GPU based simulation and visualization of fluids with free surfaces

GPU-basierte Simulation und Visualisierung von Strömungen mit freien Oberflächen

Author: Martin Schreiber
Advisor: Univ.-Prof. Dr. Hans-Joachim Bungartz
Supervisors: Dr. rer. nat. Stefan Zimmer
Dipl.-Tech. Math. Philipp Neumann
Submission Date: June 08, 2010
I assure the single handed composition of this diploma thesis only supported by declared resources.

June 08, 2010 - Martin Schreiber
# Contents

1 Introduction 11

2 CPUs vs. GPUs 13

3 OpenCL 15
   3.1 OpenCL and CUDA 15
   3.2 OpenCL architecture with NVIDIA GPUs 16
   3.3 Coalesced global memory access 19

4 Fluid Simulation with the Lattice Boltzmann Method 21
   4.1 LBM for computer scientists 22
      4.1.1 Equilibrium function 23
      4.1.2 Velocity and density 24
      4.1.3 Collision 24
      4.1.4 Propagation 25
      4.1.5 Boundaries and obstacles 25
      4.1.6 Gravity 26
      4.1.7 Convergence of collision operator 26
      4.1.8 Parametrization 27
   4.2 Derivation 28
      4.2.1 Collision step 28
      4.2.2 Maxwell-Boltzmann distribution 30
      4.2.3 Taylor expansion of equilibrium density distribution 33
      4.2.4 Multipliers for discretized equilibrium function 33
      4.2.5 Relaxation parameter and viscosity 34
   4.3 Implementation with OpenCL 35
      4.3.1 Data storage 35
      4.3.2 Memory layout 36
      4.3.3 Accessing density distributions 38
      4.3.4 Reading and writing density distributions 39
   4.4 Results for driven cavity 44
   4.5 Free surface LBM 45
      4.5.1 New cell types and values 46
      4.5.2 Mass tracking 46
      4.5.3 Interface change 47
      4.5.4 Reconstruction of density distributions from gas cells 48
   4.6 Free surface implementation with OpenCL 49
      4.6.1 Additional memory buffers and cell types 49
      4.6.2 General issues 51
      4.6.3 Kernels 51
   4.7 Lonely interface cells 53
   4.8 Results for free surface simulation on GPUs 54
      4.8.1 Testing different rules 54
      4.8.2 Benchmarking different patterns 54
5 Introduction to rendering and advanced methods
  5.1 Introduction to OpenGL
    5.1.1 OpenGL 3.2 core profile
  5.2 Render pipeline
    5.2.1 Vertex shader:
    5.2.2 Geometry shader:
    5.2.3 Fragment shader:
    5.2.4 Fragment postprocessing:
  5.3 Computations with framebuffers
  5.4 Cube Mapping
  5.5 Flat textures
  5.6 Depth peeling and front/back face extraction
  5.7 Marching Cubes
  5.8 Shadow mapping
  5.9 Photon mapping

6 LBM visualization using OpenGL
  6.1 Accessing an OpenCL array as an OpenGL volume texture
  6.2 Volume casting
    6.2.1 Volume casting for fluid fraction surface
    6.2.2 Basic implementation for ray casting
    6.2.3 Normal at intersection point
    6.2.4 Implemented algorithms
    6.2.5 “Simple” raycasting
    6.2.6 Interpolated raycasting
    6.2.7 Cube steps volume raycasting
    6.2.8 Marching cubes
    6.2.9 Results for volume casting
  6.3 Volume tracing with refractions
    6.3.1 Formulae for refractions
    6.3.2 Modifications to the volume casting shader
    6.3.3 Results for volume tracing with refractions
  6.4 Marching cubes triangle extraction
    6.4.1 Marching cubes with geometry shader
    6.4.2 Marching cubes with vertex array
    6.4.3 Marching cubes with vertex array and GL_INTEROPS
    6.4.4 Results for marching cubes
  6.5 Marching cubes with eye-space refractions
    6.5.1 Previous methods
    6.5.2 Surface approximation
    6.5.3 Intersection points in texture space
    6.5.4 Results for marching cubes with refractions in eye-space
  6.6 Photon mapping in light space
    6.6.1 Implementation
    6.6.2 Results
## Final results

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Interactive simulation and visualization with modification of the gravity vector</td>
<td>107</td>
</tr>
<tr>
<td>7.2 Oil like simulation and visualization</td>
<td>109</td>
</tr>
<tr>
<td>7.3 Website &amp; further information</td>
<td>109</td>
</tr>
<tr>
<td>Figure Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenCL Platform model</td>
</tr>
<tr>
<td>3.2</td>
<td>Global, constant and texture memory within OpenCL memory model</td>
</tr>
<tr>
<td>3.3</td>
<td>Local and private memory within OpenCL memory model</td>
</tr>
<tr>
<td>3.4</td>
<td>Overview of OpenCL work group and work items</td>
</tr>
<tr>
<td>4.1</td>
<td>LBM lattice vectors</td>
</tr>
<tr>
<td>4.2</td>
<td>Visualization of collision and propagation step</td>
</tr>
<tr>
<td>4.3</td>
<td>Bounce back and slip boundary conditions</td>
</tr>
<tr>
<td>4.4</td>
<td>Converging density distributions</td>
</tr>
<tr>
<td>4.5</td>
<td>Collision and propagation standard layout (A-B pattern)</td>
</tr>
<tr>
<td>4.6</td>
<td>Collision and propagation A-A pattern layout</td>
</tr>
<tr>
<td>4.7</td>
<td>Memory layout for density distributions</td>
</tr>
<tr>
<td>4.8</td>
<td>Threads accessing the density distributions without utilization of the local memory</td>
</tr>
<tr>
<td>4.9</td>
<td>Threads accessing the density distributions using local memory</td>
</tr>
<tr>
<td>4.10</td>
<td>Generated PTX code for if branching and wrapping method</td>
</tr>
<tr>
<td>4.11</td>
<td>Benchmark for basic implementation: MLUPS</td>
</tr>
<tr>
<td>4.12</td>
<td>Benchmark for basic implementation: SSSP</td>
</tr>
<tr>
<td>4.13</td>
<td>Free surface interface conversions</td>
</tr>
<tr>
<td>4.14</td>
<td>Overview of handling free surfaces on GPU</td>
</tr>
<tr>
<td>5.1</td>
<td>Simplified rendering pipeline</td>
</tr>
<tr>
<td>5.2</td>
<td>Computations using framebuffer and fragment shaders</td>
</tr>
<tr>
<td>5.3</td>
<td>Cubemap created for a standard scene</td>
</tr>
<tr>
<td>5.4</td>
<td>Flat textures</td>
</tr>
<tr>
<td>5.5</td>
<td>Depth peeling</td>
</tr>
<tr>
<td>5.6</td>
<td>Marching cubes cases</td>
</tr>
<tr>
<td>5.7</td>
<td>Shadow mapping</td>
</tr>
<tr>
<td>5.8</td>
<td>Photon mapping</td>
</tr>
<tr>
<td>6.1</td>
<td>Asymptots and isoline for 2D bilinear interpolation</td>
</tr>
<tr>
<td>6.2</td>
<td>Volume boxes rendering intersection points</td>
</tr>
<tr>
<td>6.3</td>
<td>Equidistant and cube steps sampling</td>
</tr>
<tr>
<td>6.4</td>
<td>Different visualization methods of a sphere created in an 8^4 volume texture</td>
</tr>
<tr>
<td>6.5</td>
<td>Volume casting with interpolation with a step size of 0.5 vs. ray casted marching cubes for a falling drop simulation with a domain size of 64^3</td>
</tr>
<tr>
<td>6.6</td>
<td>Interpolated volume casting, cube steps volume casting and marching cubes for breaking dam simulation</td>
</tr>
<tr>
<td>6.7</td>
<td>Different cases for equidistant sampling and cube steps sampling</td>
</tr>
<tr>
<td>6.8</td>
<td>Refractions with Fresnel factor</td>
</tr>
<tr>
<td>6.9</td>
<td>Screenshots for interpolated volume tracing with different limitations for ray tracing iterations</td>
</tr>
<tr>
<td>6.10</td>
<td>Screenshot for breaking dam simulation with tiled color background and up to 4 ray tracing iterations with Fresnel-term for first iteration</td>
</tr>
<tr>
<td>6.11</td>
<td>Ray tracing with a tiled color background. The left screenshot was created with the Fresnel term activated and the right one without the Fresnel term.</td>
</tr>
<tr>
<td>6.12</td>
<td>Histopyramid algorithm with a Z-Curve</td>
</tr>
<tr>
<td>6.13</td>
<td>Histopyramids: up and down pass</td>
</tr>
<tr>
<td>6.14</td>
<td>Histopyramids: reduction step</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.15</td>
<td>Converting the raw data (left texture) to the flat texture format (right texture) for a 64$^3$ falling drop simulation</td>
<td>92</td>
</tr>
<tr>
<td>6.16</td>
<td>FPS for different visualization methods</td>
<td>93</td>
</tr>
<tr>
<td>6.17</td>
<td>Computing whether the next sampling point is used to compute the final intersection point on a perspective rendered voxel texture</td>
<td>95</td>
</tr>
<tr>
<td>6.18</td>
<td>Changing the basis from view space to the intersection basis</td>
<td>98</td>
</tr>
<tr>
<td>6.19</td>
<td>Screenshots for eye-space refractions and interpolated volume tracing</td>
<td>99</td>
</tr>
<tr>
<td>6.20</td>
<td>Screenshots for falling drop simulation with a resolution of 64$^3$ with refractions based on Front/Back face extraction and volume tracing</td>
<td>100</td>
</tr>
<tr>
<td>6.21</td>
<td>Eye-space refractions with front/back face approximation using the Stanford bunny model</td>
<td>100</td>
</tr>
<tr>
<td>6.22</td>
<td>Expandable light space box for two different cases</td>
<td>102</td>
</tr>
<tr>
<td>6.23</td>
<td>Eye space refractive front/back and diffuse faces, caustic map</td>
<td>103</td>
</tr>
<tr>
<td>6.24</td>
<td>Screenshots with photon mapping</td>
<td>104</td>
</tr>
<tr>
<td>6.25</td>
<td>Screenshot with photon mapping and main caustics at an edge of the chessboard</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>Some screenshots for interactive simulation and visualization with modification of the gravity vector</td>
<td>108</td>
</tr>
<tr>
<td>7.2</td>
<td>Screenshots for a simulation with high viscosity and a domain resolution of 128$\times$64 $\times$32</td>
<td>110</td>
</tr>
</tbody>
</table>
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGK</td>
<td>Bhatnagar-Gross-Krook</td>
</tr>
<tr>
<td>BVH</td>
<td>Bounding volume hierarchy</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>FBO</td>
<td>Frame Buffer Object</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating Point Operations Per Second</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Computation on Graphics Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDR</td>
<td>High Dynamic Range</td>
</tr>
<tr>
<td>HP</td>
<td>Histo Pyramid</td>
</tr>
<tr>
<td>LB</td>
<td>Lattice Boltzmann</td>
</tr>
<tr>
<td>LBM</td>
<td>Lattice Boltzmann Method</td>
</tr>
<tr>
<td>MC</td>
<td>Marching Cubes</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>PTX</td>
<td>Parallel Thread Execution</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Threads</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extensions</td>
</tr>
<tr>
<td>SPH</td>
<td>Smoothed Particle Hydrodynamics</td>
</tr>
<tr>
<td>SSPS</td>
<td>Simulation Steps Per Second</td>
</tr>
<tr>
<td>MC</td>
<td>Marching Cubes</td>
</tr>
<tr>
<td>MLUPS</td>
<td>Mega Lattice Updates Per Second</td>
</tr>
<tr>
<td>NS</td>
<td>Navier Stokes</td>
</tr>
</tbody>
</table>
Symbols and functions

\( x \)
A position \((x_1, x_2)\) in 2D or \((x_1, x_2, x_3)\) in 3D or a multivariate parameter \(x_1, \ldots, x_n\) for functions

\( \xi \)
The velocity of one particle or an assembly of them

\( t \)
Time

\( c_g \)
Lattice speed

\( \Omega(f) \)
Collision operator

\( f(t) \)
Convenient abbreviation for \(f(x, \xi, t)\)

\( F \)
External body forces

\( d \)
Dimension (2D or 3D)

\( \Omega \)
Resolution of flow domain (dimensionless)

\( e_i \)
Lattice vectors (dimensionless)

\( u \)
Velocity vector (dimensionless)

\( \rho \)
Density of fluid

\( \tau \)
Relaxation parameter for collision operation (dimensionless)

\( \omega \)
Inverse of relaxation parameter

\( \nu \)
Kinematic fluid viscosity

\( G^d \)
Gaussian distribution of dimension \( d \)

\( L_x \)
Length of fluid domain along \( x \) axis

\( f_{eq} \)
Equilibrium density distribution function

\( f_{eq}^{ref} \)
Discretized equilibrium density distribution function

\( w_i \)
Weights for discretized equilibrium function \( f_{eq}^{ref} \)

\( s_i, \Delta x \)
Size of a cell size

\( \Delta t \)
Simulation time step

\( g \)
Gravitation vector (dimensionless)

\( g_{max} \)
Maximum dimensionless gravitational force

\( O \)
Cell type: obstacle

\( G \)
Cell type: gas

\( F \)
Cell type: fluid

\( I \)
Cell type: interface

\( T(x) \)
Type of cell \((O, G, F\) or \(I)\) at position position

\( T_0 \)
Shorthand for \(T(x)\)

\( T_a \)
Shorthand for \(T(x + e_a)\)

\( \vec{N} \)
Surface Normal

\( \vec{V} \)
Vector directed to eye position

\( \vec{V}' \)
\( \vec{V} \) reflected on \( \vec{N} \)

\( \vec{L}' \)
Vector directed to light source

\( \eta \)
Material dependent refractive index
1 Introduction

Recently the interest in interactive simulations has steadily increased. While physical rigid body simulations can already be driven with thousands of objects in real time, interactive fluid simulations are still computational much more intensive. Making use of the newest GPU hardware to gain the necessary speedup and new standard APIs from the cutting edge (OpenCL 1.0 and OpenGL 3.2 core mode) to offer compatibility with developments in the future could be a way out of this problem. The usage of the OpenCL and OpenGL specifications, which are standards supported by many hardware and software manufacturers also enhances the portability to other hardware and software architectures compared to proprietary APIs. Because the APIs used in this thesis are relatively new, the whole framework was developed from scratch.

Two tasks had to be solved; the implementation of a fluid simulation on the GPU using OpenCL as well as the visualization of the fluid surface on the same or an additional GPU with advanced rendering effects like refractions, reflections and photon mapping in real time, each with an appropriate approximation.

The first part of this thesis is about the Lattice Boltzmann Method (LBM) simulation itself. The free surface fluid simulation implemented in this thesis is based on the LBM. There already exist implementations for non-free surface simulations on GPU with different memory layouts. To keep the memory demand low, an implementation with the so called A-A pattern has been developed which is also used for the first benchmarks. Next, the implementation has been extended by a free surface model based on a simplified version of available implementations on the CPU to gain real-time results. Different memory layouts have been compared to show their advantages and disadvantages for the free surface GPU implementation developed in this thesis.

The second part of this thesis is about the visualization of the free surfaces of the fluid. Different approaches (direct volume renderers in different versions or polygonal surface extraction) already exist and have been investigated further and extended according to their performance and visual appearance to achieve the desired results.

With one of the results, we will see that the implemented framework is interesting for computer games with fixed fluid simulation domains, advanced visualization methods for scientific simulations and for accelerations of photo realistic renderings.

The source code developed during this thesis is available at [Scha] and is released under the Apache License [apa].
This chapter gives a short overview of the capabilities of CPUs and GPUs according to their usage and the requirements for scientific (high performance) computing.

Over the last few years, the performance of CPUs almost doubled every 2 years. For GPUs, this increase of performance was a factor of 4 for every 2 years [NVI10]. Especially simulations with high resolutions, computational intensive models or simulations depending on a large amount of time steps demand a high computational power. This is particularly the case for real time fluid simulations. To overcome the problems, efficient implementations depend on two crucial elements: an efficient implementation and a model which allows such an implementation. Because the implementation also partly depends on the model, this is one of the most complicated parts if models have to be implemented on GPUs of the current generation.

Using CPUs, well established standards like OpenMP [OMP] for multi-cores and MPI [MPI] for communication between the participating hosts can be used. Developing programs for CPUs is also better known and can be done more easily. A single core has the full capability which hardware oriented programmers are used to, like memory accesses with a relatively large cache, direct access to system calls, direct access to memory, standard debugging techniques and much more.

The reason why GPUs became so interesting for scientific computing was that around the year 2003 the computational power of GPUs exceeded that of CPUs and more so in the last 7 years. To give an example, fluid simulations can be run 25 times faster using only a single GPU than on multicore CPUs [BMW+09a]. It has to be noted, that such speedups depend on a good parallelizable problem while GPUs are totally useless if the problem can be solved only sequentially. Therefore the so often mentioned theoretical peak performance of GPUs used for theoretical comparisons can usually never be reached since most of the models are not designed to be solved on the GPU architecture.

For scientific computing and computer clusters, another important aspect has to be considered: the energy consumption. Most of the energy is directly emitted as heat, forcing computer centers to run expensive air conditioning systems to cool down the air in order to avoid an overheating of the computer components which also gives them a longer lifetime. Therefore we need a different way of measuring the efficiency of GPUs for scientific computing. One way is to compare the efficiency of computers (FLOPS) to its energy consumption (Watt). As previously mentioned, the efficiency also depends on the problem itself and it is impossible to consider all problems. However, one example should be given in form of a "finite element solver": Göddeke and Strzodka compared the efficiency with the power consumption to solve large system of equations [GS08]. One of their results was, that by extending a workstation with a NVIDIA GTX 280 and using the GPU to solve the equations, the performance per watt was increased by more than an order of 10. The same result was obtained by comparisons of Folding@home implementations [Tor].

1 e.g. a NVIDIA GTX 285
A good simulation, be it a religious myth or scientific theory, gives us a sense of mastery over experience.

Pagels, Heinz Rudolf (1939 - 1988)

This chapter is intended to give a brief introduction to OpenCL (Open Computing Language) as well as an introduction to the GPU architecture from the view of an OpenCL developer to allow a better understanding of the decisions which are made in the following chapters and to understand the implementation without going into great detail.

This chapter is not intended to give a complete and accurate description of the NVIDIA GPU architectures since this is out of the scope of this thesis and such information has already been published in different theses (e.g. [Str09], [Kan08]) and on websites (e.g. the NVIDIA developer website [NVD]). The interested reader can also find more information on the website of the Khronos group [khra], which maintains the specification in the OpenCL specification [Gro09b] and in the OpenCL best programming guide from NVIDIA [nVi09a].

Since the OpenCL code in this thesis was developed and optimized for NVIDIA GPUs, these optimizations are only NVIDIA specific unless otherwise stated.

3.1 OpenCL and CUDA

CUDA (Compute Unified Device Architecture) is a description for the unified computing layout in modern NVIDIA graphics cards. In the previous generations of graphics cards, the computing elements were specialized for standard operations to run either vertex or fragment shader programs (see Section 5.2 for vertex and fragment shaders). While with this new generation of graphics cards, such specialization was removed from the computing elements, unifying their abilities without restricting their computational efficiency. NVIDIA introduced the CUDA API in 2007, which allows running programs written in a language similar to C with a few restrictions as well as extensions.

OpenCL is a relatively new specification, released in 2009. It originated with the idea to establish a standard to run computations on heterogeneous processors (e.g. Intel x86 CPUs, Cell, GPUs) by using the same API and the same programming language for the programs running on the processing elements. However, the information in the OpenCL specification only acts as an abstract layer while optimization has to be done with the knowledge of the underlying hardware.

The development of the OpenCL specification was closely connected to the existing and well established CUDA [cud] API, offering a similar abstraction. Therefore the "CUDA programming guide book" and other sources of optimization hints have to be included in the improvements to gain the necessary speedup.

For this thesis, the OpenCL terminology is used in the majority of cases to describe the parts of the CUDA implementation as far as possible, otherwise the CUDA terminology is used if no synonym exists (e.g. the NVIDIA hardware specific warp and half warp).

One of the major reasons to use OpenCL was, that this specification enhances porting a developed code to hardware platforms of other vendors compared to porting programs using a
proprietary API and kernels written in a proprietary language.

It is important to note that optimizations for the GPU of one vendor will break this compatibility or at least the efficiency of the program to run on other GPUs as well as CPUs and other hardware. The kernels in this thesis have been successfully tested on multiple NVIDIA GPUs so far, but no test was performed on other GPUs and CPUs.

If not further specified, the benchmark values have been created with a NVIDIA GTX 285^1 graphics card.

### 3.2 OpenCL architecture with NVIDIA GPUs

The OpenCL architecture is built upon a hierarchy of 4 models: the *platform model*, the *memory model* describing the different available storages and their accessibility, the *execution model* to run the programs and the *programming model* with its available features for programmers. To prevent this section from becoming too abstract, each model is explained with the help of NVIDIA specific informations.

#### Platform model

An overview of the platform model hierarchy is given in Figure 3.1. A host (e. g. a computer) has so called *compute devices*. Such a compute device could be for example a graphics card, a CPU or other hardware which is programmable with OpenCL. On each compute device, there are one or more *compute units* which group together *processing elements*.

For GPUs, such processing elements are similar to single cores on a multi-core CPU but less powerful. To give an example, on NVIDIA graphics cards such processing elements do not have support for vector operations. Even if vector operations are included in the OpenCL syntax specification, the vector operations are split up into scalar operations on current generations of NVIDIA GPUs. AMD GPUs have built-in vector operations, but as mentioned in the introduction, the code was optimized for NVIDIA GPUs amongst others by using only scalar code for 3 component vector operations: So far there is no support for 3 component vectors in the OpenCL specification - only 2, 4, 8 and 16 component vectors. Thus 4 component vectors would have to be used instead, resulting in one obsolete computation on NVIDIA GPUs due to missing *Single instruction, multiple data (SIMD)* support. On GPUs, such processing elements run parallel, doing usually the same operation. Therefore the notation *Single instruction, multiple threads (SIMT)* was established by NVIDIA.

---

^1To be more specific: "Zotac NVIDIA GeForce GTX 285 amp! edition"
3.2. OPENCL ARCHITECTURE WITH NVIDIA GPUs

![Diagram of OpenCL memory model](image)

Figure 3.2: Global, constant and texture memory within OpenCL memory model

![Diagram of local work group](image)

Figure 3.3: Local and private memory within OpenCL memory model

Memory model

The OpenCL specification defines different memory buffers available to the programs running on the GPU as well as for CPUs to read/write from/to. An overview of the accessible memory is given in Figures 3.2 and 3.3.

The global memory is used to store and load data for computation kernels and to copy data from and to host memory. Storing and loading data from host memory (on CPU side) to/from global memory is slow compared to copying data on the GPU resulting in 2 GB/s vs. 134 GB/s. It is obvious, that large data transfers to/from GPU have to be avoided and should be done only during the initialization routines.

For NVIDIA cards, constants are also stored in the global memory and are accessible via a 64 KB constant memory cache.

The texture storage is also accessible via a cache, but we do not make further usage of that storage because writing to the texture storage in OpenCL is much slower than writing to the global memory [SPZ+09]. The texture cache becomes very important for the rendering process which is done using the OpenGL API.

---

2Using the OpenCL 'oclBandwidthTest' utility from the NVIDIA GPU Computing SDK, results on other systems are about 3 to 6 GB/s
CHAPTER 3. OPENCL

Figure 3.4: Overview of OpenCL work group and work items

The **private memory** is shared with the registers needed by the kernel and only accessible to each individual work item.

The **shared memory** is accessible by all work items within the same **local work group**. A **local work group** is a number of work items and is explained in more detail in the **execution model**. The advantage of the **shared memory** is, that the access is much faster than accessing the **global memory**. More details can be found in [nVi09a](#). The **shared memory** has a size of 16 KB for the NVIDIA 285 GTX and plays an important role for the data exchange and communication between the work items.

Put simply, a data exchange is achieved with the help of **synchronization barriers**, making the **local memory** persistent between all **work items** within a **local work group**: First a value is written by work item *Alice* to the **shared memory**. Then the synchronization command has to be executed by all the work items in the local work group. Otherwise a deadlock would exist and the execution is aborted by the GPU driver within a few milliseconds. After the synchronization is finished, work item *Bob* can be sure, that the value was written to the shared memory by work item *Alice*.

**Execution model**

The execution model describes how **kernels** are executed on **processing elements**. A **kernel** is a compiled program, ready to be executed on a **processing element**. A **work item** is the execution of a **kernel** on a **processing element**.

Multiple **work items** are combined to a **local work group** (see fig. 3.4) and a **global work group** is a collection of **local work groups**. The execution of kernels is always done for a single **global work group** by queueing the execution to a so called **command queue**. After the **global work group** is enqueued, the local work groups are executed according to the available hardware resources.

OpenCL also offers an **event dependent execution** which allows enqueued global work groups to wait for other events like the end of the execution of another enqueued global work group. This is useful when work groups should be executed sequentially to avoid race conditions. For this thesis, a special **barrier** command was enqueued to the command queue which is easier to implement. Such a **barrier** is enqueued to ensure that all previously enqueued tasks have ended, before executing tasks enqueued after the barrier.

---

3 access time is similar to that of accessing the cache
4 access time for global memory is more than 400 cycles
5 To be more general, all NVIDIA GPUs with compute capability 1.0 - 1.3
3.3 COALESCED GLOBAL MEMORY ACCESS

When a global work group is enqueued, the size of the global and local work group is specified individually for each execution in units of work items. The size of the local work group is the number of work items which run in parallel within a local work group. The local work groups are executed depending on the available hardware resources – either in parallel, sequentially or both. The amount of work items per local work group is also limited by further resource limitations like the available private memory. As a further restriction, the global work size has to be a multiple of the local work size.

