technische universität
dortmund

Diplomarbeit

Echtzeitfähige Interaktion von Festkörpern mit 2D Lattice-Boltzmann Flachwasserströmungen in 3D Virtual-Reality Anwendungen

Markus Geveler

18. August 2009

Gutachter:
Prof. Dr. Heinrich Müller
Prof. Dr. Stefan Turek

Fakultät für Informatik
Graphische Systeme (LS VII)
Technische Universität Dortmund
http://ls7-www.cs.tu-dortmund.de

Fakultät für Mathematik
Angewandte Mathematik und Numerik (LS III)
Technische Universität Dortmund
http://www.mathematik.tu-dortmund.de/lsiii/

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Simulation von Flüssigkeiten und Gasen, d.h. allgemein von Strömungen im weitesten Sinne mit Hilfe von Rechnersystemen (*Computational Fluid Dynamics*, CFD) ist sowohl mathematisch, als auch im Hinblick auf die Ansprüche bezüglich der Ressourcen Speicher und Rechenkapazität, ein komplexes Problem. Ingenieurstechnische Anwendungen, die eine hinreichend große Genauigkeit erfordern, genau wie Anwendungen, die eine möglichst hohe Rate bei der Verfügbarkeit von Ergebnisdaten benötigen, sind auch mit der schnellen Entwicklung der Rechenhardware der letzten Jahrzehnte noch starken Einschränkungen unterworfen. Zusätzlich verschärft wird das Problem dadurch, dass in der Entwicklung der Rechengeschwindigkeit durch Verbesserung von Prozessortechnik eine Stagnation aufgrund physikalischer Grenzen (etwa bei der Wärmeentwicklung) zu verzeichnen ist. Demgegenüber steht ein hoher Bedarf in Industrie und Forschung an hochleistungsfähiger Software für diesen Bereich: Anwendungen der CFD erstrecken sich von der Produktentwicklung, etwa bei der Simulation der Umströmung von Fahrzeugen jeglicher Art oder der Durchströmung von Maschinenbauteilen, über biomedizinische Anwendungen, wie bei der Berechnung des Blutflusses durch künstliche Herzklappen, bis hin zur Vorhersage von atmosphärischen oder ozeanographischen Strömungen in Wettervorhersage, Luft- und Raumfahrt oder Küstenbau und nicht zuletzt Anwendungen in der Unterhaltungsindustrie. Während hochgradig parallele Rechnersysteme in Form von Supercomputern sehr leistungsfähig sind, stellen sie aufgrund von hohen Kosten und geringer Verfügbarkeit für viele Anwendungen keinen Ausweg dar. Die Parallelisierung von Recheneinheiten hat sich in Form von Mehrkernprozessoren und alternativen Prozessorarchitekturen, als Reaktion der Chip-Hersteller auf das Ausflachen der Steigerung der Rechenleistung von einer Prozessorgeneration zur nächsten insbesondere in der letzten Dekade jedoch ebenfalls im Massenmarkt etabliert. Um die Leistungsfähigkeit moderner Mehrkernarchitekturen auszunutzen sind sowohl beim Entwurf von numerischen Methoden, als auch bei der Implementierung die jeweiligen Eigenarten der Zielhardware zu berücksichtigen, was die CFD zu einem hochgradig interdisziplinären Feld macht.

Ein konkretes Beispiel für die Anwendung einer Strömungssimulationssoftware ist deren softwareseitige Integration in eine virtuelle Echtzeitumgebung (*Virtual-Reality*, VR), wie in aktuelle 3D-Computerspiele, die eine beträchtliche volkswirtschaftliche Bedeutung

haben. Die wohl wichtigste Eigenschaft, die eine Softwarekomponente eines VR-Systems haben muss, ist die *Echtzeitfähigkeit*, d.h., dass zeitabhängige Lösungen der Strömungssimulation mit hinreichend hoher Frequenz berechnet werden müssen, so dass die Ausgabe (i.A. Darstellung, *rendering*) in für den Benutzer übergangsloser Weise erfolgen kann, so dass dieser in der Lage ist, auf sie zu reagieren. Eine nicht notwendige aber sehr wünschenswerte Eigenschaft ist die Möglichkeit der Interaktion von simulierter Flüssigkeit mit anderen Objekten (i.A. Festkörpern) in der virtuellen Szene, d.h. insbesondere die Möglichkeit der Kopplung von Fluidsimulation mit anderen Physikkomponenten. Die dreidimensionale Darstellung und insbesondere die Realisierung der Fluid-Struktur-Interaktion (FSI) erfordern dreidimensionale numerische Methoden zur Strömungssimulation, für die eine Echtzeitberechnung hinreichend großer Probleme auf Standard-PC Hardware noch praktisch unmöglich ist.

Die vorliegende Arbeit stellt ein Verfahren vor, zweidimensionale Strömungssimulation in dreidimensionalen Szenen zu verwenden, so dass die Flüssigkeit durch die Fluidoberfläche approximiert wird und trotzdem auf die dreidimensionale Szene (bzw. Festkörper in selbiger) reagieren kann und umgekehrt. Dabei stehen Effizienz, Stabilität, Einfachheit und Vielseitigkeit bzw. Erweiterbarkeit der numerischen Methode zur Strömungssimulation im Vordergrund, so dass sie insbesondere für die Computergrafik bzw. konkret die digitale Echtzeitbilderzeugung interessant ist. Zentral dabei sind die zweidimensionalen, inhomogenen Flachwassergleichungen (*Shallow Water Equations*, SWE) die das dreidimensionale Problem der Berechnung einer Fluidoberfläche über Tiefenmittelung lösen und insbesondere die zur Lösung verwendete numerische Methode: Die Lattice-Boltzmann Methode (LBM) hat sich vor allem in der letzten Dekade als vielversprechender Ansatz im Bereich der CFD erwiesen. Insbesondere wegen ihrer Effizienzeigenschaften im Hinblick auf moderne Rechnerhardware und ihre relative Einfachheit gegenüber anderen Methoden ist sie hier relevant. Rein methodologische Überlegungen führen in der Praxis alleine nicht zu einer Ausschöpfung des Potenzials der Zielhardware. Daher ist die *hardwareorientierte* Entwicklung des Verfahrens ein weiterer wesentlicher Bestandteil dieser Arbeit. Die Verwendung von Grafikkarten für das wissenschaftliche Hochleistungsrechnen (*high performance computing*, HPC) entwickelt sich seit Beginn dieses Jahrzehnts. Die Überlegenheit von GPUs gegenüber CPUs in Punkto Rechenleistung und die Entwicklung von Spracherweiterungen bzw. APIs für die Programmierung dieser Recheneinheiten haben zu einer breiten Verwendung von selbigen auch für andere Aufgaben als die Berechnung von grafik geführt (*General-Purpose Computation on Graphics Processing Unit*, GPGPU). Im Hinblick auf eine möglichst große Echtzeitleistung der Strömungssimulation wird die GPU in dieser Arbeit exemplarisch als Zielhardware verwendet. Ziel dieser Arbeit ist es, zu zeigen, dass auch eine Simulation komplexer Stromungen (im Hinblick auf die FSI) mit der Lattice-Boltzmann Methode unter Erhaltung ihrer Effizienz auf handelsüblicher Hardware möglich ist. Für die Computergrafik ist dabei das visuelle Ergebnis von Bedeutung. Es soll mög-

lich sein, mit der hier vorgestellten zweidimensionalen FSI-Fluidsimulation eine möglichst physikalisch korrekte Interaktion mit einer dreidimensionalen Szene zu erlauben, so dass der Eindruck eines (Echtzeit-) Flüssigkeits*volumens* entsteht.

1.2 Vorgehensweise und Struktur der Arbeit

Der Schwerpunkt dieser Arbeit liegt auf der hardwareorientierten Erweiterung der LBM hinsichtlich der Fluid-Struktur-Interaktion. Dazu ist es nötig, zunächst einige Grundlagen bezüglich HPC, hardwareorientierter Programmierung und GPUs zu erörtern. Kapitel 2 beschäftigt sich mit diesen Themen und diskutiert insbesondere das Konzept der verwendeten (und hier erweiterten) Numerik-Bibliothek HONEI (*Hardware Oriented Numerics Efficiently Implemented*). Die mathematischen Grundlagen der Strömungssimulation, d.h. insbesondere die Flachwassergleichungen und die Lattice-Boltzmann Methode werden in Kapitel 3 beschrieben, während in Kapitel 4 die mathematischen Erweiterungen hinsichtlich der Interaktion der Flüssigkeit mit Festkörpern bzw. allgemein mit der dreidimensionalen Umwelt beschrieben werden. Kapitel 5 beschreibt die konkrete Implementierung der in den Kapiteln 3 und 4 vorgestellten und erweiterten Algorithmen auf der CPU und der GPU. Letztlich werden die Ergebnisse bezüglich der numerischen und visuellen Qualität und der Leistung in Kapitel 6 dargestellt und diskutiert.

1.3 Bisherige Arbeiten in diesem Feld

Die dreidimensionale Darstellung von Flüssigkeiten ist in der Computergrafik ein wichtiger Bestandteil für eine realistische Darstellung einer Szene. Es gibt daher viele Publikationen, die sich mit diesem Feld beschäftigen, vergleiche beispielsweise die Arbeiten von Baboud et al., van der Laan et al., Peter et al. bzw. Yu et al. [2, 20, 28, 41]. Die Lattice-Boltzmann Methode wird dabei bereits in vielfältiger Weise eingesetzt und auf verschiedenen Hardwarearchitekturen implementiert [22, 29].

Die Simulation von Fluiden mit freien Oberflächen und mit sich in diesen bewegenden Festkörpern wurde bereits erfolgreich mit der Lattice-Boltzmann Methode implementiert. Dreidimensionale FSI, genauer die Kopplung eines 3D-Lattice-Boltzmann Lösers für die fluidseitige Simulation mit einem Finite-Elemente Löser für die Strukturmechanik stellen Geller et al. vor [12, 13]. Götz et al. bzw. Bogner koppeln jüngst ebenfalls dreidimensionale Lattice-Boltzmann Strömungssimulation mit bestehenden 3D-*Physik-Engines* [5, 16]. Thürey, Iglberger und Rüde implementieren einen 3D-Lattice-Boltzmann basierten Algorithmus für deformierbare bewegliche Festkörper [37]. Während diese Ansätze auf dreidimensionalen Berechnungen basieren, setzen Thürey et al. ein hybrides 2D/3D-Verfahren ein [36, 38], das eine zweidimensionale Flachwassersimulation mit einer dreidimensionalen interaktionsfähigen Strömungssimulation verbindet.

2 Wissenschaftliches Hochleistungsrechnen

2.1 Hardware

Die Entwicklung der Rechenhardware hin zu Mehr- und Vielkernarchitekturen (*Chip Multiprocessors*, CMP) konfrontiert den Entwickler hochleistungsfähiger Software für das wissenschaftliche Rechnen mit neuen Herausforderungen. Während sich bis zu Anfang dieses Jahrzehnts die Leistung von einer Prozessorgeneration zur nächsten noch stark verbesserte, haben physikalische Grenzen, Leistungsaufnahme und Wärmeentwicklung, diese Entwicklung, bezogen auf einen Rechenkern, beendet. Die fortwährende Erhöhung von Taktfrequenzen einzelner Recheneinheiten ist durch Produktionsprozesse und Grenzen bei der Signallaufzeit bzw. der Verlustleistung limitiert. Des Weiteren kann für einen einzelnen Prozess nicht ausreichend Parallelität im *instructions stream* durch die Hardware extrahiert werden, um Prozessorkerne beschäftigt zu halten, so dass durch superskalare CPUs keine größeren Verbesserungen der Rechenleistung mehr zu erwarten ist. Diese beiden Probleme, oft mit den Bezeichnungen *Power Wall* und *ILP Wall* assoziiert, haben dazu geführt, dass serielle Implementierungen nicht mehr automatisch durch eine Migration auf neue Hardware beschleunigt werden, da sich die Rechenhardware in Form von CMPs bereits im Massenmarkt etabliert hat. Während sich die Investition einer größeren Anzahl von Transistoren nun in der Vergrößerung der Anzahl von Recheneinheiten niederschlägt und damit die Leistungsfähigkeit in Bezug auf die reine Rechenleistung steigt, entwickelt sich die Geschwindigkeit von Speicherzugriffen wesentlich langsamer, ebenfalls induziert durch physikalische Beschränkungen (z.B. *pin limits*). Dadurch wird es immer schwieriger, die parallelen Recheneinheiten eines Mehrkernprozessors auszunutzen. Dieses Problem wird *Memory Wall* genannt.

2.1.1 *Memory Wall* und arithmetische Intensität

Das Problem hierbei ist im Grunde die Tatsache, dass die (theoretische) Rechenleistung mit der Anzahl der Recheneinheiten skaliert, die Speicherbandbreite jedoch mit der Anzahl der CPU-Sockel bzw. der Anzahl an Speichercontrollern. Traditionell wird dem durch größere Caches entgegen gewirkt, und durch bessere hardwareseitiger Organisation der Cache-Hierarchie und von Speicherzugriffen. Dies kann jedoch nur in begrenztem Maße

der *Memory Wall* entgegen wirken, da die Ausnutzung von Caches nur mit einem hohen Grad an Datenwiederverwendung seitens des auszuführenden Codes erfolgt. Dazu werden normalerweise fortschrittliche *blocking-* bzw. *prefetching*-Techniken eingesetzt, vergleiche hierzu beispielsweise Weidendorfer und Trinitis [39]. Eine wesentliche Schwierigkeit bei der Entwicklung von hochleistungsfähigen Softwarekomponenten ist daher, die *arithmetische Intensität* des Codes zu maximieren, falls dies möglich ist. Die arithmetische Intensität ist definiert als die Anzahl der durchgeführten Gleitkommaoperationen N_{flop} pro benötigtem Speichertransfer $N_{\text{transfer}} = N_{\text{load}} + N_{\text{store}}$ bezogen auf ein Datum, d.h.:

$$I_A = \frac{N_{\text{flop}}}{N_{\text{transfer}}}$$

Die Praxis zeigt, dass für numerische Berechnungen oft gilt, dass $N_{\text{flop}} \leq N_{\text{transfer}}$ ist. Typischerweise gilt für solche Operationen, dass sie *memory bound* sind, d.h., dass ihre Performance durch die Speicherbandbreite des Systems beschränkt wird, und nicht durch die Leistungsfähigkeit der Recheneinheiten. Im Hinblick auf die stark unterschiedliche Geschwindigkeit bei der Entwicklung von parallelen Rechnersystemen einerseits und der von Speichertechnologie andererseits sind jedoch zwingend *compute bound*-Operationen erforderlich um das Potenzial aller Recheneinheiten (*cores*) auszunutzen. Vor diesem Hintergrund verschärfen CMPs das Problem der *Memory Wall* dadurch, dass eine volle Ausnutzung der Rechenleistung immer schwieriger zu erreichen ist, da die Rechenkerne vermehrt auf Speicherzugriffe warten müssen. Daher rücken auch im Bereich des wissenschaftlichen Rechnens alternative Prozessordesigns immer mehr in den Fokus. Grafikprozessoren sind beispielsweise für einen möglichst großen Datendurchsatz optimiert und haben eine wesentlich höhere Bandbreite als Standard-CPUs.

2.1.2 GPGPU

Die immense Kaufkraft von Konsumenten von Unterhaltungssoftware hat die Entwicklung von Spezialhardware für diesen Bereich bereits seit langem vorangetrieben. Die hohen Anforderungen an die Hardware insbesondere bei aktuellen 3D-Computerspielen erfordern die Entwicklung von extra dafür ausgelegten Rechensystemen, die primär für die Erledigung von Aufgaben im Bereich des *Rendering* und der Berechnung von Physikeffekten oder andere Multimediaaufgaben verantwortlich sind. Prozessoren für aktuelle Spielekonsolen und die ursprünglich als Koprozessoren für die Beschleunigung von 3D-Grafik entwickelten GPUs sind daher insbesondere auch für einen hohen Datendurchsatz ausgelegt. Die *Cell Broadband Engine* (Cell BE), die hauptsächlich für den Einsatz in der PlayStation 3 entwickelt wurde, ist ein inzwischen populäres Beispiel für eine heterogene Mehrkernarchitektur, die mit ihren durch einen Ringbus verbundenen acht Gleitkommaeinheiten und einer Doppelkern Power 5 CPU für Steuerungsaufgaben eine hohe theoretische Rechenleistung von etwa 230 GFLOP/s in einfacher Genauigkeit bietet. Wesentlich wichtiger noch ist dabei, dass

Zugriffe auf den Hauptspeicher mit einer Rate von 26 GB/s erfolgen können - etwa doppelt so schnell wie bei Standard-CPUs. Aktuelle Grafikkarten erreichen sogar 1 TFLOP theoretische Rechenleistung bei einem Datendurchsatz von 160 GB/s[1]. Diese Leistungsfähigkeit macht gerade GPUs für das wissenschaftliche Hochleistungsrechnen interessant und hat zur Etablierung der Verwendung selbiger für andere Zwecke als die Berechnung von EchtzeitGrafik (*General-Purpose Computation on Graphics Processing Unit*, GPGPU) geführt. Die Implementierung von performanten und hinreichend genauen numerischen Softwarekomponenten unter anderem im Kontext der CFD demonstrieren beispielsweise Göddeke et al. [14, 15]. Während die Programmierung von GPUs anfänglich recht mühsam war und über eigens zugeschnittene Shader-Programme in die *Rendering-Pipeline* integriert werden musste, erleichtern dies inzwischen eigens dafür entwickelte Spracherweiterungen, vergleiche Abschnitt 2.2.3.

2.1.3 GPU Architektur

Aufbau

Die folgende Darstellung des Aufbaus einer GPU erfolgt am Beispiel der GT200-Serie von NVIDIA, da die Implementierung der in dieser Arbeit vorgestellten Erweiterungen der Lattice-Boltzmann Methode für NVIDIA GPUs erfolgt. Die GeForce GTX 285 basiert auf diesem Design.

GPUs sind hochgradig parallele Architekturen, da es ihre primäre Aufgabe ist, eine große Menge von Daten in möglichst kurzer Zeit zu verarbeiten. Der Aufbau eines solchen Grafikprozessors folgt daher einem modularen Paradigma: Recheneinheiten für die Gleitkomma- bzw. Integer- und andere arithmetische Berechnungen werden zu einer Recheneinheit zusammengefasst, diese wiederum werden kaskadiert und mit je einem geteilten Speicher für Instruktionen und Daten, sowie weiterer *special-purpose*-Recheneinheiten und einer Steuereinheit zu einem Multiprozessor aggregiert. Mehrere dieser Multiprozessoren bilden zusammen mit einer weiteren Steuereinheit und gemeinsam genutzten Speicher eine größere Einheit, usw.. Das Design kann somit einfach an verschiedene Größenordnungen bezüglich der *Performance* angepasst werden. Der schematische Aufbau eines GT200-Serie Grafikchips ist in Abbildung 2.1 dargestellt. Die atomaren Bestandteile einer GPU sind *special-purpose*-Recheneinheiten für die Gleitkomma- (*floating point* (FP)), Integer- (Int) und Address- bzw. Vergleichsarithmetik (*move and compare*), die zu einem Mikroprozessor, dem *Streaming Processor* (SP) zusammengefasst werden. SPs sind damit im Wesentlichen auf mathematische Berechnungen beschränkt und haben keinen eigenen Cache. Acht SPs zusammen mit zwei speziellen Recheneinheiten für oft benötigte mathematische Spezialaufgaben (trigonometrische Funktionen und Interpolation), die sogenannten SFUs (*Special Function Units*) werden zu den *Streaming Multiprocessors* (SM) aggregiert. Die Steuerung

[1] Mit einer NVIDIA GeForce GTX 285.

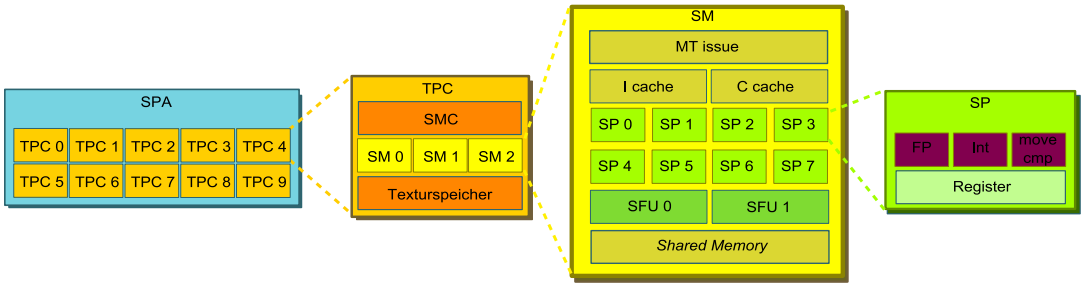


Abbildung 2.1: Vereinfachte schematische Darstellung einer NVIDIA GPU der GT200-Serie.

dieser Recheneinheiten übernimmt dabei die *Multithreaded instruction fetch and issue unit* (MT issue). Je ein kleiner Befehls- und ein Cache für Konstanten, sowie geteilter Speicher für die Daten gehören ebenfalls zu einem SM. *Streaming Multiprocessors* können 1024 Threads gleichzeitig ausführen. Dabei ist es besonders wichtig, dass *context-switches* zwischen Threads hardwareseitig durchgeführt werden, so dass keine Verzögerungen durch häufiges Umschalten zwischen Threads induziert werden. Je drei SMs können auf einen gemeinsamen Texturspeicher zugreifen, was durch den *Streaming Multiprocessor Controller* (SMC) gesteuert wird. Der Texturspeicher dient dabei als L1 Cache und ist direkt mit den physischen Textureinheiten auf der Grafikkarte assoziiert. Diesen Zusammenschluss von SMs nennt man *Texture Processor Cluster* (TPC). Schließlich werden 10 TPCs zu einem so gennanten *Streaming Processor Array* (SPA) aggregiert. Demnach hat ein GT200 Chip 240 Recheneinheiten (30 Multiprozessoren).

Speicherhierarchie und Funktionsweise

Auf unterster Ebene der Speicherhierarchie einer GPU befinden sich Hardwareregister, die die Daten aller Threads enthalten. Pro SM sind dies 16384 Register, was bei einer vollen Nutzung der 1024 Threads eine Anzahl von 16 Registern pro Thread ergibt. Jede SM stellt darüber hinaus ihren Recheneinheiten (d.h. SPs und SFUs) einen 16 kByte großen geteilten Speicher (*Shared Memory*) zur Verfügung. Auf den bis zu 2 GB großen Grafikkartenspeicher auf oberster Hierarchieebene kann auf zwei Arten zugegriffen werden: Als normaler Arbeitsspeicher oder als auf Lesezugriffe beschränkter Texturspeicher. Letzterer hat die Eigenschaft, in zwei Dimensionen gepuffert zu sein, was oftmals für zweidimensional angeordnete lokale Daten von Vorteil ist. Alle Threads können lesend auf einen zusätzlichen, sehr schnellen, 64 kByte großen geteilten Speicher der GPU zugreifen.

