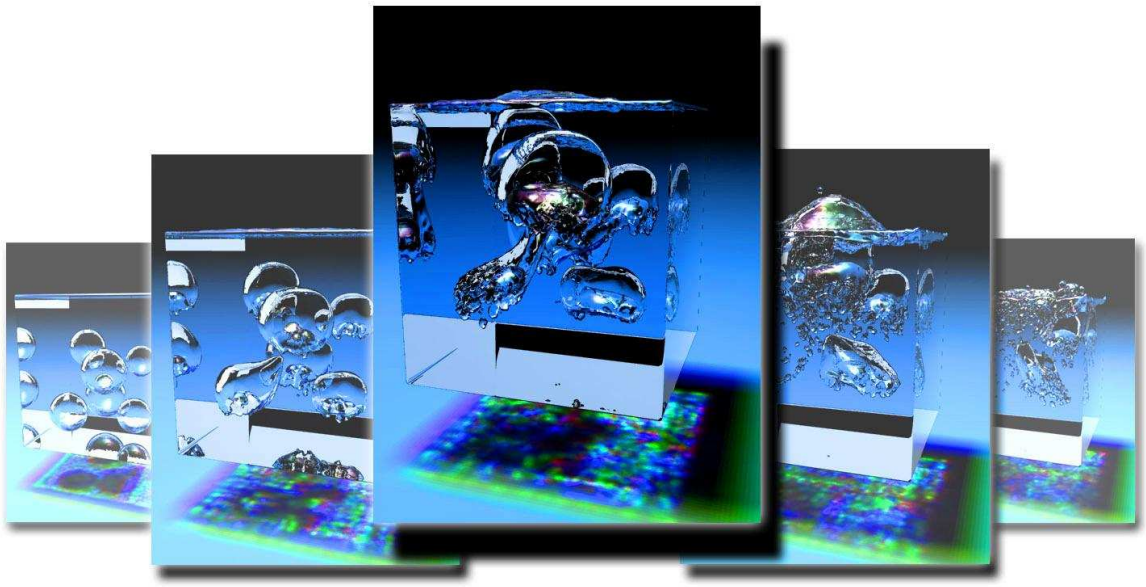
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

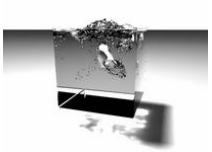
Lehrstuhl für Informatik 10 (Systemsimulation)



A single-phase free-surface Lattice Boltzmann Method

Nils Thürey

Diplomarbeit

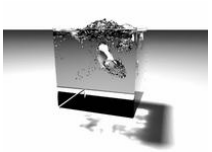


A single-phase free-surface Lattice Boltzmann Method

Nils Thürey

Diplomarbeit

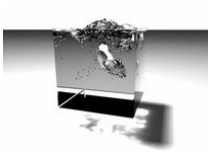
Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Thomas Pohl
Bearbeitungszeitraum:	2002-12-01 bis 2003-06-02



Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 13. Januar 2005 .....



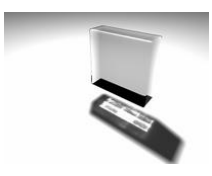
Abstract

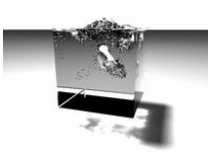
A single-phase free-surface Lattice Boltzmann Method

In the past years, the Lattice Boltzmann Method (LBM) has become popular for many applications in the field of computational fluid dynamics. Its simple update rules allow easy implementation and optimizations like parallelization and blocking, which are especially interesting in the high performance computing sector.

The LBM has been applied to metal foam simulation for two-dimensional problems in [Thies, 2000]. This thesis will present an expansion of this fluid model to 3D, as well as an alternative method to calculate the surface tension. Metal foams are interesting for material sciences, as they allow the production of light and stable structures with excellent physical properties. The process of foaming is up to now not completely understood, and often based on trial and error. Hence, the target is to use numerical simulation to gain understanding and optimize the necessary parameters of the foaming process like temperature and gas concentration. This thesis will focus on the simulation of the motion of the liquid metal phase.

The fluid solver presented here is capable of simulating a single fluid phase with a free surface, including surface tension, bubbles and coalescence. In contrast to the standard multi-phase models of [Gunstensen et al., 1991] and [Swift et al., 1996] it is not necessary to simulate the motion of the gas phase, which overcomes restrictions in difference of the viscosity of the two phases, and improves the computational time needed for the numerical simulation, as the gas volumes in the foam do not require any additional computations. The forces at the fluid interface, gas pressure and surface tension, are applied by reconstructing the missing information from the gas phase. For surface tension, points on the fluid surface are calculated with the marching cubes algorithm and used to retrieve an average curvature at the fluid-gas interface. As part of this thesis, the fluid solver is tested and validated with standard problems like the breaking dam problem and rising bubble problems.





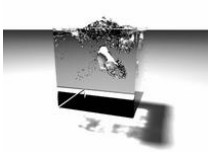
Zusammenfassung

Eine Lattice Boltzmann Methode fuer einzelne Phasen mit freien Oberflächen

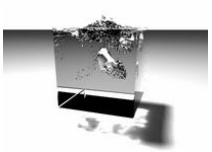
In den letzten Jahren ist die Lattice Boltzmann Methode (LBM) im Bereich der Strömungssimulation sehr populär geworden. Die einfachen Regeln des Algorithmus erlauben eine einfache Implementierung und umfangreiche Optimierungen wie Paralellisierung und Blocking, was besonders fuer den High-Performance Sektor wichtig ist.

Die LBM wurde erfolgreich für zweidimensionale Metallschaum-Simulationen in [Thies, 2000] angewendet. In dieser Arbeit wird eine Erweiterung des dort vorgestellten Algorithmus auf 3D implementiert, und gleichzeitig eine alternative Methode zur Berechnung der Oberflächenspannung verwendet. Metallschäume sind in für die Werkstoffwissenschaftan aufgrund ihrer exzellenten physikalischen Eigentschaften interessant. Die Produktion der Metallschäume ist momentan noch nicht vollständig verstanden, und ist dadurch oft auf das Ausprobieren der geeigneten Parameter bezüglich Temperatur und Gaskonzentration angewiesen. Es bietet sich darum an, numersiche Simulation einzusetzen um die Kontrollierbarkeit des Prozesses zu vergrössern. In dieser Arbeit liegt der Schwerpunkt auf der Simulation der Bewegung des flüssigen Metalls.

Der hier vorgestellte Löser ist in der Lage eine Flüssigkeitsphase mit freier Oberfläche, Oberflächenspannung und Blasen mit Koaleszenz zu simulieren. Im Unterschied zu bekannten Mehrphasen-Modellen, wie zum Beispiel in [Gunstensen et al., 1991] und [Swift et al., 1996], ist es bei dem hier vorgestellten nicht nötig die Gasphase mit zu simulieren. Dadurch ergeben sich weniger Probleme auch bei grösseren Viskositätsunterschieden der Gas- und Flüssigkeitsphase. Gleichzeitig wird viel Rechenzeit eingespart, da, vor allem bei Schäumen, die Gas-Phase grosse Bereiche des Arbeitsgebietes einnimmt. Die an der Flüssigkeitsoberfläche wirkenden Kräfte, Druck und Oberflächenspannung, werden über eine Rekonstruktion der Informationen aus der Gas-Phase angewandt. Für die Oberflächenspannung werden mithilfe des Marching-Cubes Algorithmus Punkte an der Flüssigkeitsoberfläche berechnet, welche zur Berechnung einer durchschnittlichen Krümmung an der Flüssigkeits-Gas Grenze herangezogen werden. Des weiteren werden im Rahmen dieser Arbeit Validierungen mit Standardproblemen wie dem Breaking-Dam Problem und Simulationen von aufsteigenden Blasen vorgenommen.



Contents



Chapter 1

Introduction

Fluid simulations with a free surface are needed in a variety of engineering and environmental applications that range from mold filling, bubble and wave dynamics to the simulation of ocean-atmosphere interactions. This thesis will present a three-dimensional free surface fluid simulation based on the Lattice Boltzmann Method. The targeted application, that will be described in this chapter, is the simulation of metal foams. The difference to other multi-phase methods is that the gas phase only has negligible influence on the liquid metal phase and is therefore not simulated as a fluid. This significantly reduces the computational complexity of the method. This single-phase method was presented in [Arnold et al., 2000] and is successfully used to simulate metal foams in two dimensions.

This thesis will focus on the three-dimensional fluid solver, that will be used to simulate the movement of the liquid metal phase, in order to later-on provide the tool for metal foam simulations. The gas diffusion and solidification process are not taken into consideration in the scope of this thesis.

1.1 The Lattice Boltzmann Method

The Lattice Boltzmann Method (*LBM*) emerged from the lattice gas cellular automata (*LGCA*). The first experiments with fluid simulations using cellular automata are described in [Hardy et al., 1976]. These simulations, however, were not corresponding to the Navier-Stokes equations. It took ten years, until in [Frisch et al., 1986] the importance of the *lattice symmetry* was discovered. This resulted in increased research activities, one of which was to replace the boolean particle representations of the LGCA by particle distribution functions. This reduced the problem of noise in the LGCA models. Other enhancements were proposed for the collision operator. The *Bhatnagar-Gross-Krook approximation* was introduced to the LBM by [Quian et al., 1992; Chen et al., 1992]. This model is also known as the lattice-BGK model (LBGK), and was used for the implementation of this thesis.

The idea of cellular automata is quite different from the usual representation of problems using partial differential equations (PDE's). While these have to be discretized, usually resulting in a

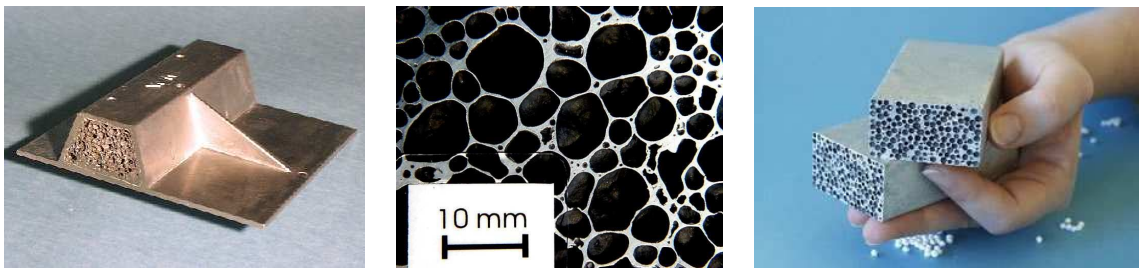


Figure 1.1: Three samples of metal foams produced at the WTM in Erlangen, the photos were taken by Michael Thies.

linear system of equations that can be numerically solved, cellular automata model a problem with a set of equal cells. All cells are treated with the same rules, the complex behavior results from interactions of the cells with their neighborhood. One of the most popular is Conway's *game of life*, which models the development of a population. The implementation of cellular automata in a computer program is usually straightforward, as they are based on regular cells with a local neighborhood and discrete values. These are also the benefits of using LBM as a fluid simulation method – the implementation is relatively easy and efficient. Furthermore it allows the application of many optimization and paralellization techniques known from other numerical algorithms, resulting in high-performance implementations (e.g. [Zeiser et al., 2002]). The LBM has been validated for a variety of applications, and is especially well suited for complex geometries (such as foams), due to the simple handling of boundary conditions.

1.2 Metal foams

Metal foams are interesting for material sciences due to their combination of very low density and excellent mechanical, thermal and acoustic properties, which makes them applicable to automotive lightweight constructions, for example. However the production process is still not completely understood, resulting in both high costs and many rejects. The resulting metal foams often have inhomogeneities with strong variations in pore structure and sizes. The resulting metal foams therefore do not have the desired physical properties. The successful production of new materials, such as metal foams, requires a high level of understanding for the process. Thus, numerical simulation can be used as a tool to increase the control of the process, ensure reproducibility and furthermore gain understanding of the production process. This is the target of the *FreeWIHR* project. Due to the high computational complexity of metal foam simulation in three dimensions the *Hitachi SR-8000 F1* at the Leibniz-Rechenzentrum in Munich will be used.

The foaming process that is described in [Körner and Singer, 1999] uses aluminum powder mixed with (TiH_2) as foaming agent. This composite is compacted and foamed by heating it above the melting temperature. The foaming agent is split up, and the Hydrogen is released into the liquid aluminium where it diffuses to the bubble nuclei that are already present in the material. The pressure in the bubbles increases, leading to an expansion of the foam due to the growing bubble volumes. The expansion of the foam continues until more gas is lost to the environment than is provided by the foaming agent. This loss occurs due to diffusion and bursting bubbles at the foam surface. Different time steps of a two dimensional simulation of this process is shown in Figure 1.2. The simulation of this foaming process in two dimensions has been described in [Körner and Singer, 2000; Körner et al., 2000].

1.3 Structure of this thesis

This thesis is structured to first give a short introduction to the implementation of Lattice Boltzmann Methods, targeted to readers that are not deeply familiar with fluid simulations. Chapter 3 will then describe the underlying math and physics, deriving the Lattice Boltzmann equation from

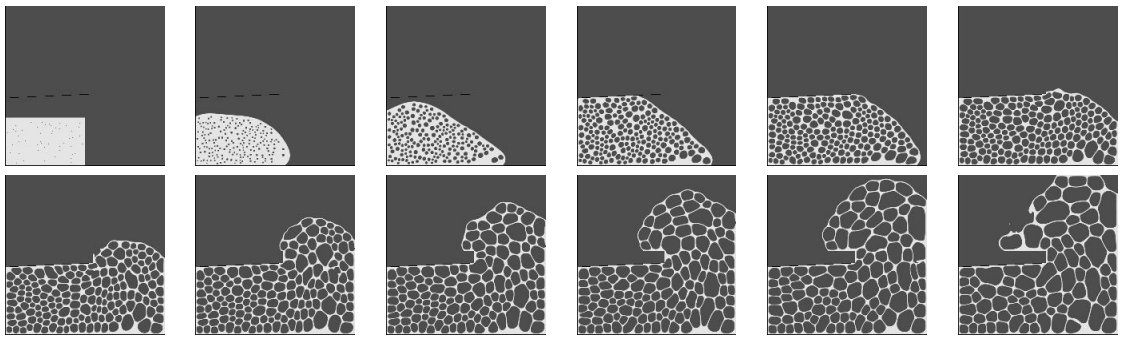
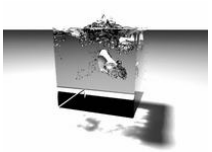
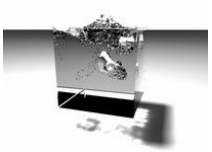


Figure 1.2: Several time steps of an animation from [Arnold et al., 2000]. A metal foam evolves in a container with an horizontal obstacle.



an arbitrary kinetic equation, and showing its correspondence to the Navier-Stokes equations. The next chapter will describe the model used in this thesis in more detail, the implementation of which is explained in Chapter 5. The results will be evaluated in Chapter 6. Standard problems like a breaking dam and falling drops were tested, as well as several problems with rising bubbles targeted to the suitability for foaming simulations. This thesis concludes with notes on the visualization techniques used, and an overview of future enhancements that will be necessary to realistically simulate metal foams in three dimensions.



Chapter 2

LBM for computer scientists

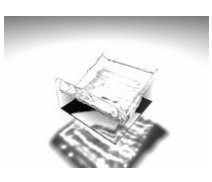
The aim of this chapter is to give an overview of the Lattice Boltzmann Method and its implementation, which is important for the next chapter, where the physical and mathematical basics will be explained.

2.1 The D3Q19 lattice

Lattice Boltzmann Methods, as the name suggests, work on a given lattice. Depending on the field of application, different lattices can be used. These are named $DXQY$, where X is the number of dimensions, 3 in this case, and Y determines the number of distinct lattice velocities. LBM can be described as a type of cellular automaton, which means that the fluid is modeled by many cells of the same type. All cells are updated each time step by simple rules, that take into account the state of the surrounding cells. The complex behavior of the automaton emerges from the interaction of the cells, and not, for example, by describing system properties as functions of space and time.

The LBM models an incompressible fluid by particles, that are allowed to move only along the lattice velocity vectors. A single cell of the D3Q19 model is depicted in Figure 2.1. Each side of the cell has length 1, so the volume of a cell is 1, too. The lattice velocities are shown and numbered. They point from the cell center to each face of the surrounding cube, and to the midpoints of all cube edges, but not to the eight corners. There is also a LBM model that includes lattice vectors to the corners, the D3Q27 model, but as it does not improve stability in this case and increases memory requirements, D3Q19 is used here. For the simulation, all cells have to store the number of particles that move along each of the lattice vectors, the *particle distribution functions*. They are usually called f_i where the values for i correspond to the numbers of the lattice vectors. If cells of this type are arranged in a three-dimensional grid, each lattice vector points to a neighboring cell, that has a particle distribution function into the inverse direction. Note, that the vector with the number 0 has length zero, and amounts for particles resting in this cell. These particles are not moving anywhere in the next time step, but some of them may be accelerated due to collisions with other particles - so the amount of resting particles can change.

From the particle distribution functions two important physical values can be calculated. By summing up all 19 distribution functions the density for the volume of this cell can be calculated, assuming that all particles have the same mass of 1. As the distribution functions contain the amount of particles moving in a certain direction for each cell, the sum of all particles in a single cell is its density (the mass per volume). Another important information for each cell is the speed and overall direction in which the particles of one cell move. For this the *momentum density* needs to be calculated. It is again the sum of all particle distribution functions, but each distribution function is first multiplied by the lattice vector. Thus, the particle distribution function 0 is multiplied with $(0,0,0)^T$, which always amounts to zero, distribution function f_1 is multiplied by $(1,0,0)^T$ and added to distribution function f_2 times $(-1,0,0)^T$ and so on. This results in a three-dimensional vector, that is scaled by the density, as the particle distribution functions contain a total amount of particles. So, simply dividing the momentum density by the density, calculated as described above, yields the velocity vector for a cell. For simplicity the density is usually set to one in the beginning of a simulation. While the LBM is used to simulate incompressible fluids, meaning that



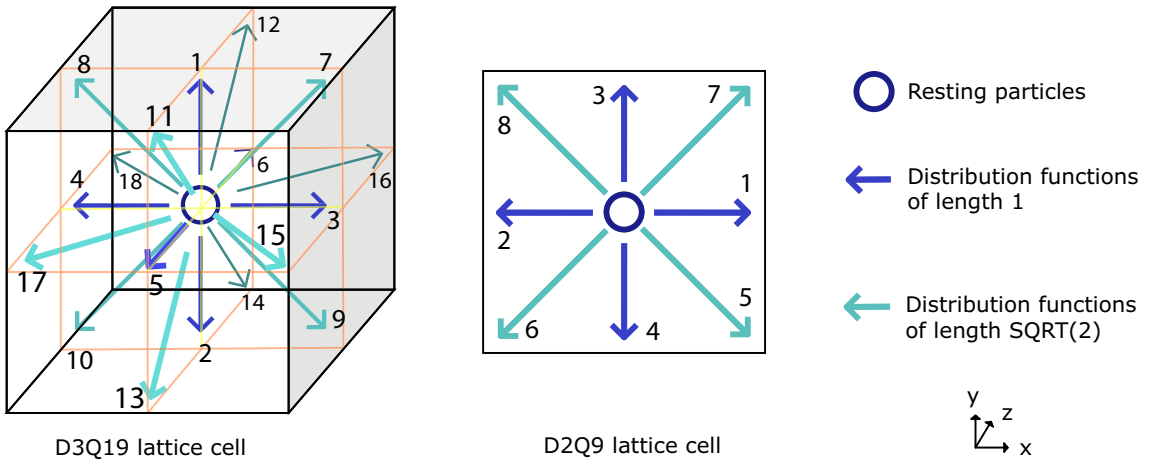


Figure 2.1: The 19 distinct velocities of the D3Q19 model point to every face and edge of a cube around the cell, except for the 8 corners. For comparison the D2Q9 model is shown to the right; both models have speeds of length 0, 1 and $\sqrt{2}$.

the density is constant everywhere in the fluid, this constraint is relaxed during a LBM simulation. In a simulation one will usually encounter density differences, but for reasonable parameters the overall behavior will equal that of an incompressible fluid.

The simulation process consists of two steps, that are repeated for each time step. One is the *stream step*, in which the actual movement of the particles throughout the grid is performed, the other accounts for the collisions that occur during this movement, and thus is called the *collide step*. For simplicity, the size of a cell, the length of a timestep and the initial density will be normalized to 1 and will not be included in the following formulas. For a more general description please refer to Chapter 3.

The stream step consists only of copying operations, as shown in Figure 2.2. For each cell, all distribution functions are copied to the adjacent cell in the direction of the lattice vector. Hence, for the cell with the coordinates $[i, j, k]$ the distribution function for the lattice vector pointing upwards is copied to the upward distribution function of cell $[i, j + 1, k]$. As the lattice vector 0 does not point anywhere, its particle distribution function is not changed in the stream step. In practice, when writing a program that performs the streaming, the easiest way to implement this behavior is to use two different grids, and copy the distribution function from one grid to the other. This is necessary to prevent any overwriting of distribution functions that are needed for the streaming of another cell.

The collide step is a bit more complicated. Performing stream steps would just result in the distribution functions moving through the grid. The velocities and densities of the cells would

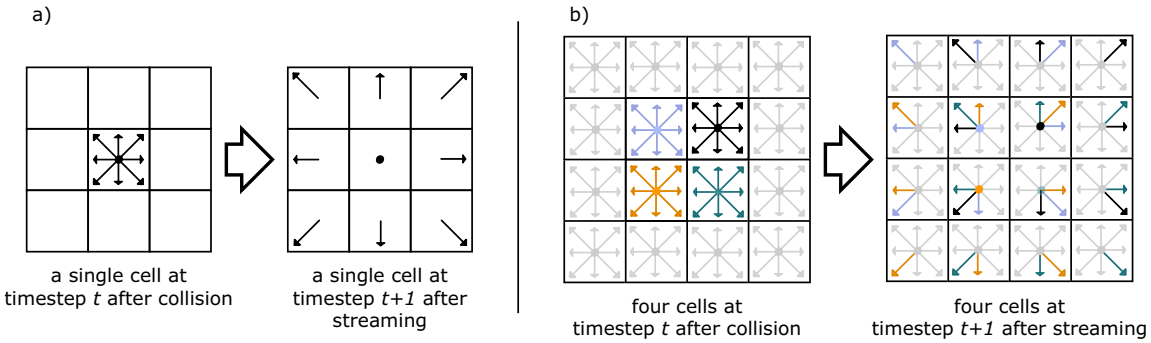
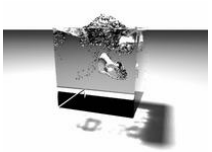


Figure 2.2: Each two pictures show the particle distribution functions of a D2Q9 LBM grid before and after streaming. To the left the distribution functions of a single cell can be seen, while the right pictures show four different LBM cells.



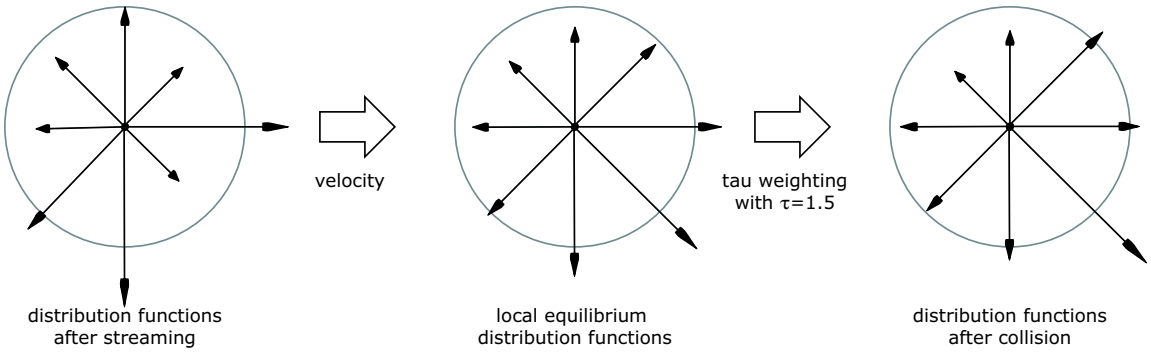


Figure 2.3: During the collide step the distribution functions from the stream step are used to calculate the velocity in each cell, which is necessary for the local equilibrium distribution functions. Thse are weighted with the parameter τ to yield the distribution functions for the next stream step.

change, with no further interaction. However, in a real fluid, the particles are constantly colliding, scattering other particles into different directions and due to their chaotic movement influencing adjacent layers of fluid. The collide step does not change the density or velocity of a cell, it only changes the distribution of the particles for all particle distribution functions.