To distinguish the work items within the kernel programs during their execution, a unique global id is assigned to each work item when a kernel is started. There are also multi variational ids available but those are not further explained, since they are not used for our purposes. The number of the local id as the id within a local work group is also available in kernel programs.

For NVIDIA GPUs, the local work groups are executed in parallel on multiple available multiprocessors on the GPU. To give a number, there are 30 of such multiprocessors for the NVIDIA 285 GTX. Warps within such a local work group are also automatically scheduled at zero cost to hide the latency of arithmetic operations, the memory latency to the global memory and other operations with stall cycles. Therefore it is important to run as many warps as possible on a multiprocessor. To increase this number of warps, the resource usage of warps has to be kept low. The initialization, execution and scheduling of the parallel execution is totally hidden by the GPU driver and hardware. More information about this crucial topic can be found in [nVi09a], Section 4.1.

3.3 Coalesced global memory access

On the NVIDIA GPU hardware level, work items within a local work group are further grouped into a so called half warp consisting of 16 work items with ids from $16 \cdot n$ to $16 \cdot (n + 1) - 1$. A memory access within these half warps has to be aligned in a specific way to maximize the bandwidth utilization. The coalescing of the memory access and thus the maximum usage of the bus to the global memory depends on the access pattern and the compute capability of the NVIDIA GPU. For CUDA capable GPUs, all float memory accesses within a half warp are coalesced to one request if those are sequentially accessed within a 64 byte memory block aligned at a 64 byte address.

For the GPU used in this diploma thesis, compute capability 1.3 was available which offers further features. One of the most important ones available since compute capability 1.2 is, that the access with an offset of 1 does not lead to 16 single read or write operations for each float as it would be the case for lower compute capabilities. Two different scenarios exist, depending on the access patterns of the work items within an 128 byte block, aligned at a 128 byte address. If all accesses are within such a block, then only a single memory transaction is needed. Otherwise only 2 memory accesses, since one or more floats are stored in a single 128 byte block while the other float values are stored within an adjacent 128 byte block. Such a feature will become very important for the so called propagation step (see Section 4.1.4) in the upcoming chapters where float variables have to be loaded with a displacement of $\pm 1$. Other features for compute capability 1.3 are mainly unused and therefore not mentioned in this thesis. More information is available in [nVi09a], Section 3.2.1.

But there are still problems if for example one half warp member is reading a value which does not lie within the memory block. This would double the memory latency for many cases. To circumvent this problem, there are two ways: Either we try to restrict the resource usage in order to hide the latency of memory access by scheduling of other local work groups or a

---

6floats with a size of 4 bytes
member of another half warp can read the value for the other work item and write the value to the local memory. The latter is also the only solution for kernels with a large demand for resources as is the case in this thesis. This workaround is further used and explained in Section 4.3.3.
Fluid Simulation with the Lattice Boltzmann Method

There are two basic approaches for fluid simulations:

- One of them is an *Eulerian* based simulation which uses a discretization with a typically spatial fixed grid.

  As we will see later on, the LBM (Lattice Boltzmann Method) uses an equidistant spaced regular grid and stores the current fluid state for each cell. Each timestep then only depends on the state of the current cell and partly of the adjacent cells.

  Another example for the Eulerian approach can be found in [GDN98], where the discretization of the Navier-Stokes (NS) equations leads to a system of linear equations and a non-linear time step. Regardless of numerical errors, solutions for such a system of linear equations can be e.g. found by using iterative methods.

- On the other hand, *Lagrangian* methods store the information about the fluid state at particle positions which follow the fluid motion. E.g. the simulation of intermolecular forces for molecules and gravity effects for planets can be simplified to particles. Then the particles are able to move around with respect to their current parameters such as mass and the force created by different potentials depending on the particle parameters (see [BZBP09], chapter 13).

  The Lagrangian approach is also used for Smoothed Particle Hydrodynamics (SPH) simulations [HKK07], assuming that an amount of fluid molecules can be represented by a single particle.

A short comparison of SPH and LBM should outline some advantages as well as disadvantages of both approaches when implementing the simulations on GPUs and their visualization pros and cons:

For SPH simulations, the number of work items is automatically adapted to the number of particles (and thus the fluid amount in the simulation) since the computation is done directly for the particles. This is a clear disadvantage of LBM, since the GPU implementation as it was implemented would also access gas cells (at least the flag field indicating the current state of the cell (fluid/gas/etc.)) – even if no computations need to be done on those cells. Another disadvantage is that the simulation is restricted to the fluid domain, since that kind of spatial adaptivity is difficult or impossible to implement efficiently on todays GPUs.

To simulate splashes, an extension to the LBM model needs to be used. Even if working implementations for splashes are available for CPUs [TRS06], it is hard to fit these algorithms to the GPU. For SPH solvers, those splashes are already included by the model itself.

---

1Splashes have not been further investigated in this thesis
CHAPTER 4. FLUID SIMULATION WITH THE LATTICE BOLTZMANN METHOD

The LBM with its free surfaces is by far more complicated to implement due to race conditions, special handling of gas-fluid boundaries and much more. However, there are also reasons to use the LBM for fluid simulations instead of SPH.

A clear disadvantage of SPH simulations is, that it is hard to extract smooth surfaces. Different methods have been investigated like smoothing the surface in viewspace with edge directed filters. Also noise was added to the fluid surface to produce a more detailed visual appearance, obviously without increasing the physical accuracy. Another approach is to use a grid and count the particles in a cell, to use marching cubes with the demand of many particles per grid cell. Using LBM simulations, the accurate amount of fluid within a cell is stored directly since the simulation depends on that value, allowing a volume tracing or a an extraction of triangle faces. A large number of particles is not only needed for a surface extraction, it is also necessary for accurate SPH simulations. LBM simulations overcome this problem by using statistical mechanics to simulate the behavior of a bulk of particles.

The free surface simulation with the LBM arose during the last 10 years. Even if this makes it a quite young method, it is already well established and tested for free surface flows.

Another reason is that most attention for interactivity was given to SPH solvers on GPUs for their use in computer games due to the restrictions of LBM mentioned above. Besides SPH, attention has also been given to a NS based free surface fluid simulation running on GPU and created by NVIDIA. This is most probably the reason, why no publications for free surface LBM simulation on GPUs exist.

The general ability to implement the LBM with free surfaces on GPUs with its massive parallel architecture was already tested in 2008 using the OpenGL shading language during a private project of this author [Schb].

Because the LBM is based on cellular automata, at least the basic LBM (without free surfaces) can be implemented quite easily and efficiently on today’s GPUs. The expectations are that this behavior will adapt to the free surface implementation.

For the same reasons and the frequent work which was done to develop SPH simulations on GPUs, this thesis was started to test and benchmark possible implementations of a Lattice Boltzmann (LB) free surface method on GPUs.

4.1 LBM for computer scientists

This section gives a short introduction as well as the basic formulae for the most traditional LB implementation with the BGK model. For convenience, the model without free surfaces is from now on denoted as the basic implementation. The extension to the free surface model is described later on in Section 4.5 after the basic implementation on GPUs was explained.

A short explanation of the LBM is given next: The LBM itself is based on the densities of particles within a small cell with a specific velocity vector and two operators executed for each timestep: The particle collisions are computed based on the density values of the cells with a so called collision operator. The movement of the particles within one timestep is then simulated by propagating the density values in the direction of the corresponding velocity vector to an adjacent cell.

First of all, we distinguish different discretization schemes which describe the dimension and the adjacent communication cells for data exchange along the lattice vectors. To abbreviate the description of used dimensions and lattice vectors, the $DdQq$ notation is used, where $d$ is the

---

2SPH simulations also simulate a bulk of particles by a single one, but not as flexible as the LBM simulation which stores the velocity, density and density distributions along the lattice vectors (the velocity and density are stored explicitly in the density distributions).
4.1. LBM FOR COMPUTER SCIENTISTS

number of dimensions and \( q \) is the number of density distributions for each cell. E.g. D3Q19 is a 3D model with 19 density distribution (Fig. 4.1). The lattice vectors are the basis of the discretized density distributions and for each lattice vector an according density distribution value is stored. Thus 19 values are stored in each cell to describe the fluid state of the cell for the basic implementation.

One cube has a side length of 1 within the simulation. The other parameters for the simulation are scaled by parametrization (Section 4.1.8).

The D3Q15 model was not used because it is more prone to numerical instability compared to the D3Q19 model and furthermore the D3Q19 model is more efficient than the D3Q27. Using the D3Q27 model would lead to additional time for doing the computations; loading additional density distribution values, however, increases the accuracy only slightly [MSY02].

4.1.1 Equilibrium function

The discretized equilibrium function represents the density of particles for a given velocity and density of the current particles in the cell in the direction of a lattice vector in case of reaching the equilibrium state. These equilibrium density distribution values are used to stay close to the thermodynamic equilibrium during molecular collisions. For the D3Q19 model, the discretized equilibrium distribution function is:

\[
f_{eq}^i(\vec{u}) = w_i \rho \left[ 1 + 3(\vec{e}_i \cdot \vec{u}) + \frac{9}{2}(\vec{e}_i \cdot \vec{u})^2 - \frac{3}{2} \vec{u}^2 \right]
\]  

(4.1)

with \( w_i = \frac{1}{18} \) for \( i \in [0; 3] \) or \([16; 17]\), \( \frac{1}{36} \) for \( i \in [4; 15] \) and \( \frac{1}{4} \) for \( i = 18 \).
4.1.2 Velocity and density

The computation of the velocity vector and density value (4.2) for the current cell is based only on existing density distribution values $f_i$ of the current cell. They are computed by:

$$\rho = \sum_{i=0}^{18} f_i \quad \rho \cdot v = \sum_{i=0}^{18} (f_i \cdot e_i)$$ (4.2)

Beneath their usage for the equilibrium function, the velocity and density field values are important to draw stream lines, to emit particles and for a general visualization of the fluid. Later on those values become of further interest for the free surface simulation, when those values are used to initialize the density distributions for gas cells which get partially filled with fluid.

4.1.3 Collision

The collision operator computes the new density distribution after each timestep. The density distribution values change due to collisions of their macroscopic simulated particles. Different types of collision operators exist. The Bhatnagar-Gross-Krook (BGK) model is widely used in many implementations because of its simplicity and reliability. Alternatives include the MRT $[dGK+02]$ which can be used to enhance the numerical stability of the fluid but the penalty is that this would be computationally more intensive. The standard relaxation (D3Q19 BGK) as well as a two-time relaxation method have been implemented in this thesis but the latter one was not further used in this thesis since the differences in the visual appearance to the single relaxation model were negligible$^3$. Simulations with a single relaxation time have already been used for simulations of free surfaces for realistic renderings (see $[BLE]$, $[Thu03]$). This collision operates only on the local density distribution values of a cell (Fig. 4.2) and is given by the equation:

$$f_i^c(x, t) = (1 - \omega)f_i(x, t) + \omega f_i^{eq}$$ (4.3)

where $\omega$ specifies the relaxation time. The closer the parameter is to 1, the faster the distributions tend towards the equilibrium distributions.

$^3$The two-time relaxation collision operator is still available in the source code and can be activated via the config file
4.1. Propagation

The propagation is responsible for the data exchange with the adjacent cells. It copies the density distributions to the adjacent cells (right hand side in Figure 4.2):

\[ f_i(x + e_i \Delta t, t + \Delta t) = f_i^c(x, t) \]  (4.4)

For the model presented in this section, the propagation is the only way to exchange information in form of density distributions with adjacent cells. This is the main reason for the simple spatial parallelization of the LBM.

The propagation step is usually combined with the collision step to avoid additional read and write operations. This optimization was also implemented and is explained in more detail in Section 4.3.2.

4.1.5 Boundaries and obstacles

Boundaries and obstacles within the fluid are specified by an array with a flag for whether the current cell is an obstacle or a fluid (and later on also a flag for the fluid surface and gas cells). The collision kernel which is executed for every cell then executes the code to handle the boundaries instead of the usual collision operator.

The modifications of the density distribution values for bounce back and slip boundary conditions are given in Figure 4.3. The left side shows the density distribution values drawn as vectors before the boundary operator is executed, the right side the distributions after the boundary operator was applied.

The **bounce back** condition is aimed to simulate very rough surfaces where the fluid particles close to the surface get stuck on it. To simulate a fluid at rest very close to a surface, the density distributions only have to be reversed. Obviously this results in a zero velocity on flat boundaries for a constant steady flow. The clear advantage of this method is its simple implementation – especially for complex geometries – without knowledge of the surface normal.

Slip boundary conditions are used to simulate slippery, hydrophobic surfaces like lotus leaves. For a slip boundary condition, the density distributions are mirrored along the surface normal vector. Slip boundaries have not been used in this thesis because more information like surface normals has to be read reducing the efficiency of GPU implementation, whereas the bounce
4.1.6 Gravity

The gravitational force for the lattice Boltzmann model was implemented as suggested in [KPR+04]. The collision operator is extended by a term modifying the density distribution value according to the gravitational force:

\[ f_i^c(x, t) = (1 - \omega) f_i(x, t) + \omega f_{i}^{eq} + w_i \rho e_i g \]  \hspace{1cm} (4.5)

4.1.7 Convergence of collision operator

The collision operator does not converge for all values of \( \omega \) to the equilibrium. Setting \( \omega \) to zero eliminates \( f^{eq} \) avoiding any convergence towards the equilibrium. To restrict values for \( \omega \), we simplify our model by analyzing the behavior of cells next to the moving surface of a driven cavity simulation. This forces the density distributions of those cells to tend to a fixed velocity. The assumption is, that this computed equilibrium distribution remains constant based on the velocity of the moving plate. Thus the values streamed by the propagation operator in the same direction are assumed to be equal:

\[
\begin{align*}
  f(x, t+1) &= (1 - \omega) f(x, t) + \omega f^{eq} \\
  f(x, t+2) &= (1 - \omega)^2 f(x, t) + (1 - \omega) \omega f^{eq} + \omega f^{eq} \\
  &\vdots \\
  f(x, t+n) &= (1 - \omega)^n f(x, t) + \sum_{i=0}^{n} (1 - \omega)^i \omega f^{eq} \\
  f(x, t+n) &= (1 - \omega)^n f(x, t) - \left( \frac{(1 - \omega)^n}{\omega} + \frac{1}{\omega} \right) \omega f^{eq} \\
  f(x, t+n) &= (1 - \omega)^n (f(x, t) - f^{eq}) + f^{eq} \tag{4.6}
\end{align*}
\]

Example iterations have been computed for constant start and equilibrium values and differing values of \( \omega \) (figure 4.4).
4.1. LBM FOR COMPUTER SCIENTISTS

Obviously, the density distribution converges to $f^{eq}$ for $\omega \in (0; 2)$. However, it is oscillating in the converging area for $\omega \in (1; 2)$ but $|f(x) - f^{eq}|$ is still reduced in every time step. Another limit for $\omega$ is given by the viscosity. The viscosity for the LBM simulation can be computed by the relaxation parameter $\tau$ (see Section 4.2.5):

$$\nu = \frac{2\tau - 1}{6}$$  \hfill (4.7)

With $\tau = \omega^{-1}$ and a positive viscosity, we get a further restriction which coincides with the previous result:

$$\nu = \frac{2\tau - 1}{6} > 0 \iff 2 \cdot \omega^{-1} - 1 > 0 \iff \omega < 2$$  \hfill (4.8)

Another restriction can be obtained by a linear stability analysis (see [WG00], section 5.6.2). Then the speed $\vec{u}$ has to be below the lattice speed $c_g$ which is $\frac{1}{\sqrt{3}}$ in our case.

4.1.8 Parametrization

Parametrization is used to scale the real world parameters to the simulation parameters like the unit cell length and to compute other parameters necessary to run the simulation. For the LBM, base units are the lattice size and the time step. For example if a simulation domain of $(1m)^3$ is used with a resolution of $64^3$, the other parameters like the viscosity have to be scaled to a unit cell. The domain resolution $\Omega_{x,y,z}$, the desired time step size $\Delta t'$, the length of the domain $L_x$ along the $x$ axis, the viscosity $\nu' \left[ \frac{m^2}{s} \right]$ of the fluid and the gravitational force $g' \left[ \frac{m}{s^2} \right]$ are used as parameters for the first parametrization step. The dimensionless viscosity is given by

$$\Delta x' = \frac{L_x}{\Omega_x}$$  \hfill (4.9)

$$\nu = \nu' \frac{\Delta t'}{\Delta x'^2}$$

With the dimensionless viscosity, we compute the dimensionless gravity $g$, the relaxation time $\tau$ and thus $\omega$:

$$\omega = \frac{2}{6 \cdot \nu + 1}$$  \hfill (4.10)

$$g = g' \frac{\Delta t'^2}{\Delta x'}$$  \hfill (4.11)

After this first step for the parametrization, two tests need to be done, to check whether a stable simulation is possible using the current values. The first test is a validation of the gravitational force: gravity is the only force acting externally onto the fluid particles and hence the density distributions. One restriction for a stable simulations is the velocity $\vec{u}$ which has to be below the lattice speed $c_g = \frac{1}{\sqrt{3}}$. If the gravitational force is too high, the velocity could exceed a level where no stable simulation would be possible. Alternative approaches track the maximum velocity of the current time step to change the maximum force adaptively resulting in adaptive time steps. Adaptive time steps as suggested in [Thu07] have not been used in this thesis. Such adaptive time steps are useless in our context because interactive fluids rendered in real time demand a constant number of time steps per frame. Instead, the maximum parametrized gravity was restricted to account for stability in the standard situations.
Therefore if the gravity exceeds a level \( |\mathbf{g}| > g_{\text{max}} \), the gravity is replaced by the maximum allowed value and the time step is adapted hence on initialization:

\[
g = g' \frac{\Delta t'^2}{s}
\]
\[
\Delta t' = \sqrt{\frac{g_{\text{max}} \cdot \Delta x'}{|g'|}}
\]

(4.12)

Again, the dimensionless viscosity \( \nu \) and \( \omega \) is computed using equations (4.9) and (4.11).

The second test accounts for \( \omega \) as mentioned in the previous section. Due to equation (4.11), \( \omega \) lies usually within the boundary \((0; 2)\). Test results showed, that even when \( \omega \) is within these boundaries, the simulation still becomes unstable if \( \omega \) is close to the boundaries. Therefore the last test checks whether \( \omega \) is close to such a boundary, and the simulation stops if this is the case.

An alternative approach which is not used in this thesis is to set the value of \( \omega \) to a fixed value, computing the time step \( \Delta t \) and finally computing the dimensionless gravity \( g \). This could result in similar problems because the dimensionless \( g \) can again exceed the maximum allowed value \( g_{\text{max}} \).

### 4.2 Derivation

Because a full derivation of the LBM would be out of the scope of this thesis, only selected topics are derived in this subsection.

The LBM used in this thesis is mainly based on the following ideas and operations:

- The simulation is computed using cellular automata with a collision kernel for computations independent of adjacent cell and a propagation step for the only way to communicate with adjacent cells. The propagation step depends only on the selected DdQq model and exchanges the density distribution values along the lattice vectors with the neighbored cells.
- An equilibrium distribution function is used to specify the average particle density distribution in an area for a given velocity in that area. The collision operator simulates the particle collisions streamed to a cell. For this thesis, a collision operator was used which linearly interpolates between the current density distributions and the values of the equilibrium distribution with a relaxation parameter.

A short overview of the derivations is given in the next subsections to give a deeper understanding of the topic for computer scientists without prior knowledge of theoretical physics.

#### 4.2.1 Collision step

The derivation in this subsection can also be found in [Tre02] and [Thu03]. We start with the Boltzmann equation, describing the change of density distribution for a time step due to particle collisions [Hua87]:

\[
\left( \frac{\partial}{\partial t} + (\xi \cdot \nabla_x) - \frac{F_e}{m} \cdot \frac{\partial}{\partial \xi} \right) f(x) = \left( \frac{\partial}{\partial t} \right)_{\text{coll}}
\]

(4.13)

The collision operator (right hand side of eq. (4.13)) can be described in different ways. Two assumptions are made: The first one is that external forces can be neglected since they have a small influence on the collision operator. The second assumption is that only binary collisions are considered with the conservation of mass, momentum and energy. Using these assumptions, a collision operator can be set up which can be used to determine crucial properties.
Bhatnagar, Gross and Krook (BGK) developed a model which replaces this collision operator \((\cdot)_{\text{coll}}\) by a relaxation term which only depends on the difference between the recent density distribution and the equilibrium distribution:

$$\left( \frac{\delta}{\delta t} + (\xi \cdot \nabla_x) - \frac{F_e}{m} \cdot \frac{\delta}{\delta \xi} \right) f(x) = \left( \frac{\delta}{\delta t} f \right)_{\text{coll}} = g(t) - f(t) \quad (4.14)$$

Here \(f\) specifies the density distribution function, \(F_e\) the external forces acting on the particle, \(m\) the mass of the particle, \(g(t)\) the equilibrium distribution which is given by the Maxwell-Boltzmann distribution. By going over from an Eulerian to a Lagrangian view with the time derivative of the density distribution along the direction of the velocity, the BGK equation can be written in a slightly different form in an ordinary differential equation. By ignoring the external forces (e.g. gravity), we get

$$\frac{\delta}{\delta t} f(t) + \left( \xi \cdot \nabla_x 1 + \frac{1}{\tau} \right) f(t) = \frac{1}{\tau} g(t)$$

An explicit solution for ordinary differential equations \(f(t)\) can be found in [BSMM01, Chapter 9.1.1.2], page 507.

$$f(t_1) = e^{-\int_{t_0}^{t_1} \frac{1}{\tau} dt} \left[ \int_{t_0}^{t_1} \frac{1}{\tau} g(t')e^{\int_{t'}^{t_0} \frac{1}{\tau} dt'} dt' + f(t_0) \right]$$

To get an explicit timestep \(\delta t\), we substitute \(t_1\) by \(t_0 + \delta t\) and for clarity \(t_0\) by \(t\):

$$f(t + \delta t) = e^{-\frac{\delta t}{\tau}} \left[ \int_{t}^{t+\delta t} \frac{1}{\tau} g(t')e^{\int_{t'}^{t} \frac{1}{\tau} dt'} dt' + f(t) \right]$$

$$= e^{-\frac{\delta t}{\tau}} \left[ \int_{0}^{\delta t} \frac{1}{\tau} g(t' + t)e^{\frac{t'}{\tau}} dt' + f(t) \right]$$

$$= e^{-\frac{\delta t}{\tau}} f(t) + e^{-\frac{\delta t}{\tau}} \frac{1}{\tau} \int_{0}^{\delta t} g(t' + t)e^{\frac{t'}{\tau}} dt' \quad (4.15)$$

Assuming that \(g(t)\) is linear or at least smooth enough within each time step, we can use a linear interpolation to get \(g(t + \chi)\) within the interval \([t; t + \delta t]\):

$$g(t + \chi) \approx \left( 1 - \frac{\chi}{\delta t} \right) g(t) + \left( \frac{\chi}{\delta t} \right) g(t + \delta t) \quad (4.16)$$

We can use (4.16) to solve the integral of (4.15) because \(g\) does not depend on \(t'\) anymore (only on the discrete points \(t\) and \(t + \delta t\):

$$\int_{0}^{\delta t} g(t + \chi)e^{\frac{\chi}{\tau}} d\chi \approx \int_{0}^{\delta t} \left( 1 - \frac{\chi}{\delta t} \right) g(t)e^{\frac{\chi}{\tau}} d\chi + \int_{0}^{\delta t} \frac{\chi}{\delta t} g(t + \delta t)e^{\frac{\chi}{\tau}} d\chi$$

$$= g(t) \int_{0}^{\delta t} e^{\frac{\chi}{\tau}} d\chi + \frac{g(t + \delta t) - g(t)}{\delta t} \int_{0}^{\delta t} \chi e^{\frac{\chi}{\tau}} d\chi$$

$$= g(t) \left[ \tau e^{\frac{\delta t}{\tau}} \right]_{0}^{\delta t} + \frac{g(t + \delta t) - g(t)}{\delta t} \left[ \tau e^{\frac{\delta t}{\tau}} - \tau \right]_{0}^{\delta t}$$

$$= g(t) \left( \tau e^{\frac{\delta t}{\tau}} - \tau \right) + \frac{g(t + \delta t) - g(t)}{\delta t} \left( \tau e^{\frac{\delta t}{\tau}} + \tau^2 \right)$$
We continue with (4.15) inserting the previously derived formula and subtracting \( f(t) \) on each side, we get

\[
f(t + \delta t) - f(t) = (e^{-\frac{\delta t}{\tau}} - 1)f(t) + e^{-\frac{\delta t}{\tau}} \frac{1}{\tau} \int_0^{\delta t} g(t + \chi)e^\frac{\chi}{\tau}d\chi
\]

\[
= (e^{-\frac{\delta t}{\tau}} - 1)f(t) + e^{-\frac{\delta t}{\tau}} \left( g(t) \left( e^{\frac{\delta t}{\tau}} - 1 \right) + \frac{g(t + \delta t) - g(t)}{\delta t} \left( (-\tau + \delta t)e^{\frac{\delta t}{\tau}} + \tau \right) \right)
\]

\[
= (e^{-\frac{\delta t}{\tau}} - 1)f(t) + g(t)e^{-\frac{\delta t}{\tau}} + (g(t + \delta t) - g(t)) \left( 1 + \frac{e^{-\frac{\delta t}{\tau}}\tau - \tau}{\delta t} \right)
\]

To avoid the computation of the exponential functions and to simplify the above equation, we use the Taylor series on the variable \( \delta t \) at point \( e^0 \):

\[
e^{-\frac{\delta t}{\tau}} = e^{-\frac{\delta t}{\tau}} - \frac{\delta t}{\tau} e^{-\frac{\delta t}{\tau}} + O(\delta t^2) = 1 - \frac{\delta t}{\tau} + O(\delta t^2)
\]

Because the notation with the Landau O symbol is only valid for asymptotic behavior, the interested reader may imagine the exponential function which changes very slowly only close to the origin. Attention has to be given to \( \tau \): if it is too small, the exponential function which depends on \( \tau \) increases faster, resulting in an increasing loss of accuracy. Thus the Taylor series is only valid for “small” \( \frac{\delta t}{\tau} \)

\[
f(t + \delta t) - f(t) \approx (f(t) - g(t)) \left( -\frac{\delta t}{\tau} \right) + (g(t + \delta t) - g(t)) \left( 1 + \frac{1 - \frac{\delta t}{\tau}}{\delta t} \tau - \tau \right)
\]

**Phase-Space collision equation:**

\[
f(t + \delta t, x, \xi) - f(t, x, \xi) \approx \frac{\delta t}{\tau} (f(t, x, \xi) - g(t, x, \xi)) \quad (4.17)
\]

### 4.2.2 Maxwell-Boltzmann distribution

The equilibrium density distribution function \( f^{eq} \) conforms to a discretized version of the Maxwell-Boltzmann distribution. We start with a derivation of the Maxwell-Boltzmann distribution, which itself can be written in many different ways. The one which is the most useful for this subsection describes the probability of a particle within the phase space. The spatial space is denoted by \( \Omega_d \) and the velocity space by \( V_d \). More information on the following derivation can be found in [RVG08].

