()` and `sqrt()`. Most functions are designed to work on scalars or vectors with up to four components, and some functions for calculations using matrixes of size up to 4x4 are available. GPUs lack support for integers and have no bitwise operations such as `and`, `or`, or `not` [Ros06]. No random number generators are available, but continuous quasi-noise functions give the appearance of randomness are available. If “true” random numbers are needed, they are typically generated offline and stored in a texture for lookup by the shaders.

Shaders have support for branching and iterations, but these features should be used with care due to the SIMD nature of the fragment processors. Experiments have shown that loops are restricted to 256 iterations [JS05, Bro07], but consecutive or nested loops can be used to remedy this. Both the vertex and fragment processors have access to texture memory, but the support in vertex processors is partial and depends on the texture format used [NVI05].

Debugging is a major obstacle when developing shaders. Currently no fully qualified debugging environment for shaders exists, and there is no conventional way of debugging through printing to console as there is for CPUs. There is neither no full-fledged development environments for developing shaders, and analyzing shader-performance can be a painstaking process. NVIDIA has released a profiling toolkit named NVPerfKit [NS06] that gives access to a number of performance counters, e.g., workload between vertex and fragment processors and texture lookup latency. This toolkit can help identify bottlenecks and inefficient code. For more information about GPU debugging and performance analysis, I refer to Hilgart [Hil].

GLSL supports three types of variables in addition to the regular, temporary variables. Uniform variables are specified per rendering pass and shared between all vertices/fragments in the vertex- and fragment-processors. Attribute variables are specified per vertex and only accessible through vertex shaders. Varying variables are set by the vertex program, interpolated by the rasterizer, and read by the fragment program. In addition, constant variables and defines are also available. GLSL is very flexible in how the variables can be accessed, and implements two features called swizzling and smearing. Swizzling feature enables programmers to easily select and rearrange components by listing their names, e.g, `pos.xy` or `color.abgr`. Smearing allows duplication of components, e.g., `var.xxyy` [Ros06].

2.4 NVIDIA GeForce 8800

GPUs are probably one of the most rapidly evolving pieces of consumer level hardware in the market, and new generations appear frequently. When the work on this thesis began, GeForce 7800 was state of the art. Today, just over a year later, 7800 is second generation and the new G80-series is about to take over. The new series has "more of everything", but also radically changes many aspects of the GPU that close the gap between the GPU and a generic stream processor. GeForce 8800 Ultra is the latest addition to the family, with 128 scalar processors running at 1.5 GHz and 768 Mb of video memory running at 2160 MHz. GeForce 8800 GTX can theoretically perform up to 520 GFLOPS [NVIc]. The Direct3D 10 API, fully supported by the G80 series, requires the GPUs to have support for 32-bit integer operations, including bitwise operations [Bly06].

With the new series, a new programmable stage is introduced, the geometry shader. This shader replaces the triangle assembly stage. The stage is executed after the vertex shader, and its input is primitives with adjacency information. The output of the geometry shader is zero or more primitives. This can be used to refine a mesh or to reject primitives at an early stage [Bly06].

As different scenes have different processing requirements, the utilization of the different shader types varies. The G80 series have unified shader hardware in opposition to the separated design used in previous generations. This improves the processing capabilities by ensuring that all processors are kept busy at all times (see Figure 2.6). For GPGPU this means that more processors are available.

In contrast to previous vectorized designs, the GPUs in the G80-series are scalar processors. On previous hardware packing schemas was used to pack data into textures with four components to utilize the vector design of the processors. This makes the G80 series

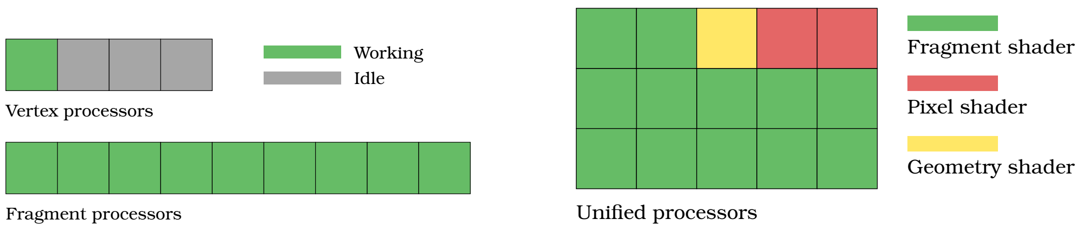


Figure 2.6: Typical GPGPU scenario on previous generations (left). The vertex processors have little or virtually no work to do and are idle most of the time, while the fragment processors are busy doing computations. The right illustration shows the situation on the new G80 series, where the number of processors utilized by each shader is dynamic.

easier to utilize for general purposes as there is no reason to attempt to apply as much operations as possible on vectors instead of scalars. The new architecture currently has up to 128 scalar processors instead of 32 four-vector processors [NVIc]. For fully vectorized programs, 128 scalar processors conforms to 32 four-vectorized processors ($4 \times 32 = 128$), but for scalar operations the new architecture is up to four times more efficient.

NVIDIA has also released a new framework named CUDA for general stream processing applications [NVI07]. This framework enables developers to implement GPU stream programs in C, and allows direct access to the GPU without interference from the graphics API. This results in less overhead and direct utilization of the GPU processors without worrying about differences between vertex, geometry and fragment processors. Many restrictions have been removed, e.g., processes can communicate with each other, giving new possibilities. Scattered memory writes are also supported by CUDA, thereby removing one of the major obstacles in GPGPU. Hardware debugging is supported, making the development process a lot less painful [NVIc]. NVIDIA have also released CUBLAS [NVI06], an implementation of a part of the BLAS (basic linear algebra subprograms) library. This implementation can be used without knowledge about CUDA.

Although not supported at this moment, NVIDIA plans to support 64-bit floating-point precision arithmetic (double) in late 2007 [NVIa]. This will move the GPU one step closer to being a full-fledged general parallel floating-point processor.

Even though the main focus will remain to be to deliver high-quality graphics for games, all these new features, in addition to improved performance, make the G80-series a more general streaming processor than its predecessors.

Chapter 3

Computation fluid dynamics on the GPU

GPGPU is a relatively new field of research that focuses on utilizing the tremendous floating-point processing capabilities of the GPUs to solve general problems. In recent years there has been an explosion in the interest for GPGPU, and new algorithms are presented regularly. The reason for this interest is the high potential floating-point performance GPUs offers at a modest cost.

Problems from many different fields have utilized GPUs, ranging from sorting and database applications, to linear algebra and algorithms for solving PDEs. Among linear algebra applications implemented with good results, is matrix multiplication [Bro07, HCH03], conjugated gradients [KW03, BFGS03] and PLU factorization [Bro07]. The fast Fourier transform has also been implemented [GLGM06], and many image processing algorithms are suitable for GPU implementation [FM05]. A number of sorting and data mining algorithms have been implemented [GZ06, GGKM06].

CFD has been a field of research for a very long time, and CFD algorithms have been implemented on GPUs in various forms in the recent years. This chapter focuses on some of the research done within the field of CFD, both implementations for real-time rendering and high-quality simulation for visual purposes. The implementations reviewed focuses on performance, plausible results and visual quality. Physical correctness is a secondary goal. Note that many of the implementations reviewed are CPU based. These implementations are mentioned to give useful background

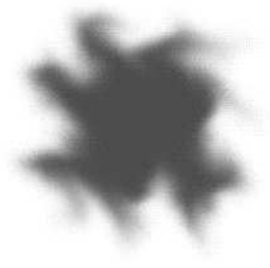


Figure 3.1: Turbulent wind simulated using Stam and Fiume's model. Figure is courtesy of Stam and Fiume.

information, as many of the ideas implemented for use by CPU implementations have been adapted for usage by GPU implementations. The chapter ends with a discussion concerning the future for GPUs as a physics coprocessor.

3.1 Real-time simulators

It is only recently that real-time fluid simulators have emerged. However, efficient simulation has been a field of research for a long time. Here, a few important methods are introduced to give some background for the methods described later in this thesis.

Early implementations utilized various tricks to resemble fluid behavior, while avoiding to use of too much processing power. In 1993, Stam and Fiume [SF93] presented a model for turbulent wind flow based on one field for large-scale motion and one field for small turbulent motion. The large-scale field is generated by numerically solving the Navier-Stokes equations and the small-scale field is a random field. The model can simulate wind fields to some extent, but as the random field is not a function of the actual motion, the simulated field looks rather artificial (see Figure 3.1).

Foster and Maxamas [FM96] presented an implementation for animating liquids based on the Navier-Stokes equations in 1996. They described three different methods for tracking the surface of fluids, which is often necessary when rendering. Marker particles track mass less particles that are convected by the fluid velocity and are ideal for animating violent phenomena such as a tsunami. The free surface technique also utilizes marker particles to track the surface. A grid of marker particles is placed along the boundary layer between the fluid and the air. These particles are also convected by the velocity field. Next, assume that cells can either be *Full* or *Empty* (of fluid), The volumes of the two fluids are grown by alternating between growing *Full* and *Empty* volumes (or areas in 2D). Initially, cells containing marker particles are marked as *Full*, the rest of the cells are marked as *Unknown*. Next the volumes are grown by marking a cell known always to be *Empty*. The *Empty* volume is grown from this cell by marking neighbouring cells marked as *Unknown* with *Empty* instead. When no more *Unknown* cells with *Empty* neighbour(s) are found, a second region is grown by marking *Unknown* cells adjacent to *Full* cells as *Full*. The process is repeated, alternating between *Empty* and *Full* regions. The authors also describe tracking of a surface through a height field. This method tracks height by calculating it based on pressure and velocity. The change in surface elevation is computed using

$$\frac{\partial h}{\partial t} = w - u\left(\frac{\partial h}{\partial x}\right) - v\left(\frac{\partial h}{\partial y}\right),$$

where h is the surface height, w is the vertical component of the fluid velocity, u and v are the components of velocity in the $\mathbb{XY}$-plane (assuming vertical component is specified along the $\mathbb{Z}$-axis).

Stam [Sta99] described an unconditionally stable discretization method for fluid flows in 1999. As opposed to the methods mentioned above, the solution is stable for any choice of Δt. This is achieved by using a combination of Lagrangian and implicit methods to solve the incompressible Navier-Stokes equations. An approximation to the Navier-Stokes equation is computed using operator splitting and projection is used to project the resulting velocity field onto its divergent free part (as required by the incompressible Navier-Stokes). As the model suffers from numerical dissipation, the flow tends to dampen too rapidly. Stams implementation of the stable method enables near real-time simulation and rendering on relative coarse grids (16^3 to 30^3) on an SGI machine.

Stam [Sta03] also described an implementation for real-time fluid dynamics for use in games in 2003. The implementation described is unconditionally stable and based on the incompressible Navier-Stokes equations, just as the previous implementation described. No new techniques are introduced in this article, but a number of extensions that could be implemented are mentioned. Figure 3.2

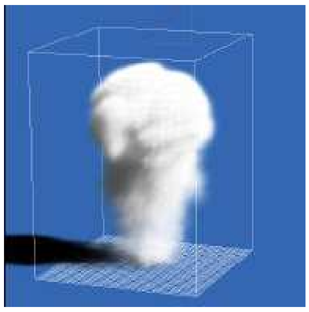


Figure 3.2: Screenshots from Stams implementation of a real-time fluid simulator. Figure is courtesy of Stam.

shows a screenshot from the example application.

Harris wrote a chapter about real-time CFD on the GPU for “GPU Gems” [Har03] released by NVIDIA in 2003. Here, an unconditionally stable method for computing solutions of the incompressible Navier-Stokes equations in the same manner as the method presented by Stam [Sta99] was presented. Harris used an iterative Jacobi solver to estimate the solutions of the Poisson equations that arise from the semi-Lagrange discretization. He used the simulator to simulate liquids and gases. Support for smoke and cloud simulations was also implemented using buoyancy and convection.

Krüger and Westermann [KW05] documented another GPU implementation of CFD in 2005. They implemented a simulator for smoke, fire or explosions that could be run on the GPU without any readback. The implementation is based on the incompressible Navier-Stokes equations and uses various tricks to enable the user to control the simulation and to support 3D simulations without using an unacceptable amount of processing power. Using insertion of pressure and velocity templates (i.e., local perturbations of the respective fields) the user can control where desired effects such as vortices should occur. To support 3D simulations without having to simulate the whole 3D domain, the application supports volumetric extrusion. This technique simulates a number of slices using slightly different initial conditions and uses spherical linear interpolation to extrude these 2D slices to 3D. The application supports both texture-based volume rendering and particle-based volume rendering methods. The conjugate-gradient method is used to solve the linear set of equations that arises when discretizing Poisson pressure equation that must be computed when

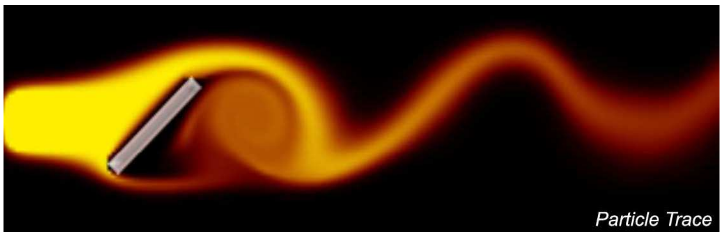


Figure 3.3: Screenshots from a Navier-Stokes simulator on the GPU. Figure is courtesy of Krüger and Westermann.

implementing projection. However, the method described is not unconditionally stable, introducing constraints on the choice of Δt. Figure 3.3 shows a screenshot from their application.

3.2 Offline rendering algorithms

The methods mentioned above focus on high performance simulators that produce a visually plausible result. In this section a few methods focusing on high quality simulations using offline rendering, are reviewed.

Stam, Fedkiw and Jensen [FSJ01], described a model based on the inviscid Euler equations for visual simulation of smoke. This model uses a technique called vorticity confinement to reinject lost energy due to numerical dissipation. Vorticity confinement is a computationally cheap technique to counteract dissipation, and has later been implemented in a number of simulators. Advanced rendering techniques used to achieve smoke of high visual quality are also described. Notice the difference between the simulations with and without vorticity confinement in Figure 3.4.

Enright, Marschner and Fedkiw [EMF02] described in 2002 a model for simulating and rendering water with very high visual quality. They described a method for tracking the water surface based on extrapolating velocity from the regions of water (or some other liquid) to the regions of air. Navier-Stokes is solved in the same manner as in the stable fluid model described by Stam [Sta99] mentioned above.

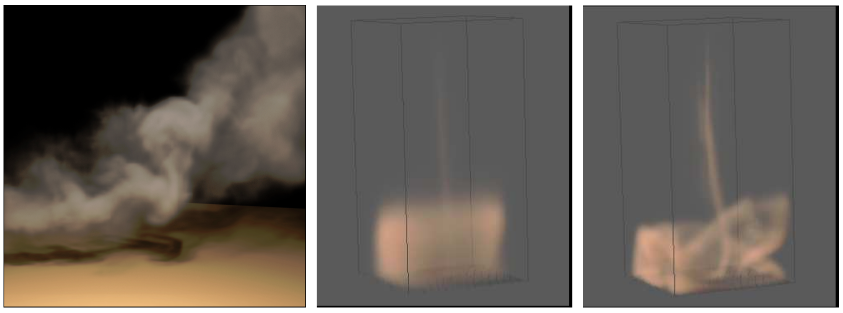


Figure 3.4: Screenshots from the implementation using vorticity confinement by Stam, Fedkiw and Jensen. The screenshot to the left is from a simulation of smoke. The middle screenshot is a simulation performed without vorticity confinement, while the right screenshot is from a simulation using the same initial conditions with vorticity confinement enabled. Figure is Stam, Fewkiw and Jensen.

3.3 The future of GPU physics

The articles presented above are only a small selection of the published work within CFD. Real-time CFD for visual purposes has received much attention from the GPGPU community, and applications have been implemented at several occasions. As the methods improve and GPUs get more powerful, this opens for usage of real-time visual CFD for interactive applications, typically games. There is a trend that new games utilize physics in a more extensive manner than older games. Rhodes [Rho04] claims that physics can reduce the cost of game production by reducing the number of animators necessary. He also states that the use of a physics engine enables behavior that is not practical when using artist-generated animations.

AGEIA has released a dedicated physics processor called PhysX [AGE]. This processor supports a wide range of physical phenomena, such as particle effects, fluid and cloth simulation and collision computation. There is also a software development kit (SDK) associated with PhysX that Sony has licensed to the Playstation 3 among others [Son05a]. Playstation 3 also supports a competing physics engine, named Havok [Son05b]. The Havok SDK is not based upon a dedicated physics processor, but has several implementations, such as one for recent ATI and NVIDIA GPUs. The SDK features collision detection, simulation of particles, cloth and liquids and effects including debris and fog. A release based upon the Shader Model 3.0 standard for using the GPU as the computational engine was released June 2006 [Hav06]. Performing physics on the GPU makes sense, because many games are CPU limited rather than GPU limited. Since the simulations will be used for rendering purposes, performing them directly on the GPU eliminates much data transfer too [Har06].

The emerge of dedicated physics engines and hardware is an indication of the increasing interest for physics in computer games. Increased focus from GPU hardware vendors on the use of GPUs for general purposes, e.g., physics, leads to increased level of programmability and makes handling of physics on the GPU a simpler process. The adaptation and implementation of physics SDKs, such as Havok for GPUs, indicate that the GPUs will not only be used for purely graphical purposes in future games, but also handle some, or even most, of the underlying physics.

Chapter 4

Incompressible Navier-Stokes

The Navier-Stokes equations are set of equations that describe changes in mass and momentum of fluids. The equations can be used to describe a large number of phenomena, ranging from water flow to the motion of stars inside a galaxy. This chapter will go through the derivation, discretization and an implementation of a simulator for computing solutions to the incompressible Navier-Stokes equations. The incompressible equations are used to describe flows for which the density changes in the fluid are negligible. This form of Navier-Stokes can be used to simulate many phenomena, such as smoke, formation of clouds, and fluid flows. In real life, incompressible fluids do not occur; all materials are compressible to some extend. However, under the proper conditions, compressible fluids can undergo near incompressible flow.

4.1 Derivation

The derivation of the incompressible Navier-Stokes equations involves three fields:

$$\begin{array}{rclcll} \vec{u} & : & \Omega \times [0, t_{end}] & \rightarrow & \mathbb{R}^n & \text{velocity field} \\ p & : & \Omega \times [0, t_{end}] & \rightarrow & \mathbb{R} & \text{pressure field} \\ \rho & : & \Omega \times [0, t_{end}] & \rightarrow & \mathbb{R} & \text{density field} \end{array} \tag{4.1}$$

In the incompressible case, the density changes in time and space are negligible, i.e., $\rho(\vec{x}, t)$ is constant.