Consider, for example, a cell $[i, j, k]$ where the fluid moves along the positive x-axis. It will not lose any particles during collision, but the movement will be scattered to other lattice velocities that point in direction of the positive x-axis. The ones pointing into the opposite directions will become smaller. In the next stream step, neighboring cells with x coordinate $i + 1$ will receive a slightly larger particle distribution function from cell $[i, j, k]$, while cell at $i - 1$ will receive slightly smaller ones. In the collide step several parameters are necessary, for which a formal justification will be given in Chapter 3. The relaxation time ω, that can have values in the range of 0 and 2, determines the viscosity of the fluid. For small values (< 1) the fluid will behave more like honey, while values close to 2 will result in very turbulent and chaotic behavior (resembling fluids like water). The density of the cell will be denoted by ρ while the velocity vector is $\vec{u} = (u_1, u_2, u_3)^T$. The lattice velocity vectors from Figure 2.1 are $\vec{e}_{0..18}$, each one having a weight w_i. The derivation of these weights is explained in more detail in Chapter 3. For colliding the equilibrium distribution function has to be calculated from density and velocity first :

$$f_i^{(0)} = w_i \left[\rho - \frac{3}{2}(\vec{u})^2 + 3(\vec{e_i} \cdot \vec{u}) + \frac{9}{2}(\vec{e_i} \cdot \vec{u})^2 \right] \tag{2.1}$$

with $w_i = \frac{1}{3}$ for $i = 0$, $w_i = \frac{1}{18}$ for $i = 1..6$ and $w_i = \frac{1}{36}$ for the remaining $i = 7..18$. This formula can be derived by a Taylor expansion of the Maxwell distribution. The three scalar products between the velocity and lattice vectors can be calculated easily. They need to be scaled accordingly and then summed up to be adjusted by the according weight and density. Now, depending on ω the fluid reaches this equilibrium faster or slower. The new particle distribution functions f_i' are calculated with:

$$f_i' = (1 - \omega) f_i + \omega f_i^{(0)} \tag{2.2}$$

These f_i''s are then stored in the corresponding cell. When all cells have been collided the next stream step can be performed.

The streaming and collide step can be combined to one formula, which is often found in literature about LBM:

$$f_i(\vec{x} + \vec{e_i}\triangle t, t + \triangle t) - f_i(\vec{x}, t) = -\frac{1}{\tau}\left(f_i - f_i^{(0)}\right) \tag{2.3}$$

As the time step $\triangle t$ was set to 1 in the other formulas, it did not yet appear. The left hand side of this equations accounts for the stream step, as $\vec{x} + \vec{e_i}$ directly points to the distribution function of another cell. The right hand side can be recognized as the combination of the current distribution function and the local equilibrium.

2.2 The code

The LBM described in Section 2.1 can be directly transformed into a program. This can be seen in Algorithm 2.1. The code is written in a pseudo-code similar to C. In Lines 3 to 9 the variables are declared. As C does not allow multidimensional arrays of variable size, the size of the computational domain is set with a macro. The variables from Line 5 on allow the handling of the particle distribution functions for each cell in a loop. All these loops could be unrolled and optimized, for example by removing multiplications with e_x-values, that are known to be zero.

First all cells are initialized with the equilibrium distribution functions, and thus are at rest. Without any external forces, all cells will remain unchanged. The stream step from Line 25 on copies the distribution functions from the old grid to the current one. A common mistake is to use the cell in the direction of the current distribution function, e.g. $e_x[l]$, instead of the inverse direction (which is e_x[inv] in Algorithm 2.1 Line 31, where *inv* is the opposite direction of l).

The collide step finally directly computes the Equations 2.1 and 2.2 and stores these values again. After this, the grids are swapped, so the next stream step will operate on the collided values.

2.3 Some improvements

Although the program described in Section 2.2 performs a full lattice boltzmann simulation, if it is used to perform a simulation, nothing will change over time. This section will describe some extensions for boundary conditions and external forces.

2.3.1 Boundary conditions

In the current version of the code, the boundary conditions are not specifically handled – they do not conserve mass, and just insert equilibrium distribution functions into the fluid. The standard boundary conditions for LBM simulations are no-slip walls, i.e. close to the boundary the fluid does not move at all. Hence, each Lattice Boltzmann cell next to a boundary should have the same amount of particles moving into the boundary as moving into the opposite direction. This will result in a zero velocity, and can be imagined as reflecting the particle distribution functions at the boundary. The reflection process is shown in Figure 2.4, left, for the no-slip case. To the right, however, the free-slip case is illustrated, for which only the velocities normal to the boundary are reflected.

For the implementation this means, that boundary and fluid cells need to be distinguished. A flag array has to be introduced and initialized to declare all boundary cells as "no-slip" and all inner cells as "fluid". The real handling of the boundary cells would happen in the streaming loop at line 31 of Algorithm 2.1. Here the flag array had to be checked, and if the neighboring cell is a boundary cell, the opposite distribution function from the current cell would be taken. The code fragment to replace line 31 is shown in Algorithm 2.2.

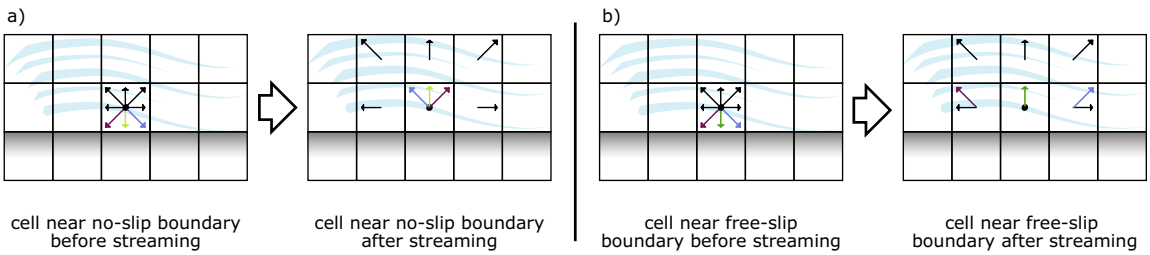
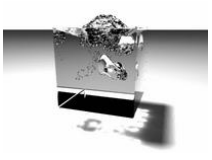
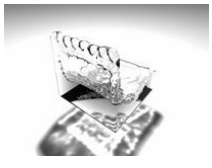


Figure 2.4: No-slip obstacle cells directly reflect the incoming distribution functions. For free-slip boundary conditions, the distribution functions are reflected along the normal direction of the boundary.



Algorithm 2.1 D3Q19 LBM

```
1: // Declare variables and initialize...
2: #define SIZE 50;
3: double ω = 1.9; int steps = 100; int current = 0, other = 1;
4: double cells[2][SIZE][SIZE][SIZE][19];
5: const double w[19] = {1/3, 1/18, 1/18, 1/18, 1/18, 1/18, 1/18, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36, 1/36};
6: const int e_x[19] = {0,0,0,1,−1,0,0,1,−1,1,−1,0,0,0,0,1,1,−1,−1};
7: const int e_y[19] = {0,1,−1,0,0,0,0,1,1,−1,−1,1,1,−1,−1,0,0,0,0};
8: const int e_z[19] = {0,0,0,0,0,1,−1,0,0,0,0,1,−1,1,−1,1,−1,1,−1};
9: const int f_inv[19] = {0,2,1,4,3,6,5,10,9,8,7,14,13,12,11,18,17,16,15};
10:
11: // All cells are resting first
12: for i = 0; i < SIZE do
13:   for j = 0; j < SIZE do
14:     for k = 0; k < SIZE do
15:       for l = 0; l < 19 do
16:         cells[0][i][j][k][l] = w[l];
17:         cells[1][i][j][k][l] = w[l];
18:       end for
19:     end for
20:   end for
21: end for
22:
23: // Start simulation
24: for s = 0; s < steps do
25:   // Stream from the other to the current grid...
26:   for i = 1; i < SIZE − 1 do
27:     for j = 1; j < SIZE − 1 do
28:       for k = 1; k < SIZE − 1 do
29:         for l = 0; l < 19 do
30:           int inv = f_inv[l];
31:           cells[current][i][j][k][l] = cells[other][ i + e_x[inv] ][ j + e_y[inv] ][ k + e_z[inv] ][l];
32:         end for
33:       end for
34:     end for
35:   end for
36:   // Collide...
37:   for i = 1; i < SIZE − 1 do
38:     for j = 1; j < SIZE − 1 do
39:       for k = 1; k < SIZE − 1 do
40:         // Calculate density and velocity
41:         double ρ = 0.0, u_x = 0.0, u_y = 0.0, u_z = 0.0;
42:         for l = 0; l < 19 do
43:           double f_i = cells[current][i][j][k][l];
44:           ρ = ρ + f_i; u_x = u_x + f_i ∗ e_x[l]; u_y = u_y + f_i ∗ e_y[l]; u_z = u_z + f_i ∗ e_z[l];
45:         end for
46:         // Perform collision
47:         for l = 0; l < 19 do
48:           double f^0 = w[l] ∗ ρ ∗ (1.0 −
49:             3.0/2.0 ∗ (u_x ∗ u_x + u_y ∗ u_y + u_z ∗ u_z) +
50:             3.0 ∗ (e_x[l] ∗ u_x + e_y[l] ∗ u_y + e_z[l] ∗ u_z) +
51:             9.0/2.0 ∗ (e_x[l] ∗ u_x + e_y[l] ∗ u_y + e_z[l] ∗ u_z)^2 );
52:           cells[current][i][j][k][l] = (1 − ω) ∗ cells[current][i][j][k][l] + ω ∗ f^0;
53:         end for
54:       end for
55:     end for
56:   end for
57:   // We're done for one time step, swap the grids...
58:   other = current; current = 1 − current;
59: end for// Simulation is done!
```



Algorithm 2.2 No-slip boundary condition handling

```
1: // Check flag array
2: if flags[ i + e_x[inv] ][ j + e_y[inv] ][ k + e_z[inv] ] == NOSLIP then
3:    // Boundary cell
4:    cells[current][i][j][k][l] = cells[current][i][j][k][inv];
5: else
6:    // Normal fluid cell
7:    cells[current][i][j][k][l] = cells[other][ i + e_x[inv] ][ j + e_y[inv] ][ k + e_z[inv] ][l];
8: end if
```

2.3.2 External forces

In the current version of the code, constant forces like gravity do not exist. To introduce these, a force can be applied to all fluid cells. The only necessary change would be to insert a statement into Algorithm 2.1 after line 45 and add an acceleration to the velocity vector $\vec{u}$. For example "$u_y- = 0.01$;" would result in a force along the negative y-axis, corresponding to an acceleration vector of $(0, -0.01, 0)^T$. When the simulation is started now, the fluid will be compressed by the external force. After some time the fluid will be at rest again and a density gradient will be visible. For the example above, the density at the bottom of the domain will be higher than the density at the top.

The standard test problem for two-dimensional fluid solvers is the *lid-driven cavity*. It consists of a rectangular domain filled with fluid and no-slip boundary conditions at the sides. One of the walls is moving along the domain, accelerating the fluid – in Figure 2.5 it is the upper wall. After some time, a vortex in the middle of the domain is usually visible, accompanied by smaller vortices in the corners which rotate in the opposite direction of the center vortex. This test problem can be directly transferred to 3D. It can even be experienced in real life – imagine sitting in a court yard, with wind moving the clouds into a certain direction above you. When small objects like leaves are thrown into the air at the bottom of the court yard, these will move in the opposite direction of the clouds. For the implementation of a lid driven cavity, a third type of cells called accelerator cells has to be introduced. The velocity in these cells will be set to a fixed value, e.g. $(0.01, 0, 0)^T$, and the density is set to one. This is done instead of the usual computation of density and velocity in the collide step. Afterwards the cells are collided like standard fluid cells by relaxing the particle distribution functions with the local equilibrium distribution. Typically the fluid cells with $y = (SIZE - 2)$ will be initialized as accelerator cells, so the domain will be surrounded by no-slip boundary cells, and the topmost fluid layer will have a fixed velocity. The disadvantage of this simple implementation is that it does not conserve mass, as the density of the accelerator cells is always one, but for testing purposes this error is usually negligible.

2.4 Conclusion

This chapter has shown that the implementation of the LBM is relatively easy. The program discussed here is capable of a complete Navier-Stokes compliant fluid simulation, including external forces and complex boundaries. By setting selected inner cells to be no-slip obstacles, complex geometries can be approximated easily. Still, the nature of the LBM allows parallelization of the algorithm and efficient implementation.

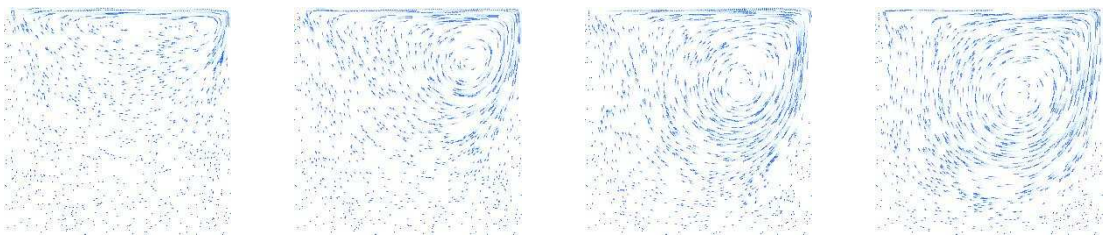
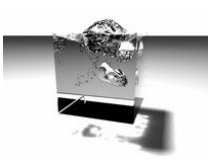


Figure 2.5: Four time steps from a two-dimensional lid driven cavity simulation.



Chapter 3

Mathematical and physical background

This chapter will shortly explain the basics of the Boltzmann equation, the derivation of the Lattice Boltzmann equation and the connection to the Navier-Stokes equations. Furthermore an overview of some basic properties of fluids will be given.

3.1 The Navier-Stokes equations

The origins of the nowadays established Navier-Stokes equations reach back to Isaac Newton, who around 1700 formulated the basic equations for the theoretical description of fluids. These were used by L. Euler half a century later to develop the basic equations for momentum conservation and pressure. Amongst others, Louis M. H. Navier continued to work on the fluid mechanic equations at the end of the 18th century, as did Georg G. Stokes several years later. He was one of the first to analytically solve fluid problems for viscous media. The Navier-Stokes equations could not be practically used until in the middle of the 20th century the numerical methods were developed, that are necessary to solve the resulting equations.

One of the most important aspects for a fluid mechanical problem is the conservation of mass M, which has to remain constant for the fluid system. For a fluid in a container the domain boundaries are obvious, but even for more complex systems, boundaries in which the total mass has to remain constant can usually be found. The equation for mass conservation – the *continuity equation* — can be derived by bringing a mass conservation law in Lagrangian variables (considering a fluid element with a given position, velocity and constant mass, see Figure 3.1) into Eulerian form. Hence, the mass conservation will be described by continuous variables, and not variables bound to a certain fluid element. For compressible fluids the continuity equation has the following form:

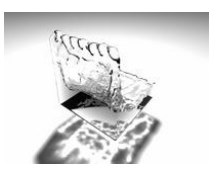
$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_i)}{\partial x_i} = 0 \tag{3.1}$$

Note that the second partial derivative is written in Einstein summation convention, meaning that the i subscript appearing twice denotes a sum over all possible values. In this case it is a sum over all three dimensions of $\vec{u}$ and $\vec{x}$. The continuity equation is used together with the Navier-Stokes equations to prevent any loss of mass.

The Navier-Stokes equations themselves can be derived by applying Newton's second law to a fluid element, hence, choosing a Lagrangian formulation. In Eulerian form, these *momentum equations* are:

$$\underbrace{\rho\left(\frac{\partial u_j}{\partial t} + u_i \frac{\partial u_j}{\partial x_i}\right)}_{\text{advection}} + \underbrace{\frac{\partial P}{\partial x_j}}_{\text{pressure}} + \underbrace{\frac{\partial \tau_{ij}}{\partial x_i}}_{\text{momentum}} = \rho g_j, \quad j = 1, 2, 3 \tag{3.2}$$

Three parts of this equation can be distinguished. The first part of the equation is responsible for mass forces like advection. The partial derivatives of the pressure P are surface forces acting upon the fluid. The third, and most complicated part, contains the tensor τ_{ij}, and introduces momentum



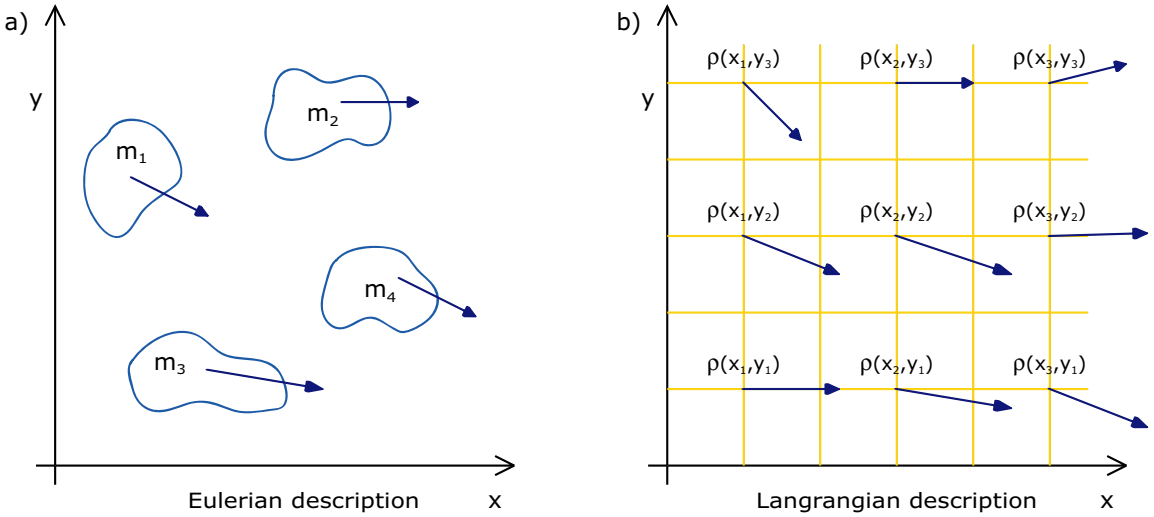


Figure 3.1: This picture shows the difference of a Lagrangian (to the left) and a Eulerian description (to the right). The first case considers fluid elements with certain properties such as mass and velocity, whereas the Eulerian description consists of a continuous field of values for the density and velocity.

effects due to molecule movement. Imagine a cubic fluid element, with molecules "jumping" in and out at all borders due to the natural Brownian movement of the molecules. Even molecules moving in the same direction with different velocities, can move perpendicularly into an adjacent element, and influence it. Thus τ_{ij} accounts for the diffusion of the velocities, separately for each component in all three directions. This effect is similar to a friction between the fluid layers, but is in reality caused by the molecule exchange described above [Durst, 2002].

For Newtonian fluids (i.e., the viscosity is independent of the shear rate), τ_{ij} can be computed as follows:

$$\tau_{ij} = -\mu \left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j} \right) + \frac{2}{3} \delta_{ij} \mu \frac{\partial u_k}{\partial x_k} \tag{3.3}$$

In this equation, μ denotes the dynamic shear viscosity, a value depending on the physical properties of the fluid. The Kronecker symbol denotes the tensor where $\delta_{ij} = 1$ for $i = j$ and $\delta_{ij} = 0$ otherwise. As the τ_{ij} can be computed with Equation (3.3), this leaves five unknown variables in four Equations (3.1), (3.2). However, for incompressible ($\rho = const$), Newtonian ($\mu = const$) and energy conserving fluids ($\frac{\partial^2 u_i}{\partial x_j \partial x_i} = 0$) the resulting four equations are:

$$\frac{\partial u_i}{\partial x_i} = 0 \quad \text{(continuity equation)} \tag{3.4}$$

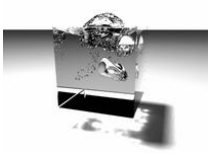
$$\rho \left(\frac{\partial u_j}{\partial t} + u_i \frac{\partial u_j}{\partial x_i} \right) + \frac{\partial P}{\partial x_j} = \mu \frac{\partial^2 u_j}{\partial x_i^2} + \rho g_i \quad \text{(Navier-Stokes equations)} \tag{3.5}$$

With adequate initial and boundary conditions, these equations can be discretized, using finite differences or finite volumes, and solved using numerical algorithms such as Gauss-Seidel or multigrid. For example, a C implementation for a finite-difference discretization using explicit time stepping is given in [Griebel et al., 1988].

In fluid mechanics these equations are usually treated in a dimensionless way. This is valid as fluids behave similar at different size and time scales. The most important value for the characterization of a problem is the *Reynolds number* R. It is dimensionless and can be calculated in the following way:

$$R = \frac{\rho \, \vec{U} \, L}{\nu} \tag{3.6}$$

where ρ is the fluid density, $\vec{U}$ the macroscopic flow speed and L the characteristic length or distance of the problem. Thus, a fluid with a given velocity and viscosity behaves similar to one with a lower



velocity and smaller viscosity. Or two problems with the same fluids, and viscosities, respectively, are comparable when the flow speed is increased and the characteristic length is decreased. For example, in order to measure the flow around an aerodynamic body, this can be performed with a smaller model and increased flow speed in a wind-tunnel.

3.2 The Boltzmann Equation

The Boltzmann equation is known since 1872, named after the Austrian scientist Ludwig Boltzmann. It is part of the classical statistical physics, and describes the behavior of a gas on the microscopic scale.

The kinetic theory of gases deals with the description of gas states on the molecular level [Frohn, 1979]. It introduces a function to describe the state of the gas by considering the position and velocity of each molecule in the gas. In a three-dimensional space both require three values, hence a molecule can be described by the six values $(x_1, x_2, x_3, \xi_x, \xi_y, \xi_z)$ where $(x_1, x_2, x_3) = \vec{x}$ is the position, and $(\xi_x, \xi_y, \xi_z) = \vec{\xi}$ is the velocity of the molecule. A gas with N molecules can then be described by N points in the six-dimensional space (usually called the *μ-space*) for position and velocity by specifying the six values for each molecule. Since a typical gas like air contains about $2.7 \cdot 10^{19}$ molecules per cm^3, it is at the time of this writing not possible to successfully perform simulations at this level of detail. The required memory capacity would exceed those of current machines, and even small uncertainties in the initialization would lead to dynamical instabilities. To handle the molecular description the kinetic theory of gases defines a function for a volume $d\vec{x}d\vec{\xi}$ at $(x_1, x_2, x_3, \xi_x, \xi_y, \xi_z)$ in μ-space describing the density of the gas. Let dN be the number of points in this volume (all molecules with position and velocity within the specified range), then

$$f(\vec{x}, \vec{\xi}) = \frac{dN}{d\vec{x}d\vec{\xi}} \tag{3.7}$$

is the *molecular velocity distribution function*. Integrating this function over all possible values for position and velocity would yield N, the total number of molecules. f now only includes all particles in an interval around x that move in the same direction. This function is a fundamental value in the Boltzmann equation. The following three approximations are necessary to derive the Boltzmann equation:

1. Particle collisions are only considered between two particles. In this form it restricts the equation to *diluted* gases, where it can be assumed that in each collision only two particles are involved.
2. The particles are *point-like* and *structureless*. Hence, it is assumed, that their velocities are not correlated before and after the collision.
3. The third approximation is, that the collision dynamics are instantaneous and not influenced by an external force (which is equivalent to a *short-range* potential for the interaction).

The second and third assumption need to be made, as strictly speaking the interaction potential is infinite and all particles are constantly influencing each other, but their strength decreases quickly with respect to the distance. So it can be assumed that the region of influence is just in a very short range around the particle, and two particles collide when their influence regions touch each other. In effect, nearly all of the time, the considered particles move around only affected by an external force, the only interaction being a binary collision, that changes the momentum and direction of the colliding particles only. This collision process between two particles is considered instantaneous, and as such not affected by any external forces.