We start with the formulae of well known physical values by integrating a function of the density...
distribution for the particle positions $x$ and velocities $v$:

\[
N = \int\int_{\Omega_d V_d} f(x, v) dv dx \quad (4.18)
\]

\[
P = \int\int_{\Omega_d V_d} mvf(x, v) dv dx \quad (4.19)
\]

\[
E = \int\int_{\Omega_d V_d} \frac{1}{2}mv^2 f(x, v) dv dx \quad (4.20)
\]

$N$: Number of particles in simulation domain $\Omega$

$P$: Total momentum

$E$: Total energy

Here, the function $f(x, v)$ specifies the density distribution of $N$ particles residing in the domain $\Omega_d$ at the position $x$ and velocity $\theta$. We are allowed to express $N$ in this way if we assume that the density of the particles is 1.

To derive the Maxwell-Boltzmann distribution, the famous Boltzmann H-Theorem

\[
H(P) = -\int P(z) \ln P(z) dz \quad (4.21)
\]

computing the entropy from the probabilities $P(z)$ is used. This function was introduced 1872 by Boltzmann describing the entropy of an ideal gas in the phase space. To clarify the reading of the following subsection, $f(t)$ was written for $f(x, \xi, t)$. Assuming, that $\int f = 1$ and $f \geq 0$ holds for every particle (similar to the laws for probabilities), the entropy of the density distribution can be written as

\[
H(f(x, v)) = -\int\int f(x, v) \ln f(x, v) dv dx \quad (4.22)
\]

We are interested in finding the function $f$ which maximizes the entropy of $H(f)$ according to $N$, $P$ and $E$ as side constraints. This maximization problem can be solved using the method of Lagrange multipliers. A short introduction [Kli04] is given here:

For a given function $f(x_1, ..., x_n)$ and $m$ side constraints $g_i(x_1, ..., x_n) = 0$ we search for the extremes where the normals of both functions are equal beside the scalars $\lambda_i$. The unnormalized normal vectors can be computed using the partial derivatives $\nabla$ on the functions $\nabla f(x) = \lambda_i \nabla g_i(x)$. Using the Lagrangian form $\Lambda(x, \lambda_i) = f(x) - \lambda_i g_i(x)$, we can use the partial derivative of this formula to express our problem of finding the extreme of $f$ with its side constraints $g$ in the formula $\nabla \Lambda(x, \lambda) = 0$, where $\nabla$ is the partial derivative in form of the gradient according to $x$ and $\lambda$.

For our problem, we would like to find the extreme of the function $f$ without knowledge of what this function looks like. Using a more intuitive description, the function itself can be seen as a variable selecting one of all available functions. An introduction to this topic about functionals can be found in [Sve05]. Now we can solve our problem by finding a function which maximizes the entropy with the given side restrictions by searching for our desired function in the function space using a modified version of the Lagrange multiplier method. Using $\nabla f = \left( \frac{\delta}{\delta f} \right)$
for the functional derivative and setting the derivative of the Lagrangian to zero, we get

\[ \nabla f \left( H(f) - (a, \vec{b}, c) \cdot \mathbf{g} \right) = 0 \]

\[ \Leftrightarrow \nabla f \iint_{\Omega_d, V_d} - \ln(f(x, v)) f(x, v) dv dx = \nabla f a \iint_{\Omega_d, V_d} f(x, v) dv dx - N \]

\[ + \nabla f \vec{b} \iint_{\Omega_d, V_d} m v f(x, v) dv dx - P \]

\[ + \nabla f c \iint_{\Omega_d, V_d} \frac{1}{2} m v^2 f(x, v) dv dx - E \] (4.23)

The derivative of the entropy function works similar to the usual derivations \((\nabla f(f \ln f) = (1 + \ln f) \nabla f f)\). We transform the equation 4.23 to

\[ \iint_{\Omega_d, V_d} - (\ln f(x, v) + 1) \nabla f f(x, v) dv dx = a \iint_{\Omega_d, V_d} 1 \cdot \nabla f f(x, v) dv dx \]

\[ + \vec{b} \iint_{\Omega_d, V_d} m v \nabla f f(x, v) dv dx \]

\[ + c \iint_{\Omega_d, V_d} \frac{1}{2} m v^2 \nabla f f(x, v) dv dx \] (4.24)

and by excluding the integral parts, we get

\[ - \ln f(x, v) - 1 = a + \vec{b} \cdot (mv) + c \frac{1}{2} m v^2 \]

\[ \Leftrightarrow f(x, v) = e^{-\left(1+a-\frac{m v^2}{2kT}\right) - \frac{1}{2} m c \left(\frac{v}{kT}\right)^2} \] (4.25)

For \(b = -\frac{u}{kT}\) and \(c = \frac{1}{kT}\), equation (4.25) becomes

\[ f(x, v) = e^{-\left(1+a-\frac{m v^2}{2kT}\right) - \frac{1}{2} m \frac{m}{kT} (v-u)^2} \] (4.26)

and by setting \(e^{-\left(1+a-\frac{m v^2}{2kT}\right)} = N \left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}}\), solving for \(a\) and replacing \(a, \vec{b}\) and \(c\) in (4.25), we get the more commonly known formula for the Maxwell-Boltzmann distribution:

**Maxwell-Boltzmann particle distribution:**

\[ f(x, v) = N \left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}} e^{-\frac{m |v-u|^2}{2kT}} \] (4.27)

The Maxwell-Boltzmann distribution in (4.27) describes the amount of particles which have a specific velocity \(v\) within an area with bulk velocity \(u\) at an infinitesimally small volume. Bulk velocity \(u\) is the macroscopic velocity of the particles in the volume and \(k_b\) the Boltzmann constant connecting the energy of a particle with the temperature. For a deeper understanding
4.2. DERIVATION

\[ u: \text{Bulk velocity} \quad k_b: \text{Boltzmann constant} \]

\[ v: \text{Velocity of particles} \quad T: \text{Temperature} \]

\[ N: \text{Number of particles} \quad d: \text{Dimensions (usually 2 or 3)} \]

\[ m: \text{Mass of a single particle} \]

of this formula, it can be written in a different way such that it is equal to the Gaussian distribution for one particle \((N = 1)\) with the standard deviation \(\sigma = \frac{k_b T}{m}\). E.g. for the one dimensional case \(d = 1\) the equilibrium distribution can be written as the Gaussian distribution \(G^d\):

\[ G^1(v) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{|v-u|^2}{2\sigma^2}} \quad (4.28) \]

Using this expression, it is obvious that the standard deviation is directly dependent on the temperature \(T\) whereas a higher mass lowers the standard deviation. For further usage, we use only one particle \((N = 1)\) and use the mass density \(\rho\) to scale the equilibrium distribution to gain the density distribution function. We are allowed to do this, since the Gaussian integral \(\int G^d(v)\) is equal to 1:

Maxwell-Boltzmann density distribution:

\[ g^d(v) = \rho \left( \frac{m}{2\pi k_b T} \right)^{d/2} e^{-\frac{m(v-u)^2}{2k_b T}} \quad (4.29) \]

4.2.3 Taylor expansion of equilibrium density distribution

For the following Taylor expansion, we assume that a particle has a specific speed \(\vartheta\) in a macroscopic amount of particles with the average velocity \(\xi\). We use the Taylor series up to 3rd order around 0 + \(\vartheta\) to get the equilibrium distribution \(f^{eq}\):

\[ g'(\vartheta) = \rho \left( \frac{m}{2\pi k_b T} \right)^{d/2} \left( \frac{m(\xi - \vartheta)}{k_b T} e^{-\frac{m(\xi - \vartheta)^2}{2k_b T}} \right) \]

\[ g''(\vartheta) = \rho \left( \frac{m}{2\pi k_b T} \right)^{d/2} \left( \frac{m^2 \xi^2}{k_b T^2} e^{-\frac{m(\xi - \vartheta)^2}{2k_b T}} - \frac{m}{k_b T} e^{-\frac{m(\xi - \vartheta)^2}{2k_b T}} \right) \]

\[ g(\vartheta) = g(0) + \vartheta g'(0) + \frac{\vartheta^2}{2} g''(0) + O(\vartheta^3) \]

\[ \approx \rho \left( \frac{m}{2\pi k_b T} \right)^{d/2} \left[ e^{-\frac{m\xi^2}{2k_b T}} + \vartheta e^{-\frac{m\xi^2}{2k_b T}} \left( \frac{m\xi}{k_b T} \right) + \frac{\vartheta^2}{2} e^{-\frac{m\xi^2}{2k_b T}} \left( \frac{m^2 \xi^2}{k_b T^2} - \frac{m}{k_b T} \right) \right] \]

Taylor expanded equilibrium density distribution:

\[ f^{eq}(\vartheta) = \rho \left( \frac{m}{2\pi k_b T} \right)^{d/2} e^{-\frac{m\xi^2}{2k_b T}} \left( 1 + \frac{m\xi \vartheta}{k_b T} + \frac{(m\xi \vartheta)^2}{2(k_b T)^2} - \frac{m \vartheta^2}{2k_b T} \right) \quad (4.30) \]

4.2.4 Multipliers for discretized equilibrium function

To use the equilibrium distribution function (Eq. (4.1)), the weights \(w_i\) have to be computed somehow. Computing the equilibrium distribution directly using the Maxwell-Boltzmann distribution is not possible due to the discretized velocity vectors. Using those directly in the
equilibrium distribution would only result in a solution for the infinitesimal small area at \( x \) for a discrete velocity vector \( u \).

One method to compute the weights \( w \) is comparing the moments of the lattice vectors with the moments of the equilibrium density distributions (see [WG00], chapter 5.3). We start with an additional nomenclature for the lattice velocity vector by adding an indexed greek symbol \( \alpha_j \). This symbols selects one of the three cartesian vector components.

The moments up to order \( n \) are written as:

\[
\sum_i w_i \prod_{j=1}^n e_i,\alpha_j = \int_{-\infty}^{\infty} g^e(u) \prod_{j=1}^n u_i,\alpha_j \, du
\]  

(4.31)

where \( g^e \) is the continuous Maxwell-Boltzmann density distribution function. The only useful moments for the derivation are those of zero, second and fourth order and those which are not equal to 0. They can be identified as the even moments (the zero, second and 2 times the fourth moment) with the following selected \( \alpha_j \) components and integrating the equilibrium distribution over \( u = [-\infty, \infty] \):

\[
\sum_i w_i = \int g^e(u) \, du 
\]  

(4.32)

\[
\sum_i w_i \cdot e_i,\alpha_j^2 = \int g^e(u) \cdot u_i^2,\alpha_j \, du 
\]  

(4.33)

\[
\sum_i w_i \cdot e_i,\alpha_j^4 = \int g^e(u) \cdot u_i^4,\alpha_j \, du 
\]  

(4.34)

\[
\sum_i w_i \cdot e_i,\alpha_j^2 \cdot e_i,\alpha_k^2 = \int g^e(u) \cdot u_i^2,\alpha_j u_i^2,\alpha_k \, du 
\]  

(4.35)

For symmetry reasons, we set \( W_0 = w_{18}, W_1 = w_i \) for \( i \in \{0, 1, 2, 3, 16, 17\} \) and \( W_2 = w_i \) for \( i \in \{4, 5, ..., 14, 15\} \). To solve the integrals on the right hand side of the equations, solutions for the integration of equations of type \( \int e^{x^2} \) are found in [BSMM01] (page 477) together with the extrema \( \lim_{x \to \infty} \text{erf}(x) = 1 \) for the error function. With further integration techniques, we get a system of four linear equations with \( W_i \) and \( \frac{k_b T}{m} \) as unknowns:

\[
W_0 + 6 \cdot W_1 + 12 \cdot W_2 = \rho 
\]  

(4.37)

\[
2 \cdot W_1 + 8 \cdot W_2 = \rho \left( \frac{k_b T}{m} \right) 
\]  

(4.38)

\[
2 \cdot W_1 + 8 \cdot W_2 = 3\rho \left( \frac{k_b T}{m} \right)^2 
\]  

(4.39)

\[
4W_2 = \rho \left( \frac{k_b T}{m} \right)^2 
\]  

(4.40)

Subtracting (4.39) from (4.38), we get \( \frac{k_b T}{m} = \frac{1}{3} \). The other solutions are determined bottom-up: \( W_0 = \frac{\rho}{3}, W_1 = \frac{\rho}{15} \) and \( W_2 = \frac{\rho}{36} \).

### 4.2.5 Relaxation parameter and viscosity

It is not only important for validation reasons to derive the Navier-Stokes equation from the Lattice Boltzmann methods, it is also necessary to find the relationship of the relaxation value
with physical parameters as the viscosity. One way to derive the Navier-Stokes equation is using the Chapman-Enskog expansion. Then the relaxation parameter can be directly computed from the kinematic shear viscosity parameter (see [WG00], page 182) using:

\[ \omega = \frac{2}{6 \cdot \nu + 1}. \]

4.3 Implementation with OpenCL

This section describes the implementation of the basic LBM with the OpenCL API. Generally, all the computations for the basic LBM can be rearranged to fit chiefly to the GPU architecture. Several different layouts are available for the implementation. Starting from the memory layout for the density distribution values, we first search for a suitable memory saving layout to load and store those. Such a layout also has the requirements for efficiency of loading and storing the density distributions. In this section, the more complex A-A pattern is used, while the A-B pattern can be implemented more easily. For the free surface implementation in the next section, both patterns have been implemented.

4.3.1 Data storage

We start with the necessary storage demand. The following datasets have to be accessed by the kernels to run the simulation:

<table>
<thead>
<tr>
<th>storage</th>
<th>size</th>
<th>access</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>density distributions</td>
<td>19 floats</td>
<td>RW</td>
<td>input/output channels for the propagation to adjacent cells, parameters and results for collision operator</td>
</tr>
<tr>
<td>cell flags</td>
<td>1 int</td>
<td>R</td>
<td>distinguish between fluid and obstacle cells</td>
</tr>
</tbody>
</table>

Obviously, only one bit would be sufficient to distinguish the cell types which have been stored using a 32 bit integer value for every cell in this implementation. There are other ways to load the flags for the basic implementation:

The first one is to bit-combine the values of up to 32 cells into a single integer word. This has the drawback of the utilization of a synchronization barrier for a local work group.

The second approach is to abuse the sign of the density distribution value for the lattice vector \( \mathbf{e}_{18} \). This value is positive for a velocity below \( \sqrt{\frac{2}{3}} \) which is usually the case since the velocity has to be below \( \sqrt{\frac{1}{3}} \) to run a stable simulation.\(^4\) The drawback is the utilization of a few more operations for the flag extraction, to compute the absolute value of the density distributions and to combine the flag value with the density distribution value for storage. On the other side, the load operation for the flag is skipped.

However, both improvements have been avoided since this basic LBM implementation should be extended to act on different cell types (free surfaces) and thus more than 1 bit is necessary to distinguish the different cell types. Secondly, no implementation should be developed which can be used only for performance considerations without any further usage.

\(^4\) For \( \mathbf{e}_{18} = (0, 0, 0)^T \), we search for positive value of \( f_{18}^{eq}(\mathbf{u}) \) depending on \( \mathbf{u} \):

\[
0 < f_{18}^{eq}(\mathbf{u}) = w_\rho \left[ 1 + 3(\hat{e}_i \cdot \mathbf{u}) + \frac{9}{2}(\hat{e}_i \cdot \mathbf{u})^2 - \frac{3}{2} \mathbf{u}^2 \right] \iff 0 < 1 - \frac{3}{2} \mathbf{u}^2 \iff |\mathbf{u}| < \sqrt{\frac{2}{3}}
\]
For visualization purposes, the velocity and density values also have to be stored when the simulation step is the one before the next frame is rendered:

<table>
<thead>
<tr>
<th>storage</th>
<th>size</th>
<th>access</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>velocity</td>
<td>3 floats</td>
<td>W</td>
<td>written for visualization purposes</td>
</tr>
<tr>
<td>density</td>
<td>1 float</td>
<td>W</td>
<td>written for visualization purposes</td>
</tr>
</tbody>
</table>

For the benchmarks, storing the velocity and density values was deactivated.

### 4.3.2 Memory layout

During the last years, different memory layouts to store the density distributions have been developed (see [PKW+03], [BMW+09b], [LFWK03]). For this thesis, it was also important to test the performance behavior of different memory layouts, since a layout which avoids doubling the memory consumption due to an additional buffer while maintaining almost the same performance is desired. This is because the global memory on the standard workstation GPU is still far below the accessible memory for CPUs. Another reason to save memory is the usage of the simulation in computer games or when sharing the GPU with other simulations or visualization tasks which would have to share the available memory between the simulation and gaming data.

For the basic LBM, the A-A pattern memory layout is used, which uses the lowest possible amount of memory. Because one of the main tasks of this thesis is not the development of the basic implementation, further memory layouts are only compared for the free surface implementation (Section 4.8).

Next, we take a look on three different ways to implement the basic kernels.

**A-B pattern**

The data layout and propagation step for the straightforward implementation of the kernels is shown in Fig. 4.5. Writing to cells from where data has to be read in the same time step needs to make use of an additional buffer for a parallel implementation to avoid read/write conflicts. For an efficient computation on GPUs when work items are queued to handle the whole domain at once, this would force the use of two density distribution arrays which are switched in every time step.

**Grid compression**

A different memory layout is the grid compression [PKW+03]. To make use of this, the kernels have to be executed in a specific order which is not ensured by the OpenCL specification due to work groups being executed in parallel. Therefore it is not possible to use grid compression on GPUs.

**A-A pattern**

The A-A pattern uses different kernels for odd and even time steps to enhance the memory demand by using only one density distribution data storage. The A-A pattern was already implemented by [BMW+09b] using the CUDA API. Their results satisfy the requirements of avoiding excessive memory use, and the performance is similar to the implementation with the A-B pattern. Therefore this pattern was used for the basic implementation. Next, we go into further details of the A-A pattern:
4.3. IMPLEMENTATION WITH OPENCL

**Density distributions (model):**

```
\[ \text{Density distributions (model):} \]
```

**Data storage (implementation):**

```
\[ \text{Data storage (implementation):} \]
```

Figure 4.5: Standard layout (A-B pattern) for collision and propagation operator:
The direction of the lattice vector represents index \( i \) of the density distribution value \( f_i \).
For the standard layout, the density distribution values are stored in the same memory location
after applying the collision operator (the movement of the red dashed arrow within a cell from
drawing on the left hand side in the upper row is only used to visualize the collision operation).
The propagation operator then reads the density distribution value from the adjacent cell in
the opposite direction of the lattice vector belonging to the density distribution (middle image)
and stores it in the according density distribution memory location of the current cell (right
image).
Both steps can be combined when using an additional buffer.

**Density distributions (model):**

```
\[ \text{Density distributions (model):} \]
```

**Data storage (implementation):**

```
\[ \text{Data storage (implementation):} \]
```

Figure 4.6: Collision and Propagation Layouts: Layout for A-A pattern implementation
To distinguish the two different kernels, they are denoted as alpha and beta kernel according to the memory layout used (Fig. 4.6). Using the A-A pattern layout, the propagation step can be implemented efficiently on GPUs without the use of memory buffers by avoiding an explicit propagation step and thus additional read/write collisions by combining it with the collision operator. The distribution and data storage scheme for the alpha kernel is given on the left hand side of Fig. 4.6 and those of the beta kernel on the right hand side.

For the alpha kernel, the distributions are read in the same way as the A-B pattern after the propagation step. After computing the collision or the bounce back operation, the density distribution values are stored in the opposite lattice vectors of the current cell. Thus the alpha kernel only accesses the values for the density distribution values stored in the current cell for the collision and propagation. Together with the beta kernel, this access rule already implements the propagation operator!

The implementation of the beta kernel does not access the current cells’ density distributions at all. Instead, the beta kernel is reading the density distributions from the adjacent cells, which lattice vector is directed from the current to the adjacent cell. After computing the change of the density distributions due to collisions or boundaries, the results are written to the adjacent cell in the direction of the lattice vector, which belongs to the computed density distribution.

Looking at Fig. 4.6 it should be clear that the memory access is exclusive for every kernel when executed in parallel.

To drive a realtime simulation, we need to have a closer look at another side effect. Using different memory layouts with a different amount of coalesced memory readings and a different resource usage would most probably result in different computation times for the kernels. This is no problem for our real time free surface simulation since the difference in the computation time is hidden by simulating more than one time step per frame. Executing more than one time step for a usual framerate of 30 is also indispensable because of the restriction to a small time step for the test cases created here. Another more intuitive reason is that for the desired real time simulation, with a domain size of at least $64^3$, 32 FPS, and an amount of fluid which should travel from one domain side to the other within one second, needs at least a few timesteps in each frame due to the limited communication speed to one cell per time step caused by the underlying model of cellular automata.

4.3.3 Accessing density distributions

Each work item is started to handle one domain cell. For efficiency reasons, the density distribution values have to be stored in memory to allow optimized, coalesced read and write operations for the work items running within the local work group.

To achieve this, density distributions for a specific lattice vector are simply stored linearly in the global memory of the GPU. Those blocks of lattice vector specific distributions are again stored to be accessed coalesced in a OpenCL memory buffer (Fig. 4.7). The enumeration of the density distribution (Fig. 4.1) was optimized for an implementation using OpenGL (see [LFWK03]) which also optimizes the OpenCL memory access. After 4 density distribution values are loaded (One table row in Figure 4.1), the computations depending on the already loaded data are immediately done before triggering the loading of other density distributions. This is expected to hide more memory latency due to scheduling of warps of other local work groups compared to a solution which first loads all density distribution values, doing the computations after that.

It is also possible to use this technique to immediately handle the bounce back boundary conditions after loading pairs of density distributions since the computations which have to be

---

\footnote{usually more than 10 due to the restrictions on the lattice velocity}
4.3. IMPLEMENTATION WITH OPENCL

Figure 4.7: Memory layout for density distributions for a $4^3$ simulation domain: The values for a lattice vector are stored adjacent to maximize utilization of the memory bus.

done for this cell type do not depend on the computed density. This depends on all computed density distribution values, whereas the collision operator depends on the density. Because this would reduce the ability to implement more complex boundary conditions depending e.g. on the velocity, it was also not implemented.

The *density* values are stored consecutively in a separate buffer, the *cell type flags* in the same way while the *velocity* vectors are stored similarly to the density distribution values. First, all $x$ components are stored, then the $y$ and $z$ components to achieve a coalesced memory access.

4.3.4 Reading and writing density distributions

**Alpha kernel**

The alpha kernel accesses density distributions only for the current cell for which the work item is executed. If $i$ is the index of the lattice vector (e.g. with $i = 0$, the lattice vector $e_0$ is $(1, 0, 0)^T$) and $\text{DOMAIN\_SIZE}$ is the number of cells in the domain, a work item with the global id $\text{gid}$ accesses the density distribution array indexed with $\text{gid} + i \cdot \text{DOMAIN\_SIZE}$ with other threads coalesced, if the array is properly aligned (which is usually the case) and if the domain size is a multiple of 16 (see Section 3.3 for a reminder of coalesced access).

**Beta kernel**

For a 0 displacement within $x$, an aligned access can be guaranteed if the number of volume cells along the $x$-axis is initialized with a multiple of 16 (the number of work items within a half warp). The beta kernel accesses only the density distributions for lattice vectors with a zero $x$-displacement directly and thus coalesced (dd2, dd3, dd16, dd17, dd12, dd13, dd14, dd15, dd18). For an efficient implementation, the other density distributions values with a ±1 displacement in $x$ have to be read from and written by the ‘adjacent’ threads. An example for a density distribution with a displacement along the $x$ axis is given in figure 4.8.

The displacements in the $x$-direction have to be handled in a special way to gain coalesced access. This is explained in the next section.

---

6 Domain sizes below 16 are not treated
Wrapping method

An overview of a memory access increasing the coalescence using the shared memory is given in figure 4.9. There are two ways to achieve a reduction of uncoalesced memory accesses. Making use of branching instructions or modulo computations. The latter one is further denoted as wrapping method.