The Navier-Stokes equations are derived from conservation of mass and momentum. Conservation of mass means that an amount of fluid occupying a domain Ω_0 will later occupy domain Ω_t. The mass of the fluid occupying a domain equals the integral of the density $\rho(\vec{x}, t)$ over the domain. Thus

$$\int_{\Omega_t} \rho(\vec{x}, t) d\vec{x} = \int_{\Omega_0} \rho(\vec{x}, 0) d\vec{x}. \tag{4.2}$$

In other words, the time derivative of the integral on the left side must be zero:

$$\frac{d}{dt} \int_{\Omega_t} \rho(\vec{x}, t) d\vec{x} = 0. \tag{4.3}$$

The *transport theorem* gives an expression for the time derivative of the integral of a field λ over a domain that changes with time:

$$\frac{d}{dt} \int_{\Omega_t} \lambda(\vec{x}, t) d\vec{x} = \int_{\Omega_t} \left[\frac{\partial \lambda}{\partial t} + \nabla \cdot (\lambda \vec{u}) \right] (\vec{x}, t) \; d\vec{x}. \tag{4.4}$$

By applying this theorem to $\rho(\vec{x}, t)$ and using (4.3) it is clear that the left term of (4.4) vanishes. After rearranging the equation and using that the result is valid for arbitrary domain Ω_t, the equation reads

$$\begin{aligned} 0 &= \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) \\ &= \frac{d\rho}{dt} + \rho \nabla \cdot \vec{u} + \vec{u} \nabla \rho \\ &= \frac{D\rho}{Dt} + \rho \nabla \cdot \vec{u}, \end{aligned}$$

where $\frac{D}{Dt}$ is the Lagrangian derivate defined as $\frac{D}{Dt} = \frac{\partial}{\partial t} + (u \cdot \nabla)$. Since the density changes in time and space are negligible the first term can be left out. This results in

The continuity equation.

$$\nabla \cdot \vec{u} = 0. \tag{4.5}$$

The second equation of the incompressible Navier-Stokes equations is a result of conservation of momentum (product of mass and velocity). Since the velocity varies in space, the product must be written as an integral:

$$\vec{m}(t) = \int_{\Omega_t} \rho(\vec{x},t)\vec{u}(\vec{x},t)\; d\vec{x}. \tag{4.6}$$

Newton‘s second law states that the change in momentum equals the sum of the forces acting on the fluid

$$\frac{d}{dt}\vec{m}(t) = \sum \vec{f}(t). \tag{4.7}$$

The forces can be separated into two terms, external forces and surface forces. External forces account for, e.g., gravity, and can be written as the integral of the product between density and force density per unit volume. Surface forces account for pressure and internal friction and can be written as the integral of the product of a stress tensor τ and the surface normal $\vec{n}$. Thus,

$$\vec{f}_{\text{ext}} = \int_{\Omega_t} \rho(\vec{x},t)\vec{g}(\vec{x},t)\; d\vec{x}, \tag{4.8}$$

$$\vec{f}_{\text{surf}} = \int_{\partial\Omega_t} \tau(\vec{x},t)\vec{n}(\vec{x},t)\; ds, \tag{4.9}$$

where $\vec{g}$ is the force density and $\partial\Omega_t$ is the boundary of the domain Ω_t. By replacing $\vec{m}(t)$ in (4.7) by the right hand side of (4.6), Newton’s second law reads

$$\frac{d}{dt}\int_{\Omega_t} \rho\vec{u}\; d\vec{x} = \vec{f}_{\text{ext}} + \vec{f}_{\text{surf}}. \tag{4.10}$$

Here the assumption that ρ is constant has been used. In order to derive the Navier-Stokes momentum equation the transport theorem (4.4) and the divergence theorem are needed. The *divergence theorem* reads

$$\int_{\partial\Omega_t} \vec{\kappa}\cdot\vec{n}\; ds = \int_{\Omega_t} \nabla\cdot\vec{\kappa}\; d\vec{x}, \tag{4.11}$$

where $\vec{n}$ is the surface normal and $\vec{\kappa}$ is a vector field. By applying the divergence theorem (4.11) to the expression for surface forces (4.9) we get

$$\vec{f}_{\text{surf}} = \int_{\partial\Omega_t} \tau\vec{n}\; ds = \int_{\Omega_t} \nabla\cdot\tau\; d\vec{x}. \tag{4.12}$$

By using that ρ is constant, a simplified expression for $\vec{f}_{\text{ext}}$ is found:

$$\vec{f}_{\text{ext}} = \rho \int_{\Omega_t} \vec{g}(\vec{x}, t) \; d\vec{x} \tag{4.13}$$

The transport theorem (4.4) is applied to the left hand side of (4.10):

$$\frac{d}{dt} \int_{\Omega_t} \rho \vec{u} \; d\vec{x} = \rho \int_{\Omega_t} \left[\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} \right] d\vec{x} = \vec{f}_{\text{ext}} + \vec{f}_{\text{surf}}. \tag{4.14}$$

Since Ω_t is arbitrary, the integrals can be removed from each term of (4.10). After dividing by ρ, (4.10) can be expressed as

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} = \frac{1}{\rho} \nabla \cdot \tau + \vec{g}. \tag{4.15}$$

The stress tensor τ for incompressible flows includes a Lamé dynamic viscous term μ and reads

$$\tau = -p\mathbf{I} + 2\mu\epsilon, \tag{4.16}$$

where ϵ is the deformation tensor given by

$$\epsilon = \frac{1}{2} \left(\nabla \vec{u} + (\nabla \vec{u})^T \right).$$

In the 2D case, the tensor reads

$$\tau = \left[\begin{array}{cc} \mu \frac{\partial u_x}{\partial x} - p & \mu \left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right) \\ \mu \left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right) & \mu \frac{\partial u_y}{\partial y} - p \end{array} \right].$$

Now, $\nabla \cdot \tau$ in (4.15) can be simplified by assuming that μ is constant and applying the continuity equation ($\nabla \cdot \vec{u} = 0$). For tensors, the ∇-operator is defined as applying ∇ to each row, resulting in a vector:

$$\begin{aligned} \nabla \cdot \tau &= -\nabla p + \nabla \cdot (2\mu\epsilon) \\ &= -\nabla p + \mu \nabla \cdot (\nabla \vec{u} + (\nabla \vec{u})^T) \\ &= -\nabla p + \mu \nabla^2 \vec{u}. \end{aligned}$$

The momentum equation can now be formed by inserting the value of $\nabla \cdot \tau$ into (4.15) and rearranging.

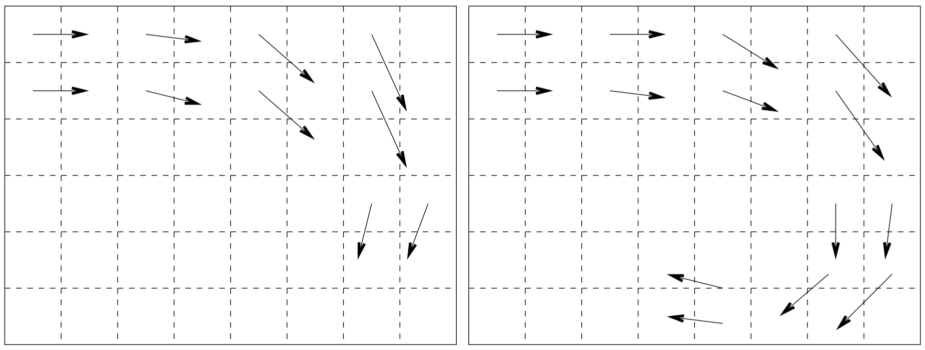


Figure 4.1: The velocity field advects itself

The momentum equation.

$$\frac{\partial \vec{u}}{\partial t} = -\left(\vec{u}\cdot\nabla\right)\vec{u} + \nu\nabla^2\vec{u} - \frac{1}{\rho}\nabla p + \vec{f}. \qquad (4.17)$$

Here $\vec{f}$ is defined as $\vec{f} = \vec{g}$ to emphasize that the term may include other forces than gravity and $\nu = \frac{\mu}{\rho}$ is the kinematic viscosity constant [Mar04].

4.2 The incompressible Navier-Stokes equations

By combining the momentum and continuity equations from the previous section, the Navier-Stokes equations for incompressible fluids appear:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{u} + \nu\nabla^2\vec{u} - \frac{1}{\rho}\nabla p + \vec{f}, \qquad (4.18)$$

$$\nabla\cdot\vec{u} = 0. \qquad (4.19)$$

The first equation (4.18) is the momentum equation. It has four terms on the right hand side sums up to the total acceleration:

Advection – The first term, $-(\vec{u}\cdot\nabla)\vec{u}$, is called the advection term. The velocity field causes quantities to follow, or be advected, by the flow. This happens to particles, e.g., dye, but the velocity field also advects itself (see Figure 4.1).

Diffusion – The second term, $\nu\nabla^2\vec{u}$, accounts for viscous diffusion. This is an effect of resistance to deformation due to shear stress and ν is the viscosity of the fluid (high values meaning highly viscous fluid). The special case when $\nu = 0$ is called an inviscid fluid. Such fluids do not occur in real life, but can be useful in practice when simulating fluids with low viscosity (e.g., air) and high velocities to save computation [Wik07d]. This corresponds to simulations with a high Reynolds number, which is defined as

$$Re = \frac{\text{Inertial forces}}{\text{Viscous forces}} = \frac{|\vec{u}|L}{\nu},$$

where $\vec{u}$ is the fluid velocity, ν is the viscosity and L is the characteristic length, which is related to the size of the domain of interest. The Reynolds number is dimensionless and used to identify different types of flows [Wik07b]. Laminar flow is relative steady flow with Reynolds number up to about 2000. In this type of flow, streamlines follow the obstacles or time-invariant stable regions where the flow is reversed is formed. Turbulent flow is characterized by its unpredictable behavior. No periodicity can be observed and the flow is chaotic and irregular. A flow with Reynolds number larger than about 4000 is called turbulent. Transitional flow fills the gap between laminar and turbulent flow. This type of flow often starts out as laminar, but starts to behave turbulent after a certain distance from the inflow [GDN98]. This thesis mainly focuses on inviscid fluids to narrow the field of focus.

Pressure – Next is the pressure term, $-\frac{1}{\rho}\nabla p$, which reflects the effect of internal pressure caused by particles packed together causing outward pressure. As shown in the next section, the pressure term can be disregarded when semi-Lagrange discretization is used.

External forces – The last term of (4.18), $\vec{f}$, accounts for external forces. This can typically be gravity or in the case of computer graphics, an artificial force added to produce a desired visual effect.

The second equation (4.19), is called the continuity equation and ensures conservation of mass. This equation is the incompress-

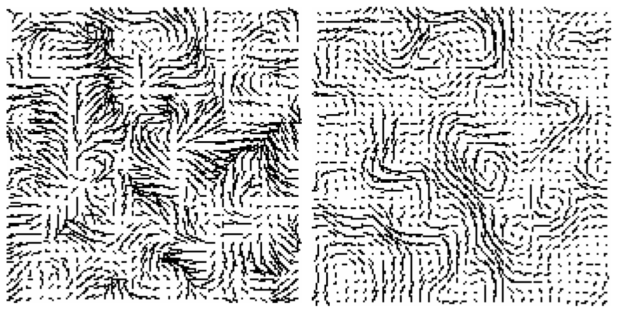


Figure 4.2: The effect of the continuity equation. The image to the left shows a vector-field with non-zero divergence ($\nabla \cdot \vec{v} \neq 0$), and the right image shows the same vector-field projected to the corresponding field with zero divergence. Figure is courtesy of Jos Stam.

ible requirement, that is, it allows us to assume that density is constant. Figure 4.2 shows the effect of the continuity equation.

4.3 Semi-Lagrange discretization

The focus of this thesis is real-time simulations for visual purposes. Unconditional stability with respect to Δt is desirable to allow time steps of arbitrary size. An unconditionally stable solver based on the semi-Lagrange scheme is presented in this section. The discretization leads to numerical dissipation, and vorticity confinement is used to reinject lost energy in order to retain small-scale effects, i.e., vortices. For simplicity, it is assumed that the grid spacing is equal in both directions throughout the chapter; that is, $\Delta x = \Delta y$.

Implicit discretization tends to result in more stable solvers than using explicit methods. In addition to using Lagrangian description of the advection term, this is utilized to achieve a unconditionally stable solver as done by Stam [Sta99]. The derivation of this solver uses the Holmholtz-Hodge decomposition, which states that a vector field $\vec{w}$ always can be uniquely written as

$$\vec{w} \quad = \quad \vec{u} + \nabla q, \tag{4.20}$$

where $\vec{u}$ is a divergence-free (incompressible) vector field and q is a scalar field [CM05]. The projection-operator $\mathbb{P}$ projects any vector field $\vec{w}$ onto its divergence-free part $\vec{u}$. This operator is defined as multiplying (4.20) with ∇:

$$\nabla \cdot \vec{w} = \nabla \cdot \vec{u} + \nabla^2 q = \nabla^2 q. \tag{4.21}$$

This is the Poisson equation for a scalar field q. By solving this for q we can calculate the projection from $\vec{w}$ to its incompressible part $\vec{u}$:

$$\vec{u} = \mathbb{P}\vec{w} = \vec{w} - \nabla q. \tag{4.22}$$

By applying $\mathbb{P}$ to (4.18) we get

$$\frac{\partial \vec{u}}{\partial t} = \mathbb{P}\left[-(\vec{u}\cdot\nabla)\vec{u} + \nu\nabla^2\vec{u} + \vec{f}\right]. \tag{4.23}$$

Here two important properties have been utilized. First, from the continuity equation (4.19) we know that $\mathbb{P}\vec{u} = \vec{u}$ since $\vec{u}$ is divergence free. The second property utilized is that when $\mathbb{P}$ is applied to ∇p the result is 0. This is shown by applying $\mathbb{P}$ to (4.20):

$$\mathbb{P}\vec{w} = \mathbb{P}\vec{u} + \mathbb{P}\nabla q.$$

By definition $\mathbb{P}\vec{w} = \mathbb{P}\vec{u} = \vec{u}$, so $\mathbb{P}\nabla q = 0$ or $\mathbb{P}\nabla p = 0$ in this case.

The projected Navier-Stokes equation is used to compute (4.18) in four steps:

$$u(\vec{x},t) \xrightarrow{\text{Add Forces}} v_1(\vec{x}) \xrightarrow{\text{Advect}} v_2(\vec{x}) \xrightarrow{\text{Diffuse}} v_3(\vec{x}) \xrightarrow{\text{Project}} u(x, t+\Delta t).$$

Each term is applied to an intermediate solution and $\vec{v_3}(x)$ is “projected” such that the final result $\vec{u}(x, t+\Delta t)$ is incompressible. This technique is known as operator splitting for projection.

Add forces. The first step is straight forward to implement, and simply boils down to adding two fields together, where the force field $\vec{f}$ is weighted by Δt:

$$\vec{v_1}(\vec{x}) = \vec{u}(\vec{x},t) + \Delta t \cdot \vec{f}(\vec{x}). \tag{4.24}$$

In other words, $\frac{\partial \vec{u}}{\partial t} = \vec{f}$ is solved using the forward Euler method.

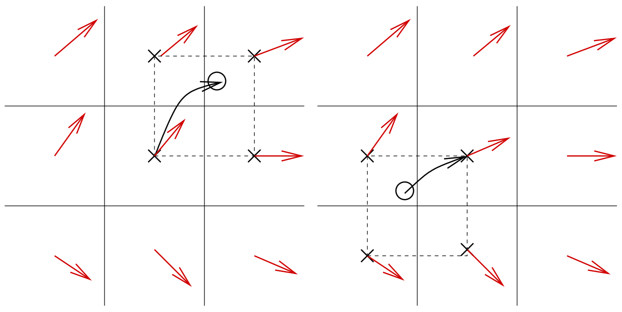


Figure 4.3: Two ways of approximating advection. The method illustrated to the left considers which cells the current cell will contribute to by integrating forward in time. The right method integrates backward in time to find its origin and interpolates the four-cell neighbourhood to find the new value of the cell.

Advection. Advection transports the velocity field along itself. The advection term reads $\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u}$ and could also have been computed by using an explicit update. However, this method has two serious drawbacks. The operation is a scatter operation which makes it ill-suited for GPU implementation. The second drawback is related to the constraint on the size of the time step used in the simulation. The Courant-Friedrichs-Lewy (CFL) condition,

$$\frac{\Delta t}{\Delta x} |\vec{v_1}(\vec{x})| < C, \tag{4.25}$$

is a condition for a number of equations to be stable. If it is not complied, that is, when a quantity is transported through one or more cells in one time step, the solution may blow up [Har03]. By introducting Lagrangian coordinates and using the method of characteristics, both of these problems are avoided. In mathematical terms, the discretized equation for advection now reads:

$$\vec{u}\,(\vec{x}, t + \Delta t) \quad = \quad \vec{u}(\vec{x} - \Delta t \cdot \vec{u}(\vec{x}, t), t). \tag{4.26}$$

This method is unconditionally stable [Sta99]. By integrating backward in time, this method resolves the new values at each grid point (see Figure 4.3).

Diffusion. The next term is the diffusion term, which reads

$$\frac{\partial \vec{u}}{\partial t} \quad = \quad \nu \nabla^2 \vec{u}. \tag{4.27}$$

The kinematic viscosity constant, ν, controls the amount of diffusion and can be considered to be the thickness of the fluid [GDN98]. For water at 20^{o}C this constant equals $1.004 \times 10^{-6}\ m^2/s$ and $1.330 \times 10^{-5}\ m^2/s$ for air at the same temperature [Box]. The diffusion term can be approximated explicitly by integrating forward in time:

$$\vec{u}(\vec{x}, t + \Delta t) \;\approx\; \vec{u}(\vec{x}, t) + \Delta t \cdot \nu \nabla^2 \vec{u}(\vec{x}, t). \tag{4.28}$$

Unfortunately this is numerically unstable for large Δt and ν [Har03]. By instead using an implicit discretization schema, a Poisson type equation is obtained:

$$\begin{aligned} \frac{u^{k+1} - u^k}{\Delta t} &\approx \nu \nabla^2 u^{k+1} \\ u^{k+1} - u^k &\approx r\left(u^{k+1}_{i-1,j} + u^{k+1}_{i+1,j} + u^{k+1}_{i,j-1} + u^{k}_{i,j+1} - 4u^{k+1}_{i,j}\right), \end{aligned}$$

where $r = \frac{\nu \Delta t}{\Delta x^2}$. By moving terms u^{k+1} to one side and term u^k to the other, and rewriting to matrix form the discrete implicit diffusion equation appears:

$$(\mathbf{I} - r\mathbf{L})\, u^{k+1} \;=\; u^k, \tag{4.29}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{L} \in \mathbb{R}^{n^2 \times n^2}$ is the Laplacian matrix given by (n is the number of unknowns in each direction)

$$\mathbf{L} = \begin{bmatrix} -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 \\ \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 \\ 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 \\ & \ddots & & & & \ddots & \ddots & \ddots & & & & \ddots \\ 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 \end{bmatrix}. \tag{4.30}$$

How solutions to this problem can be computed is discussed in Chapter 5.

Projection. After performing the previous steps (adding forces, advection and diffusion), the projection operator $\mathbb{P}$ is applied to the intermediate result. Recall that this operator projects a vector field $\vec{w}$ onto its incompressible part $\vec{u}$, which satisfies the continuity requirement in (4.19). This is achieved by applying the $\mathbb{P}$-operator to the result from the last operator splitting pass, $\vec{v}_3$:

$$\nabla \cdot \vec{v}_3 = \nabla \cdot \vec{u} + \nabla^2 q = \nabla^2 q. \tag{4.31}$$

This is also a Poisson equation, and will from now on be called the pressure equation. By solving for q and using this scalar field and the identity $\vec{u} = \vec{v}_3 - \nabla q$, the result from each time step will be incompressible (if exact arithmetic and methods were used).

In order to determine the projection to the incompressible velocity field, the pressure must be computed. The pressure equation can be discretized as follows. The right side of (4.31) is discretized using central differences, yielding

$$\left(\nabla^2 q\right)_{i,j} = \frac{1}{\Delta x^2}\left(q_{i+1,j} + q_{i-1,j} + q_{i,j+1} + q_{i,j-1} - 4q_{i,j}\right). \tag{4.32}$$

The discrete pressure equation can now be expressed in matrix form:

$$\frac{1}{\Delta x^2}\mathbf{L}\vec{q} = \vec{g}. \tag{4.33}$$

The right hand side $\vec{g}$ is formed by stacking columns of $\nabla \cdot \vec{v}_3$ on top of each other. The unknown vector $\vec{q}$ is formed in a similar fashion, and the coefficient matrix $\mathbf{L}$ is the Laplacian matrix (4.30). Efficient methods for computing solutions to this problem will be discussed in Chapter 5.

4.4 Vorticity confinement

The semi-Lagrangian solver causes numerical dissipation that result in loss of fine scale details. Vorticity confinement is specifically designed to preserve small vortices in a flow, giving a more realistic appearance. The method was designed by Steinhoff and Underhill [SU94] and was later utilized by Fedkiw, Stam and Jensen [FSJ01] to preserve the vorticity field which is smoothed out due to the numerical dissipation. By reinjecting energy where the vorticity field is pointing from low to high values, vorticity confinement tends to reinject energy where it is lost. This is the strength of vorticity confinement. It results in very few or no artificial vortices at undesired

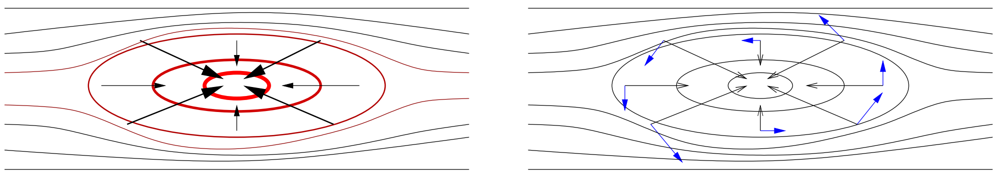


Figure 4.4: A vector field is generated from the gradient of the vorticity field (left). The red contour lines indicate areas with high vorticity and the arrows point in the direction of the gradient. Force is added to the velocity field (blue arrows) in the direction perpendicular to the gradient of the vorticity field (right).

locations as many other techniques do (e.g. adding a random vector field according to some energy function [SF93]). The algorithm first calculates a vorticity field (the curl of the velocity field). In 2D, the vectors always point out from the plane, and can therefore be represented using a scalar field. Next, the normalized gradient of the vorticity is calculated, revealing areas where vorticity goes from low to high concentrations. Force is added in the direction perpendicular to this field (see Figure 4.4). This tends to keep the vortices alive and localized and yields small vortices in the boundary layer between high and low velocities.