Including an external force g, the Boltzmann equation for f can be written as:

$$\frac{\partial f}{\partial t} + \vec{\xi} \cdot \frac{\partial f}{\partial \vec{x}} + \vec{g} \cdot \frac{\partial f}{\partial \vec{\xi}} = \Omega(f) \tag{3.8}$$

Here the left hand side describes the overall motion of the molecules with the microscopic velocity $\vec{\xi}$ through the force field that is given by $\vec{g}$ at $\vec{x}$, while the right hand side models the interaction of molecules with the collision operator Ω. It is an integral equation that includes the *differential*

collision cross section σ for the two particles, which can be calculated geometrically by approximating the molecules with rigid spheres for the collision [Frohn, 1979]. The incoming particle velocity is transformed into the outgoing velocity, and as such, the equation represents the link to the underlying molecular dynamics. Ω for two particles with the subscripts 1 and 2 can be written as:

$$\Omega(f) = \int (f_1' f_2' - f_1 f_2)\ \sigma(|\vec{u_1} - \vec{u_2}|, \vec{\omega})\ d\vec{\omega} d\vec{x_2}, \tag{3.9}$$

where $\vec{\omega}$ denotes the solid angle over which is integrated and around which collisions are considered, respectively. As can be seen in Equation (3.9), the cross section σ is calculated with the solid angle of the collision and the relative speed of the two particles $|\vec{u_1} - \vec{u_2}|$. This equation furthermore includes Boltzmann's *closure assumption*, that leads to the aforementioned assumptions of a diluted gas with very localized, short-range interactions of the molecules :

$$f_{12} = \ f_1 \cdot f_2\ . \tag{3.10}$$

Using the *BBGKY hierarchy* (after Bogoliubov, Born, Green, Kirkwood and Yvon, see [Bogoliubov, 1962] for details) an approximation for f_N can be calculated, that can be used to get an equation for f_{12}. The problem is, that it includes a distribution function for three bodies f_{123}, which again depends on a four-body distribution function etc. However, the BBGKY hierarchy can be truncated to approximately calculate f_{12} this way.

In his famous *H-theorem* (named after the function $H(t)$, see below) Boltzmann showed in 1872, that the quantity

$$H(t) = -\int f\ \ln(f)\ d\vec{\xi}\ d\vec{x} \tag{3.11}$$

obeys

$$\frac{dH}{dt} \geq 0. \tag{3.12}$$

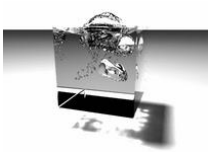
Here $f(\vec{x}, \vec{\xi}, t)$ is any function that fulfills the Boltzmann equation, furthermore the equality sign in Equation (3.12) only holds when f is a Maxwell distribution (see Equation (3.17) below). Although Boltzmann did not show that solutions to his theorem exist, it is celebrated for its connection of mechanics and thermo-dynamics, as well as its use as a bridge between micro- and macro-dynamics. Equation (3.12) can be imagined as a quantitative measure for the *irreversibility* from thermo-dynamics, whereas the H-theorem states that it is a monotonically increasing function in time. The evolution of the system only stops when the system has reached its global equilibrium with a maximal *entropy* H.

Due to the complicated nature of the collision operator Ω, it is often replaced by simpler expressions, that still preserve the collision invariants (these are explained in [Wolf-Gladrow, 2000]) and, as stated in the H-theorem, tend towards a Maxwellian distribution. The standard model for this is the *BGK approximation* that was independently proposed in [Bhatnagar et al., 1954] by as well as Welander [Welander, 1954]. It reads:

$$\Omega_{BGK}(f) = \frac{f^e - f}{\tau} \tag{3.13}$$

Here f^e is a Maxwellian distribution representing the *local equilibrium*, that is parametrized by the conserved quantities density ρ, speed $\vec{\xi}$ and temperature T. Each collision changes the distribution function f_1 proportional to the departure from the local equilibrium f^e, where the amount of this correction is modified by the relaxation time τ. The typical collision time τ is in principle depending on properties of the gas and it's current state, but for the BGK approximation simplified as a single value.

The local equilibrium is reached when $\Omega(f^e, f^e)$ vanishes. With this property it can be shown that f is a *collision invariant*, and as such does not change under the effect of a collision. The density ρ, momentum ξ_a and energy E are the Lagrangian parameters. Assuming a normalized



particle mass of 1, they can be computed in the following way:

$$\int f d\vec{\xi} = \rho \tag{3.14}$$

$$\int f u_a d\vec{\xi} = \rho \xi_a \tag{3.15}$$

$$\int f \frac{u^2}{2} d\vec{\xi} = \rho E \tag{3.16}$$

The macroscopic flow speed ξ_a, density ρ and fluid temperature T parametrize the Maxwell distribution (sometimes also called Maxwell-Boltzmann distribution). For three dimensions it is:

$$f^M = \rho \left(\frac{m^2}{2\pi RT} \right)^{3/2} e^{\frac{-(\vec{\xi}-\vec{u})^2 m^2}{2RT}} \tag{3.17}$$

Where R is the Boltzmann constant, and m the mass of a particle.

3.3 Derivation of the Lattice Boltzmann equation

The following section will explain the derivation of the Lattice Boltzmann equation from the continuous Boltzmann equation. It is based on [He and Luo, 1997] and the more detailed description in [Treibig, 2002]. The method described here allows the derivation of the Lattice Boltzmann equation from an arbitrary kinetic equation, although it historically emerged from the Lattice Gas cellular automata. The connection to the Navier-Stokes equations will be shown in the next section, where these equations will be derived from the Lattice Boltzmann equation via Chapman-Enskog expansion. The following abbreviations will be used from now on: $f(\vec{x}, \vec{\xi}, t) = f(t)$ and $f(\vec{x} + \vec{\xi}a, \vec{\xi}, t + a) = f(t + a)$. The same abbreviations hold for g.

As a starting point, the Boltzmann equation with BGK collision approximation will be used:

$$\frac{\partial f(t)}{\partial t} + \vec{\xi} \nabla f(t) = -\frac{1}{\lambda} \Big(f(t) - g(t) \Big) \tag{3.18}$$

where f is the particle distribution function at time t, position $\vec{x}$ for the microscopic velocity $\vec{\xi}$. $1/\lambda = A \cdot n$ is the relaxation time for the collision, that is calculated from the number of particles n and the proportional coefficient A. Here, the collision term has been linearized according to Equation 3.13 for simplicity, without loosing generality. g is the Maxwell distribution f^M from Equation 3.17.

The hydrodynamic properties of the fluid, the density ρ, velocity $\vec{u}$ and the temperature T can be calculated with the *moments* of the function f. Here, the energy ϵ from the energy density $\rho\epsilon$ can be used to determine the temperature of the fluid.

$$\rho = \int f(\vec{x}, \vec{\xi}, t) d\vec{\xi} \tag{3.19}$$

$$\rho\vec{u} = \int \xi f(\vec{x}, \vec{\xi}, t) d\vec{\xi} \tag{3.20}$$

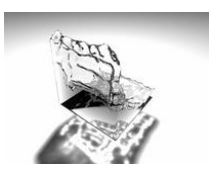
$$\rho\epsilon = \int \frac{1}{2} (\vec{\xi} - \vec{u}) f(\vec{x}, \vec{\xi}, t) d\vec{\xi} \tag{3.21}$$

Note that the equilibrium distribution function g is calculated with these hydrodynamic moments, although it is written as a function of time and velocity. Hence, these values have to be correctly approximated after discretization.

3.3.1 Time discretization

Equation 3.18 can be formulated as an ordinary differential equation (ODE):

$$\frac{Df}{Dt} + \frac{1}{\lambda} f = \frac{1}{\lambda} g \tag{3.22}$$



where

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \vec{\xi}\nabla \tag{3.23}$$

is the time derivative along the microscopic velocity. Equation 3.22 is a linear ODE of first order, hence, using a standard formula to solve this type of equation (see [Bronstein et al., 1999]), solutions can be found with:

$$f(t+\delta_t) = f(t) \cdot e^{-\frac{\delta_t}{\lambda}} + \frac{1}{\lambda} e^{-\frac{\delta_t}{\lambda}} \cdot \int_0^{\delta_t} e^{\frac{t'}{\lambda}} g(t+t') dt' \tag{3.24}$$

Assuming that δ_t is very small and g is a smooth function, $g(t+t')$ can be approximated with linear interpolation for $0 \leq t' \leq \delta_t$:

$$g(t+t') = (1 - \frac{t'}{\delta_t}) g(t) + \frac{t'}{\delta_t} g(t+\delta_t) + O(\delta_t^2) \tag{3.25}$$

This can be used to solve the integral from Equation 3.24.

$$\begin{aligned}
&\frac{1}{\lambda} e^{-\frac{\delta_t}{\lambda}} \int_0^{\delta_t} e^{\frac{t'}{\lambda}} g(t+t') dt' = \\
&\frac{1}{\lambda} e^{-\frac{\delta_t}{\lambda}} \int_0^{\delta_t} e^{\frac{t'}{\lambda}} (1 - \frac{t'}{\delta_t}) g(t) + \frac{t'}{\delta_t} g(t+\delta_t) dt' = \\
&\frac{1}{\lambda} e^{-\frac{\delta_t}{\lambda}} \left[e^{\frac{t'}{\lambda}} \lambda g(t) - e^{\frac{t'}{\lambda}} \frac{\frac{1}{\lambda} t' - 1}{\frac{1}{\lambda^2}} \frac{g(t)}{\delta_t} + e^{\frac{t'}{\lambda}} \frac{\frac{1}{\lambda} t' - 1}{\frac{1}{\lambda^2}} \frac{g(t+\delta_t)}{\delta_t} \right]_0^{\delta_t} = \\
&\frac{1}{\lambda} e^{-\frac{\delta_t}{\lambda}} g(t) \left[\lambda e^{\frac{\delta_t}{\lambda}} - (\lambda - \frac{\lambda^2}{\delta_t}) e^{\frac{\delta_t}{\lambda}} - \lambda - \frac{\lambda^2}{\delta - t} \right] + g(t+\delta_t) \left[(\lambda - \frac{\lambda^2}{\delta_t}) e^{\frac{\delta_t}{\lambda}} + \frac{\lambda^2}{\delta - t} \right] = \\
&g(t) - e^{-\frac{\delta_t}{\lambda}} g(t) + \left[1 + \frac{\lambda}{\delta_t} (e^{-\frac{\delta_t}{\lambda}} - 1) \right] [g(t+\delta_t) - g(t)]
\end{aligned} \tag{3.26}$$

With this, Equation 3.24 can be rewritten as:

$$f(t+\delta_t) - f(t) = \left(e^{-\frac{\delta_t}{\lambda}} - 1 \right) [f(t) - g(t)] + \left(1 + \frac{\lambda}{\delta_t} (e^{-\frac{\delta_t}{\lambda}} - 1) \right) [g(t+\delta_t) - g(t)] \tag{3.27}$$

Furthermore, $e^{-\frac{\delta_t}{\lambda}}$ can be Taylor expanded in δ_t in the following way:

$$e^{-\frac{\delta_t}{\lambda}} = e^{-\frac{0+\delta_t}{\lambda}} = 1 + \delta_t \left(-\frac{1}{\lambda} e^0 \right) + O(\delta_t^2) \approx 1 - \frac{\delta_t}{\lambda} \tag{3.28}$$

So Equation 3.27 simplifies to:

$$\begin{aligned}
f(t+\delta_t) - f(t) &= \left(1 - \frac{\delta_t}{\lambda} - 1 \right) [f(t) - g(t)] + \left(1 + \frac{\lambda}{\delta_t} (1 - \frac{\delta_t}{\lambda} - 1) \right) [g(t+\delta_t) - g(t)] \\
f(t+\delta_t) - f(t) &= -\frac{\delta_t}{\lambda} (f(t) - g(t))
\end{aligned} \tag{3.29}$$

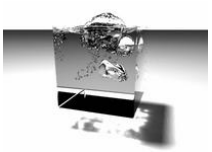
Here, the relaxation time $\frac{\delta_t}{\lambda}$ is usually written as $\frac{1}{\tau}$. This formula is already similar to Equation 2.3 from Chapter 2. What is still missing, is the discretization of the velocity space and an equilibrium function g that is consistent with the Navier-Stokes equations.

3.3.2 Approximation of the equilibrium distribution

The Maxwell distribution that is used as the equilibrium distribution function g was already explained in Section 3.2 (see Equation 3.17). For a particle mass of 1 and D dimensions it reads:

$$g(\vec{u}) = \frac{\rho}{(2\pi RT)^{D/2}} e^{-\frac{(\vec{\xi} - \vec{u})^2}{2RT}} \tag{3.30}$$

This function will be Taylor expanded in $\vec{u}$ up to the third order. For smaller velocities (or low Mach numbers) this approximation will be exact enough. Expanding the quadratic form in the exponent



of e and Taylor expanding the result yields the following equations (here $(\frac{\rho}{(2\pi RT)^{D/2}} e^{-\frac{\vec{\xi}^2}{2RT}})$ will be abbreviated by β) :

$$\begin{aligned} g(0+\vec{u}) &= \beta e^{\frac{\vec{\xi}\vec{u}}{RT} - \frac{\vec{u}^2}{2RT}} \\ &= g(0) + \vec{u}g'(0) + \frac{\vec{u}^2}{2}g''(0) + O(\vec{u}^3) \\ &= \beta \cdot \left\{ 1 + \vec{u} \cdot \left[\beta \frac{1}{RT} e^{\frac{\vec{\xi}\vec{u}}{RT} - \frac{\vec{u}^2}{2RT}} \cdot (\vec{\xi} - \vec{u}) \right] \right. \\ &\quad \left. + \frac{\vec{u}^2}{2}\beta \left[\frac{1}{(RT)^2} e^{\frac{\vec{\xi}\vec{u}}{RT} - \frac{\vec{u}^2}{2RT}} \cdot (\vec{\xi} - \vec{u})^2 + \frac{1}{RT} \cdot (-1) \right] \right\} + O(\vec{u}^3) \\ &= \beta \cdot \left(1 + \frac{\vec{\xi} \cdot \vec{u}}{RT} + \frac{(\vec{\xi} \cdot \vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT} \right) \end{aligned}$$

The terms of this last equation step are recognizable as those from Equation 2.1. For this formula the velocities (the phase space) is still not properly discretized. The following formula will be used as the local equilibrium distribution for the following derivations:

$$f^{(eq)} = \frac{\rho}{(2\pi RT)^{D/2}} e^{-\frac{\vec{\xi}^2}{2RT}} \left(1 + \frac{\vec{\xi} \cdot \vec{u}}{RT} + \frac{(\vec{\xi} \cdot \vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT} \right) \tag{3.31}$$

3.3.3 Discretization of the velocities

For simplicity, the D2Q9 model will be derived in the following section. As can be seen in Equation 3.19 the moment integrals over the whole velocity space are needed. As the velocity is not yet discretized, these run from $-\infty$ to $+\infty$ in both x- and y-direction for a two-dimensional model. The moments of the particle distribution functions are important for the consistency with the Navier-Stokes equations. Another important property that has to be retained by the discretization is the isotropy, which is the most important of the Navier-Stokes symmetries. So the lattice should be invariant to rotations of the problem – this can be shown by isotropy-tensors as in [Wolf-Gladrow, 2000]. But for the LBM derivation, the moments are directly used as constraint for the numerical integration method.

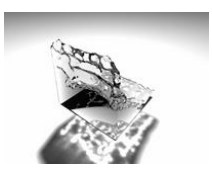
For accurate results, hence, the integrations of the second moment still has to be correct for models that include the temperature. As an isothermal model will be used, only the first moment, the velocity will be required. The moments of Equation 3.31 in two dimensions can generally be written as follows:

$$I = \int \psi(\vec{\xi}) f^{(0)} d\vec{\xi} = \frac{\rho}{(2\pi RT)^{D/2}} \int \psi(\vec{\xi})\ e^{-\frac{\vec{\xi}^2}{2RT}} \left(1 + \frac{\vec{\xi} \cdot \vec{u}}{RT} + \frac{(\vec{\xi} \cdot \vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT} \right) d\vec{\xi} \tag{3.32}$$

where ψ is the moment function, that contains powers of the velocity components

$$\psi(\vec{\xi}) = \xi_x^m \xi_y^n \tag{3.33}$$

This is necessary, as the moments equation above also contains powers of the velocity in the parentheses to the right. So after restructuring of the equation, moments of up to the third order will occur in the equation - one from the velocity moment, and two from the $(\vec{\xi} \cdot \vec{u})^2$ term. For numerical



treatment Equation 3.32 can be written in the following way:

$$\begin{aligned}
I &= \frac{\rho}{(2\pi RT)^{D/2}} \int \psi(\vec{\xi})\; e^{-\frac{\vec{\xi}^2}{2RT}} \left(1 + \frac{\vec{\xi}\cdot\vec{u}}{RT} + \frac{(\vec{\xi}\cdot\vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT}\right) d\vec{\xi} \\
&= \frac{\rho}{\pi}(\sqrt{2RT})^{-2} \int \xi_x^m \xi_y^n\; e^{-\frac{\vec{\xi}^2}{(\sqrt{2RT})^2}} \left(1 - \frac{\vec{u}^2}{(2RT)} + \frac{2\vec{\xi}\cdot\vec{u}}{(2RT)} + \frac{2(\vec{\xi}\cdot\vec{u})^2}{(\sqrt{2RT})^4}\right) d\vec{\xi} \\
&= \frac{\rho}{\pi}(\sqrt{2RT})^{-2} \cdot \Big(\int \xi_x^m \xi_y^n\; e^{-\frac{\xi_x^2+\xi_y^2}{(\sqrt{2RT})^2}}\,(1)\, d\vec{\xi} - \\
&\quad \int \xi_x^m \xi_y^n\; e^{-\frac{\xi_x^2+\xi_y^2}{(\sqrt{2RT})^2}} \left(\frac{u_x^2+u_y^2}{(2RT)}\right) d\vec{\xi} + \\
&\quad \int \xi_x^m \xi_y^n\; e^{-\frac{\xi_x^2+\xi_y^2}{(\sqrt{2RT})^2}} \left(\frac{2(\xi_x u_x + \xi_y u_y)}{(2RT)}\right) d\vec{\xi} + \\
&\quad \int \xi_x^m \xi_y^n\; e^{-\frac{\xi_x^2+\xi_y^2}{(\sqrt{2RT})^2}} \left(\frac{2(\xi_x^2 u_x^2 + 2\xi_x u_x \xi_y u_y + \xi_y^2 u_y^2)}{(\sqrt{2RT})^4}\right) d\vec{\xi}\;\Big)
\end{aligned} \tag{3.34}$$

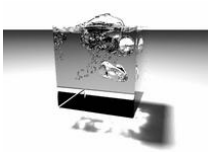
The next steps are necessary for each of the four integral-terms of Equation 3.34, but all proceed as shown for the first term:

$$\begin{aligned}
&\frac{\rho}{\pi}(\sqrt{2RT})^{-2} \cdot \left(\int \xi_x^m \xi_y^n\; e^{-\frac{\xi_x^2+\xi_y^2}{(\sqrt{2RT})^2}} d\vec{\xi} \right) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{-2} \cdot \left(\int\int e^{-\frac{\xi_x^2}{(\sqrt{2RT})^2}} \xi_x^m e^{-\frac{\xi_y^2}{(\sqrt{2RT})^2}} \xi_y^n\; d\xi_x\; d\xi_y \right) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{-2} \cdot \left(\int e^{-\frac{\xi_x^2}{(\sqrt{2RT})^2}} \xi_x^m\; d\xi_x \cdot \int e^{-\frac{\xi_y^2}{(\sqrt{2RT})^2}} \xi_y^n\; d\xi_y \right) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{-2} \cdot \Big((\sqrt{2RT})^m \int e^{-(\frac{\xi_x}{(\sqrt{2RT})})^2} (\frac{\xi_x}{(\sqrt{2RT})})^m\; d\frac{\xi_x}{(\sqrt{2RT})} \\
&\cdot (\sqrt{2RT})^n \int e^{-(\frac{\xi_y}{(\sqrt{2RT})})^2} (\frac{\xi_y}{(\sqrt{2RT})})^n\; d\frac{\xi_y}{(\sqrt{2RT})}) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{m+n-2} \cdot \Big(\int e^{-(\frac{\xi_x}{(\sqrt{2RT})})^2} (\frac{\xi_x}{(\sqrt{2RT})})^m\; d\frac{\xi_x}{(\sqrt{2RT})} \\
&\cdot \int e^{-(\frac{\xi_y}{(\sqrt{2RT})})^2} (\frac{\xi_y}{(\sqrt{2RT})})^n\; d\frac{\xi_y}{(\sqrt{2RT})}) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{m+n-2} \cdot \left(\int e^{-\zeta_x^2} \zeta_x^m\; d\zeta_x \;\cdot \int e^{-\zeta_y^2} \zeta_y^n\; d\zeta_y \right) \\
= \;& \frac{\rho}{\pi}(\sqrt{2RT})^{m+n-2} I_x^m I_y^n
\end{aligned} \tag{3.35}$$

It can be seen that I_i^m is the m-th moment of the function $e^{-\zeta^2}$. Equation 3.34 can be rewritten using these moments:

$$\begin{aligned}
I &= \frac{\rho}{\pi}(\sqrt{2RT})^{m+n-2} \Bigg[\left(1 - \frac{\vec{u}^2}{(2RT)}\right) I_x^m I_y^n + \\
&\quad \frac{2(u_x I_x^{m+1} I_y^n + u_y I_x^m I_y^{n+1})}{(\sqrt{2RT})} + \\
&\quad \frac{u_x^2 I_x^{m+2} I_y^n + 2u_x u_y I_x^{m+1} I_y^{n+1} + u_y^2 I_x^m I_y^{n+2}}{RT} \Bigg]
\end{aligned} \tag{3.36}$$

The crucial step for the derivation of the LBM is to use the proper quadrature formula to numerically



integrate these moments. The quadrature works in the following way:

$$\int f(x)W(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{3.37}$$

Where $W(x)$ is the weighting function, e^{-x^2} in this case, and $f(x)$ is a polynomial in x, e.g. $f(\zeta_x) = \zeta_x^m$. The integral over the multiplication of these two functions is approximated by the sum over function evaluations at the values $x = x_j$ with the weights w_j. The values x_i are also called abscissas. In total, there are N abscissas and weights. For functions like $e^{-\zeta^2}$ Gauss-Hermite quadrature can be applied, which is correct for W-polynomials up to the order $(2N-1)$. The order of the Gauss-Hermite quadrature has to be chosen according to the order of the moment-polynomial ψ. Although the model is isothermal the energy due to the temperature has to be kept constant. So there is no additional level of freedom for the temperature, but for the moment integration it has to be considered. Hence, for a ψ of second order, and second order terms in the $(\vec{\xi}\cdot\vec{u})$ term, moments of up to fourth order have to be integrated correctly. This requires an Gauss-Hermite quadrature of third order $(N = 3)$:

$$I_i^m = \sum_{j=1}^{3} w_j(\zeta_j)^m \tag{3.38}$$

with the following weights and abscissas:

$$\zeta_1 = -\sqrt{3/2}, \quad \zeta_2 = 0, \quad \zeta_3 = +\sqrt{3/2} \tag{3.39}$$

$$w_1 = \frac{\sqrt{\pi}}{6}, \quad w_2 = \frac{2\sqrt{\pi}}{3}, \quad w_3 = \frac{\sqrt{\pi}}{6} \tag{3.40}$$

Having applied the Gauss-Hermite quadrature, the moment function can be again shortened to the following form:

$$I = \frac{\rho}{\pi}\sum_{i=1}^{3}\sum_{j=1}^{3} w_i w_j \psi(\zeta_{i,j})\left(1 + \frac{\vec{\xi}\cdot\vec{u}}{RT} + \frac{(\vec{\xi}\cdot\vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT}\right) \tag{3.41}$$

where $\zeta_{i,j}$ is the vector given by the quadrature abscissas $\zeta_{i,j} = (\sqrt{2RT})(\zeta_i, \zeta_j)^T$. As the two sums run over three values for i and j each, there are a total of nine possible values for $\zeta_{i,j}$ and $w_i w_j$. For these a new single index will be introduced. Furthermore, a number of substitutions can be made. As an isothermal model is used, the temperature T has no physical relevance, and can be replaced by a constant $c = \sqrt{3RT}$. The speed of sound $c_s = 1/\sqrt{3}$ in the model yields $c_s^2 = c^2/3 = RT$. The weights, divided by π read:

$$\begin{aligned} w_0 &= w_2 w_2 = 4/9 \\ w_{1..4} &= w_1 w_2, w_2 w_1, w_3 w_2, w_2 w_3 = 1/9 \\ w_{5..8} &= w_1 w_3, w_3 w_1, w_1 w_1, w_3 w_3 = 1/36 \end{aligned} \tag{3.42}$$

Each component of the vectors $\zeta_{i,j}$ is either 0 or $\pm\sqrt{2RT}\sqrt{3/2} = \pm\sqrt{3RT} = c$:

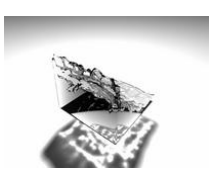
$$\begin{aligned} \vec{e}_0 &= \zeta_{1,1} = (0,0)^T \\ \vec{e}_{1..4} &= \zeta_{1,2}, \zeta_{2,1}, \zeta_{3,2}, \zeta_{2,3}, = (\pm 1, 0)^T c, (0, \pm 1)^T c \\ \vec{e}_{5..8} &= \zeta_{1,3}, \zeta_{3,1}, \zeta_{1,1}, \zeta_{3,3}, = (\pm 1, \pm 1)^T c \end{aligned} \tag{3.43}$$

With these discrete velocities, Equation 3.41 reads:

$$I = \sum_{\alpha=1}^{9} W_\alpha \psi(\vec{e}_\alpha) f_\alpha^{eq} \tag{3.44}$$

Here W_α can be identified as $2\pi RT e^{\frac{\vec{\xi}^2}{2RT}}$. This yields the known form of the equilibrium distribution function for each of the nine velocities:

$$f_\alpha^{eq} = w_\alpha \rho\left(1 + \frac{3\vec{e}\cdot\vec{u}}{c^2} + \frac{9(\vec{e}\cdot\vec{u})^2}{2c^4} - \frac{3\vec{u}^2}{2c^2}\right) \tag{3.45}$$



Note that the lattice velocity vectors were given by the chosen Gauss-Hermite quadrature. The configuration of the lattice was in this case also obtained from these velocities. It is possible to discretize velocities and lattice configuration differently, as has been shown in [He and Luo, 1996] and [Bouzidi et al., 2001].