To show the differences between both implementations, we first take a look at a sample code for a better understanding:

<table>
<thead>
<tr>
<th>If-Branching</th>
<th>Wrapping method</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define LOCAL_SIZE 128</td>
<td>#define LOCAL_SIZE 128</td>
</tr>
<tr>
<td>__kernel void main(__global int* val)</td>
<td>__kernel void main(__global int* val)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>const size_t lid = get_local_id(0);</td>
<td>const size_t lid = get_local_id(0);</td>
</tr>
<tr>
<td>size_t write_index = lid+1;</td>
<td>size_t write_index = lid+1;</td>
</tr>
<tr>
<td>if (lid == LOCAL_SIZE-1)</td>
<td>write_index &amp;= ~(LOCAL_SIZE-1);</td>
</tr>
<tr>
<td>write_index = 0;</td>
<td>write_index &amp;= ~(LOCAL_SIZE-1);</td>
</tr>
<tr>
<td>val[write_index] = lid;</td>
<td>val[write_index] = lid;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

The task is to redirect the work item with the maximum possible local id within a local work.
If-Branching

```c
.entry main
( // parameter "val" for main()
.param .b32 main_param_0
){

  // REGISTERS ALLOCATION
  // allocate 2 x 1 bit predicate regs
  .reg .pred %p<2>;  
  // allocate 7 x 32 bit regs
  .reg .s32 %r<7>;

  _main:
  {
    // get_local_id(0)
    cvt.u32.u16 %r1, %tid.x;
    }

    // add "1" to lid and store to r2
    add.s32 %r2, %r1, 1;

    // set predicate register, if lid==127
    setp.eq.s32 %p1, %r1, 127;

    // load r3 with 0, if lid==127
    selp.b32 %r3, 0, %r2, %p1;

    // left shift r3, store to r4
    shl.b32 %r4, %r3, 2;

    // array access...
    ld.param.u32 %r5, [main_param_0];
    add.s32 %r6, %r5, %r4;
    st.global.u32 [%r6], %r1;
    ret;
  }
}
```

Wrapping method

```c
.entry main
( // parameter "val" for main()
.param .b32 main_param_0
){

  // REGISTERS ALLOCATION
  // allocate 7 x 32 bit regs
  .reg .s32 %r<7>;

  _main:
  {
    // get_local_id(0)
    cvt.u32.u16 %r1, %tid.x;
    }

    // add "1" to lid and store to r2
    add.s32 %r2, %r1, 1;

    // r2 & 0x3FFFFF80, store to r3
    and.b32 %r3, %r2, 1073741696;

    // left shift r3, store to r4
    shl.b32 %r4, %r3, 2;

    // array access...
    ld.param.u32 %r5, [main_param_0];
    add.s32 %r6, %r5, %r4;
    st.global.u32 [%r6], %r1;
    ret;
  }
}
```

Figure 4.10: Generated PTX code for if branching and wrapping method

The write operation in the example code is implemented only to produce the desired PTX code, which is the intermediate compiled code of a program running on a processing element. Without the write operation, the according PTX code in which we are interested in would not be generated by the OpenCL compiler due to a missing RAW dependency.

Using OpenCL, the PTX code can be read back for NVIDIA hardware by reading out the binary code supported by the OpenCL API after building the program. This PTX code (created with NVIDIA driver version 193.36.15) is given in 4.10. For convenience, the header of PTX code is left out and the lines have been commented for easier understanding for readers who are not familiar with PTX code or any kind of assembler languages.

We can see that the wrapping method avoids allocating and setting the single bit predicate register, which is used to ignore the result of operations for specific warps according to the predicate register. E.g. "selp.b32 %r3, 0, %r2, %p1" in the PTX code for the if-branching method sets the value of %r3 to 0 if the predicate register %p1 is set, otherwise it is set to %r2.

---

Figure 4.10: Generated PTX code for if branching and wrapping method

*group* to write to the first array element indexed with 0, whereas all other work items write to the array element indexed with "local_thread_id + 1". The write operation in the example code is implemented only to produce the desired PTX code, which is the intermediate compiled code of a program running on a processing element. Without the write operation, the according PTX code in which we are interested in would not be generated by the OpenCL compiler due to a missing RAW dependency.

Using OpenCL, the PTX code can be read back for NVIDIA hardware by reading out the binary code supported by the OpenCL API after building the program. This PTX code (created with NVIDIA driver version 193.36.15) is given in 4.10. For convenience, the header of PTX code is left out and the lines have been commented for easier understanding for readers who are not familiar with PTX code or any kind of assembler languages.

We can see that the wrapping method avoids allocating and setting the single bit predicate register, which is used to ignore the result of operations for specific warps according to the predicate register. E.g. "selp.b32 %r3, 0, %r2, %p1" in the PTX code for the if-branching method sets the value of %r3 to 0 if the predicate register %p1 is set, otherwise it is set to %r2.

---

7 Read After Write
It is very important to know, that PTX code is only a hint, since the PTX code is used to create a hardware specific code. It is also possible to optimize the code for each different type of NVIDIA GPU using decuda [DEC], disassembling specialized code for each individual GPU. Because of this specialized code, such optimizations have been neglected in this thesis since they can nullify the performance increase for other GPUs.

To give the interested reader further information about the capabilities of current NVIDIA GPUs, the predicate register can also be combined with most of the PTX operations to indicate whether the result of the operation should be written to the destination register or not. E.g. such opcodes are created for the free surface implementation when reconstructing the incoming density distributions. Since the predicate register is set individually for every work item, this is helpful for if-branchings with only a few operations. This is because it avoids jump instructions, which would take one or more cycles instead of executing the code sequentially and ignoring the results according to the predicate register.

Now we continue to use the wrapping method to properly create coalesced memory access. Making use of the precompiler to setup the bit operations for the wrapping method, this technique is able to compute the array index for every density distribution with fewer arithmetic operations (one displacement and one modulo operation) compared to an if-branching. Bit operations can be used instead of modulo operations if the size in x-direction is a power of two.

For more abstraction and better readability, the wrapping method is hidden by the *_WRAP-functions as expected functions in listing 4.1. This also implements either bit operations or the fallback solution using the divide operator and modulo computations. On the GPU which was used in this thesis[^], a difference in runtime comparing the usage of the bit operations with the forced usage of the modulo operator was not measurable during the benchmarks for the driven cavity simulation.

[^]: NVIDIA GTX 285

---

**Listing 4.1: Pseudocode for wrapping method**

```c
1 INPUT:
  2 int gid; // global id
  3 int lid; // local id within workgroup
  // preprocessor directives:
  4 int DOMAIN_CELLS; // domain size
  5 int LWG_SIZE; // size of local work group
  6 int DELTA_NEG_X; // DOMAIN_CELLS - 1
  7 int DELTA_POS_X; // 1

FUNCTIONS:
  8 LWG_WRAP(int A); // A modulo local_work_group_size
  9 DOMAIN_WRAP(int A); // A modulo domain_cells

ALGORITHM PREPROCESSING STEPS:
 10 int local_pos_x_wrap = LWG_WRAP(lid + 1);
 11 int local_neg_x_wrap = LWG_WRAP(lid + (LWG_SIZE - 1));
 12 int global_delta_neg_x =
     DOMAIN_WRAP(gid - lid + pos_x_wrap + DELTA_NEG_X);
 13 int global_delta_pos_x =
     DOMAIN_WRAP(gid - lid + neg_x_wrap + DELTA_POS_X);
```

---

42
4.3. IMPLEMENTATION WITH OPENCL

For reading convenience, the usage of bit operations or modulo operations to apply the wrapping method is further denoted as wrapping.

The array indices `local_pos_x_wrap` and `local_neg_x_wrap` for reading and writing the arrays to the local memory for loading and storing data for other work items with a ±1 displacement have to be wrapped with the local work group size after applying the displacement. Since taking modulo a negative number or using bit operations does not return the desired result, `+(LWG_SIZE-1)` is used instead of −1 which is equivalent in quotient rings for ±1 memory access.

Similar to the computation of the array indices for read and write operations to the local memory, the displacements `global_delta_neg_x` and `global_delta_pos_x` for the ±1 x-displacements are computed to access the density distribution values which have to be read or written for an adjacent thread (4.1). `gid` is already included in this computation to avoid a further add operation since we always start computing the array element after all displacements at the array element indexed with `gid`. Here, the parameter for `DOMAIN_WRAP` can never be negative since `gid` is always equal or greater than `lid`.

With this small preprocessing from the listing, we are able to access the individual density distributions faster. The variables `global_delta_neg_x` and `global_delta_pos_x` can be directly used as displacements representing the final array index for a single displacement in x to read the density distributions for other work items.

Displacements in the y and z-direction are handled by using a similar displacement as for the x-direction: E.g. for a displacement of −1 in y-direction, the preprocessor constant `DOMAIN_CELLS-DOMAIN_CELLS_X` is added to the array index.

After this, the array index has to be wrapped with `DOMAIN_SIZE` to avoid an out of array accesses, e.g. accessing a density distribution value at the back layer in direction of z for a positive z-displacement. For this example, the wrapping is reading and writing obsolete and invalid data from the other domain border slice, which was not prohibited in this implementation. Another solution would be to use additional halo layers along each direction on the z-axis which would slightly increase the memory demand to store the additional halo density distribution values. It is also possible to avoid loading this data by if-branching instructions to avoid the memory access at all.

When the density distribution value for another work item is read from the global memory it is written to a buffer array in the local memory, which is indexed with the work item local id. After synchronization of all work items, `neg_x_wrap` and `pos_x_wrap` is used as an index into the buffer in the local memory to read the density distribution value written by another work item.

A further optimization was done when the number of domain cells in x-direction is a multiple of the local work group size. In this case, the preprocessor is used to set `global_delta_neg_x` and `global_delta_pos_x` directly to the `global id`. Obsolete, invalid data from the other domain sizes would be read for the domain boundaries. Due to the bounce back boundary condition, this data would never be streamed into the fluid domain and can thus be ignored.

However, a branching instruction is still necessary for the basic implementation. A single switch operation has been used to differ between the collision operator, the boundary condition and the velocity injection to run the driven cavity test simulation for the basic implementation.

---

9E.g. for the modulo operation, the result would be negative and has to be fixed to positive values which involves more computations and thus cycles
Using pointers to access density distributions

As it was already used for the wrapping method, the PTX code was used as an optimization criteria to further improve the access of density distributions[10]. Using array indices to access density distribution values creates more PTX code, which would usually result in a longer runtime.

To gain another increase in performance, two different ways to access the density distribution in the alpha and beta kernel are used:

For the **alpha kernel**, a pointer to the first density distribution \( dd_0 \) of a cell belonging to the work item is set, making it directly accessible via the pointer using the \(^*\)-operator for dereferencing pointers. After reading a density distribution, the next address of the density distribution is available directly by incrementing the pointer with \( \text{DOMAIN} \times \text{CELLS} \) and dereferencing the new pointer value for each new desired density distribution.

For the **beta kernel**, the pointer is set to the first density distribution of the cell at \((0,0,0)\). The density distributions are accessed by incrementing the pointer by \( \text{DOMAIN} \times \text{CELLS} \). The relative access within the current density distribution block in the global memory can then be done efficiently with the previously computed variables \( \text{read} \_\text{delta} \_\text{neg} \_x \) and \( \text{read} \_\text{delta} \_\text{neg} \_y \) and additional constant displacements for the \( y \) or \( z \) displacement as described in the previous section about the wrapping method.

Write operations in the alpha and beta kernel are implemented in the same way.

### 4.4 Results for driven cavity

Since it was not the overall aim of this thesis to implement a simple implementation of a driven cavity simulation on GPUs, only a few benchmark values are given in Figure 4.11 and 4.12.

The benchmarks have been created on a NVIDIA GeForce 285 GTX. The gravitational force as well as writing the velocity and density values to global memory in each time step was disabled. The density distribution values of the topmost fluid cells have been set to the equilibrium distributions for a given velocity to run a driven cavity simulation.

---

[10] It is again important to note that such an intermediate PTX code can only be used as a hint since this is not the byte machine code executed on GPUs
4.5. FREE SURFACE LBM

The free surface model is implemented with a single phase model. The meaning of free surface is that we can distinguish between fluid and gas cells. Single phase stands for the simulation of only one fluid type. This is still a good approximation because we are allowed to assume,

\footnote{Gas can also be simulated with the laws of fluids}
that the effects of the gas on the fluid are very small and thus negligible.

This free surface method was already implemented and tested for different types of simulations (rising bubbles, breaking dam and many more) [Thu07] and is also implemented in a 3D modeling & rendering software [BLE]. Furthermore, a simplified version of the free surface model has been tested previously with an OpenGL shader implementation to make sure that the ability to run the free surface simulation on massive-threaded GPUs is given [Schb].

The established free surface algorithm was slightly modified since it is not possible to handle the interface cells (cells which are partly filled with gas and fluid) adaptively with this implementation. Instead of including the surface tension in the simulation (as was done in many other implementations) (see [Thu03], [Thu07], [Bog09]) it is simply neglected. The reason is that the computation of surface tension would take more time and even without the surface, the results have already been satisfactory. It would be still possible to implement the surface tension, but it is expected that this would slow down the simulation tremendously. An idea for adaptive handling of the interface cells is given in chapter ??.

4.5.1 New cell types and values

To handle the free surfaces, the cell state variables are extended by two variables. To describe the mass within the current cell, \( m \) is used to keep track of the fluid mass within the cell. Due to the slightly incompressibility, another state variable is introduced. The fluid fraction \( \epsilon \) which represents the filling grade of the current cell. For each timestep, the new fluid fraction of a cell can then be computed by the fraction of the mass and the cells density:

\[
\epsilon = \frac{m}{\rho}
\]

To implement the different kinds of free surface cells, the basic LBM which uses only two different cell types is extended to four different types for a cell. For convenience, the type of a cell at position \( x \) is denoted as \( T(x) \):

<table>
<thead>
<tr>
<th>Cell type ( T(x) )</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle cells</td>
<td>( O )</td>
<td>This cell acts as a domain boundary or obstacle within the fluid</td>
</tr>
<tr>
<td>Fluid cells</td>
<td>( F )</td>
<td>Cells which are totally filled with fluid (( \epsilon \geq 1 ))</td>
</tr>
<tr>
<td>Gas cells</td>
<td>( G )</td>
<td>Cells which are empty (( \epsilon \leq 0 ))</td>
</tr>
<tr>
<td>Interface cells</td>
<td>( I )</td>
<td>Partially filled cells (0 &lt; ( \epsilon &lt; 1 ))</td>
</tr>
</tbody>
</table>

Negative values for the fluid fraction \( \epsilon \) and values above 1 due to mass tracking are further explained in the next section. The interface cells are used to separate the fluid cells from the gas cells. This is also one of the most crucial parts of the implementation: for a fluid cell, adjacent cells must not be gas cells and vice versa.

4.5.2 Mass tracking

The mass \( m \) changes according to the cell type and the fluid fraction of the current and the neighboring cells. Since the density distributions directly describe the amount of particles which leaves the cell within the next timestep in the direction of \( e_i \), they can be directly used to compute the outgoing mass. The incoming mass can be determined similarly by looking at
the adjacent cells and their density distributions aimed at the currently handled cell:

\[ \Delta m(x) = \sum_{i=0}^{17} A_i \cdot \left[ \text{incoming mass} \cdot f_i(x + \epsilon_i) - \text{outgoing mass} \cdot f_i(x) \right] \]

The scalar value \( A_i \) describes the area to exchange the fluid and depends on the neighboring flags and fluid fractions. For a easier reading, \( T_a \) is used for \( T(x + e_a) \) and \( T_L \) for \( T(x) \) in the following sections. The values for \( A_i \) account for the mass exchange using the adjacent cell types and fluid fractions as given in the following table:

<table>
<thead>
<tr>
<th>Rule#</th>
<th>( A_i )</th>
<th>condition</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>( T_L = G ) or ( T_i = G )</td>
<td>There is no mass exchange from or to a gas cell</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>( T_L = O ) or ( T_i = O )</td>
<td>There is no mass exchange from or to an obstacle cell</td>
</tr>
<tr>
<td>3</td>
<td>( \epsilon(x) + \epsilon(x + e_i) ) ( / 2 )</td>
<td>( T_L = I ) or ( T_i = I )</td>
<td>The mass exchange area to or from an interface cell is averaged by their fluid fractions</td>
</tr>
<tr>
<td>4</td>
<td>0/1</td>
<td>else: ( T_L = F ) and ( T_i = F )</td>
<td>2 different implementations (see table below)</td>
</tr>
</tbody>
</table>

For rule #4 with \( A_i = 1 \), an implementation was tested by involving the incoming and outgoing mass for fluid cells without fixing the fluid fraction afterwards, resulting in instabilities like fluid cells filled with much more fluid mass than makes a sense. This can be seen in cells which seemed to be a source of water for the final results in this thesis.

To circumvent this, the fluid mass is always set to the density \( \rho \) regardless of the incoming and outgoing flow, resulting in a fluid fraction of 1.

### 4.5.3 Interface change

The whole movement of the interface is based only on the change of type of the interface cells (Figure 4.13). Similar to the speed of only one cell length per time step to spread some information (e.g. an increased velocity at some point) in the basic implementation, the fluid interface can only move for a maximum of one cell per time step, further restricted by the maximum lattice velocity. Due to mass exchange, interface cells are converted to fluid cells or to gas cells according to their fluid fraction \( \epsilon \). No gas cell can be converted directly to a fluid cell and vice versa.

An additional small threshold value \( \epsilon \) is used for cell type conversions. This value is commonly used to avoid conversions of interfaces back to interface cells after a few time steps caused by errors introduced by the model and numerical limitations.

#### Interface to gas conversion:

If the fluid fraction within an interface cell is below \( 0.0 - \epsilon \), then the cell is converted to a gas cell. After conversion, a valid interface is reconstructed by converting adjacent fluid cells to interface cells.

#### Interface to fluid conversion:

For interface cells whose fluid fraction rises above \( 1.0 + \epsilon \), the cell is converted to a fluid cell. Similar to the previous conversion rule, a valid interface is maintained by converting adjacent...
4.5.4 Reconstruction of density distributions from gas cells

The density distributions streamed from gas cells to an interface cell have to be constructed. If the opposite density distribution is also streamed from a gas or obstacle cell, we use the equilibrium density distribution. Otherwise the incoming density distribution is reconstructed using the opposite density distribution. The rules are described more detailed in Table 4.2 and have to be traversed top-down until a matching rule is found.
4.6. FREE SURFACE IMPLEMENTATION WITH OPENCL

<table>
<thead>
<tr>
<th>storage</th>
<th>size</th>
<th>access</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fluid mass</td>
<td>1 float</td>
<td>RW</td>
<td>track fluid mass in cells</td>
</tr>
<tr>
<td>density distributions</td>
<td>19 floats</td>
<td>RW</td>
<td>input/output channels for the next timestep</td>
</tr>
<tr>
<td>cell flags</td>
<td>1 int</td>
<td>RW</td>
<td>distinguish between different cell types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Written for interface movement</td>
</tr>
<tr>
<td>velocity</td>
<td>3 floats</td>
<td>RW</td>
<td>written for collision and read for gas to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>interface conversion</td>
</tr>
<tr>
<td>density</td>
<td>1 float</td>
<td>RW</td>
<td>written for collision and read for gas to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>interface conversion</td>
</tr>
<tr>
<td>NEW cell flags</td>
<td>1 int</td>
<td>R</td>
<td>additional buffer to avoid race conditions</td>
</tr>
<tr>
<td>NEW fluid fraction</td>
<td>1 float</td>
<td>RW</td>
<td>additional buffer to avoid race conditions</td>
</tr>
</tbody>
</table>

Table 4.3: Data storages for free surface implementation

Special attention has to be given to obstacle cells. During interface movement, a new interface cell could be created next to the obstacle cell while the density distribution values within the obstacle cell are still invalid. One solution is simply to handle obstacle cells next to interface cells as gas cells. Another solution would be to initialize the density distribution values of obstacle cells when an adjacent gas cell is converted to an interface cell. While the first solution was developed for the A-A and A-B pattern, the latter one is only available for a specific version of the A-B pattern.

4.6 Free surface implementation with OpenCL

Different versions of the free surface extension have been implemented. Starting with the development using an A-A pattern, also a version using the A-B patterns memory layout was developed to compare the performance of both implementations. Since the A-B pattern is much easier to implement if an A-A pattern implementation is understood and implemented, only the A-A pattern is explained in detail. Some notes about the differences in the implementation between both patterns are given at appropriate points.

For the free surface implementation, the most crucial part is to maintain the free surface. An overview over the execution of one kernel is given in figure 4.14 which will be explained further in this section.

4.6.1 Additional memory buffers and cell types

To handle free surfaces, the data storage as described in Section 4.3.1 has to be modified and extended (table 4.3):

The storage areas marked with NEW are used to avoid race conditions when an interface is converted. For the same reason, a multipass method is used to make use of the GPUs parallel processing, introducing further cell types which represent the current processing state. For completeness, all flag types are given in the Table 4.4.

---

12 This can be activated with the DD_FROM_OBSTACLES_TO_INTERFACE option
Figure 4.14: Overview of LBM free surface handling for one kernel (alpha or beta). The new_* storage is used to avoid race conditions and has to be swapped with their original storage when executing the next kernel (field "A-A Opposite Kernels"). The kernels marked with "F: flag_name" handle cells with the flag "flag_name"

<table>
<thead>
<tr>
<th>cell type flag</th>
<th>abbreviation</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG_OBSTACLE</td>
<td>O</td>
<td>Obstacle cell</td>
</tr>
<tr>
<td>FLAG_FLUID</td>
<td>F</td>
<td>Fluid cell</td>
</tr>
<tr>
<td>FLAG_INTERFACE</td>
<td>I</td>
<td>Interface cell</td>
</tr>
<tr>
<td>FLAG_GAS</td>
<td>G</td>
<td>Gas cell</td>
</tr>
<tr>
<td>FLAG_INTERFACE_TO_FLUID</td>
<td>IF</td>
<td>The interface cell is converted to a fluid cell</td>
</tr>
<tr>
<td>FLAG_INTERFACE_TO_GAS</td>
<td>IG</td>
<td>The interface is converted to a gas cell</td>
</tr>
<tr>
<td>FLAG_GAS_TO_INTERFACE</td>
<td>GI</td>
<td>The gas cell is converted to an interface cell and its state properties are initialized</td>
</tr>
</tbody>
</table>

Table 4.4: Different flag types for free surface implementation
4.6. Free Surface Implementation with OpenCL

4.6.2 General issues

Cell flags

Instead of enumerating the cell flags, we use shifted bitmasks (e.g. \(1 << n\)) to speed up computations when a check for more than one cell type has to be done. E.g. for this implementation, fluid cells are tagged with \(1 << 1\) and interface cells with \(1 << 2\). A test if the cell is a fluid or an interface cell can then be done by testing the flag value \(v\) using \(\text{if } (v \& 6) \ldots\)” where \& is the bitwise AND operator.

Shared memory

As already explained in the previous section, the shared memory can be used to avoid uncoalesced reading for the basic implementation without free surfaces. For the basic implementation, the performance gain using shared memory was about 10 percent compared to the implementation without utilization of the shared memory.

For the development of a free surface implementation, using a shared memory has a large disadvantage: if the current cell for which a kernel is executed is a cell of type gas, we must not ignore loading density distributions to the shared memory because we do not know of what type the adjacent cell is. A solution would be that every kernel stores its cell flag to shared memory, doing a synchronization and then to read the neighbored flag data. According to that test, the density distribution is either loaded or not. Another solution which was used in this thesis was to leave the coalesced loading to the memory controller on the GPU. Concrete statements on which solution is better are not possible because the performance gain or loss strongly depends on the amount of fluid and interface cells within the domain.

An exemplary discussion of the coalesced access pattern and the shared memory usage will follow in Chapter 4.8 for the A-B pattern.

4.6.3 Kernels

In Figure 4.14 kernel names starting with an "F:" are followed by a kernel specific cell type. Those kernels return immediately after execution, when the current cells flag does not coincide with the kernel specific cell type.

Special attention is given to the deterministic behavior of some kernels to allow repeatable results and parallel computing with more GPUs on different computers for a further development in the future.

For a better overview the notation

\[
(T(n, t)|T(n + 1, t)|T(n + 2, t)) \rightarrow (T(n, t + 1)|T(n + 1, t + 1)|T(n + 2, t + 1))
\]

is used right after the subsections heading. The tuple describes a specific flag configuration of 3 cell flags in a 1D domain before and after the execution of the kernel.

PRE MAIN:

The first kernel computes how much mass is going out to adjacent cells. We are not allowed to join this kernel with the main collision kernel. When writing the density distributions after the MAIN kernel is executed, this would most probably produce race conditions.

For the A-B pattern, such race conditions cannot occur since we do not modify the density distributions read by adjacent work items. Therefore if the simulation has not been optimized for memory consumption, which is the case for an A-B pattern, a secondary buffer could be used to join this preprocessing kernel with the next kernel.
CHAPTER 4. FLUID SIMULATION WITH THE LATTICE BOLTZMANN METHOD

MAIN: \((G|I|F) \rightarrow (G|IF|F)\)

The main kernel first loads the fluid cell flag. If the current cell is of type interface, the reconstruction of the density distribution from gas and obstacle cells is prepared.

Pairs of opposite density distributions are loaded successively, together with the adjacent fluid fraction and cell types. The special enumeration of the lattice vectors (Fig. 4.1) stores the opposite lattice vectors next to each other, allowing us immediately to do computations for the current cell after the loading of two opposite density distributions. By doing the possible computations immediately after loading, other global memory accesses of different local work groups and warps can be executed to get a better memory latency compared to a solution which first loads all density distribution values and doing the computations subsequently.

After such a pair of density distributions was loaded, the density distributions are reconstructed in the case of an interface cell. In addition the fluid mass is incremented by the adjacent incoming density distributions according to the fluid fraction and cell types of this and the adjacent cells. To distinguish the different possibilities (no mass exchange with gas cells, scaled mass exchange if one of the involved cells is an interface cell and full mass exchange if both cells are of type fluid), an if-branching was used. To avoid the branching, another implementation has been tested using linear interpolation\(^{13}\) to select the appropriate scaling values without any performance increase. The reason for this is most likely the support of predicate registers resulting in a similar runtime as the linear interpolation.

The implementation of the collision operator and the boundary condition is equal to the code of the basic LBM implementation.

After the collision or boundary handling operator was applied, the values in Table 4.3 are written back to the global memory.

As the first pass of the multi pass surface movement handling, if the current cell is of type interface and the fluid fraction is above \(1.0 + \epsilon\), the flag of the cell is set to FLAG_INTERFACE_TO_FLUID to be handled by the next kernel.

F: INTERFACE_TO_FLUID: \((G|IF|F) \rightarrow (GI|F|F)\)

This kernel updates the adjacent cell type to FLAG_GAS_TO_INTERFACE, if the adjacent cell type is FLAG_GAS. After the possible updates, the flag of the current cell is set to FLAG_FLUID. This conversion was not done in the kernel ‘MAIN’ to avoid a non-deterministic behavior for that kind of conversions.