Mathematically the method can be described in the following way: Vorticity is defined as the cross-product between ∇ and a vector $\vec{u}$, in this case the velocity-field,

$$\vec{\omega} = \nabla \times \vec{u}. \tag{4.34}$$

We then want to find the areas where vorticity goes from low to high concentrations and define

$$\vec{\eta} = \nabla \left| \vec{\omega} \right|, \tag{4.35}$$

$$\vec{\psi} = \frac{\vec{\eta}}{|\vec{\eta}|}. \tag{4.36}$$

Now $\vec{\psi}$ points from low to high concentrations in the vorticity field. The magnitude and direction of the resulting force is calculated using

$$\vec{f_{vc}} = \beta \Delta x \left(\vec{\psi} \times \vec{\omega} \right), \tag{4.37}$$

where β is a scalar that defines how much energy to reinject and Δx is the grid-spacing. I have chosen β to 0.35 as default, as experiments have shown that a reasonable amount of energy is re-injected when using this value. The cross-products needed can be simplified and specialized for the 2D-case:

$$\vec{\omega} = \nabla \times \vec{u} \quad = \quad \left(\frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \right) \mathbf{k}. \tag{4.38}$$

This can be discretized using central differences, resulting in

$$\omega_{i,j} = (\nabla \times \vec{u})_{i,j} \quad = \quad \left(\frac{u_{i,j-1} - u_{i,j+1}}{2\Delta x} - \frac{u_{i-1,j} - u_{i+1,j}}{2\Delta y} \right) \mathbf{k}. \tag{4.39}$$

Now $(\vec{\eta})_{i,j}$ can be calculated using the discrete version of (4.35),

$$(\vec{\eta})_{i,j} \quad = \quad \left(\frac{\left|\omega^z_{i+1,j}\right| - \left|\omega^z_{i-1,j}\right|}{2\Delta x} \right) \mathbf{i} + \left(\frac{\left|\omega^z_{i,j+1}\right| - \left|\omega^z_{i,j-1}\right|}{2\Delta y} \right) \mathbf{j}, \tag{4.40}$$

where $\omega^z_{i,j}$ refers to the z-component of $\vec{\omega}$ at position (i, j). Since $\vec{\eta}$ is later normalized and it is assumed that $\Delta x = \Delta y$, the denominator can be skipped:

$$\overline{(\vec{\eta})_{i,j}} \quad = \quad \left(\left|\omega^z_{i+1,j}\right| - \left|\omega^z_{i-1,j}\right| \right) \mathbf{i} + \left(\left|\omega^z_{i,j+1}\right| - \left|\omega^z_{i,j-1}\right| \right) \mathbf{j}. \tag{4.41}$$

Recall that $\vec{\psi} = \frac{\vec{\eta}}{|\vec{\eta}|}$. Since $\vec{\psi}$ only has x and y components and $\vec{\omega}$ only has a z-component, $\vec{\psi} \times \vec{\omega}$ simplifies to

$$\vec{\psi} \times \vec{\omega} \quad = \quad \omega_z \left(\psi_y \mathbf{i} - \psi_x \mathbf{j} \right), \tag{4.42}$$

and in discrete form

$$\left(\vec{\psi} \times \vec{\omega} \right)_{i,j} \quad = \quad \omega^z_{i,j} \left(\psi^y_{i,j} \mathbf{i} - \psi^x_{i,j} \mathbf{j} \right). \tag{4.43}$$

This cross-product is used to calculate the resulting force, defined by (4.37).

The algorithm is implemented using two shaders. The first shader calculates the vorticity-field $\vec{\omega}$ using the discrete formula for the cross product $\nabla \times \vec{u}$ in 2D (4.39). $\vec{\eta}$ is computed by the second shader and $\vec{\psi}$ calculated using this result and normalizing it. In order to avoid numerical artifacts, a safe normalization is used:

$$\vec{\psi} = \frac{\vec{\eta}}{\sqrt{max(\epsilon, \vec{\eta} \cdot \vec{\eta})}}, \tag{4.44}$$

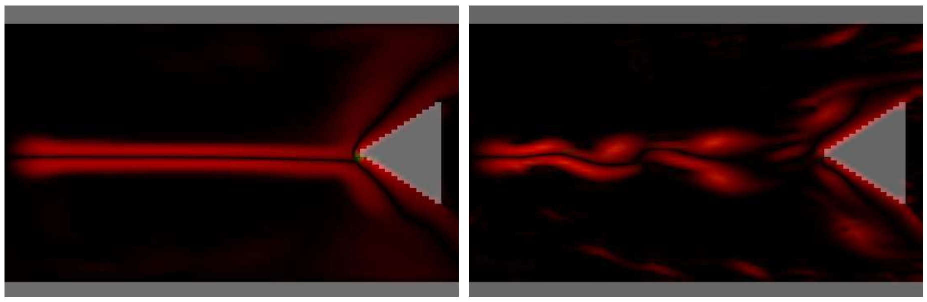


Figure 4.5: A vorticity field without vorticity confinement (left) and with (right) resulting from the same initial conditions.

where $\epsilon \ll 1.0$ is a small number. The cross product $\vec{\psi} \times \vec{\omega}$ is computed and force is calculated using (4.37). The code for this fragment shader is listed in Listing 4.1.

The result from using vorticity confinement is clearly visible. Figure 4.5 shows the vorticity field of a simulation with and without vorticity confinement. When vorticity confinement is used, the flow tends to contain a lot more small vortices than it would do when not using this technique. The flow becomes much less uniform, and for visual purposes, small vortices are very desirable, as the flow feels much more alive and lifelike.

```
uniform sampler2DRect vorticity;
uniform sampler2DRect field;

uniform float scalar;
uniform float h;
uniform float dt;

void main()
{
        // Fetch direct neighbours
        float vT = texture2DRect(vorticity, gl_TexCoord[0].xy
                + vec2(0,1)).x;
        float vB = texture2DRect(vorticity, gl_TexCoord[0].xy
                - vec2(0,1)).x;
        float vR = texture2DRect(vorticity, gl_TexCoord[0].xy
                + vec2(1,0)).x;
        float vL = texture2DRect(vorticity, gl_TexCoord[0].xy
                - vec2(1,0)).x;
        float vC = texture2DRect(vorticity, gl_TexCoord[0].xy).x;

        // n = grad(vorticity)
        vec2 n = vec2(abs(vT) - abs(vB), abs(vR) - abs(vL));

        // Safe normalize
        const static float EPSILON = 2.4414e-4;
        float isqrt = rsqrt(max(EPSILON, dot(n, n)));
        // omega = n / |n|
        vec2 omega = n * isqrt;

        // f = e(omega x n)*dx
        vec2 force = scalar * h * (vC * vec2(omega.y, -omega.x));

        // Apply force
        gl_FragColor = texture2DRect(field, gl_TexCoord[0].xy);
        gl_FragColor.xy += force * dt;
}
```

Listing 4.1: Shader code for the vorticity confinement shader.

4.5 Transport of quantities

A velocity field alone is not very interesting; it is when the velocity field is applied to some quantity like particles or dye the visually interesting results appear. A number of techniques exist to simulate and visualize different quantities. This section will only cover visualization of vector fields and advection of dye. The techniques described are designed to reveal the underlying properties of the flow, such as the vorticity field. More advanced techniques such as simulation and visualization of smoke and other rendering techniques is left out to retain focus on the research questions selected.

Visualization of vector fields. In order to visualize different vector fields, a visualization technique for general vector fields have been implemented. This visualizer supports visualization of two concurrent fields. One field, called the *scalar field* is simply scaled and written to the result. This is typically used to visualize scalar fields containing only a R-component such as vorticity and pressure. Scalar fields can also be used to visualize multi-component textures, e.g., images that have been advected through the flow.

The other field that is visualized is called the *vector field*. The R- and G-components of the vector field is mapped to G and B of the result. This field is typically used to visualize some 2D field, such as the velocity field.

The visualizer also supports masking of boundaries. Although a very simple technique, this have proven to be sufficient to show many details in the underlying fields is able to clearly show the relationship between two fields. Figure 4.6 shows a screenshots from a visualization of a velocity and a pressure field.

Dye is the simplest form of simulation. This is implemented using some initial image that is simply advected through the velocity field without affecting it. The technique is a pure visualization method, and is implemented for visualization of different properties with a high level of control. The application also supports injection of dye by applying them with the mouse or advection of some texture through the flow. Figure 4.7 shows an example of visualization using this technique.

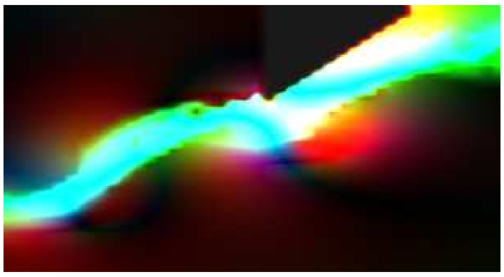


Figure 4.6: Visualization of pressure as the scalar field (red) and velocity as the vector field (green and blue). Notice that green conforms to velocity in the x-direction, while blue is velocity in y-direction.

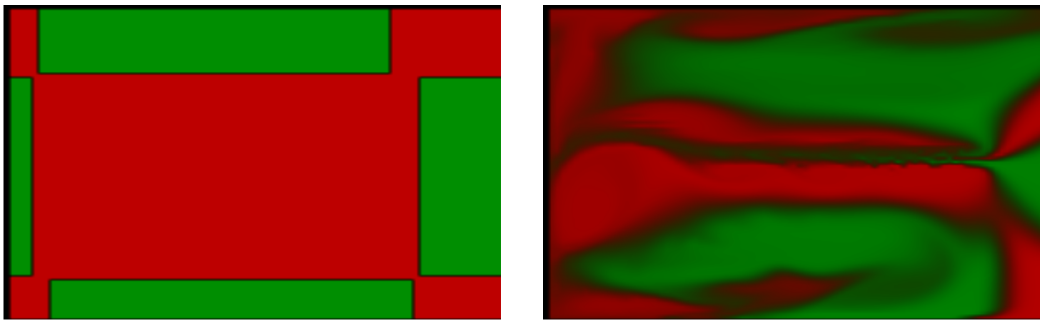


Figure 4.7: Visualization using advection of an initial image (left). The right image shows the situation after a little while.

4.6 Implementation

Due to the use of operator splitting, one time step in the simulation is performed by a number of small programs that form several stages. One stage typically conforms to one term in the incompressible Navier-Stokes equations, but can also be artificial computations, such as vorticity confinement. All computation is performed by the GPU using fragment programs. Most of these programs are small and executes very efficiently, however there are some programs that are critical to performance. Boundary conditions must be applied after all stages affecting the velocity field in the simulator to ensure that the result remains valid with respect to the boundary conditions. Boundary conditions must also be applied after each iteration in the iterative solvers, resulting in boundary conditions being applied many times per time step. Since boundary conditions are applied frequently, the mechanism for doing so must be as efficient as possible. Such an implementation is presented in Chapter 6. The iterative solvers used to compute solutions of the discrete Poisson equations must also be efficient. Efficient solvers for these terms are reviewed in Chapter 5. Figure 4.8 on the next page shows the workflow for the simulator. Some of the stages in the simulator can be enabled and disabled, such as vorticity confinement and diffusion. Other stages have several implementations, such as calculation of pressure which is supported by two iterative solvers. Note that not all parts of the simulator have been mentioned yet, such as the rigid object simulation and updating boundary conditions. This is discussed in Chapter 6.

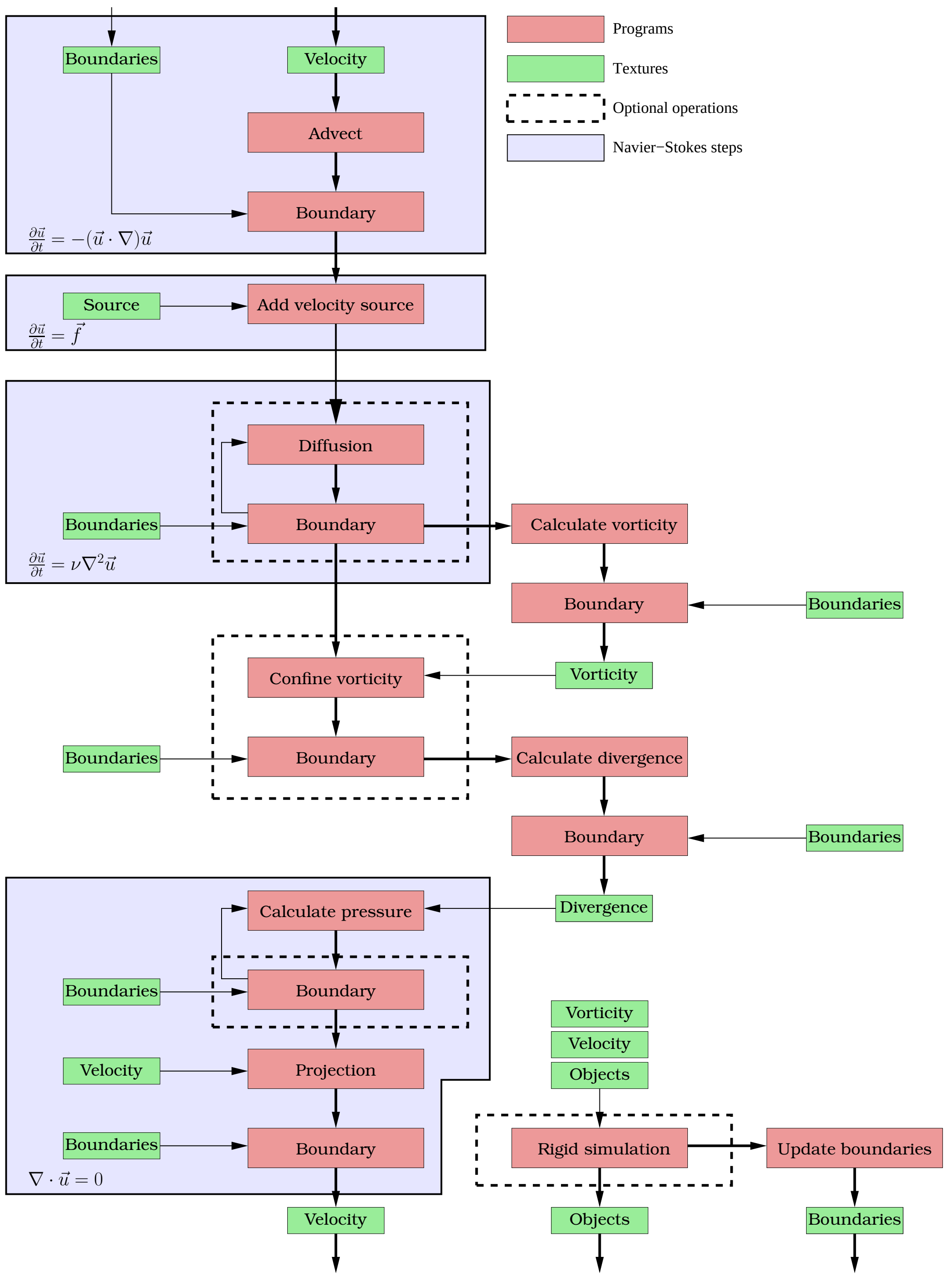


Figure 4.8: Simulator workflow. Since operator splitting is used, the simulation is performed in many steps. The red boxes are fragment programs that perform the computations necessary. The green boxes are either input or output textures. Dashed, black boxes indicate an optional step in the simulator, and blue boxes are groups of programs that conform to a term in the incompressible Navier-Stokes equations.

Chapter 5

Solving the Poisson equations

Two Poisson equations arise when solving Navier-Stokes using the semi-Lagrange discretization:

$$\begin{array}{rcll} \nu\nabla^2\vec{u} & = & \frac{\partial\vec{u}}{\partial t} & \text{Viscous diffusion} \\ \nabla^2 q & = & \nabla\cdot\vec{u} & \text{Poisson equation for pressure} \end{array}$$

At first glance they may look rather different, but both can be solved using the same methods. Most of the work performed per time step by the simulator involves solving these equations, so efficient solvers are crucial for the performance. This chapter assumes the domain is a rectangle without internal obstacles. In Chapter 6 it is shown how this assumption can be relaxed so that the simulator supports arbitrary, internal obstacles as well. However, since the obstacles are treated separately from the solving of the Poisson equations by the simulator, this assumption will not pose a problem.

Recall from Section 4.3 that the discretized diffusion equation reads

$$\left(\mathbf{I} - r\mathbf{L}\right) u^{k+1} \quad = \quad u^k, \tag{5.1}$$

where $\mathbf{I}$ is the identity matrix and

$$r = \frac{\nu\Delta t}{\Delta x^2}.$$

This equation uses the intermediate velocity field as the right hand side and computes new velocity values. The discretized pressure equation reads

$$\frac{1}{\Delta x^2}\mathbf{L}\vec{q} = \vec{g}. \tag{5.2}$$

The right hand side $\vec{g}$ is formed by stacking columns of $\nabla \cdot \vec{u}$ on top of each other ($\vec{u}$ is in this case the velocity resulting from the last operation). The vector of unknowns, $\vec{q}$, is formed in a similar fashion. Both the discrete diffusion equation (5.1) and the discrete pressure equation (5.2) involves the Laplacian matrix,

$$\mathbf{L} = \begin{bmatrix} -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 \\ \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 \\ 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 \\ & \ddots & & & & \ddots & \ddots & \ddots & & & & \ddots \\ 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 1 & -4 \end{bmatrix}. \tag{5.3}$$

Clearly, (5.1) and (5.2) are very similar and both matrices $\mathbf{L}$ and $(\mathbf{I} - r\mathbf{L})$ can be expressed as $\mathbf{M}(\alpha, \beta) = \mathbf{M}$:

$$\mathbf{M} = \begin{bmatrix} \alpha & \beta & 0 & \cdots & 0 & \beta & 0 & 0 & 0 & 0 & 0 & 0 \\ \beta & \alpha & \beta & 0 & \cdots & 0 & \beta & 0 & 0 & 0 & 0 & 0 \\ 0 & \beta & \alpha & \beta & 0 & \cdots & 0 & \beta & 0 & 0 & 0 & 0 \\ \cdots & 0 & \beta & \alpha & \beta & 0 & \cdots & 0 & \beta & 0 & 0 & 0 \\ 0 & \cdots & 0 & \beta & \alpha & \beta & 0 & \cdots & 0 & \beta & 0 & 0 \\ \beta & 0 & \cdots & 0 & \beta & \alpha & \beta & 0 & \cdots & 0 & \beta & 0 \\ & \ddots & & & & \ddots & \ddots & \ddots & & & & \ddots \\ 0 & 0 & \beta & 0 & \cdots & 0 & \beta & \alpha & \beta & 0 & \cdots & 0 \\ 0 & 0 & 0 & \beta & 0 & \cdots & 0 & \beta & \alpha & \beta & 0 & \cdots \\ 0 & 0 & 0 & 0 & \beta & 0 & \cdots & 0 & \beta & \alpha & \beta & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta & 0 & \cdots & 0 & \beta & \alpha & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & \beta & 0 & \cdots & 0 & \beta & \alpha \end{bmatrix}. \tag{5.4}$$

The discrete pressure equation can be expressed using $\mathbf{M}$ by letting $\mathbf{L} = \mathbf{M}(-4, 1)$, so the equation reads

$$\frac{1}{\Delta x^2}\mathbf{M}(-4, 1)\vec{q} = \vec{g}. \tag{5.5}$$

The discrete diffusion equation can be expressed using $\mathbf{M}$ by

$$\mathbf{M}(1 + 4r, -r)u_{k+1} = u_k. \tag{5.6}$$

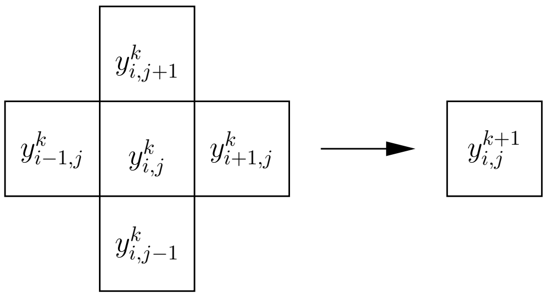


Figure 5.1: The stencil for updating a value of a cell using the Jacobi iterative method.