Other LBM models like the D3Q27 model can be derived in the same way. For the more often used three-dimensional models D3Q15 and D3Q19, however, it was not possible to apply this method. Problems arise from the more irregular arrangement of the velocity vectors, that cannot be easily formulated as a quadrature term. For these models the *ansatz method* has to be used [Wolf-Gladrow, 2000]. For a given kinetic equation like Equation 3.18 together with an equilibrium distribution as the one from Equation 3.31 the velocity weights for a specific lattice can be calculated. With multi-scale analysis constraints for the moments of f can be set up, and solved to yield the required coefficients.

3.4 Chapman-Enskog Expansion

To show that the Boltzmann equation can be used to describe fluids, the Navier-Stokes equations are derived by a procedure called *Chapman-Enskog expansion*, or *multi-scale analysis*. It relys on the *Knudsen number* C_{Kn}, which is the ratio between the *mean free path length* λ and the characteristic shortest scale of the macroscopic system that needs to be considered (L_C). Hence, λ is the mean length a particle travels between two collisions, while L_C contains for example the size of an obstacle in the fluid. The Knudsen number has to be less than one, for the treatment of the fluid as a coninuous system.

$$C_{Kn} = \frac{\lambda}{L_C} \tag{3.46}$$

For the derivation, a splitting of the Boltzmann equation into different scales for space and time variables (or a hierarchy of these) is performed. It is based on the epxansion parameter ϵ for which the Knudsen number C_{Kn} will be used. The expansion is done in a way, that each scale models a process of interest. Usually the expansion is truncated after terms of second order. For the time variables, the following representation is chosen:

$$t = \epsilon t_1 + \epsilon^2 t_2. \tag{3.47}$$

The time t represents the very fast local relaxations in a fluid by collisions. Sound waves, as well as advection, are of the scale t_1, and considerably slower than the local relaxations. Still, these are faster than diffusion processes, that are of time scale t_2. Only one spatial expansion has to be considered, giving the following expansion of first order:

$$\vec{x} = \epsilon \vec{x}_1. \tag{3.48}$$

This is due to the fact, that advection and diffusion both are considered in similar spatial scales x_1. The representation of the differential operators is similar:

$$\frac{\partial}{\partial x_a} = \epsilon \frac{\partial}{\partial x_a} \tag{3.49}$$

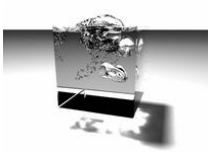
$$\frac{\partial}{\partial t} = \epsilon \frac{\partial}{\partial t} + \epsilon^2 \frac{\partial}{\partial t} \tag{3.50}$$

For a consistent expansion, the second order terms in space are also necessary. The moment equations of f are directly expanded to a sum of the form:

$$f = \sum_{n=0}^{\infty} \epsilon^n f^n \tag{3.51}$$

Furthermore, it is assumed that the time dependance of f is only caused by the variables $\rho, \vec{u}$ and T. Expanding Equation (3.8) in both space and time up to second order yields the following equation:

$$\epsilon \frac{\partial f}{\partial t_1} + \epsilon^2 \frac{\partial f}{\partial t_2} + \epsilon u_a \cdot \frac{\partial f}{\partial x_a} + \frac{1}{2}\epsilon^2 u_a u_b \frac{\partial^2 f}{\partial x_a \partial x_b} = \Omega(f^0) + \epsilon \frac{\partial \Omega(f^1)}{\partial f} \tag{3.52}$$



Note that f^0 is a Maxwell distribution, and as such, due to the definition of the BGK collision approximation in Equation (3.13), $\Omega(f^0)$ is zero. The three scales from $\mathcal{O}(\epsilon^0)$ to $\mathcal{O}(\epsilon^2)$ can be distinguished in Equatioen 3.52, and are seperately handled. From there on, subsequent expansions of the conservation equations can be performed. Expanding mass up to second order gives an equation of the form:

$$\epsilon\hat{m}_1 + \epsilon^2\hat{m}_2 = 0 \tag{3.53}$$

As ϵ is the Knudsen number, it is necessarily larger than zero. So are the terms $\hat{m_1}$ and $\hat{m_2}$ that represent the expansions for first and second order, respectively [Harris, 1971]. For first order terms of Equation (3.52), using a mass and momentum of zero, results in the following two equations:

$$\frac{\partial\rho}{\partial t_1} + \frac{\rho\partial u_a}{\partial x_{1a}} = 0 \tag{3.54}$$

$$\frac{\partial\rho u_a}{\partial t_1} + \frac{\partial\int u_a u_b f^0 d\vec{u}}{\partial x_{1b}} = 0 \tag{3.55}$$

The continuity equation is already recognizable. When the integral of the second equation is analytically performed, it can be replaced by $\rho u_a u_b + \rho T\delta_{ab}$:

$$\frac{\partial\rho u_a}{\partial t_1} + \frac{\partial\rho u_a u_b}{\partial x_{1b}} + \frac{\partial\rho T\delta_{ab}}{\partial x_{1b}} = 0 \tag{3.56}$$

This can be recognized as the Euler equation for inviscid flows without dissipation. To get the Navier-Stokes equations from here, the second order equations have to be considered. These are much more complicated to handle, as both equilibrium and non-equilibrium levels are needed. Still, using first order conservation terms of zero, and restoring the continuous form of the equations, the Navier-Stokes equations as in Equation (3.2) emerge. This is possible as terms of $\mathcal{O}(u^3)$ can be neglected, due to the assumption of small velocities for the expansion. The expansion also yields the equation for the calculation of the viscosity from the LBM parameters. For more details refer to [Harris, 1971; Wolf-Gladrow, 2000].

3.5 Further enhancements

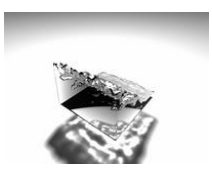
The LBGK model of the previous sections can cause problems due to the relaxed incompressibility constraint. When considering a constant mass flow for an incompressible fluid, a reduced density would lead to a higher velocity. This is physically incorrect, and leads to wrong results. The incompressibility can be enhanced by substituting the density in Equation 3.45 with $\rho = \rho_0 + \delta\rho$. For smaller Mach numbers M (which means that $\vec{u}/c_s \to 0$) the changes in density are of order $\mathcal{O}(M^2)$. Moreover, as $\vec{u}$ is the second moment of f it is also of $\mathcal{O}(M^2)$. After the aforementioned substitution of the density, terms like $\delta\rho(\vec{u}/c_s)$ and $\delta\rho(\vec{u}/c_s)^2$ are of order $\mathcal{O}(M^3)$ and $\mathcal{O}(M^4)$, respecitvely, hence they are negligible. After removing these terms, the Equation 3.45 has the following form for each of the nine velocities:

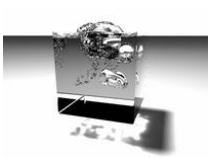
$$f_\alpha^{eq} = w_\alpha\left(\delta\rho + \rho_0\frac{3\vec{e}\cdot\vec{u}}{c^2} + \frac{9(\vec{e}\cdot\vec{u})^2}{2c^4} - \frac{3\vec{u}^2}{2c^2}\right) \tag{3.57}$$

Here ρ_o can again be normalized to 1, and the density computed from Equation 3.19 is taken as $\delta\rho$. Thus, the main difference is, that the $\vec{u}$-terms are not multiplied with ρ anymore. Note that, as ρ_o equals 1, the velocity $\vec{u}$ is not divided by ρ anymore – so the implementation requires less floating point divisions than the original method.

Another way to stabilize the Lattice Boltzmann method, is to use the *moment method* [d'Humières, 1992]. In this case the distribution functions of the LBM are transformed into velocity moments. Of these, only a few (three for the D2Q9 model – the density and the to momentum velocities) have to be relaxed with τ, while the relaxation parameter can be chosen from a wider range for the other moments. [d'Humières et al., 2002; Lallemand and Luo, 2000] The resulting equations are only slightly more complicated than the normal LBGK equations, and have an enhanced stability, that allows simulations of high Reynolds numbers even on coarse grids.

For the implementation explained in Chapter 5 the incompressible LBGK model was chosen, as it's stability is sufficient for the targeted problems.





Chapter 4

The model

The model used in this simulation will be described in this chapter. It uses a D3Q19 LBM with a free boundary between fluid and gas phase and surface tension forces. The D3Q19 lattice is known from Chapter 2. The topology representation, boundary conditions for the free surface and the surface tension will be explained in more detail in the following sections.

4.1 The single-phase free-surface model

Other multi-phase models, like the gradient approach in [Shan and Chen, 1993] or the free energy methods [Swift et al., 1996] simulate all phases with separate particle distribution functions for each phase. As a foam like structure mainly consists of gas, with only thin membranes of fluid between the bubbles, it would be a huge task to also simulate the gas motion in all bubbles, although it's effect on the foam structure is negligible. So for this application, the simulation of metal foams, it was decided to not include the gas phase in the simulation. The different phases only use a single lattice, and are distinguished by flags for each cell in the grid, that are also used to identify for example the no-slip and free-slip boundary cells. The topology of the problem is described with the following three types of cells:

- filled with fluid: this cell is completely filled by the fluid, and treated as explained in Chapter 2.
- interface: these cells contain both liquid and gas. Care has to be taken during streaming.
- gas: these cells are not considered in the fluid simulation.

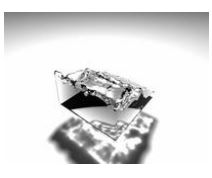
Hence, the complicated part of this model deals with the interface cells. To prevent large gradients and mixing of fluid and gas, it is required that there is only a single layer of interface cells around the fluid cells. This is shown in Figure 4.1.

For each interface cell the mass of the liquid in this cell is tracked. This cell mass is used to compute the fill level of a cell by dividing by the density of this cell. So the mass is in the range of 0 (cell is empty) and ρ (cell is completely filled with fluid). In these two cases, minimal and maximal fill level of the cell, the type of the cell is changed - empty interface cells become gas cells, completely filled interface cells become fluid cells. When this happens, the surrounding empty or fluid cells have to be initialized as interface cells accordingly. This is important, as the model described here does not work for adjacent fluid and gas cells. This would result in a loss of mass, invalidating the simulation.

The new mass $m(\vec{x})$ of each interface cell is calculated after the collide step as the difference between the mass, or number of particles, leaving and entering the cell:

$$m(\vec{x}) = m(\vec{x}) + \sum_{l=1}^{19} \epsilon_I f_I(\vec{x} + \vec{e_i}) - \epsilon_i f_i(\vec{x}) \tag{4.1}$$

where f_I is the distribution function that points into the opposite direction of f_i. The ϵ values represent the current fill level of the corresponding cells. They account for the area of the interface between the two adjacent cells. This change of mass is calculated during the stream step. The streaming already performs this mass transport correctly for normal fluid cells, but it cannot work



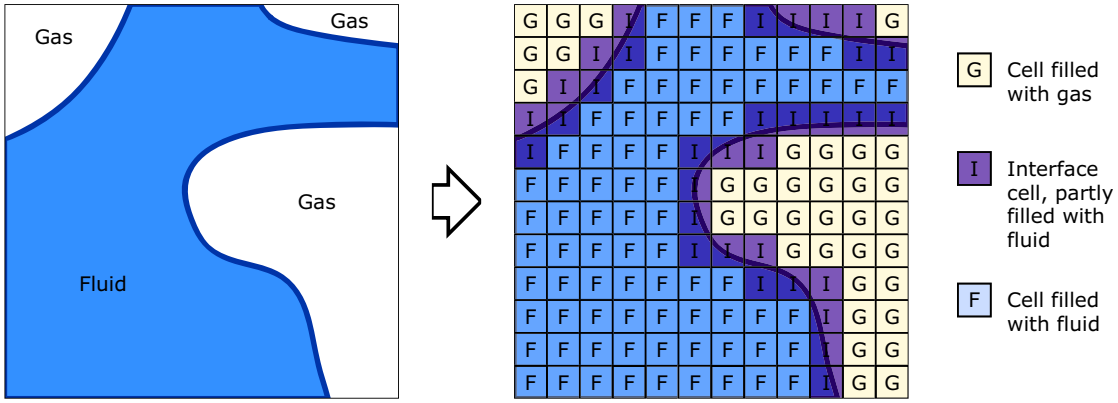


Figure 4.1: Each LBM cell is either a fluid, interface or gas cell. A fluid configuration is subdivided into cells, which are completely, partly and not filled with fluid.

for interface cells, as these have neighboring cells that do not contain fluid. So for all interface cells in all directions where the neighboring cells contain fluid (either interface or fluid cells) the change of mass is updated with Equation 4.1.

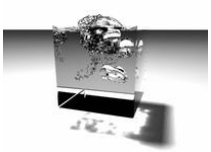
It is assumed, that the fluid is much less viscous than the gas surrounding it. For example air has a dynamic viscosity of ca. $1.5 \cdot 10^{-5} m^2/s$, while that of water is ca. $9 \cdot 10^{-7} m^s/s$. So the fluid has a much higher Reynolds number than the gas. In this case the effects of the gas phase on fluid motion can be neglected, as the gas reaches the local equilibrium much faster and at the interface just moves the same way the fluid does. This assumption is needed to reconstruct the distribution functions that are needed from gas cell in the neighborhood of interface cells. This is done in the following way:

$$f_I(\vec{x}, t+1) = f_i^{(0)}(\vec{u}_B, \rho_B) + f_I^{(0)}(\vec{u}_B, \rho_B) - f_i(\vec{x}, t) \tag{4.2}$$

where f_i is again the particle distribution function that is opposite of the distribution function f_I. $f_i^{(0)}$ is just the local equilibrium with the velocity of the fluid at the bubble interface. The two parameters of $f^{(0)}$ are known, as the velocity is just that of the interface cell and the density is known for bubbles (see Section 4.2) and for the atmosphere ($\rho_B = 1$). This directly corresponds to the assumption of a negligible effect from the gas phase upon the fluid phase, as the gas phase is assumed to reach the local equilibrium in each time step that is simulated for the fluid. So the distribution function that would be streamed from a gas cell is determined by the two opposite equilibrium distribution functions for the gas and the motion of the fluid (given in this direction by f_i). This formula can be expanded to include surface tension and pressure, as will be explained in the following sections.

4.2 Bubbles

The only important effect of bubbles in the fluid is the pressure upon the fluid surface that changes when the bubble is compressed or expands. For this, the mass of the gas in the bubble has to be calculated (upon initialization at the moment a bubble forms), and by tracking the volume of the bubble, the pressure in each time step can be calculated. This requires that each empty and interface cell is identified with a bubble, so additionally a bubble ID has to be stored for each cell. The pressure is a surface force that is applied to the interface cells in the stream step by scaling the distribution functions from gas cells according to the pressure. This, for ρ_B from Equation 4.2, the pressure that results from the bubble at $\vec{x}$, can be taken. It is given by the bubble ID stored in the corresponding cell. The pressure force usually applies an acceleration normal to the surface depending on the surface area. This is approximated by the reconstruction mechanism. The pressure acts for each distribution function streamed from an empty cell, which in total approximates the normal direction.



Coalescence is done by merging two bubbles that touch each other. This case occurs when to adjacent gas cells exist, that have different bubble IDs. A new bubble with the combined volume and mass of the two touching bubbles is created. The bubble IDs of all cells from the old two bubbles have to be changed accordingly.

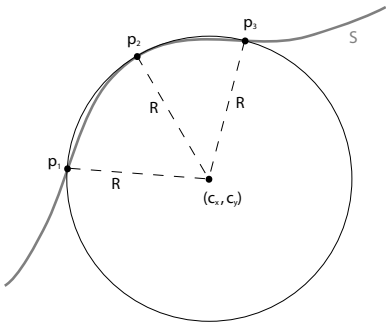
For realistic simulation of bubbles, an identification of gas regions surrounded by fluid is required. This partitioning is not implemented in the current version of the program. The bubbles have to be defined at the beginning of the simulation, during which the total number of bubbles only decreases due to coalescence.

4.3 Surface tension

The surface tension, like the pressure, is a surface force that is added in the reconstruction term:

$$f_I(\vec{x}, t+1) = \left(f_i^{(0)}(\vec{u}_B, \rho_B) + f_I^{(0)}(\vec{u}_B, \rho_B)\right) \cdot \sigma - f_i(\vec{x}, t) \tag{4.3}$$

where σ is a factor that is influenced by the curvature of the surface at $\vec{x}$ and the strength of the surface tension of the fluid. The curvature can be computed in the following way. In two dimensions, the curvature of a line can be computed by reconstructing a circle through three points that lie on the surface. Given three points $\vec{p}_0, \vec{p}_1$ and $\vec{p}_2$ the center of the circle $(c_x, c_y)^T$ that touches all three points is:



$$\begin{aligned}
d_{1x} &= p_{1x} - p_{0x} \\
d_{2x} &= p_{2x} - p_{0x} \\
d_{1y} &= p_{1y} - p_{0y} \\
d_{2y} &= p_{2y} - p_{0y} \\
a_0 &= p_{1x}^2 + p_{1y}^2 - p_{0x}^2 - p_{0y}^2 \\
a_1 &= p_{2x}^2 + p_{2y}^2 - p_{0x}^2 - p_{0y}^2 \\
d &= d_{1x}d_{2y} - d_{2x}d_{1y} \\
c_x &= (a_0 d_{2y} - a_1 d_{1y})/2d \\
c_y &= (a_1 d_{1x} - a_0 d_{2x})/2d
\end{aligned} \tag{4.4}$$

The curvature at one of the three points is then one divided by the radius of the circle:

$$\sigma = \frac{1}{R}, \quad R = |\vec{p} - (x_c, y_c)^T|_2 \tag{4.5}$$

The calculation of the curvature at a point $\vec{p}$ of a surface in three dimensions is more complicated. The curvature can be calculated for all curves that pass through the point on the surface (see Figure 4.2a). For each curve the curvature can be calculated with Equation 4.4 and Equation 4.5, but usually curvatures from normal sections are desired. This requires the normal of the curve $\vec{n}_C$ to be equal to the normal of the surface $\vec{N}$ in the point $\vec{p}$. For a given curve C which goes through $\vec{p}$, the curvature σ_C can be transformed into the curvature σ_N of the normal section through $\vec{p}$ by the formula of *Meusnier*:

$$\sigma_N = \frac{\sigma_C}{cos(\vec{N}, \vec{n}_C)} \tag{4.6}$$

Usually, the principal curvatures are of interest. These are the minimum and maximum of the curvatures through all possible curves in normal sections through $\vec{p}$. These two normal sections are perpendicular to each other, and unique for each surface point. For the simulation in this thesis, the principal curvatures were not calculated (see [Bronstein et al., 1999] for details), due to the higher computational complexity. Two curvatures along axis aligned planes are calculated (see Section 5.3 for details), and the mean curvature of these two is used to scale the distribution functions from Equation 4.3.

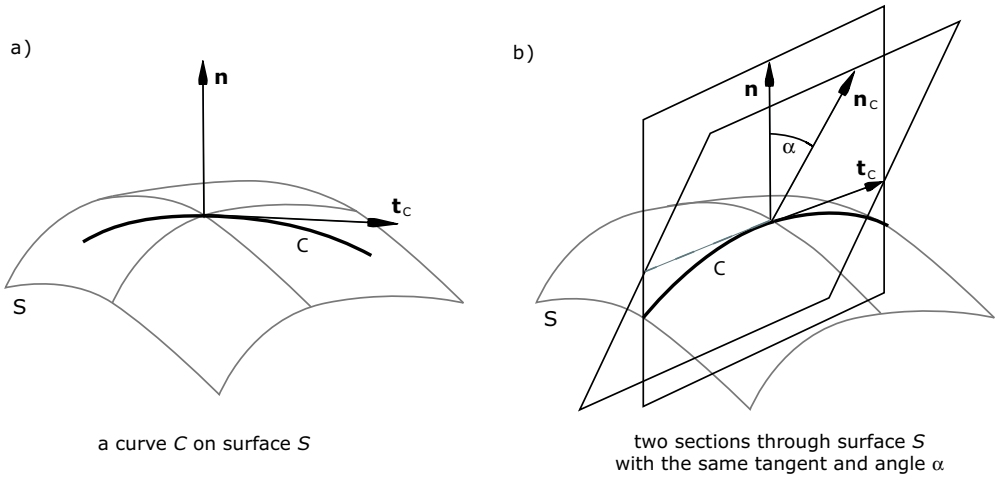


Figure 4.2: The curvature can be calculated along any curve C on the surface S, as shown to on the left. To the right, two sections through S can be seen, one containing the normal of the surface $\vec{n}$, while the other is rotated by the angle α.

4.4 Gravity

External forces like gravity can be incorporated into the LBM in various ways (see [Buick and Greated, 2000]). The easiest way, that was explained in Chapter 2, is to modify the calculated velocity in each cell in the collision step. Thus, the local equilibrium is calculated with a modified velocity $\vec{u}^*$ that is given by:

$$\vec{u}^* = \vec{u} + \tau\vec{F} \tag{4.7}$$

where $\vec{F}$ is the external force, and τ the LBM relaxation time. Care has to be taken when calculating the velocity in each cell for e.g. particle tracing. It is now defined as the average velocity before and after the collision $\vec{u} + \frac{1}{2}\tau\vec{F}$.

Another possibility is to modify the equilibrium functions. For each calculation with Equation 2.1 the particle distribution function is modified by the contribution of the force along this direction. A combination of both methods is also possible. As the different methods did not significanlty change the simulation, the first one was chosen, as it requires less floating point operations than the second and the combined method.

4.5 LBM Parametrization

While all important aspects of the Lattice Boltzmann Method were described in the previous sections, it is not obvious how to parametrize a simulation to represent a real fluid. The presented method is based upon the parametrization of the metal foam simulations from [Körner et. al].

4.5.1 Parameter calculation

To specify a given problem, the following values have to be known:

- the viscosity of the fluid ν $[\frac{\mathrm{m}^2}{\mathrm{s}}]$
- the surface tension σ $[\frac{\mathrm{N}}{\mathrm{m}}]$
- the strength of the external force (e.g. gravity) g $[\frac{\mathrm{m}}{\mathrm{s}^2}]$
- the density of the fluid ρ $[\frac{\mathrm{kg}}{\mathrm{m}^3}]$
- the size of a LBM cell Δx [m]

These will be used to set a reasonable time step size Δt [s] and limit the mass difference in the simulation Δm [kg]. Generally, two different sets of values have to be distinguished: the one mentioned above, which contains the physical values, and another dimensionless one, which contains

the lattice values. The corresponding dimensionless lattice values can be calculated in the following way:

- lattice viscosity $\nu^* = \nu \cdot \frac{\Delta t}{\Delta x^2}$
- lattice surface tension $\sigma^* = \sigma \cdot \frac{\Delta t^2}{\Delta m}$
- lattice force $g^* = g \cdot \frac{\Delta t^2}{\Delta x}$
- lattice density $\rho^* = \rho \cdot \frac{\Delta x^3}{\Delta m}$

The basis for the parameter calculation is the following equation, which can be used to calculate the dynamic viscosity ν of a fluid that is simulated with given the LBM parameters:

$$\nu^* = c_s^2 \left(\tau - \frac{1}{2}\right) = \frac{2\tau - 1}{6} \tag{4.8}$$

Here $\tau = 1/\omega$ is the relaxation time. As usually the fluid viscosity ν is given, this formula can be used to calculate the relaxation time needed for a simulation. Trying this, it will be noticed that the LBM only yields valid results for τ within certain limits. Especially when τ is close to $\frac{1}{2}$, the simulation can quickly become instable. For this thesis the following limits were used: $0.51 \leq \tau \leq 2.5$. Hence, Equation 4.8 can be used to restrict τ in the following way:

$$\tau = \frac{6 \cdot \nu^* + 1}{2} \quad \Rightarrow \quad 0.67 \leq \nu^* \leq 3.3 \cdot 10^{-3} \tag{4.9}$$

The time scale can be calculated by imposing a restriction on the compressibility for the simulation. A limit of 10^{-4} will be used here, but depending on the problem, this value might be smaller for stronger forces acting upon the fluid. The upper limit for the length of a time step is then calculated by:

$$\begin{aligned} g \cdot \frac{\Delta t^2}{\Delta x} &\leq 10^{-4} \\ \Delta t &\leq \sqrt{\frac{10^{-4} \cdot \Delta x}{g}} \end{aligned} \tag{4.10}$$

The lattice viscosity ν^* depends upon the size of the time step, so it is necessary to check whether ν^* is in the boundaries given by Equation 4.9. If this is the case, the relaxation time τ can be calculated.