Die Ausführung eines GPU Programms ist durch das *Single Instruction Multiple Thread*-Prinzip[2], d.h. die parallele Ausführung einer einzelnen Instruktion (bzw. Folge von Instruktionen) mit vielen Threads, d.h. insbesondere auf unterschiedlichen Daten. Ebenso, wie die

[2]In Anlehnung an das bekannte SIMD-Prinzip (*Single Instruction Multiple Data*)

Hardware hierarchisch in einem zweidimensionalen Gitter angeordnet ist, erfolgt dies auch für die Menge aller Threads: Ein Gitter (*grid*) von Threads enthält Threadblöcke, die jeweils einem der SMs zugeordnet sind. Ein Threadblock ist also ein virtualisierter Multiprozessor. Diese Blöcke werden weiter in die sogenannten *warps* unterteilt, die aus jeweils 32 Threads bestehen. Die Threads eines solchen *warp* führen parallel dieselbe Instruktion aus. Die *MT issue* ist dabei für das Dispatching der *warps* zuständig, wenn eine SM bereit für die Ausführung der nächsten Instruktion wird. Die kleinste Organisationseinheit von Threads sind die *half-warps*. Jeweils die Hälfte der Threads eines *warp* können gleichzeitig auf den Grafikkartenspeicher zugreifen, dabei werden nach Möglichkeit die einzelnen Speichertransfers zu einem gemeinsamen Transfer zusammengefasst (*Coalescing*, vergleiche Kapitel 5).

Die Threads eines Threadblocks werden parallel auf einem SM ausgeführt. Wann immer ein solcher *block* (das heisst alle seine Threads) terminiert, wird ein neuer auf dem Multiprozessor zur Ausführung gebracht. Der Multiprozessor *mappt* jeden Thread auf einen seiner (skalaren) SPs. Ein *warp* führt eine Instruktion zu jedem Zeitpunkt aus - idealerweise sollten sich die Ausführungspfade seiner einzelnen Threads daher nicht unterscheiden (beispielsweise durch eine konditionale Verzweigung), da die einzelnen Ausführungspfade sonst seriell bearbeitet werden. Die Anzahl der Blöcke, die gleichzeitig auf einem SM bearbeitet werden können[3], hängt vom Bedarf des entsprechenden CUDA-Codes an Registern und *shared memory* ab, da diese auf alle Threads der gegebenen Blöcke aufgeteilt werden müssen. Die SMs sind daher die zentrale Einheit der GPU-Architektur: Sie erzeugen und verwalten die Threads und bringen sie zur Ausführung. Die oben erwähnte hardwareseitige Implementierung von *context-switches* und die von den SMs bereitgestellte Threaderzeugung erlauben zusammen mit einer hardwareseitigen Synchronisierung der Threads eine Parallelisierung auf Elementebene, wie beispielsweise das *mapping* von Threads auf die Bearbeitung einer Zelle in einer gitterbasierten Anwendung, wie im Fall der hier vorliegenden Arbeit.

2.1.4 Das CUDA-Programmiermodell

Die Implementierungen in dieser Arbeit basieren auf Version 2.2 von NVIDIA's CUDA *Compute Unified Device Architecture* [27] Das CUDA Programmiermodell basiert auf einer Erweiterung der Programmiersprache C, die Sprachkonstrukte zur Verfügung stellt, die darauf abzielen, den Entwickler in die Lage zu versetzen, die gesamte Speicherhierarchie der GPU explizit verfügbar zu machen. Dabei wird der CUDA C Compiler `nvcc` verwendet. Operationen, bzw. genauer der *device-code* für die GPU, werden mit CUDA seriell formuliert, d.h. man beschreibt die Arbeit eines einzelnen Threads, der dann entsprechend oft gestartet wird. Dies ermöglicht die Formulierung von numerischen Operationen

[3] Bis zu einem Maximum von acht Blöcken.

im üblichen „C-Stil“: Auf die Operanden wird jeweils wie auf ein Array mit der eindeutigen Thread-Nummer als Index zugegriffen. Die Thread-Nummer ist durch die Position im *Grid* (vergleiche Abschnitt 2.1.3) eindeutig bestimmt. Die eigentliche Operation ruft diese CUDA-*kernel* dann mit einer durch die Problemgröße (Beispielsweise die Länge der Vektoren) bestimmten Anzahl auf. Damit entspricht jeder Thread in diesem geräteseitigen Code auch einem Hardwarethread. So werden die Threads im Programmiermodell direkt mit den entsprechenden Komponenten der Hardwarehierarchie assoziiert, und wie im letzten Abschnitt beschrieben auf der Hardware zur Ausführung gebracht.

Ein CUDA-Programm besteht prinzipiell aus *host*-seitigem Code, der die Partitionierung des vorgegebenen Problems in ein *grid* von Threadblöcken vornimmt und den *device*-seitigen Code aufruft. Die Parameter und Daten für das zu lösende Problem werden wie an jede andere C-Prozedur als Argumente übergeben. Im Fall der Daten sind diese Argumente GPU-Speicheradressen. Diese können mit von der CUDA-Laufzeitumgebung zur Verfügung gestellten Methoden durch Allokation von Grafikkartenspeicher bezogen werden. An dieser Stelle soll auf die Angabe von expliziten Beispielen verzichtet werden, da die Beschreibung der Implementierung der in dieser Arbeit beschriebenen und erweiterten Algorithmen in Kapitel 5 die Beschreibung der wesentlichen CUDA-spezifischen Teile des Codes umfasst.

2.2 HONEI

2.2.1 Motivation

Der Entwickler numerischer Softwarekomponenten sieht sich durch die steigende Parallelisierung (und Heterogenisierung) der Hardware vor allem mit dem Problem konfrontiert, dass neben der Kenntnis von Aufbau und Funktionsweise der Zielhardware auch das Erlernen neuer Programmiertechniken oder gar Spracherweiterungen nötig ist. Zusätzlich zu der benötigten Anwendungsfunktionalität wird eine hardwarespezifische Laufzeitumgebung benötigt, die beispielsweise die Datentransfers von der koordinierenden Host-CPU (bzw. dem Hauptspeicher) zu den Koprozessoren wie etwa der GPU steuert. Ein Ansatz für die Entwicklung eigener Anwendungen oder Operationen für eine spezifische Hardware ist daher, auf bestehende herstellerseitige Bibliotheken (BLAS, LAPACK oder FFT) zurückzugreifen. Letztere sind zunächst insofern problematisch, als dass sie *closed source* sind und damit eine Möglichkeit der Erweiterung des Funktionsumfangs auf der Ebene der implementierten Operationen nicht gegeben ist. Damit ist man auf eine feste Herangehensweise eingeschränkt: Ganze Applikationen oder applikationsspezifische Operationen (*kernel*) werden aus solchen Basisoperationen zusammengesetzt, falls dies möglich ist. Zusätzlich erschwert wird dieser Ansatz durch das weitgehende Fehlen von Standardisierung über einzelne Bibliotheken hinaus. Die Implementierung komplizierterer, anwendungsspezifischer *kernel* durch Konkatenation von Aufrufen mehrerer einfacher Operationen kann

außerdem in vielen praktischen Situationen nicht die volle Leistungsfähigkeit der Hardware ausnutzen. Die Optimierung neuer *kernel* ist nicht zuletzt deshalb aufwändig, als dass Compiler im Allgemeinen noch nicht in der Lage sind, Code automatisch zu vektorisieren bzw. zu parallelisieren, was oft „handgemachte“ Lösungen wie die manuelle Assemblierung von Programmteilen in Maschinencode bzw. den Rückgriff auf compiler- bzw herstellerspezifische Spracherweiterungen erfordert[4].

Einerseits ist daher ein möglichst hoher Grad an Abstraktion von der Zielhardware erwünscht, um den Entwicklungsprozess zu erleichtern, während auf der anderen Seite das Ziel einer möglichst großen Flexibilität im Hinblick auf die Funktionalität bei gleichzeitiger Ausnutzung der Leistungsfähigkeit der Hardware steht. Die OpenSource C++ Numerik-Bibliothekssammlung HONEI (*Hardware-Oriented Numerics Efficiently Implemented*) wurde im Hinblick auf eine Erfüllung dieser konträren Ziele im Rahmen der Projektgruppe 512 an der Technischen Universität Dortmund entworfen [3], bei der der Author der hier vorliegenden Arbeit mitgewirkt hat. Dabei steht die Abstraktion von der Zielhardware im Vordergrund: HONEI stellt einerseits hardwarespezifische Implementierungen von häufig verwendeten Operationen aus dem Bereich der linearen Algebra, der numerischen Mathematik und der CFD für x86 Multicore und Distributed-Memory Architekturen, die Cell BE und NVIDIA GPUs bereit. Dadurch können Applikationen, die auf diesen Operationen aufsetzen, unmittelbar von der entsprechenden Hardware profitieren, was in dieser Arbeit von größerer Bedeutung ist. Auf der anderen Seite wird durch HONEI die Entwicklung eigener anwendungsspezifischer Funktionalität für diese Architekturen durch Bereitstellung von entsprechender Infrastruktur begünstigt: Die Implementierung von hardwarespezifischen Notwendigkeiten, wie der oben genannten Speichertransfers ist nicht nötig, da HONEI's Laufzeitumgebung diese übernimmt.

Die Leistungsfähigkeit des Ansatzes haben wir bereits in einer früheren Veröffentlichung am Beispiel eines Mehrgitterlösers für das Poisson-Problem und eines expliziten Lösers für die zweidimensionalen Flachwassergleichungen demonstriert [10]. Im Folgenden sollen die für diese Arbeit relevanten Entwurfsparadigmen und die Struktur von HONEI, sowie die Infrastruktur der GPU-Laufzeitumgebung beschrieben werden.

2.2.2 Generelle Konzepte

Architektur

Die Idee hinter einer Softwarebibliothek für numerische Anwendungen ist es, für die Effizienz der Anwendung kritische Teile in einigen *kerneln* zu konzentrieren, die separat implementiert, auf die Zielhardware portiert und optimiert werden können. Da HONEI verschiedene Hardwarearchitekturen unterstützt, ist dieses Konzept durch einen *frontend-backend*-Ansatz realisiert: Die Operations- bzw. Anwendungsbibliotheken (*frontends*) bil-

[4] wie beispielsweise die intrinsischen SSE-Befehle von Intel für die SIMD-Programmierung.

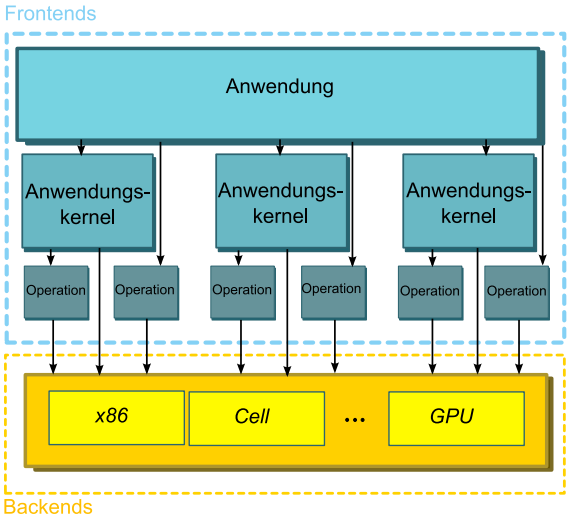


Abbildung 2.2: Hardwareabstraktionsstufen in HONEI

den die Schnittstelle zwischen Anwendung und hardwarespezifischen Implementierungen (*backends*). Sie stellen Datenstrukturen (*container*), sowie numerische Operationen und anwendungsspezifische Operationen bereit, während die *backends* die hardwarespezifische Implementierung und die zugehörige Infrastruktur enthalten, vergleiche Abbildung 2.2. Die Unterteilung in *frontends* und *backends* ermöglicht eine einheitliche Programmierschnittstelle für den Anwendungsentwickler. Details der hardwarenahen Implementierung sind für ihn nicht sichtbar. HONEI's *frontends* sind in C++ implementiert, so dass über generische Programmierung zwischen den verschiedenen *backends* differenziert wird. Für Details zur Nutzung von HONEI sei auf das Tutorial auf der Homepage des Projektes verwiesen [9].

Anwendungskernel

Die Implementierung einiger anwendungsspezifischer *kernel* ist nicht nur für die Bereitstellung der gewünschten Funktionalität erforderlich. Um eine möglichst hohe Leistung zu erzielen, ist es nicht ausreichend, stets kompliziertere *kernel* aus Basisoperationen zusammenzusetzen. Man betrachte dazu das Beispiel der folgenden Linearkombination, die für kleine (fest durch die entsprechende Anwendung gegebene) $n > 2$ bereits nicht mehr Bestandteil einer typischen BLAS-Implementierung ist:

$$\mathbf{y} \leftarrow \sum_{i=1}^{n} \alpha_i \mathbf{x}_i = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + ... + \alpha_n \mathbf{x}_n$$

Bei Verwendung einer Basisoperation für die gewichtete Summe zweier Vektoren[5] würde jedes mal, wenn die Operation terminiert, das Zwischenergebnis in den Hauptspeicher (bzw. den entsprechenden *device*-Speicher) transferiert werden müssen (*store*). Dieselben Daten

[5] Diese entspräche der BLAS Operation `AXPY`, $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$.

würden bei erneutem Aufruf der Operation zur Gleitkommaeinheit transferiert (*load*). Damit hätte man bei einer Vektorlänge von N eine Anzahl von benötigten Transfers von $N_{\text{transfer}} = O(2(n-1)N)$ bei gerade einmal $O((n-1)N)$ Rechenoperationen. Berechnet man das Datum $y^j = \sum_{i=1}^{n} x_i^j$, d.h. die j-te Komponente von $\mathbf{y}$, ohne Zwischentransfers, benötigt man lediglich etwa die Hälfte an Transfers: $N_{\text{transfer}} = O((n+1)N)$. Anhand dieses Beispiels ist offensichtlich, dass die arithmetische Intensität durch speziell zugeschnittene Anwendungskernel erhöht werden kann. Neben der Bereitstellung von bereits optimierten und portierten (auch applikationsspezifischen) Operationen, ist ein wesentliches Konzept von HONEI daher, dass die Entwicklung von solchen applikationsspezifischen Operationen erleichtert werden soll. Im Fall der GPU bedeutet dies insbesondere die Bereitstellung eines Speichermanagers, der für die Persistenz der Daten sorgt und unnötige Speichertransfers verhindert.

2.2.3 GPU (CUDA) *backend*

Um die Persistenz von Daten in einer Anwendung zu sichern, an deren Berechnungen sowohl CPU als auch GPU beteiligt sind, ist ein Speichermanagement unerlässlich. Es muss gewährleistet werden, dass von der CPU geänderte Daten auf der GPU (bzw. allgemein auf einem Koprozessor mit eigenem *device*-Speicher) aktualisiert werden und umgekehrt. Andererseits sollen einmal in den Grafikkartenspeicher transferierte Daten, die von keinem anderen *device* verändert werden, bei wiederholtem Zugriff nicht noch einmal transferiert werden müssen. Diese Aufgabe übernimmt in HONEI ein Speichermanager, `MemoryArbiter`, der von Ribbrock vorgestellt wird [32]. Dieser erlaubt die Implementierung von CUDA Operationen ohne dass explizite Speichertransfers durchgeführt werden müssen, was die Entwicklung von anwendungsspezifischen *kerneln* stark vereinfacht.

3 Strömungssimulation

3.1 Problemstellung

Die Simulation von Fluiden stellt bei hoher Auflösung des Rechengebiets alleine bereits beträchtliche Anforderungen an die zur Verfügung stehende Rechnerhardware, insbesondere im dreidimensionalen Fall. Die Idee, das dreidimensionale Problem der Berechnung einer physikalischen Quantität eines Fluids (etwa Wasserhöhe, Dichte, Temperatur, Geschwindigkeit) über ein lediglich zweidimensionales Verfahren durchzuführen liegt daher nahe. Im Folgenden werden die zu lösenden partiellen Differentialgleichungen beschrieben und die Lösungsmethode erläutert. Letztere soll dabei die Eigenschaft haben, effizient auf handelsüblicher Hardware berechenbar zu sein. GPUs sollen in der Lage sein, ein mit vernünftiger Auflösung diskretisiertes Fluid zu behandeln. Hierbei ist für diese Arbeit besonders relevant, dass aus den zeitlich diskreten Lösungen des Fluid-Lösers die gewünschte physikalische Quantität (für die Visualisierung zumeist die Wasserhöhe bzw. Oberfläche des Fluids) in Echtzeit gewonnen werden kann. Dadurch ist eine notwendige Eigenschaft der auszuwählenden Lösungsmethode, dass sie *echtzeitfähig* ist, d.h. eine Sequenz von Lösungen berechnet werden kann, so dass die Anzahl der (zeitabhängigen) Lösungen pro Sekunde mindestens so groß ist, dass das bilderzeugende System die Veränderung der Flüssigkeit in für das menschliche Auge übergangsloser Weise darstellen kann. [1]

3.2 Die Flachwassergleichungen

3.2.1 Motivation

Viele praktisch relevante Strömungen sind dadurch charakterisiert, dass bei guter Approximation des Fluids *vertikale* Effekte bei der Berechnung vernachlässigt werden können. Dies ist dann der Fall, wenn insbesondere eine Annäherung der Oberfläche des Fluids maßgeblich ist und es gilt:

$$1 > \frac{\Delta h}{\lambda}$$

[1] Es gibt unterschiedliche Definitionen von Echtzeit. In dieser Arbeit relevant ist jedoch ausdrücklich *nicht* die Eigenschaft, dass die Simulationszeit (also die Anzahl der Zeitschritte multipliziert mit der Größe der Zeitschritte im mathematischen Modell) der verstrichenen Realzeit entspricht. In Kapitel 6 werden wir diesen Begriff geeignet formalisieren.

dass also die Wellenlänge λ im Allgemeinen wesentlich größer als die Wellenhöhe Δh ist. Beispiele für solche Strömungen sind sog. *open channel flows*, unter die auch die Strömungen in Flüssen fallen. Sie sind eine spezielle Form der Strömungen mit freien Oberflächen, sogenannte *free surface flows*, bei denen die Ausdehnung des Fluids in positiver z -Richtung unbeschränkt ist. Weitere Beispiele sind sogenannte Dammbruchszenarien (*dambreak flows*), bei denen das Fluid von einem höher gelegenen Reservoir in ein tiefer gelegenes Gebiet strömt[2]. Von großer Bedeutung sind auch Flutungs- (und auch Abebb-) Vorgänge, bei denen die Flüssigkeit auf ein trockenes Gebiet strömt bzw. von diesem abfließt. Abbildung 3.1 zeigt die oben beschriebenen (und weitere) Strömungssimulationen im Prinzip.

Die Eigenschaft der Vernachlässigbarkeit von vertikalen Effekten (der Tiefenwirkung) einer Strömung erlaubt auf mathematischer Ebene eine erhebliche Vereinfachung des Modells, was im nächsten Abschnitt genauer betrachtet wird. Dabei wird gezeigt, dass die sogenannten *Flachwassergleichungen* nicht nur aufgrund dieser Vereinfachung für diese Arbeit relevant sind: Zusätzlich kann durch sie das prinzipiell dreidimensionale Problem der Bestimmung einer zeitabhängigen Flüssigkeitsoberfläche durch einen zweidimensionalen Ansatz approximativ gelöst werden. Des Weiteren beziehen diese Gleichungen einen Einfluss des Bodenprofils auf die Strömung (etwa durch Gefälle im Flussbett oder materialabhängige Reibung) auf natürliche Weise mit ein, was dann in Kapitel 4 genauer behandelt wird.

3.2.2 Herleitung

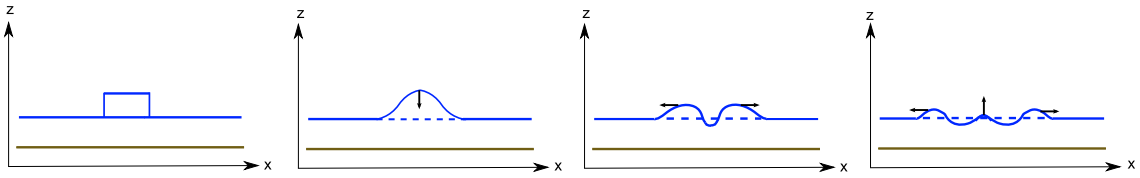
Kontinuitäts- und Navier-Stokes- Gleichungen

Gesucht ist ein System von partiellen Differentialgleichungen, so dass für jeden Zeitpunkt t und an jedem Ort $\mathbf{x} \in \mathbb{R}^d$ die Unbekannten des Systems die gesuchten physikalischen Quantitäten (und damit zunächst kontinuierliche Funktionen) sind. Im hier vorliegenden Fall ist, da eine Fluidoberfläche aus den Lösungen des Verfahrens konstruiert werden soll, zunächst die Tiefe (oder Höhe) $h(\mathbf{x}, t)$ des Fluids über dem Bodenprofil (der Höhe $b(\mathbf{x})$ des Bodens über der (x, y)-Ebene) von Relevanz. Der Zustand eines Flüssigkeitskörpers ist durch die Momentangeschwindigkeiten des Fluids an jedem Ort erst vollständig beschrieben. Dabei ist $\mathbf{u}(\mathbf{x}, t) = (u(\mathbf{x}, t), v(\mathbf{x}, t), w(\mathbf{x}, t))^T$ der Geschwindigkeitsvektor am Ort $\mathbf{x}$ zur Zeit t, vergleiche hierzu Abbildung 3.2.