Many optimized methods exist to solve these systems, and several considerations must be made when choosing a method for use on the GPU. To minimize overhead associated with initiating rendering, it is desirable with a method that requires as few iterations as possible. Also, as boundary conditions are applied after each iteration of the Poisson solver, the number of iterations necessary is crucial for the performance. Another desired property is that the method achieves high arithmetic intensity so the execution is not stalled when waiting for texture reads. Also, readback should be avoided as this generally is a huge performance bottleneck. Last, but not least, the coefficient matrix of the Poisson systems are known in advance and should not be stored on the GPU to save texture lookups. Two iterative solvers have been implemented, namely the Jacobi iterative solver and the successive overrelaxation iterative solver.

5.1 The Jacobi iterative solver

The Jacobi iterative solver is suitable for GPU implementation and is very simple to implement. The iteration is defined as

$$y_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} y_j^k \right), \tag{5.7}$$

which approximates the system $\mathrm{A}\vec{y} = \vec{b}$. The solver requires quite a few iterations to converge satisfactory. The strength of the method is that it is straightforward to implement. It also allows for great control, in the sense that only a few iterations can be executed

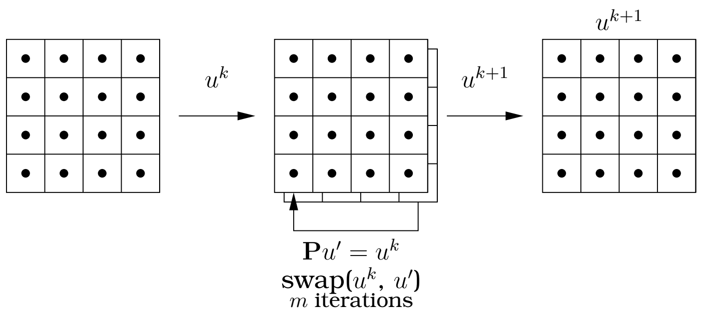


Figure 5.2: The Jacobi solver reads $\vec{u}$ from one texture and writes the updated $\vec{u}$ to another texture, and swaps the two textures after each iteration.

when there is a lack of computational resources or precision is of little interest. Since the solver is explicit, it can easily be implemented on a GPU. For matrices on the form of (5.4), each cell requires five texture lookups per iteration. All required cells are neighbours of a center cell (see Figure 5.3), so it can be assumed that cache hit ratio is high. The cell values are scaled and summed together to form a result. When considering the discrete general Poisson coefficient matrix (5.4) and the Jacobi iteration formula (5.7), the expressions sums to

$$y_{i,j}^{k+1} = \frac{1}{\alpha} b_{i,j} - \frac{\beta}{\alpha}\left(y_{i-1,j}^k + y_{i+1,j}^k + y_{i,j-1}^k + y_{i,j+1}^k\right). \qquad (5.8)$$

The Jacobi solver is implemented using a ping-pong strategy between two buffers. One buffer is used for input and one for output. After each iteration the roles of the buffers are swapped, such that the updated buffer is used as input for the next iteration. Figure 5.2 illustrates the algorithm.

When using the Jacobi iterative solver to approximate a solution for the discrete viscous diffusion equation, both the unknowns $\vec{y}$ and the right hand side $\vec{b}$ are the velocity field $\vec{u}$. Matrix coefficients are $\alpha = 1 + 4\frac{\nu\Delta t}{\Delta x^2}$ and $\beta = -\frac{\nu\Delta t}{\Delta x^2}$. Experiments have shown that 30-40 iterations is a good compromise between performance and accuracy. As mentioned in Section 4.2, for many purposes, fluids can be simulated as an inviscid fluids. For such fluids $\nu = 0$, so the viscous diffusion equation is reduced to $\frac{\partial \vec{u}}{\partial t} = 0$. This cancels out the viscous term in the incompressible Navier-Stokes equations. Due to the relative heavy computations necessary to compute a solution of the discrete viscous equation, this may often be a good option to save computation and thereby increase performance.

The Jacobi solver can also be used to solve the pressure equation, but as optimizations can be done for this equation and the fact that a solution always must be computed in order to perform the projection step, a more efficient solver has been implemented for this purpose.

5.2 The successive overrelaxation solver

The successive overrelaxation (SOR) solver improves Jacobi iteration in two ways. The first improvement is that it always uses the most recent data available. This can be expressed as two sums, one for the already calculated cell values and one for the uncalculated cell values:

$$y_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} y_j^{k+1} - \sum_{j>i} a_{ij} y_j^k \right). \tag{5.9}$$

This iterative method is called the Gauss-Seidel iterative method. It cannot be straightforward parallelized, since the update of a cell depends on cells that may be in the progress of being updated themselves. By exploiting the structure of the Poisson problem, this can be avoided. Each cell is dependent on the value of the cell itself at the previous iteration and the four direct neighbours. In the black-red scheme, every other cell is marked as black and red. This is utilized as red cells only depend on black neighbours and vice versa (see Figure 5.3). Each iteration is split in two, one that uses the red cells to update black cells and one that uses the calculated black cells to update the red ones. This is done using the following scheme:

$$y_i^{k+1} = \begin{cases} \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} y_j^k \right) & \text{black cells} \\ \\ \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} y_j^{k+1} \right) & \text{red cells} \end{cases} .$$

First, the black cells are updated in parallel using values from the previous iteration, and then the red cells are updated using the newly calculated black cells.

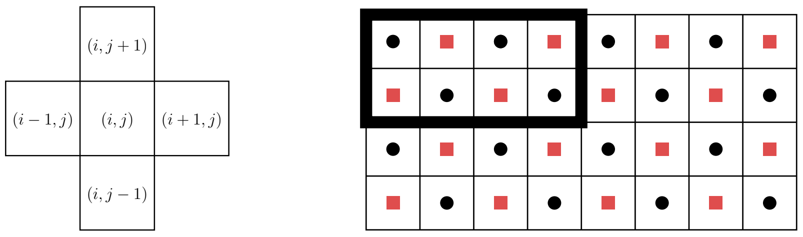


Figure 5.3: The left figure shows Poisson equation stencil. Update of a cell value depends on values of the four direct neighbours and the cell value itself at previous location. The right figure shows the red-black scheme. Red elements only depend on black ones and vice versa.

The second improvement SOR uses is to introduce an overrelaxation factor. The idea behind this is that if a direction c_i^k is a good direction to move to make y_i^k a better solution, the approximation should move further in that direction than it would do when using Gauss-Seidel. The overrelaxation factor ω specifies how much to move in the direction known:

$$y_i^{k+1} = (1-\omega)y_i^k + \omega c_i^k. \tag{5.10}$$

As $\omega < 1$ corresponds to underrelaxation and $\omega = 1$ corresponds to Gauss-Seidel, ω is always larger than 1[1]. It has also been shown that the method fails to converge when $\omega \geq 2$ [Lyc06]. Altogether, the SOR method reads

$$y_i^{k+1} = \begin{cases} (1-\omega)y_i^k + \omega\left[\frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}y_j^k\right)\right] & \text{black cells} \\ (1-\omega)y_i^k + \omega\left[\frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}y_j^{k+1}\right)\right] & \text{red cells} \end{cases}.$$

When extending $\vec{y}$ to 2D and applying this method to the discrete Poisson coefficient matrix (5.4), the expressions sums up to

$$y_{i,j}^{k+1} = \begin{cases} (1-\omega)y_{i,j}^k + \frac{\omega}{\alpha}b_{i,j} - \omega\frac{\beta}{\alpha}\left(y_{i-1,j}^k + y_{i+1,j}^k + y_{i,j-1}^k + y_{i,j+1}^k\right) & \text{black cells} \\ (1-\omega)y_{i,j}^k + \frac{\omega}{\alpha}b_{i,j} - \omega\frac{\beta}{\alpha}\left(y_{i-1,j}^{k+1} + y_{i+1,j}^{k+1} + y_{i,j-1}^{k+1} + y_{i,j+1}^{k+1}\right) & \text{red cells} \end{cases}.$$

When using SOR to approximate solutions to the discrete pressure equation, the unknown field y equals the pressure field q used to

[1]Underrelaxation can be used to establish convergence for diverging iterative processes.

project the velocity field onto its divergence free part. The right hand side b is the divergence of the velocity field, that is $\nabla \cdot \vec{w}$ where $\vec{w}$ is the non-divergence free intermediate velocity field. The matrix coefficients are $\alpha = -\frac{4}{\Delta x^2}$ and $\beta = \frac{1}{\Delta x^2}$.

The overrelaxation-factor ω is central factor when using SOR. It has been proven that the optimal choice for the Poisson problem [Lyc06] is

$$\omega_{\text{opt}} = \frac{2}{1 + \sin(\pi \Delta x)}. \tag{5.11}$$

5.3 Convergence of SOR and Jacobi

This section will investigate the convergence properties and the time complexities for the Jacobi method and the SOR method. First, error estimates are introduced and some plots shown to give an impression of the convergence properties of the two methods. Next, the time complexities are deduced and it is proven that the error when using Jacobi always is larger than, or equal to, the error when using SOR to compute an approximation to the solution of the discrete pressure equation.

Error estimates. The Jacobi iterative solver decreases the error of an approximation to the solution of the discrete pressure equation by a factor $\rho_J(n) < 1$, where $n \times n$ is the size of the domain. It turns out that the error factor can be expressed by [Deh96]

$$e_J(m) \leq (\rho_J(n))^m e_J(0), \tag{5.12}$$

where m is the number of iterations taken, $e_j(0)$ is the inital error, and

$$\begin{aligned} \rho_J(n) &= \cos\left(\frac{\pi}{n+1}\right) && (5.13) \\ &\approx 1 - \frac{1}{2}\left(\frac{\pi}{n+1}\right)^2, \text{ when } n \text{ is large.} && (5.14) \end{aligned}$$

This factor approaches 1 as n grows. The error is defined as the squareroot of the sum of the error squared at each grid-point:

$$e(m) = \sqrt{\sum_{i,j} \left(s_{i,j} - x_{i,j}^m\right)^2}, \tag{5.15}$$

where s is the exact solution and x^m is the approximation after m iterations.

The SOR iterative solver has better convergence properties than Jacobi. When using the optimal relaxation parameter (5.11), it can be shown that a SOR iteration decreases the error in the approximation by a factor [Deh96]

$$\begin{aligned}\rho_S(n) &= \left(\frac{\cos(\pi/(n+1))}{1+\sin(\pi/(n+1))}\right)^2 && (5.16)\\ &\approx 1-\frac{2\pi}{n+1}, \text{ when } n \text{ is large.} && (5.17)\end{aligned}$$

After m SOR iterations, the error is reduced by factor

$$e_S(m) \le (\rho_S(n))^m e_S(0). \qquad (5.18)$$

Both ρ_J and ρ_S approach 1 as $n \to \infty$, so convergence slows down as n grows. However, the rate of slowdown is less for SOR than for Jacobi, as seen in the left plot of Figure 5.4. Note that the axis for number of iterations is logarithmic, so using the SOR method requires an order of magnitude less iterations than using the Jacobi method. The right plot in Figure 5.4 shows the relation between precision and iterations required for a 256×256 grid. From this plot it is clear that relative growth in iterations relative to precision is similar. Since the growth relative to grid size is larger for Jacobi than SOR, and the growth relative to precision is equal for the two methods, it can be stated that SOR requires increasingly fewer iterations relative to Jacobi as grid size increase. Now follows two proofs to support these claims.

Time complexities. Using ρ_J and ρ_S, expressions for the number of iterations m necessary to get a desired precision ξ can be found:

$$\begin{aligned}\xi &> \rho(n)^m,\\ m &< \frac{\ln(\xi)}{\ln(\rho(n))}.\end{aligned}$$

Replacing $\rho(n)$ with $\rho_J(n)$ and $\rho_S(n)$, the approximations for the number of iterations required for the two methods appear:

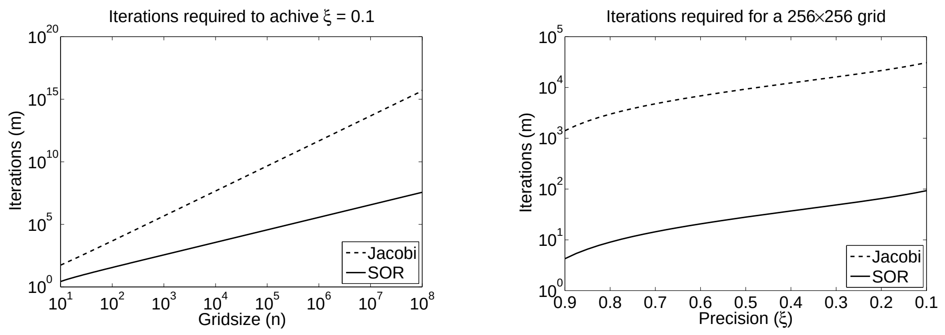


Figure 5.4: The left plot shows the approximate number of iterations required to achieve $\xi = 0.1$ for a given grid size. The right plot shows the approximate number of iterations required to achieve a desired precision for a 256x256 grid.

$$m_J \approx \ln(\xi)/\ln\left(1 - \frac{1}{2}\left(\frac{\pi}{n+1}\right)^2\right) \qquad \text{Jacobi,}$$

$$m_S \approx \ln(\xi)/\ln\left(1 - \frac{2\pi}{n+1}\right) \qquad \text{SOR.}$$

Using these approximations and the identity

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots ,$$

time complexities of the Jacobi and SOR iterative solvers can be established. To reduce the error of an approximation to the discrete pressure equation by a given factor, the Jacobi method requires $O(n^2)$ iterations while the SOR method only requires $O(n)$ iterations [SB93]. Each iteration requires $O(n^2)$ operations, so Jacobi requires $O(n^4)$ operations to achieve a given precision, while SOR requires $O(n^3)$ to achieve the same precision. This is potentially a huge performance boost, and linear increase in iterations required to achieve a given precision allows for simulation on larger grids without too much loss in precision.

Convergence. It can also be proven that the error of using the SOR method with optimal ω always is equal or smaller than the error of using the Jacobi method. Note that this also is a consequence of the time complexities derived above. However, this will be shown by proving that SOR always reduces the error with an factor equal to, or smaller than, the factor from Jacobi (recall that the error is reduced more when the factor is small), or in mathematical terms

$$\rho_J(n) \geq \rho_S(n). \tag{5.19}$$

First, let $\alpha = \pi/(n+1)$ and calculate the difference between $\rho_J(n)$ and $\rho_S(n)$:

$$\begin{aligned}\rho_J(n) - \rho_S(n) &= \cos\alpha - \left(\frac{\cos\alpha}{1+\sin\alpha}\right)^2 \\ &= \cos\alpha\left(1 - \frac{\cos\alpha}{(1+\sin\alpha)^2}\right).\end{aligned}$$

Since $n \in [1, \infty)$ it is clear that $\alpha \in (0, \pi/2]$. Then,

$$\frac{\cos\alpha}{(1+\sin\alpha)^2} \in (0, 1],$$

so

$$\rho_J(n) - \rho_S(n) \geq 0. \tag{5.20}$$

This proves that the convergence using the SOR method always is better, or as good, as when using the Jacobi method. In addition, it is clear from the approximations (5.14) and (5.17) that the error reduction factor for the Jacob method approaches 1 much more rapidly than the reduction factor for the SOR method.

5.4 A GPU-optimized SOR solver

Since computing the solutions to the Poisson equations is such a major part of the simulation, it is crucial that the solvers are as efficient as possible. I have implemented an SOR solver optimized for the pressure equation. Extending the implementation to solve the implicit viscous diffusion equation will be straightforward.

The SOR solver utilizes the red-black structure of the Poisson problem by packing the data in a manner efficient for lookup and updating values. It also uses a large overrelaxataion parameter ω, thereby increasing the rate of convergence. The SOR solver is implemented on the GPU as three separate operations:

1. Pack the divergence texture.

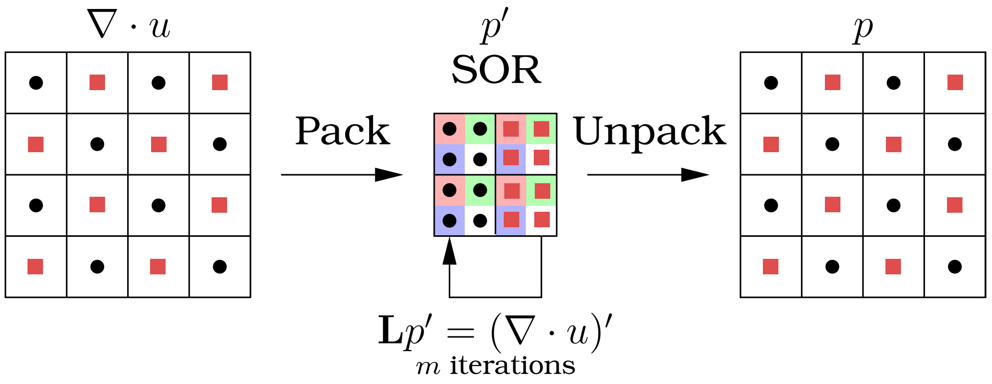
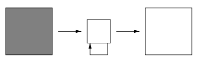


Figure 5.5: The SOR algorithm. Divergence is packed in a red-black fashion, and m iterations of SOR is performed to find packed pressure, p'. This texture is unpacked to form the result, pressure p.

2. Compute the solution to the discrete pressure equation using SOR.
3. Unpack the pressure texture for usage later.

Figure 5.5 illustrates how the algorithm works. Each of the steps is implemented using a separate shader. Now follows an introduction to these shaders, where the figures in the margin below refers to Figure 5.5.

Packing. The first shader converts a single-component texture into a four-component texture suitable for red-black computation. More specifically, the shader is used to pack the pre-calculated divergence to the format used by the SOR solver. The packing scheme used packs a scalar field by separating red and black cells to separate parts of an intermediate texture. More specifically are black cells stored in the left half of the texture and red cells are stored in the right half. Next, 2×2 blocks of the intermediate texture are stored in a single texel holding four values, reducing the size of the texture holding the data from $n \times n$ to $n/2 \times n/2$. The actual implementation does not utilize an intermediate texture. Figure 5.6 illustrates the result of the packing.



The packing algorithm is based on a fragment program that calculates coordinates and fetches the texels to store in its R, G, B and A-components. If c is the coordinate of the fragment that is

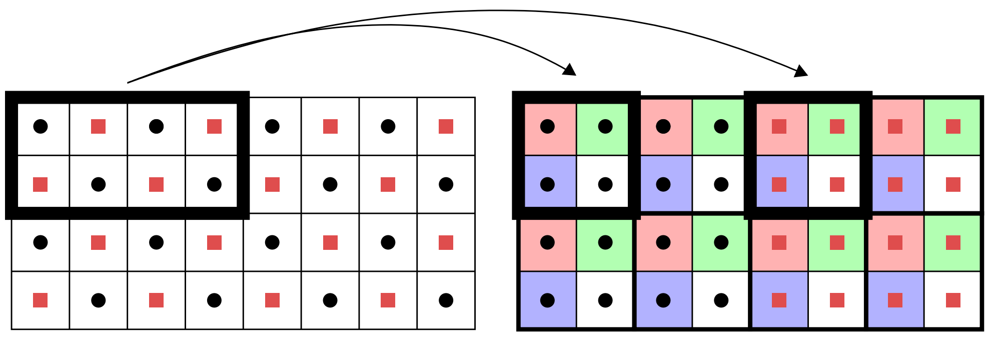


Figure 5.6: Packing of the divergence and pressure texture. The texture is converted from a single-component texture (left) into a four-component texture (right) of half the size in each dimension. The texture is split in two parts, one for red elements and one for black elements, making updates very efficient.

being processed and dim is the dimension of the input texture, the algorithm for packing the red or black cells of a 2×4-block into a single fragment is:

1. Determine if the fragment is in the red or black part of the output by considering if the x-coordinate is more or less than half the dimension (black if less).
2. Calculate coordinate vector. This vector points to the coordinate of the texel that will be stored in the red component.
   - For fragments holding black cells this is $\langle 4c.x, 2c.y \rangle$.
   - For fragments holding red cells this is $\langle 4(c.x - dim/2), 2c.y \rangle$.
3. Fetch texels from input texture. The following offsets are used to fetch values for the respective components (see Figure 5.7):
   - For fragments holding black cells: R: $\langle 0, 0 \rangle$, G: $\langle 2, 0 \rangle$, $B:$ $\langle 1, 1 \rangle$ and A: $\langle 3, 1 \rangle$.
   - For fragments holding red cells: $R:$ $\langle 0, 0 \rangle$, $G:$ $\langle 2, 0 \rangle$, $B:$ $\langle -1, 1 \rangle$ and $A:$ $\langle 1, 1 \rangle$.

Note that the actual computation is performed without branching to maximize performance.