The density of the fluid only affects the surface tension, as the lattice density is always set one $\rho^* = 1$. Now the mass scale can be calculated as:

$$\Delta m = \frac{\rho}{\rho^*} \cdot \Delta x^3 \tag{4.11}$$

It is needed to calculate the lattice surface tension, as described above, $\sigma^* = \sigma \cdot \frac{\Delta t^2}{\Delta m}$. Now all LBM parameters are given, and the simulation can be run. The size of the computational domain is then given by the number of LBM cells that are used for the simulation.

Note that the speed of sound, which is often mentioned in theory of the LBM, does not correspond to the physical speed of sound of the simulated fluid. It just represents the speed with which information is transported through the grid. A realistic speed of sound, for example $1500 \frac{\mathrm{m}}{\mathrm{s}}$ in water, would result in a very small time step length and thus necessitate a huge amount of time steps and computations for a simulation.

4.5.2 An example problem

Water plays an important role in everyday life, and as such most model problems in this thesis were targeted to be comparable to known situations. Furthermore the liquid aluminium used to create metal foams has a similar viscosity as water, hence the experiments can also be used to simulate model problems that are relevant for foaming simulations. For simulating water the following values were used:

- viscosity $\nu = 10^{-6}\frac{m^2}{\mathrm{s}}$
- surface tension $\sigma = 7 \cdot 10^{-4}\frac{\mathrm{kg}}{\mathrm{s}^2}$
- gravitational constant $g = 9.81\frac{\mathrm{m}}{\mathrm{s}}$
- density $\rho = 1000\frac{\mathrm{kg}}{\mathrm{m}^3}$

As length scale, e.g. $\Delta x = 100\mu\mathrm{m} = 0.0001\mathrm{m}$ can be used. The earth gravity results in a time step length of $\Delta t = 3 \cdot 10^{-5}\mathrm{s}$. When checking this for the limits of Equation 4.9, the time step is slightly smaller than the equation allows, but the resulting $\tau = 5.09$ was still large enough for the implementation of this thesis. The resulting surface tension is 0.063, which is relatively high for the simulation, but as the spatial scale is very small, it is realistic.

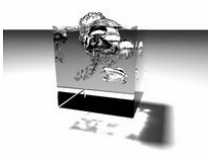
Assuming a grid size of 100^3, this would correspond to a cube filled with water having a side length of 1cm. The simulation would need roughly 300MB of memory, and 33.000 time steps are required to simulate one second.

For the simulation of metal foams on the SR8000, the limiting factors are memory and performance requirements. There are a total of 168 nodes, with roughly 6.5GB of available memory each. This allows the simulation of domains with up to 250^3 LBM cells on each node. Assuming a simple and regular domain decomposition for 5^3 nodes, the whole domain would be of size 1250^3 or $12.5\mathrm{cm}^3$. This would be sufficient to simulate even complex foaming processes.

4.5.3 Problem scales

A standard computer, with 1GB of RAM and a CPU of at least 2GHz allows the simulation of roughly 120^3 LBM cells. With the calculations above this would correspond to only $1.2\mathrm{cm}^3$. As it is desirable to simulate larger regions of fluid even with desktop computers, the first idea that comes to mind is to choose a larger spatial scale, e.g. $\Delta x = 1cm$. The problem with this is, that the lattice viscosity decreases, and the simulation quickly becomes instable.

Another possibility is to change the parameters as described in Section 3.1. The Reynolds number remains constant if the characteristic length is decreased, and the flow speed increased by the same factor. A smaller spatial scale would result in a more stable simulation, and is as such desirable. In this case another LBM restriction becomes problematic. The speed in each cell should not be larger than roughly 0.1. This is, however, easily the case when e.g. using the gravity of the earth. The only way to overcome these problems would be to use a different discretization of the Lattice Boltzmann equation. These have been proposed by e.g. [Coa et al., 1997], and allow more flexible time steps and spatial discretizations. However, they also require much more computational overhead, and as the velocities of a foaming process are small enough, these methods were not implemented in this thesis.



Chapter 5

Implementation

The single-phase free-surface lattice Boltzmann method described in the previous chapters of this thesis was implemented in C/C++. For performance reasons, the program was not designed in a fully object-oriented way, as virtual functions and irregular data layouts should be avoided in high performance codes. The relevant details of the model implementation are explained in this chapter.

5.1 Performance considerations

For a standard LBM implementation the memory bandwidth of the used platform will be the limiting factor. Each cell requires at least 19 floating point values, that will usually be stored with double precision requiring 8 bytes each. Furthermore, the complete grid is required twice. Hence, a current Intel Pentium 4 Processor with 512KB Level 2 Cache will not be able to fit even a 12^3 grid into cache. As has been shown in [Wilke, 2002] there are many different ways of implementing a lattice Boltzmann method, that differ significantly in performance. Although the variants in [Wilke, 2002] are also applicable for 3D LBM, the calculation of surface tension limits the selection. After every simulation step, a surface reconstruction is necessary (this will be explained in more detail in Section 5.3). This would make very large stencils and offsets necessary if multiple time steps were to be blocked. Hence, for now the implementation does not block multiple steps.

Generally, blocking is difficult due to the high memory requirements and limited reuse of the calculations. For the data layout, there are several possibilities, shown in Figure 5.1 (most illustrations in this chapter will show two-dimensional problems, but are directly transferable to 3D). As all 19 distribution functions are usually modified simultaneously, the spatial locality can be increased by not storing the distribution functions in $19 \cdot 2$ separate arrays, but storing 2 arrays with sets of

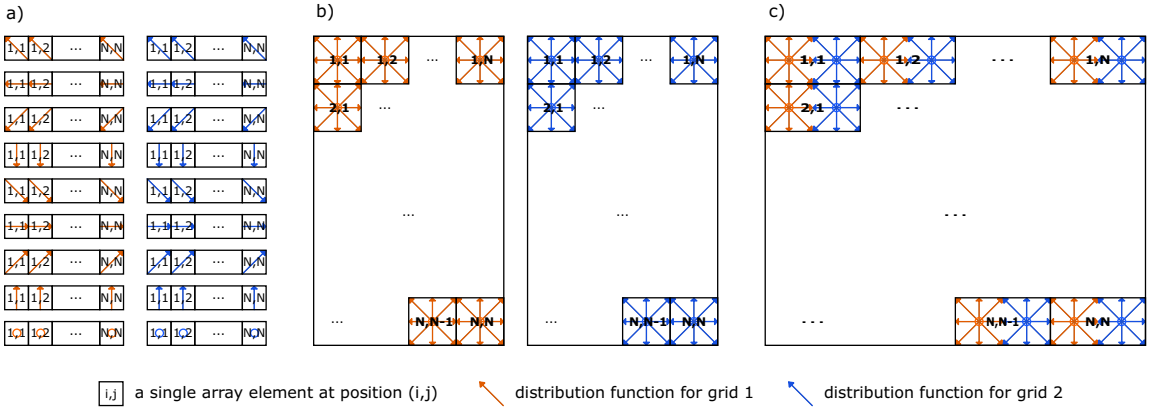


Figure 5.1: The datalayout for an LBM implementation can consist of either: distinct arrays for each distribution function type, two arrays containing all necessary distribution functions for each cell in a struct or a single array with both sets of distribution functions.

all 19 functions. Upon access to one distribution function, the next ones will be loaded into cache with the same cache line, speeding up accesses to these distribution functions.

This locality can be further increased by merging the two arrays, as shown in Figure 5.1. Only a single array is stored, that contains two sets of distribution functions for each cell. However, this locality is not always desired. During the stream step, both arrays are accessed – one for reading, the other one for writing. In all other calculations only one array is needed, so the interleaved second array may even slow down the memory accesses, as data is loaded into the cache, that is not needed in the program.

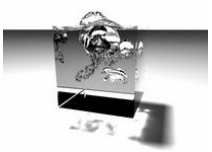
More performance can be gained by a method that is called *compressed grid* in [Wilke, 2002]. In this case, the memory required is not given by $2 \cdot N^3$ but by $(N+1)^3$, for a cubic problem with N cells in each direction. This is achieved by storing data from two different time steps in the same grid, overwriting the cells that are not needed for further calculations. For the lattice Boltzmann method, the only moment that requires both grids is the stream step. Even when surface tension and bubbles are included, the modifications can be handled during streaming. So after a single cell has streamed all its distribution functions to the adjacent cells, these old values are never accessed again. In other words, assuming an implementation with three loops i, j, k each running from 1 to N, after cell $(i+1, j+1, k+1)$ has copied the distribution function from cell (i, j, k), the values in cell (i, j, k) are never accessed again. So the new distribution functions of cell $(i+1, j+1, k+1)$ could be written to cell (i, j, k), and so on. Then the streamed cell $(i+2, j+1, k+1)$ can be written to cell $(i+1, j, k)$. This requires an extra layer of cells, otherwise the cell $(0, 0, 0)$ would access cell $(-1, -1, -1)$ that does not exist. So in the beginning the grid is stored in cells from $(1, 1, 1)$ to $(N+1, N+1, N+1)$, after the streaming the updated values can be found in the cells from $(0, 0, 0)$ to (N, N, N). The next streaming has to copy the values back into the other direction.

A LBM implementation usually has many loops running over all distribution functions for each cell. The easiest way to implement this, as shown in Chapter 2, is to use loops and arrays with the constants needed for the computations, e.g. the lattice vectors or equilibrium function weights. This implementation is close to the mathematical description, and as such easy to handle when testing variations. However, all calculations with the lattice vectors contain many multiplications with zero. Removing these terms can save many floating point operations. Loops that include multiplications with the lattice vectors usually can skip the calculations with lattice vector 0. It is $(0, 0, 0)$ and as such does not contribute to the momentum density, among others. When calculating the velocity in x-direction, only terms multiplied by lattice vectors with a non zero x-component are necessary. In this example only ten floating point additions need to be done, instead of the eighteen performed with a standard loop. On the other hand, this makes debugging and testing much harder.

To have both the performance of unrolled loops and the good handling of fewer statements, for the implementation of this thesis a small Perl program was written, that unrolls loops running over the distribution functions on demand. This can significantly increase performance for simple LBM programs. For this implementation, as will be shown later on (see Chapter 6), it is not that important, as the free-surface calculations dominate the overall performance.

5.2 Multiphase implementation

The different phases that were explained in Chapter 4 are identified by an additional integer containing the flags to identify the type of each cell. There are flags for fluid, interface and empty cells. Furthermore boundary cells with slip or no-slip boundary conditions are distinguished. During the initialization of the program, it is made sure that the layer of interface cells is not containing holes, or has unnecessary interface cells. In each time step, during the streaming process, the cell masses are updated for each interface cell. Once it is recognized that a cell is completely empty or filled with fluid (when the mass equals the density) it is written into a list for later handling. The cell flag can not be immediately updated, as the mass exchange process relies heavily on symmetry. The mass that is added to a cell has to be subtracted from another one. If the cell flags from the stream step and the mass update process differ, mass will be lost or generated. If this occurs, distribution functions will be streamed into empty cells, or streamed from newly initialized cells that were not taken into account during mass exchange calculation into existing cells. Furthermore, the cell masses usually don't exactly reach zero or the density of the cell during an update. Empty



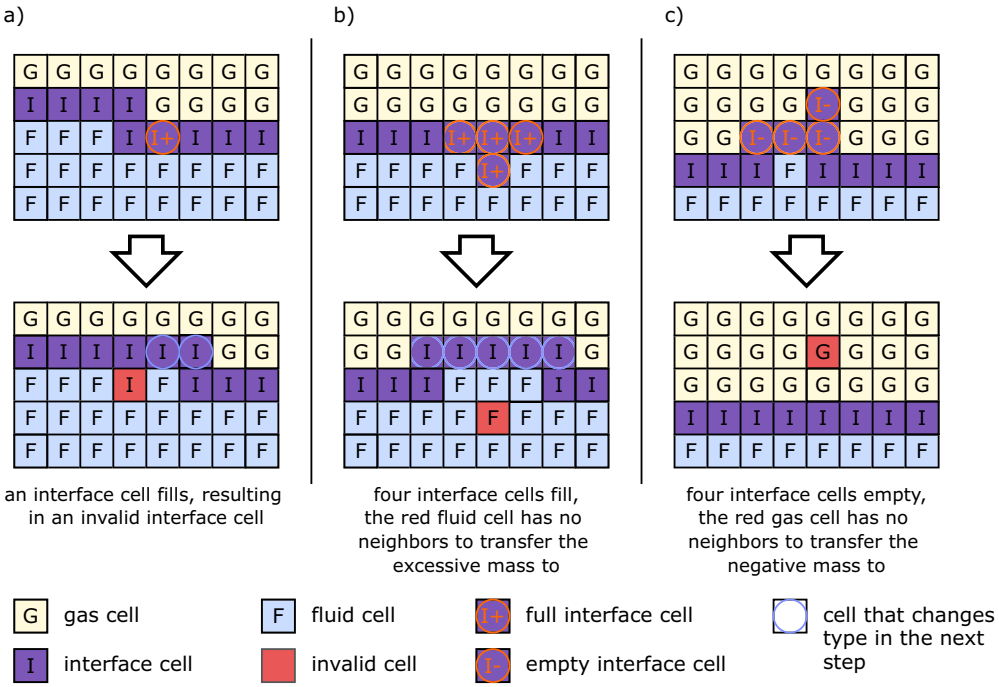
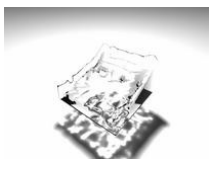


Figure 5.2: Figure a) shows how invalid interface cells can result from standard cell type changes. b) and c) illustrate cases where the mass from an interface cell cannot be correctly distributed to neighboring interface cells.

cells will usually have masses below zero, and filled cells more than the density of this cell would allow. This mass difference, be it positive or negative, has to be distributed among the surrounding cells during reinitialization.

Thus, during the stream step, the new mass values for each cell are calculated. Filled and emptied cells are stored in a temporary list without changing any cell flags. Then the streaming and distribution function reconstruction, as well as the collision are performed. When this is finished, the cell lists are processed. First the surrounding interface cell layer is constructed, which means that for empty cells the adjacent fluid cells are changed to interface cells, as are all empty cells next to filled cells. This is done before the cells in the lists are changed or the mass is distributed, to keep the symmetry of the method. Otherwise, for a row of neighboring interface cells are full or empty, each cell would have a different surrounding when some of it's neighbors are already updated. Care has to be taken when interface cells, that are needed as surrounding layer for filled interface cells, should be deleted. This would result in a hole in the interface layer – thus, the cells around new fluid cells are marked with an extra flag to prevent deletion.

When all cell flags are correctly initialized, the mass difference of the cells to be changed is calculated and distributed to the surrounding interface cells. For emptied cells the negative mass is added to the neighbors, while for filled cells, the difference between the density calculated from the distribution functions and the mass of the cell is considered. Still it can occur that empty interface cells should be deleted, but the surrounding cells empty as well, so that some cells do not have any neighbor to distribute the negative mass to (see Figure 5.2). The same can happen when a region of interface cells all fill at once. Some cells may not have any interface neighbors, as all surrounding cells are also changed to fluid cells in the same step. These cases rarely occur, but need to be recognized, although the mass difference cannot be easily handled. In the current implementation this mass is stored and distributed to all interface cells in the domain in the next step. This is computationally fast and does not destroy any symmetry. Furthermore the mass that is added to the cells this way is usually too small to immediately affect the simulation.



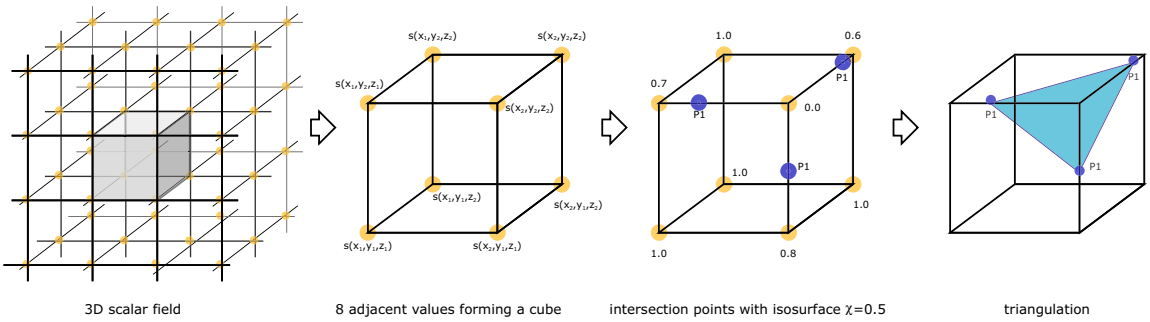


Figure 5.3: The marching cubes algorithm proceeds by choosing eight adjacent values from a given three-dimensional scalar field, triangulating the points of intersection with the isosurface.

5.3 Surface reconstruction

To apply Equation 4.3 during the stream step, the curvature for each cell has to be calculated. This means, that the curvature along two normal sections is needed, which requires at least three points for each section. An easy way to reconstruct a surface is the *marching cubes* algorithm that is known from volume visualization (see [Watt and Watt, 1992] and [Lorensen and Cline, 1987]). It can be used to triangulate the isosurface given by a regular three dimensional scalar field. The algorithm proceeds by considering eight scalar values forming a cube (see Figure 5.3). It is assumed that the scalar values change linearly along each edge. A point along an edge on the isosurface for a given value χ can then be easily calculated. For edges from $\vec{p}_1$ to $\vec{p}_2$ that have scalar values larger, respectively smaller, than χ (s_1 and s_2), the intersection $\vec{x}$ is given by:

$$\vec{x} = \vec{p}_1 + \frac{\chi - s_1}{s_2 - s_1}(\vec{p}_2 - \vec{p}_1) \tag{5.1}$$

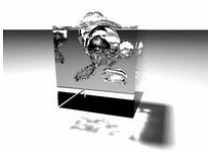
These points can be calculated for all edges intersecting the isosurface. A standard marching cubes implementation would then proceed by building the triangles from these points. In total $2^8 = 256$ different possibilities for edge intersections of a cube can be identified (each of the eight corners is either inside or outside the isosurface). All 256 cases of triangulations can be precalculated, and together with the exact intersection points be used to build a isosurface triangulation without holes.

For this LBM implementation, the fill grade of each cell (mass divided by the density) was taken as a scalar field with an isosurface at $\chi = 0.5$. The assumption of linear interpolation between the neighboring fill grades is not correct, still the points on the isosurface successfully approximate the contour of the fluid. And as the marching cubes algorithm can be implemented with good performance, it was chosen as surface reconstruction algorithm in this case. Other algorithms compute higher order surfaces through the scalar values and could be used to calculate the surface intersections with higher precision, but these algorithms require much more operations and are significantly slower.

Although it might be nice to have a triangulation for visualization of the fluid, it is not needed for calculating the surface tension. As the edges on which isosurface intersections are calculated are aligned with the coordinate system axes, the easiest way to find points on plane for the surface is to use axis aligned planes. Depending on the direction of the normal vector of the surface, which can be approximated by computing the gradient of the scalar field, either the xy-, the xz- or yz-plane are searched for points in the neighborhood of the current cell.

The edges shown in Figure 5.4 are searched for points. Since often more than three points are found, some of them have to be discarded. In the current implementation the two points that are closest to each other are identified, and the point with the smaller distance to one of the remaining points is discarded. This ensures that the three points taken for curvature calculation are spread as far as possible, as numerical errors increase when the points are close together.

The other special case that can occur is, when there are too few points to calculate the curvature. This happens when the surface passes through the far corner of a region searched for points, as shown in Figure 5.5. In this case, the direction and amount of curvature cannot be correctly



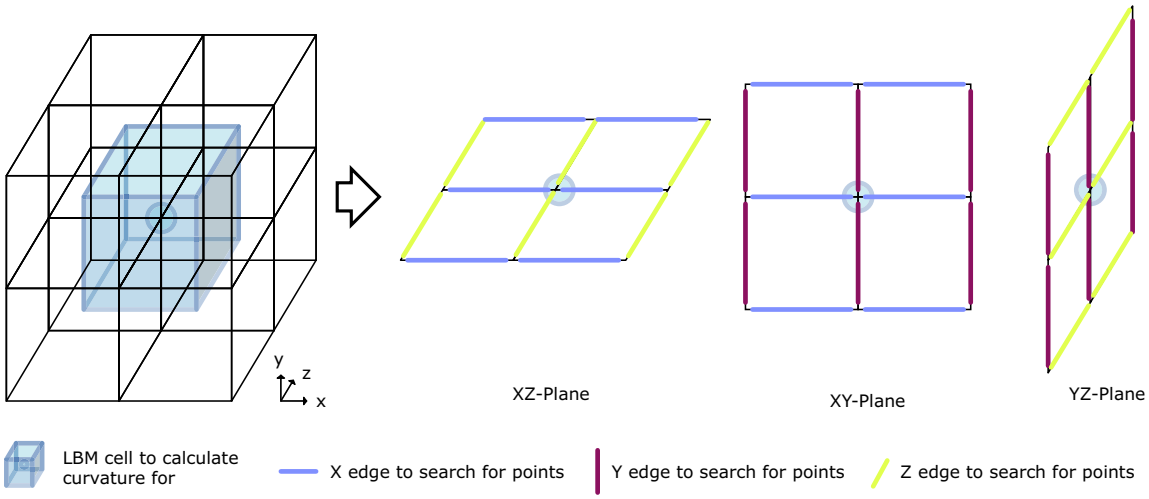


Figure 5.4: Depending on the normal direction at the cell, different planes are searched for points to calculate the curvature with.

estimated from the information in the shown region. The only reliable way to handle this, is to take the curvature calculated for the neighboring cell that contains surface intersections.

5.4 Problems

The implementation of this algorithm is required to deal with several difficulties. One important aspect is that fluid in interface cells that do not fill, is not able to move through the lattice. As the creation of new interface cells only occurs when cells completely fill or empty, an amount of fluid that is not able to fill a cell will not trigger the initialization of new interface cells. This effect is often caused by high lattice velocities, hence it can be reduced by choosing a parametrization that has sufficiently small time steps.

Another problem can be caused by regions of interface cells in the fluid. As the distribution function reconstruction is only done in the direction of gas cells, for adjacent interface cell regions no distribution functions are reconstructed. This means that no bubble pressure or surface tension can be applied to these cells, preventing them from quickly being filled. This effect could be mitigated by a distribution function reconstruction that also takes the normal of the surface into account.

Generally, care has to be taken with cell type changes. Unnecessary cell initializations can cause uncontinuous changes of the velocities, which results in disturbances of the fluid, especially for relaxation times close to two. In the worst case these changes trigger new cell type changes,

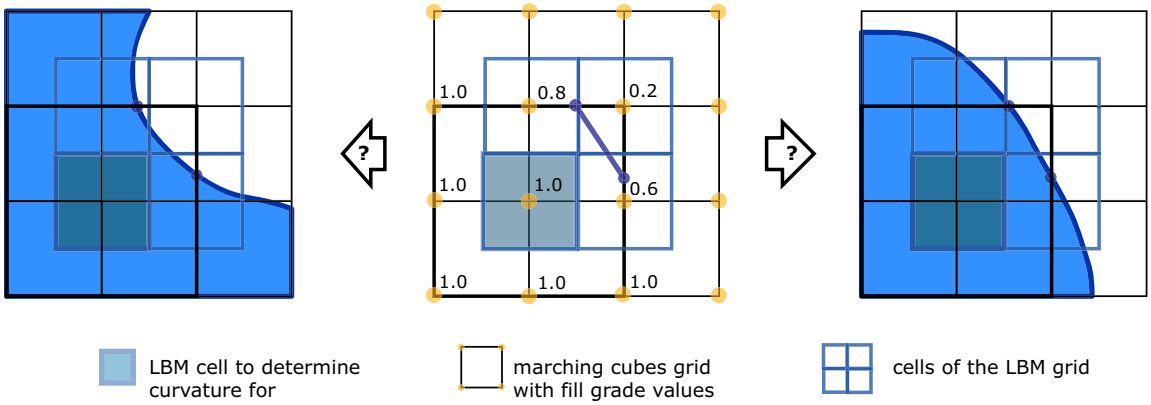


Figure 5.5: In the middle an completely undecidable configuration for calculating the curvature of a cell is shown. Depending on the surrounding fluid, the curvature can be either positive or negative.

leading to a "flickering" of the cell types. This could completely dominate the behaviour of the fluid, causing ghost velocities or deceleration of cell velocities. With the implementation described in this chapter these effects were minimized, but can still occur for high ω values.