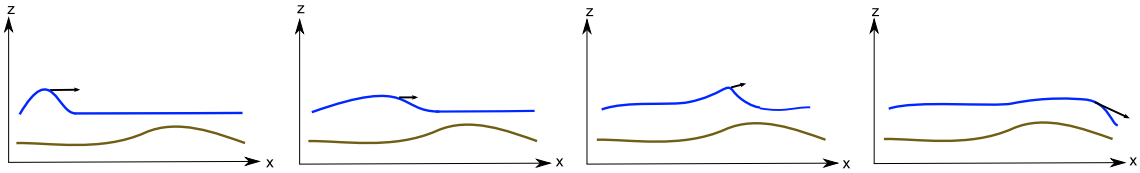
Die Flachwassergleichungen (*Shallow Water Equations* (SWE)) entstehen aus den 3D Kontinuitäts- und (inkompressiblen) Navier-Stokes-Gleichungen, die in Tensorform

$$\frac{\partial u_j}{\partial x_j} = 0 \tag{3.1}$$

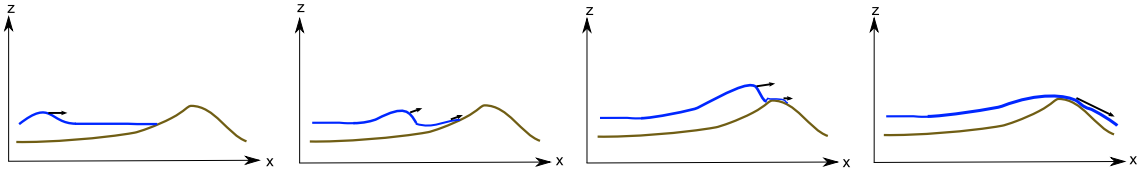
[2] Dabei sind nicht notwendigerweise Festkörper als Dämme vorhanden: Unter Dammbrüchen versteht man hier auch einen Vorgang, bei dem ein Fluidkörper (oft ein Quader oder Zylinder) über einem Gebiet, das ebenfalls mit der Flüssigkeit gefüllt ist, kollabiert, vergleiche auch Abbildung 3.1.



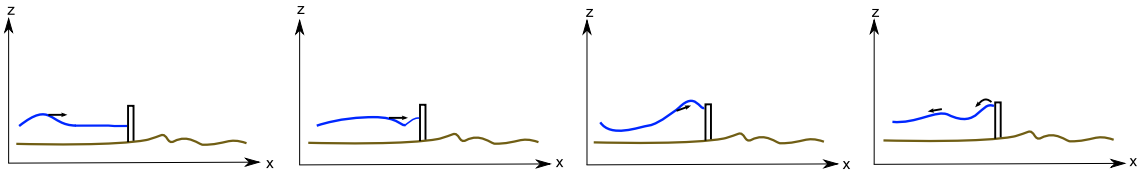
(a) unkritischer Volldammbruch



(b) unkritischer Fluss über unebenes Bodenprofil

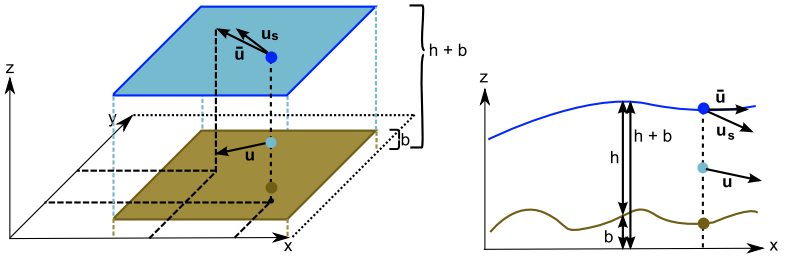


(c) Flutungsvorgang



(d) begrenzter Fluss

Abbildung 3.1: Beispiele für Flachwasserströmungen mit Ausgangssituation und Strömungszustand zu verschiedenen Zeitschritten (Seitenansicht): blaue Linie: Fluidoberfläche, braune Linie: Oberfläche der Bodentopographie. (a): Ein Fluidkörper (Quader oder Zylinder) kollabiert über einem Fluidgebiet, wie etwa bei einem Regentropfen. (b): Eine Strömung über unebenes Bodenprofil: Die Welle wird bei Anstieg der Steigung abgebremst und bei Gefälle beschleunigt. (c): *Dry-states*: Eine Welle erreicht trockenes Gebiet und wird an der Küste aufgetürmt, bevor sie es überflutet, wie etwa bei einem Tsunami. (d): Eine Strömung wird gestaut.



(a) 3D-Sicht (b) 2D-Sicht

Abbildung 3.2: Höhenfeld und Bodenprofil in den SWE, tiefengemittelte Variablen

bzw.

$$\frac{\partial u_j}{\partial t} + \frac{\partial (u_i u_j)}{\partial x_j} = f_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{3.2}$$

lauten. Dabei handelt es sich bei u_j um die entsprechende Komponente des Geschwindigkeitsvektors, wobei die Einsteinsummation verwendet wird, d.h. es ist $\frac{\partial u_j}{\partial x_j} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$ und ρ die Dichte, p der Druck und ν die Viskosität. Weiterhin ist f_i die i-te räumliche Komponente des Vektors der Kraft, die auf den Fluidkörper einwirkt. Der Vektor $\mathbf{f}$ wird auch *Volumenkraftdichte* genannt. Die beiden Gleichungen (3.1) und (3.2) sind die beherrschenden Gleichungen für *inkompressible* Strömungen, d.h., dass im Folgenden jegliche, durch interne oder externe Kräfte auftretende Kompression des Fluids vernachlässigt wird, was für diese Arbeit jedoch keine Einschränkung bedeutet. Die physikalischen Repräsentationen der einzelnen Terme der Navier-Stokes Gleichung (3.2) sollen im Folgenden kurz behandelt werden. Die gesamte linke Seite wird auch *Trägheitsterm* genannt. Sie beschreibt die Summe der totalen zeitlichen Änderung der Geschwindigkeit und der sogenannten *Konvektion* die durch den Konvektionsterm $\frac{\partial (u_i u_j)}{\partial x_j}$ ausgedrückt ist, d.h., dass dabei der Energietransport zwischen den Raumpunkten beschrieben wird, der durch die Bewegung von Teilchen induziert wird. Die Konvektionsbeschleunigung ist also ein Term für die Änderung der Geschwindigkeit eines Fluides durch Änderung seiner *Position*, wie man es etwa in einer sich verengenden Düse beobachtet. Die rechte Seite der Gleichung zerfällt in einen *Kraftterm*, einen *Druckterm* und einen *Viskositätsterm*. Auf der Erde wirken zunächst zwei Kräfte auf jeden Flüssigkeitskörper: Die *Coriolis*-Kraft wirkt in der (x, y)-Ebene und wird durch die Erdrotation induziert. Im Folgenden soll diese jedoch vernachlässigt werden, womit $f = (0, 0, -g)^T$ ist und die Gravitation als einzige in z-Richtung wirkende Kraft verbleibt und zwar bezogen auf das Einheitsvolumen[3]. Dabei ist g die Gravitationsbeschleunigung. Der Druckterm hingegen beschreibt die totale Veränderung des Drucks im Raum. Die Internalisierung von Viskosität durch $\nu \frac{\partial^2 u_i}{\partial x_j \partial x_j}$ ist dadurch erklärt, dass Viskosität als *Diffusion von Impuls* aufgefasst werden kann, d.h., dass auch der Impuls (das Produkt von Masse

[3]Damit sind zunächst Kräfte gemeint, die auf den gesamten Körper des Fluids einwirken, nicht jedoch ortsabhängigen Kräfte, die durch Gefälle oder Reibung im Bodenprofil entstehen.

und Geschwindigkeit eines Objektes, $\mathbf{p} = m\mathbf{v}$), ähnlich wie Partikel[4] dazu neigen sich von einem Ort größerer Partikeldichte zu einem Ort mit kleinerer zu bewegen, sich im Raum verteilt (in Fall von Flüssigkeiten sich also über die Fluidpartikel verteilt).

Übergang auf tiefengemittelte Quantitäten

Die Gleichungen (3.1) und (3.2) dienen nun als Ausgangspunkt für die Herleitung der Flachwassergleichungen[5]. Dazu müssen die *tiefengemittelten* Quantitäten h, u und v eingeführt werden[6]. Dies geschieht durch Integration über die Tiefe des Fluids (genauer: die absolute Höhe der Fluidoberfläche über Null, $h + b$): Im Falle der Kontinuitätsgleichung erhält man

$$\int_b^{h+b} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) dz = 0 \tag{3.3}$$

was man vereinfachen kann zu

$$\int_b^{h+b} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) dz + w_s - w_b = 0 \tag{3.4}$$

wobei w_s und w_b die vertikalen Geschwindigkeiten der freien Oberfläche des Fluids und des Bodenprofils sind, wie in Abbildung 3.2 dargestellt. An diese beiden (gesuchten) Größen muss nun jeweils die kinematische Anforderung gestellt werden, dass sie gleich den Summen der zeitlichen und räumlichen Änderungen der jeweiligen Größen $h + b$ bzw. b sind, d.h., es muss gelten:

$$w_s = \frac{\partial}{\partial t}(h + b) + u_s \frac{\partial}{\partial x}(h + b) + v_s \frac{\partial}{\partial y}(h + b) \tag{3.5}$$

und

$$w_b = \frac{\partial}{\partial t} b + u_b \frac{\partial}{\partial x} b + v_b \frac{\partial}{\partial y} b. \tag{3.6}$$

Substituiert man w_s und w_b in Gleichung (3.4) mit den Gleichungen (3.5) und (3.6), dann erhält man mit einigen Umformungen die *Kontinuitätsgleichung für Flachwasserströmungen*,

$$\frac{\partial h}{\partial t} + \frac{\partial (h\overline{u})}{\partial x} + \frac{\partial (h\overline{v})}{\partial y} = 0 \tag{3.7}$$

wobei

$$\overline{u} = \frac{1}{h} \int_b^{h+b} u dz$$

und

$$\overline{v} = \frac{1}{h} \int_b^{h+b} v dz$$

[4] In dieser Arbeit werden die Begriffe *Partikel* und *Moleküle* synonym verwendet. Gemeint ist stets ein sehr kleiner Volumenanteil der Flüssigkeit.

[5] Die folgende Herleitung richtet sich mit einigen Vereinfachungen nach Zhou [42].

[6] Im Folgenden wird $u = u_x$ bzw. $v = u_y$ geschrieben, um dDoppelindizierung zu vermeiden.

die tiefengemittelten Geschwindigkeiten in den jeweiligen Richtungen sind. Auf analoge Weise kann die Impulsgleichung (3.2) unter vernachlässigung der Viskosität und bei Abwesenheit externer Kräfte hergeleitet werden. Lässt man die Querstriche für die tiefengemittelten Quantitäten zur Vereinfachung weg und verwendet wieder die Einsteinsummation, so ergeben sich die (zweidimensionalen) Flachwassergleichungen zu

$$\frac{\partial h}{\partial t} + \frac{\partial (h u_j)}{\partial x_j} = 0 \tag{3.8}$$

$$\frac{\partial h u_i}{\partial t} + \frac{\partial (h u_i u_j)}{\partial x_j} + g \frac{\partial}{\partial x_i}(\frac{h^2}{2}) = 0. \tag{3.9}$$

Der Grund dafür, dass in der Impulsgleichung (3.9) der Druck nicht mehr auftaucht, ist die sogenannte *hydrostatische Druckapproximation für Flachwasserströmungen*, die auf den folgenden Überlegungen beruht:

1. Die vertikale Beschleunigung des Fluids ist vernachlässigbar klein und damit $w = 0$, wodurch die Impulsgleichung in z-Richtung vereinfacht werden kann zu $\frac{\partial p}{\partial z} = -\rho g$. Insbesondere kann der Druck dann nach Integration als eine Größe dargestellt werden, die neben der Wasserhöhe nur noch von der Dichte, der Gravitation und dem *atmosphärischen Druck* an der Wasseroberfläche abhängt.

2. Räumliche Unterschiede im atmosphärischen Druck sind wiederum vernachlässigbar und werden als Null angenommen. Da der Druck nach Integration über die Tiefe linear von ρ abhängt, löscht sich die Dichte ebenfalls aus der Gleichung.

Aus diesen Gründen ist auch die Gravitation als Kraft in vertikaler Richtung in der Impulsgleichung präsent.

3.2.3 Physikalische Interpretation, Anfangs- und Randbedingungen, Flachwassersimulation

Die im vorherigen Abschnitt hergeleiteten Gleichungen (3.8) und (3.9) können in Form eines Erhaltungssatzes geschrieben werden. Sei dazu der Vektor der makroskopischen Größen (Masse und Impulse dargestellt durch das Produkt von Masse und Geschwindigkeiten) $Q = (h, h \cdot u, h \cdot v)^T$, dann ist

$$\frac{\partial}{\partial t} Q + \frac{\partial}{\partial x} F(Q) + \frac{\partial}{\partial y} G(Q) = 0 \tag{3.10}$$

mit

$$F(Q) = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2} g h^2 \\ huv \end{pmatrix}, \qquad G(Q) = \begin{pmatrix} hu \\ huv \\ hv^2 + \frac{1}{2} g h^2 \end{pmatrix}, \tag{3.11}$$

wenn man weiterhin die Abwesenheit von externen und internen Kräften annimmt. Die Funktionen F und G können dabei jeweils als Fluss über ein Kontrollvolumen in den jeweiligen Raumrichtungen interpretiert werden. Die Bedeutung dieser Erhaltungsform ist, dass sich an jedem Ort des Fluids (im diesem Fall also an jedem Punkt der freien Oberfläche über einem Ort (x, y) in der Parameterebene) die zeitliche Veränderung von Tiefe und Geschwindigkeiten in der Summe mit den räumlichen Änderungen der Flüsse in x- und y-Richtung gegenseitig aufheben müssen, damit die Kontinuität dieser Größen gewährleistet ist.

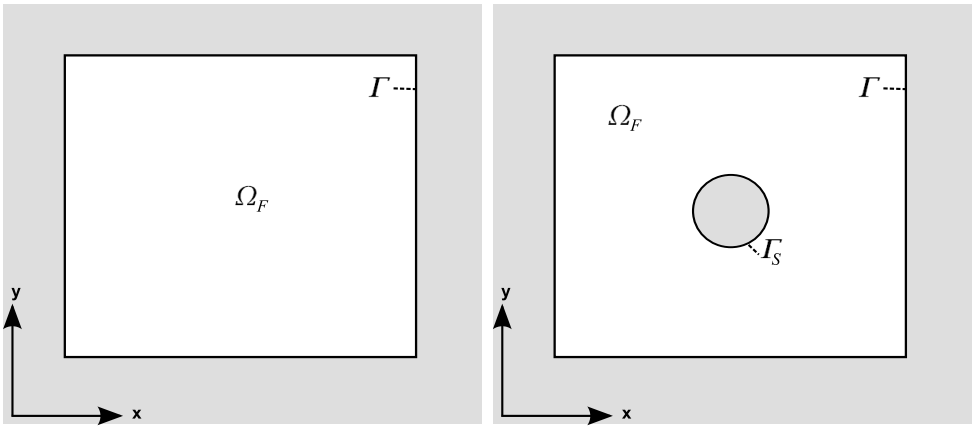
Um ein Strömungssimulationsproblem konkret zu formulieren benötigt man neben den Gleichungen für das Verhalten der physikalischen Eigenschaften des Fluidkörpers noch eine Definition des *Kontrollgebietes*, also der Grenzen des Fluidkörpers oder des Parameterbereiches, über dem das Fluid definiert ist. In Abbildung 3.3 ist das Kontrollgebiet (im Folgenden insbesondere im diskreten Fall auch *Rechengebiet* genannt) ein rechteckiger Ausschnitt $\overline{\Omega} \subset \mathbb{R}^2$. Eine solche Begrenzung erzwingt jedoch die Festlegung geeigneter Randbedingungen, da Strömungsgleichungen auf Differentialoperatoren beruhen und damit am Rand des Gebietes die entsprechenden Größen für die zeit- und ortsabhängigen Variablen in auswärtiger Richtung fehlen. Dazu sei das *verallgemeinerte Fluid* $\overline{\Omega}$ für diese Arbeit zunächst so definiert, dass gilt

$$\overline{\Omega} = \Omega_F \cup \Gamma.$$

Zur Notation vergleiche Abbildung 3.3. Für jedes $\mathbf{x} \in \Gamma$ sind nun geeignete Randbedingungen zu wählen, so dass $\mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t)$ ist, d.h., dass $\mathbf{u}_B$ ein Skalarfeld mir vorgegebenen Werten (*Dirichlet'schen* Randwerten) ist. Diese können zeit- und/oder ortsabhängig sein, oder global vorgegeben werden. In Kapitel 4 wird dies im Kontext der numerischen Methode genauer erörtert. Zusätzlich zu diesen Bedingungen werden noch Anfangsbedingungen benötigt, die die initialen Größen für h und $\mathbf{u}$ festlegen. Diese können ortsabhängig sein und ein intuitives Beispiel dafür ist eine global vorgegebene gerichtete Geschwindigkeit $\mathbf{u}_0$ mit einer vorgegebenen Funktion für die Wassertiefe, $h_0(\mathbf{x}, 0)$, die beispielsweise den auf die Fluidoberfläche aufgesetzten Zylinder in Abbildung 3.2.1(a) beschreibt. Mit diesen Vorüberlegungen und Definitionen ist das Problem, eine zeitabhängige Lösung für die 2D-Flachwassergleichungen zu bestimmen, folgendermaßen definiert, wobei die Randbedingungen global vorgegeben sind und Dirichlet-Randbedingungen verwendet werden:

$$\begin{cases} \frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = 0, & \mathbf{x} \in \Omega_F, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t) = 0, & \mathbf{x} \in \Gamma, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}, t = 0 \\ h(\mathbf{x}, t) = h_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}, t = 0 \end{cases} \tag{3.12}$$

Es sei bemerkt, dass außer für wenige Spezialfälle normalerweise keine analytischen Lösungen für die SWE zur Verfügung stehen. Dies gilt auch für die allgemeinen Strömungsglei-



(a) Rechteckiges Rechengebiet

(b) Rechteckiges Rechengebiet mit kreisförmigem Loch

Abbildung 3.3: Einfache Rechengebiete: (a): rechteckig, (b): rechteckig mit Hindernis

chungen (Navier-Stokes-Gleichungen). Im Laufe dieser Arbeit wird dieses einfache Modell für eine homogene Flachwasserströmung (d.h. insbesondere ohne Anwesenheit eines Kraftterms) inkrementell erweitert, um ein möglichst großes Spektrum verschiedener Strömungsszenarien berechnen zu können. Bevor auf die numerischen Lösungsverfahren eingegangen werden wird, soll hier für das Problem (3.12) bereits die Erweiterung auf kompliziertere Rechengebiete vorgenommen werden, die im Folgenden als Ausgangspunkt für weitere Spezialisierungen dienen soll. Bisher betrachtet wurden lediglich einfache Kontrollgebiete rechteckiger Form, wie die in Abbildung 3.3(a). Für komplexe Strömungsvorgänge ist es jedoch unerlässlich, einen allgemeineren und damit flexibleren Begriff eines Rechengebietes zur Verfügung zu haben. Dabei sollen prinzipiell beliebig geformte Untermengen des $\mathbb{R}^2$ als Rechengebiete dienen. Ein oft verwendetes Szenario ist beispielsweise das in Abbildung 3.3(b) dargestellte rechteckige Gebiet mit einem kreisförmigen Loch, über dem das Fluid nicht definiert ist. Damit stellt dieser Kreis in der physikalischen Repräsentation ein stationäres Hindernis mit zur Parameterebene senkrechten Wänden dar, das als Festkörper interpretiert werden kann, der weder beweglich noch deformierbar ist und dessen Höhe unbeschränkt ist. Sei Γ_S der Rand des stationären Hindernisses, dann ist Problem (3.12) so zu modifizieren:

$$\begin{cases} \frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = 0, & \mathbf{x} \in \Omega_F, t > 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_B(\mathbf{x},t) = 0, & \mathbf{x} \in \Gamma, t > 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_0(\mathbf{x},t), & \mathbf{x} \in \overline{\Omega}, t = 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_B(\mathbf{x},t) = 0, & \mathbf{x} \in \Gamma_S, t > 0 \\ h(\mathbf{x},t) = h_0(\mathbf{x},t), & \mathbf{x} \in \overline{\Omega}, t = 0 \end{cases} \tag{3.13}$$

Problem (3.13) betrachtet demnach die Gebiete, die nicht zu Ω gehören, bezogen auf das Fluid als Hindernis, bezieht aber eine mögliche Bewegung der senkrechten Wände dieser

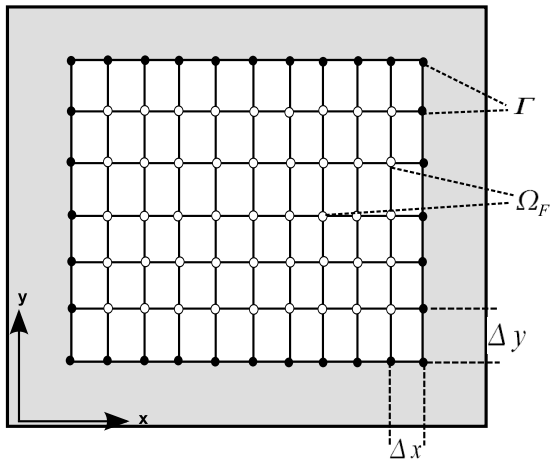


Abbildung 3.4: Räumliche Diskretisierung mit einem Rechtecksgitter

Festkörper, wie sie etwa bei einem rotierenden Zylinder vorliegen würde, nicht mit ein (vergleiche Kapitel 4). Für dieses Problem soll nun eine numerische Lösungsmethode gefunden werden.

3.2.4 Lösungsverfahren

Diskretisierung

Um mit Hilfe von Rechensystemen eine Lösung für die SWE errechnen zu können, ist es nötig die kontinuierlichen Gleichungen zunächst sowohl räumlich, als auch zeitlich zu *diskretisieren* um die gesuchten Quantitäten für eine endliche Menge von Punkten in der Parameterebene und zu festen Zeitpunkten zu ermitteln. Für die räumliche Diskretisierung wird ein Gitter eingeführt, durch dass $\overline{\Omega}$ in eine endliche Menge von Simplexen überführt wird, deren Knoten das diskretisierte Rechengebiet darstellen, vergleiche Abbildung 3.4, die die räumliche Diskretisierung des einfachen Rechengebietes (siehe oben) mit einem Rechtecksgitter zeigt. Diese liefert die Gitterschrittweiten Δx und Δy. Unter zeitlicher Diskretisierung ist die Einführung einer diskreten Menge von Zeitpunkten $T = \{t_0, t_0 + \Delta t, ..., t_0 + k\Delta t, ..., t_0 + n\Delta t\}, 1 \leq k \leq n$ zu verstehen, zu denen die gesuchten Größen berechnet werden, wobei Δt die Länge des Zeitschritts darstellt.