INTERFACE_TO_GAS: \((G|I|F) \rightarrow (G|IG|F)\)

This kernel handles all interface cells which have to be converted to gas cells. When the fluid fraction is below the threshold \(-\epsilon\), the cells flag is set FLAG_INTERFACE_TO_GAS.

F: INTERFACE_TO_GAS: \((G|IG|F) \rightarrow (G|G|F)\)

Similar to the kernel F: INTERFACE_TO_FLUID, this kernel takes care about interface cells which are converted to gas cells. To maintain a gapless interface, all adjacent cells of type FLAG_FLUID are set to FLAG_FLUID_TO_INTERFACE.

Now the implementation can take care of the ‘negative’ mass \((m \leq -\epsilon)\) within the cell, which can be activated in the existing implementation of this thesis as a non-deterministic step. For the test cases driven during the thesis, such a spreading of the remaining difference of mass was not necessary for an appealing visualization. But for completeness, the implementation of

\(^{13}\) e. g. if \((a) x = 2; else x = 3;\) can be implemented with \(x = 2 \cdot a + 3 \cdot (1 - a)\) to avoid branching instructions.
4.7. LONELY INTERFACE CELLS

A possible spreading of the negative mass for an increased accuracy is given here. Please note that such a spreading of too much mass in the FLAG_INTERFACE_TO_FLUID kernel has to be implemented for accuracy which was not done in this thesis.

An easy implementation was used which divides the mass by the amount of adjacent fluid and interface cells to store this amount back to an additional array. With this implementation, the number of adjacent fluid cells can change for the current cell within the same time step due to runtime conditions since cells of type FLAG_FLUID can be changed by concurrently running kernels. A deterministic behavior could be included by inserting another ”multipass-layer” after this one. The amount of remaining mass which was stored in this pass is then read back by the MASS_GATHER kernel.

F: GAS_TO_INTERFACE: \((GI|F|F) \rightarrow (I|F|F)\)

This kernel takes care of the gas cells which should be converted to interface cells. This demands a reconstruction of valid density distributions and the initialization of the fluid fraction and mass with legal values. To reconstruct valid density distributions, the velocity and density value of the adjacent cells of type fluid is averaged. No interface cells are included for deterministic reasons. Then, the density distributions are computed using the equilibrium distribution, stored to the appropriate array and the cell flag is set to the type FLAG_INTERFACE.

To give a deeper understanding of the deterministic problems, the accuracy of the simulations and the performance of the simulation which had to be handled during this thesis, we shall think about how to improve this kernel for an increased physical accuracy. To improve the accuracy, also the adjacent interface cells can be included in the averaging process. When this is done, another ”multipass-layer” has to be included to achieve deterministic behavior, since taking an average of the adjacent fluid and interface cells is not possible anymore in a deterministic step. The reason is that the type of some gas cells could be converted concurrently to the type FLAG_INTERFACE.

MASS_GATHER:

This optional kernel concerns the mass conservation when converting interface to gas cells and the mass exceeds below 0. The negative mass which was written by the kernel F: INTERFACE_TO_GAS is read back and added to the current mass of the cell.

A-A Opposite Kernels:

To avoid duplicate descriptions, A-A opposite kernels stands for the same kernels as previously described but with the opposite memory access of density distribution values according to the A-A pattern.

Every time, the kernels do not access the density distributions, the kernels can be used unmodified for the alpha or beta simulation step. Such shared kernels are F: INTERFACE_TO_FLUID, F: INTERFACE_TO_GAS, INTERFACE_TO_GAS. The kernel of type F: GAS_TO_INTERFACE is alpha and beta specific, since it has to update the density distribution values.

4.7 Lonely interface cells

The problem of “lonely” interface cells arises when the fluid moves very quickly. Single interface cells remain in the domain and cannot be converted to gas or fluid cells due to missing mass ex-
CHAPTER 4. FLUID SIMULATION WITH THE LATTICE BOLTZMANN METHOD

change with neighbored GUIs. Thus, they get “stuck in the air”. A similar effect is visible, when an interface cell exists within a fluid without a connection to a gas cell. As a simple solution, those cells are either converted to gas or fluid cells, according to the type of the surrounding cells. Such conversion rules are appended to the kernels \( F: \text{INTERFACE}_\text{TO}_\text{GAS} \) and \( F: \text{INTERFACE}_\text{TO}_\text{FLUID} \). The lonely interface killer was implemented as a non-deterministic step.

4.8 Results for free surface simulation on GPUs

4.8.1 Testing different rules

For the implementation in this diploma thesis, different rules for \( A_i \) compared to those in Section 4.5.2 have been tested:

1. Always averaging the area \( A_i \) even between fluid cells by replacing the value of \( A_i \) for rule 4 with \( A_s \)’s value of rule 3.

2. Replacing the condition of rule 3 by “\( T(x) == I \) AND \( T(x + e_i) == I \)” which means to average the area \( A_i \) for rule 3 only for adjacent interface cells.

3. Weighting the incoming and outgoing mass individually by their corresponding fluid fraction.

For the first two implementations, the differences between some tested fluid simulations have been negligible. The third implementation did not work at all, even if the mass conservation is fulfilled by this rule. A simple reason that this implementation cannot work was found by thinking about the fluid exchange between a fluid cell and an interface cell at zero velocity. The mass is changing even if it should not due to the zero velocity.

4.8.2 Benchmarking different patterns

Next, we compare performance results of the free surface implementation with different patterns and start with an explanation of details about the different patterns and versions:

**A-A pattern:** This pattern is equal to the free surface implementation described in this chapter.

**A-B pattern, version 1:** This implementation uses the A-B pattern with two density distribution buffers. The version number refers to the way the density distribution values are accessed. For version 1, the density distribution values are read with offsets similar to the beta kernel, but are written without any offset to the current cell’s storage. The advantage of this version is, that the reconstructed values of the density distributions for gas to interface conversions can be also written without any displacement.

**A-B pattern, version 1 utilizing the shared memory:** This implementation is identical to the previous one, but utilizes the shared memory to increase coalesced memory access.

**A-B pattern, version 2:** For the A-B pattern with version 2 the density distribution values are read from the current cell’s storage and thus without displacements like in the alpha kernel.

---

\[ \text{In the implementation, this can be activated via the LONELY\_INTERFACE\_KILLER preprocessor switch} \]
To combine the collision with the propagation operator, the modified density distribution values are written to the adjacent cells.

For a benchmark of the free surface implementation, the test scenario was initialized with a breaking dam simulation: One third of the domain at the right side was filled with water, the viscosity was set to 0.0001, the domain length along the x size to 0.30 and the gravity to 9.81. Furthermore the maximum velocity was round off to 0.25 within the kernel, which allows us to increase the time step and circumvent instabilities\footnote{Clearly, such a way to improve the time step size shall not be used for accurate simulations!}. Parameters have not been modified by this limitation. For a domain size of $64^3$ and by limiting the dimensionless maximum gravitation in the collision time step to 0.015625, one simulation step represents 0.00273241 seconds in real time.

The MLUPS depending on the fluid domain size, the size of the local work group and the used patterns and versions are given in Figure 4.5. Compared to the basic implementation, only about 20\% of the performance is reached. The reasons are mainly the increased code size with many branching instructions to handle the free surfaces and also the strongly increased utilization of the available bandwidth. Among others, this additional bandwidth was utilized for the fluid fraction values, which are read for the adjacent cells and the velocity and density

<table>
<thead>
<tr>
<th>Work group size:</th>
<th>32</th>
<th>64</th>
<th>80</th>
<th>128</th>
<th>160</th>
</tr>
</thead>
</table>

Table 4.5: Free surface breaking dam benchmark results for different work group, domain sizes and different pattern.
values which are stored in every time step and which are also read back in the pass for the reconstruction of the density distribution values for gas to interface conversions.

Due to the totally different resource usage of the kernels implemented for each pass, further optimizations can gain a slightly increased speedup: Such optimizations can be done by setting individual work group sizes for each kernel and/or to limit the register usage for each kernel which is available via an NVIDIA OpenCL extension [NVC]. As already mentioned in the section about the results of the basic implementation, this thesis is not intended to create all kinds of benchmarks to cover the possible combinations, since such results are usually only specific for one special kind of GPU.

However, one example should be given here by setting individual work group sizes for each kernel\textsuperscript{16}. We try to optimize the best MLUPS value with the A-B version 1 pattern from Figure 4.5 for a domain size of 128\textsuperscript{3} and a work group size of 160. According to a more accurate benchmark, 129.28 MLUPS (±0.02) are reached on average. When trying individually different work group sizes and by leaving the work group size fixed to 160 for the collision and propagation kernel, only an increase of 1 MLUPS can be reached with the following settings:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Local work group size</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm_coll_prop</td>
<td>160 (fixed)</td>
</tr>
<tr>
<td>interface_to_fluid_neighbors</td>
<td>256</td>
</tr>
<tr>
<td>interface_to_gas</td>
<td>160</td>
</tr>
<tr>
<td>interface_to_gas_neighbors</td>
<td>256</td>
</tr>
<tr>
<td>gather_mass</td>
<td>256</td>
</tr>
<tr>
<td>ab_flag_gas_to_interface</td>
<td>256</td>
</tr>
<tr>
<td>lbm_mass_scale</td>
<td>256</td>
</tr>
</tbody>
</table>

Since the collision and propagation kernel with its free surface handling is computationally the most intensive part, individual work group size optimizations for the other kernels only lead to a small performance increase.

Instead of testing all possible combinations, the NVIDIA Occupancy calculator [COC] worksheet can also be used to detect possible bottlenecks caused by a high amount of register usage limiting the number of threads executed in parallel. The occupancy number describes the relative amount of warps for which the resources are available on a multiprocessor to the maximum number of warps which can ever be executed on a multiprocessor. However, such information can only be used as a hint for the free surface fluid simulation. An occupancy close to 1 does not always coincide with a good performance. For example the the occupancy worksheet does not include the amount of coalesced and uncoalesced memory accesses.

When comparing the performance of the A-B pattern with the shared memory utilization to the implementations without the utilization, the MLUPS values go down about 10 % for domain sizes larger than 64. This behavior can be explained by the extra loading instructions in work items running on gas cells. Whenever the shared memory usage is activated, gas cells also have to load adjacent density distributions since they do not know the type of the adjacent cell. Therefore it seems that such superfluous operations of loading data for adjacent work items are slowing down the simulation. Since we only consider a standard breaking dam simulation so far, it is important to know that this performance difference also depends on the amount of fluid and interface cells within the domain as well as their constellation.

The performance differences between the two different versions of the A-B pattern implementations are almost negligible.

\textsuperscript{16}The number of work items and the limitation of the registers can be specified via the command line options -T and -R
4.8. RESULTS FOR FREE SURFACE SIMULATION ON GPUS

Beside the performance, memory consumption also plays an important role, depending on the requirements. While the A-A pattern saves almost the half of memory, the A-B patterns are usually 10–20% faster.

When benchmarking the initialization of the kernels (loading from file, compiling and linking), their initialization is much faster using a separate OpenCL program for every kernel. The resulting speedup for loading, compilation and binding of the kernels are almost 75% for initialization of the kernels. To gain more performance, the kernels can be compiled off-line.
This chapter gives a short introduction to the OpenGL (Open Graphics Library) render pipeline, describes some advanced features and explains some necessary and basic ingredients to assemble the algorithms in the next chapter. Readers who are already familiar with the basic rendering concepts, the rendering pipeline and advanced rendering techniques, like photon mapping, can omit this chapter.

5.1 Introduction to OpenGL

We start with an introduction to OpenGL [OGL], especially the OpenGL 3.2 core specification [Gro09a].

OpenGL is a widely adopted industry standard for 2D and 3D graphics. It is important to note, that this standard is not created by a reference implementation. Instead, OpenGL (as well as the OpenCL specification) is administered by the Khronos [KHRb] group, which is a member-funded organization of many vendors, which releases the OpenGL specifications and offers use of it free of charge. The API calls, which are described in the specification with its side effects, are usually implemented by the vendors of graphics cards or by open source projects (see [MHP], [NOU]) to offer developers a standard programming interface.

As an extension-scalable specification, which is implemented on many workstations as well as on supercomputers, it offers an ideal basis for the development of a visualization tool for scientific datasets. Furthermore, OpenGL is also available for many different platforms (e.g. Linux, MacOS, OPENStep, BeOS), even on embedded systems with OpenGL ES.

Due to its capability to offer bleeding edge visualization effects, it is also used for gaming consoles (e.g. for the Playstation 3 [PS3]) and 3D engines (e.g. UNiGine [UNI]), giving the visualization of scientific datasets the capability of advanced rendering features e.g. HDR rendering.

5.1.1 OpenGL 3.2 core profile

For this thesis, OpenGL 3.2 [Gro09a] is used, which was released in August 2009 [2]. Since the specification was published and an API was available only a few months before the start of this thesis, almost no sample code existed.

The OpenGL 3.2 specification also offers a compatibility mode, making it easier to adopt programs using the older APIs. Despite this, the code for this thesis was developed using only the core mode for the following reasons:

1. Apple, AMD, IBM, Intel, NVIDIA to give a few examples
2. Meanwhile, the OpenGL 3.3 and 4.0 version was released during the development of this thesis. This versions should be directly compatible with the code developed in this thesis.
The OpenGL 1.x specification was created for the GPUs at the time when the fixed function pipeline was the only way to program GPUs, where each functionality was programmed via CPU API calls. With OpenGL 2.0, the programmable pipeline was introduced via the shader programs offering more flexibility. It took over 3 years to release the specification for the OpenGL 3.2 core mode, removing features which became obsolete because of the programmable pipeline, without removing the functionality. The same functionality can still be implemented using new shader features like setting the point size in the vertex shader which offers more flexibility to the programmer. For readers who already developed OpenGL programs with OpenGL 1.x/2.x, the glBegin and glEnd functions as well as their respective functions to emit vertices and set the color or texture coordinate have been removed. Therefore, vertices can be rendered only using vertex lists stored in GPU buffers, which is usually faster compared with frequent API calls to emit one vertex and its attributes.

The GLSL (OpenGL shader language) was updated to version 1.5, implementing the necessary features to make the features of the newest GPUs accessible, and by moving some features from API calls to the more flexible shader programs.

The matrix stack and its dependent operations was completely removed, making the programming more flexible. With the features of current CPUs it is faster to compute the matrix operations in the developed program than calling the OpenGL API for that. To use the matrices, they have to be uploaded to the shaders via the API. Also one drawback should be stated: with this increased flexibility the API is no longer aware of what kind of matrix is uploaded. Therefore it is not possible to use a wrapper function, capturing matrix modifications for the projection matrix to allow an instant stereoscopic rendering for a closed source program.

Compared to older specifications, there is no longer any difference between vertex positions, normals and further vertex attributes for the API. Vertex shader input variables are bound to a vertex attribute array which is accessed in the shader successively via an input variable. Such vertex attributes can be the vertex position itself, the vertex normal, the texture coordinate or more generally scalar or vector expressions.

The developed framework is intended to be used in the future for further development. Therefore it is usually better to avoid using an old API and associated problems with further specification updates.

One major question remains. Should we prefer OpenGL for the visualization or is it better to use OpenCL copying the results to a texture or framebuffer? The answer strongly depends on the problem. For this thesis, OpenGL was used exclusively for rendering purposes. Since OpenCL optimizations strongly depend on the underlying hardware, OpenGL computations are usually faster in average on different vendor GPUs. For example the exact determination of the optimum work group size for OpenCL programs has to be tested by running the program many times. For OpenGL GPGPU computations, the work group size is completely hidden from the programmer and automatically scales without the need to specify and determine the optimum ‘work group size’.

Additionally, the developed program should also run on other GPUs without OpenCL support. OpenGL supports efficient write operations to textures. OpenCL write operations to textures are much slower than writing to the global memory whereas the performance of writing to textures in OpenGL using an FBOs (see Section 5.3) is much faster.
The NVIDIA GTX 285 also has no cache between the global memory and the work items\(^3\) whereas using OpenGL, we can make use of this texture cache. This is very important for shader units since they automatically load data from many adjacent voxels into the cache instead of only a single voxel (see Section 6.2 about volume casting as well as the histopyramid algorithm in Section 6.4.2).

To stay platform independent with the creation of the rendering context, like a window, and to make it easier to port the program to other platforms (e.g. MacOSX), the SDL [SDL] 1.3 development branch was taken to create an OpenGL 3.2 core context.

As a replacement for the missing matrix operations in OpenGL 3.2 core mode, GLM [GLM] was used as a replacement in the beginning of the development process but was replaced later on by a self developed smaller lib, since only a small subset of the matrix operation supported in GLM has been used.

### 5.2 Render pipeline

This section gives a short introduction to the render pipeline. A detailed description of this subtopic is beyond the scope of this thesis. However understanding the rendering pipeline (rendering primitives, shader units and the rasterization process) is necessary for the following sections.

We start with an explanation of shaders. To avoid any confusion, a little historical information about the word *shading* is given here. This word originated from shading programs many years ago, which have been used only to compute the fragment color. This was the only programmable part in the early days. Nowadays, even if other programmable parts in the pipeline are not directly connected to shading (e.g. vertex transformations), they are still called shaders. The shaders are further explained in context of the OpenGL 3.2 core mode.

A simplified version of the programmable render pipeline is given in Figure 5.1. Shaders are small programs directly executed on the GPU (similar to kernels running on the processing elements to use OpenCL notions). For the OpenGL 3.2 core specification, the different shader types are vertex, geometry and fragment shaders. Each shader type is first compiled and multiple compiled shader types can then be linked to a *program* which is capable of being run on the GPU. Such a program is enabled before any vertex is emitted.

Vertices are queued to the programmable pipeline with an OpenGL API call with, among others, the vertex type (point, line, triangle) and the number of vertices as parameters.

#### 5.2.1 Vertex shader:

Values from vertex attribute arrays are successively read and used as “input variables” for vertex shaders. For a speedup, those arrays have to be uploaded to the GPU memory during initialization. During rendering, the vertex attribute arrays can be quickly read without the CPU↔GPU bandwidth bottleneck. Vertex attributes can be 3D positions, normals, colors, texture coordinates or just an array with values of other semantics. After the execution of the vertex shader has started, it can do computations of almost any kind on these input variables. Typical transformations like rotation, scaling and translations are done by multiplication of the original vertex attribute like the vertex position with a matrix. Different methods exist to setup the matrix, however for this thesis, the matrix was only initialized to a fixed value for each vertex array rendering by setting the so called *uniform* variable to the values of the matrix via an OpenGL API call.

\(^3\)This is going to be changed for the FERMI cards
CHAPTER 5. INTRODUCTION TO RENDERING AND ADVANCED METHODS

Figure 5.1: Simplified rendering pipeline.

Since the GPU has to know the corners of a triangle to rasterize it by execution of fragment shaders, the standard output variable `gl_Position` has to be set in any way if no geometry shader is linked. If no geometry shader is linked, the output variables are immediately sent to the fragment shader using an interpolation to get texture coordinates in the inner area of a triangle without a distortion. This is hidden by the GPU using the homogeneous vertex coordinate stored in output variables.

After the vertex shader was executed, the vertices are assembled into geometric primitives (points, lines, triangles).

5.2.2 Geometry shader:

If a geometry shader is linked, it takes the vertices of one primitive stored in an array as input and constructs a number of output primitives. The type of output primitives for the geometry shader may vary from the input, but is fixed for a program. Geometry shaders have the ability to either emit no further primitive or to emit one or more primitives – e.g. to refine a surface.

5.2.3 Fragment shader:

Fragment shaders are executed for every pixel which is drawn on screen to usually fill the primitive. For computer games, the usual purpose of the fragment shader is to load the material parameters of the current fragment from a texture or another appropriate location and to compute the lighting. The output of the fragment shader is not sent directly to the framebuffer, but undergoes postprocessing.

5.2.4 Fragment postprocessing:

The fragment postprocessing is not part of any shader and is totally set up via API calls. Beside the depth test which is expected to be familiar to the reader, another important postprocessing operation which is important for this thesis is the blending operation. Blending becomes important for the implementation of photon mapping in Section 6.6.

Using blending, the source fragment color from the output of the fragment shader can be combined with the destination fragment color on the fragment buffer using different kinds of formulae like taking the maxima or minima; or simple linear interpolation. The most useful...
5.3 Computations with framebuffers

Talking about GPGPU using the OpenGL API usually means to abuse the fragment shaders to do computations. First of all, the computations can be rendered to textures by using a Frame Buffer Object (FBO). Instead of storing the fragments to a buffer which can be directly shown on the screen, a FBO can be bound to redirect the rendering to a texture. On a later stage, such a texture can then be accessed as an array or via a sampler doing a bilinear interpolation for 2D textures between the adjacent values for example.

With such FBOs, a fragment shader program can then be used to read data from textures and write data to a fragment location. This fragment location is fixed to an individual texel for each execution of the fragment shader (Figure 5.2). In OpenCL semantics, this is like executing a kernel with a texture as a read only memory buffer and storing the data to a per-work-item fixed position in another texture. The really important thing, which was already mentioned in the OpenCL chapter, is that the OpenGL has a built-in cache to read from the textures and also supports efficient write operations to the output texture – most probably due to the pre-execution fixed output position.

The execution of such GPGPU fragment shader programs is usually triggered by rendering a primitive (e.g. a quad covering the full viewport or an oversized triangle which is automatically clipped to a quad covering again the whole viewport). Then the shader program figures out its unique id by reading its fragment output coordinate. After doing the computations, the shader program stores the result to the output color or discards the fragment if no modification of the pixel in the currently bound framebuffer is to be done. The output is not restricted to one color only, by binding multiple textures to a FBO. Then the fragment shader has to write to multiple variables to set the texel color of each bound texture.

For every fragment shader, the output position is fixed for every fragment, which is one of the main disadvantages of OpenGL over OpenCL. Therefore, some algorithms have to be reordered to fit to the restrictions of the OpenGL “data output” and, very often, such a reordering is not possible or would significantly reduce the efficiency of the implementation. There are ways to

---

4gl_FragCoord variable in the fragment shader
circumvent this problem, like rendering points with geometry shaders, which are usually very slow and thus not considered further.

## 5.4 Cube Mapping

Cube mapping is frequently used to simulate reflections on glossy mirroring surfaces for far distant objects. The cube map is accessed via the OpenGL API as one texture, but with different so called **texture targets** – one target for every cube side (Fig. 5.3). The cube map is created by rendering the scene six times with the field of view set to 90°, the view directions along the axes and the up vector according to the cube map specification. To store the cube map to a texture, a FBO is used, bound to the according **texture target**.

Once created, the lookup of a value in the cube map, by reading the texture value with a 3D vector (e.g. the reflected vector), is **hardware accelerated**. First the vector is normalized, then the component with the largest absolute value is selected as $T \in \{x, y, z\}$. With $T$, the two cube sides pierced by the axis $T$ are selected (e.g. $x$ selects the left and right cube side) and the sign of the 3D lookup vector component $T$ selects one of those cube sides (e.g. minus selects the left side). The other components of the lookup vector are divided by the absolute value of $T$ and can then be used as texture coordinates in the selected cube side texture.

The lookup operation is natively available in shaders, so the only things which have to be implemented are the creation of the cubemap and a texel fetch in the cube map with the 3D vector.

## 5.5 Flat textures

Flat texture have been introduced to overcome the missing availability of GPUs to write to 3D textures. On today’s GPUs, writing to 3D textures is only available for selected layers. Different ways exist to select the layer, e.g. manually via CPU API calls by binding a texture to the framebuffer, which has the disadvantage that a quad has to be drawn for every slice. For current generations of GPUs, it is also possible to select the output layer of a 3D texture via the geometry shader. Since the computation time of the latter method is expected to be similar to that of flat textures and, since flat textures are more common, they are also used in this thesis to do computations on volumetric textures.

To run a computation on a volumetric dataset stored in a flat texture, a single quad is rendered while the volumetric data is stored in a 2D texture (see Fig. 5.4). In flat textures,
5.6 Depth peeling and front/back face extraction

Depth peeling or front/back face rendering is usually used to extract a mesh, or some other kind of object representation like volumetric data sets, to 2D textures as an approximation.

Volume slices are stored next to each other in a 2D texture. The slices are also stored with a y-displacement to overcome the limited texture size.

For a volumetric texture of size $V_{x,y,z}$, which stores $n$ slices in each flat texture row, the 3D position $P_{x,y,z}$ is then computed from the fragment position $F_{x,y}$ with

$$
P_x = F_x \mod V_x \\
P_y = F_y \mod V_y \\
P_z = \lfloor F_x/V_x \rfloor + n \cdot \lfloor F_y/V_y \rfloor
$$

In the case that $V_{x,y}$ are both a power of two, the modulo and division operations can be replaced by bitwise and shift operators.

---

5. With a maximum texture size of $8192^2$ on current NVIDIA GPUs, storing a domain with a size of $128^3$ with slices next to each other would take a texture with a size of at least $16384 \times 128$ which would exceed the maximum allowed texture width.
CHAPTER 5. INTRODUCTION TO RENDERING AND ADVANCED METHODS

for the mesh surface. It is used in many different algorithms, like rendering transparent and non-refractive surfaces in correct ordering [Eve01], to voxelize an object into a flat volume texture [LFWK03]. Here, we use it to approximate refractions and photon mapping of caustics on the surfaces extracted by depth peeling or front/back face extraction.

Depth peeling

Multiple depth peeling passes of an object use a second depth buffer. The peeling passes are done while no further fragments are drawn to the framebuffer, or up to a fixed number of peeling passes.

In the first pass, face culling is enabled as for usual 3D renderings and the depth of the front faces is stored to the depth buffer.

For the 2nd and further peeling passes, face culling is disabled to render both front and back faces. The depth texture of the previous peeling pass (including the first pass) is used to discard fragments if the previous depth value is larger than the current one. This means, that the current fragment is closer to the viewpoint than the fragment from the previous peeling pass. To avoid flickering caused by slightly different interpolations for every frame, an additional very small offset is used:

    if (gl_FragCoord.z <= prev_depth+0.000001) /* discard fragment */ ...