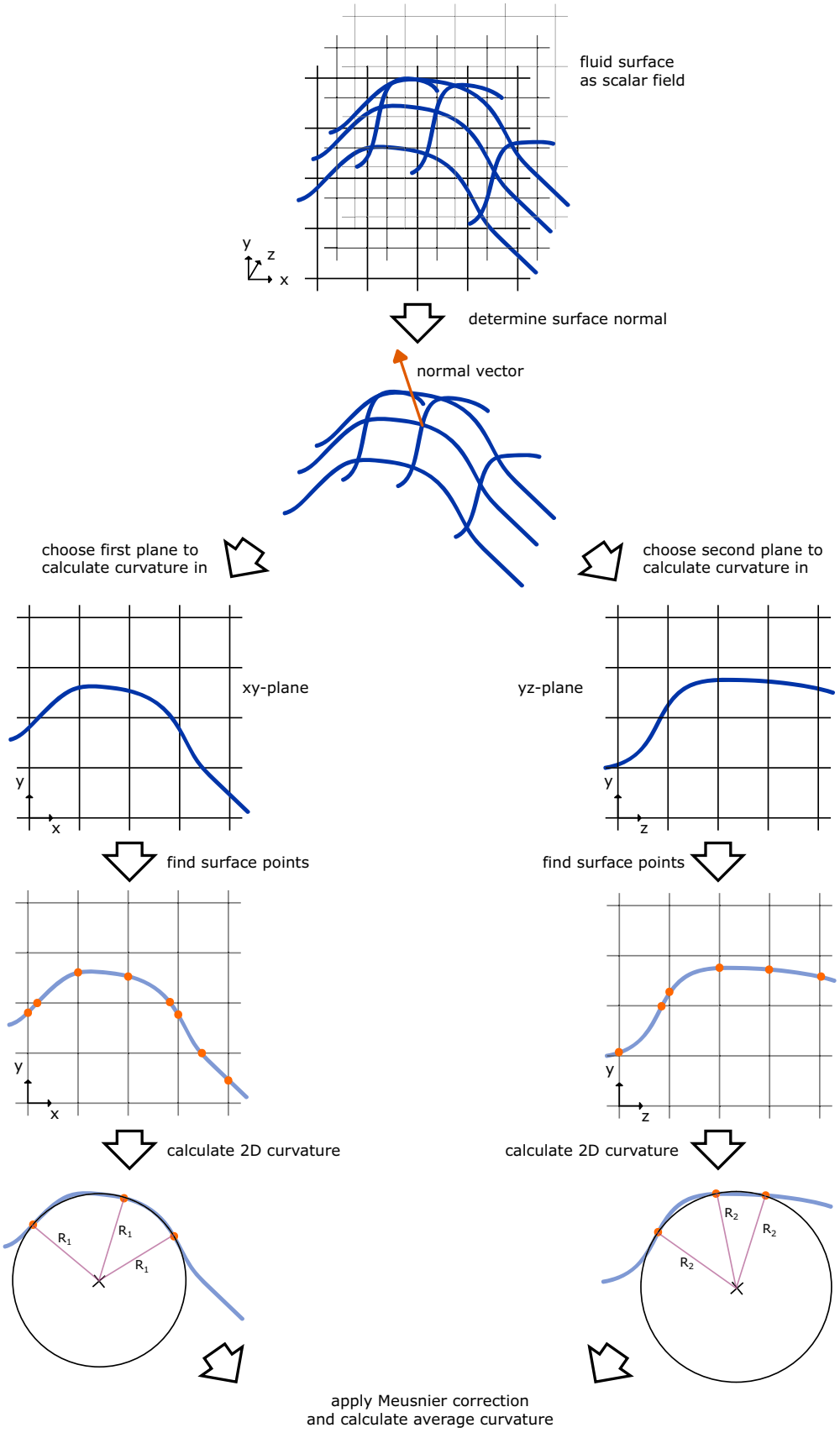
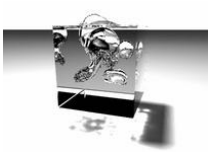


Figure 5.6: Here the whole process of curvature calculation that is necessary for each LBM cell is illustrated.



Chapter 6

Results

This chapter presents the results of simulations with the method described in the previous chapters for various problems. The images from this chapter were created with the raytracer that is described in more detail in Chapter 7.

Animated sequences for the following images can be found on the enclosed CD or at the websites [Thürey, 2003; Pohl, 2003].

6.1 Free surface

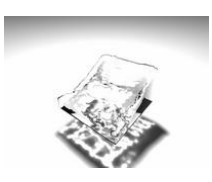
The standard test problem for free surfaces is the *breaking dam problem*. It can be imagined as a container where a section of the domain is filled with fluid and seperated from the rest by an obstacle. The obstacle is instantaneously removed, and the fluid, previously at rest, will start to swash towards the other wall due to the gravitational force. Here a setup is used, where $\frac{1}{3}$ of the width of the domain is filled up to a height of $\frac{3}{4}$ with fluid. A relatively small domain of 50^3 cells is used.

Several time steps of three animations can be seen in Figure 6.2. The difference between the three columns are the boundary conditions. For the first column, free-slip boundary conditions were used, while no-slip boundary conditions were set for the second one. The third column again shows a simulation with free-slip boundary conditions. In contrast to the first, however, it also includes surface tension.

Especially for a small grid like the one used in these simulations, the effect of the boundary conditions is strong. The results show the behaviour expected for the described setups. The free-slip boundary conditions allow a fast movement along the floor, and the fluid stays in motion for a long time. The no-slip boundary conditions, on the other hand, slow down the motion – there is almost no swashing, and the fluid quickly comes to rest. It can also be seen that parts of the fluid remain "sticking" to the wall, flowing downwards much slower than the fluid further away from the wall. While the simulation with surface tension shows the same overall motion as the first simulation, the strong surface tension removes all sharper edges of the fluid surface. The waves have rounder corners than in the first case, and when the fluid comes to rest, the surface has a visible bulge in contrast to the flat fluid surface of the simulation without surface tension.

6.2 Falling drops

In Figure 6.3 two simulations of a drop falling to the ground are shown. The images to the left are from a simulation with no surface tension, while the two columns to the right show pictures of the same drop of fluid with surface tension. The drop without surface tension slowly spreads over the floor, which is no-slip for the two simulations. The drop with fluid tension, however, is pulled together again as the surface tension applies an especially strong force at the flat edge of the drop towards the middle. The fluid reaches a stable form when the gravitational force and the surface tension are balanced. It can be seen that the surface tension is not completely symmetric, as the form of the drop, when it is pushed towards the middle, is not perfectly round. This effect is mainly



caused by a restriction of the maximal force applied due to the surface tension for stability of the LBM, and might be reduced by choosing a parametrization with smaller time steps.

6.3 Rising bubbles

The problem setup of Figure 6.4 is interesting for future foaming simulations. Each of the three columns show a rising bubble in a container. Although the pictures show equally large bubbles, the LBM resolution is different each time, resulting in differently sized bubbles. The diameter of the bubble to the left is 18 LBM cells, for the bubble in the middle 24 cells and the bubble to the right has a diameter of 36 cells. An increased bubble size results in an increased ascendency – the pictures at the bottom show that the largest bubble already breaks through the surface, while the two smaller bubbles are still several cells below it. The larger bubbles also show a stronger inwards bulge at the lower end, that is caused by the increased downward flow speed around the bubble.

A problem with the reinitialization of the cell types can be seen at the lower edges of the bubbles, especially for the two larger bubbles. When the speed of the rise is too large, the algorithm is not able to correctly empty the corresponding cells anymore, resulting in small regions of interface cells that are not connected to the main bubble. As the algorithm is currently not performing a segmentation of the domain to identify new bubbles, these smaller bubbles are treated as if they were still connected to the main bubble, and as such have the same pressure.

6.4 More drops and breaking dams

The images from Figure 6.5 and Figure 6.6 both show settings similar to those of Section 6.1 and Section 6.2. In the first case four drops of fluid with surface tension fall on a rectangular obstacle. Note that the four drops are falling from different heights, resulting in an irregular splash when they hit the obstacle.

The second breaking dam from Figure 6.6 has a much higher viscosity than those from Figure 6.2 and an increased gravity. This results in a more turbulent and asymmetric flow. Due to the high ω-parameter even small discontinuities from the cell type reinitializations are amplified. Due to the high strength of the external force, some distribution function even become smaller than zero and have to be reset to preserve stability. However, this effect is negligible for a simulation of metal foams, as the forces acting upon the fluid are much smaller in this case.

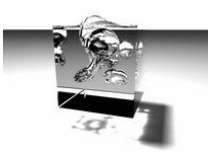
6.5 Six rising bubbles

A simulation of six bubbles is shown in Figure 6.7. The flow between the bubbles causes coalescence before they break through the surface. Here again the effect of large lattice velocities causing left behind interface cells can be seen. Due to the good resolution of the simulation ($90 \times 120 \times 90$ cells) the surface tension causes a realistic burst of the bubbles at the fluid surface. In the images from later time steps, the effect of a missing segmentation can be seen. The small bubbles left in the fluid are connected to the atmosphere since the main bubble they belonged to broke through the surface. Hence, once another bubble touches one of these small bubbles it is treated as if it is connected to the atmosphere and will not cause an increased pressure when it is compressed.

6.6 A validation experiment

To validate the results of this thesis, a small breaking dam was built. It has the a length of 30cm and depth and height of 25cm. From Section 4.5.1 it is known that a realistic representation of a domain that big would require too much memory for any current workstation. Hence, a resolution of $100 \times 83 \times 83$ cells was chosen that only requires 300 MB of memory. To retain the stability – a parametrization with such a coarse resolution would require an ω of almost 2 – a higher visocisity than water was used.

Figure 4.5.1 shows several time steps of the simulation (to the left) and photos of the real breaking dam (to the right). While the overall shape of the wave is comparable, the coarse resolution of the simulation lacks many turbulent details that are visible in the photos.



6.7 Performance measurements

Although the algorithm has not been implemented with a focus on high performance several measurements were performed to test various optimizations. The work distribution for a standard breaking dam simulation run is shown in Table 6.1. It was measured using a Pentium 4 with 2.4GHz and 1GB of RAM. The program used to measure these numbers performs a seperate sweep through the lattice for each of the listed steps. The streaming step takes up almost 40% of the running time. This is caused by the memory accesses to the neighboring cells and the reconstruction of the free surface distribution functions, which is also performed during streaming. The collide step takes almost as much time due to the complicated calculations that are performed for the local equilibrium distribution functions. The calculation of the mass exchange and the calculation of the points on the isosuface both need about 10% of the total time.

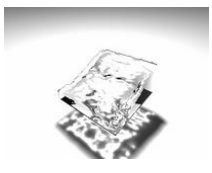
In Figure 6.1 several graph of performance measurements can be seen. The left column shows the results for the standard version that was also used to create Table 6.1. The numbers were measured with PAPI [Mucci et al., 2003] and show the millions of floating point operations per second (MFLOPS) and the level 1 and level 2 data cache misses per LBM cell update. The three lines show different problems that were used. The first setup has a domain completely filled with fluid, so no additional work has to be done for the handling of interface cells, except for checks of the cell type which were not removed from the code. The second version is a breaking dam problem with a free surface, the third one also applies surface tension. From the level 2 data cache misses it can be seen that the problems quickly become too large to fit into the cache, leading to increased numbers of cache misses. These misses do not increase further once the lattice is too large to allow the reuse of any data in the cache (a grid size of around 48^3 in this case). It can be seen that the overall performance, with an average of 90 MFLOPS for the version with surface tension, is very low compared to the 4800 MFLOPS the CPU would theoretically be able to perform.

As the implementation used for these experiments was written to allow the testing of different algorithms, it performs each step of Table 6.1 in a seperate sweep over the grid. In total four sweeps are necessary: for streaming, collision, mass exchange and bubble volume calculation. These four steps could be combined, fusing the four different loop nests into a single loop over all cells, to hopefully increase the low performance. First the mass exchange is calculated, then streaming with distribution function reconstruction and bubble volume calculation can be performed. The streamed distribution function can then be collided and written to the other lattice. This optimized version was also measured with PAPI – the results can be seen in the right column of Figure 6.1. Although for larger grids the number of level 2 data cache misses pert cell are almost half the number of those for the standard version (35 instead of 60), the MFLOPS performance does not increase. On an average it slighty decreases for the optimized version. This can be explained by other effects that were not measured, but seem to dominate the overall performance. Especially the large number of conditional statements in the loop body may lead to many branch mispredictions and pipeline flushes. Furthermore the huge loop body (more than thousand lines of C++ source code in this case) may exceed the instruction cache, and lead to an increased number of cache misses there.

It can be concluded that an obvious optimization like the loop fusion performed here is not enough to really speed up a complex code like this. It will be a topic of further research to create an optimized implementation (possibly in FORTRAN) for the SR8000.

Workload percentage: %	Program module:
39.4 %	Streaming
34.0 %	Collision
9.8 %	Mass exchange
9.6 %	Isosurface reconstruction
7.2 %	Initialization and other functions

Table 6.1: Work distribution of the LBM implementation for a breaking dam simulation run.



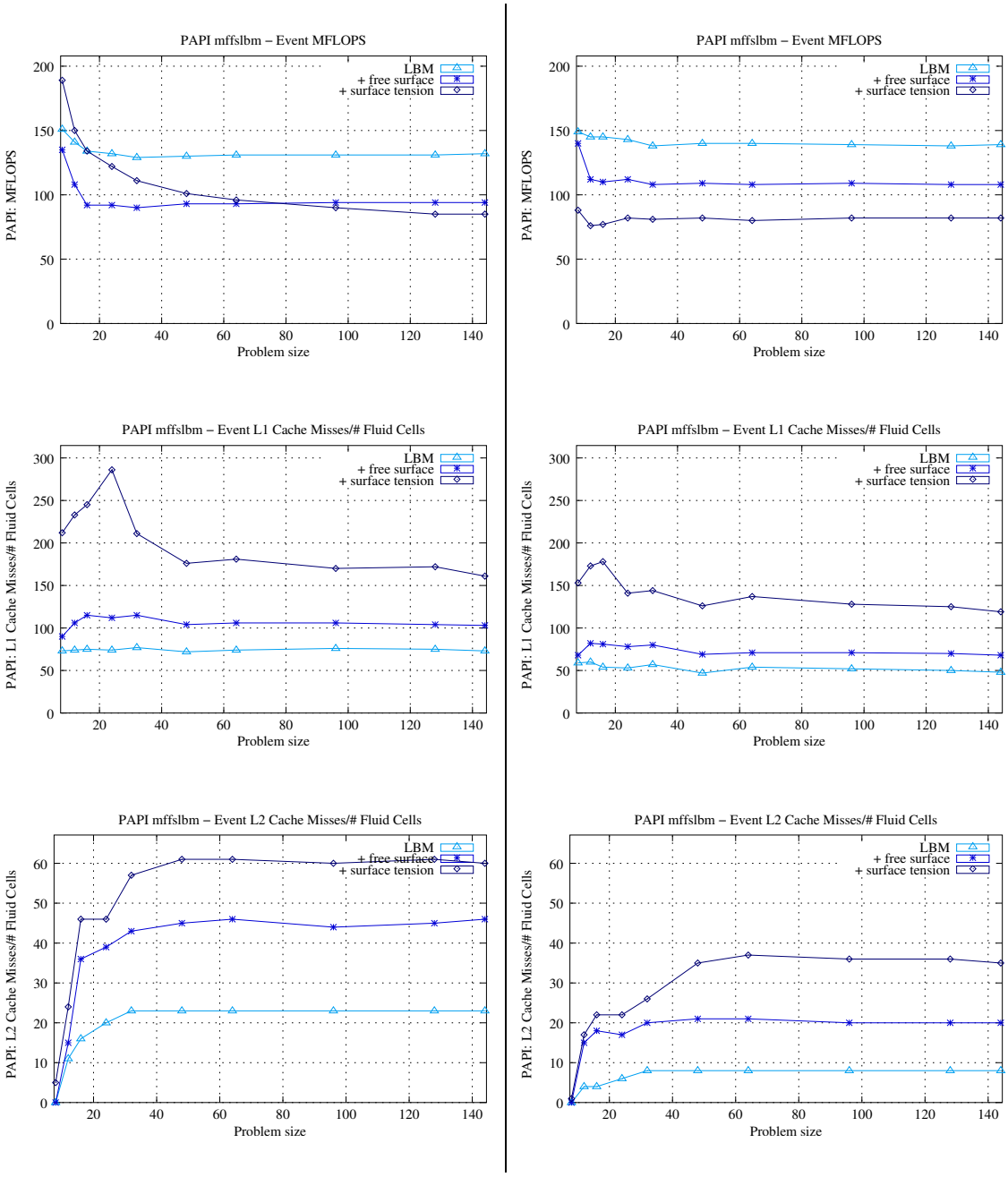
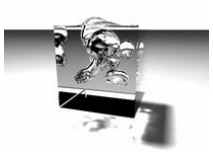


Figure 6.1: Measurements of the MFLOPS, level 1 and level 2 data cache misses per cell update for various problem sizes on a Pentium 4 with 2.4 GHz and 4GB of dual-channel DDR-SDRAM (using PAPI and GCC 3.2.2). The graphs to the left show the original unoptimized version, while the graphs on the right are for the program version with fused loops.



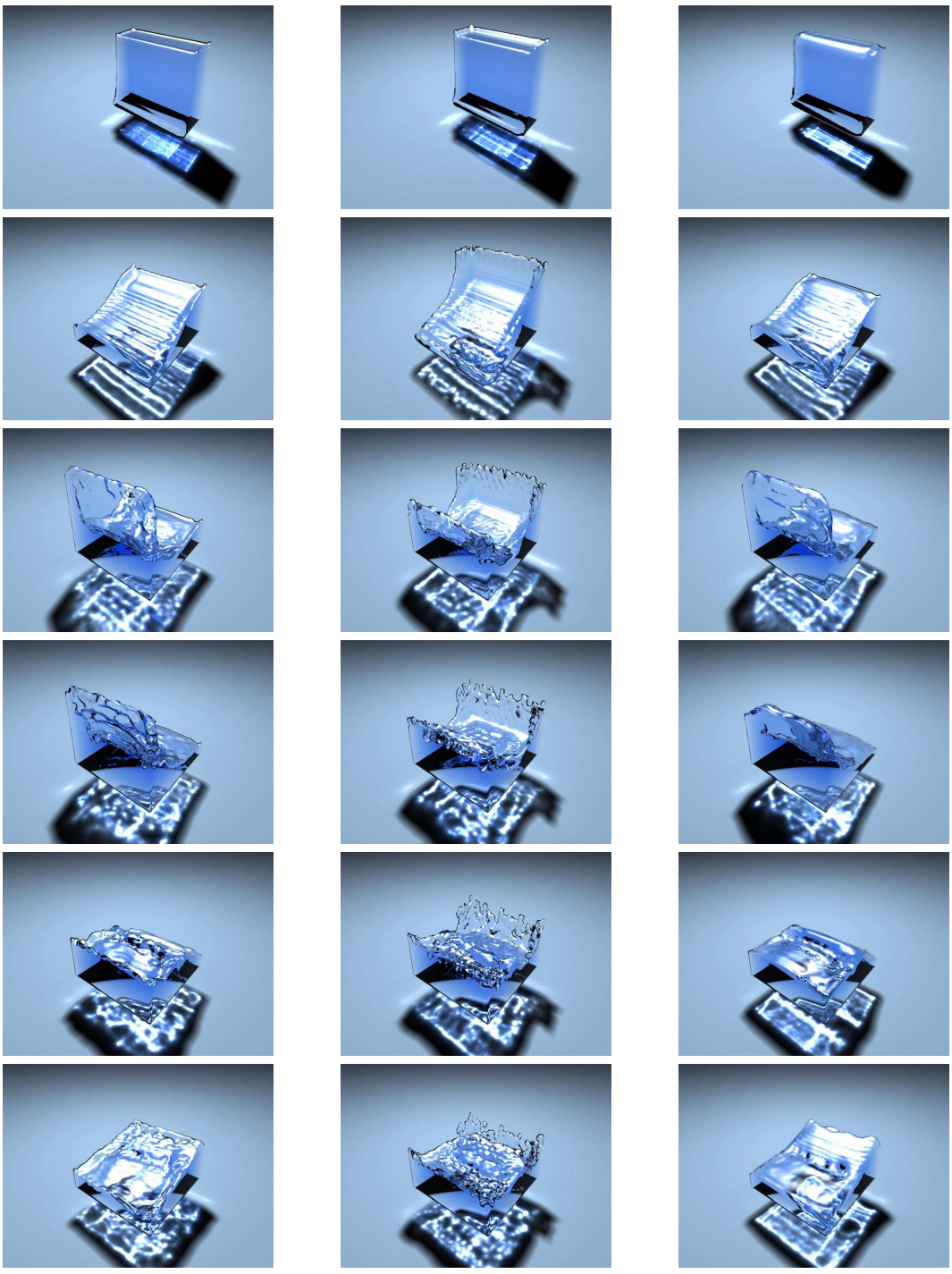
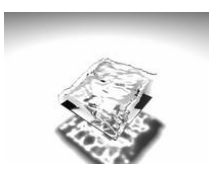


Figure 6.2: The standard breaking dam problem. To the left with free-slip boundary conditions, with no-slip walls in the middle and to the right with free-slip walls and surface tension.



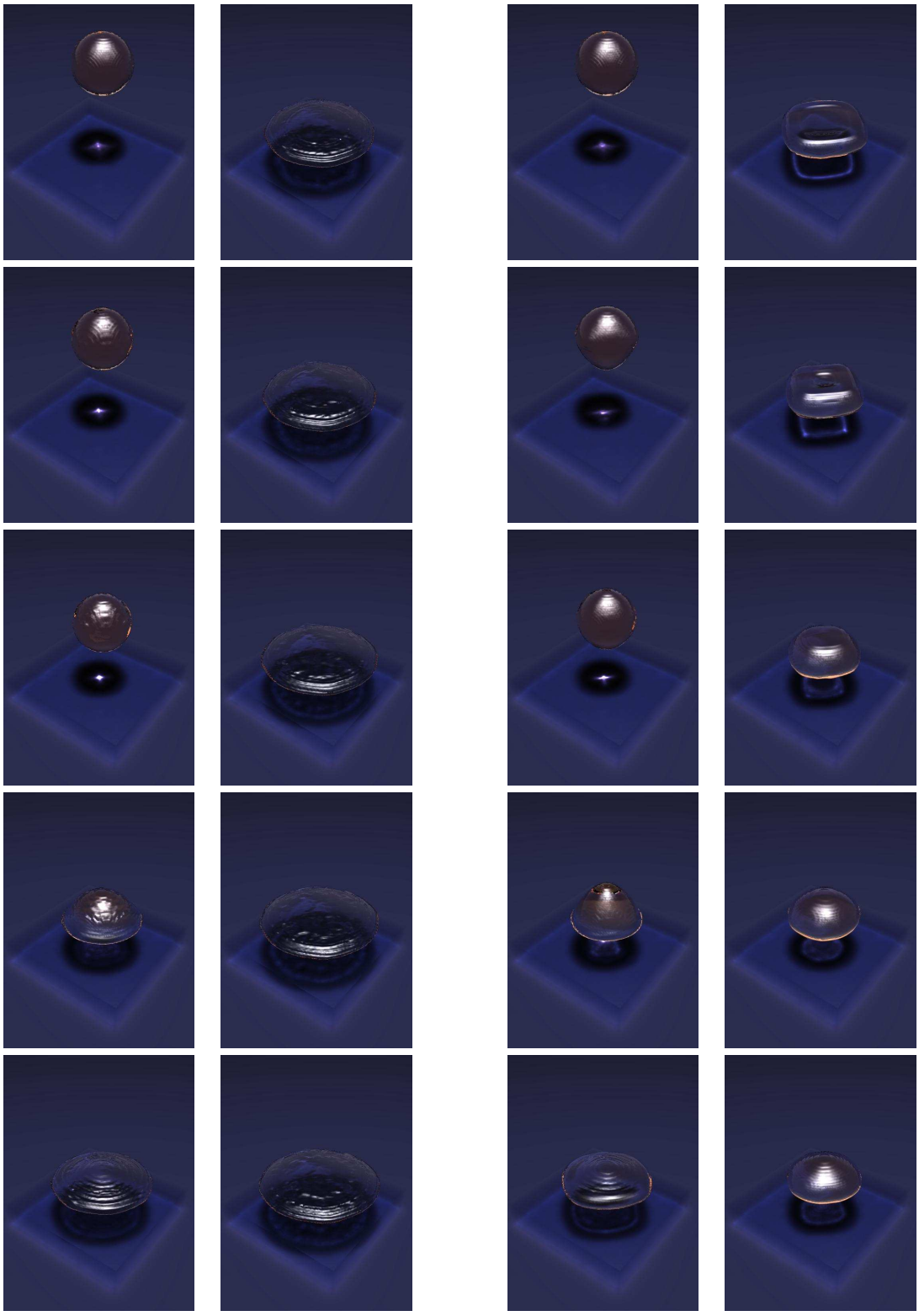
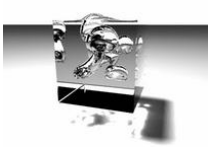


Figure 6.3: Two drops falling onto a no-slip floor. To the left without, and to the right with surface tension.