Direkter Ansatz, makroskopische Verfahren

Strömungsgleichungen wie die SWE werden in der Regel mit einem direkten Ansatz gelöst, d.h., dass sie mit einem geeigneten finiten Ansatz (Finite-Differenzen-, Finite-Volumen- oder Finite-Elemente-Methode) zunächst diskretisiert werden, um dann in ein System von Gleichungen überführt werden zu können, dass entweder *implizit* oder *explizit* gelöst wird. Bei impliziten Verfahren hängen die zu berechnenden Größen von anderen Größen zum

selben Zeitpunkt ab, weshalb das Verfahren auf ein (zumeist sehr großes) lineares Gleichungssystem führt, während diese Abhängigkeit bei expliziten Verfahren nicht vorhanden ist. Die Unbekannten zu jedem diskreten Zeitpunkt hängen nur von Werten anderer Variablen zum vorherigen Zeitpunkt ab, wodurch nur explizite Rechnungen vonnöten sind. Die Diskretisierung der Zeit erfolgt entweder durch den Euler'schen Ansatz oder die Lagrange'sche Integrationsmethode, vergleiche hierzu die Diplomarbeit von Becker [4], in der auch ein implizites Verfahren für die Lösung der SWE beschrieben wird. Ein Beispiel für ein explizites Verfahren ist das von Delis und Katsaounis entwickelte Relaxationsschema [8]. Während implizite Verfahren auf der Lösung sehr großer linearer Gleichungssysteme beruhen und damit oft mit dem Nachteil verbunden sind, für die Assemblierung von Systemmatrix und der rechten Seite des Systems viel Rechenzeit investieren zu müssen, da diese oft mit irregulären Zugriffsmustern auf den Speicher einhergehen und typischerweise ein sehr schlechtes Verhältnis von nötigen Speichertransfers zu Gleitkommaoperationen haben (also *memory bound* sind, vergleiche Kapitel 2), führen explizite Ansätze für die Lösung der SWE zu numerischen Unsicherheiten (insbesondere Stabilitätsproblemen) und sind im allgemeinen auch nicht ohne weiteres echtzeitfähig, da besondere numerische Behandlung für beispielsweise die Quellterme nötig werden.

Indirekter Ansatz, mikroskopische Verfahren

Eine völlig andere Herangehensweise ist die indirekte Lösung von Strömungsgleichungen, bei denen mit einem diskreten Ansatz begonnen wird um dann zu zeigen, dass die Strömung sich im Ergebnis im Einklang mit Strömungsgesetzen befindet. Die im Kontext dieser Arbeit wohl am weitesten verbreitete dieser Methoden, insbesondere im interaktiven Bereich, ist die partikelbasierte und damit *mikroskopische* Methode der *Smoothed Particle Hydrodynamics* (SPH). Partikelbasiertes Berechnen von Fluiden hat gegenüber den oben beschriebenen makroskopischen Verfahren eine Reihe von Vorteilen. Da alle Berechnungen für jeden Partikel separat durchgeführt werden, sind die Strömungen nicht an ein finites Gitter gebunden, und sind daher besonders in interaktiven virtuellen Umgebungen höchst flexibel, da sie prinzipiell überall hinfließen können, was mit klassischen finiten Ansätzen nur mit einer sehr großen Anzahl von Gitterpunkten bzw. adaptiven Gittern möglich ist. Zusätzlich sind diese Verfahren leicht in andere Physiksimulationen zu integrieren: Partikelweise Interaktion (hauptsächlich Kollision) mit anderen Partikeln (beispielsweise denen einer anderen Flüssigkeit, eines Gases oder Festkörpers) oder generell mit anderen Szenenobjekten, die beispielsweise aus Dreiecksnetzen bestehen, ist verhältnismäßig einfach und insbesondere effizient implementierbar. Der wesentliche Nachteil partikelbasierter Strömungssimulation gegenüber Ansätzen mit finitem Raumgitter liegt darin, dass die Rekonstruktion des Fluidkörpers bzw. seiner Oberfläche aufwändig ist und zumeist, abhängig vom gewählten Darstellungsverfahren, „gelartig“ oder unzusammenhängend und

damit irreal wirkt. Auf Modellierungsebene sind klassische Ansätze sehr kompliziert und lassen nur schwer Modifikationen zu, da diese stets auf den partiellen Differentialgleichungen über die makroskopischen Quantitäten erfolgen, führen aber direkt zu einer glatten Lösung für die Fluidoberfläche. Verfahren auf Partikelebene haben diesen Nachteil nicht, erlauben jedoch die Rekonstruktion einer Fluidoberfläche oft nur durch eine beträchtliche Anzahl von Partikeln, die auch aktuelle Computerhardware stark fordert.

Indirekter Ansatz, mesoskopische Verfahren

Aus dieser Sicht ist ein Verfahren, dass zu möglichst großen Anteilen *compute bound* ist und nach Möglichkeit höchst lokal auf seinen Daten arbeitet (idealerweise elementweise, d.h. ohne komplizierte und zeitaufwändige Zugriffe auf Daten in (bezogen auf die Koordinaten des entsprechenden Ortes) der Nachbarschaft) erwünscht, insbesondere im Hinblick auf die Entwicklung der Rechenhardware hin zu hochgradig parallelen Architekturen. Andererseits sollte eine Rekonstruktion einer Fluidoberfläche einfach möglich sein und zu visuell ansprechenden Ergebnissen führen, bei gleichzeitiger Erhaltung der Eigenschaft, dass im diskreten Modell eine Erweiterung des Verfahrens leicht möglich ist (im Hinblick auf das Ziel dieser Arbeit, mit der Umgebung und anderen Physikmodellen interagieren zu können) und die Effizienzeigenschaften des Verfahrens dabei nicht verloren gehen. In der letzten Dekade hat sich die Lattice-Boltzmann-Methode (LBM) insbesondere aufgrund ihrer relativen Einfachheit gegenüber direkten Lösungen von Strömungsgleichungen und ihren Effizienzeigenschaften etabliert. Der Tatsache zum Trotz, dass es sich um ein relativ junges Verfahren handelt (die LBM in ihrer am weitesten verbreiteten Form wurde erst Anfang der 1990er Jahre erstmals eingesetzt, vergleiche Qian [31]), gibt es bereits eine Fülle von Anwendungen und Erweiterungen der Methode. Während bei den bisher beschriebenen Methoden die Variablen und das Ergebnis entweder makroskopisch oder mikroskopisch sind, ist dies bei der LBM anders: Hier werden die Berechnungen auf mikroskopischer Ebene durchgeführt, wobei die Bewegung von Partikeln auf wenige, fest vorgeschriebene Richtungen beschränkt wird. Dies erlaubt eine effiziente Rekonstruktion der makroskopischen Quantitäten, insbesondere der Fluidoberfläche.

3.3 Die Lattice-Boltzmann Methode

3.3.1 Motivation

Dieses Kapitel führt in die LBM ein und leitet die Lattice-Boltzmann-Gleichung her, wobei mit der historischen Entwicklung begonnen wird. Dann soll für den konkreten Fall der SWE eine Lattice-Boltzmann Methode beschrieben werden. Obwohl sich letztlich herausgestellt hat, dass man die LBM als eine Finite Differenzen diskretisierte Lattice-Boltzmann Gleichung ansehen kann, ist die Methode urprünglich aus den zellulären Automaten (*cellular*

Automata, CA) und den Lattice-Gas-Automaten (*Lattice-Gas-Cellular-Automata*, LGCA) entstanden. Da die mikroskopischen Vorgänge in dieser Arbeit von großer Bedeutung sind, ist es sinnvoll mit der Entwicklung der Methode bei den gasdynamischen Vorgängen und damit in historisch korrekter Reihenfolge zu beginnen. Die folgende Darstellung der CAs und LGCAs ist auf das für diese Arbeit Wesentliche beschränkt. Eine umfassende und anschauliche Einführung liefert das Buch von Wolf-Gladrow [40]. Mathematisch sehr detailliert sind die Bücher von Hänel [17], nach dessen Beschreibung der Boltzmann-Gleichung sich die Darstellung in Abschnitt 3.3.2 richtet, und Succi [35]. Letztlich wird für die Beschreibung der konkreten Methode für die SWE das Buch von Zhou [42] herangezogen.

3.3.2 Herleitung

Zelluläre Automaten

Um die Darstellung der Funktionsweise und Eigenschaften der CAs auf das für diese Arbeit Wesentliche zu beschränken, soll an dieser Stelle direkt mit der Beschreibung von *zweidimensionalen* CAs begonnen werden. Zweidimensionale zelluläre Automaten bestehen im Wesentlichen aus

1. einer *regulären* Anordnung von Zellen gleichen Typs, die jeweils eine *endliche* Anzahl von Zuständen haben können: Dabei bedeutet die Regularitätsforderung, dass es sich bei den Zellen um (im zweidimensionalen Fall) achsenparallele Rechtecke handelt,

2. einer Definition einer *Nachbarschaftsrelation*, die für jede Zelle im Gitter die Nachbarn liefert, und letztlich

3. der Definition von *Update*-Regeln, die anhand der Zustände der Nachbarn einer Zelle den Folgezustand bestimmt: Für diese Regeln wird dabei zusätzlich gefordert, dass sie *synchron* für alle Zellen ausgeführt werden und sich sowohl räumlich als auch zeitlich nicht verändern.

Die Zellen sind damit an den ganzzahligen Eckpunkten des zweidimensionalen euklidischen Gitters (*lattice*) $L \subset \mathbb{Z}^2$ lokalisiert und die Zustände $q_{i,j}$ sind aus demselben Zustandsraum Z, um die Bedingung zu erfüllen, dass alle Zellen prinzipiell gleichförmig sind. Die sogenannten *Moore*-Nachbarschaften[7] zum Radius r sind hierfür so definiert, dass gilt:

$$N_{i,j} := \{(k,l) \in L \mid |k-i| \leq r \wedge |l-j| \leq r\} \tag{3.14}$$

Ein sehr populäres Beispiel für einen einfachen Zellulären Automaten, der ein überraschend reichhaltiges Verhalten zeigt, ist Conway's *Game of Life* von 1970. Dabei können

[7] Es gibt weitere Definitionen von Nachbarschaft, beispielsweise die Von-Neumann Nachbarschaften - da die Moore'schen Nachbarschaften an die hier verwendete LBM vererbt wird, soll sich jedoch in dieser Arbeit auf diese beschränkt werden.

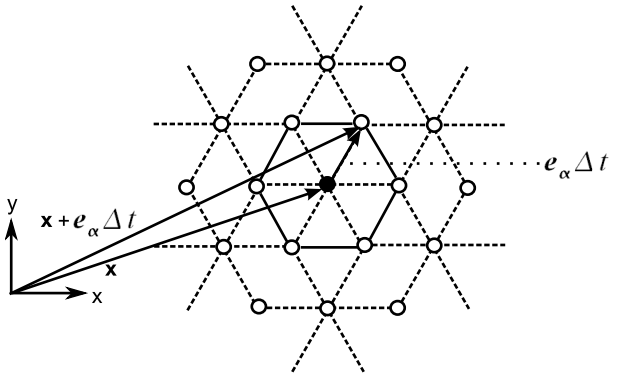
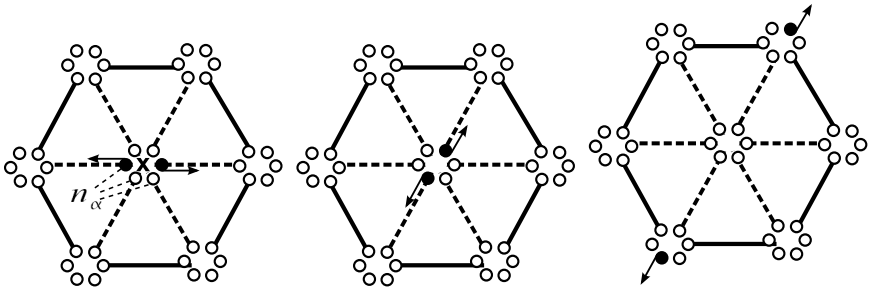


Abbildung 3.5: Hexagonales *Lattice* für einen 2D-LGCA und *Lattice-velocity* in Richtung $[i+1, j+1]$

Zellen lediglich die Zustände 0 („tot“) oder 1 („lebendig“) haben und die Update-Regeln lauten so, dass eine Zelle mit Zustand 0 im nächsten Zeitschritt den Zustand 1 erhält, wenn sie exakt drei Nachbarn mit Zustand 1 hat und eine Zelle mit Zustand 1 wird diesen Zustand beibehalten, wenn sie mindestens zwei und höchstens drei 1-Nachbarn hat, ansonsten wird ihr Nachfolgezustand 0. Aufgrund der Vielzahl komplexer Phänomene, die dieser und andere CA-Formulierungen in der Lage waren trotz ihrer Einfachheit in Bezug auf die Update-Regeln zu erzeugen (selbststabilisierende Anfangskonfigurationen, sich ständig wiederholende Konfigurationen und insbesondere sich wie Strömung fortpflanzende Populationen) mutmaßte man die Nutzbarkeit von CAs für die Simulation komplexer physikalischer Vorgänge. Durch die strikte Lokalität ihrer Arbeitsweise sind sie ideal für massiv parallele Rechnersysteme und gleichzeitig einfach zu implementieren. Um jedoch für Problemstellungen wie die der Lösung von Strömungsgleichungen geeignet zu sein, muss sie die Eigenschaft haben, dass sie sowohl *konservativ* ist, im hier vorliegenden Fall also masse- und impulserhaltend ist, als auch die entsprechenden makroskopischen Quantitäten im Raum *propagieren* kann. Die Tatsache, dass diese beiden Eigenschaften nur für sehr wenige CAs gezeigt werden können, führte zur Entwicklung der Lattice-Gas Methoden.

Lattice-Gas-Automaten, Mikrodynamik von Fluiden

Die zentrale Idee der LGCAs ist, den Update-Schritt der CAs in zwei sequentielle Schritte aufzuteilen, einen *Kollisionsschritt*, (*collision*) und einen *Transportschritt*, (*streaming*), um korrekte Fortpflanzung und Erhaltung von Quantitäten zu gewährleisten. Dazu ist es notwendig, den Begriff der *Lattice-Zelle* neu zu definieren. Die Nachbarschaftsbeziehung aus Gleichung (3.14) bleibt dabei erhalten und es werden Vektoren eingeführt, die nächste Nachbarn miteinander verbinden. Diese festen Trajektorien $\mathbf{e}_\alpha$ werden *Lattice-Vektoren* oder *lattice-velocities* genannt, vergleiche Abbildung 3.5 für ein Beispiel einer hexagonalen *Lattice-Zelle*. Bereits hier ist die enge Verknüpfung der Methode mit gasdynamischen



(a) Vor Kollision (b) Nach Kollision (c) Nach Transport

Abbildung 3.6: Kollisions- und Transportschritt bei LGCAs, boole'sche Besetzungsfunktionen n_α

(i.A. strömungsmechanischen) Vorgängen zu sehen. Durch die *Lattice-Vektoren* erhält jede *Lattice-Zelle* abhängig von ihrer Form (bzw. der Anzahl von direkten Nachbarn) eine Kapazität, die die maximale Anzahl von Zuständen der Zelle, anschaulich von gleichzeitig in ihr residierenden *Partikel* bestimmt. Die An- bzw. Abwesenheit eines Partikels mit einer bestimmten Bewegungsrichtung α wird mit boole'schen Variablen der Form $n_\alpha(\mathbf{x}, t)$, den sogenannten *Besetzungsfunktionen*, angezeigt, vergleiche Abbildung 3.6. Der Kollisionsschritt bestimmt dann eine Nachfolgekonfiguration am selben Ort, der im Transportschritt progagiert wird, indem sich die den entsprechenden *lattice-velocities* zugeordneten Partikel entlang dieser Trajektorien zu den Nachbarzellen bewegen, wie in Abbildung 3.5 dargestellt. Formal findet also Partikelbewegung vom Ort $\mathbf{x}$ zum Ort $\mathbf{x} + \mathbf{e}_\alpha \Delta t$ statt, wobei bei LGCAs stets $\Delta t = 1$ ist, d.h. dass die Partikel eine Einheitmasse m und eine Einheitsgeschwindigkeit haben, die lediglich eine Bewegung gerade bis zur nächsten Nachbarzelle in der entsprechenden Richtung erlaubt. Definiert man nun die erlaubten Nachfolgekonfigurationen wie in Abbildung 3.6 so, dass jeder Partikel eine eindeutige Nachfolgeposition an derselben Zelle haben muss und keine zwei Partikel dieselbe Nachfolgeposition haben dürfen, dann findet ein Impulsaustausch statt und für den Impuls gilt

$$\mathbf{p}(\mathbf{x}, t + \Delta t) = \sum_\alpha m \mathbf{e}_\alpha = \mathbf{p}(\mathbf{x}, t)$$

und damit Impulserhaltung. Um von den durch die n_α gegebenen Besetzungen der einzelnen Zellen auf makroskopische Quantitäten zu führen, muss entweder räumlich oder zeitlich in der Nachbarschaft einer Zelle gemittelt werden. Dieser Vorgang wird *coarse graining* genannt und bezeichnet somit den Übergang von den Besetzungen an $\mathbf{x}$ zu reellwertigen *Verteilungsfunktion* N_α. An dieser Stelle soll nicht genauer darauf eingegangen werden. Wichtig zu erwähnen ist jedoch, dass die Notwendigkeit der Mittelung der wesentliche Nachteil der LGCAs ist, da dadurch nicht nur statistisches Rauschen in das Verfahren induziert wird, sondern auch ein nicht zu unterschätzender Aufwand damit verbunden ist, da oft viele Nachbarzellen für eine hinreichend glatte Lösung in die Mittelung miteinbe-

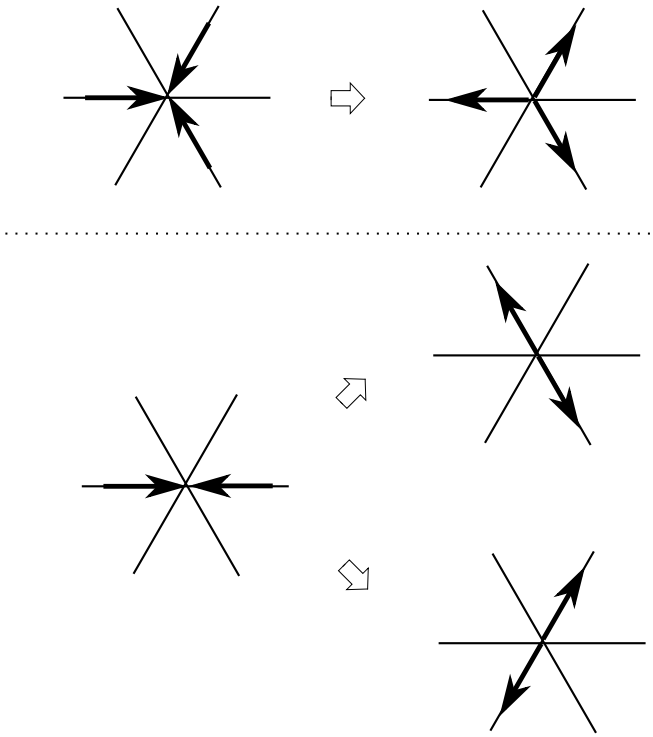


Abbildung 3.7: Nichteindeutigkeit der Kollisionen bei LGCAs

zogen werden müssen. Die makroskopischen Größen Masse $\overline{m}$ und Geschwindigkeit $\mathbf{u}$ sind mit $N_\alpha(\mathbf{x}, t)$ gegeben durch

$$\overline{m}(\mathbf{x}, t) = \sum_\alpha N_\alpha(\mathbf{x}, t) \tag{3.15}$$

und

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\overline{m}} \sum_\alpha \mathbf{e}_\alpha N_\alpha(\mathbf{x}, t) \ . \tag{3.16}$$

Obwohl die LGCAs die Schwächen der CAs nicht haben, haben auch sie neben der Notwendigkeit zur Mittelung noch einen wesentlichen Nachteil: Die boole'schen Verteilungsfunktionen erlauben keine eindeutige Definition des Kollisionsoperators. Abbildung 3.7 zeigt diese Ambiguität am Beispiel der oben erklärten hexagonalen *Lattice-Zelle*. Um für nichteindeutige Konfigurationen Nachfolgekonfigurationen zu ermitteln ist daher zusätzlich ein Zufallsschritt nötig. Trotz dieser Nachteile ist die LGCA-Methode der LBM bereits sehr ähnlich. Man kann die Partikelanzahl am Ort $\mathbf{x} + \mathbf{e}_\alpha \Delta t$ zur Zeit $t + \Delta t$ bei LGCAs bereits in der Form einer *Lattice-Boltzmann-Gleichung* schreiben, indem man die neue gemittelte Anzahl der Partikel gleich der alten zuzüglich eines Kollisionsoperators setzt. Sei dazu k die Anzahl der *lattice-velocities*, dann ist die *Bilanzgleichung* für einen Partikel der Richtung α gegeben durch:

$$N_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = N_\alpha(\mathbf{x}, t) + Q(N_\alpha, N_\beta), \qquad \beta = 1, ..., k \tag{3.17}$$

Diese Form ist der *Boltzmann-Gleichung* sehr ähnlich, was der LBM ihren Namen verleiht. Diese soll im folgenden kurz erklärt werden.