By packing in this manner, cache hit ratio should be improved as the input texture range can be specified to cover only the texels

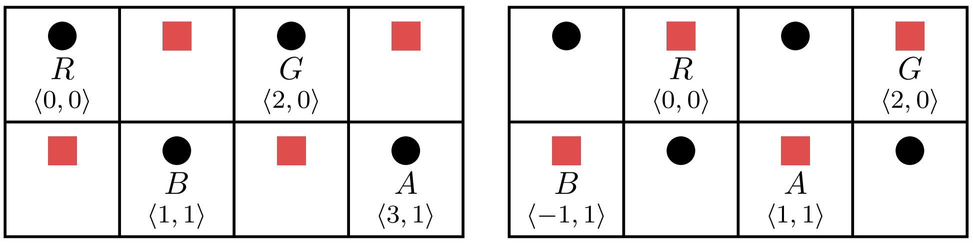
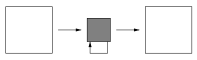


Figure 5.7: Illustration of the offsets used when fetching texels to a black fragment (left) and a red fragment (right).

necessary for calculation (that is, red texels when computing black ones, and vice versa). Specifying the texture range improves pre-fetching, thus improving the performance. Another advantage is that only half the domain needs to be processed when computing the values of cells of a given color in the SOR iteration. If no packing were to be used, the whole domain would have to be processed when calculating values of cells of each color. In this case, values for all cells of the unselected color would have to be discarded. This can be done at an early stage in the fragment shader, but due to the SIMD nature of the fragment processors, the discarding processors would have to wait for the rest of the processors to complete before new fragments can be processed. When assuming that the processors process fragments in a spatially coherent block-wise manner, roughly half of the fragment processors would be idle or processing results that would be discarded. Packing also enables the GPU to utilize its full processing potential, by utilizing the four long vectorized design of the GPU. Therefore, the red-black-packing is a very good method of packing the data for the SOR algorithm.



SOR iteration. After the divergence field has been packed in a red-black fashion, multiple SOR iterations are performed. Each SOR iteration is split into one pass for updating black cells using red cells, and one for updating red cells using the newly computed black cells. Since cells are separated by their color, updating cells of a given color is done by drawing a quad covering the part of the texture that conforms to this color. This effectively filters out cells from being processed when they should not be updated. Figure 5.8 illustrates one SOR iteration.

When using the red-black scheme, the new value of a cell is de-

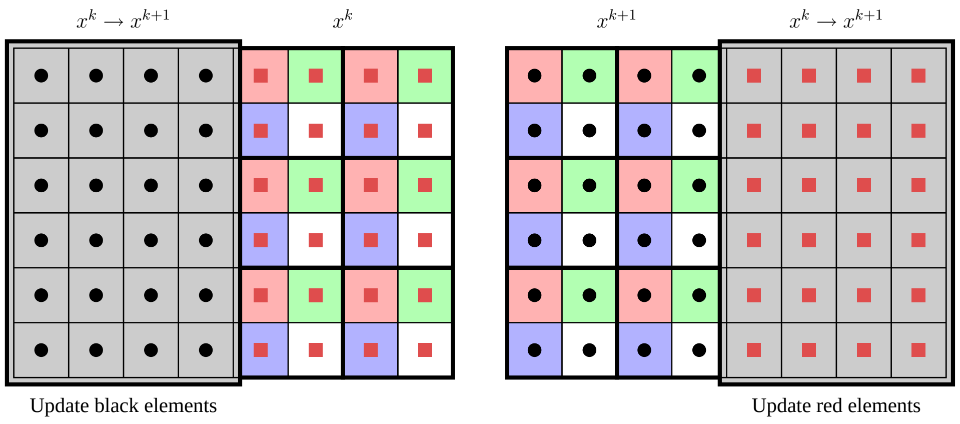


Figure 5.8: One SOR iteration. The iteration is split into two passes, one that updates the black cells to timestep $k + 1$ using red cells computed for timestep k (left), and one that updates the red cells to timestep $k + 1$ using the newly calculated black cells (right).

pendent on the previous value of the cell and its direct neighbours which is of the opposite color of the cell itself. Due to the packing scheme used, neighbours are located in the half of the texture that is not being processed. Five texels are fetched from this texture to retrieve the neighbours of the cells of the fragment being processed. In addition the texel conforming to the cells being updated are fetched. Figure 5.9 illustrates the texels that contains neighbours in the packed texture. Each of the cells of the current fragment are updated using different components of the fetched texels. Figure 5.10 shows two stencils for updating the result. Notice that the stencil differs when updating the red and the black

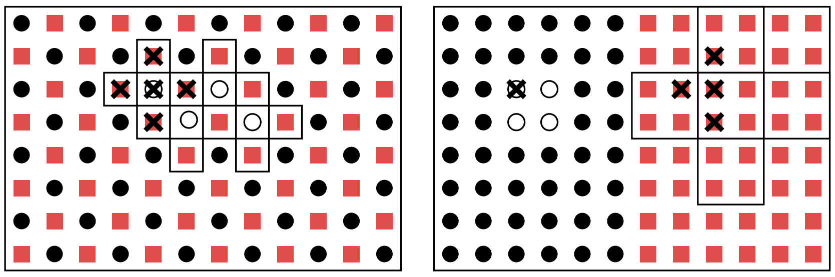


Figure 5.9: Illustration of what texels to fetch when updating the cells marked with hollow circles. Crosses marks a selected cell and its neighbours in both figures. Note that not all elements of the texels fetched in the packed case is used to update the marked cells.

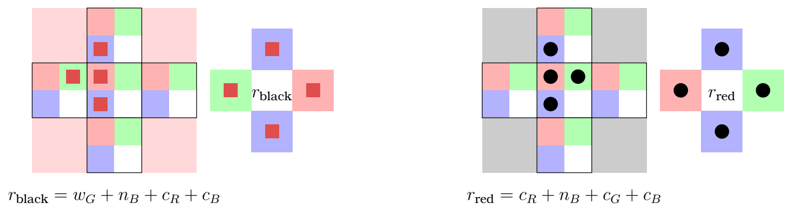

r_{black}
$r_{\text{black}} = w_G + n_B + c_R + c_B$
r_{red}
$r_{\text{red}} = c_R + n_B + c_G + c_B$

Figure 5.10: Stencils for updating the packed pressure texture. The left stencil is for updating the R-component of the black cells, and the right stencil is for updating the R-component of the red cells. The variables refer to the direction from the center cell, that is north, east, south, east or center. The subscripts refer to the components of the texels, that is R, G, B or A.

cells. Listing 5.1 lists some of the code used by the SOR shader. Note that the texture coordinates used are computed earlier in the shader.

As mentioned in Section 5.2, the theoretical optimal relaxation factor is

$$\omega_{\text{opt}} = \frac{2}{1 + \sin(\pi \Delta x)}.$$

However, using this value in the SOR shader leads to "unnatural" behavior and artificially unstable flow. This may be a consequence of using discrete arithmetic. Using

$$\omega = \frac{1.9}{1 + \sin(\pi \Delta x)},$$

works fine and convergence is still very good. By default, the simulator uses 15 SOR iterations to compute solutions to the pressure equation. This results in satisfactory results with respect to visual quality and computational time.

Unpacking. After SOR has completed, the results are unpacked for utilization by the projection step. This step subtracts the gradient of the pressure field from the velocity field, resulting in an incompressible field. The resulting pressure field may also be used for visualization purposes.

The unpacking algorithm performs the inverse transformations as the packing algorithm performs, such that cells are positioned as they was before they was packed using the red-black packing

```
        // Lookup x_ij and right-hand side b
        const vec4 self = texture2DRect(p, gl_TexCoord[1].xy);
        const vec4 b = texture2DRect(div, gl_TexCoord[1].xy);

        // Neighbouring texels
        const vec4 c = texture2DRect(p, cc);
        const vec4 n = texture2DRect(p, nc);
        const vec4 w = texture2DRect(p, wc);
        const vec4 s = texture2DRect(p, sc);
        const vec4 e = texture2DRect(p, ec);

        // Neighbour stencils
        vec4 dir;
        dir.r = isred*(c.r + c.b + c.g + n.b) +
                isblack*(n.b + c.b + c.r + w.g);
        dir.g = isred*(n.a + c.a + c.g + e.r) +
                isblack*(c.r + c.g + c.a + n.a);
        dir.b = isred*(c.b + c.r + s.r + w.a) +
                isblack*(c.r + c.a + c.b + s.r);
        dir.a = isred*(c.b + c.a + c.g + s.g) +
                isblack*(c.a + c.g + s.g + e.b);

        // SOR iteration
        gl_FragColor = self + 0.25*omega*(dir - 4.0*self - dx*dx*b);
```

Listing 5.1: SOR shader. Two sets of texture coordinates are used, one for fetching divergence values and values from the previous pass, and one for fetching values of the opposite color. `isred` and `isblack` are constant per pass and either 0 or 1. This is used to avoid `if`-tests.

scheme. The algorithm can be considered as first converting each fragment of the four-component texture to 2×2 blocks of an intermediate single-component texture. Red and black cells are still separated in the intermediate texture. This texture is unpacked by positioning every other red and black cell in the final output, starting with black cells for odd rows and red cells for even rows (counting from one). Just as the packing algorithm, the actual implementation does not utilize an intermediate texture in order to maximize performance.

Boundary handling. Boundary handling will be discussed further in Chapter 6, but note that boundaries have not been implemented for the SOR pressure solver. The reason for this is that a new mechanism for applying boundaries will have to be implemented in order to support boundaries for packed textures. Due to the time limitation, this has not been implemented. This leads to "leakage" through the obstacle cells, and must be implemented for any real-life applications.

5.5 Results

The performance gain from packing pressure and divergence in a red-black fashion and using an optimized SOR solver is quite high. Using a 50-iteration Jacobi iterative solver on a 512×512 grid, the simulation runs steadily at about 15 frames per second. Using a 20-iteration SOR solver for pressure, the simulation runs at about 30 frames per second on an NVIDIA GeForce 7800GT. In addition, the SOR solver converges faster than the Jacobi solver, resulting in a more accurate result.

Since the SOR solver does not support boundaries, all performance results presented are measured without boundary handling for both the Jacobi and the SOR solver. However, since the boundaries are applied after each iteration in the iterative solver, the cost is proportional to the number of iterations required. Since Jacobi requires more iterations to achieve an accurate result, the cost of applying boundary conditions to Jacobi will be higher than the cost of applying boundaries for the SOR solver.

The divergence after projection is known to be exactly zero on the

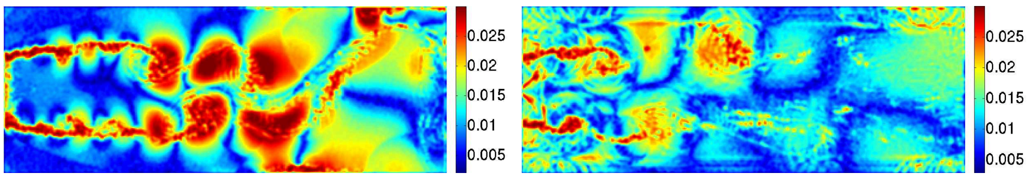


Figure 5.11: Error in the resulting velocity field when using Jacobi (left) and SOR (right) after 500 time steps using the same initial conditions. The plots shows the square root of the absolute error to increase contrast. Red means high error and blue means low error.

entire domain, due to the continuity equation (4.19). Since the projection-step is very simple, it will not introduce any mentionable error other than the error for using discrete math. Non-zero divergence value is a consequence of the error in the iterative method used to compute the pressure. The divergence of the velocity field after the projection step can therefore be used as a measurement of the error of the method used to approximate the pressure. Figure 5.11 shows the error in divergence field after the projection step for both the Jacobi and the SOR solver. The figure shows the error in the cover of an obstacle creating unsteady flow. Using the SOR method gives a slightly smaller maximum error than Jacobi for the segment of the simulation shown, more precisely the Jacobi results in a maximum error that is 26% larger than the maximum error using the SOR solver. The total error is much larger for Jacobi than for SOR. The total sum of the absolute error using SOR for the segment of the result shown is 58% larger when using the Jacobi solver over the SOR solver. Note that only a segment of the domain is checked, due to erroneous values near the boundaries as boundaries are disregarded by the optimized SOR solver. It is clear that the SOR method leads to more correct results, which generally results in a better visual result. Figure 5.12 shows an example of this.

5.5.1 Comparison between Jacobi and SOR

Achieving the same accuracy using the Jacobi solver as the SOR solver achieves is not possible when requiring simulations to be performed real-time. Therefore, the accuracy requirement is lowered when using Jacobi to allow real-time simulation. More specifically,

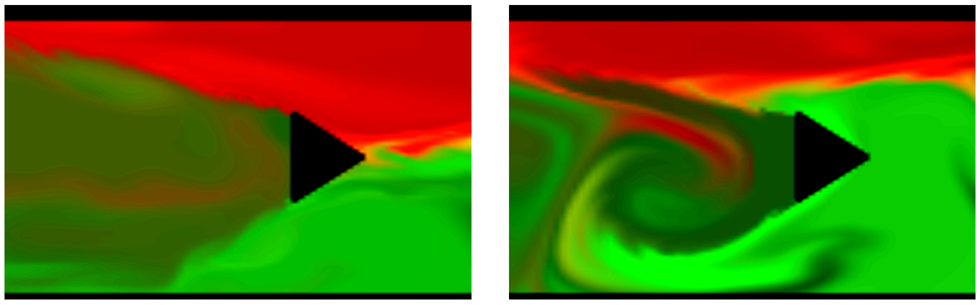


Figure 5.12: Comparison between simulations using a 40-pass Jacobi solver (left) and a 15-pass SOR solver (right) for computing pressure. The inflow and surroundings are the same for both simulations. SOR tends to give a more lively simulation due to better convergence, which is clearly shown in the screenshots.

Jacboi and SOR are compared using a 8:3 ratio in the number of iterations used. This ratio is chosen because using 40 Jacobi iterations produces results comparable to the result when using 15 SOR iterations.

Performance gains achieved using the optimized SOR solver over the Jacobi solver to solve the discrete pressure equation is substantial. Using SOR typically results in speedups in range 100-200% over Jacobi. Table 5.1 lists some of the results observed when measuring the speedup of using the optimized SOR solver over the Jacobi solver. The trend is clear; the SOR solver performs increasingly better than the Jacobi solver as the number of iterations and grid size increases.

Very much of the performance gain is a result of the packing and other optimizations that have been implemented. A plain red-black implementation of SOR without packing only increases the performance by a fraction of the gain experienced when using the optimized SOR. As stated in Section 5.4, this is a result of the fact that roughly half of the fragment shaders will remain idle at all times due to the structure of the problem for the unoptimized SOR solver. Another reason is that the optimized SOR solver utilizes the four-vector design of the GPUs, the plain red-black implementation only utilizes one fourth of the potential processing power. Typically, speedup using an unoptimized SOR solver is only about 5-10% compared to the Jacobi solver. Even though there is some overhead associated with packing and unpacking textures, the op-

Table 5.1: Performance gain using SOR over Jacobi for different choice of number of iterations and grid sizes. The timings are measured after 500 time steps.

N	Steps (Jacobi/SOR)	Jacobi	SOR	Speedup
256	40/40	4932 ms	4059 ms	22%
256	40/15	4932 ms	2590 ms	90%
256	53/20	6474 ms	2945 ms	120%
256	80/30	9722 ms	3425 ms	180%
512	40/15	18411 ms	6261 ms	194%
512	53/20	24330 ms	6861 ms	254%
1024	40/15	72695 ms	25038 ms	190%

timized version has always proven more efficient, even when using as few as five iterations. One can expect substantial speedup by using packing-techniques for Jacobi too, but the convergence properties remain better for the SOR solver, which therefore should be preferred.

5.5.2 Analysis of the optimized SOR solver

The major bulk of the work done by the SOR solver is in the SOR computation itself. Packing and unpacking data is relatively cheap in comparison. For a simulation using 15 SOR iterations on a 512×512 grid, the distribution of the work done by the solver is found in Table 5.2. Note that the packing could have been avoided in some sense if the divergence calculating shader and projection shader supported the packed format. However, for visualization and implementation purposes this is left out. The performance gain for doing so would probably be mentionable, but not comparable to the gain already achieved by using SOR over Jacobi, only < 15% compared to the current optimized solver.

Table 5.2: Distribution of work for the SOR solver using on a 512×512 grid after 500 time steps.

Shader	**Time**	**Relative load**
Pack divergence	691 ms	11.7%
SOR calculation	4844 ms	82.4%
Unpack result	346 ms	5.9%

5.6 Other solvers

Many specialized methods exist to compute solutions of the discrete Poisson equation. Several of these methods theoretically outperform the SOR solver. A Fast Fourier Transform (FFT) solver will directly solve the discrete Poisson equation exactly (not considering floating-point round-off errors). This is a very efficient solver that can compute the solution in $O(\log(n^2))$ operations given n^2 processors [Deh02]. Generally, we are not interested in an exact solution for real-time simulation and the FFT-solver may therefore be a bad choice for this purpose.

Multigrid is a very efficient iterative method. It is not as efficient as the FFT-solver given n^2 processors to process data, but the number of elements will far exceed the number of processors available, even on GPUs. The multigrid solver converges to a given precision in $O(n^2)$ operations on serial architectures. In other words, the multigrid solver is theoretically optimal [Deh02].

The presented iterative solvers only transfer information to its direct neigbours. For proper convergence, information has to travel back and forth several times. The idea behind multigrid is to use extrapolated approximations from coarser grids as initial guesses for the finer grids. The coarse grids are usually of half the dimension of the finer grid, that is if the original domain Ω^h is of dimension $N \times N$, the coarser grid Ω^{2h} is of dimension $N/2 \times N/2$. This reduces the number of elements to one quarter. Using extrapolated coarser approximations as initial guesses for the coarser problems is applied recursively. At the coarsest level an optional method is used to compute the approximation to the solution. The process is illustrated in Figure 5.13. Since each multigrid iteration requires $O(n^2)$ operations and the total running time for the solver also is $O(n^2)$, it is clear that the method converges in a constant number of iterations. For an introduction to multigrid methods I refer to

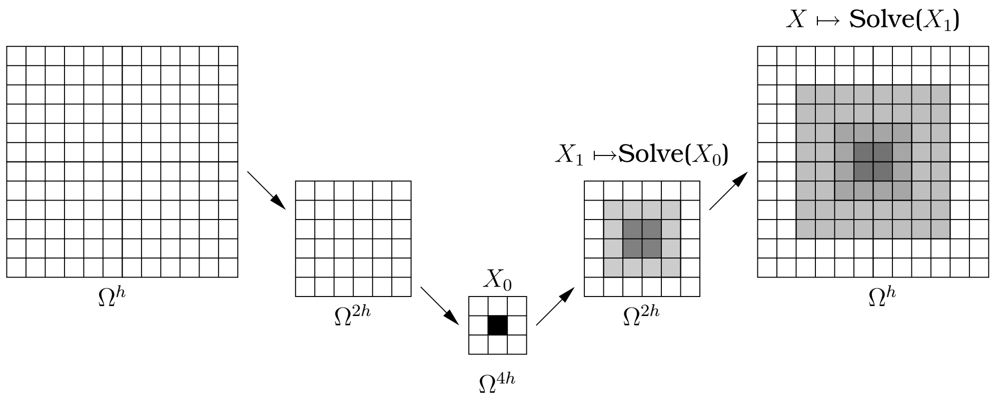


Figure 5.13: The idea behind multigrid. Extrapolation of the approximation of a solution of the coarser version of the problem is used as an initial guess.

Briggs [Bri].

Chapter 6

Boundary conditions

Simulation on a simple rectangular grid with some fixed boundary is rarely sufficient for real applications. Often it is desirable with internal obstacles and support for different types of boundary conditions. For some applications it is also desired to have objects following the flow, e.g., bottles flowing through water and getting stuck in eddies. A natural extension to this is that the objects themselves act as obstacles. This requires extra care as the boundaries are changing, requiring extra work at each time step. Boundary conditions must be applied after each step in the incompressible Navier-Stokes simulator. The Poisson solvers presented in Chapter 5 require boundaries to be processed after each iteration. This means that the boundary conditions must be applied a large number of times each time step, so the mechanism for doing this must be as efficient as possible.