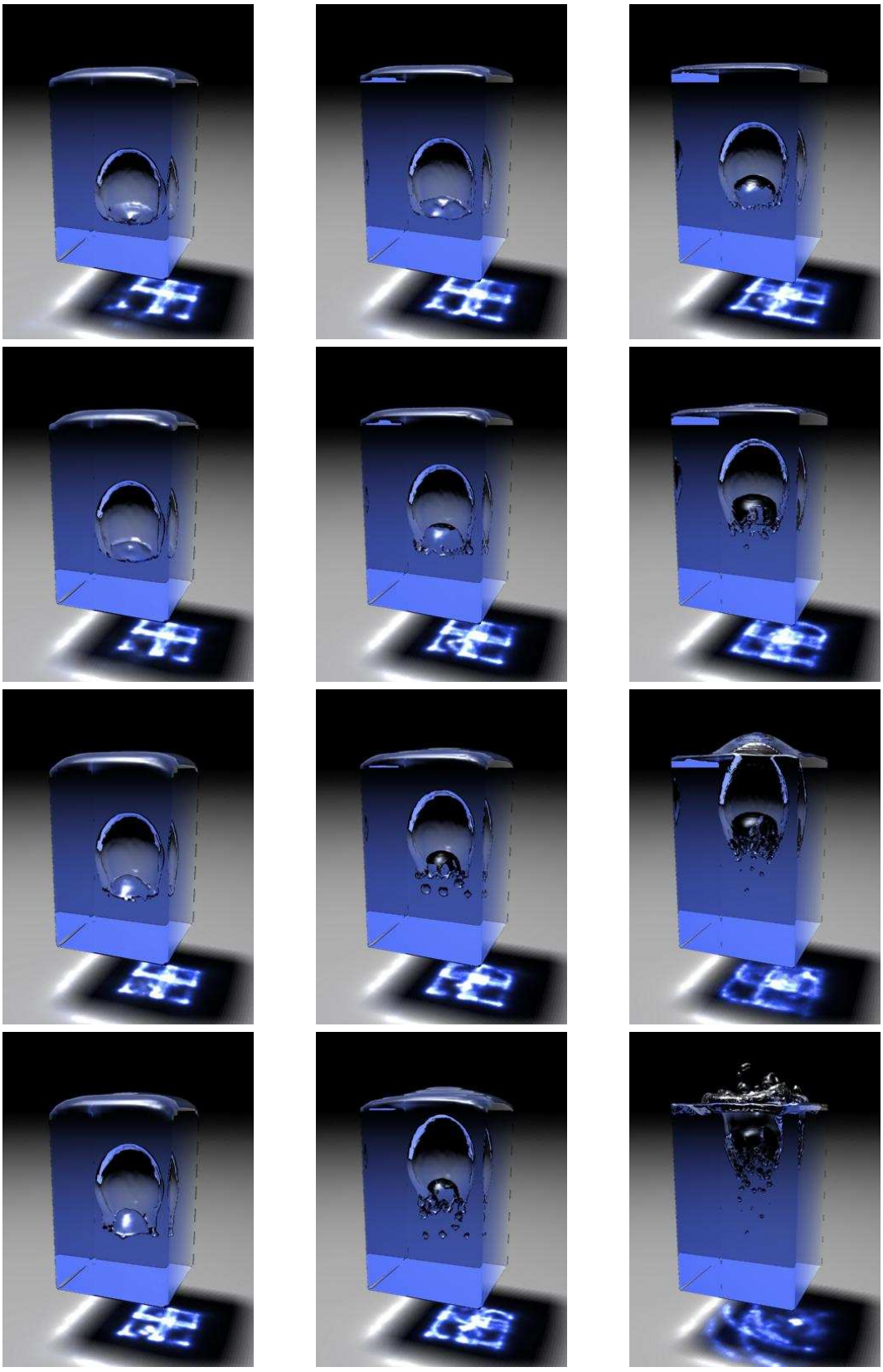


Figure 6.4: Three differently sized rising bubbles.

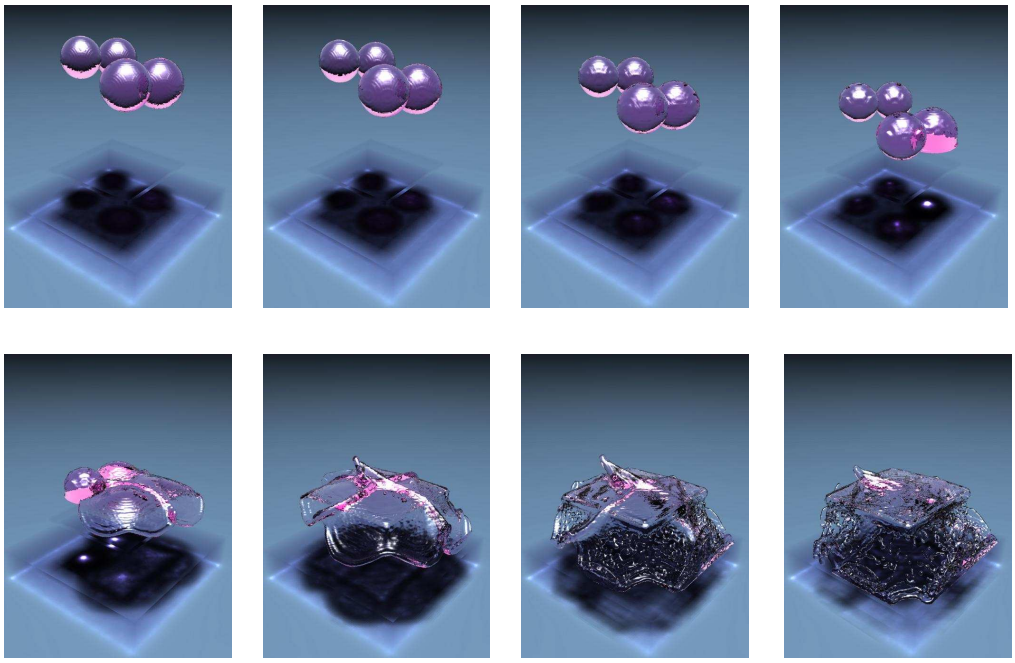


Figure 6.5: Four drops of fluid falling onto a square obstacle.

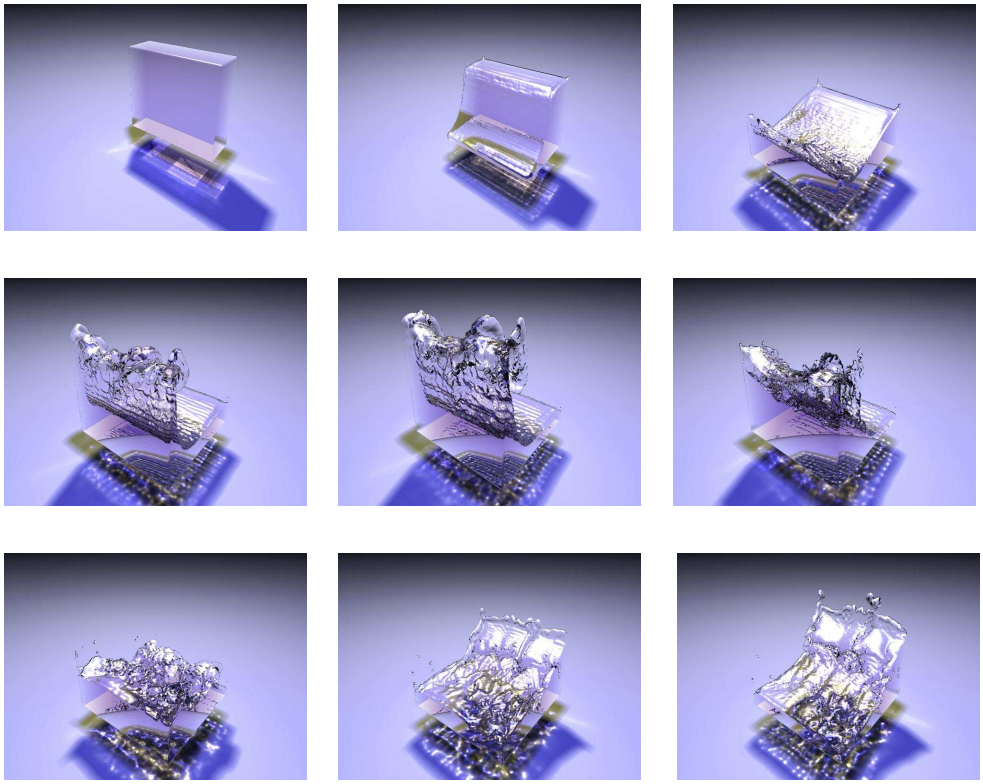


Figure 6.6: Another breaking dam with low viscosity and strong gravitational force.

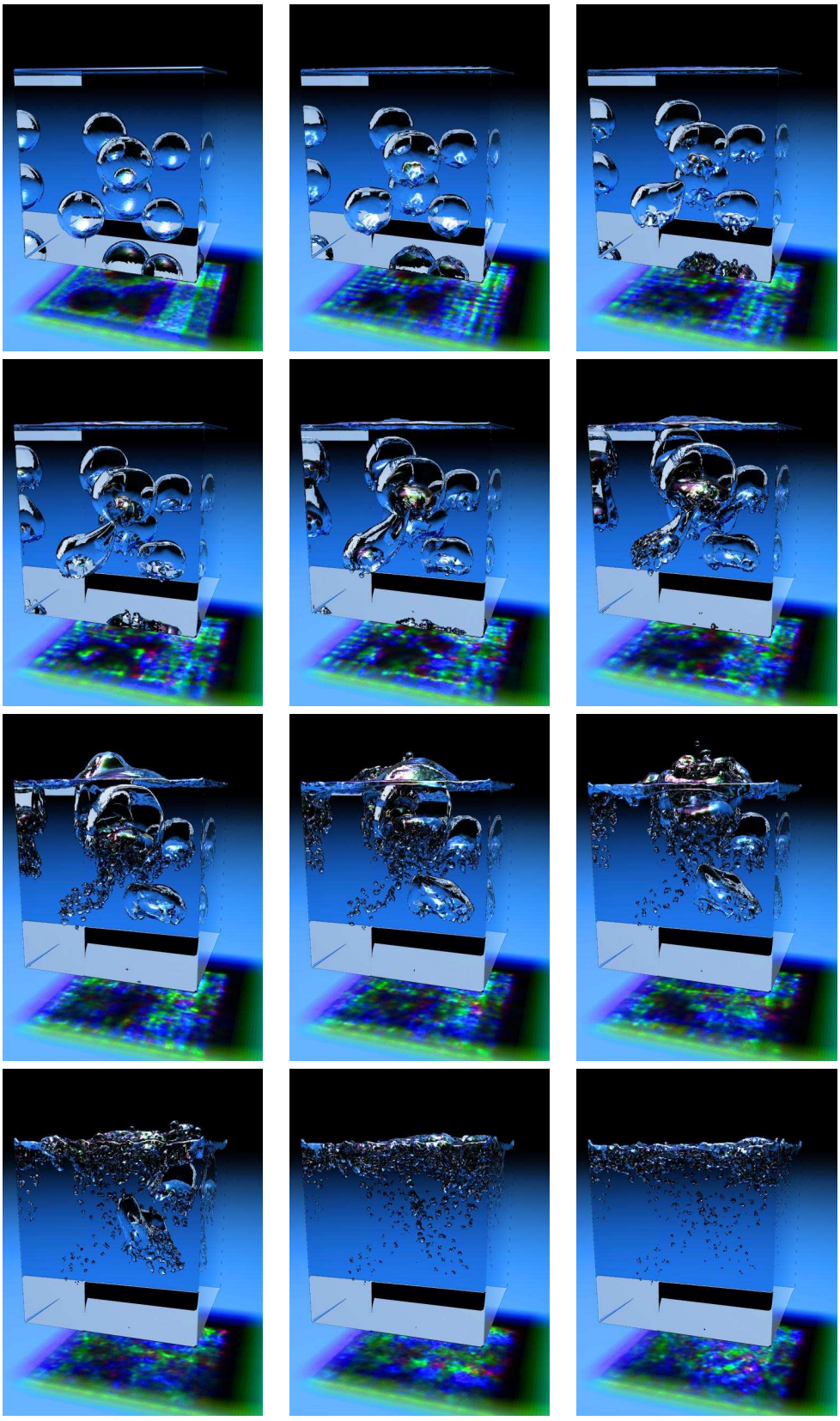


Figure 6.7: Six rising bubbles in a container with fluid.

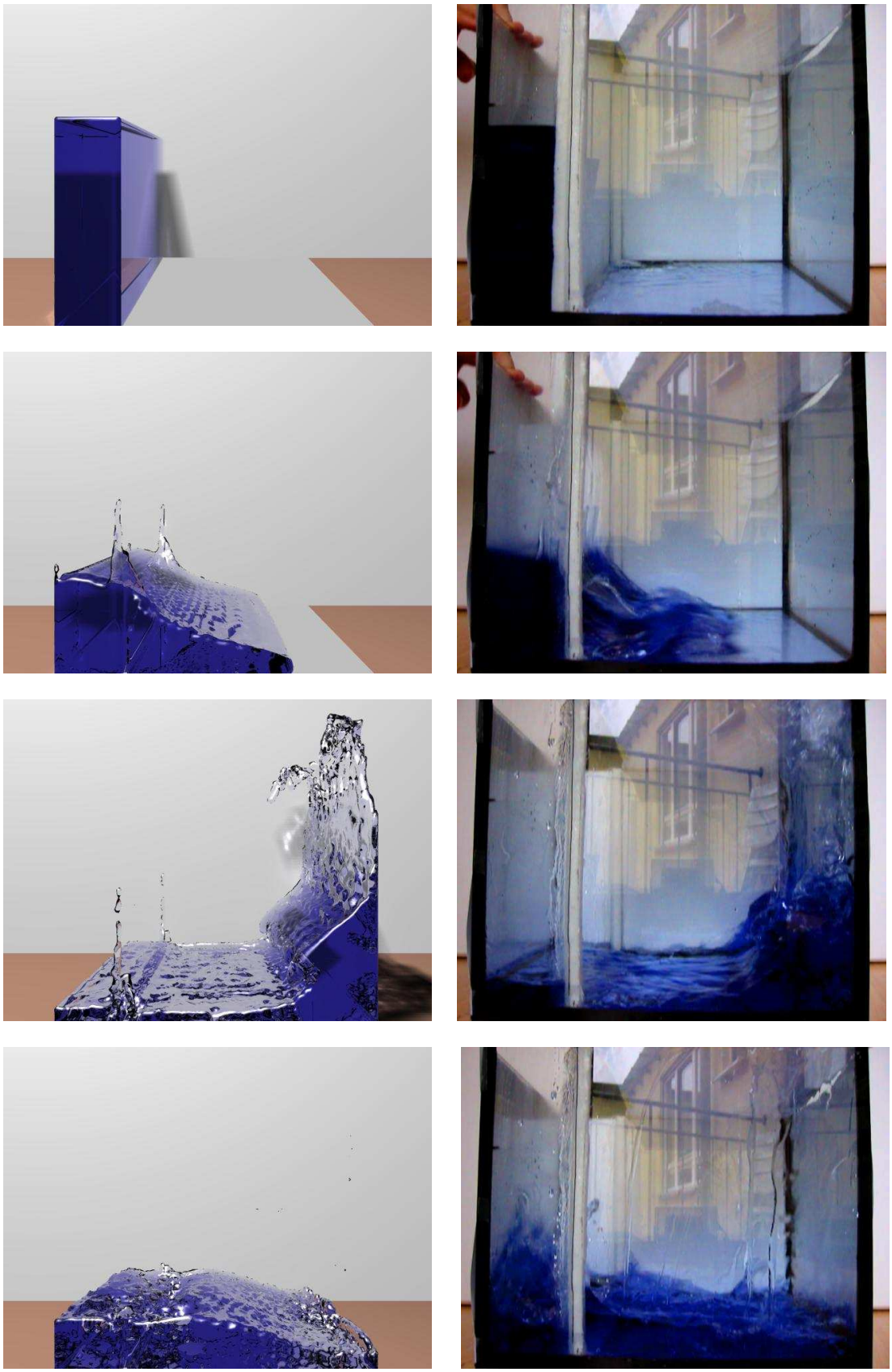


Figure 6.8: Comparison between a simulation (to the left) and a real breaking dam (to the right).

Chapter 7

Visualization

This chapter will explain the two different types of visualization that were used during the LBM implementation. First the simpler and quicker visualization using OpenGL will be described. This is useful for interactive evaluation of the simulation and fast debugging of the program. After that the difficulties of creating realistic images from simulation data will be explained in more detail.

7.1 Real-time visualization with OpenGL

The OpenGL library is a standard tool for the creation of fast and portable three-dimensional visualizations. Although several libraries focusing of scientific visualization exist (like OpenDX and Tecplot), the visualization of this implementation is handled by a module in the program due to the simplicity of OpenGL and flexibility gained by a self-written solution. Several screenshots of the OpenGL module can be seen in Figure 7.2. The small red cube visible in most pictures is the cell selector, that can be moved within the grid to select specific cells, and display more information for these.

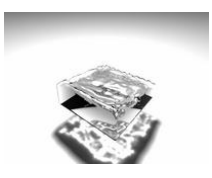
For quick evaluation of the free surface simulation, the marching cubes implementation from the surface tension can be used. As the points are generated for the curvature calculation, these can be used to create the triangles used for the OpenGL visualization without much overhead. This furthermore allows easy debugging of the surface tension calculations, as the points used for a circle reconstruction are the same that are used for the triangles. Additionally particles in the fluid can be easily displayed as dots or, for showing their previous positions, as lines.

The overall performance of the simulation is in most cases not affected by the OpenGL visualization. The fluid is only redrawn in certain intervals to display the progress of the simulation. This does not slow the simulation down. Other possible methods, like continuing the user interactions with separate processes for simulation and visualization on the other hand, would decrease the performance. To prevent this, the triangulations are stored, and can be played with a desired frame rate after the simulation is finished.

7.2 Realistic visualization with raytracing

The visual appearance of transparent fluids is mainly determined by the reflection and refraction of light at the fluid surface. Depending on the refraction index of the material, this effect is more or less strong. The background as seen through the material is distorted, as are photons that illuminate other surfaces around the fluid. Due to this light focussing brighter patterns are usually visible around fluids or other reflecting objects that are known as *caustics*. To capture these effects when visualizing the simulation of fluids, *raytracing* is used, as this technique is nowadays established for the creation of realistic images.

The algorithm works by tracing rays from an eye or camera into a virtual scene, calculating the illumination for each of the points found along the ray. This is the inverse of the usual process, where photons are emitted from light sources, illuminating certain objects scattering the photons until they hit a camera or eye – which is justifiable by the bidirectionality of the underlying equations. For details on raytracing refer to [Glassner, 1989] and [Foley et al., 1995]. Many professional (and



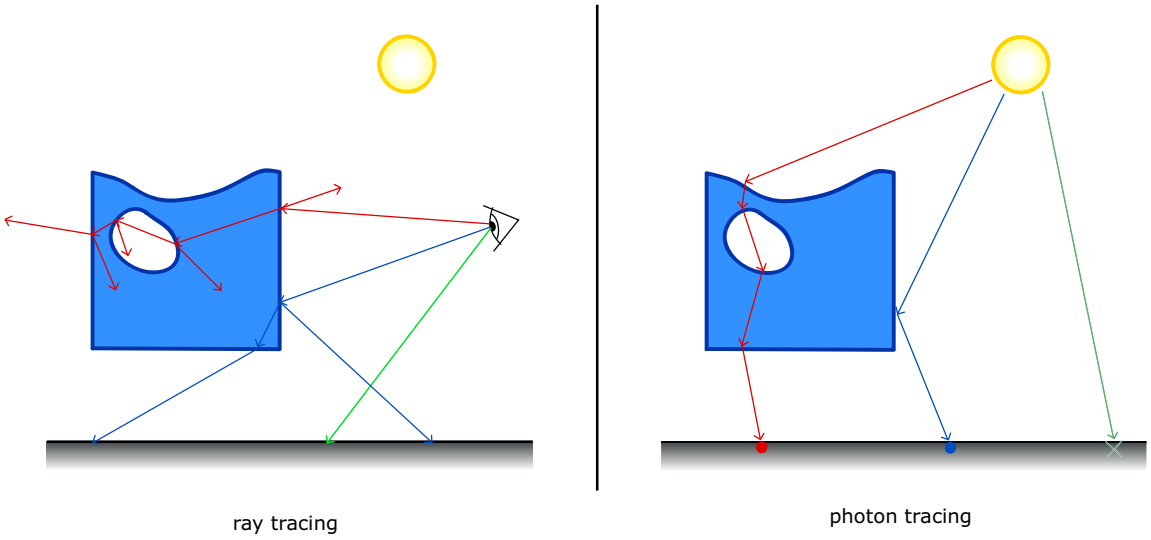


Figure 7.1: The left pictures shows three different rays traced from the eye into the scene are shown. The green ray directly hits a diffuse surface (the floor), the color can be calculated from material properties and light source position. The blue ray hits a fluid surface, a refracted and a reflected ray are recursively traced on. For the red ray, many new rays need to be spawned – note that reflections can also occur on the inside of an object. To the right photon mapping is illustrated (see Section 7.5).

often expensive) raytracers are currently available (e.g. mental ray from mental images, Houdini's Mantra or Pixar's Reyes renderer Renderman), as are open source raytracers like Persistance of Vision (POV), the Blue Moon Rendering Toolkit (BMRT) or Blender. As it turned out that all of these were lacking flexibility, a self-developed raytracer was implemented with efficient algorithms for the effects that are important when visualizing fluids.

Raytracing is a recursive process, as the rays sent into the scene can be traced on into a new direction upon intersection with an object. This can be used to easily implement effects like reflection and refraction. The color found for the new rays is simply weighted and used as color for the initial ray. This is also the main difference between using e.g. OpenGL or a raytracer. OpenGL directly draws all objects with a given color – for reflections or refractions, a preprocessing step has to be performed. By using raytracing these effects require only small changes. The penalty of the increased realism is the additional time required to create an image. While modern graphics hardware is able to display thousands or even millions of triangles many times a second, images created with raytracing can take hours to compute. Both techniques can be combined well, by using OpenGL to quickly display, evaluate and parametrize the simulation. Once the simulation proceeds as desired, an image sequence can be created with raytracing.

For the surface description of the objects in the scene, only triangles are used as basic primitives. These are easy to store and intersections can be computed efficiently. The intersection calculations of rays with the scene determine the performance of the image creation. As a scene can contain millions of triangles, not all can be intersected for each ray that is traced. *Binary space partitioning trees* are used to reduce the number of triangles that need to be intersected for each ray.

7.3 Reflection, Refraction and Fresnel

As the reflectivity and transparency of a fluid also depends on the angle between surface and ray, *Fresnel's law* can be used to calculate the necessary coefficients. A visually corresponding approximation is given by the following formula from Blinn (see also [Blinn, 1977]):

$$R = (1 - r_i)^4 + (\vec{d} \cdot \vec{n})^5 \tag{7.1}$$

The reflectivity coefficient at the surface for a ray leaving it in direction $\vec{d}$ is then R. Hence, the refractive coefficient is $(1 - R)$. In Equation 7.1 the surface normal is denoted with $\vec{n}$, and r_i is

the refraction index of the material. The difference between simple reflection, refraction and the more physically correct use of Equation 7.1 can be seen in Figure 7.3. The visual effect of a more realistic transparent material with Fresnel-coefficients is demonstrated in these four pictures. The sphere to the left is completely reflective, mirroring the floor and the white background usually not visible in the images. The image on its right side is a completely transmissive sphere, that is slightly colored. As the material has the same refraction index as the "air" around it, the rays pass straight through it. The next picture is that of a sphere where the reflective and refractive coefficients are calculated with Equation 7.1. Note how the reflectivity increases as the rays hit the sphere surface at a lower angle. The last picture to the right is similar to it's left neighbor, the only difference being the refraction index of 1.15 instead of 1.0. The light is focused on the floor, and the rays passing through the sphere are distorted.