Die Boltzmann-Gleichung

Mit der Boltzmann-Gleichung wird die *molekulare Geschwindigkeitsverteilungsfunktion* $f(\mathbf{u}, \mathbf{x}, t)$ in Raum und Zeit bestimmt, d.h., dass man mit ihr die gesuchten Größen Masse und Geschwindigkeit gemittelt über eine Anzahl von Partikeln, berechnen kann, eine entsprechende numerische Methode vorausgesetzt. Zu ihrer Herleitung betrachtet man ein sehr kleines aber endliches Kontrollvolumen $\delta V = \delta x \delta y \delta z$ um einen Ort $\mathbf{x} = (x, y, z)^T$, wie in Abbildung 3.8 dargestellt, sowie ein Kontrollvolumen im Geschwindigkeitsraum $\delta \mathbf{U} = \delta u \delta v \delta w$. Diese Kontrollvolumina kann man sich als sehr kleine Kugeln oder (wie hier Quader) vorstellen, die den Ort $\mathbf{x}$ einschließen. Dann kann man die Anzahl der Moleküle δN im *Phasenvolumen* $\delta V \delta \mathbf{U}$ schreiben als[8]

$$\delta N_t + \delta N_r = \delta N_G - \delta N_V \ .$$

Die linke Seite der Gleichung beschreibt Änderungen in Folge von Molekültransport über das Kontrollvolumen und die rechte Seite ist die Bilanz der Moleküle in Folge von Kollisionen. Genauer stellen die Terme

- δN_t die zeitliche Änderung von δN Molekülen im Volumen δV, d.h.:

$$\delta N_t = (f(\mathbf{u}, \mathbf{x}, t + \delta t) - f(\mathbf{u}, \mathbf{x}, t)) \delta V \delta \mathbf{U} = \frac{\partial f}{\partial t} \delta V \delta \mathbf{U} \delta t, \tag{3.18}$$

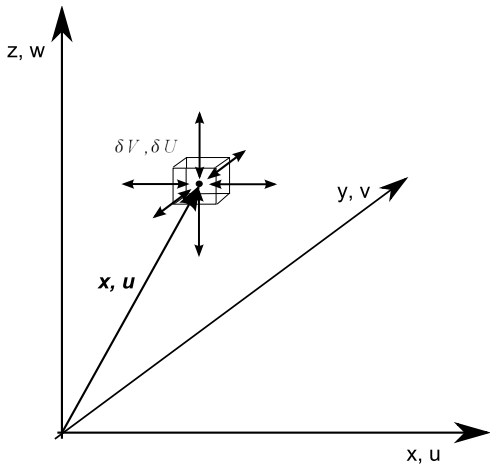
- δN_r die räumliche Änderung von δN Partikeln, die durch Unterschiede in den Flüssen über die Oberflächen von δV verursacht wird, vergleiche Abbildung 3.8(b) und als Flussbilanz für jede räumliche Dimension einzeln definiert ist und in der Summe δN_r ergibt,

$$\delta N_r = u_i \frac{\partial f}{\partial x_i} \delta V \delta \mathbf{U} \delta t$$

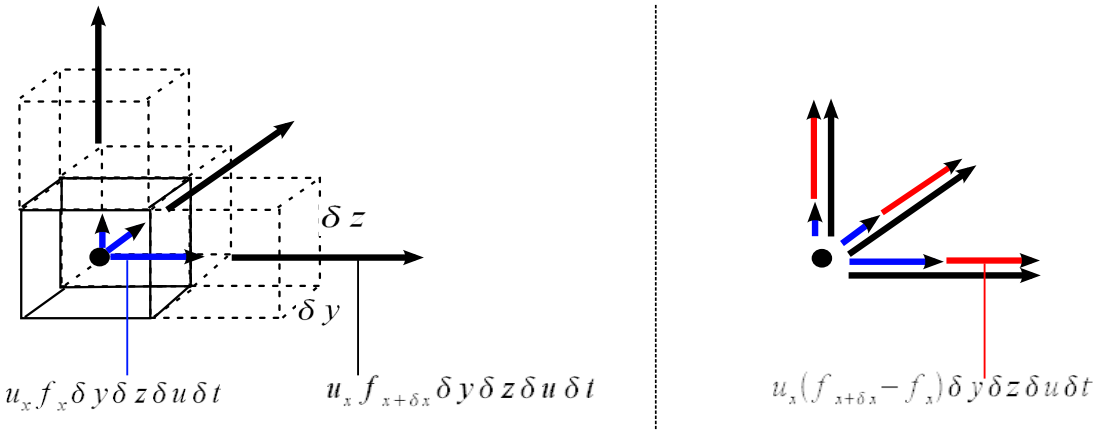
- δN_V den Verlustanteil, der durch Stöße mit Molekülen aus anderen Phasenvolumen dadurch entsteht, dass die kollidierten Moleküle das Kontrollvolumen verlassen, und

- δN_G den entsprechenden Gewinnanteil dar. Dieser Sachverhalt wird in Abbildung 3.8(c) veranschaulicht.

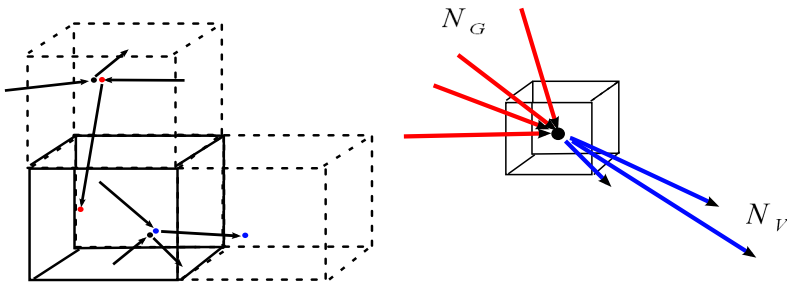
[8] Wie bisher werden hier noch externe Kräfte vernachlässigt, die zu einer Beschleunigung der Moleküle führen würden.



(a) Orts- und Phasenkontrollvolumen der Boltzmann Gleichung



(b) Räumliche Änderung der Partikelflüsse



(c) Kollisionsanteile

Abbildung 3.8: Zur Herleitung der Boltzmann Gleichung

Die Boltzmann-Gleichung ergibt sich aus der Summe dieser Anteile und wird in Tensorform (unter Verallgemeinerung des Kollisionsanteils) wie folgt geschrieben:

$$\frac{\partial f}{\partial t} + u_i \frac{\partial f}{\partial x_i} = Q \ .$$

Bei bekannter Verteilungsfunktion f können wie zuvor auch im Fall der Verteilungsfunktionen N_α bei den LGCAs alle makroskopischen Quantitäten hergeleitet werden. Die LBM geht allerdings nicht von Besetzungsfunktionen aus, sondern verwendet auch im Kollisions- und Transportschritt reellwertige Verteilungsfunktionen.

Von LGCAs zur LBM

Eine Ersetzung der N_α als räumlich oder zeitlich gemittelte Besetzungsfunktionen durch kontinuierliche Verteilungsfunktionen f_α in Gleichung (3.17) führt zur *Lattice-Boltzmann-Gleichung*

$$f_\alpha(\mathbf{x} + \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) + Q(f_\alpha, f_\beta), \qquad \beta = 1, ..., k \ . \tag{3.19}$$

Diese Substitution löst beide Probleme der LGCAs. Zum einen wird das *coarse graining* überflüssig, da die einzelnen Verteilungsfunktionen bereits die reellwertige Gesamtmasse von Partikeln mit einer bestimmten Bewegungsrichtung α bereitstellt. Zusätzlich gibt es keine Ambiguitäten bei der Kollision mehr: Die entsprechenden Anteile werden einfach auf die zur Verfügung stehenden Konfigurationen verteilt. Im Folgenden soll die konkrete Lattice-Boltzmann-Methode für die Lösung der Flachwassergleichungen entwickelt werden. Diese schließt eine Definition der gewählten Form der Lattice-Zelle und der zugehörigen *lattice-velocities* mit ein, genauso wie eine Beschreibung des Kollisionsoperators.

3.4 LBM für SWE

3.4.1 Diskreter Phasenraum, D2Q9-Modell

Unter Verwendung von Moore-Nachbarschaften zum Radius $r = 1$ (vergleiche Gleichung (3.14)) auf einem zweidimensionalen kartesischen Gitter mit *quadratischen Lattice-Zellen*, ist der diskrete 9-Geschwindigkeiten Geschwindigkeitsraum der Lattice-Boltzmann Methode gegeben durch das D2Q9-Modell, wie in Abbildung 3.9 dargestellt. Dabei ergeben sich die *lattice-velocities* mit $\Delta x = \Delta y$ und $e = \frac{\Delta x}{\Delta t}$ wie folgt:

$$\mathbf{e}_\alpha = \begin{cases} (0,0) & \alpha = 0 \\ e[\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}] & \alpha = 1,3,5,7 \\ \sqrt{2}e[\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}] & \alpha = 2,4,6,8 \end{cases}$$

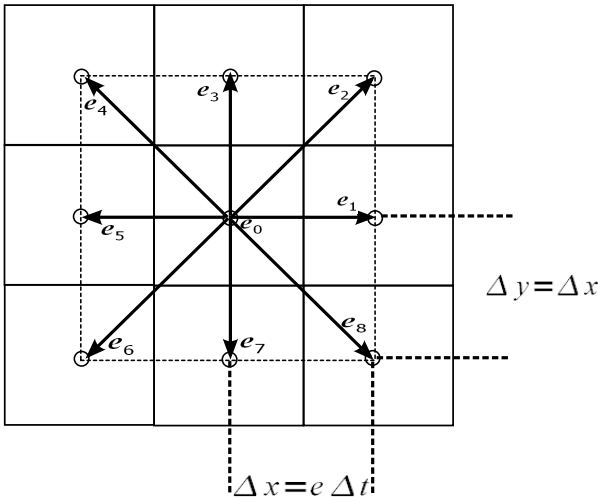


Abbildung 3.9: D2Q9-*Lattice*

Diese Form der *Lattice-Zelle* hat sich bei zweidimensionalen Problemen aufgrund ihrer Anschaulichkeit und Nähe zu klassischen Lösungsverfahren und insbesondere durch ihre Einfachheit in Bezug auf komplexe Randbedingungen bewährt, obwohl die hexagonale D2Q6 *Lattice-Zelle* als theoretisch gleichwertig (in Bezug auf die Symmetrie, s.u.) gelten muss. Mit der Festlegung von vorgeschriebenen Geschwindigkeiten erfolgt bei der LBM auch eine implizite Festlegung des Zeitschritts: Da wie bei LGCAs die Länge der *lattice-velocities* gerade so vorgegeben wird, dass die Partikel, die sich entlang dieser Trajektorien bewegen, gerade die nächste (in diesem Fall kartesische) Gitterzelle erreichen und es gilt:

$$\Delta x = e\Delta t$$

Des Weiteren sind die *lattice-velocities* zwar prinzipiell symmetrisch, es existieren jedoch drei unterschiedliche Beträge von Geschwindigkeiten: Eine Nullgeschwindigkeit ($\alpha = 0$), und je vier Geschwindigkeiten mit Betrag Δx (α *ungerade*) bzw mit Betrag $\sqrt{2}\Delta x$ (α *gerade*). In vielen mikroskopischen Überlegungen besonders in Kapitel 4 spielen daher richtungsabhängige Gewichtungen eine wichtige Rolle. Mit den bisherigen Herleitungen dieses Abschnitts ist man nun in der Lage, sich der Frage zuzuwenden, wie die in Abschnitt 3.2 beschriebenen Flachwassergleichungen durch die Lattice-Boltzmann-Methode gelöst werden können.

3.4.2 Kollisionsoperator und BGK Approximation

Der Term $Q(f_\alpha, f_\beta)$ aus Gleichung (3.19) ist eine Matrix, die durch die mikrodynamischen Vorgänge auf Partikelebene definiert ist und von dem Integral Q in der Lattice-Boltzmann-Gleichung stammt. Noch vor Beginn der 1990er Jahre erkannten Higuera und Jimenez [18], dass der Kollisionsoperator um einen *lokalen Gleichgewichtszustand* herum linearisiert wer-

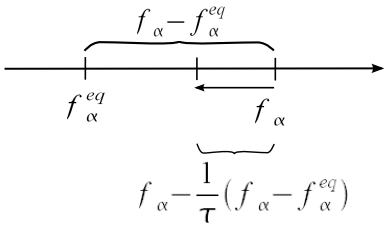


Abbildung 3.10: Gleichgewichtsanpassung mit einer Relaxationszeit τ im Lattice-BGK-Modell

den kann, was von Qian aufgegriffen wurde [31], der ein einfaches, linearisiertes Modell für die LBM entwickelte, das *Lattice-BGK*-Modell genannt wird, nach dem mikrodynamischen Kollisionsmodell von Bhatnagar, Gross und Krook aus den 1950er Jahren. Dazu wird der Kollisionsschritt der LBM im Einklang mit der Mikrodynamik von Molekülen so aufgefasst, dass dabei die an einem Ort ankommenden Partikel miteinander interagieren und ihre Geschwindigkeitsrichtungen ändern, wie bei den LGCAs bereits erklärt. Damit kann man den Kollisionsschritt schreiben als

$$f_\alpha^{\text{temp}}(\mathbf{x}, t) = f_\alpha(\mathbf{x}, t) + Q_\alpha(f(\mathbf{x}, t)) \ , \tag{3.20}$$

wobei f_α^{temp} die Verteilungsfunktion für Richtung α *vor* dem *streaming* ist. Um den Kollisionsoperator über einen Gleichgewichtszustand zu relaxieren, führt man nun *lokale Gleichgewichtsverteilungsfunktionen* f_α^{eq} ein. Diese sind Funktionen der (makroskopischen) Momente zum Zeutpunkt t. Man erinnere sich: f_α^{temp} ist nur ein Zwischenschritt auf dem Weg zu $f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t)$. Daher sind die f_α^{eq} Funktionen der makroskopischen Quantitäten (im Fall der SWE also h, u, und v) zum *letzten* Zeitpunkt. Mit dieser Sichtweise bedeutet der Kollisionsschritt, dass sich f_α seinem Gleichgewichtszustand entgegenbewegt, d.h. $f_\alpha \to f_\alpha^{eq}$. Abbildung 3.10 veranschaulicht die beim Kollisionsschritt der LBM beteiligten Größen und Vorgänge. Dabei wird deutlich, dass der Wert für f_α^{temp} zwischen dem Wert der Verteilungsfunktion und seinem Gleichgewichtswert liegen muss. Somit kann man den Kollisionsoperator als negative gewichtete Differenz dieser beiden Werte auffassen und schreibt:

$$f_\alpha^{\text{temp}}(\mathbf{x}, t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) \tag{3.21}$$

Die Größe τ kann dann als Geschwindigkeit aufgefasst werden, mit der sich f_α dem Gleichgewicht nähert. Diese wird *Relaxationszeit* genannt (*single relaxation time*). Mit dieser Approximation kann man Gleichung (3.21) in Gleichung (3.19) einsetzen und erhält die *Lattice-BGK-Gleichung*:

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) \ . \tag{3.22}$$

Um die Gleichgewichtsverteilungsfunktionen zu definieren, die zur Lösung der SWE führen, muss man nun noch die Form der *Lattice-Zelle* (und damit den diskreten Geschwindigkeitsraum) festlegen.

3.4.3 Lokale Gleichgewichtsverteilungsfunktionen

Die für den Kollisionsoperator hergeleiteten lokalen Gleichgewichtsverteilungsfunktionen, vergleiche Gleichung (3.21), bestimmen in der LBM, welche Strömungsgleichungen der Kollisionsoperator auf Partikelebene anwendet. In diesem Abschnitt werden die f_α^{eq} für die SWE hergeleitet. Einleitend sei bemerkt, dass die LBM normalerweise mit der sogenannten *Maxwell-Boltzmann* Gleichgewichtsfunktion assoziiert ist, mit der man das Verhalten von Strömung im Einklang mit den Navier-Stokes-Gleichungen simulieren kann. Der Vorteil der im Folgenden dargestellten Gleichgewichtsverteilungsfunktionen gegenüber letzteren wird in Kapitel 4 deutlich, wenn Kraftterme nötig sind.

Damit die hier vorgestellte LB-Methode die SWE lösen kann, muss die Gleichgewichtsverteilungsfunktion in Richtung α eine Reihe von Anforderungen erfüllen:

1. Durch die Theorie der LGCAs weiß man, dass die Rekonstruktion der makroskopischen Masse wie in Gleichung (3.15) durch Summation über die Richtungen erfolgt. Da die Masse in den SWE repräsentiert ist durch die ortsbezogene Tiefe des Fluids, muss gelten:
$$\sum_\alpha f_\alpha^{eq}(\mathbf{x},t) = h(\mathbf{x},t) \tag{3.23}$$

2. Analog gilt für die Geschwindigkeit nach Gleichung (3.16):
$$\sum_\alpha e_{\alpha i} f_\alpha^{eq}(\mathbf{x},t) = h(\mathbf{x},t) u_i(\mathbf{x},t) \tag{3.24}$$

3. Letzlich muss noch der Gravitationsterm der Impulsgleichung aus Gleichung (3.9) berücksichtigt werden:
$$\sum_\alpha e_{\alpha i} e_{\alpha j} f_\alpha^{eq}(\mathbf{x},t) = \frac{1}{2} g h^2(\mathbf{x},t)\delta_{ij} + h(\mathbf{x},t) u_i(\mathbf{x},t) u_j(\mathbf{x},t) \ , \tag{3.25}$$

wobei δ_{ij} die Kronecker'sche Deltafunktion ist mit

$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & \text{sonst} \end{cases} \tag{3.26}$$

Die Idee ist nun, jede Gleichgewichtsfunktion durch eine Potenzreihe darzustellen, vergleiche Zhou [42], so dass eine Reihe von Koeffizienten als unbekannte Konstanten mit den obigen drei Bedingungen ermittelt werden können:

$$f_\alpha^{eq} = C_1(\alpha) + C_2(\alpha) e_{\alpha i} u_i + C_3(\alpha) e_{\alpha i} e_{\alpha j} u_i u_j + C_4(\alpha) u_i u_i \tag{3.27}$$

Dabei nutzt man nun die Symmetrie der Phasenraumdiskretisierung (siehe vorherigen Abschnitt) aus, da die Gleichgewichtsfunktion dieselbe Symmetrie aufweisen muss. Dadurch weiß man, dass die Koeffizienten C_k jeweils für $\alpha = 0$ und α gerade bzw. ungerade gleich sein müssen und kann Gleichung (3.27) schreiben als

$$f_\alpha^{eq} = \begin{cases} C_1 + C_4 u_i u_i, & \alpha = 0, \\ \overline{C}_1 + \overline{C}_2 e_{\alpha i} u_i + \overline{C}_3 e_{\alpha i} e_{\alpha j} u_i u_j + \overline{C}_4 u_i u_i, & \alpha = 1, 3, 5, 7, \\ \tilde{C}_1 + \tilde{C}_2 e_{\alpha i} u_i + \tilde{C}_3 e_{\alpha i} e_{\alpha j} u_i u_j + \tilde{C}_4 u_i u_i, & \alpha = 2, 4, 6, 8, \end{cases} \tag{3.28}$$

wobei bereits berücksichtigt ist, dass $e_{0i} = 0$ ist. Sukzessives Einsetzen von Gleichung (3.28) in die Gleichungen (3.23), (3.24) und (3.25) ergibt, nach einigen Umformungen (vergleiche für Details das Buch von Zhou [42]), die lokalen Gleichgewichtsverteilungsfunktionen für das Lattice-BGK Modell mit D2Q9 *Lattice*:

$$f_\alpha^{eq} = \begin{cases} h - \frac{5gh^2}{6e^2} - \frac{2h}{3e^2} u_i u_i & \alpha = 0 \\ \frac{gh^2}{6e^2} + \frac{h}{3e^2} e_{\alpha i} u_i + \frac{h}{2e^4} e_{\alpha j} u_i u_j - \frac{h}{6e^2} u_i u_i & \alpha = 1, 3, 5, 7 \\ \frac{gh^2}{24e^2} + \frac{h}{12e^2} e_{\alpha i} u_i + \frac{h}{8e^4} e_{\alpha j} u_i u_j - \frac{h}{24e^2} u_i u_i & \alpha = 2, 4, 6, 8 \end{cases} \tag{3.29}$$

3.4.4 Transportschritt und Randbehandlung

Mit der Definition der f_α^{eq} ist der Kollisonsschritt der Lattice-BGK Methode vollständig beschrieben. Der Transport der Verteilungen f_α kann mit Hilfe von Gleichung (3.20) dann einfach als

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha^{\text{temp}}(\mathbf{x}, t) \tag{3.30}$$

geschrieben werden. Dies erfordert jedoch die Spezifikation von Randbedingungen, da die Orte $\mathbf{x} \in \Gamma$ (bzw. Γ_S) in bestimmten Richtungen keine Nachbarn haben, wie Abbildung 3.11 zeigt. Die einfachste solcher Randbedingungen für die LBM ist die sogenannte *no-slip-* oder *bounce-back-* Regel. Dabei wird der sequentielle Charakter von Kollision und Transport insofern ausgenutzt, als dass nach Kollision innerhalb der Skalarfelder f_α^{temp} mit auswärtiger Richtung α an $\mathbf{x} \in \Gamma \cup \Gamma_S$ die jeweiligen Werte einfach der entsprechenden Funktion der Gegenrichtung zugeteilt werden kann. Dabei wird stets angenommen, dass sich der Rand des Rechengebiets (also gewissermaßen die Wand des Festkörpers) auf halber Strecke zwischen den beiden Nachbarzellen, die von dem Transport betroffen sind, befindet. Sei β im Folgenden der Index einer Verteilungsfunktion und $\mathbf{x}_B \in \Gamma \cup \Gamma_S$, wobei der Vektor $\mathbf{x}_B + \mathbf{e}_\beta \Delta t$ den Rand überschreitet, dann ist die bounce-back Regel für das D2Q9 *lattice* definiert als

$$f_{-\beta}^{\text{temp}}(\mathbf{x}_B, t) = f_\beta^{\text{temp}}(\mathbf{x}_B, t)$$

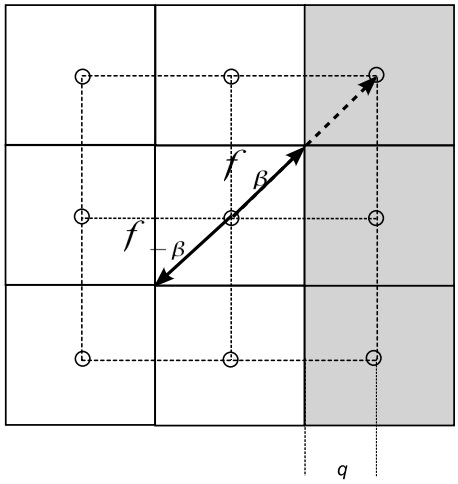


Abbildung 3.11: Bounce-Back Regel

mit

$$-\beta = \begin{cases} \beta + 4, & 1 \leq \beta \leq 4, \\ \beta - 4, & 5 \leq \beta \leq 8. \end{cases}$$

Damit wird das *streaming* also umgeleitet auf eine andere Verteilungsfunktion, wobei diese Regel genau dann konservativ ist, wenn die Forderung erfüllt ist, dass die soliden Wände der Begrenzung exakt eine Distanz von $q = \frac{1}{2}\Delta x$ zu den inneren *lattice*-Zellen haben. Nur dann ist die von den mikroskopischen Partikeln zurückgelegte Distanz (wegen Symmetrie) genau so groß wie die aller inneren Partikel und zwar $\mathbf{e}_\beta \Delta t$. Für $q \neq \frac{1}{2}\Delta x$ muss die bounce-back Regel angepasst werden, vergleiche Kapitel 4. Ein *streaming* in eine unzulässige Richtung findet nicht statt und die Dirichlet'schen Randbedingungen in Problem (3.13) werden automatisch berücksichtigt.

3.4.5 Makroskopische Quantitäten

Mit den Bedingungen für die Gleichgewichte in Abschnitt 3.4.3 kann man nun unmittelbar die makroskopischen Größen bestimmen. Nach Kollisionsschritt und Transport stehen die Verteilungsfunktionen zur Verfügung und die makroskopischen Quantitäten Fluidhöhe, und -geschwindigkeiten zum Zeitpunkt $t = k\Delta t$ am Ort $\mathbf{x}$ werden wie bei den LGCAs über Summation der richtungsbezogenen Verteilungsfunktionen ermittelt. Dabei ist

$$h(\mathbf{x}, t) = \sum_\alpha f_\alpha(\mathbf{x}, t) \tag{3.31}$$

die Tiefe des Fluids und

$$u_i(\mathbf{x}, t) = \frac{1}{h(\mathbf{x}, t)} \sum_\alpha e_{\alpha i} f_\alpha(\mathbf{x}, t) \tag{3.32}$$

die Geschwindigkeit in Richtung i.