6.1 Boundaries of arbitrary shape

Applying boundaries of arbitrary shape is a complex operation and some approximations has been made to allow real-time simulation. The boundaries are approximated using a small set of linear segments that are joined together to form a piecewise linear curve. Approximating arbitrary boundaries with a small set of linear segments reduces the number of different cases that must be handled to only a few. This is exploited by precomputing normals and tangents for the set of linear segments available. The precomputed

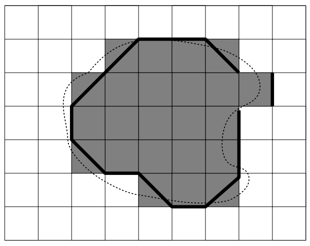


Figure 6.1: Boundaries (dotted, thin curve) are drawn using OpenGL calls and rasterized to pixels (grey squares). From this texture an approximated piecewise linear curve (bold, black lines) is constructed.

normals and tangents are stored in a texture for rapid lookup, and used when applying boundary conditions. The boundary mechanism consists of two parts, one algorithm for precomputing the piecewise linear curve that approximates the boundaries and one algorithm for using the precomputed curve to apply boundary conditions. Now follows a detailed introduction to how the algorithms work.

Approximating the boundary curve. Boundaries may be drawn by loading a pre-generated image of the boundaries or using regular OpenGL vertex calls. The result is rasterized to form a texture where zeros represent obstacles/boundaries and ones represent interior cells. The generated texture is processed by a fragment shader that forms an approximation of the obstacles by joining eight types of constant curves together (the unit vectors on the x and y axis in both directions and the four diagonals). The approximated curve formed (see Figure 6.1) is never used directly, only the normals and tangents will be used, so it is only necessary to store these attributes for each cell. However, instead of storing the normals and tangents directly, a cell code is stored for each cell in a texture instead. This code is calculated by convolving the boundary mask texture with a kernel that is based on the neighbourhood of each cell (see Figure 6.2). The kernel checks if the direct neighbours (north, south, east and west) are obstacles and generates an unique code between 0 and 31 for each layout. Figure 6.3 shows an example of the process. Note that cells lying on the outer border will not pose a problem as texture coordinates are clamped to the domain. Due to this clamping, neighbours outside the domain are treated as obstacles if the cell on the outer border is an obstacle.

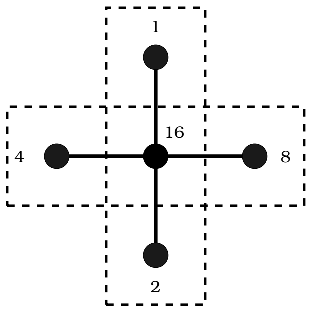
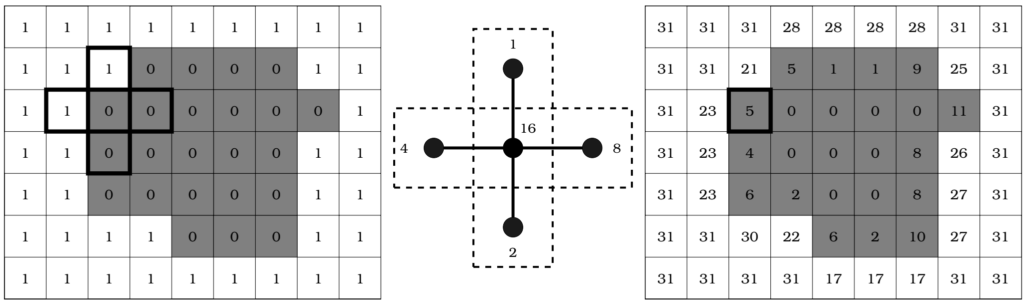


Figure 6.2: Boundary code pattern. The pattern is convolved over an underlying texture where 1 represents interior and 0 represents an occupied cell, generating an unique code for each case.

Therefore, an obstacle on the outer border will never have a normal pointing outside the domain.

Precomputed normals and tangents. The generated code texture is later used to lookup normals and tangents from a precomputed texture when applying boundary conditions. The precomputed lookup texture is generated offline and contains normals and tangents for all approximated cases. Because tests showed that there was virtually no difference in performance between using a precomputed lookup texture and storing the cell code and storing the normal and tangent directly, I have chosen to store the



1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	1	1
1	1	0	0	0	0	0	0	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	0	0	0	1	1
1	1	1	1	1	1	1	1	1

31	31	31	28	28	28	28	31	31
31	31	21	5	1	1	9	25	31
31	23	5	0	0	0	0	11	31
31	23	4	0	0	0	8	26	31
31	23	6	2	0	0	8	27	31
31	31	30	22	6	2	10	27	31
31	31	31	31	17	17	17	31	31

Figure 6.3: Generation of cell codes. First a texture where zero means boundary and one means interior is created (left). Next a kernel (the middle) is convolved with the generated texture generating a code for each element, resulting in a texture with cell codes (right).

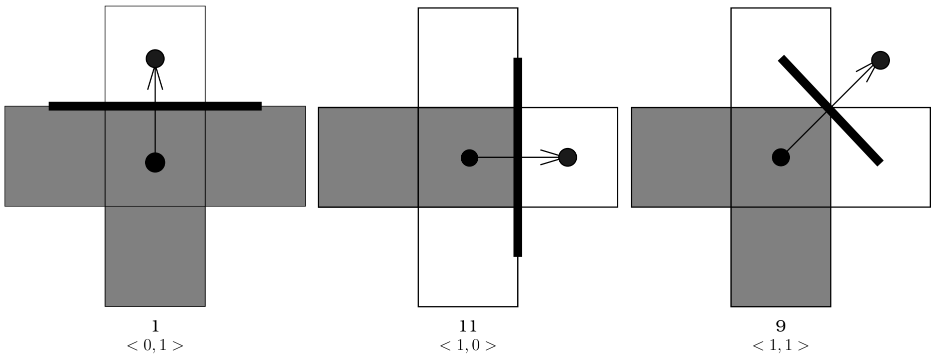


Figure 6.4: The left image shows the case for code 1 and the middle image shows the case for code 11. For this case the normal is set to $< 1, 0 >$ since east is between the two other choices. The right image shows a case where the normal is set to chose an element that has not been checked if it is an obstacle or not. The thick lines are the local curve approximations.

code and use a precomputed lookup texture. This is a bit more flexible approach, e.g., since different normal/tangent textures may be applied for different operations. Another advantage of storing the code is that it only requires a single-element texture making it very compact. The code texture is represented using a two-component 16 bits floating point native [NVI05, page 39] texture format and a four-component 16 bits format is used to store the precomputed normals and tangents. The code is stored in the red-channel, the use of the green-channel is explained later.

The precomputed normals and tangents are stored as a 4×4 texture containing 16 elements, one texel for each case where the center-cell is an obstacle, that where the cell contained by the fragment is an obstacle. Cells with code larger than 15 are interior cells and should not be affected by boundary conditions. The texture contains four components per texel, where $\langle R, G\rangle$ represents the normal and $\langle B, A\rangle$ represents the tangent corresponding to a given code. The precomputed normals and tangents are determined by considering what neighbours are obstacles for each of the cases. Figure 6.4 shows a few examples from the precomputed texture. In the 3D case the tangent and normal cannot be packed into a four component texture and some trick will have to be used in order to represent normal and tangent, e.g., can two textures be utilized or by representing each case by two texels where one texel contains the normal/offset while the other contains the tangent. It would also be possible to only store the normal and calculate the tangent on the fly in the shaders.

Most types of boundaries perform some operations on a neighbouring cell to determine the value of the boundary cell. For such calculations, an offset is necessary in order to determine what neighbouring cell to get values from. In this case the nearest neighbour in the normal direction is used, that is, the normal is also used as an offset.

The small size of the pre-computed lookup texture results in that it easily fits completely in texture cache making the penalty of normal/tangent lookup very small. This is why storing the actual normal/tangent instead of the cell code is not more efficient than storing the cell code.

Ambigious cases. As mentioned above, the algorithm for code generation only checks the neighbours directly north, south, east and west. The performance penalty of checking the complete neighbourhood would be neglible when considering the amount of work done by the rest of the computation each time step, but there is only a few cases that suffer from problems when discarding neighbours on the diagonal. When two or more elements adjacent to a boundary are marked as interior, the case is ambiguous and several offsets are valid. The offset chosen is the one in the direction that lies in the middle of the possibilities. When two opposite offsets are valid, e.g., north and south, one of them is chosen. This case only occurs when using one-pixel wide obstacles, which is adviced against since such obstacles will appear to leak. The reason for this is that the value of each cell is calculated from its neighbours and cells in one-pixel wide obstacles will be used for calculation on both sides of the obstacle. The case where two neighbouring cells share a corner (north and west, south and west, etc.) and are open, the offset is chosen to be the sum of the offsets these cells would have generated if they were the only possibility (see Figure 6.4). The ambiguous cases can lead to problems for some rare cases, but in practice this is a good solution when the obstacles do not contain thin lines. Figure 6.5 shows an example of an obstacle this could have caused some problems for. However,

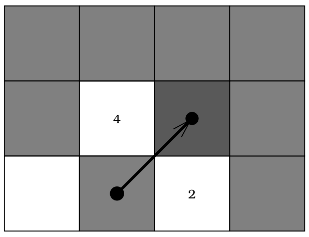


Figure 6.5: Boundary case 6. East and south element is marked as interior. In this case offset is set to $< -1, -1 >$ (south-east) even though this element is an boundary

since the main goal is visual quality this is not an intolerable drawback. There are only a few cases that will suffer from this flaw, and this will most probably not be visible anyways. Therefore, I have chosen not to look at the complete neighbourhood as this complicates the calculation of the offset-texture significantly as the size of the offset texture increases from 16 texels to 256, making it somewhat necessary to write a program to generate the normals and tangents. This texture would probably be too big to fit completely in cache (recall that the cache probably is of size 8×8), thereby reducing performance.

Applying boundaries. When applying boundaries, the boundary code is looked up from the boundary texture. If the code is greater than 15 the element is not a boundary and the fragment is discarded, leaving the value of the field that boundaries are applied to unchanged. If not, the offset is looked up from the precomputed offset texture. This returns four values, where the two first represent the spatial offset to lookup the new value from. In order to support zeroing of the interior of the obstacles, the output may be scaled with `clamp(code, 0.0, 1.0)` which corresponds to 0 for the interior of obstacles and 1 else. This is typically useful when applying boundary conditions for iterative methods where non-zero values in the interior of the obstacles will spread as more iterations are applied.

The computations related to boundary conditions are performed solely on the GPU so no readback to the CPU is necessary. Since the application supports moving obstacles, the boundaries may change at each time step. Therefore, the boundaries are drawn and the boundary code texture is recalculated after each time step.

6.2 Types of boundary conditions

In addition to obstacles of arbitrary shapes, different types of boundary conditions supported is generally required to get a desired effect. The application supports four types of boundaries: no-slip, free-slip, outflow and inflow. At no-slip boundaries, no fluid penetrates and the fluid is at rest at the boundary. This means that the layer of fluid near the surface sticks to the surface. If σ_n is the component of the velocity normal to the boundary and σ_t is the component in the tangential direction, the no-slip boundary condition can be described mathematically by

$$\sigma_n(x, y) = \sigma_t(x, y) = 0. \tag{6.1}$$

Free-slip conditions are similar to no-slip, except that there is no friction at the boundary. A free-slip boundary imposes no drag on the fluid in contrast to no-slip. Mathematically free-slip is defined as

$$\sigma_n(x, y) = 0, \qquad \frac{\partial \sigma_t}{\partial n}(x, y) = 0. \tag{6.2}$$

On an inflow boundary, the velocity is given such that the boundary acts as a source of velocity:

$$\sigma_n(x, y) = \alpha, \qquad \sigma_t(x, y) = \beta. \tag{6.3}$$

Outflow boundaries simply let the flow pass through with unchanged velocity, so

$$\frac{\partial \sigma_n}{\partial n}(x, y) = \frac{\partial \sigma_t}{\partial t}(x, y) = 0. \tag{6.4}$$

In addition to these, the Poisson solver for pressure utilizes pure Neumann boundaries, meaning that the change in pressure at the boundary is zero, or

$$\frac{\partial p}{\partial n}(x, y) = 0. \tag{6.5}$$

The type of boundary used may be specified in two ways in the application implemented. As mentioned above, boundaries are specified by drawing zeros and ones to the red-channel of a two-component texture. By utilizing the green-channel as a boundary

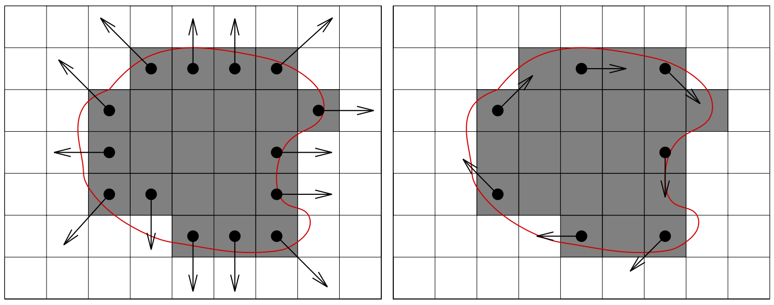


Figure 6.6: Approximation of normals and tangents. The normals (left) is set to the offset generated for a given cell. I have chosen to use the tangents in the clockwise direction (right).

type specifier, boundaries of different types may be drawn to the texture. The shader generating the code texture now has access to both mask and type of the boundaries by reading both components of the mask-texture. The type-flag is simply preserved through the generation of cell code from the mask and is stored in the green-channel of the code-texture. The type flag is read when applying boundaries, and the boundary is treated according to the type.

Discretized versions of the boundary conditions are necessary when implementing them. Arbitrary boundaries are supported so the boundary conditions must be supported for all shapes, not just the simple boundaries that surround the domain. The conditions are not directly defined in the $\langle \vec{e}_x, \vec{e}_y \rangle$-direction, but in the $\langle \vec{n}, \vec{t} \rangle$-direction where $\vec{n}$ is the normal and $\vec{t}$ the tangent of the obstacle. From now on, these bases will be called $\mathbb{XY}$ and $\mathbb{NT}$ respectively, and mappings from $\mathbb{XY}$ to $\mathbb{NT}$ and back are necessary when applying the boundary conditions. A vector $\vec{\sigma} \in \mathbb{XY}$ can be mapped to $\vec{\sigma'} \in \mathbb{NT}$ and back again using

$$\vec{\sigma'} = \mathbf{P}^{-1}\vec{\sigma}, \tag{6.6}$$

$$\vec{\sigma} = \mathbf{P}\vec{\sigma'}, \tag{6.7}$$

where

$$\mathbf{P} = \left[\vec{n}, \vec{t}\right].$$

As discussed in Section 6.1, the normal is defined by the local approximation of the boundary. The tangents are stored in the same texture as the normal are stored, where the R- and G-components

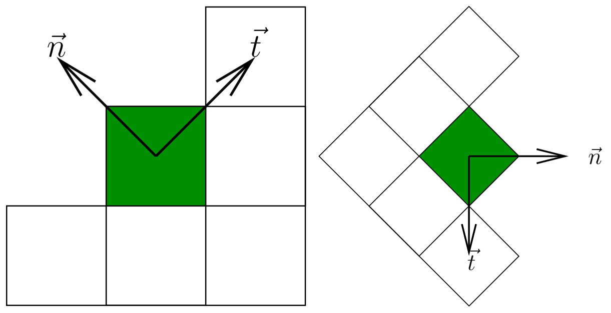


Figure 6.7: Mapping from $\mathbb{XY}$ (left) to $\mathbb{NT}$ (right).

contain the normals and the tangents are stored in B and A. Given $\vec{n}$ and $\vec{t}$, the algorithm for applying the boundary conditions can be implemented in the following manner:

1. Read the neighbouring element $\vec{\sigma_n} = \vec{\sigma}(x + o_x, y + o_y)$ where $\vec{o}$ is the neighbour offset of the current element.
2. Calculate a projection matrix $\mathbf{P} = \left[\vec{n}, \vec{t}\right]$ and its inverse $\mathbf{P}^{-1}$.
3. Change basis of $\vec{\sigma_n} \in \mathbb{XY}$ to the normal-tangential basis $\mathbb{NT}$ (see Figure 6.7) by
$$\vec{\sigma'_n} = \mathbf{P}^{-1}\vec{\sigma_n}.$$
4. Apply boundary conditions in the $\mathbb{NT}$-basis. This step will be discussed for the different types of boundaries later. The result is stored as $\vec{\sigma'_c}$.
5. Change basis back from $\vec{\sigma'_c} \in \mathbb{NT}$ to $\mathbb{XY}$ using
$$\vec{\sigma_c} = \mathbf{P}\vec{\sigma_c}'.$$
6. Store $\vec{\sigma_c}$ as the new value at the boundary.

All of these steps are performed by one shader. Cells that are not marked as obstacles are simply discarded, thereby saving output bandwidth. This requires the use of an if-statement that potentially could be inefficient. However, as boundaries often is of a certain size it can be assumed that the there is some spatial coherence, reducing the chance of idle processors as a result of different branches being processed. Discarding fragments also frees memory bandwidth for the other processors, reducing memory latency.

6.3 Discrete boundary conditions

Now the discretizations of the different types of boundary conditions can be discussed, starting with no-slip.

6.3.1 Discrete no-slip boundary conditions

Recall the no-slip boundary condition:

$$\sigma_n(x, y) = \sigma_t(x, y) = 0 \tag{6.8}$$

For boundaries that are aligned with the x- or y-axis this is simply obtained by setting the x component to zero in the case of a boundary aligned with the y-axis, and vice versa. The non-zero component is set to the negate of the respective component of the neighbouring element. However, since arbitrary boundaries are supported, things are a bit more complicated. Boundaries are defined to lie on the edge between the boundary cell and the nearest neighbour. Since the value at the boundary should be zero, the average between the neighbour and the boundary element that is processed should be zero in the normal and tangential direction, or

$$\begin{aligned} \left[(\vec{\sigma}_n')_n + (\vec{\sigma_c}')_n\right]/2 &= 0, \\ \left[(\vec{\sigma}_n')_t + (\vec{\sigma_c}')_t\right]/2 &= 0, \end{aligned} \tag{6.9}$$

where $(\vec{\sigma}_n')_n$ and $(\vec{\sigma}_n')_t$ are the normal and tangential components of the neighbour. $(\vec{\sigma_c}')_n$ and $(\vec{\sigma_c}')_t$ are the same components of the current element. From the discussion above it is known that $\vec{\sigma}_c'$ has to be computed in $\mathbb{NT}$ and mapped to $\mathbb{XY}$. By assuming that the neighbouring element in tangential direction also is a boundary, the boundary condition for the tangential component is satisfied by always setting it to zero. The normal component is set to the negate of the respective component of the neighbouring, yielding

$$\vec{\sigma}_c' = \begin{bmatrix} (\vec{\sigma_c}')_n \\ (\vec{\sigma_c}')_t \end{bmatrix} = \begin{bmatrix} -(\vec{\sigma_n}')_n \\ 0 \end{bmatrix}. \quad \text{(Discrete no-slip)} \tag{6.10}$$

Clearly this satifies (6.9). Figure 6.8 shows an example on how the no-slip boundary condition is applied.

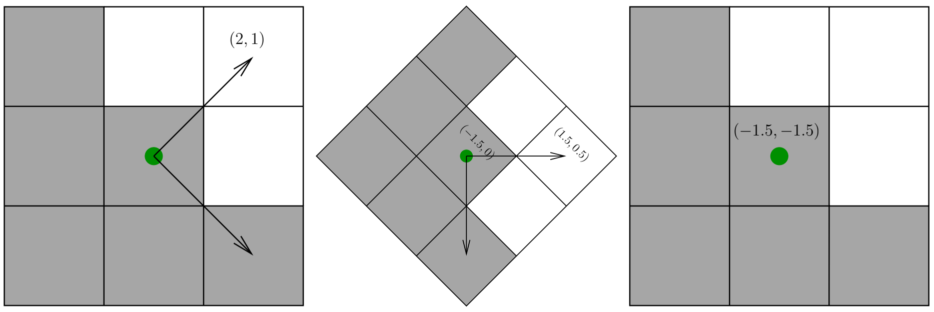


Figure 6.8: Applying no-slip boundary conditions. First the normal and tangent is determined (left) and the problem mapped from $\mathbb{XY}$ to $\mathbb{NT}$. The conditions are applied in $\mathbb{NT}$ (middle). Last, the new value of the cell is mapped back to $\mathbb{XY}$ and stored (right).

6.3.2 Discrete freeslip boundary conditions

Freeslip boundary conditions (6.2) are similar to no-slip and can be treated in the almost same manner. Discretized the condition reads

$$\begin{aligned} \left[(\vec{\sigma}'_n)_n + (\vec{\sigma_c}')_n\right]/2 &= 0, \\ \left[(\vec{\sigma}'_n)_t - (\vec{\sigma_c}')_t\right]/h &= 0. \end{aligned}$$

For free slip, no assumption about the element in the tangential direction is necessary and by solving for $\vec{\sigma_c}'$, the condition reads

$$\vec{\sigma}'_c = \begin{bmatrix} (\vec{\sigma_c}')_n \\ (\vec{\sigma_c}')_t \end{bmatrix} = \begin{bmatrix} -(\vec{\sigma_n}')_n \\ (\vec{\sigma_n}')_t \end{bmatrix}. \quad \text{(Discrete free slip)} \tag{6.11}$$

This is translated to $\mathbb{XY}$ using (6.7).