While this calculation of the Fresnel-coefficients works fine for rays passing from a medium with low refraction index, like air which has approximately 1, into a medium with higher refraction index, problems can occur when this ray leaves the object again. In some cases, especially for lower angles, *total reflection* can occur, which means that Equation 7.1 is not applicable any more, and the ray is completely reflected on the inside of the object. More generally this is possible when the light passes into a medium with lower refraction index. For total reflection the reflection coefficient is simply set to one and no refraction is considered.

7.4 Soft shadows

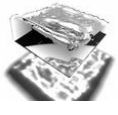
In nature a light source usually has a certain extent. For a raytracer, the easiest way to implement a light source is to use a infinitely small point, from which light is sent in all directions. This produces the typical shadows for raytracing with sharp and clearly visible edges. In nature this only occurs when a strong light source is far away from the shadow casting object. Usually shadows have a penumbra that result from regions that are only partly illuminated by the light source (see Figure 7.4). The left picture shows a simple sphere lying on a plane illuminated by a light source in the far left corner. Note how the soft edge of the shadow is smaller to the left side, where the sphere edge casting the shadow is closer to the plane. The two images to the right show the same scene – a rectangular box with a sphere hovering above it, the latter of which is not really visible on the pictures. For the picture in the middle the light source is close to the sphere resulting in a very soft shadow to the left. The light source was moved further away from the objects in the rightmost image, resulting in overall sharper shadow edges. The difference in the width of the partly illuminated areas is still similar to that of the middle picture.

These soft shadows can be computed by sampling a light source with an area a certain number of time. So called *shadow rays* are casted from a surface point in the direction of the light source to determine the visibility. For example rectangular light sources can be used, and shadow rays traced N times along each side, to determine how much of the light source is visible from a point on a surface. Hence, this requires N^2 shadow rays instead of only one for a point light source. This increases the time for raytracing by an order of magnitude, as usually relatively large numbers for N, e.g. $N = 8$, are necessary for shadow edges without aliasing.

A technique known from real-time visualization is *shadow mapping*. In a separate rendering pass, the distance from the light source into the scene is stored in a map. During raytracing, the distance to the light source and the corresponding value in the shadow map can be compared. If the shadow map value is smaller, the light source is not visible. Although care has to be taken when choosing the shadow map resolution or evaluating the depth comparisons, shadow maps can be computed quickly with raytracing. Moreover, shadow maps allow filtering, which produces soft shadows that, although not physically correct, are visually more appealing than the sharp edges of raytraced shadows – and are at times even quicker to compute.

7.5 Caustics with photon mapping

The limitations of the raytracing algorithm become apparent when more complex effects like caustics or color bleeding are desired. When a ray hits a surface that could contain a caustic, the correct way to compute these would be to integrate the light over the hemisphere at the intersection point from all reflecting or refracting objects, repeating this process for each intersection. This would



be necessary for all diffuse objects to correctly visualize color bleeding instead of caustics. Global illumination algorithms that can compute these effects are for example *global illumination algorithms* or *Monte-Carlo raytracing*. For this implementation *photon mapping* was used. It is significantly faster than the other methods in most cases, and can be easily integrated into an existing raytracer, as it requires similar calculations. The photon mapping technique is described in more detail in [Jensen, 2001].

It is basically a preprocessing step, that performs "physically correct" raytracing. So for all light sources a certain number of photons are traced into the scene, reflected or refracted at the intersection points, and finally stored when they hit a diffusely reflecting surface. These stored photons are organized as a *kd-tree*, that allows efficient retrieval of photons in a certain region. During raytracing, for each surface intersection a fixed number of photons, which are closest to the intersection point, are retrieved via the kd-tree, and averaged to compute the illumination that is not already computed from the raytracing algorithm.

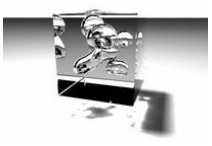
In the current implementation, a photon map is used only for visualizing caustics, as other global illumination effects are not as important for fluid visualization. To speed up the initial step of tracing photons into the scene, a technique similar to that described in Section 7.4 is used. To identify the regions that contain objects which could produce caustics, a map is precomputed to quickly select those photons which move into a reasonable direction. This is especially useful when the light source is far away from objects that produce caustics, while it will only slow the process down a bit when the light source is for example placed into a glass ball.

As the resolution of these *caustics maps* can be fairly low (it should just roughly capture the caustics producing geometry), and only a boolean value needs to be stored, they can be computed quickly with low memory requirements.

Pictures comparing raytraced images of caustics with different numbers of photons can be seen in Figure 7.5. The number of photons increases from left to right and from top to bottom, the exact numbers can be found in the table below the images. The number of photons gathered for each diffuse intersection is increased together with the number of photons shot, to guarantee a noise free picture. While the number of photons increases by a factor 10 for each picture, the number of gathered photons is increased by a factor of roughly 4. The number of photons directly influences the time needed to store enough photons in the scene, while the effects of increased photon gathering numbers significantly lengthens the rendering time only for the last two pictures. It can be seen that an increased number of photons enhances the sharpness of the caustics, but of course increase the time required to create the image. However, the visual importance of caustics for a realistic appearance becomes apparent.

Picture	Photons shot	Photons gathered	Photon tracing time	Ray tracing time
1	0	0	0s	253.3s
2	1.000	50	2.8s	268.3s
3	10.000	200	3.7s	292.1s
4	100.000	800	27.6s	406.8s
5	1.000.000	2.400	273.9s	810.1s
6	10.000.000	10.000	2870.6s	3308.1s

Table 7.1: Numbers for shot and gathered photons that were used to generate Figure 7.5.



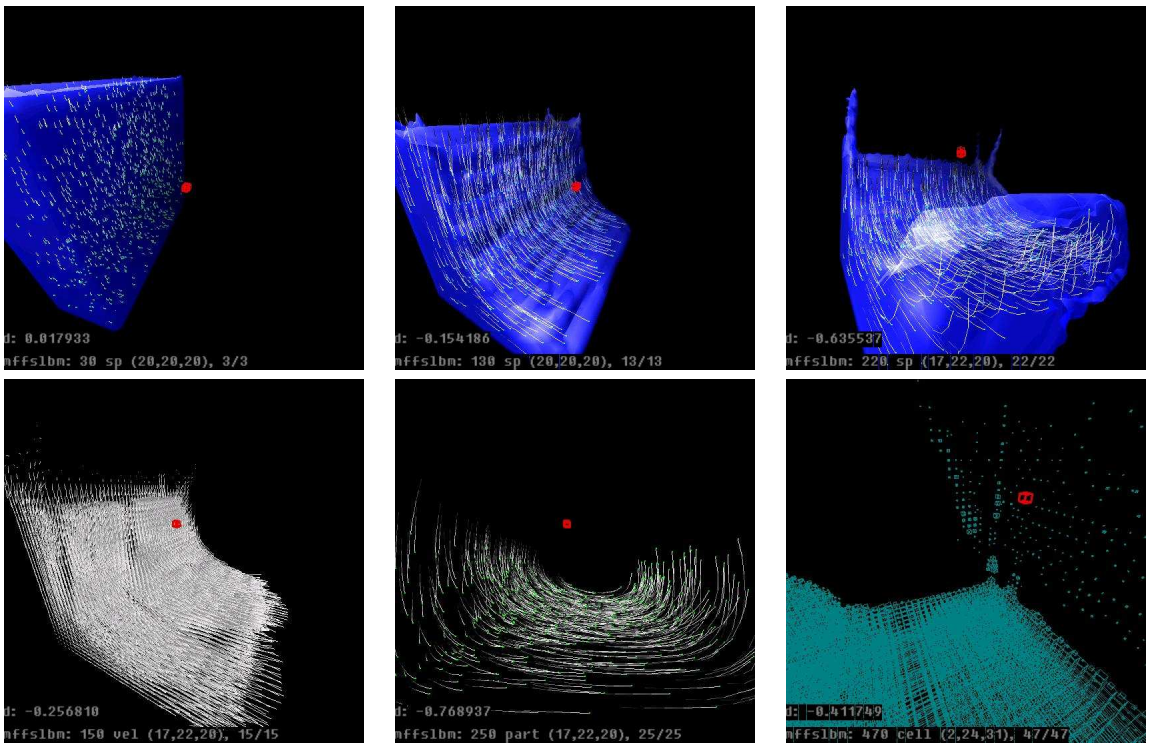


Figure 7.2: Several screenshots from the OpenGL visualization. The pictures in the upper row show several time steps from a running breaking dam simulation. The bottom row shows (from left to right) a view of the velocity field, the particle visualization and the fill levels.

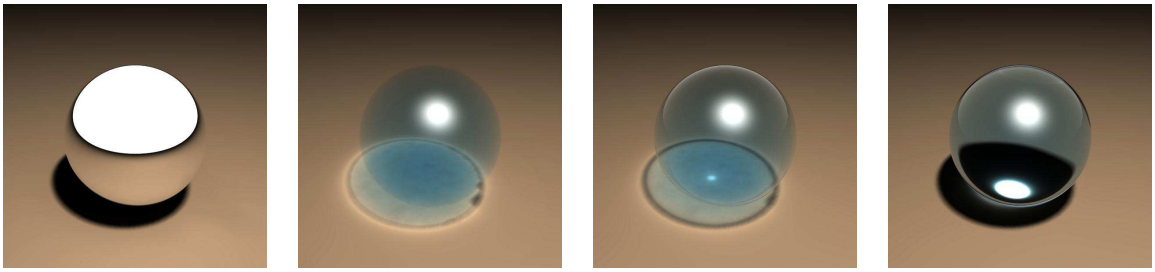


Figure 7.3: From the left to the right, a completely reflective sphere, a transmissive sphere and two spheres with Fresnel-coefficients for reflection and refraction can be seen. Only the rightmost sphere has a refraction index different to 1.

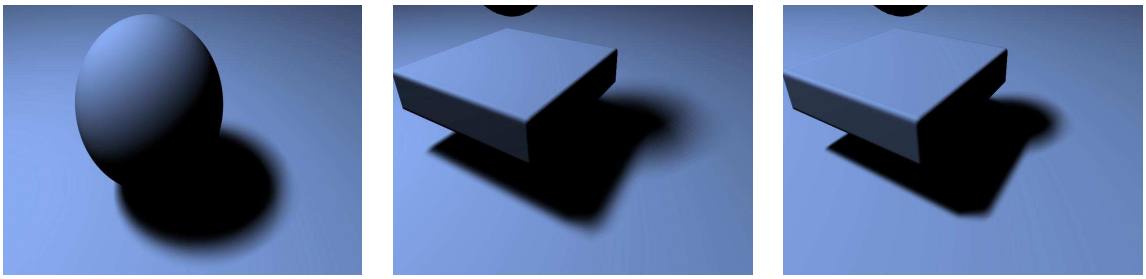


Figure 7.4: Two simple scenes demonstrating penumbras are shown in these pictures (a single sphere and a sphere only partly visible above a rectangular box). For the rightmost picture the light source was moved further away from the scene.

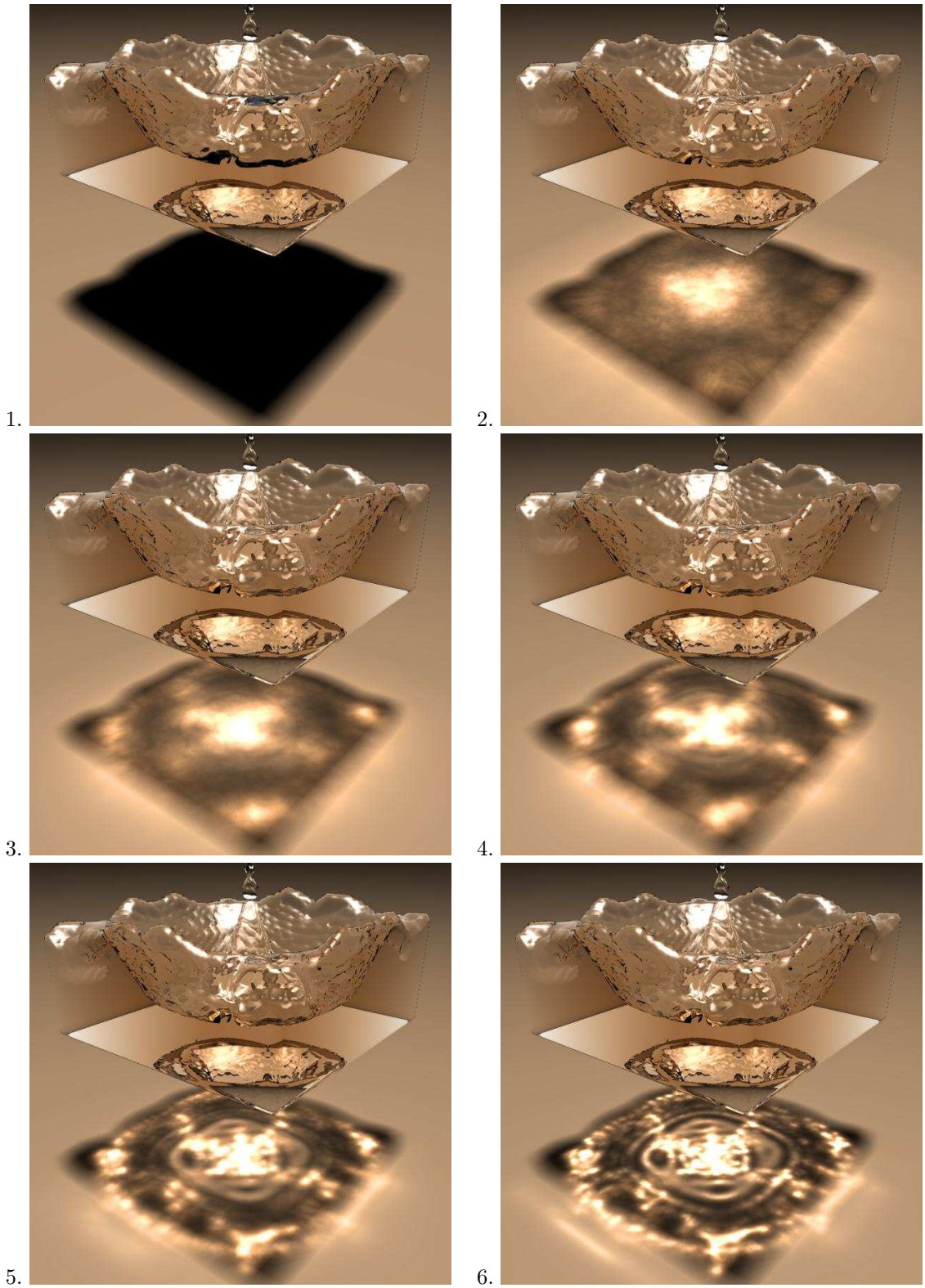


Figure 7.5: Images for the same scene with different numbers of photons shot and gathered can be seen. The exact numbers are shown in Table 7.1.

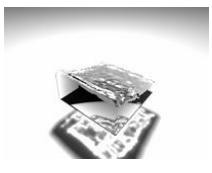
Chapter 8

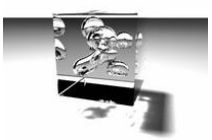
Conclusions

This thesis has presented a Lattice Boltzmann Method for a single fluid phase with a free surface and surface tension. The targeted field of application is the simulation of metal foams to enhance the production process. The fluid simulation uses the three-dimensional D3Q19 lattice and the LBGK method. The algorithm works with a topology representation of three different cell types – fluid, interface and gas cells. As the gas cells are only needed to track the bubble volumes, additional work is necessary at the interface cells. The missing information from the gas can be reconstructed with the fluid velocity and the bubble density. Furthermore it is necessary to explicitly track the mass at the interface cells, and reinitialize the cell types whenever interface cells are full or empty. Here the correct order of cell type changes is necessary to prevent disturbances of the fluid. The surface tension is calculated by reconstructing points on the isosurface using the marching cubes algorithm. For each interface cell two circles are constructed, that each touch three points of the isosurface in the neighborhood of the cell and lie in a coordinate plane. The average curvature is then used to apply a force on the fluid surface according to strength and direction of the curvature. For visualization a real-time OpenGL viewer and a raytracer with soft shadows and caustics were implemented.

As has been shown in Chapter 6, the version of the LBM developed in this thesis is applicable to a wide range of problems, yielding good results. The basic LBM is retained, although care has to be taken when handling the different cell types. When velocities in the lattice get too large, interface cells may not fill or empty fast enough and thus enhancements to the free surface boundary conditions may still improve the quality of the simulations. The surface tension calculation using marching cubes is relatively fast. The correct selection of points to prevent "flickering" and alternative isosurface generations may, however, require additional overhead, and are a topic of future research. It has been shown that the realistic parametrization of the LBM is relatively difficuilt, and correct paramertizations often would require large numbers of cells, resulting in huge memory requirements. This is, however, not problematic for simulations of metal foam, as these only require smaller spatial scales without strong external forces. The algorithm is especially well suited for simulations of foams, as the huge volumes of gas don't require additional computations.

It could be shown that the LBM solver of this thesis provides all necessary basic functionality for foaming simulations – the motion of the liquid metal phase is correctly reproduced. Future work will additionally include the gas diffusion, separation pressure and temperature into the program. It may even be possible to simulate the whole cooling process including the development of capillary cracks in the metal foam. For simulations with the SR8000, additional optimizations and an implementation in FORTRAN, also handling the domain decomposition and load balancing, will be necessary.



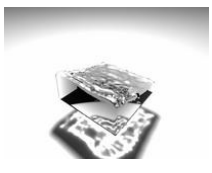


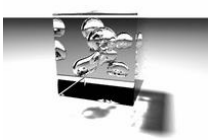
Chapter 9

Acknowledgements

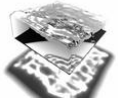
I would like to thank Prof. Dr. Ulrich Rüde and Thomas Pohl for the support. Furthermore Dr. Carolin Körner, Markus Kowarschik, Michael Thies, Jan Treibig and Thomas Zeiser helped me with various questions.

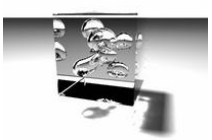
Thanks also to Mario Fritz, Harald Köstler, Michael Martinides and Stefan Thürey for the corrections.





List of Figures





Bibliography

Arnold, M., Thies, M., Körner, C., and Singer, R. (2000). Experimental and numerical investigation of the formation of metal foam. *Materialsweek*.

Bhatnagar, P. L., Gross, E. P., and Krook, M. (1954). A model for collision processes in gases. *Phys. Rev.*, 94.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *Computer Graphics*, 11:192–198.

Bogoliubov, N. (1962). Problems of a dynamical theory in statistical mechanics. In *Studies in Statistical Mechanics, Vol .1*, Amsterdam, Netherlands. J. de Boer, G. E. Uhlenbeck.

Bouzidi, M., d'Humières, D., Lallemand, P., and Luo, L.-S. (2001). Lattice Boltzmann equation on a two-dimensional rectangular grid. *JCP*, 172(2):704–717.

Bronstein, Semendjajew, Musiol, and Mühlig (1999). *Taschenbuch der Mathematik*. Verlag Harri Deutsch.

Buick, J. M. and Greated, C. A. (2000). Gravity in a lattice boltzmann model. *Pysical Review E*, 61.

Chen, H., Chen, S., and Mattheaus, W. (1992). Recovery of the navier-stokes equations using a lattice-gas boltzmann method. *Phys. Rev. A*, 45(8).

Coa, N., Chen, S., Jin, S., and Martinez, D. (1997). Physical symmetry and lattice symmetry in the lattice boltzmann method. *Phys. Rev. E*.

d'Humières, D. (1992). Generalized lattice-Boltzmann equations. In Shizgal, B. D. and Weaver, D. P., editors, *Rarefied Gas Dynamics: Theory and Simulation*, Progrss in Astronautics and Aeronautics, pages 450–458, Washington. AIAA.

d'Humières, D., Ginzburg, I., Krafczyk, M., Lallemand, P., and Luo, L.-S. (2002). Multiple-relaxation-time lattice Boltzmann models in three dimensions. *PhilTransRSocLondA*, 360(1792):437–452.

Durst, F. (2002). *Grundlagen der Strömungsmechanik*.

Foley, J. D., VanDam, A., and Feiner, S. K. (1995). *Computer Graphics in C. Principles and Practice*. Addison-Wesley.

Frisch, U., Hasslacher, B., and Pomeau, Y. (1986). Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56(14).

Frohn, A. (1979). *Einführung in die kinetische Gastheorie*. Akademische Verlagsgesellschaft Wiebaden.

Glassner, A. S. (1989). *An Introduction to Ray Tracing*. Harlekijn.

Griebel, M., Dornseifer, T., and Neunhöffer, T. (1988). *Numerical Simulation in Fluid Dynamics*. SIAM.

Gunstensen, A. K., Rothman, D. H., Zaleski, S., and Zanetti, G. (1991). Lattice boltzmann model of immiscible fluids. *Pysical Review A*, 43.

Hardy, J., de Pazzis, O., and Pomeau, Y. (1976). Molecular dynamics of a classical gas: Transport properties and time correlation functions. *Phys. Rev. A*, 13(5).

Harris, S. (1971). *An Introduction to the Theory of the Boltzmann Equation*. Holt, Rinehart and Winston Inc.

He, X. and Luo, L.-S. (1996). A priori derivation of the lattice boltzmann equation. *Rapid Communications*, 55.

He, X. and Luo, L.-S. (1997). Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation. *Physical Review*, E.

Jensen, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping*. A K Peters Ltd.

Körner, C., Berger, F., Arnold, M., Stadelmann, C., and Singer, R. (2000). Influence of processing conditions on morphology of metal foams produced from metal powder. *Materials Science and Technology*, 16:781–784.

Körner, C. and Singer, R. (1999). Numerical simulation of foam formation and evolution with modified cellular automata. *Metal Foams and Porous Metal Structures*, pages 91–6.

Körner, C. and Singer, R. (2000). Processing of metal foams - challenges and opportunities. *Advanced Engineering Materials*, 2(4):159–65.

Lallemand, P. and Luo, L.-S. (2000). Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *PRE*, 61(6):6546–6562.

Lorensen, W. and Cline, H. (1987). Marching cubes: A high resolution 3d surface reconstruction algorithm. In *Computer Graphics Vol. 21, No. 4*, pages 163–169.

Mucci, P. J., Dongarra, J., et al. (2003). Papi performance application programming interface. WWW page. http://icl.cs.utk.edu/projects/papi/.

Pohl, T. (2003). Freewihr. WWW page. http://www10.informatik.uni-erlangen.de/en/Research/Projects/FreeWiHR.

Quian, Y. H., d'Humières, D., and Lallemand, P. (1992). Lattice bgk models for navier-stokes equation. *Europhys. Lett.*, 17(6).

Shan, X. and Chen, H. (1993). Lattice boltzmann model for simulating flows with multiple phases and components. *Pysical Review E*, 47.

Swift, M. R., Orlandini, S. E., Osborn, W. R., and Yeomans, J. M. (1996). Lattice boltzmann simulation of liquid-gas and binary-fluid systems. *Pysical Review E*, 54.

Thies, M. (2000). Arbeitsbericht (numerischer teil).

Thürey, N. (2003). Fluid simulation with lbm. WWW page. http://www.ntoken.com.

Treibig, J. (2002). *Simulation von Gas-Feststoff-Mehrphasensystemen mit dem Lattice Boltzmann Verfahren*. PhD thesis, Universitaet Erlangen-Nuernberg, Erlangen, Germany.

Watt, A. and Watt, M. (1992). *Advanced Animation and Rendering Techniques*. Addison-Wesley.

Welander, P. (1954). On the temperature jump in a rarefied gas. *Arkiv Fysik*, 7.

Wilke, J. (2002). Performance optimization of lattice-boltzmann methods.

Wolf-Gladrow, D. A. (2000). *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer.

Zeiser, T., Brenner, G., and Durst, F. (2002). Application of the lattice boltzmann cfd method on hpc systems to analyse the flow in fixed-bed reactors. *High Performance Computing in Science and Engineering '02, Transactions of the High Performance Computing Center*, pages 439–450.