When using many depth peeling textures in other shaders (e.g. for eye-space refractions with multiple layers), a spilling of the limited amount of textures which can be bound to a shader at the same time is avoided by using a 2D texture array. Each layer of such a texture array is then accessible by the 3rd texture component similar to 3D texture volumes.

To accelerate the depth peeling, advanced methods have been developed. For example stencil routed a-buffers [MB07], which make use of multisample antialiasing (MSAA), render target textures to store multiple layers within one rendering process. Such acceleration methods have not been further investigated in this thesis since only two depth levels have been extracted and the performance increase would not largely accelerate the visualization overall.

Front/Back face extraction

To extract only the front and back faces of a mesh, the front faces are rendered as usual, while the back faces are rendered to a second texture by culling all front faces, inverting the depth test and clearing the depth buffer with 0.0.

The advantage of the front/back face extraction and depth peeling is, that the mesh is approximated by only two textures. Compared to the depth peeling with two levels, the front/back face extraction has the great advantage that the viewspace refractions look more realistic (see Section 6.5 for more details).

5.7 Marching Cubes

The MC (Marching Cubes) algorithm was created to extract 3D iso surfaces out of volumetric datasets [LC87], and was patented in 1985. Since the patent has now expired the algorithm can be used without royalties.

The iso surface is extracted using only adjacent cell information. For every cell, an 8-bit MC index is computed from the values at the cube edges. Every bit is computed from the value at a cube corner. The bit is set to 1 if the value at the vertex is above the iso value, otherwise
it is set to 0. By removing duplicates of the 256 possible combinations due to symmetries, 15 basic cases remain (Fig. 5.6).

The triangle table stores the vertex indices (aligned at the cube edges) which have to be rendered as triangles for the surface representation.

Then the indices of vertices at the cube edges belonging to the triangles, which represent the iso surface of the cell, can be read directly from an array storing the triangle vertex indices for all 255 cases, using the MC index of the current cell as the array index.

For a smoother surface, the vertices are displaced along the edges according to the values at the corners using linear interpolation.
5.8 Shadow mapping

Shadow mapping is one of the easiest methods to determine shadowed surfaces so as to approximate interactive direct lighting with moving objects causing shadows.

The shadow map is created by rendering the scene in light space and storing the depth value to a framebuffer (Fig. 5.7). The accuracy of shadow mapping strongly depends on the resolution of the shadow map texture. A surface is assumed to be lit, when the surface is “visible” in the shadow map.

When rendering the scene as usual, a lookup in the shadow map is done at the end of the fragment shader by projecting the point \( \vec{p} \) from the view space coordinate back to the world space by using the inverse of the view matrix: \( V^{-1} \). Then the point is projected with the same view matrix \( V \) and projection matrix \( P \) which was used for the creation of the shadow map. Finally, the point is transformed by a scaling and translation matrix to the texture space with:

\[
X := \begin{pmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0.5 & 0.5 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Putting it all together, the homogeneous texture lookup coordinate is computed with \( \vec{t} = X \cdot P \cdot V \cdot V^{-1} \cdot \vec{p} \). Before a lookup in the shadow map texture can be done, the homogeneous point components have to be divided by the fourth component to get the inhomogeneous, and thus correct, 2D-texture coordinate.

5.9 Photon mapping

The photon mapping is approximated with a GPU based approach.

Photon mapping was published in 1994 as an approximation for a global illumination using a Monte Carlo method (see [Jen01]). It is capable of handling a fast estimation of the indirect illuminated surfaces by emitting photons.

For the photon mapping on GPU in Section 6.6 a modified version of the original algorithm is used to compute only caustics. Therefore, the algorithm is not explained in every detail since most parts are going to be replaced and approximated for GPU optimized rendering.

The original photon mapping algorithm can be split up in two passes: emission of the photons from the light source to the world, and the utilization of the photons for lighting purposes.

Emission of photons

For point lights, photons are emitted in a random direction as in Figure 5.8. Each photon transports an amount of energy, depending on the energy of the light source and the fraction of the light source energy which this photon represents. The path of the photon is traced, until a surface is hit. After such an intersection was found, the next step of a photon depends on the used material parameters. To simplify the explanation, we only consider diffuse cases. Subsequently, the photon can be either uniformly spread using a random value (diffuse indirect lighting) or remains at the surface. The decision whether the photon is spread or remains at the surface is taken by a random variable dependent on the material parameter. After a maximum amount of intersections, the position and incident direction of the photon are stored to the photon map.

\[^{6}\text{For this implementation the triangle table was taken from http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/}\]
5.9. PHOTON MAPPING

Figure 5.8: Simple sketch for photon mapping: Photon path is emitted in a random direction at the point light source. On diffuse surfaces, the decision to further emit the photon is done via a Monte Carlo method. The direction is also randomly chosen for diffuse surfaces.

Lighting

The illuminated objects are rendered as usual, but with an additional step to simulate the lighting. The emitted light on a surface caused by an incoming photon depends on the light intensity, the incoming photon direction, the surface normal and the material parameters. Using this information, the emitted light for every pixel is computed by averaging the incoming light energy of photons within a half sphere of a specific radius.

Optimization of the photon map

Since photons have to be searched within a specific radius, it is desirable that the photons are stored in a data structure which offers a fast access to search for photons within a specific radius.

Well known straightforward ways to implement such a fast access are either hash tables or a spatial discretization. However to reduce the tests for valid photons, which are inside the half sphere and to optimize the access on average, the photons are stored in a so called kD-tree, where $k$ specifies the number of dimensions which would be 3 in our case. A short explanation of the kD-tree is given to look at the computational complexity.

Every tree node represents an axis aligned box within the 3D-tree, whereas the root node represents the whole $\mathbb{R}^3$ space. Each tree node separates the space of the parent node with a plane perpendicular aligned to one axis. Such a plane usually divides the photons to an almost balanced number on each side to achieve a balanced tree. To balance the number of photons, they are usually sorted. Apart from building up the tree, this again takes some computation time of at least $O(n \log n)$ for each dimension, since a sorting algorithm is used to balance the tree.

To solve such a complexity, which would be overwhelming to handle with todays GPUs, for our purposes we again use an approximation in Section 6.6, which is indeed a kind of a spatial discretization.
6

LBM visualization using OpenGL

This chapter is about the visualization of volumetric datasets produced by the free surface simulation explained in Section 4.6. The desired result of this thesis is not only a visualization of a shaded fluid surface, but also advanced rendering effects like refractions on the fluid surface as well as a simple form of photon mapping for the simulation of caustics.

This chapter is split up into three parts:
The first one is about the surface extraction with volume casting algorithms and refractions with volume tracing.
The second part extracts the surface with the marching cubes algorithm. Since intersection tests with polygons are usually only done on off-line optimized data, the refractions have been implemented with eye-space approximations for the second part.
The first two parts have been developed to compare two different implementations according to their surface representation and their performance.
The last part explains an implementation of a photon mapping algorithm in light space to account for caustics. Both surface extraction methods from the first and second part have been used to extract the front and back faces to compare their performance and visual appearance.

To describe some performance effects of OpenGL parts, sometimes the OpenCL layout and how OpenCL is accessing the GPU hardware is used for the explanations, since programming OpenCL hides not as much information about the underlying hardware as is the case for OpenGL. Basic information about OpenCL is available in Chapter 3.

6.1 Accessing an OpenCL array as an OpenGL volume texture

After a few simulation time steps, the fluid fraction dataset has to be made available to the OpenGL context. Since OpenCL global memory buffers are not available for OpenGL contexts, they have to be copied to an OpenGL texture to make them accessible with an OpenGL shader.

To accelerate such copy operations, OpenCL offers an extension\textsuperscript{1} which can be used to copy to and from OpenGL textures with specific internal dataformats. Unfortunately, the copy operation from OpenCL buffers to a red channel volumetric texture that is not supported by the current NVIDIA drivers and copying to a RGBA texture would result in an incorrect volumetric dataset. The copy operation assumes that the GBA components also have to be initialized with the OpenCL buffer data and thus only the R component of the first voxel would be initialized with its corresponding fluid fraction. As a solution, the copy operation was done via a copy operation to the CPU memory and then back to the GPU to a volume texture. However, this performance dependency was not large enough to be one of the main problems for running the program in real time.

For the marching cubes extraction, an improved version was developed without the CPU↔GPU copy operation (see Section 6.4.3). Because the MC extraction is based on a

\textsuperscript{1}CL\_GL\_CONTEXT\_KHR
flat texture, whereas the volume tracing shaders in this section have been developed on a 3D volume texture as input.

6.2 Volume casting

The casting algorithm was intended to compute only the first intersection with a surface. For example it was used in games like Wolfenstein-3D\footnote{http://en.wikipedia.org/wiki/Wolfenstein_3-D} for 2D wall intersections of the view ray and in Outcast\footnote{http://en.wikipedia.org/wiki/Outcast_%28video_game%29} with a combination of ray casting of a height map for terrain rendering and rendering other objects via rasterization. Nowadays, raycasting is mainly used on volumetric datasets.

Volume ray casting has also been done plenty of times on GPUs – such as rendering CT scans (see \cite{KW03}). Those implementations were aimed to render high resolution datasets\footnote{At least $512 \times 512 \times 256$}, while the rendering was accelerated by a preprocessing step for empty space skipping. Also, no refractions have been used for volumetric datasets since the visualization was intended to be used for medical and engineering purposes, while the visualization in this thesis should give a good visual appearance of a fluid surface.

Other optimization ideas like the assumption that small changes in the view direction change the intersection point only slightly, and thus reusing the last intersection point, cannot also be used due to the possibly high frequency of the fluid surface (bubbles, droplets, ...).

6.2.1 Volume casting for fluid fraction surface

When rendering fluids directly from real time simulations, new problems arise.

Empty space skipping would force us to do a preprocessing step which has to be done in every frame. As the fluid domains are very small for our real time examples, the expectations of a great acceleration are low and thus no effort has been put towards this optimization. This should bring only a large speedup for higher domain resolutions.

Typical volumetric datasets of existing GPU algorithms like CT datasets have a size of $512 \times 512 \times$ depth, whereas the resolution of the fluid volumetric dataset is much smaller for our real time purposes. Therefore the iso surface of the fluid fraction would be rendered highly oversampled (pixels covered by a single voxel of a volume in horizontal or vertical direction $\gg 1$). This kind of oversampling also forces the use of a different surface extraction technique than for usual ray casting algorithms running on GPU.

Since the interface layer of our fluid simulation is mostly only one cell in size, accurate intersection points are necessary to extract accurate normals. Slight changes in normals due to wrong surface extractions would be negligible for the iso surface representation, but the refractions are very sensitive to small changes in normals, producing much more noise.

6.2.2 Basic implementation for ray casting

So far, we considered problems that arose due to our special case to render real time created volumetric free surface datasets. Now the standard implementation of ray casting algorithms on GPUs will be explained.

For volumetric datasets, it is the easiest way to store those datasets into a 3D texture since the graphics card offers an automatic trilinear interpolation from neighboring cell values. This trilinear interpolation creates curved isosurfaces. To stay simple we first look at a bilinear
6.2. VOLUME CASTING

![Figure 6.1: Left image: Asymptotes and isolines for 2D isolines using bilinear interpolation, Right image: Red lines: Bilinear interpolated values for $v(0,0) = 1.0$, $v(1,0) = 0.4$, $v(0,1) = 0.4$, $v(1,1) = 0.45$, Blue lines: Surface with isovalue 0.5](image)

Interpolation in 2D as shown in Figure 6.1. In 3D, the trilinear interpolation makes it hard to find the exact intersection point; zero crossings of a polynomial of degree 3 have to be solved. Therefore only approximations are used in the upcoming sections.

To iterate stepwise through a volume, the two piercing points of the volume for each ray have to be known. In the GPU algorithm suggested in [KW03], those two intersection points have been determined explicitly by a 2-pass method. The first pass is to compute the far distant intersection point rendering a cube with activated front face culling to an active FBO. When the color of the cube vertices are set to the according volume coordinate\(^5\) (Fig. 6.2), the color on the fragment pixel is the corresponding piercing point in volume space. At the second pass, the ray casting is initiated by rendering the cube with the default activated back face culling and reading the exit position from the previously computed texture. However, for the current GPU generations it is faster to replace the first pass by computation of the exit point in the fragment shader of the second pass, giving a slight performance increase. Those computations are given in Algorithm 1.

For adjacent screen space fragments, the sampling makes use of the built-in texture cache. Even if it is not manifested in the OpenGL specification, the textures on GPUs are stored using space filling curves to gain a data locality for interpolation. This significantly enhances the performance even of oversampled volume casting.

A perspective on possible performance enhancements for strongly oversampled volumes is given in Chapter ??.

\(^5\)When we work with normalized volume coordinates, those are $\in [0;1]^3$
Algorithm 1 Computation of the second intersection point for $[0;1]^3$ cube in pseudo GLSL code

**Vertex Input:**

```vertex-code
vec3 vpos; // vertex position
dvec3 vol_pos; // volume coordinate
uniform mat3 vmn_matrix3; // inverse transp. of view model matrix
uniform mat3 vm_matrix3; // view model matrix
```

**Vertex Shader:**

```vertex-code
gl_Position = pvm_matrix * vpos;
dir_in = vpos.xyz + vec3(vm_matrix[3])*vmn_matrix3;
vol_pos_in = vol_pos;
```

**Vertex OUTPUT / Fragment INPUT:**

```vertex-code
vec3 vol_pos_in; // volume coordinate
dvec3 dir_in; // direction of incoming ray for fragment shader
```

**Fragment Shader:**

```fragment-code
// Positive, if the collision surface does not pass the origin
vec3 collide_surface = vec3(greaterThanEqual(dir_in, vec3(0)));
// Distance for collision with possible cube quad
vec3 dist = (collide_surface - vol_pos_in)/dir_in;
// Use the minimum intersection distance as first intersection
float max_dist = min(min(min(dist[0], dist[1]), dist[2]), sqrt(3.0));
...```

6.2.3 Normal at intersection point

For each intersection point $P$, the normal is approximated by central differences with a sampling distance of a voxel’s unit length; linearly interpolated values at the points $(P_x\pm1, P_y, P_z)$, $(P_x, P_y\pm1, P_z)$ and $(P_x, P_y, P_z\pm1)$.

A faster and frequently used way to compute a normal is to use only forward or backward differences. Using the fluid fraction, such forward differences produced unnatural looking surfaces and were therefore avoided every time when normals are computed in this thesis.

6.2.4 Implemented algorithms

This section is about the different implementations which have been developed for this thesis. Their penalties as well as their advantages are described whilst the overall results can be found in the following sections.

6.2.5 “Simple” raycasting

We start with a straightforward approach. The volume is sampled with equidistant step sizes. The ray casting stops, if a sample value is above the desired iso value. If the ray exits the volume before an exceeded iso value could be detected, the fragment is simply discarded since no iso surface was hit by the ray.

Beside performance optimizations, this simple ray casting was originally developed to render datasets with resolutions similar to the screen resolution. Therefore, either the step size has to be set to a lower value to render low-resolution volumetric datasets, resulting in a high render time, or a large step size is used, resulting in noisy surfaces.
6.2. VOLUME CASTING

Figure 6.3: Equidistant sampling with a step length of 0.5 and cube steps sampling with no fixed step length

6.2.6 Interpolated raycasting

The interpolated raycasting works in a similar way to the equidistant sampling of the previous subsection, but is extended in two places. First, the old 3D sampling point and its according sampling value is stored. The second extension is implemented after the current sampling value is exceeding the iso value. To improve the 3D position, the previous sampling point is also involved in the computation of the real piercing point to run a linear interpolation. Let \( n \) be the current step when the value \( v_n \) at the current position \( p_n \) exceeds the iso value \( i \). Then the improved approximated intersection position \( p_{final} \) is computed by:

\[
p_{final} = p_{n-1} \cdot (i - v_{n-1}) + p_n \cdot (v_n - i) \\
\frac{v_n - v_{n-1}}{v_n - v_{n-1}}
\]

Even if the intersection point is improved, it is still not accurate since the iso surface is created with a trilinear interpolation.

When we think about optimizations in the OpenCL context, and by knowing that more local work groups can be executed to hide memory latency, we can also improve the register requirements by moving the buffering of the current sampling position and its register usage to the intersection test itself. Since the previous sampling position is known simply by going back one equidistant step and the sampling value would most probably still be in the texture cache, the buffering can be removed by doing a few more computations in the intersection. Go one step back and read at this sampling point.

Even if interpolation is also used by the cube steps algorithm in the next subsection, the term interpolated ray casting only refers to this algorithm here.

6.2.7 Cube steps volume raycasting

Since the simple, interpolated volume ray casting works with equidistant steps, the shader program samples the cell mostly in the inner area where the surface values are approximated with a trilinear interpolation even if the accurate surface is not known. Therefore the cube steps algorithm was developed to sample on the side of the cells only where the values are sampled with a bilinear interpolation. The intersection point with the iso surface is then approximated using a linear interpolation between the last two sampling points. This also leads away from a sampling method with a fixed step size and towards adaptive step sizes. The differences in the sampling methods are shown in Figure 6.3 for an arbitrary ray.
Apart from the variable step size, further advantages of the cube steps method are presented in Section 6.2.9.

To compute the piercing points, a few modifications to our previously explained volume casting algorithm are needed. The initialization and the computation of the piercing points on the voxel borders are given in Listing 6.1.

---

Listing 6.1: Pseudocode for cube steps shader

```cpp
// Fragment INPUT:
vec3 P; // start point for sampling
vec3 D; // sampling direction

// collision plane increment values (-1, 0 or +1) for axes
cvec3 cpi = sign(D);

// setup position and align to the edge of
// the cell for accurate sampling
vec3 pos = P + cpi*vec3(0.5);

// "cache" inverse direction (frequently used in volume sampling)
vec3 inv_dir = vec3(1.0)/(D);

// compute plane for next ray crossing to get positive distances
vec3 next_plane = floor(pos) +
 vec3(greaterThanEqual(cpi, vec3(0,0,0)));

for (...) // for all sampling points
{
  // compute distances to next axes
  vec3 dist = (next_plane - P)*inv_dir;

  // linear interpolation is faster
  // see the text for further information
  float dist0leq1 = float(dist[0] <= dist[1]);
  float dist1leq2 = float(dist[1] <= dist[2]);
  float dist0leq2 = float(dist[0] <= dist[2]);

  vec3 mdist; // selection of minimum intersection distance
  mdist[0] = dist0leq1*dist0leq2;
  mdist[1] = dist1leq2*(1.0f-dist0leq1);
  mdist[2] = (1.0f-dist0leq2)*(1.0f-dist1leq2);