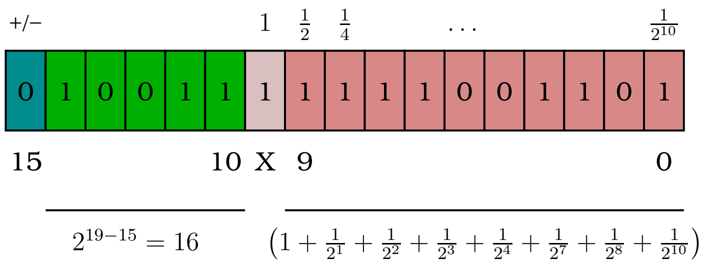
6.3.3 Discrete outflow boundary conditions

Outflow (6.5) is discretized in the same way as no-slip and free slip. The discretized version reads

$$\begin{aligned} \left[(\vec{\sigma}'_n)_n - (\vec{\sigma_c}')_n\right]/h &= 0 \\ \left[(\vec{\sigma}'_n)_t - (\vec{\sigma_c}')_t\right]/h &= 0 \end{aligned}.$$

Solving for $\vec{\sigma_c}'$ yields

$$\vec{\sigma}'_c = \begin{bmatrix} (\vec{\sigma_c}')_n \\ (\vec{\sigma_c}')_t \end{bmatrix} = \begin{bmatrix} (\vec{\sigma_n}')_n \\ (\vec{\sigma_n}')_t \end{bmatrix}. \quad \text{(Discrete outflow)} \tag{6.12}$$



+/−
1 $\frac{1}{2}$ $\frac{1}{4}$ . . . $\frac{1}{2^{10}}$
0 1 0 0 1 1 1 1 1 1 1 0 0 1 1 0 1
15 10 X 9 0
$2^{19-15} = 16$
$(1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^{10}})$

Figure 6.9: Binary 16 bit floating-point representation of 31.21. The number cannot be stored exactly, and the result is 31.203125.

6.3.4 Discrete inflow boundary conditions

Inflow cannot be specified in the same manner as the other boundary conditions as both the magnitude and direction of the inflow have to be specified. A straightforward solution is to represent the inflow velocities in a separate texture, but this complicates specification of boundaries as two textures have to be painted to for inflow boundaries. Instead, a decomposition of the green component of the boundary code texture into magnitude and direction is used. The integer part (`floor(`G`)`) represent the magnitude multiplied with a constant and the remaining fraction (G - `floor(`G`)`) represent the angle divided by 2π. In order to control precision, an upper limit for magnitude (`MAXMAG`) and a magnitude granularity (`MAGSCALAR`) is defined. The magnitude of a vector is clamped to the supported range and divided by the granularity. I have set `MAXMAG` to 3.0 and `MAGSCALAR` to 30.0 so magnitude is scaled to $[0.0, 30.0]$. In order to separate the inflow from the other types of boundary conditions, 1.0 is added to the G-component such that the component is guaranteed to be greater than a given threshold.

The algorithm for specifying inflow converts a vector to angle and magnitude and packs these values into a float. Calculation from float to vector is performed very frequently in the boundary shader and must be very efficient. The loss of precision in this conversion is considerable, but acceptable as the visual effect of the error is very small. In order to examine the precision of the result, it is necessary to look at the representation of 16 bit floating point numbers. Recall from Section 2.2.4 that such numbers are represented in the `s10e5` format on NVIDIA GeForce series 6 and 7. Also recall that the absolute precision is best near zero. Since the transformation splits the number into decimal and fractional part, it is important to remain some precision in the fractional part. This is the reason for keeping the result relatively small. Figure 6.9 shows the representation of 31.21, a relative high value when considering the maximum which is 32. The unbiased exponential part is 4, biased by 15 it is 19. The fraction part should now be as close as possible to $31.21/2^4 = 1.950625$. Since the hidden bit represents 1, the fraction bits should represent 0.950625. The most significant bit represents $\frac{1}{2}$, the next bits $\frac{1}{4}$ through $\frac{1}{2^{10}}$. The closest approximation is 1.1111001101 in binary or as a decimal, 1.9501953125. Since multiplying with 16 corresponds to shifting the decimal point four digits to the right in binary, the result can be represented as 11111.001101. The left bits represents the magnitude while the right bits represents the angle. 32 numbers can be represented using five bits, so in worstcase the angle has precision of $360^o/32 = 11.25^o$, but for inflows of magnitude $<$ 3.0 the precision is much better.

Since `glColor` clamps each component to $[0.0, 1.0]$, `glTexCoord` is used to specify boundaries. A fragment shader that writes the unclamped texture coordinate to the color component of the output is used to generate the mask texture.

Better packing methods can be developed, however, for this purpose the method described is sufficiently accurate. Note that this conversion only supports the 2D case, but there is possible to create similar transformations for 3D vectors with further decreased precision. This will require use of 32 bits textures instead of 16 bits.

Listing 6.1 lists the code for conversion from vector to a float and Listing 6.2 lists code the conversion back again.

6.3.5 Pure Neumann boundary conditions

The Poisson pressure equation requires pure Neumann conditions when computed [Har03]. This condition reads

$$\frac{\partial p}{\partial \vec{n}} = 0,$$

where p is the pressure field and $\vec{n}$ is the normal to the boundary. This condition can be approximated using

$$\frac{\partial p}{\partial \vec{n}} \approx \frac{p_n - p_c}{h},$$

```c
/**
 * Returns a value between [1.0, MAGSCALAR] for any input v.
 * Magnitude is clamped to [0.0, MAXMAG] if necassary.
 */
float inflowToFloat(const float x, const float y) const {
        const float len = sqrt(x*x + y*y);

        // Shift and scale magnitude to [0.0, MAGSCALAR] to preserve
        // some decimals
        float sMag = (MAGSCALAR / MAXMAG) * clamp(len, 0.0f,
                MAXMAG);
        // Resolve angle
        float angle;
        if (x > 0 && y >= 0)
                angle = atan(y/x);
        else if (x > 0 && y < 0)
                angle = 2*M_PI + atan(y/x);
        else if (x < 0)
                angle = M_PI + atan(y/x);
        else if (x == 0 && y > 0)
                angle = 0.5f*M_PI;
        else // if (x == 0 && y < 0)
                angle = (3.0f/2.0f)*M_PI;
        angle /= (2.0f*M_PI);

        // Form a float mag.angle in [0,MAGSCALAR]
        const float result = round((MAGSCALAR/(MAGSCALAR +
                        1.0f))*sMag) + angle;
        // Shift result to 1.0f+
        return result + 1.0f;
}
```

Listing 6.1: Function that converts a vector to float (C code)

```
vec2 floatToInflow(float val)
{
        val -= 1.0;

        float magPart = floor(val);
        float mag = MAXMAG  * magPart / (MAGSCALAR + 1.0);
        float angle = (val - magPart) * 2.0*M_PI;

        // Recreate vector
        return mag*vec2(cos(angle), sin(angle));
}
```

Listing 6.2: Function that converts from float to vector (GLSL code)

where p_n is the neighbouring pressure value and p_c is the value of the cell itself. This can very easily be implemented using

$$p_c = p_n. \tag{6.13}$$

This condition will only be used by the pressure solver, and can therefore be implemented directly by the solver. By reading the boundary code and type from the boundary code texture, the condition can simply be applied in the same pass as computation of new cell values. Each cell is checked to discover boundary cells that are not inflow or outflow boundary cells. These types are ignored, as they do not represent solid obstacles. If a cell is an obstacle cell, the neighbouring offset is looked up just like by the general boundary mechanism. However, as the pressure field is a scalar field, the value can simply be copied from the neighbour according to (6.13) instead of going through a series of transformations.

The performance penalty for using such a mechanism to handle boundaries is small compared to the penalty when using the general boundary mechanism. Branching is used to separate interior cells and obstacle cells, so the performance penalty grows as the complexity of the boundaries increases. This is due to the SIMD nature of the processors, making spatial incoherent branches more expensive than coherent ones. However, tests using 2000 random distributed small obstacles showed that the penalty of incoherent obstacles was next to nothing. On a simulation on a 512×512 grid with relative complex boundaries, a Jacobi iterative solver using 40 iterations runs at about 20 frames per second when not handling boundaries. Using the general mechanism for applying boundary conditions, the frame rate drops to about 6 frames per second. The penalty for using the specialized mechanism is much less, as the simulation runs at about 13 frames per second using the embedded boundary handling. The time spent to calculate pressure drops from about 80% to about 62% when using this mechanism. Note that the values used as neighbours by the optimized mechanism actually are values from the previous time step. This is a result of the parallel nature of the GPUs. As several values are computed simultaneously, there is no guarantee for the values necessary to apply the boundary conditions are calculated yet. Alternatively, the mechanism could have been implemented as an optimized shader that only supports Neumann conditions. Applying boundaries after each iteration would allow using values from the same step when applying boundaries. However, much of the performance penalty associated with the general boundary mechanism is a result of rendering initiating overhead. A separate, optimized mechanism for Neumann conditions would be subject for this overhead too.

6.4 Non-stationary obstacles

In addition to stationary boundaries, objects flowing through the flow acting as obstacles are also supported. The simulation of these flowing obstacles is implemented purely on the GPU using shaders. This section will first go through the representation and simulation of the obstacles, and how collision detection with stationary obstacles is handled. Then, the mechanism making objects act as obstacles and the visualization technique are explained. The mechanism is implemented for the 2D case, but can be extended to the 3D case using the same ideas.

6.4.1 Representation

All attributes of the flowing obstacles are stored in textures. Two textures are used, one called `PropData` and another one called `ObjectData`. Table 6.1 lists the attributes in the two textures and in what components of the textures the attributes are stored in. The position is a spatial coordinate between $\langle -1, -1\rangle$ and $\langle 1, 1\rangle$,

Attribute	Texture	Component(s)
Position	`PropData`	R, G
Angular position	`PropData`	B
Size	`PropData`	A
Velocity	`ObjectData`	R, G
Mass	`ObjectData`	B
Angular velocity	`ObjectData`	A

Table 6.1: Different flowing obstacle attributes and how they are stored.

while the angular position is measured in radians. The mass is a dimensionless quantity implemented to support inertia. All flowing obstacles are represented as quads, and size is measured as distance from the center to the corners of the flowing obstacles in terms of texels.

6.4.2 Simulation of movement

Several considerations must be made with respect to movement of the flowing obstacles. Two types of movement should be supported, spatial and angular. Another important aspect is collision detection between flowing obstacles and stationary obstacles, so flowing obstacles do not pass through stationary obstacles.

The implementation of movement is pretty straight forward. Velocity $\vec{v}_{k+1}$ of the flowing obstacles are computed using

$$\vec{v}_{k+1} = \vec{v}_k + \frac{\Delta t}{m + \Delta t} \left(\vec{v}_{flow} - \vec{v}_k \right), \tag{6.14}$$

where $\vec{v}_k$ the current velocity, $\vec{v}_{flow}$ the current value in the underlying velocity-field and m is the mass of the flowing obstacle. Note that Δt is added to the mass to support mass less flowing obstacles, where changes in velocity is immediate.

Computation of angular velocity α is similar to the computation of spatial velocity:

$$\alpha_{k+1} = \alpha_k + \frac{\Delta t}{m + \Delta t} \left(\kappa \omega_{flow} - \alpha_k \right), \tag{6.15}$$

where ω_{flow} is the value of the underlying vorticity-field. The field is scaled with κ to give some control on how much the flowing obstacles rotate. Tests have shown that $\kappa = 0.1$ results in natural angular movement. The formulas used here have no physical origin, but yields realistic behaviour. The angular and spatial velocity approach the values in the underlying fields as time pass (assuming the fields are relatively monotone), which naturally is expected.

Using the computed values for $\vec{v}$ and α makes computing the change in spatial position, $\vec{p}$, and angular position, θ, a straight forward task:

$$\vec{p}_{k+1} = \vec{p}_k + \Delta t \vec{v}_k, \tag{6.16}$$

$$\theta_{k+1} = \theta_k + \Delta t \alpha_k. \tag{6.17}$$

By using modulus, obstacles flowing outside of the domain (i.e. at outflow boundaries), will reappear on the opposite side of the domain again.

Using two render targets, one shader is sufficient to calculate the new velocities and positions of the flowing obstacles in a single pass. Extra care must be taken when implementing the mechanism as the flowing obstacles act as obstacles themselves. Several problems arise from this.

Erroneous values in underlying fields. Since the flowing obstacles act as obstacles themselves, the underlying fields will not have values at the position of the flowing obstacles, or the values are not valid. Therefore, velocity and vorticity are averaged from the values on the corners of the flowing obstacles (see Figure 6.10) instead.

Small values in cover of obstacles. Another problem that arises from using the flowing obstacles as boundaries and using the average technique mentioned above, is that there most likely will form an area behind the flowing obstacles that contains small values as the area is in cover of the obstacles. Since the averaging technique uses values outside all corners of the object, the resulting value will probably be too low or even negative of values on the opposite side (see Figure 6.11). As the mechanism already is a brute estimate of the actual movement of objects, this is simply avoided by dividing by 3 instead of 4 when averaging the values. This will often result in an over-estimate of the correct value, but for visual purposes this is a sufficient solution to the problem.

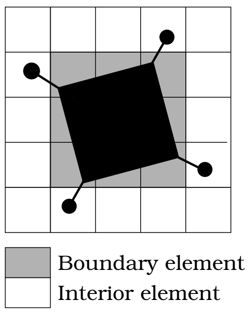


Figure 6.10: Fetching velocity and vorticity values from underlying fields. Since flowing obstacles act as obstacles themselves, the values will be erroneous underneath the obstacles. Instead, values are fetched from texels one element outside each corner of the obstacle and averaged.

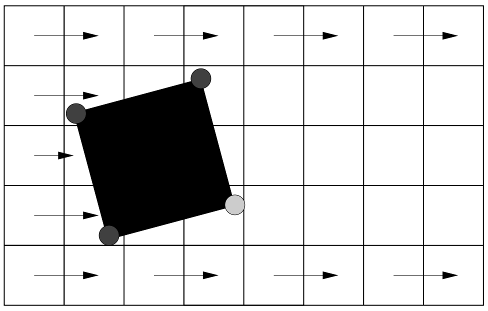


Figure 6.11: Eddies form behind objects and results in small values or even values with the opposite sign compared to the values in front of the objects.

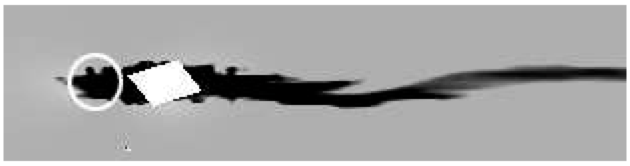


Figure 6.12: An artificial "tail" (marked with a circle) with small values in the velocity field in front of a moving object. The flowing obstacles flows from left to right. The tail must not be mistaken with the area with naturally small values in the cover of the obstacle.

Obstacles leave a footprint. When flowing obstacles are advected through the flow, they will leave a trace of small values in its path of the underlying fields since they act as obstacles (see Figure 6.12). This happens because the values in the cover of the objects are small, as stated in the previous paragraph. When flowing obstacles move over this area, the small values are preserved. As objects move along, this area will appear again, resulting in the appearance of a "tail" with small values behind the moving objects. The effect of this is not visually conspicuous when considering fields not directly related to the simulation (e.g., a density field). However, for a better simulation this problem should be avoided by computing the values of the fields beneath the objects in some manner, as the "tail" will result in loss of energy.

6.4.3 Avoiding stationary obstacles

It is crucial that the flowing obstacles are able to avoid stationary obstacles, such that they do not flow through walls etc. De Chiara et al. [ECST04] used a force field around obstacles to simulate obstacle avoidance for a flock of simulated birds. This idea will be used here too. The force field is generated using a multi-pass approach. First, the initial field is generated by looking up in the boundary code texture, generating a field with vectors pointing outwards from the obstacles. The type of boundary per cell is considered, such that outflow and inflow boundaries do not appear as solid walls. Next, the force field is smoothed using multiple passes. The 3×3 neighbourhood of a cell is smoothed using a Gaussian

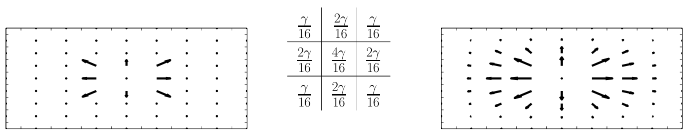

$\frac{\gamma}{16}$ $\frac{2\gamma}{16}$ $\frac{\gamma}{16}$
$\frac{2\gamma}{16}$ $\frac{4\gamma}{16}$ $\frac{2\gamma}{16}$
$\frac{\gamma}{16}$ $\frac{2\gamma}{16}$ $\frac{\gamma}{16}$

Figure 6.13: Force field before smoothing (left), the stencil used to smooth the field (middle) and after three passes of smoothing (right).

filter:

$$\mathbf{G} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}. \tag{6.18}$$

In addition, an accelerator γ parameter is introduced. This can be used to achieve a force field where the magnitude grows as more passes are applied. Increased magnitude is often desirable to get sufficiently large force near the obstacles. Tests have shown that an accelerator parameter $\gamma = 1.5$ generally results in vectors that grows enough to keep flowing obstacles advecting through stationary obstacles. The smoothing will result in a force field with gradually increasing force when approaching obstacles (see Figure 6.13). The number of passes used to smooth the force field varies, but the larger the velocities involved in the simulation are, the more passes must be performed to create a sufficiently strong force field. However, the more passes is performed, the earlier will the force be applied as flowing objects are approaching stationary obstacles. This is not always desirable, e.g., will stationary obstacles that are close interfere with each other. For most purposes, ten passes seems to be a good choice.

The force is applied by the same shader that calculates new velocities and positions. When objects move parallel to, or away from obstacles, no force should be applied. This is determined using the dot-product between the value of the normalized force field and the normalized velocity of the flowing obstacle:

$= 0$, the object is moving parallel to the obstacle, no adjustment to the velocity is performed.

> 0, the object is moving away from the obstacle, no adjustment to the velocity is performed.

< 0, the object is approaching the obstacle, adjust velocity such that the object does not crash into (and possible through) the stationary obstacle.

The result of the product is negated and clamped between 0 and 1 to gradually apply force, rather than using an on/off function. Mathematically, the operation of applying obstacle force can be expressed as

$$\vec{v}_{\text{obstacle}} = \vec{v}_{k+1} + \phi \Delta t \Delta x \cdot \psi(\vec{v}_k, \vec{f}) \cdot \vec{f}, \tag{6.19}$$

where ϕ is a parameter to control how much force is applied (set to 4 in the implementation, which leads to natural behavior in the test cases used), and

$$\psi(\vec{v}, \vec{f}) = \texttt{clamp}\left(-\frac{\vec{v} \cdot \vec{f}}{|\vec{v}||\vec{f}|}, 0, 1\right), \tag{6.20}$$

is the scalar to ensure force only is applied when objects are approaching stationary obstacles. This mechanism works fine for most simulations, but for problems that include high velocities or stationary obstacles that are very close to each other (causing the force fields of the obstacles to interfere) some adjustments may be required.

6.4.4 Visualization and updating the boundaries

Although severing two different purposes, visualization and updating the boundaries are highly related and performed by the same algorithm.

The visualization technique is rather crude. The `PropData`-texture for flowing obstacles is read back to main memory to obtain position, angular position and size. The texture is used to draw the objects using OpenGL calls. Some other technique would probably be more efficient, such as using pre-initialized vertex buffers with accompanying texture coordinates, leaving specification of the actual vertex coordinates to the vertex shader instead. However, due to simplicity of implementation, the readback approach has been implemented instead.

Since the obstacles are moving, the boundary textures must be updated after each time step. The mechanism for doing this is rather simple. Just as stationary obstacles may be initiated by drawing them onto a boundary mask texture through OpenGL calls, moving obstacles are simply drawn onto the same texture using the same algorithm used to visualize the objects. Next, the boundary code is calculated using the same mechanism explained in Section 6.1.

6.4.5 Results

The simulator is able to handle a large number of moving obstacles. 2000 objects can be simulated without any mentionable performance loss. Using the SOR solver to compute pressure, simulating 2000 number of objects on a 512$\times$512 grid uses less than 2% of the total simulation time. However, updating boundaries takes somewhat longer than when not simulating any rigid objects, and the time spent by updating boundaries increases from about 5% to 12%. This can be improved by implementing a more efficient rendering technique.