3.4.6 Anfangs- und Stabilitätsbedingungen, Basisalgorithmus

Unabhängig von der Implementierung ergibt sich für die LBM eine mehr oder weniger festgelegte Reihenfolge der einzelnen Schritte. Zunächst müssen die Anfangsbedingungen definiert werden. Üblicherweise werden die Skalarfelder $h(\mathbf{x}, 0)$, $u(\mathbf{x}, 0)$ und $v(\mathbf{x}, 0)$ als glatte Skalarfelder vorgegeben und damit initiale Gleichgewichte f_α^{eq} berechnet, während für $f_\alpha(\mathbf{x}, 0) = 0, \forall \mathbf{x} \in \Omega_F$ gilt. Zusätzlich sind die Orts- und Zeitschrittweiten Δx und Δt, sowie die Relaxationszeit τ vorzugeben. Die Stabilitätseigenschaften der LBM werden aufgrund der Tatsache, dass es sich um ein recht junges Verfahren handelt, hauptsächlich numerisch erforscht. Es gibt allerdings einige von der Methode und den Flachwassergleichungen induzierte Stabilitätsbedingungen. Die erste dieser Bedingungen ist durch die Einführung der Relaxationszeit induziert. Die kinematische Viskosität des D2Q9 Modells kann ausgedrückt werden als

$$\nu = \frac{e^2 \Delta t}{6}(2\tau - 1) > 0$$

woraus unmittelbar folgt, dass

$$\tau > \frac{1}{2}$$

sein muss. Weiterhin dürfen die Geschwindigkeiten (makroskopisch) nicht größer sein, als die Ausdehnung der *lattice-Zelle* erlaubt. Mit anderen Worten heißt das, dass, genau wie sich die Fluidpartikel durch den diskreten Phasenraum nicht schneller bewegen können, als von einem Gitterpunkt zum nächsten, diese Einschränkung auch für die Geschwindigkeit des (initialen) Fluidkörpers gilt:

$$u_j u_j < e^2 \tag{3.33}$$

Durch die Flachwassergleichungen ist zusätzlich die vertikale Geschwindigkeit dieser Ungleichung unterworfen:

$$gh < e^2 \tag{3.34}$$

Zuletzt ist die Methode in ihrer bis hierher beschriebenen Form nicht in der Lage, Trockenzustände zu berücksichtigen, d.h. es gilt:

$$\frac{u_j u_j}{gh} < 1$$

und damit insbesondere $h > 0$. Ein wesentliches Ergebnis dieser Arbeit wird es sein, eine stabile Variante der LBM zu beschreiben, so dass $h = 0$ sein darf. Insbesondere Gleichung (3.33) schränkt die Geschwindigkeit von Festkörpern, die sich in der Flüssigkeit bewegen ein, vergleiche Kapitel 4.

Die Initialisierung des Algorithmus unter Berücksichtigung der obigen Bedingungen wird in dieser Arbeit `Preprocessing` genannt. Ein Zeitschritt des Lattice-BGK-Algorithmus kann dann so erfolgen, dass zunächst die Gleichgewichte f_α^{eq} (`EquilibriumDistribution`

nach Gleichung (3.29)) berechnet werden. Darauf folgen die Kollision (`Collision` mit Gleichung (3.21)) und der Transport (`Streaming` nach Gleichung (3.30)), wobei nach Kollision die Randbedingungen zu beachten sind (`NoSlipBC`). Zuletzt werden die makroskopischen Quantitäten nach Gleichungen (3.31) und (3.32) berechnet (`Extraction`), die dann für die Gleichgewichtsbestimmung des nächsten Zeitschritts zur Verfügung stehen. Sei im Folgenden stets n die Anzahl durchzuführender Zeitschritte. Damit ergibt sich der Algorithmus der Lattice-BGK-Methode für die zweidimensionalen Flachwassergleichungen wie folgt:

Algorithmus 3.1 Lattice-BGK-Methode für SWE, D2Q9

`Preprocessing`$\rightarrow h(\mathbf{x},0), \mathbf{u}(\mathbf{x},0)$
for $k = 1$ **to** n
  for all $\mathbf{x} \in \Omega_F$:
    for $\alpha = 0$ **to** 8:
      `EquilibriumDistribution`$\rightarrow f_\alpha^{eq}(\mathbf{x}, (k-1)\Delta t)$
      `Collision`$\rightarrow f_\alpha^{\text{temp}}(\mathbf{x}, k\Delta t)$
      `Streaming`$\rightarrow f_\alpha(\mathbf{x} + \mathbf{e}_\alpha k\Delta t, k\Delta t)$
      for all $\mathbf{x} \in \Gamma \cup \Gamma_S$:
        `NoSlipBC`
      `Extraction`$\rightarrow h(\mathbf{x}, k\Delta t), \mathbf{u}(\mathbf{x}, k\Delta t)$

In vielen Implementierungen werden Kollisions- und Transportschritt der Lattice-BGK-Methode zu einem Kernel zusammengefasst. Dadurch können Lade- und Speichervorgänge (*loads* und *stores*) eingespart werden. So betrachtet sind die Operationen `Collision` und `Streaming` lediglich eine einfache Linearkombination von f_α und f_α^{eq}. Diese Eigenschaft gilt auch für den `Extraction`-Schritt. Die eigentliche Rechenarbeit findet bei der Berechnung der Gleichgewichte f_α^{eq} statt, wo die neuen mikroskopischen Eigenschaften über die makroskopischen Größen des letzten Zeitschritts berechnet werden müssen. Dabei finden alle Rechnungen elementweise bezogen auf die beteiligten nicht-skalaren Operanden h, u, und v statt, d.h., dass dort keine örtlichen Nachbarschaften eine Rolle spielen, was für eine parallele Implementierung optimal ist. Zusätzlich ist die arithmetische Intensität hoch. Sei im folgenden N die Anzahl der *lattice-Zellen*. Im Normalfall, d.h. $\alpha \neq 0$, ergibt sich bei sinnvoller Implementierung (also insbesondere Ausklammern von h in Gleichung (3.29)), dass bei $3N + O(1)$ nötigen loads und N stores $27N + O(1)$ Gleitkommaoperationen nötig sind, womit sich das Verhältnis $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = 6.75$ ergibt, was die Lattice-BGK-Methode aus Sicht der in Kapitel 2 beschriebenen Entwicklung der Rechenhardware zu einer vielversprechenden Methode macht. Dies gilt jedoch zunächst nur für den Grundalgorithmus, mit dem lediglich relativ einfache Strömungen simuliert werden können. Im folgenden Kapitel werden die Erweiterungen im Blickfeld dieser Arbeit aufgebaut. Jede solche Erweiterung muss die wünschenswerten Eigenschaften der Methode erhalten. Für die Verwendung zur Bilderzeugung in graphischen Systemen (in der Computergraphik, CG) sind, im Gegen-

satz zu ingenieurswissenschaftlichen Anwendungen, für die die numerische Genauigkeit im Fokus liegt, besonders Einfachheit und Leistungsfähigkeit sowie Vielseitigkeit und Stabilität wichtig. Erstere ist bei der hier vorliegenden Lattice-BGK-Methode besonders dadurch gegeben, dass es sich um eine *matrixfreie* Methode handelt und somit nur Operationen mit relativ einfachen Linearkombinationen vonnöten sind und insbesondere keine (direkten oder iterativen) Löser für lineare Gleichungssysteme. Dadurch ist das Potenzial für eine möglichst große Ausnutzung der *Peak-Performance* von moderner Rechnerhardware asymptotisch und praktisch gegeben, was Ribbrock [32] in der Schwesterarbeit der hier vorliegenden Arbeit für obigen Algorithmus demonstriert und wie hier in Kapitel 6 im Hinblick auf Interaktions- und Echtzeitfähigkeit gezeigt wird. Dabei wird besonders wichtig sein, inwieweit man mit Rechenzeit (d.h. Leistungs-) Einbußen bei Erweiterung der Fähigkeiten des Lösers im Hinblick auf komplexere Simulationen rechnen muss. Die im Zusammenhang mit Computergraphik größte Schwäche der LBM, besonders im Hinblick auf den Einsatz im *interaktiven* Bereich, ist die (numerische) Stabilität. In virtuellen Umgebungen, bei der der Benutzer aktiv Einfluss auf ein Fluid nehmen kann, etwa durch Steuerung eines Wasserfahrzeugs oder Werfen von Gegenständen in die Flüssigkeit, bzw. die gesamte virtuelle Umwelt Einfluss darauf nimmt, etwa durch Regentropfen, Wind, etc., sind für die stabile Simulation des Fluids notwendige Parameter a priori nicht bekannt. Beispiele dafür sind maximale Wassertiefe und makroskopische Geschwindigkeiten (vergleiche etwa die Bedingungen in Gleichungen (3.33) und (3.34)). Diese Tatsache und die, dass die Stabilitätsbedingungen der Methode prinzipiell auch noch nicht vollständig bekannt sind, stellen ein ernstes Problem für den Einsatz etwa in Computerspielen dar. Dadurch müssen die nachfolgend entwickelten Erweiterungen in Kapitel 6 insbesondere im Hinblick auf die Stabilität getestet werden.

4 Fluid-Struktur-Interaktion

4.1 Einführung

Die Einbindung einer Flüssigkeit in eine virtuelle Umgebung bedeutet, dass einerseits die Strömung auf die Festkörper und gegebenenfalls auch auf die Atmosphäre bzw. eventuell vorhandene Gase und andere Flüssigkeiten einwirkt und andererseits die Umwelt durch Stöße oder allgemein Kräfte die Flüssigkeit beeinflusst. Im Folgenden soll die für eine realistische und interaktive (dreidimensionale) Szene wohl wichtigste dieser Wechselwirkungen, die Fluid-Struktur-Interaktion (*Fluid-Structure-Interaction, FSI*) in die im vorangegangenen Kapitel beschriebene Lattice-Boltzmann-Methode integriert werden. Dadurch werden komplexe Strömungssimulationen möglich, wie die bereits in Abbildung 3.1 gezeigten Strömungen über unebenen oder sogar zuvor trockenen Untergrund aber insbesondere auch in die Flüssigkeit eintauchende und eingetauchte und darin treibende bzw. sich mit eigenem Antrieb bewegende Festkörper. Dies erfolgt mit dem Ziel, möglichst realistisch wirkende Strömungen zu berechnen, wobei Einfachheit (in Bezug auf die Implementierbarkeit und Portierbarkeit auf spezielle Rechenhardware), Stabilität und Effizienz Vorrang haben vor quantitativer physikalischer Korrektheit bzw. numerischer Genauigkeit. Mit der in Kapitel 3 erfolgten Einschränkung der numerischen Methode auf zwei räumliche Dimensionen ist eine physikalisch genaue Repräsentation der dreidimensionalen Fluid-Struktur-Wechselwirkung nicht möglich. Das Problem wird vielmehr auf die Wechselwirkung zwischen *Flüssigkeitsoberfläche* und dreidimensionaler Szene abgebildet.

Die Interaktion von zweidimensionalen Flachwasserströmungen mit einer Szene erfolgt grundsätzlich zunächst über die Form der Bodentopographie, über die das Fluid fließt. Das Gefälle in diesem Untergrund und materialabhängige Reibung induzieren Kräfte, die beschleunigend (bzw. verlangsamend) auf die Flüssigkeitsoberfläche einwirken. Neben dieser in den inhomogenen Flachwassergleichungen (s.u.) selbst bereits berücksichtigten Kräfte, wirken sich in die Flüssigkeit vollständig oder teilweise eingetauchte Objekte durch ihre Bewegung durch lokale Verdrängung der Flüssigkeit auf die Strömung aus. Zuletzt können stationäre Objekte, die die Flüssigkeit horizontal begrenzen, wiederum (hauptsächlich durch Reibung aber auch durch eine Bewegung ihrer Außenwände) eine Kraft auf das Fluid induzieren. Umgekehrt kann die Strömung den Untergrund erodieren und eingetauchte Objekte beschleunigen und deformieren. Durch die Flachwassergleichungen als beherrschende Strömungsgleichungen der hier verwendeten LBM und durch die zweidimensionale Metho-

de selbst ergeben sich einige Einschränkungen dafür, inwieweit diese Effekte berücksichtigt werden können. Diese werden in den jeweiligen Abschnitten beschrieben.

4.2 Externe Kräfte durch die Bodentopographie

4.2.1 Problemstellung

Problem (3.12) bezieht lediglich die *homogenen* SWE ein: Ein *Quell-*(oder *Kraft-*)term ist nicht vorhanden. Damit sind die Szenarien für diesen Ansatz auf solche beschränkt, die ein homogenes Bodenprofil parallel zur (x, y)-Ebene betrachten, wobei die Bodentopographie in der Gleichung deshalb entfallen kann, da ohne lokale Unterschiede in der Höhe des Bodens keine Kräfte durch Gefälle oder Reibung entstehen können. Im Folgenden soll diese erste Formulierung um einen solchen Quellterm erweitert werden. Damit erhält man ein Modell, in dem man bereits eine Reaktion des Fluids auf Festkörper in direktem Kontakt hat: Im Hinblick auf die Ziele dieser Arbeit werden hier die betrachteten externen Kräfte auf solche beschränkt, die durch ein beliebig geformtes Bodenprofil auf das Fluid (in diesem Fall auf dessen Oberfläche) einwirken. Andere Kräfte, wie die in 3 beschriebene Coriolis-Kraft oder Scherungskräfte (etwa durch Wind oder lokale Strömungen) werden dabei weiterhin nicht berücksichtigt.

Im Zusammenhang mit der Bodentopographie, über die sich eine Flüssigkeit bewegt, sind zwei Kräfte maßgeblich, induziert durch:

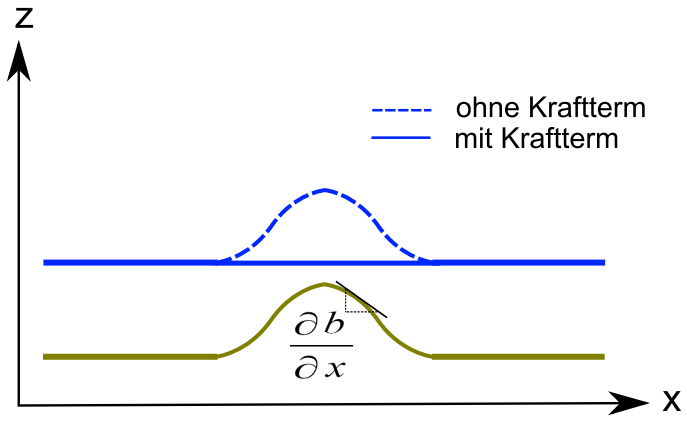
- *Lokale Unterschiede* in der Höhe des Bodens (*bed*) $b(\mathbf{x})$, also *Gefälle*: *bed-slope*,
- Materialabhängige *Reibung*: *bed-friction*.

Erstere ist für das visuelle Ergebnis kritisch, da die unbekannte Fluidoberfläche bei den SWE gegeben ist durch die Höhe des Fluids *über dem Untergrund*: $h+b$. Daher würde, ohne Einsatz eines Quellterms, die Flüssigkeitsoberfläche beispielsweise bei einer Dammbruchsimulation im sogenannten *Steady-State*, d.h. in einem Zustand, bei dem die Unterschiede in den makroskopischen Quantitäten sich zeitlich nicht mehr verändern, parallel zum Boden verlaufen, wie in Abbildung 4.1(a) gezeigt. Die bed-friction Kraft berücksichtigt das Material, aus dem der Untergrund besteht. Über diese Kraft kann man grundsätzlich die Fließgeschwindigkeit durch Veränderung eines Materialparameters steuern und damit die visuelle Erscheinung der Strömung. Beide Quellterme werden im Folgenden hergeleitet.

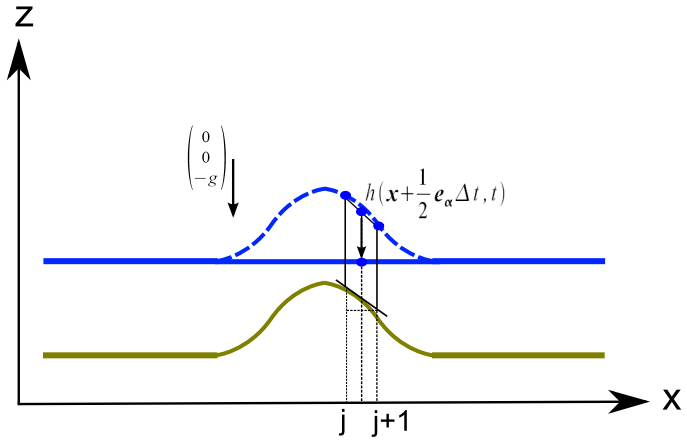
4.2.2 Erweiterung der SWE: Quellterme

Gefälle

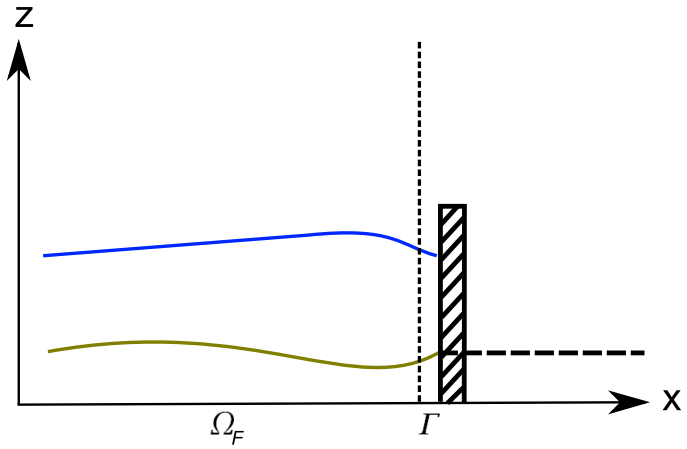
Die Steigung des Bodenprofils kann durch die Summe der partiellen Ableitungen $\frac{\partial b}{\partial x_i}$ dargestellt werden. Es wirkt die Gravitation g in negativer z-Richtung, weshalb proportional zum



(a) *Steady-State* mit und ohne Gefälle-Kraftterm



(b) zur Wirkungsweise des Gefälle-Kraftterms



(c) Randbehandlung

Abbildung 4.1: Quell- /Kraftterme

Gefälle die lokale Fluidtiefe h um einen mit $-g\frac{\partial b}{\partial x_i}$ gewichteten Anteil verringert werden muss, vergleiche Abbildung 4.1(b). Damit ergibt sich für den Quellterm zunächst:

$$S_i^{\text{slope}} = -gh\frac{\partial b}{\partial x_i} \tag{4.1}$$

Anders ausgedrückt bedeutet dieser Term, dass eine *negative* Steigung der Bodenoberfläche zu einer Zunahme der Tiefe führen muss, während umgekehrt eine *positive* Steigung eine Verringerung der Tiefe impliziert.

Reibung

Weniger anschaulich ist der Reibungskraftterm (auch *Sohlschubspannung*, vergleiche Schwanenberg [33]), da er auf empirischen Gesetzen beruht: entweder auf der Manning-Gleichung oder der Colebrook-White-Gleichung [42]. Die am häufigsten verwendete Formulierung dieses Quellterms erfolgt über das Manning'sche Reibungsgesetz, nach dem

$$S_i^{\text{friction}} = -gn_b^2 h^{-\frac{1}{3}} u_i \sqrt{u_j u_j} \tag{4.2}$$

geschrieben werden kann. Auch hier ist die Kraft proportional zur (negativen) Gravitation und zu n_b, dem Manning-Koeffizienten, der gewissermaßen die Rauheit des Materials ausdrückt. Der Parameter bestimmt damit also insbesondere den Betrag der Kraft. Ein rauherer Untergrund impliziert damit einen größeren Wert für n_b: Natürliche Gewässer, beispielsweise mit einem Kiesbett, werden daher mit einem größeren Manning-Koeffizienten assoziiert, als beispielsweise ein künstlicher Kanal aus Beton. Die Werte für n_b varriieren grundsätzlich zwischen etwa 0.01 (etwa für glatten Asphalt) und 0.08 (sehr rauhes natürliches Flussbett), vergleiche die Tabelle auf den Seiten des Forest Science Labs Research Network [25].

Gesamtkraft, inhomogene SWE

Setzt man nun

$$S_i^b = S_i^{\text{slope}} + S_i^{\text{friction}}, \tag{4.3}$$

so kann man die inhomogenen Flachwassergleichungen schreiben als

$$\frac{\partial h}{\partial t} + \frac{\partial (hu_j)}{\partial x_j} = 0 \tag{4.4}$$

$$\frac{\partial hu_i}{\partial t} + \frac{\partial (hu_i u_j)}{\partial x_j} + g\frac{\partial}{\partial x_i}(\frac{h^2}{2}) = S_i^b, \tag{4.5}$$

bzw. in Erhaltungsform:

$$\frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = S^b(Q). \tag{4.6}$$

Dabei ist $Q = (h, hu_x, hu_y)^T$ und $F(Q)$ und $G(Q)$ sind wie in Gleichungen (3.11) definiert. Zusätzlich ist der Quellterm in konservativer Form gegeben durch

$$S^b(Q) = \begin{pmatrix} 0 \\ S_x^b \\ S_y^b \end{pmatrix}. \tag{4.7}$$

Durch den Term S_i^{slope}, der auf den räumlichen Ableitungen des Bodenprofils basiert, sind wiederum Randbehandlungen nötig. Wird die Richtungsableitung beispielsweise mit finiten Vorwärtsdifferenzen durchgeführt, ist (im Fall eines einfachen rechteckigen Rechengebiets) der Wert für $b(\mathbf{x} + \mathbf{e}_1\Delta t)$ nicht verfügbar, wie in Abbildung 4.1(c) gezeigt. Da die bisherigen noslip-Randbedingungen implizieren, dass das Rechengebiet vollständig von *senkrechten* Wänden eingeschlossen ist, ist eine Approximation sinnvoll, bei der angenommen wird, dass zwischen Oberfläche des Bodenprofils und begrenzender Wand stets ein rechter Winkel liegt. Damit ist der Wert der Ableitung an den kritischen Stellen einfach Null. Mit dieser Vereinfachung geht das Problem der Simulation von Strömung mit den inhomogenen SWE für diese Arbeit aus Problem (3.13) hervor, indem der Quellterm für die Bodentopographie, $S^b(Q)$, berücksichtigt wird. Zusätzliche Randbedingungen für h, u und v sind nicht erforderlich und damit ist das Gesamtproblem definiert durch:

$$\begin{cases} \frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = S^b(Q), & \mathbf{x} \in \Omega_F, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t) = 0, & \mathbf{x} \in \Gamma, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}, t = 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t) = 0, & \mathbf{x} \in \Gamma_S, t > 0 \\ h(\mathbf{x}, t) = h_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}, t = 0 \end{cases} \tag{4.8}$$

4.2.3 Erweiterung der LBM: Kraftterme

Um die durch obige Kraftterme induzierten Kräfte in der in Kapitel 3 dargestellten Lattice-BGK-Methode zu repräsentieren, müssen sie additiv nach dem Kollisionsschritt hinzugefügt werden. Dies ist möglich, weil in der Boltzmann'schen Gasdynamik äußere Kräfte die Kollision von Partikeln nicht stören, wohl aber deren Flugbahn und Geschwindigkeit nach der Kollision. Es ist dabei zu berücksichtigen, dass die Quellterme S_i^{friction} und S_i^{slope} für die makroskopischen Größen definiert sind und daher in der LBM eine Aufteilung auf die jeweiligen Richtungen (*lattice-velocities*) erfolgen muss. Diese erfolgt in der Lattice-Boltzmann-Gleichung durch Multiplikation der Kraftterme mit der *Lattice-Velocity* in Richtung α und einer Konstanten, die gegeben ist durch die Form des *Lattice* sowie die Zeit- und Ortsschrittgrößen. Die LBE mit einem verallgemeinerten Kraftterm kann man dann schreiben als

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha\Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{N_l e^2} e_{\alpha i} F_i \ , \tag{4.9}$$

wobei wie in Kapitel 3 $e = \frac{\Delta x}{\Delta t}$ ist und N_l eine Konstante, die durch das *Lattice* bestimmt wird:

$$N_l = \frac{1}{e^2} \sum_{\alpha} e_{\alpha i} e_{\alpha i}. \tag{4.10}$$

Für das D2Q9-*Lattice* ergibt sich damit ein Wert von 6 für N_l. Mit den beiden Quelltermen (4.1) und (4.2) ist die Lattice-Boltzmann Gleichung für die inhomogenen SWE mit Krafttermen für die Sohlschubspannung und das Gefälle gegeben als

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2} e_{\alpha i} F_i^b \ , \tag{4.11}$$

bzw.