  // compute next axes for collision
  next_plane += cpi*mdist;
  // compute new position of current sampling point
  current_pos = P + D*dot(dist, mdist);
  ...
}
```

To enhance the computation time slightly, the shader program was modified to make use of a linear interpolation to avoid branching instructions, which was also an unexpected result.

---

[6]This is a well known shader improvement and has nothing to do with the previously often mentioned interpolation on the volumetric dataset.
6.2. VOLUME CASTING

6.2.8 Marching cubes

As the last implementation to extract an iso surface directly from a volume, a straightforward approach was developed using the marching cubes algorithm directly on volumetric datasets. Even if the expectations on the runtime were tremendously low, it was implemented mainly for debugging purposes. This method is a simple way to implement a computation of accurate refractions for the later section to compare and validate their results with other eye-space approximated ray tracings (see Section 6.5).

The implementation of this ray casting computes the collision points of each ray and all possible MC triangles of the current cell. The cube steps algorithm is used to iterate over the cells since we are only interested in sampling once per cell. For each new cell, the marching cubes index is computed and the face indices are loaded for the MC index. Next, the interpolated position on the cube edges for all vertices are determined and the main computational part is started. The intersection of all faces with the ray \cite{MT05} is computed, while keeping the nearest intersection. If no intersection was found, the iteration continues on the next cell along the ray.

6.2.9 Results for volume casting

A concrete comparison is not possible due to the different visualization methods and their performances. For example when the iso surface is extracted to be used on posters, aliasing effects should be suppressed as much as possible since they are clearly visible. On the other side, when rendering frames for a movie the aliasing effects on the borders could almost be invisible due to the border movement.

Visual appearance

We start with a comparison of the visualization methods with different test scenarios:

- The first test scenario is given in Figure 6.4 with a sphere stored at a volume with a low resolution of $8^3$. The ray casting methods presented in the previous sections have been used to render the volumetric datasets. Obviously, the simple ray casting method produces the worst results due to the totally oversampled volume. The almost invisible
CHAPTER 6. LBM VISUALIZATION USING OPENGL

Interpolated volume casting  Ray casted marching cubes

Figure 6.5: Volume casting with interpolation with a step size of 0.5 vs. ray casted marching cubes for a falling drop simulation with a domain size of $64^3$.

differences between the other methods are not caused by limited printing or display color range, they are indeed very small for this test case.

- The second test scenario is a falling drop simulation. Two screenshots for the interpolated ray casting and the marching cubes are given in Figure 6.5. The left handed screenshot shows one of the problems of the interpolated ray casting method. The silhouette can be noisy since the volume is sampled with equidistant step sizes which makes it possible to skip the iso surface at sharp-edged borders. We will have a closer look at such kinds of sampling problems in the next scenario.

One drawback for visualization with marching cubes is the visibility of some edges shared by faces. This effect can be explained with a large difference in change of the normal for two adjacent faces and the approximation of the normal with barycentric interpolation.

- The third and final scenario is a breaking dam simulation. The most interesting part for our purposes is the area close to the thin fluid layer on the right side of the domain, which is created after a few time steps. Screenshots for three different visualization methods are given in Figure 6.6. For some constellations which depend on the fluid fraction of the thin layer and the view, strong artifacts are created for such a thin fluid layer when the interpolated ray casting is used, similar to the previous test scenario. Since these artifacts are undesired, we have a closer look at different cases given in Figure 6.7.

Example rays and the corresponding sampling points for an equidistant step length are shown in the top row. For (A), rendering the iso surface is skipped, even if the ray goes straight through the sampling point. Only slight changes in view are necessary to render the iso surface as shown in (B) and (C).

Cube step sampling is not as prone to such artifacts as sampling with equidistant step sizes. A few examples are shown in the bottom row. Even when changing the view angle slightly, the iso surface is rendered as sketched in (D) and (E) since the edges are not skipped. However, there are constellations when the trilinear interpolated iso surface is not detected correctly (F).

It is worth mentioning here that the real iso surface can be totally different from the iso surface created by the trilinear interpolation.

When the volumetric dataset is rotated, the cube steps and MCs algorithms also have the further advantage that the artifacts do not appear and disappear with a random behavior.
6.2. VOLUME CASTING

Figure 6.6: Different visualisations for one timestep of the breaking dam simulation

Figure 6.7: Different cases for equidistant sampling and cube steps sampling. See the text for an explanation.
CHAPTER 6. LBM VISUALIZATION USING OPENGL

<table>
<thead>
<tr>
<th>Implementation / Domain Size</th>
<th>16³</th>
<th>32³</th>
<th>64³</th>
<th>128³</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>2015</td>
<td>1420</td>
<td>850</td>
<td>501</td>
</tr>
<tr>
<td>Interpolation</td>
<td>1941</td>
<td>1351</td>
<td>819</td>
<td>491</td>
</tr>
<tr>
<td>Cube steps</td>
<td>1343</td>
<td>832</td>
<td>482</td>
<td>261</td>
</tr>
<tr>
<td>MCs (raycasted)</td>
<td>196</td>
<td>107</td>
<td>60</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 6.1: FPS results for volume casting

Performance

Average FPS for given volume sizes initialized with a sphere with a radius of \( \frac{1}{4} \) of the volume size are given in Table 6.1. The height of the volume cube covers the screen height. The sphere covers \( \frac{1}{4} \) of the viewport’s height and the viewport resolution is 800 × 600. The step size for the two equidistant sampling methods is set to 0.5.

The simple version is the fastest, but also useless due to the bad visual appearance for our purposes. However the simple method is interesting for previews of larger datasets.

The interpolation method gives the best performance results next to the simple version with only a slight performance decrease.

The cube steps version is slower than the interpolated ray casting on the NVIDIA GTX 285, which the benchmarks are driven on. This version should be used when an accurate surface has to be rendered with an appropriate averaging between the performance and visual appearance. To give yet another example for the differences when running the same program on different GPUs, the cube steps ray casting is faster on a NVIDIA 8600 GS.

Another way to create more accurate surfaces without the cube steps method would be to decrease the step size for the interpolated ray casting method which also decreases the FPS.

Using MCs with ray casting is by far the worst idea. Since MCs do not suffer from the artifacts, more interest is given to them in one of the next sections for a different kind of faster implementation and approximation.

6.3 Volume tracing with refractions

Volume tracing can be seen as an extension to the volume casting algorithm, shooting out one or more rays from each first intersection point to improve the visual appearance. For ray tracing of polygonal scenes the new rays at the intersection points can be used to figure out if there is an object between the intersection point and a light source, which thus shades the intersection point or simply to compute refractions which will be the case for our volume visualization.

To compute refractions, intersection points have to be known for arbitrary start points and directions of a ray. Basically, two ways exist to extract a surface out of a volumetric dataset in order to compute the possible intersection points.

The first is to extract a polygonal data structure from the volumetric dataset and use it for the computation of the intersection points of the ray and the possible polygons to render the surface. Compared to the volume casting of previous section, the extracted polygonal vertex array can be directly used to “compute” the intersection points by simply rendering them, as the first intersection rays all have the same starting point. This is no longer the case for refracted rays. Methods still exist for this approach to implement ray casting and ray tracing on GPU, but the current generation of those algorithms (e.g., [PGSS07]) depend on an optimized tree data structure storing the triangles. For our fluid simulation, such triangle faces first have to be extracted and then optimized for a data structure which is made compatible with an efficient
6.3. VOLUME TRACING WITH REFRACTIONS

This optimization is usually done off-line in a preprocessing step on CPU, incapable of extracting and optimizing the triangle faces in real time for our demands – especially when a high resolution dynamically changing volumetric dataset is given. So far, no known algorithm can convert the triangles, which change in every frame to the optimized tree data structure in order to compute ray-triangle intersections in real time. As an outlook, an approximation of such a data structure is used in Section 6.4 to circumvent this problem.

The second way to extract an iso surface is to tackle the problem directly by continuing to sample the volume using ray casting with a refracted or reflected ray until the desired surface intersection point is found.

This solution is also going to be used in this section, but first we review a few basic laws which are useful for our case.

6.3.1 Formulae for refractions

Figure 6.8 gives an overview of the incoming ray direction $D_{in}$, the normal $N$ at the intersection point, the outgoing ray direction after refraction $D_{out}$, the total internal reflection direction $V'$ and other vectors which are described in the according sections.

Refraction angles which are independent of the wavelength can be simply computed using Snell's law:

$$\frac{\sin(\theta_{out})}{\sin(\theta_{in})} = \frac{n_{in}}{n_{out}} = \eta$$

with $\theta$ denoting the angle between the ray direction and the surface normal. Since we usually have a direction vector of the ray, we would like to get the refracted direction of the vector. For an intersection point with the surface normal $\vec{N}$ and $\vec{D}_{in}$ aiming to the origin of the incoming

---

This ray tracing is only based on triangle intersection tests and has nothing to do with volume ray tracing or casting!
ray, the refracted ray direction $\vec{D}_{\text{out}}$ is determined (see [Len03]) by:

\[ \alpha = 1 - \eta \left( 1 - (\vec{N} \cdot \vec{D}_{\text{in}})^2 \right) \]  

\[ \vec{D}_{\text{out}} = (\eta(\vec{N} \cdot \vec{D}_{\text{in}}) - \sqrt{\alpha}) \vec{N} - \eta \vec{D}_{\text{in}} \]  

(6.1) \hspace{5cm} (6.2)

Clearly, the refraction vector is undefined if $\alpha < 0$. Then, the incoming ray is not refracted on the surface, but reflected due to the so called total internal reflection. The reflection vector is then computed by:

\[ \vec{D}_{\text{out}} = 2\vec{N}(\vec{N} \cdot \vec{L}) - \vec{L} \]  

(6.3)

For a further improvement of the visualization, rendering of the fluid surface was improved using the Fresnel factor. Using only Snells law to compute refractions and reflections would result in an unsatisfying result. When looking at the surface of a still water such as a lake inside an angle $\theta_{\text{in}}$ of about 80°, the ground of the lake as well as a partial reflection is visible in reality.

The Fresnel factor is the weight for the linear interpolation between the amount of reflected and refracted light depending on the angle of the view ray $V$, the halfway vector $H$ and the spectral distribution $f$ which is the amount of reflected light if viewing perpendicular to the surface assuming that the surface is built upon well oriented microfacets. Originally depending on the wavelength and a computationally expensive formula, the Fresnel factor $F$ for an incoming ray can be approximated with an error of less than 1% using Schlicks approximation [Sch94]. With the vectors drawn in Fig. 6.8, the specular reflection term for an incoming light with $\alpha = \vec{H} \cdot \vec{V}$ (similar to the Blinn-Phong model) is given by:

\[ F \approx f + (1 - f) \cdot \alpha^5 \]  

(6.4)

For total internal reflections on the surface used for the first intersection point, $\vec{H}$ is equal to the normal $\vec{N}$ and thus $\alpha$ equals $\vec{N} \cdot \vec{V}$.

### 6.3.2 Modifications to the volume casting shader

The shader for raycasting has to be extended in several parts:

**Multiple intersections**

To compute multiple intersections, the first intersection is handled in a special way. The coordinate of the first intersection point, the normal at this point and the reflected view vector are stored separately for lighting purposes eventually. This data about the first intersection is also used to apply the Fresnel term. For the remaining intersections, a simple for-loop is used, computing the intersections and then the reflection or refraction vector.

**Refraction/Reflection**

With the normal (computed equally as in Section 6.2.3), the refracted vector or in case of a total internal reflection, the reflected vector is computed using equations (6.2) and (6.3).

**Intersection test**

For rays searching the piercing point to enter the fluid matter, the sampling value has to be above the iso value. For rays which are already in the fluid matter, the sampling value is below the iso value in each sampling step. Both can be combined to one intersection function without
6.3. VOLUME TRACING WITH REFRACTIONS

further if-branchings by using an in/out state float number $S$. $S$ has to be set to $+1$ if the ray
start position is out of the fluid and to $-1$ if the ray start position is inside the fluid. Doing so,
the collision test is extended by two multiplications. For the current sampling value $V$, the test
for whether this value exceeds the iso value or not is modified.

The following pseudo code shows the basic differences:

<table>
<thead>
<tr>
<th>Volume casting</th>
<th>Volume tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find_next_intersection();</code></td>
<td><code>find_next_intersection();</code></td>
</tr>
<tr>
<td><code>if (V &lt;= ISO_VALUE)</code></td>
<td><code>if (V*S &lt;= ISO_VALUE*S)</code></td>
</tr>
<tr>
<td><code>{ ... intersection code ... }</code></td>
<td><code>{ ... intersection code ... }</code></td>
</tr>
<tr>
<td><code>main():</code></td>
<td><code>main():</code></td>
</tr>
<tr>
<td><code>find_next_intersection();</code></td>
<td><code>S := 1;</code></td>
</tr>
<tr>
<td></td>
<td><code>for up to I intersections:</code></td>
</tr>
<tr>
<td></td>
<td><code>compute refraction/reflection;</code></td>
</tr>
<tr>
<td></td>
<td><code>if (refraction) { S = -S; }</code></td>
</tr>
<tr>
<td></td>
<td><code>find_next_intersection();</code></td>
</tr>
</tbody>
</table>

To summarize the changes for the intersection test, the algorithmic differences are only the
modification of the intersection test and the multiplication of $S$ by $-1$ for every refraction.

Environment lookup

After refractions or total internal reflections up to a fixed amount of numbers or if the ray is
leaving the volume after at least one iso surface intersection was found, the intersection with
environment objects has to be computed. Computing those using ray tracing methods based
on polygons and a preprocessing step as mentioned in the previous section is computationally
possible but very intensive and also prohibits ray tracing of an environment which is built up
with arbitrary, moving and deformable objects. Therefore different ways to approximate the
environment should be considered.

One way is to use the frustum environment which has to be rendered prior to the current
frame. The scene is rendered as usual using a framebuffer object, storing the eye-space 3D
position and the final fragment color to a texture. For the environment intersection test,
the ray which is leaving the volume either uses an approximation (e.g. distance impostors) or
samples (binary intersection test or equidistant sampling) the eye-space position texture.

Another approximation is to use a cube map (Section 5.4). This is the easiest method, but
also the most expensive one for creation and valid only for far distant objects. For creation, the
eye position is set to the center of the volume, the view directions are aligned with the world
space axes and the up vector is perpendicular to them.

The main disadvantage of a frustum environment approximation is that obviously no envi-
ronment is valid for outgoing volume rays intersecting out of the view frustum. This is also the
main reason, why the cube map is the preferred method for this thesis.

6.3.3 Results for volume tracing with refractions

The first important task is to figure out a useful limitation for the ray tracing iterations. Therefore
two screenshots of a fluid with a frequent spatially changing fluid fraction have been created

---

8 Or the depth value to recompute the 3D position.
9 This only accounts for forward shading technique. E.g. for deferred shading, it is necessary to compose the
environment before computing refractions.
CHAPTER 6. LBM VISUALIZATION USING OPENGL

Figure 6.9: Screenshots for interpolated volume tracing with different limitations for ray tracing iterations.

(Figure 6.9). The screenshot on the left hand side with up to 4, the other one with up to 8 iterations. Without further tests, 2 intersections are clearly not enough to account for details like the total internal reflection on the bottom fluid surface after the second intersection. This visualization of the fluid after the total internal reflection is only created for at least three iterations. Since using 8 iterations almost halves the FPS for the given test scenario and the visual improvement compared to only 4 iterations is almost negligible, only up to 4 iterations are used subsequently.

Another screenshot of a breaking dam simulation is given in Figure 6.10 with a tiled color background. Up to 4 refractions or total internal reflections are computed and the Fresnel reflection is used only for the first intersection point. As we can see in the screenshot, the fluid layer after the first 2 refractions on the thin fluid layer on the right side of the screenshot is correctly approximated.

Two further screenshots in Figure 6.11 have been created to show the differences between fluid surfaces rendered with and without the Fresnel term. With activated Fresnel term, some parts of the monkey's face are reflected on the top fluid surface similar to the reflections for still lakes.

Another note should be given to the chin of the monkey. The refractions are slightly displaced to the right which is not physically correct. This is caused by the way the environment lookup is done. We only use the direction of the outgoing ray without utilization of the exit position since we assume that the cube map is created only for objects far from the center of the volumetric dataset. As a rule of thumb, we can say that the closer an object is to the fluid surface, the worse the environment approximation gets.

A standard model available in Blender

84
6.3. VOLUME TRACING WITH REFRACTIONS

Figure 6.10: Screenshot for breaking dam simulation with tiled color background and up to 4 ray tracing iterations with Fresnel-term for first iteration.

Figure 6.11: Ray tracing with a tiled color background. The left screenshot was created with the Fresnel term activated and the right one without the Fresnel term.
6.4 Marching cubes triangle extraction

This section is about a surface extraction algorithm based on marching cubes. While volume casting directly samples the volume for each fragment, the marching cubes algorithm first extracts a surface representation in form of a vertex array. Such a vertex array can then be rendered very quickly to find the first intersection point.

For this section, the semantics of a cell are changed from a cell with a fluid fraction to a cell with the fluid fraction values stored at the cell corners. This also reduces the resolution of the cube cells by one in each dimension which we can simply ignore. Since the cells at the fluid volume borders are obstacle cells, their fluid fraction is zero. When using flat textures, domain border cells together with the obstacle cell of the opposite domain size would not create any faces.

In the following section, different algorithms to extract the surface are explained. Since two of the textures are always used in the same way, they are described here:

**MC index texture (Read/Write)**

For an iso value of 0.5, the MC index is computed for every cell from the fluid fraction texture and stored to this texture to reuse it for different passes.

**MC edge index texture (Read only)**

This texture is used to store edge indices for each MC index, making this list available to the shader program via a texel fetch. The texture height is set to 256 for the number of MC indices while the width of the texture is set to 15 as the maximum number of edge indices which have to be stored for each MC index. We extend the width by one to store an additional value, counting the number of valid edge indices.

An alternative approach is to make the MC edge indices directly available to the shader via an array in the shader program. Comparing the runtime of both versions, storing the edge indices in the texture gave a significant performance improvement. To explain this with OpenCL-arguments, fewer local work groups in form of the OpenGL equivalent can be executed most probably due to the higher resource usage for the array which avoids hiding the memory latency.

6.4.1 Marching cubes with geometry shader

This method represents a straight forward implementation, using the geometry shader to emit the triangle vertices directly for every cell.

Additional normal and vertex textures are used to share their usage in different shader programs. With a maximum of 12 edges and thus 12 vertices per cell, only up to three normals and vertices have to be computed and stored for each cell since the other 9 vertices are computed by adjacent executed shaders. Thus we use a texture which is 3 times larger than the fluid fraction flat texture and use the RGB color format to store the normal vector and vertex coordinate.

Three passes are used to draw the iso surface with MC triangles:

The first pass creates the MC index texture.

The second pass computes the vertices and normals. The vertices are displaced by a linear interpolation using the fluid fractions at the two edge vertices whereas the normals are computed by central differences at the vertex position.
6.4. MARCHING CUBES TRIANGLE EXTRACTION

The third pass emits points covering the whole flat texture to the render pipeline. The vertex attributes are set to an index list, giving each vertex shader a linear incremented unique id. Without binding any vertex attributes to the primitives, no primitive is drawn at all\footnote{This should be fixed in the newer driver versions and creation as well as binding of such vertex attributes is not necessary anymore.}. At the next shading stage, the geometry shader then takes the points as input primitives. The texture coordinate of the texel within the MC flat textures is computed from the index value. After reading the MC index from the corresponding MC index texture, the shader loads the number of vertices which have to be emitted for the MC index. Within a loop iterating over a row in the MC edge index texture, the geometry shader emits the 3 vertices and its corresponding normals for each triangle within one iteration.

6.4.2 Marching cubes with vertex array

Using the geometry shader from the previous section forces us to call the geometry shader for every cell. Such geometry shaders on current generations of NVIDIA GPUs are not very fast. To circumvent the use of geometry shaders, we can either emit 15 times more vertices to cover all possible triangles which is expected to be slower than using the geometry shader or an efficient data structure in form of a vertex array is extracted out of the volumetric dataset, which can be rendered very quickly.

One efficient way to get such a vertex array is using the algorithm of Dyken et al [DZTS08]. This algorithm uses so called Histogram Pyramids. For our problem, those are used to enumerate each triangle which would be emitted for the current volume value by the MC algorithm. Based on this enumeration, the vertices and normals can be computed and stored into a vertex array, allowing an efficient rendering of the MC faces. When writing about the computation of the vertex position in this section, this usually includes the computation of the normal using central differences.

Histopyramid algorithm

We start with an explanation of the Histo Pyramid (HP) algorithm. This algorithm is not only useful for MC triangle extraction. An almost identical algorithm is also used in scientific high performance computing, better known as Z-curve. In this area, Z-curves are used to increase the adjacency of memory access when multi-variational data is stored to a memory and thus projected to a 1D field or to balance the computational tasks to a set of computing nodes.

Our task is quite similar to the previous one. Writing with OpenCL semantics, to find an enumeration which spreads the computational tasks to a number of work groups in a way that every work item does the computations for one vertex. Also the adjacency of data access for adjacent work items is increased due to the Z-curve enumeration, since the textures themselves are stored with a Z-like curve. This tremendously enhances the utilization of the texture cache. An example is given in Figure [6.12]. For each cell, the computational amount stored is the number of face vertices in each cell in the case of a fluid fraction dataset (left hand image). Then the HP algorithm uses a Z-curve based algorithm to enumerate the cells (right hand image). The output of the HP algorithm is used to let work items know which face data they should perform the computations for without running a computation for a vertex, which does not exist, or twice for a vertex.

The HP algorithm is further explained in the context of the extraction of a fluid surface.

For each cell, the number of vertices per cell is stored to the base texture of the HP, which acts as the input for the algorithm. This number can be directly read out of the MC edge
CHAPTER 6. LBM VISUALIZATION USING OPENGL

Figure 6.12: Left image: The number of faces for each cell is stored as a preprocessing step. Right image: A linearly increasing enumeration is created to spread the computational task equally to all computing nodes.

index texture when the MC index is known. The output of the HP algorithm is a mipmap texture, allowing a fragment shader to determine the 3D cell index and the vertex at the cell’s edge belonging to a linearly incremented ID where no ID refers to an unused vertex. Then the shader programs are able to compute the vertex position and the normal for a linearly incremented ID.

Basically, there are two passes in the HP algorithm:

The first one is to count all vertices by a bottom-up process (Left hand image in Figure 6.13). The layers created in this step are reused by the next pass.

The second pass is a top-down process, storing the “path” to the cell for each ID when going top-down (Right hand image in Figure 6.13).

The layers are stored using mipmaps, further denoted by UP mipmap for the bottom-up steps and DOWN mipmap when writing to the top-down layers. Using mipmaps instead of single textures becomes very important when the shader has to figure out, which cell and cell vertex it has to compute the vertex position and normal. Only a limited number of textures are available to a shader. To write to each mipmap level, a separate framebuffer object is created for each layer.

The HP algorithm starts with the base texture, stored at layer 0 of the UP mipmap texture with a reduction operation. To gain maximum performance due to the massive parallel architecture, a hierarchical approach is used by shrinking the texture size to the half size in each dimension for the next layer above the current one. The shader program for a layer then sums up the values in the color channels of the four hierarchically underlying pixels. Finally, there is a single RGBA texel value at the top of the mipmap pyramid (right side of Fig. 6.14), which stores the number of all MC vertices ($R + G + B + A$).

As a preprocessing step for the second pass, the CPU reads the $(R, G, B, A)$ value from the topmost layer of the UP mipmap for later purposes and writes back $(0, R, R + G, R + G + B)$ to the topmost texture of the DOWN mipmap as the first step of the next pass. This pass uses both mipmaps to compute the index. The DOWN mipmap is used as the base displacement and the UP mipmap as the “fine” displacement relative to the base displacement. An example is given in the yellow box in Fig. 6.13.

For the shader, which computes the vertex and normal, the corresponding MC cell is found by going top-down in the DOWN mipmap, following the texels belonging to the valid area for the index value. The short code for this top-down process is given in Algorithm 2.

Two versions of the HP algorithm have been implemented. One version uses only the R channel and the other version stores the fluid fraction of $2 \times 2 \times 1$ cells into the RGBA components.
6.4. MARCHING CUBES TRIANGLE EXTRACTION

Figure 6.13: Up and down process to create data for HP

Figure 6.14: Reduction step for HPs
CHAPTER 6. LBM VISUALIZATION USING OPENGL

as originally suggested in [DZTS08]. A small speedup could be detected for the RGBA version. A further advantage of using RGBA, instead of the red color channel only, is that storing the fluid fraction in RGBA texels avoids a special handling of texels which are “out of texture”. This is the case when the second top most layer has a size of $2 \times 1$ for example.

To avoid a copy operation to initialize the bottom layer of the UP mipmap, the shader program for the bottom-up process can be reused, since the bottom layer of the mipmap texture is accessed in the shader equally like a usual 2D texture.

Algorithm 2 Computation of the MC cell with HP DOWN mipmap texture

**INPUT:**

// Unique linear id in [0; max_vertices]
int linear_id;

**OUTPUT:**

// Position in flat texture of the desired cell
ivec2 flat_pos;

// Start texel position for top layer
flat_pos = ivec2(0, 0);
for (int i = MIPMAPLAYERS-1; i >= 1; i--)
{
    uvec4 data = texelFetch(histopyramid_texture, flat_pos, i);
    flat_pos <<= 1; // Shift position for next layer
    flat_pos.y += int(linear_id >= data.b); // Fix y coordinate
    flat_pos.x += int((linear_id >= data.a) || (linear_id < data.b && linear_id >= data.g));
}

Creating a vertex array with HP

After knowing how the HP algorithm works, it is now assembled with other passes to create a vertex array. Four passes are necessary to gain the vertex array.

The first pass creates the MC index texture and additionally stores the number of triangles for each cell to a texture which is used as the basement texture for the UP mipmap. Since the data is stored for $2 \times 2$ blocks, each fragment shader handles 4 cells per execution, which is also the case for the upcoming passes.

Then the HP implementation is used as the second pass to build up the DOWN mipmap.

The third pass is independent from the second pass and is equal to the second pass for the implementation of the previous section, which utilized the geometry shader. The vertex positions and normals are computed and stored to an additional texture.

The fourth pass stores the vertices and their according normals to the desired vertex and normal array. To identify a fragment and its vertex, each shader is associated with a unique ID which starts at 0 for every HP usage and has to be linearly incremented for the next shader execution. To achieve this, different solutions exist for the different implementations, which are explained in more detail below.

When using a fragment shader to compute the vertex and normal, the current fragment coordinate can be used in every fragment program to determine the ID.

When the vertex shader is used to output the desired data, an index buffer has to be created and bound as a vertex attribute, storing the values from 0 up to the maximum number
of vertices. Depending on the implementations which are explained in more detail below, the resulting vertices and normals are stored in a buffer or a texture. But for fluid simulations, we do not know how many vertices and normals have to be stored in each frame. Initializing the buffer to store the indices and the texture or buffer to store the vertex and normals to a size to fit all possible MC combinations would clearly waste memory. Therefore this implementation makes use of an adaptive expansion of the buffers by chunks of a fixed size: In this implementation, the size of one chunk is set to

\[
\left( \frac{1024^2}{\text{texture\_width}} + 1 \right) \cdot \text{texture\_width}
\]

and the vertex and normal textures or buffers are initialized with a size of one chunk. Whenever modifying the vertex and normal buffers, the index buffer is resized appropriately and its values are also updated. Whenever more indices are needed, the vertex and normal buffers are expanded by one chunk.

To store the vertices and normals, three different implementations have been tested. The test environment was to render a volumetric sphere stored in a domain of size \(128^3\). The performance values are given in the format \([xxx \text{ FPS (relative performance)}]\).

1. **Vertices and normals from textures:**
   This version does not directly create a vertex array list, but stores the vertices and normals into a texture. Therefore the computation can be directly done in the fragment shader without utilization of the index buffer. The rendering of triangle primitives is then done by looking up the vertex coordinates and normals in the aforementioned texture within the vertex program. This is also a major penalty, since existing “standard” shaders have to be modified when a mesh should be rendered with the data output from this implementation.
   Performance: 170 FPS (100%)

2. **Transform feedbacks:**
   Compared to the previous method, transform feedbacks are one way to really gain a pure vertex array. With activated transform feedback, the output of the vertex shaders is stored to a buffer which can be reused as a vertex array. Since the output is caught before an utilization of the fragment shader, the index buffer is used to get the unique ID. Compared to the previous implementation, one of the major advantages is that such an output in form of a vertex array fits better to existing shaders. Also the rendering itself is slightly faster, while the extraction with transform feedbacks and the rendering is about 34 percent slower compared to previous method.
   Performance: 113 FPS (66%)

3. **Copy texture to buffer:**
   This implementation combines the best parts of the previous ones. The vertices and normals are computed via the execution of frequent fragment shaders and stored to a vertex/normal texture which is bound to a FBO. Then a copy operation is done using glReadPixels to copy the vertex data to a vertex array buffer. Copying directly from textures without usage of FBO is also possible, but should be avoided since this is about 10 times slower. When comparing the performance, there is a cost of roughly 5% for the copy operation.