The boundary handling mechanism performs well and seems to not be as dependent on the layout of the obstacles as expected. The mechanism for applying boundary conditions uses branching to determine if a cell is an obstacle cell. Spatially coherent branches are preferred on GPUs, as all fragment processors must process the same instruction at all times. However, tests have shown that the penalty for incoherent boundaries is small, leading to predictable performance.

The simulation is rather robust, and in most cases collisions between moving and stationary obstacles are handled flawlessly. The movement of the objects seems plausible, although the equations for calculating the velocity of the objects are not physically correct. In order to make the simulation of the moving obstacles feel more natural, they should be able to interfere with each other to some extent so they do not flock together in a tight spot as they sometimes have a tendency to do in the current implementation. Even so, for many purposes the techniques presented are sufficient. The mechanism can, e.g., be used to simulate light obstacles, such as leafs that advects through a wind field. Another application is simulation of obstacles following the water flow, getting stuck in eddies and avoiding stationary obstacles, e.g., rocks.

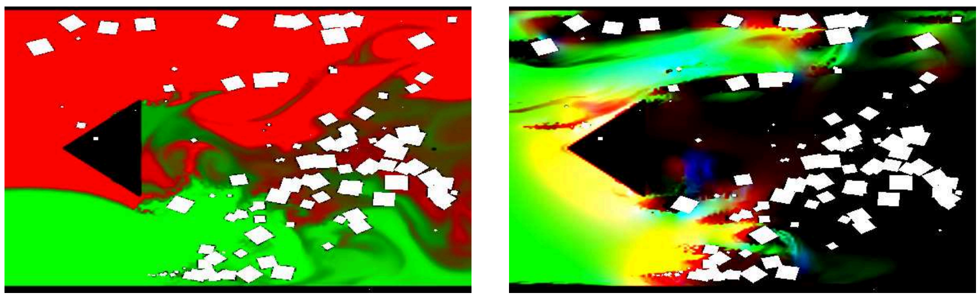


Figure 6.14: An excerpt of the result of a simulation with 2000 moving obstacles. The left screenshot shows the density field, and the right field shows the pressure field (red) and the velocity field (green and blue) at the same number of time steps.

Chapter 7

Summary and results

The aim of this thesis is to present a simulator for the incompressible Navier-Stokes equations that focuses on visual quality rather than physical correctness. The computation is performed solely on the GPU, avoiding expensive readback. Support for arbitrary boundaries is implemented as it is important for real-life applications to support general domains and interaction with the environment (e.g., objects passing through the domain). To support applications such as objects flowing through a water flow, a mechanism for obstacles moving through the flow is implemented. It is crucial that the implementation is efficient, allowing real-time simulation in addition to real-time rendering of the scene the simulation is a part of.

The implementation of the incompressible Navier-Stokes simulator on the GPU presented is based on the semi-Lagrange discretization of the equations. This discretization is unconditionally stable with respect to Δt, which can be exploited by allowing arbitrary adjustment of the length of the time step. The time step may be adjusted to the available processing power by skipping simulation for most rendered frames and executing simulation passes with larger time steps every tenth frame for instance. Arbitrary time steps also allows for synchronization of the simulation with some other time scale (e.g., the time scale of a game).

The semi-Lagrange discretization of the incompressible Navier-Stokes equations used leads to numerical dissipation. Vorticity confinement has been implemented to reinject energy where it is lost, and preserves small vortices in the flow. This improves the visual qual-

Table 7.1: Workload for the different simulation operations after 1500 time steps on a 512×512 grid using an optimized 15 iterations SOR solver and a Jacobi solver using 40 iterations to compute pressure. Boundary conditions were not handled when computing pressure, as SOR does not support this. The fluid simulated is inviscid (no diffusion term) and no moving obstacles are simulated.

Operation	*Jacobi*		*SOR*	
	Time (ms)	**Load**	**Time (ms)**	**Load**
Calculate pressure	54995	78.7%	18423	55.3%
Apply boundaries	6266	9.0%	6272	18.8%
Vorticity confinement	2134	3.1%	2152	6.5%
Update boundaries	1800	2.6%	1850	5.5%
Advect velocity	1199	1.7%	1198	3.6%
Add sources	1056	1.5%	1062	3.2%
Projection	866	1.2%	862	2.6%
Calculate divergence	764	1.1%	764	2.3%
Calculate vorticity	738	1.1%	759	2.3%

ity of the result, making the flow become more “alive”.

The simulator is implemented using a technique called operator splitting for projection. This results in a simulator consisting of several stages, each representing one term in the incompressible Navier-Stokes equation. Each stage consists of a number of separate operations and together these stages form the complete simulator. Some of the operations are very lightweight while other operations require heavy computations.

Table 7.1 lists the workload of the different operations after 1500 time steps. Clearly, the majority of the work performed is computing the pressure. The second most expensive operation is the boundary mechanism. Therefore, it is desirable that these operations perform as efficiently as possible. The main areas of focus in the thesis have therefore been to optimize these steps.

The majority of the computations performed involves solving the Poisson equations that arise from the discretization. An iterative SOR solver optimized for solving the pressure equation has been implemented. This solver has resulted in good performance gains over the iterative Jacobi solver also implemented. The convergence properties of the SOR solver results in much more accurate results than the Jacobi solver achieves using the same number of iterations.

Arbitrary boundaries are approximated using a set of predefined linear segments. By considering if the neighbours of a cell are obstacle cells or not, an unique cell-code depending on the surroundings of each cell is established. This code is computed for every cell in the grid used by the simulation and stored in a texture that is used to efficiently apply boundary conditions. Several types of boundary conditions are supported, including freeslip, no-slip, outflow and inflow.

The simulator also supports obstacles that follow the flow. The movement of the flowing obstacles is calculated using approximations that results in plausible movement. Collision detection between stationary obstacles and flowing obstacles has been implemented using force fields constructed around the stationary obstacles.

7.1 Discussion

This section discusses the research questions asked considering the results observed. First, recall the research questions asked in the introduction:

1. Is it plausible that GPUs will be used to handle physics computations in the future, and more specifically, fluid dynamics simulations?
2. How can solutions of the Poisson equations that arise from the semi-Lagrange discretization of the Navier-Stokes equations be computed efficiently on the GPU?
3. Can arbitrary stationary and non-stationary obstacles be incorporated in a real-time simulator for the incompressible Navier-Stokes equations?

Question 1 - GPU physics. Recently, the GPU has been utilized to compute physics. Computational frameworks for physics simulation such as Havok are being adapted to use GPUs [Har06] as the computational engine. AGEIA has released dedicated hardware for performing physics [AGE], but only a handful of games supports hardware accelerated physics. GPU vendors such as ATI and NVIDIA have noticed the interest for dedicated physics computations, and will probably continue to improve support for physics simulation. Effect physics are very closely related to graphics, and performing simulations directly on the GPU saves data transfer. The parallel nature of much physics makes the GPU architecture suitable for physics simulation. As GPUs become increasingly programmable, they can easier be utilized for such computations. These trends indicate that GPUs will be used for more physics simulations in the future, and especially physics related to purely visual effects.

CFD simulations require relatively much computational power, and it is only in the recent years real-time simulations have been possible. Much research has been presented to make real-time simulations possible, and several GPU implementations of CFD simulators have been presented. Harris [Har03] has presented a GPU implementation that simulates the incompressible Navier-Stokes equations in GPU Gems, a book that has received much attention from the game development community. The Havok physics framework also supports some fluid dynamics, a clear indication of the attention from game developers for such effects. As GPUs become more powerful, it is increasingly likely that they will be used to simulate fluids in games too.

Question 2 - Poisson solvers. Several solvers for the discrete Poisson problem have been presented in this thesis. The Jacobi method is the simplest of the iterative solvers, but suffers from bad convergence properties. Boundary conditions must be applied once per iteration to ensure that the result remains valid, leading to an extra penalty for using many iterations. The SOR method has better convergence properties than the Jacobi method. Therefore, the SOR solver requires fewer iterations to converge to a satisfactory accuracy. A straightforward implementation of SOR leads to poor utilization of the processing power of the GPUs, so an efficient packing scheme has been implemented. Typically, performance gains in the range 100-200% are experienced when using the optimized SOR solver over the Jacobi solver. However, the packing scheme used makes implementation more difficult. Therefore, boundary support for the optimized SOR solver has been left out due to time limitations.

The SOR solver is a fairly efficient solver for the discrete Poisson equation. However, several theoretically more efficient solvers exist. Multigrid converges even faster than SOR, in fact, it converges in theoretically optimal time on serial architectures. Using a multigrid method would probably improve the performance even further, and would probably result in better accuracy without increasing workload.

Question 3 - Arbitrary boundaries. In order to support advanced interaction with the environment, support for arbitrary boundaries has been implemented. The implementation clearly shows that support for arbitrary boundaries can be implemented for real-time incompressible Navier-Stokes simulations on the GPU. The boundary mechanism pre-processes the boundaries in the simulation to allow boundary conditions to be applied in an efficient manner. Specifying boundary conditions is very easy and flexible in the sense that several types of boundary conditions are supported.

The simulator is able to handle real-time simulation of several thousands concurrent obstacles that follow the flow. The flowing obstacles interact with the stationary obstacles, avoiding obstacles passing through walls.

7.2 Future work

This section mentions some of the most important improvements that can be implemented to improve the performance and visual result of the simulator.

Support for boundaries by the SOR solver. To be of any actual usage, support for boundaries must be implemented for the SOR solver for computation of pressure. This mechanism should be implemented in the same manner as the embedded boundary handling mechanism implemented for the Jacobi solver. More specifically, boundary conditions should be performed “on the fly”, by the shader computing the values for the next SOR iteration to reduce overhead associated with render initialization.

More efficient diffusion simulation. Diffusion has not received much attention in this thesis in order to narrow the field of focus. However, since diffusion can be computed using the same techniques used to compute pressure, applying SOR to diffusion may lead to similiar performance and accuracy gains.

Multigrid. Multigrid methods should be very suitable for implementation on GPU as interpolation and extrapolation are a substantial part of the algorithm. Both of these can be implemented efficiently on the GPU as they closely resemble texture filtering. Bolz et al. [BFGS03] have successfully implemented a multigrid solver for sparse matrices and applied this to the discrete Poisson equation. Implementing an optimized multigrid solver for the discrete Poisson equation with real-time simulation in mind would be very interesting, and can potentially boost performance even further. Some optimizations will probably have to be implemented to achieve maximum performance, e.g., packing of textures.

Improved simulation of flowing obstacles. The mechanism for simulation of flowing obstacles should be simplified, to avoid excessive use of configurable parameters. Interaction between stationary obstacles and flowing obstacles are supported, but it is also desirable that flowing obstacles interact with each other. Another important improvement is moving implementation of rendering of the flowing obstacles from the CPU to the GPU, avoiding readback and expensive OpenGL calls. Texturing of obstacles and obstacles of various boundary types and shapes should also be implemented.

Support for 3D simulations. Support for 3D simulations is desirable for most applications, as most of the applications that is subject to the use a fluid simulator, e.g., games, are based on 3D graphics. Full fledged 3D simulations are still too expensive on grids of reasonable size. An alternative to full 3D simulation can be to use 3D extrusion as implemented by Krüger et al. [KW05].

Visualization techniques. This thesis has only focused on simulation of the incompressible Navier-Stokes equations. No techniques for simulations of real-life phenomena such as smoke and clouds have been presented. Such extensions are straightforward to implement and are necessary for usage in real-life applications.

Bibliography

[AGE] AGEIA. The First Dedicated Physics Processor For PC Games! http://www.ageia.com/pdf/ds_product_overview.pdf. [Online: accessed 11-May-2007].

[BFGS03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[BH] I. Buck and P. Hanrahan. Data parallel computation on graphics hardware. http://hci.stanford.edu/cstr/abstracts/2003-03.html [Online: accessed 18-November-2006].

[Bly06] D. Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[Box] The Engineering Tool Box. Material Properties. http://www.engineeringtoolbox.com/material-properties-t_24.html. [Online: accessed 13-June-2007].

[Bri] W. L. Briggs. A Multigrid Tutorial. http://www.llnl.gov/CASC/people/henson/mgtut/ps/mgtut.pdf. [Online: accessed 4-July-2007].

[Bro07] A. R. Brodtkorb. A MATLAB Interface to the GPU. Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, May 2007.

[Buc05a] I. Buck. High level languages for GPUs. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 109, New York, NY, USA, 2005. ACM Press.

[Buc05b] I. Buck. Taking the Plunge into GPU Computing. In M. Pharr, editor, *GPU Gems 2*, chapter 32, pages 509–519. Addison Wesley, March 2005.

[CM05] A. J. Chorin and J. E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Dar03] J. D. Darcy. What Everybody Using the Java™ Programming Language Should Know About Floating-Point Arithmetic. 2003.

[Deh96] R. C. Dehmel. Lecture notes from CS267: Solving the Discrete Poisson Equation using Jacobi, SOR, Conjugate Gradients, and the FFT. `http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html`, 1996. [Online: accessed 13-June-2007].

[Deh02] R. C. Dehmel. Lecture notes from CS267: Applications of Parallel Computers - Solving Linear Systems arising from PDEs. `http://www.cs.berkeley.edu/~demmel/CS267_2002_Poisson_1.ppt`, 2002. [Online: accessed 6-June-2007].

[EBL05] D. Etiemble, S. Bouaziz, and L. Lacassagne. Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing. *Proceedings of the 2005 3rd Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2005, September 22-23, 2005, New York Metropolitan Area, USA*, pages 61–66, 2005.

[ECST04] U. Erra, R. De Chiara, V. Scarano, and M. Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In *Vision, Modelling and Visualization 2004 (VMV '04)*, pages 21–51, November 2004.

[EMF02] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744, New York, NY, USA, 2002. ACM Press.

[FK03] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FM96] N. Foster and D. Metaxas. Realistic animation of liquids. *Graphical models and image processing: GMIP*, 58(5):471–483, 1996.

[FM05] J. Fung and S. Mann. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM Press.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press.

[GDN98] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[Gee05] D. Geer. Taking the graphics processor beyond graphics. *Computer*, 38(9):14–16, 2005.

[GGKM06] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM Press, 2006.

[GLGM06] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. Memory—a memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM Press.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[GPU] GPUBench. GPUBench Results. `http://graphics.stanford.edu/projects/gpubench/results/`. [Online: accessed 25-June-2007].

[GZ06] A. Greß and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[Har03] M. Harris. Fast Fluid Dynamics Simulation on the GPU. In R. Fernando, editor, *GPU Gems*, chapter 38, pages 637–665. Addison Wesley, 2003.

[Har06] M. Harris. Havok FX: Game Physics Simulation on GPUs. In *SUPERCOMPUTING 2006 Tutorial on GPGPU*, November 2006.

[Hav06] Havok. Havok FX™. Technical report, 2006. `http://www.havok.com/content/view/187/77/` [Online: accessed 25-June-2007].

[HB05] M. Harris and I. Buck. GPU Flow-Control Idioms. In Matt Pharr, editor, *GPU Gems 2*, chapter 34, pages 547–555. Addison Wesley, March 2005.

[HCH03] J. Hall, N. Carr, and J. Hart. Cache and bandwidth aware matrix multiplication on the GPU, 2003.

[Hil] M. Hilgart. Step-Through Debugging of GLSL Shaders. `http://facweb.cti.depaul.edu/research/TechReports/TR06-015.pdf` [Online: accessed 10-July-2007].

[HSU$^+$01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. Technical report, Desktop Platforms Group, Intel Corp., 2001.

[JS05] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, Washington, DC, USA, 2005. IEEE Computer Society.

[KF05] E. Kilgariff and R. Fernando. The GeForce 6 Series GPU Architecture. In Matt Pharr, editor, *GPU Gems 2*, chapter 30, pages 471–491. Addison Wesley, March 2005.

[KW03] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, July 2003.

[KW05] J. Krüger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[LBM$^+$05] A. E. Lefohn, I. Buck, P. S. McCormick, J. Owens, T. J. Purcell, and R. Strzodka. General Purpose Computation on Graphics Hardware. In *IEEE Visualization*, page 121. IEEE Computer Society, 2005.

[Lyc06] T. Lyche. Lecture notes from INF-MAT3350/4350 - 2006, University of Oslo. 2006.

[Mar04] K.-A. Mardal. Lecture notes from INF5670 - Numeriske metoder for Navier-Stokes likninger, University of Oslo. Lecture 1, 2004.

[MB05] T. McReynolds and D. Blythe. *Advanced Graphics Programming Using OpenGL (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Mic] Microsoft. Reference (DirectX HLSL. `http://msdn2.microsoft.com/en-us/library/bb509638.aspx`. [Online: accessed 6-July-2007].

[Mic04] Microsoft. PCI Express FAQ for Graphics. `http://www.microsoft.com/whdc/device/display/PCIe_graphics.mspx`, 2004. [Online: accessed 9-May-2007].

[Mic07] Microsoft Corporation. Microsoft DirectX. `http://www.microsoft.com/directx`, 2007. [Online: accessed 22-May-2007].

[NS06] J. Kiel (NVIDIA) and S. Dietrich (Compsite Studios). GPU Performance Tuning with NVIDIA Performance Tools. In *Presentations from 2006 Game Developers Conference*, 2006.

[NVIa] NVIDIA. CUDA Toolkit Version 0.8 Release Notes. `http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_SDK_releasenotes_readme_win32_linux.zip`. [Online; accessed 8-May-2007].

[NVIb] NVIDIA. GeForce 7800. [Online; accessed 8-November-2006].

[NVIc] NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. `http://www.nvidia.com/object/IO_37100.html`. [Online; accessed 8-May-2007].

[NVId] NVIDIA. The GeForce 7800 GTX GPU. Online: `http://images.tweaktown.com/imagebank/news_G70intro.pdf`. [Online: accessed 4-May-2007].

[NVI05] NVIDIA. NVIDIA GPU Programming Guide. `http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf`, August 2005. [Online; accessed 18-April-2007].

[NVI06] NVIDIA. The CUBLAS Library, version 0.8. http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUBLAS_Library_0.8.pdf, 2006. [Online; accessed 8-May-2007].

[NVI07] NVIDIA. NVIDIA CUDA - Compute Unified Device Architecture, 2007. [Online; accessed 8-May-2007, version 0.8.2].

[OLG+05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[Pop07] B. Pops. Google Beats Nvidia, Intel and Microsoft. And buys PeakStream. http://news.softpedia.com/newsPDF/Google-Beats-Nvidia-Intel-and-Microsoft-56563.pdf, 2007. [Online: accessed 11-May-2007].

[Rap] RapidMind. RapidMind Development Platform Product Overview. http://www.rapidmind.net/pdfs/RapidmindDatasheet.pdf. [Online: accessed 6-July-2007].

[Rho04] G. Rhodes. *Game Programming Gems 4*, chapter PHYSICS - Introduction, pages 207–208. Charles River Media, 2004.

[Ros06] R. J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[SB93] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in applied mathematics*. Second edition, 1993.

[SF93] J. Stam and E. Fiume. Turbulent wind fields for gaseous phenomena. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 369–376, New York, NY, USA, 1993. ACM Press.

[Son05a] Sony. SONY COMPUTER ENTERTAINMENT ENTERS INTO STRATEGIC LICENSING AGREEMENT WITH AGEIA. [Online: http://www.gamesindustry.biz/content_page.php?aid=10281 accessed 11-May-2007, 2005.

[Son05b] Sony. SONY COMPUTER ENTERTAINMENT ENTERS INTO STRATEGIC LICENSING AGREEMENT WITH HAVOK. [Online: `http://www.scei.co.jp/corporate/release/pdf/050721de.pdf` accessed 11-May-2007, 2005.

[Sta99] J. Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Sta03] J. Stam. Real-time fluid dynamics for games. `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf`, 2003. [Online: accessed 25-April-2007].

[Sto04] J. Stokes. *Understanding pipelining performance*, 2004. `http://arstechnica.com/articles/paedia/cpu/pipelining-1.ars/1` [Online: accessed 7-May-2007].

[SU94] J. Steinhoff and D. Underhill. Modification of the Euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluids*, 6(8):2738–2744, 1994.

[SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide, Fifth Edition, The Official Guide to Learning OpenGL, Version 2.* Addison-Wesley, 2005.

[Wik07a] Wikipedia. IEEE 754 — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 19-April-2007].

[Wik07b] Wikipedia. Reynolds number — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 11-April-2007].

[Wik07c] Wikipedia. Status register — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 16-March-2007].

[Wik07d] Wikipedia. Viscosity — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 30-January-2007].