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2} (e_{\alpha x} F_x^b + e_{\alpha y} F_y^b) \ , \tag{4.12}$$

wobei in beiden Gleichungen $F_i^b = S_i^b$ gilt und damit

$$f_\alpha(\mathbf{x}+\mathbf{e}_\alpha \Delta t, t+\Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2} e_{\alpha i} (-g(h \frac{\partial b}{\partial x_i} + n_b^2 h^{-\frac{1}{3}} u_i \sqrt{u_j u_j})) \ . \tag{4.13}$$

Typischerweise werden Krafterme in modularen Implementierungen additiv berechnet. Um Algorithmus 3.1 so zu erweitern, dass das Bodenprofil mitberücksichtigt wird, werden die Krafttermberechungen im Anschluss an die Kollision durchgeführt. Da in dieser Arbeit stets beide Krafterme zusammen (oder gar nicht) auftreten, werden sie zu F_i^b zusammengefasst:

Algorithmus 4.1 Lattice-BGK-Methode für inhomogene SWE

`Preprocessing`$\rightarrow h(\mathbf{x}, 0), \mathbf{u}(\mathbf{x}, 0)$
for $k = 1$ **to** n
  for all $\mathbf{x} \in \Omega_F$:
    for $\alpha = 0$ **to** 8:
      `EquilibriumDistribution`$\rightarrow f_\alpha^{eq}(\mathbf{x}, (k-1)\Delta t)$
      `Collision`$\rightarrow f_\alpha^{\text{temp}}(\mathbf{x}, k\Delta t)$
      `Force`$\rightarrow f_\alpha^{\text{temp}}(\mathbf{x}, k\Delta t) + \frac{\Delta t}{6e^2} e_{\alpha i} F_i^b$
      `Streaming`$\rightarrow f_\alpha(\mathbf{x} + \mathbf{e}_\alpha k\Delta t, k\Delta t)$
      for all $\mathbf{x} \in \Gamma \cup \Gamma_S$:
        `NoSlipBC`
      `Extraction`$\rightarrow h(\mathbf{x}, k\Delta t), \mathbf{u}(\mathbf{x}, k\Delta t)$

Diese Formulierung ist unabhängig von der konkreten Implementierung der Differentiation $\frac{\partial b}{\partial x_i}$. Das Modul `Force` muss dafür sorgen, dass bei der Interpolation und der Berechnung der diskreten partiellen Ableitungen auf das Basisschema $F_i(\mathbf{x}, t)$ zurückgegriffen wird. Die konkrete numerische Behandlung der Krafterme ist auch im Hinblick auf ein visuell ansprechendes Ergebnis entscheidend. Dieser Arbeit vorausgehende Experimente mit einer

einfachen Evaluation des Kraftterms ergaben bereits, dass die gewünschte Auslöschung der zusätzlichen Anhebung der Fluidoberfläche im *Steady-State*, wie in Abbildung 4.1(b) mit einem Schema, das den Kraftterm an jedem Ort direkt auswertet, also

$$F_i = F_i(\mathbf{x}, t) \tag{4.14}$$

nicht möglich war. In dieser Arbeit wird daher das semi-implizite *centred* Schema von Zhou [42] verwendet:

$$F_i = F_i(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t), \tag{4.15}$$

welches den Kraftterm am Mittelpunkt zwischen zwei Nachbarzellen bezüglich α auswertet. Dabei werden h, u, v und b also in Richtung α interpoliert. Wie in Kapitel 6 gezeigt wird, ist dafür eine 2-Punkte Vorwärtsapproximation ausreichend. Am Rand des Rechengebiets fällt das Schema dann einfach auf Gleichung (4.14) zurück. Das heißt insbesondere, dass wir die Differentiation auch dann mit finiten Vorwärtsdifferenzen durchführen, wenn eigentlich interpoliert werden müsste. Am Beispiel des Kraftterms für das Gefälle und in x-Richtung erhält man dann für die *Lattice-Velocity* $\alpha = 1$ (der Übersichtlichkeit halber seien hier die üblichen räumlichen Indizes verwendet anstelle der Notation $h(\mathbf{x}, t)$)

$$F_x^{\text{slope}}(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t) = -g \frac{(h_{ij} + h_{i,j+1})}{2} \frac{(b_{i,j+2} - b_{i,j})}{2\Delta x}. \tag{4.16}$$

Der Term F_y^{slope} entfällt wegen $e_{1y} = 0$. Dabei vereinfachen wir den rechten Faktor so, dass nur noch direkte Nachbarn zur Berechnung nötig sind, verwenden Interpolation also nur noch bei h und damit ergibt sich:

$$F_x^{\text{slope}}(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t) = -g \frac{(h_{ij} + h_{i,j+1})}{2} \frac{(b_{i,j+1} - b_{i,j})}{\Delta x}. \tag{4.17}$$

Für den Reibungsterm ist keine Interpolation nötig.

4.3 *Dry-States*

4.3.1 Problemstellung

Die Lattice-Boltzmann-Methode in der hier in Kapitel 3 dargestellten Form ist auf *subkritische* Flachwasserströmungen beschränkt. Dies bedeutet insbesondere, dass eine Fluidtiefe von 0 bzw. bereits sehr kleine, von Null verschiedene Werte für die Tiefe den Algorithmus destabilisieren. Maßgeblich dafür verantwortlich ist der Extraktionsschritt für die makroskopische Geschwindigkeit, (3.32), in dem durch die Tiefe geteilt wird. In den bisherigen Erweiterungen dieses Kapitels ist außerdem der Reibungskraftterm (4.2) problematisch, der ebenfalls eine Division durch h enthält. Die Fähigkeit des Verfahrens zum dynamischen Trocknen bzw. Nasswerden eines Ortes im Rechengebiet, d.h. Unterstützung von *dry-states* ist allerdings nicht nur optisch eine wünschenswerte Eigenschaft einer Strömungssimulation. Dafür muss ein numerisches Schema im Wesentlichen leisten, dass

- Division durch Null verhindert wird,
- durch Division durch sehr kleine Werte für h die makroskopischen Geschwindigkeiten an der Grenze von *Trocken* zu *Nass*, nicht zu hoch werden können und
- die Kräfte dort ebenfalls nicht zu hoch werden können.

Im Folgenden wird ein einfaches Verfahren vorgestellt, dass *dry-states* stabil und mit guter visueller Approximation in der LBM behandeln kann.

4.3.2 Implizite, orts- und zeitabhängige Trockenzustände, Limiter-Ansatz

Zunächst ist eine Unterscheidung von trockenen und nicht-trockenen Zellen erforderlich, die die Grenzen des Gleitkommasystems der zugrundeliegenden Hardware berücksichtigt. Ein einfacher Ansatz ist, bei der Berechnung der makroskopischen Geschwindigkeiten die Wasserhöhen in der Nähe der Größenordnung des Abschneidefehlers als trocken zu definieren und dort die Geschwindigkeiten auf Null zu setzen. Dies führt jedoch dazu, dass die Gleichgewichte für den nächsten Schritt die Fließrichtung der Strömung nicht richtig repräsentieren. Um dies zu vermeiden kann man eine Funktion verwenden, die die (zu großen) Geschwindigkeiten auf ein charakteristisches Intervall begrenzt. Tatsächlich werden so in vielen numerischen Verfahren Singularitäten, d.h. lokale Ausreißer im Geschwindigkeitsvektorfeld, verhindert. In dieser Arbeit wird daher eine adaptive Variante des *Min-Mod-Limiters*

$$\phi(x) = \max(0, \min(1, x)) \tag{4.18}$$

verwendet, die berücksichtigt, dass Geschwindigkeitskomponenten auch negativ sein können. Dazu sei U eine vorgegebene Grenze für ein symmetrisches Intervall, auf das die Geschwindigkeiten abgebildet werden. Die Limiterfunktion $\phi_U(x)$ sei dann definiert als

$$\phi_U(x) = \max(-U, \min(U, x)) \tag{4.19}$$

mit deren Hilfe man die Extraktion der makroskopischen Größen u und v schreiben kann als

$$u_i(\mathbf{x}, t) = \begin{cases} \frac{1}{h(\mathbf{x},t)} \sum_\alpha e_{\alpha i} f_\alpha(\mathbf{x}, t), & h(\mathbf{x}, t) > \epsilon \wedge h(\mathbf{x}, t) < -\epsilon, \\ \phi_U(\frac{1}{h(\mathbf{x},t)}) \sum_\alpha e_{\alpha i} f_\alpha(\mathbf{x}, t), & \text{sonst.} \end{cases} \tag{4.20}$$

Das ϵ in der Bedingung für den Einsatz des Limiters ist dabei ein Wert nahe Null, so dass die Verteilungsfunktionen in jedem Zeitschritt zwar stets wachsen, für den enstsprechenden Ort aber erst gilt, dass er nicht mehr trocken ist, wenn bei der Extraktion die Fluidtiefe zumindest ϵ ist. Dieses Verfahren hat sich als recht robust erwiesen, wie in Kapitel 6 gezeigt wird. Dieselbe Limiterfunktion kann man einsetzen, um die Stabilität des Algorithmus generell zu erhöhen, indem man sie in der Extraktionsphase stets auf h mit einem bezogen auf die durchschnittlichen Tiefen hohen Wert für U anwendet. Dadurch kann verhindert

werden, dass sich Singularitäten ausbreiten, d.h., dass es zwar unerwünschte Ausreißer im Höhenfeld gibt, diese jedoch im nächsten Zeitschritt wieder geglättet werden. Besonders in der Entwurfsphase (von Szenarios für den Löser, neuen Algorithmen oder Szenen) ist dies nützlich, da so das Divergieren der Lösungen oder gar Programmabstürze verhindert werden können. Dieselbe Technik kann prinzipiell auch auf die Kraftterme angewendet werden. In allen Modulen ist selbstverständlich eine Division durch Null stets durch eine mit ϵ zu ersetzen.

4.4 Generelle Konzeption der Strömungsinteraktion mit eingetauchten 3D-Festkörpern

4.4.1 Problemstellung

Durch die in den beiden vorherigen Abschnitten eingeführten Kraftterme und Erweiterungen bezüglich *dry-states* besitzt die in dieser Arbeit vorgestellte Methode zur Strömungssimulation bereits die Eigenschaft, mit einer dreidimensionalen Szene, gegeben durch die Oberfläche des Bodens, gekoppelt werden zu können[1]. Durch die in Kapitel 3 eingeführte *No-Slip*-Randbehandlung, werden außerdem bereits stationäre Festkörper, deren Wände bis auf die Reflektionsbedingung für die Verteilungsfunktionen der LBM keine Kräfte (Stöße) auf das Fluid ausüben, berücksichtigt.

Die Reaktion des Fluids auf die Bodentopographie und die Möglichkeit, dass die Flüssigkeit zeitabhängig ganz unterschiedliche Regionen der Szene überfluten kann, ist bereits ein wünschenswertes Moment in einer dreidimensionalen virtuellen Umgebung. Die zweite wesentliche Komponente für eine Einbindung in ein digitales Bilderzeugungssystem ist die Möglichkeit, die Flüssigkeit in eine bestehende Festkörperphysik-Simulation einzubinden. Bewegung, Kollision und Deformation von Festkörpern sind nicht nur wegen des hohen Potentials zusätzlicher Handlungsfreiheit für den Benutzer und Dynamik einer dreidimensionalen Echtzeitumgebung sondern insbesondere durch die Steigerung der Leistung von Rechnerhardware und die dadurch entstehenden Möglichkeiten in den letzten Jahren im Bereich der Unterhaltungsindustrie von großem Interesse. Echtzeitberechnung von Umgebungs- und Physikeffekten sind durch hochleistungsfähige Physikbeschleunigungshardware, insbesondere GPUs, im Begriff, vorberechnete Sequenzen ohne Interaktionsmöglichkeit abzulösen. Nicht zuletzt hat die gesteigerte Verfügbarkeit der Rechenleistung von Grafikkarten durch Entwicklung von Spracherweiterungen wie NVIDIA's CUDA [27], ATI's STREAM, [1] bzw. APIs für die Bereitstellung von Physikeffekten, wie NVIDIA's PhysX [26] diese Entwicklung herbeigeführt. Durch die hohe (mathematische) Komplexität und den hohen Rechenaufwand von numerischen Methoden zur Strömungssimulation, sind

[1] Tatsächlich werden die inhomogenen zweidimensionalen Flachwassergleichungen daher oft auch als $2\frac{1}{2}$-dimensional bezeichnet.

Echtzeitströmungssimulationen auch in aktueller 3D-Software (bzw. Physik-APIs[2]) praktisch nicht vorhanden. Die Oberfläche von Wasser und Flüssigkeit im Allgemeinen wird hauptsächlich durch wenig dynamische Geometrie angenähert und mit Textureffekten versehen[3], so dass sie wie solche aussehen, ohne dass die Flüssigkeit einen Einfluss auf das Geschehen hat bzw. umgekehrt man darauf Einfluss nehmen kann.

Das Problem der Fluid-Struktur-Interaktion besteht einerseits darin, die durch den Festkörper induzierten lokalen Kräfte zu bestimmen, die auf das Fluid an der Grenze (*interface*) zwischen Festkörper und Flüssigkeit einwirken und andererseits darin, die Kräfte zu berechnen, die am *interface* zwischen selbigen durch die Strömung auf den Festkörper einwirken. Die Vereinfachung, Strömung, bzw. die Fluidoberfläche der Strömung mit einem zweidimensionalen Verfahren zu berechnen, bietet den Vorteil, wesentlich weniger Rechen- und Speicherresourcen zu beanspruchen. Gleichzeitig wird ein Festkörper wie in Kapitel 3 erklärt, dann als Teilmenge des $\mathbb{R}^2$ aufgefasst, das zugehörige *interface* ist dann dessen Umrandung Γ_S. Somit kann man mit einem solchen Löser auch nur die Kräfte berechnen, die an der Oberfläche des Fluids auf den Festkörper einwirken. Im Fall der Flachwassergleichungen sind die makroskopischen Geschwindigkeiten jedoch *tiefengemittelt*. Des Weiteren sind Kräfte in diesem Zusammenhang gleichbedeutend mit Stößen, induziert durch einen *Impulsaustausch* zwischen Flüssigkeit und Festkörper. Da Impulse das Produkt von Masse und Geschwindigkeit sind, ist man mit einem Verfahren, das auf den Flachwassergleichungen beruht, in der Lage, auch die tiefengemittelten, durch die Strömung induzierten Krafte auf einen Festkörper zu berechnen. Diese Herangehensweise ist aus physikalischer Sicht dadurch problematisch, dass sich die Tiefenmittelung auf die Tiefe h des Fluids bezieht, also insbesondere auf die Höhe der Oberfläche über dem Bodenprofil und damit die eigentliche *Eintauchtiefe* des Festkörpers nicht berücksichtigt. Eine wesentliche Idee dieser Arbeit ist es, trotz des durch das zweidimensionale Verfahren gegebenen Fehlens von vertikalen Effekten die zur Verfügung stehenden Informationen (d.h., die aus dem zweidimensionalen Impulsaustausch resultierenden Kräfte) für eine visuell ansprechende und qualitativ korrekte Fluid-Struktur-Interaktion zu verwenden. Dabei steht jedoch die Flüssigkeit im Vordergrund, d.h., die Einflüsse der Festkörperbewegung auf das Fluid. Eine umfassende Beschreibung und Implementierung einer Festkörpersimulation kann im Rahmen dieser Arbeit nicht erfolgen. Trotzdem benötigt man eine einfache Approximation der Festkörperreaktion, um die generelle Fähigkeit des hier vorgestellten Verfahrens zu demonstrieren, mit einer beliebigen Simulation für die Festkörperphysik *koppelbar* zu sein.

Die mesoskopische Formulierung der Lattice-Boltzmann-Methode vereinfacht insbesondere Erweiterungen im Hinblick auf komplexe Geometrie bezogen auf das Rechengebiet. Diese Tatsache hat in jüngster Vergangenheit dazu geführt, dass einige Veröffentlichungen

[2] Beispielsweise gibt es in PhysX bereits vorgefertigte Module für Stoffsimulation, Festkörperphysik und diverse Partikeleffekte, jedoch keines für die Simulation von Flüssigkeiten.

[3] Wie beispielsweise mit *Qube-Mapping, Environment-Mapping.*

auch die Fluid-Struktur-Interaktion miteinbezogen haben. Insbesondere die Doktorarbeit von Caiazzo [7] ist dabei zu nennen, die einen umfassenden Überblick enthält. Verschiedene Verfahren für die Evaluierung von Kräften am Fluid-Struktur *interface* und insbesondere die Behandlung von komplizierter Geometrie und den in einer Abwandlung hier verwendeten Algorithmus für den Impulsaustausch beschreiben erstmals Mei et. al. [23]. Das in der vorliegenden Arbeit entwickelte Verfahren basiert auf einem *minimalen* Ansatz zur Erweiterung der Lattice-BGK-Methode für die SWE im Hinblick auf die Fluid-Struktur-Interaktion und ist als Einstiegspunkt für die Verwendung in der Computergraphik gemeint. Insbesondere soll dabei die Erweiterbarkeit der LBM demonstriert werden, im Hinblick auf einen möglichst minimalen Verlust ihrer primären Vorteile Einfachheit und Effizienz.

4.4.2 Vorgehensweise

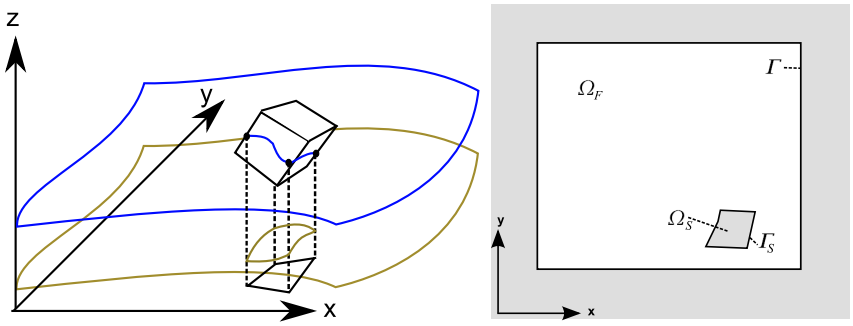
Das FSI Problem im Blickfeld dieser Arbeit zerfällt in mehrere Teilprobleme:

- Die **Parametrisierung** in das Fluid eingetauchter dreidimensionaler *instationärer* Festkörper, die zudem insbesondere nicht deformierbar sind;
- die **Reaktion** der zweidimensionalen Strömung auf die Bewegung der zweidimensionalen Repräsentation eingetauchter instationärer Festkörper aufgrund von *Impulsaustausch*,
- die Berechnung von **Kräften** auf das eingetauchte Objekt, bzw. dessen Parametrisierung durch Impulsaustausch und
- die **Reaktion** des dreidimensionalen Objektes auf die Strömung durch diese Kräfte und gegebenenfalls eine neue Parametrisierung des Festkörpers.

Zentral in dieser Arbeit sind dabei die Fluidreaktion und der Impulsaustausch. D.h., dass sowohl für die Parametrisierung, als auch die Festkörperphysik jeweils Black-Box-Verfahren eingesetzt werden.