   Performance: 162 FPS (95%)

While the rendering itself is slightly faster when using vertex and normal arrays in buffers, the last method ‘copy texture to buffer’ was used further since the vertices have to be rendered multiple times and unmodified standard shaders can be used for the visualization of the fluid surface.
6.4.3 Marching cubes with vertex array and GL_INTEROPS

This subsection explains a possible way to circumvent the problems of the OpenCL GL_INTEROPS copy operation mentioned in Section 6.1.

Figure 6.15: Converting the raw data (left texture) to the flat texture format (right texture) for a $64^3$ falling drop simulation.

Since current NVIDIA drivers do not support copy operations from OpenCL memory to RED 3D textures but to RGBA 2D textures, and fast write operations to flat 2D textures are possible, we can circumvent the GPU↔CPU bottleneck for large datasets.

As a workaround, the OpenCL data is copied directly from a buffer to an RGBA 2D texture which obviously would not result in the usual flat texture layout. For a fluid drop over a water surface, the resulting texture would look like the left image of Fig. 6.15.

To get a texture with the usual flat texture layout and with $2\times2\times1$ cells stored in one RGBA color, a shader is used to convert this raw data to a 2D flat texture (right image of Fig. 6.15).

Since the fluid simulations for our test cases only run on datasets which are less or equal than $64^3$ cells, the fluid fraction data is also below 1MB. This could be one reason why the performance was not measurably increased (and also not decreased) when using this method to access the fluid fraction values within OpenGL shaders.

6.4.4 Results for marching cubes

Since the rendered marching cubes are all the same, even if rendered with the ray casting method, an appropriate screenshot can be found in the right image of Figure 6.5 in the section about ray casting.

We start by extending the FPS benchmark for specific ray casting methods from Table 6.1 with the values for the different implementations explained in this section and by rendering them with the same parameters. Those values can be found in Table 6.2 and are visualized in Figure 6.16. As we can see, the marching cubes algorithm seems to be slower for most of the test cases. But this is not the case in general for two reasons:

A vertex list with the marching cubes faces can be rendered much faster to represent the surface, once it is extracted. This can bring advantages when the object has to be rendered multiple times for each frame.
6.5 MARCHING CUBES WITH EYE-SPACE REFRACTIONS

<table>
<thead>
<tr>
<th>Implementation / Domain Size</th>
<th>16³</th>
<th>32³</th>
<th>64³</th>
<th>128³</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation</td>
<td>1941</td>
<td>1351</td>
<td>819</td>
<td>491</td>
</tr>
<tr>
<td>Cube steps</td>
<td>1343</td>
<td>832</td>
<td>482</td>
<td>261</td>
</tr>
<tr>
<td>MCs (raycasted)</td>
<td>196</td>
<td>107</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>MCs (geometry shader)</td>
<td>3550</td>
<td>622</td>
<td>138</td>
<td>23</td>
</tr>
<tr>
<td>MCs (vertex array)</td>
<td>1440</td>
<td>1200</td>
<td>680</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 6.2: FPS results for volume casting and marching cubes methods

Figure 6.16: FPS for different visualization methods

Another advantage for face extraction methods exists at higher resolutions. Compared to the volume tracing, which strongly depends on the resolution for which the volume is rendered, using vertex array methods is not so dependent on the screen resolution since the original purpose of GPUs was to render such faces. The extraction of the vertex array is independent of the screen resolution and rendering is almost achieved for free on GPUs, compared to the extraction phase. To give an example benchmark, a sphere approximated in a volumetric dataset with a resolution of 64³ is rendered to a viewport of size 1600 × 1040. The top and bottom surface of the sphere “.touches” the window borders. 140 FPS are achieved when the interpolated ray casting method is used, while the MC vertex array method renders the sphere with about 570 FPS.

6.5 Marching cubes with eye-space refractions

The vertex array extracted in the previous section now becomes the input data for the tracing algorithm in this section. As already mentioned in Section 6.3, using the vertex array directly to compute the ray-triangle intersections would break our demand of a realtime application. Therefore the refractions are computed based on an approximation of the fluid surface in eye-space with depth peeling or front/back face extraction (see Sections 5.6 and 5.6).

6.5.1 Previous methods

Plenty of methods still exist, which approximate refractions in eye-space. However, these methods cannot be used without modifications, since they are simply not directly applicable to
6.5.2 Surface approximation

We start by discussing two ways to approximate the fluid surface.

The first method is depth peeling which can be used to extract different layers of the fluid. This has several disadvantages. Involving many layers to compute intersections increases the bandwidth demand since more voxel data has to be loaded. This also increases the computational time due to more intersection tests. A version using more than two depth peeling textures for eye-space refractions was also developed, iterating through the texture array layers for every sampling point until an intersection is found. Strong artifacts appeared, produced by voxels which are not object-space adjacent to the previously sampled voxels. To avoid such ‘wrong’ intersection computations, voxels which are not within a specific distance of the ray have been ignored, suppressing most of the artifacts, but the results are still disappointing.

Apart from discontinuities at the object silhouette in a layer approximation, discontinuities can also appear in the inner texels for concave objects, making it hard to compute an intersection point – especially when a linear interpolation between the current and previous sampling point is done to improve the position of the intersection point. Therefore only a surface extraction with two layers was considered further with the additional assumption, that the first intersection point always lies on the first peeling texture, while the second intersection point lies on the second peeling texture.

The front/back face extraction produces a better visual appearance than the depth peeling with two layers. One obvious example can be imagined, when looking in one line through the ears of the Standford bunny model. Using depth peeling, the front ear would cause a dropping of all approximation data of the ear right behind the other ear, resulting in a disappearance of the more distant ear. With the front/back face extraction, the far distant ear will be still visible and thus the visual appearance is improved in this case. However, this also has a drawback: a gas bubble within a fluid would be omitted, while the bubble would be at least visible in some way with the depth peeling.

For the rest of this chapter, only front/back face extraction is considered since the approximations are better for most of our test cases.

http://www.cs.uiowa.edu/~cwyman/pubs.html
Regardless which surface approximation is chosen, the 3D viewspace coordinate of the fragment is stored to the texture. Also an implementation using the depth map to recompute the 3D viewspace position has been implemented, which gave no significant performance differences for our default GPU.

Using the perspective eye-space for layer extraction also has a big advantage compared to an orthogonal rendering. It avoids the computation of the first intersection point, since a texel fetch in the first layer using the current fragment coordinate as a texture coordinate fetches the first intersection point.

To allow z-depth culling in the post processing stage of the fragment when rendering the objects with refractions, the depth value is stored to the alpha channel in the surface approximation pass for the front surface to set the depth value to its accurate value when rendering the refractions in the fragment shader. An unused alternative approach would be to compute the depth value in the shader by multiplying the viewspace coordinates of the fragment with the projection matrix and making it inhomogeneous. Then the depth value of the current fragment is stored in the $z$ component.

### 6.5.3 Intersection points in texture space

![Diagram](image)

Figure 6.17: Computing whether the next sampling point is used to compute the final intersection point on a perspective rendered voxel texture.

The method to find the intersection point in this thesis is similar to displacement mapping [Don05]. Working without preprocessed data, we sample the 2D texture directly. Compared to

---

13Multiplying the inverse of the projection matrix with a vector which is built up with the fragment coordinate as $x$ and $y$ component and the depth as the $z$ component

14The default graphics card is a NVIDIA GTX 285, while on a GeForce GT 130M using the depth texture was faster.
CHAPTER 6. LBM VISUALIZATION USING OPENGL

the computation of the intersection point in relief mapping [POC05], the fixed sampling step width is also used, but the sampling is made in the perspective rendered 2D texture space. Since we do not know the frequency of the fluid surfaces approximated in the front and back textures, this forces us to set a small sampling step size due to the possible occurrence of small water drops from the simulation dataset. Because the voxel textures are extracted in view space, special account has to be taken for the computation of the collision point. A sketch of the problem is given in Figure 6.17.

Sampling in texture space

The collision point is detected by sampling the voxelization texture with equidistant step sizes similar to the volume casting.

Alternative approaches sample with an equidistant step size in 3D and projecting the 3D point to a 2D texture coordinate using the projection matrix. Sampling in 3D with a fixed step size, the accurate computation of the sampling point can be guaranteed only if the step size is set to a low value which creates more sampling points than sampling the 2D texture directly.

To simplify the upcoming argument, it is assumed that the surface is approximated with a continuously differentiable function, sufficiently smooth over each cell. Then according to the Nyquist-Shannon theorem, a texture with unknown maximum frequency has to be sampled with the half texel size to avoid loss of information. Clearly, the step size can usually be increased, since most surfaces show linear behavior over most adjacent texels and the Nyquist-Shannon theorem only gives a maximum limit for different kinds of textures. However we still set our step size to 0.5 for an accurate computation of the intersection at the silhouette of the object, to allow a fair comparison with the volume tracing method and to avoid omitting the rendering of small bubbles or drops in the fluid.

When writing about the sampling of fluid bubbles or drops in the context of 2D texture sampling, we have to be careful with the meaning of the maximum frequency of information. Since the volume is usually rendered in such a way that plenty of fragment pixels are rendered for one cell, the maximum frequency is lower than would be the case if the volume is directly sampled – assuming that the signal is generated only by the fluid fraction in the cell. We simply ignore this because the volume can be also placed far from the viewer or scaled down, increasing the maximum frequency. Also an implementation has been tested to use an eye-space FBO with a lower resolution for the surface approximation, which produced artifacts at the silhouette of the object due to the missing information.

We start with a given 3D position \( P \) and a direction \( D \) giving us the ray equation \( R(t) = P + t \cdot D \). There are two tasks which have to be solved:

Search direction in texture space

The first task in the fragment shader program is to compute the search direction in the texture space, which cannot be gained directly.

Using matrices without perspective distortions, the inverse transpose of the matrix is usually used to apply rigid body matrix transformations on normals. Since we project the direction of the ray with a perspective distortion, the projected ray direction also depends on the position.

For a better understanding of the problem, we imagine a simple example. We project a vector with its components \((0, 0, -1)\) once with its start location at \((0, 0, -5)\) and again with \((1, 0, -5)\). Clearly, the end and start point of the projected vector coincide in one point for the first start location, whereas the projected vector for the second example is again projected to

\[15\] It is hard to compare both visualizations due to the different advantages and disadvantages.
6.5. MARCHING CUBES WITH EYE-SPACE REFRACTIONS

a vector with a length greater than 0. Thus the projected vector depends on both the vector itself and its position.

As a simple and truly straightforward solution, the start and endpoint of the vector are separately projected using the projection matrix \texttt{proj\_matrix}. After dividing by the inhomogeneous component, the projected points are used to compute the direction vector in texture space.

To sample the texture with a fixed step size, relative to the size of one texel within the texture, further processing has to be done. After the projection matrix was applied, the resulting \(x-y\) components lie within the normalized range \([-1;1]\) if they lie within the rendering frustum. For screen sizes where the width differs from the height, the 2D texture direction has to be multiplied with the texture size and normalized afterwards. Then the result is divided again by the texture size which gives us the sampling distance for the size of one texel.

\begin{algorithm}
\textbf{Algorithm 3} Conversion of view space direction vector to texture sampling vector.
\begin{itemize}
\item [\textbf{Input variables:}]\begin{itemize}
\item \texttt{mat4 proj\_matrix}; // Projection matrix
\item \texttt{vec3 pos}; // Viewspace 3D position to start search from
\item \texttt{vec3 dir}; // Viewspace 3D search direction
\item \texttt{vec2 viewport\_size}; // Size of viewport and texture
\end{itemize}
\end{itemize}
\begin{itemize}
\item [\textbf{Program sketch:}]\begin{itemize}
\item \texttt{vec4 pl = proj\_matrix * vec4(pos, 1);} \quad \texttt{p1.xy /= p1.ww}; \quad \texttt{// Convert to inhomogenous coordinates}
\item \texttt{vec4 p2 = proj\_matrix * vec4(pos+normalize(dir), 1);} \quad \texttt{p2.xy /= p2.ww}; \quad \texttt{// Convert to inhomogenous coordinates}
\item \texttt{vec2 tex\_iter\_dir = (normalize((p2.xy - p1.xy) * viewport\_size) / viewport\_size;}
\end{itemize}
\end{itemize}
\end{algorithm}

Detecting intersections

The second task is to detect possible intersections as shown in Figure 6.17. A collision is assumed, when the last voxel (V5) was “below” the ray \(R\) and the current voxel (V4) is above \(R\). Since the voxels are sorted in \(z\) direction (because of depth peeling or front/back face extraction), there cannot be an intersection for the first step after an intersection and thus only a test has to be done whether the current voxel is in a positive direction of the ray \(R\).

A straightforward implementation would be to test whether the current \(X\) and \(Y\) values of the voxel are greater than the intersection of the search ray with the plane at the \(Z\) position.

Using this method only would result in unexpected behavior since we work with projective coordinates. E.g. an intersection could be found for a negative parameter of \(R\). Test implementation without further handling of more special cases which has to implemented for accurate refractions was also much slower than the method described next which overcomes many problems considered so far.

Two important tests have to be performed for a voxel. Firstly, the voxel has to lie within the direction of the search ray (relatively to the start position) – only intersection for a positive \(t\) is allowed. Secondly, a fast test should be possible, to identify whether the voxel lies “above” the search ray or not.

Therefore the voxels are transformed to another basis by setting up the \textit{ray intersection}
space as shown in Figure 6.18. The desired basis has to be built up with the direction ray $D$ as one axis. Another useful axis $U$ is found by the vector perpendicular to $D$ and the vector $P'$ from the ray start position $P$ to the eye. $U$ is perpendicular to the plane created by the texture sampling ray and the view space sampling ray. The third and last orthogonal component is computed by the vector perpendicular to $D$ and $U$. Normalizing all axes and using the result to build up an orthogonal matrix, we can invert the matrix by using its transpose to get the desired matrix $B$ transferring voxels from the view space to the ray intersection space:

$$
U := D \times (D \times P)
$$

$$
U := \frac{U}{|U|}
$$

$$
B := \begin{pmatrix}
D_x & D_y & D_z \\
U_x & U_y & U_z 
\end{pmatrix}
$$

The third row of the matrix $B$ was dropped since it has no use. Voxels read from the peeling texture along the ray and projected by $B$ would have a third component which would be close to 0. It is important to note, that setting up $B$ has to be done once after each refraction.

The intersection test itself is done in ray intersection space by testing the first and second component of the projected voxel after applying the matrix $B$. If both values are above 0, the ray intersects.

Now we are able to implement the algorithm by iterating over the texture until an intersection was detected or its borders are reached. To avoid checks on whether the current sampling point is still within the texture, we compute the distance to the borders and use this limitation within the iteration loop to avoid ‘out of texture’ tests within the loop. After an intersection was found, the intersection point is enhanced by linear interpolation between the current voxel and the last voxel, finding its zero crossing. Again, this method is quite similar to the method used in ray casting.

### 6.5.4 Results for marching cubes with refractions in eye-space

Similar to the previous sections, the results are split up into visualization and performance aspects.
6.5. MARCHING CUBES WITH EYE-SPACE REFRACTIONS

Eye space refractions with front and back faces approximation

Interpolated volume tracing with 4 iterations

Figure 6.19: Screenshots for eye-space refractions and interpolated volume tracing

Visualization

The results for the implementation are presented with screenshots of different simulations and one or more particular time steps.

- Since the volume tracing is based on the full dataset while the front/back face extraction only uses an approximation of the fluid surface, those two approaches are compared. Two screenshots are given in Figure 6.19 visualizing a fluid constellation equal to that of Figure 6.9 from Section 6.3.3. As we can see on the left image, which was rendered with the eye-space approximation, this approximation is not sufficient to show details of the fluid surface, making it looking unusual. The domain corner at the most distant bottom side is clearly visible and total reflections are also missing. This unusual look is much more obvious when the fluid is moving in real time.

- Another example given in Figure 6.20 also shows the problems of the front/back face extraction for a simple falling drop simulation. The gas in the middle of the fluid is not visible with the eye-space refractions since this area cannot be stored in either the front or back face texture.

- Apart from the visualization of fluid fractions based on volumetric datasets, the eye-space refraction technique can also be used for every mesh-like dataset as it is done usually in other implementations [DW07]. To give an example, the Stanford bunny with reduced polygons is used together with the front/back face approximated with eye-space refraction. A screenshot of the scene which is rendered with 215 FPS and a viewport resolution of 800 × 600 is given in Figure 6.21.
CHAPTER 6. LBM VISUALIZATION USING OPENGL

Eye space refractions with front and back face approximation

Interpolated volume tracing with 4 iterations

Figure 6.20: Screenshots for falling drop simulation with a resolution of 64³ with refractions based on Front/Back face extraction and volume tracing

Figure 6.21: Eye-space refractions with front/back face approximation using the Stanford bunny model
Performance

Also a short performance benchmark shall be given here for a $64^3$ domain size and a falling drop simulation similar to that in Figure 6.20. Three different implementations for the eye-space refractions have been implemented, each one with a different step size. Increasing this step size only slightly changes the appearance, which is also the reason why no screenshots are given here for the different cases. A few benchmark values are given in the following table:

<table>
<thead>
<tr>
<th>Domain size: $64^3$</th>
<th>Size of viewport</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>800 × 600</td>
</tr>
<tr>
<td>Interpolated volume tracing refractions</td>
<td>132</td>
</tr>
<tr>
<td>Eye-space refractions with step size of 1.0</td>
<td>144</td>
</tr>
<tr>
<td>Eye-space refractions with step size of 2.0</td>
<td>158</td>
</tr>
<tr>
<td>Eye-space refractions with step size of 10.0</td>
<td>160</td>
</tr>
</tbody>
</table>

6.6 Photonmapping in light space

For this diploma thesis, the task was not only to render the scene with direct lighting effects, shadows and refractions, but also an approximation for so called caustics should be implemented. Caustics are areas on a surface which are highly illuminated compared to the surrounding areas. An every day example can be seen when a glass is standing on a table, while some of the light rays, which pass through the glass gather at the other side on the tables surface. For our case, the glass will be the fluid fraction data.

6.6.1 Implementation

To allow a realtime rendering, a few simplifications have to be done:

*Only refractions are considered* since those are responsible mainly for the caustics created by fluids. Photons which undergo a *total internal reflection* are ignored as well as photons which are absorbed or scattered in the fluid matter. The total internal reflections are far fewer than the refractions for most of the test cases and also take much more computation time, since multiple textures have to be sampled for each intersection test.

For the surface shaded by the fluid, shadow mapping can be used for point lights when the fluid also emits a shadow. The objects within such a shadow map are only illuminated by the ambient light and the caustics.

We have already implemented the basic parts of an appropriate approximation for the photon mapping in the previous section when refractions have been computed in eye-space textures.

The idea is to render the scene from the light position, looking at the fluid volume and the diffuse surface which the photons can hit. Instead of doing a lookup in the environment map at the end of the refraction fragment shader, a further intersection search is done to find the position where a photon hits the surface of an object. Then the position of the photon and its further attributes are stored to the photon map texture.

Since it is desirable to modify the position of the light source the light space frustum should be adopted to cover only the parts of the scene which are used for photon mapping with the side constraint of keeping the light space frustum small. To achieve this the idea of an expandable light frustum is used. We further distinguish between the *projection frustum* which is used to project the corner points of objects to determine the parameters for the *expanded frustum*. All previously projected corner points should then lie within the ‘expanded frustum’.
CHAPTER 6. LBM VISUALIZATION USING OPENGL

Figure 6.22: The expandable light space box is drawn in green for two different cases. Left: Expandable light space box for a light source right above the fluid domain. Right: Expandable light space box with a light source far above the fluid domain.

- In the first step, the ‘projection frustum’ is initialized with the view direction aiming at the center of the fluid volume. If the angle between the y-axis and the vector from the origin to the light position is less than 0.01, the up vector of the frustum is set to $(1, 0, 0)^T$, otherwise to $(0, 1, 0)^T$. To create an expanded frustum covering only the important objects which are relevant for the photon mapping, the distances to the new expanded frustum borders at the near plane are initialized with $(\text{left} = \infty, \text{right} = -\infty, \text{top} = -\infty, \text{bottom} = \infty)$. The parameters for the near and far plane are initialized respectively with $\infty$ and 0. The near and far planes are also adapted dynamically to increase the accuracy of the light space depth map.

- In the expansion steps, the corners of the bounding boxes of the diffuse objects are projected with the ‘projection frustum’ matrix. For each projected corner point the parameters of the ‘expanded frustum’ are modified to render the corner point within the viewport when using the ‘expanded frustum’ matrix.

This technique has the advantage of using a similar sampling rate for the volume and the diffuse objects independent of the light source position. Two examples for an expandable light space box are given in Figure 6.22.

In a further rendering pass, the photon map is used to draw photon splats to a caustic map texture. In the simplest form, splats are points with a radius set by the fragment shader with the alpha value fading out to the border to account for a smoothness. Such splats are rendered to approximate the lighting of a surface area due to a photon hitting this surface. The splat radius and its color and thus brightness have to be set according to the attributes like the photons energy, the angle between the incident angle and the surface normal.

The caustic map texture works similarly to the depth map and thus the caustics are rendered by modifying the parameter for the incoming diffuse light by the color read from the caustic map.

Three passes are used for the photon mapping:

**Pass 1 - computing the caustic photons:**
The front and back faces, with their normal are extracted in light space, as a sufficient approximation of the refractive surface. Then the diffuse surfaces with their normals are extracted in light space. See the left of Figure 6.23 for an example.
6.6. PHOTONMAPPING IN LIGHT SPACE

Figure 6.23: Left: Eye space refractive Front/Back faces and diffuse surfaces. Viewspace voxel map (top row), normal map (middle row), depth map (bottom row).
Right: Caustic map

From the position of light, photons are emitted to compute their refractions with the front and back textures. After a refraction with the front face was computed, refraction with the back face is computed regardless of further refractions with the front faces. The final tracing step has to find the intersection point of the photon within the texture storing the diffuse surfaces. If no such intersection point was found, the photon is discarded by setting its position to be out of the frustum so that the photon is getting clipped.

The position on the diffuse texture \((u, v)\), the angle between the photon direction and the surface normal \(\angle(\vec{D}, \vec{N})\) and the length of the path the photon took to hit the diffuse surface \(d\) are stored to the framebuffers \( RGBA \) components.

Pass 2 - creating the caustic map:
Since the caustic map is used in a similar way to the depth map, the caustic map is also created with the same view and projection matrix. An example for a caustic map is given on the right of Figure 6.23. To simulate the photons, they are rendered as splats and blended to the caustic map.

One problem arises when a few thousand photons need to be rendered. When increasing the amount of photons to increase the accuracy of the photon mapping, each photon transports less energy for reasons of energy conservation. Particularly for a large number of emitted photons, storing photons on a texture with internal byte format would cause problems since the attenuation of the photon may not be below \(\frac{0.5}{256}\). Therefore the texture to store the caustic map uses floating point numbers for the color representation\(^\text{16}\). Another advantage is, that the blending for many photons is more accurate.

Pass 3 - using the caustic map:
While rendering the objects, lookups in the shadow and caustic map have to be done to get an

\(^{16}\) The internal texture format GL_RGBA32F was used in this thesis
CHAPTER 6. LBM VISUALIZATION USING OPENGL

approximation of the local illumination. The front or back face depth map can be used as a shadow map for the diffuse surfaces. The texture coordinate used for the lookup in the shadow map is the same as for the lookup in the caustic map.

6.6.2 Results

Three screenshots are given in this section to show the capabilities of the previously explained modified version of photon mapping.

The first two screenshots in Figure 6.24 show frames of a falling drop simulation. The shadow emitted by the fluid was approximated with the shadow map, which was already computed in the pass to extract the front/back faces of the fluid. Also the shadow map for the diffuse objects, which was used to cast the shadow of the monkey head, was already computed in the pass for the surface extraction of the diffuse objects.

A third screenshot is given in Figure 6.25. In this screenshot, we are interested in the area on the chessboard, which is strongly lit. The material parameters at this caustic are different for the table, the white and black fields and the border of the chessboard.
Figure 6.25: Screenshot with photon mapping and main caustics at an edge of the chessboard
This chapter gives two example simulations on the visualization combined with the simulation. Specific results for only the simulation or the visualization are available in the last parts of the appropriate chapters.

7.1 Interactive simulation and visualization with modification of the gravity vector

Figure 7.1 shows some screenshots of the first simulation. Firstly, the parameters for the simulation are explained, then the way the simulation becomes interactive.

- For real time purposes a large velocity could lead to an unstable simulation (see Section 4.1.7). This instability originates with a velocity $v$, which exceeds the stability limitations. One limitation is that this velocity has to be far below $\frac{1}{\sqrt{3}}$. To overcome this, in order to allow larger time steps, the maximum lattice velocity is rounded to 0.25 after computing the velocity with Equation (4.2). This value has been determined empirically, since a value close to $\frac{1}{\sqrt{3}}$ still leads to unstable simulations.

- Since the maximum possible velocity is restricted, the gravitational force does not lead to further unstable behavior. Therefore the upper limit of the lattice gravity is set to the relatively large value of 1 to avoid restrictions on this parameter (see Section 4.1.7 for more details about the restrictions of the gravity).

- There is still the problem of time steps that are too small for our purposes when the viscosity is set to the viscosity of water. Therefore the viscosity of the fluid is set to the higher value 0.00013.

- The $x$-length of the simulation domain is set to 0.3 m.

- The default acceleration due to gravity for earth is used: $9.81 \text{ m/s}^2$.

- The domain resolution is set to $64 \times 64 \times 32$.

With activated volume tracing refractions and photon mapping with front/back face approximations 18 FPS are reached on average. About 10 simulation timesteps are done per frame.

The gravitation vector is used to give the user the feeling of rotating the cube. A physically more accurate model would have to compute the momentum caused by the moving domain borders to the fluid. This was avoided as it would involve many more operations. For this implementation the gravitation vector is updated always to aim at the bottom of the drawn scene whenever the cube is rotated. For example when rotating the cube, the vector on which the gravitation force acts has to be rotated in the opposite direction.
Figure 7.1: Some screenshots for interactive simulation and visualization with modification of the gravity vector
7.2 Oil like simulation and visualization

Figure 7.2 shows another test scenario with a domain resolution of $128 \times 64 \times 32$ and a length of the domain along the $x$ axis of 3 m. It differs from the previous simulation due to the increased viscosity and the larger domain size. This enhances the framerate to 40 FPS on average.

For the visualization the transparency was deactivated and the diffuse color of the fluid was set to black.

7.3 Website & further information

Since there is no way to show interactivity in form of a written thesis, videos for this thesis, the source code and further information are available at [Scha]. Also a small flipbook has been printed on the edges of this thesis.
CHAPTER 7. FINAL RESULTS

Figure 7.2: Screenshots for a simulation with high viscosity and a domain resolution of $128 \times 64 \times 32$
The secret to creativity is knowing how to hide your sources.

Albert Einstein (1879 - 1955)

Bibliography

[apa] Apache license, version 2.0. 
  http://www.apache.org/licenses/LICENSE-2.0.txt


[BLE] Blender website. 
  http://www.blender.org/


[COC] Cuda occupancy calculator. 
  http://developer.download.nvidia.com/compute/cuda/3_0/sdk/docs/ CUDA_Occupancy_calculator.xls

[cud] Cuda zone. 
  http://www.nvidia.com/object/cuda_home.html

  http://wiki.github.com/laanwj/decuda/


BIBLIOGRAPHY


[MHP] Mesa home page. 
http://www.mesa3d.org/

http://www.mpi-forum.org/.

[MSY02] Renwei Mei, Wei Shyy, and Dazhi Yu. Lattice boltzmann method for 3-d flows with curved boundary, 2002.


[NOU] nouveau wiki. 

[NVC] Nv compiler options. 
http://developer.download.nvidia.com/compute/cuda/3.0/toolkit/docs/opengl_extensions/cl_nv_compiler_options.txt

[NVD] Nvidia developer web site - tools, presentations, whitepapers. 
http://developer.nvidia.com/


http://www.nvidia.com/content/CUDA-pxt_1.4.pdf.


[OGL] Opengl - the industry standard for high performance graphics. 
http://www.opengl.org/

http://www.openmp.org/


BIBLIOGRAPHY

[PS3] Opengl news comments - sony confirms playstation 3 to use opengl es.
http://www.opengl.org/news/comments/sony_confirms_playstation_3_to_use_opengl_es/


[Scha] Martin Schreiber. Homepage of this diplomathesis.
http://www.martin-schreiber.info/diplomathesis.html

http://www.martin-schreiber.info/lattice_boltzmann_opengl.html


[SDL] Simple directmedia layer.
http://www.libsdl.org/

[SPZ+09] Jessica Schmidt, Cécile Piret, Nan Zhang, Benjamin J. Kadlec, David A. Yuen,
Yingchun Liu, Grady Barrett Wright, and Erik O. D. Sevré. Modeling of tsunami
waves and atmospheric swirling flows with graphics processing unit (gpu) and
radial basis functions (rbf). 2009.

[Str09] Severin Strobl. GPU-Based Rigid Body Dynamics. Diplomarbeit, Friedrich-
http://www10.informatik.uni-erlangen.de/Publications/Theses/2009/
Strobl_DA09.pdf


[Thu03] Nils Thuerey. A lattice boltzmann method for single-phase free surface flows in

[Thu07] N. Thuerey. Physically based animation of free surface flows with the lattice

[Tor] Gabriel Torres. Seeking the best performance per watt for folding@home.
http://www.hardwaresecrets.com/article/640/1

[Tre02] J. Treibig. Simulation von gas-feststo-mehrphasensystemen mit dem lattice boltz-
mann verfahren, 2002.

Eurographics/ACM SIGGRAPH Symposium on Computer Animation, pages 157–
166, Jun 2006.

[UNI] Unigine - 3d engine of virtual worlds (games and virtual reality systems).
http://unigine.com/

[WG00] Dieter A. Wolz-Gladrow. Lattice-Gas Cellular Automata and Lattice Boltzmann