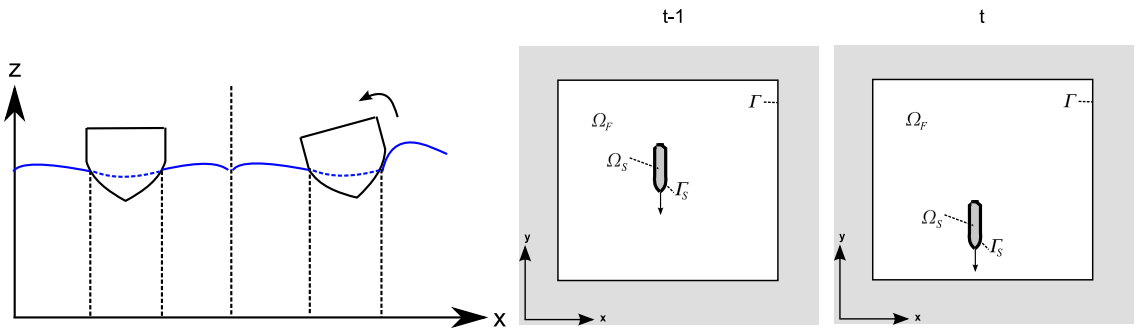
Parametrisierung und zeitliche Veränderung des Rechengebiets

Das Problem der Repräsentation eines dreidimensionalen Festkörpers derart, dass das *interface* zwischen Fluidoberfläche und Festkörper möglichst gut im $\mathbb{R}^2$ angenähert wird, um als Begrenzung (*boundary*) für die Strömungssimulation zu dienen, kann als orthogonale Projektion der *Schnittfläche* des Festkörpers mit der Fluidoberfläche auf die (x, y)-Ebene aufgefasst werden, wie in Abbildung 4.2(a) dargestellt. Vereinfachend kann man annehmen, dass dieser Schnitt aus einer diskreten Menge von Punkten auf der Fluidoberfläche besteht, deren z-Koordinate Null wird. Die resultierenden Punkte im $\mathbb{R}^2$ liefern unmittelbar einen Polygonzug, der als *boundary* für den Festkörper fungiert, wie Abbildung 4.2(b)



(a) 3D-Sicht (b) Resultierendes Rechengebiet

Abbildung 4.2: Parametrisierung von 3D-Festkörpern im $\mathbb{R}^2$



(a) Auslenkung eines Schiffsrumpfes, Invariantes Rechengebiet (b) Festkörperbewegung in zwei Dimensionen, Rechengebiet

Abbildung 4.3: Körper mit unwesentlicher zeitlicher Veränderung der *boundary*

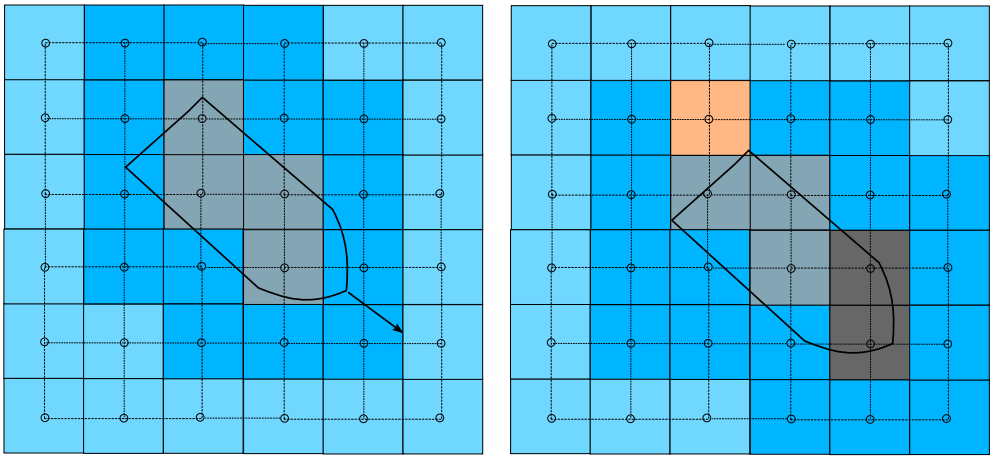


Abbildung 4.4: Diskreter Fall der Veränderung des Rechengebietes: links: Ein Festkörper und seine 2D-Verrasterung - Mögliche Zellzustände sind *solid* (hellgraue Zellen), *fluid* (hellblaue Zellen) und *boundary* (dunkelblaue Zellen); rechts: nach Bewegung des Festkörpers verschieben sich die *boundary* Zellen, Zellen müssen initialisiert werden (*solid to fluid*, rote Zellen) und werden aus dem Fluidgebiet entfernt (*fluid to solid*, dunkelgraue Zellen)

zeigt. Für diese Arbeit soll die Parametrisierung jedoch nachrangig sein. Daher beschränken wir uns auf solche Festkörper, deren zweidimensionale *boundary* sich in der Form zeitlich nur unwesentlich verändert. Diese Approximation ist für die meisten schwimmenden Objekte, insbesondere Wasserfahrzeuge sinnvoll, da sich ihre Eintauchtiefe nur unwesentlich verändert, wie Abbildung 4.3(a) zeigt. Mit dieser Annahme kann man jegliche Festkörperbewegung für die Evaluation der Fluidreaktion im $\mathbb{R}^2$ durchführen, vergleiche Abbildung 4.3(b). Damit erhält man, wenn die Ränder der eingetauchten Objekte bezogen auf das Rechengebiet mit Γ_{Si}^t und bezeichnet werden, das Rechengebiet zum Zeitpunkt t:

$$\overline{\Omega}^t = \Omega_F^t \cup \Gamma \cup \Gamma_S \cup (\bigcup_i \Gamma_{Si}^t) \tag{4.21}$$

Im diskreten Fall ist die Bestimmung der Randzellen dann gegeben durch eine Verrasterung des *boundary*-Polygonzuges über dem *lattice*. Die zeitliche Veränderung von $\overline{\Omega}^t$ bzw. die Veränderung der Festkörpergebiete $\Omega_{S,i}^t$ erzwingt in jedem Zeitschritt des Lattice-BGK-Algorithmus eine Klassifizierung der *lattice*-Zellen, wie in Abbildung 4.4 dargestellt. Dabei können Zellen ihren Zustand derart ändern, dass sie einer der folgenden Mengen zuzuordnen sind:

- $Z_{ij} \in Z_{\mathrm{F}}$: Z_{ij} ist eine *Fluidzelle*, d.h.:

$$\mathbf{x} \in \Omega_F^t$$

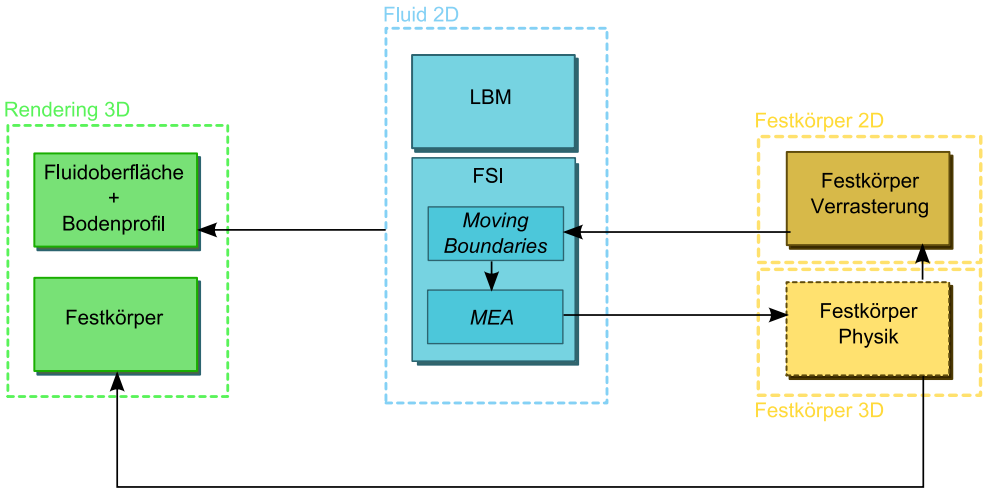


Abbildung 4.5: Überblick über das Gesamtverfahren

- $Z_{ij} \in Z_{\mathrm{S}}$: Z_{ij} ist eine *Festkörperzelle*, d.h.:

$$\mathbf{x} \in (\bigcup_i \Omega_{S,i}^t)$$

- $Z_{ij} \in Z_{\mathrm{FTS} \subset Z_{\mathrm{S}}}$: Z_{ij} war im letzten Zeitschritt *Fluidzelle* und wird eine *Festkörperzelle*, d.h. es gilt:

$$\mathbf{x} \in \overline{\Omega}^{t-1} \wedge \mathbf{x} \in (\bigcup_i \Omega_{S,i}^t)$$

- $Z_{ij} \in Z_{\mathrm{STF}}$: Z_{ij} war im letzten Zeitschritt *Festkörperzelle* und wird eine *Fluidzelle*, d.h.:

$$\mathbf{x} \in \overline{\Omega}^{t} \wedge \mathbf{x} \in (\bigcup_i \Omega_{S,i}^{t-1})$$

- $Z_{ij} \in Z_{\mathrm{B}}$: Z_{ij} ist eine *Boundaryzelle*, also:

$$\mathbf{x} \in (\bigcup_i \Gamma_{Si}^t) \cup \Gamma \cup \Gamma_S$$

Überblick über das Verfahren

Die Bewegung des Festkörpers läßt sich als mikroskopische Randbedingung (*Moving Boundaries*) für die LBM formulieren. Diese Erweiterung wird in den Abschnitten 4.5 und 4.6 beschrieben. Der Impulsaustauschalgorithmus (*Momentum-Exchange-Algorithm*, MEA), also insbesondere die Berechnung der auf den Festkörper einwirkenden diskreten Kräfte am *interface*, wird in Abschnitt 4.7 detailliert dargestellt. Nach diesen Vorüberlegungen stellt sich das Verfahren wie in Abbildung 4.5 gezeigt dar. Zentral dabei sind die in diesem

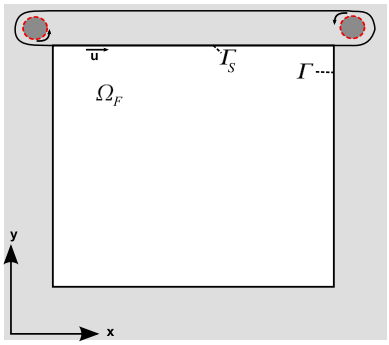


Abbildung 4.6: Prinzip des *Driven-Cavity*-Problems, resultierendes Rechengebiet

Kapitel vorgestellten Erweiterungen für die Lattice-BGK-Methode zur diskreten Lösung der Flachwassergleichungen, deren hardwareorientierte Implementierung im Kapitel 5 beschrieben und deren Effizienz in Kapitel 6 demonstriert wird. Die Leistungsfähigkeit des Gesamtverfahrens hängt von der Physiksimulation und der Darstellung (*rendering*) der Fluidoberfläche und der Szenenobjekte ab.

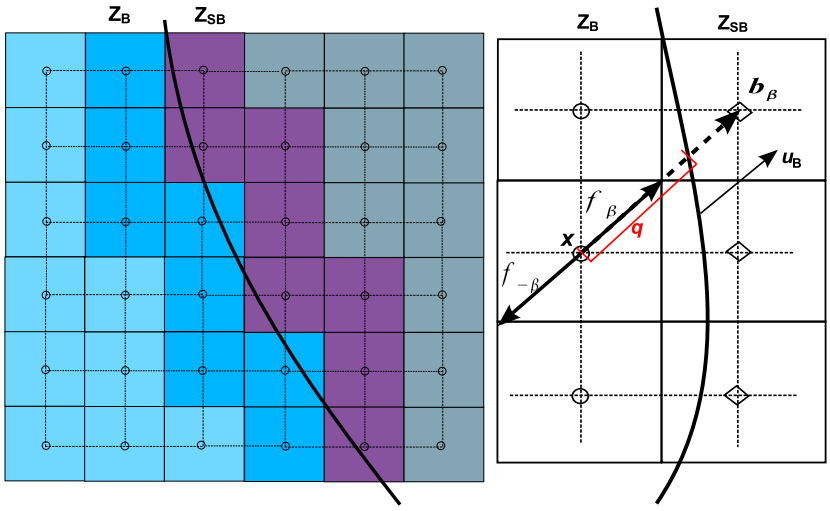
4.5 *Moving Boundaries* I: Fluidreaktion auf stationäre Festkörper mit beweglichen Wänden

4.5.1 Problemstellung

Die in Abschnitt 3.4.4 vorgestellte einfache *Bounce-Back*-Regel zur Randbehandlung in der LBM ist gültig für stationäre Festkörper, deren Wände eine Geschwindigkeit von Null haben. Bereits für eine sehr einfache Erweiterung der bisher betrachteten Probleme (3.12) und (3.13) muss diese angepasst werden. Man betrachte dazu ein einfaches rechteckiges Rechengebiet, bei dem sich die obere Seite der Begrenzung (eine begrenzende Wand) mit einer Geschwindigkeit $\mathbf{u}_B = (u, 0)$ bewegt. Dieses Problem wird *Lid-Driven Cavity* genannt und ist in Abbildung 4.6 schematisch dargestellt. Dabei induziert die sich bewegende Wand eine lokale Veränderung der Fluidgeschwindigkeiten, die bei der Randbehandlung berücksichtigt werden, d.h. konsistent zu Dirichlet'schen Randbedingungen für Nicht-Null-Werte sein muss.

4.5.2 Erweiterung der Randbehandlung

Im konkreten Fall der Lattice-Boltzmann-Methode muss für die *Lattice*-Zellen, die eine ausgehende *lattice-velocity* β haben, so dass $\mathbf{e}_\beta \Delta t$ den Rand des Rechengebiets schneidet, ein zusätzlicher Impuls zu der Verteilungsfunktion f_β^{temp} *nach der Kollision* addiert werden. Diese Impulse hängen von der Geschwindigkeit des Randes (der diskreten Geschwindigkeiten der Randzellen) des Festkörpers bzw. des Rechengebietes ab: Genau, wie



(a) Diskretisierung des *interface*

(b) *Bounce-Back*-Regel mit *Moving Boundary*

Abbildung 4.7: *Interface* und modifiziertes *Bounce-Back* Schema für *Moving Boundary*

das Fluidgebiet eine *Boundary* hat, definieren wir den Rand eines Nicht-Fluid-Gebietes als die Menge $Z_{\mathrm{SB}} \subset Z_{\mathrm{S}}$ der Zellen, die in zumindest einer ausgehenden Richtung zu einer Fluidzelle benachbart sind, vergleiche Abbildung 4.7(a). Dieser Rand bewegt sich mit einer Geschwindigkeit $\mathbf{u}_B$, die wie in Kapitel 3 die Dirichlet'schen Randbedingungen für die Strömung darstellt; im Fall des *Driven-Cavity*-Problems sind die diskreten Geschwindigkeiten an jedem Ort des Randes gleich groß.

Die Idee der *BFL*-Regel, benannt nach Bouzidi, Firdaouss und Lallemand [6], ist es nun, die fehlenden Momente für die Konsistenz mit den Dirichlet-Randbedingungen aus den Verteilungsfunktionen benachbarter Fluidzellen in zu β entgegengesetzter Richtung $-\beta$ zu interpolieren, vergleiche hierzu auch Abbildung 4.7(b). Dabei seien die Interpolationskoeffizienten C_1, C_2, C_3 und C_4 abhängig vom Abstand der Grenze des Fluidgebietes, bezogen auf den Ort $\mathbf{x} \in \Gamma^t_{Si}$ in Richtung β:

$$q = \frac{|\mathbf{b}_\beta - \mathbf{x}|}{\Delta x} \tag{4.22}$$

Dann kann man die Interpolation beschreiben als:

$$\begin{aligned} f^{\mathrm{temp}}_{-\beta}(\mathbf{x}, t+\Delta t) = C_1(q) f_\beta(\mathbf{x}, t) + C_2(q) f_\beta(\mathbf{x}+\mathbf{e}_{-\beta}, t) + C_3(q) f_{-\beta}(\mathbf{x}, t) + \\ C_4(q) 2\Delta x w_{-\beta} c_s^{-2} [\mathbf{u}_B(\mathbf{b}_\beta) \cdot \mathbf{e}_{-\beta}] \end{aligned} \tag{4.23}$$

Dabei ist $c_s = 1/\sqrt{3}$ die *Lattice-Schallgeschwindigkeit* und w_α von der Länge der *lattice-velocity* α abhängige Gewichte, die sich im Fall des D2Q9-*lattice* ergeben zu:

$$w_\alpha = \begin{cases} \frac{4}{9}, & \alpha = 0 \\ \frac{1}{9}, & \alpha \in \{1,3,5,7\} \\ \frac{1}{36}, & \alpha \in \{2,4,6,8\} \end{cases}$$

Die Interpolationskoeffizienten sind:

$$C_1(q) = \begin{cases} 2q, & q < \frac{1}{2} \\ \frac{1}{2q,} & q \geq \frac{1}{2} \end{cases}, \qquad C_2(q) = \begin{cases} 1-2q, & q < \frac{1}{2} \\ 0, & q \geq \frac{1}{2} \end{cases}, \qquad C_2(q) = \begin{cases} 0, & q < \frac{1}{2} \\ \frac{2q-1}{2q,} & q \geq \frac{1}{2} \end{cases}$$

und

$$C_4(q) = \begin{cases} 1, & \frac{q \leq 1}{2} \\ \frac{1}{2q}, & q \geq \frac{1}{2} \end{cases}$$

Im Gegensatz zu den *No-Slip*-Randkorrekturen wird hierbei also nicht nur der Wert für f_β^{temp} einer anderen Verteilungsfunktion zugeordnet, sondern der Wert unter Berücksichtigung der Geschwindigkeit $\mathbf{u}_B$ und der Verteilungsfunktionen der Nachbarzellen rekonstruiert. Zudem löst dieser Ansatz das Problem , dass die *No-Slip*-Randbehandlung nur stückweise lineare Approximationen der tatsächlichen Begrenzung des Fluidbereichs behandeln kann. Diese führt bei krummlienigen Begrenzungen bzw. Strecken, die nicht parallel zu einer der Koordinatenachsen sind, zu einem Abfall der Genauigkeit[4]. Die Erweiterung auf beliebige Geometrie ist jedoch mit dem Nachteil verbunden, dass in jedem Zeitschritt aufwändige Abstandsberechnungen nötig werden. Mit der Annahme, dass die Genauigkeitsunterschiede das visuelle Ergebnis bei hinreichend großer Auflösung nicht wesentlich beeinflussen, kann man die oben beschriebene BFL-Regel stark vereinfachen, indem man das Festkörpergebiet durch die Verrasterung über dem *lattice* ersetzt und trotzdem die bewegliche *boundary* berücksichtigt, d.h. es ist stets $q = \frac{1}{2}$. Damit vereinfacht sich (4.23) wegen $C_1(\frac{1}{2}) = C_2(\frac{1}{2}) = C_3(\frac{1}{2}) = 0$ und $C_4(\frac{1}{2}) = 1$ zu:

$$f_{-\beta}^{\text{temp}}(\mathbf{x}, t + \Delta t) = 6\Delta x w_{-\beta}[\mathbf{u}_B(\mathbf{b}_\beta) \cdot \mathbf{e}_{-\beta}] \tag{4.24}$$

[4]Die Genauigkeit der LBM ist zweiter Ordnung im *Innern*, mit der *Bounce-Back*-Regel an der *boundary* jedoch nur erster Ordnung.

Damit entfällt die Berechnung von q (Gleichung (4.22)) und insbesondere die Interpolation vollständig und wir können nun Probleme mit instationären Festkörpern mit beweglicher *boundary* lösen:

$$\begin{cases} \frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = S^b(Q), & \mathbf{x} \in \Omega_F, t > 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_B(\mathbf{x},t), & \mathbf{x} \in \Gamma, t > 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_0(\mathbf{x},t), & \mathbf{x} \in \overline{\Omega}, t = 0 \\ \mathbf{u}(\mathbf{x},t) = \mathbf{u}_B(\mathbf{x},t), & \mathbf{x} \in \Gamma_S, t > 0 \\ h(\mathbf{x},t) = h_0(\mathbf{x},t), & \mathbf{x} \in \overline{\Omega}, t = 0 \end{cases} \tag{4.25}$$

Dazu ist der Algorithmus so zu erweitern, dass die in diesem Abschnitt beschriebene Randbehandlung nach dem Kollisionsschritt durchgeführt wird (`MBNoSlipBC`):

Algorithmus 4.2 Lattice-BGK-Methode für inhomogene SWE, *Moving Boundaries* I

`Preprocessing`$\rightarrow h(\mathbf{x},0), \mathbf{u}(\mathbf{x},0)$
for $k = 1$ **to** n
  for all $\mathbf{x} \in \Omega_F$:
    for $\alpha = 0$ **to** 8:
      `EquilibriumDistribution`$\rightarrow f_\alpha^{eq}(\mathbf{x},(k-1)\Delta t)$
      `Collision`$\rightarrow f_\alpha^{\text{temp}}(\mathbf{x},k\Delta t)$
      for all $\mathbf{x} \in \Gamma_S$:
        `MBNoSlipBC`
      `Force`$\rightarrow f_\alpha^{\text{temp}}(\mathbf{x},k\Delta t) + \frac{\Delta t}{6e^2}e_{\alpha i}F_i^b$
      `Streaming`$\rightarrow f_\alpha(\mathbf{x} + \mathbf{e}_\alpha k\Delta t, k\Delta t)$
      for all $\mathbf{x} \in \Gamma \setminus \Gamma_S$:
        `NoSlipBC`
      `Extraction`$\rightarrow h(\mathbf{x},k\Delta t), \mathbf{u}(\mathbf{x},k\Delta t)$

4.6 *Moving Boundaries* II: Fluidreaktion auf instationäre Festkörper

4.6.1 Problemstellung

In den Abschnitten 4.4 und 4.5 haben wir bereits gesehen, welche Zustände von Zellen in jedem Zeitschritt möglich sind, und wie sie sich ergeben. Mit der im vorangehenden Abschnitt beschriebenen Methode kann bereits der Impuls, der durch die Bewegung von Festkörpern in das Fluid induziert wird, berechnet werden. Um die Beschreibung der Fluidreaktion abzuschließen benötigt man jedoch zusätzlich eine Methode, die *Lattice*-Zellen, die im letzten Zeitschritt noch einem Festkörper zuzuordnen waren (Zellen vom Typ Z_{STF}), mit Werten für die diskreten Verteilungsfunktionen zu initialisieren. Dabei gibt prinzipiell zwei Möglichkeiten:

1. **Direkte** Berechnung der Verteilungsfunktionen: Dabei werden die Verteilungsfunktionen extrapoliert, üblicherweise in Gegenrichtung der Bewegung des Festkörpers.

2. **Indirekte** Berechnung der Verteilungsfunktionen: Hier werden lediglich die makroskopischen Quantitäten des letzten Zeitschritts extrapoliert, um über die Berechnung der Gleichgewichte die Verteilungsfunktionen zu rekonstruieren.

Beide Ansätze werden von Lallemand und Luo [21] vorgestellt.

Obwohl Letztere durch das Fehlen der entsprechenden Nicht-Gleichgewichtsanteile aus dem letzten Zeitschritt weniger genau ist, als die direkte Methode, ist sie auch wesentlich weniger aufwändig. Im Folgenden wird daher eine abgewandelte indirekte Methode vorgestellt, der eine einfache Approximation der Extrapolationsrichtung zugrunde liegt, so dass diese stets nur in Richtung der *lattice-velocities* durchgeführt werden muss.

4.6.2 Initialisierung von Fluidzellen

Die Idee der indirekten Berechnung der fehlenden Verteilungsfunktionen ist es, die makroskopischen Quantitäten h und $\mathbf{u}$ für alle Zellen in Z_{STF} zu extrapolieren. Unabhängig von der gewählten Extrapolationsmethode, ist eine eindimensionale Extrapolation wesentlich weniger aufwändig, als die zweidimensionale Extrapolation eines Skalarfeldes. Aus diesem Grund ist es sinnvoll, eine Extrapolationsrichtung festzulegen, entlang dieser die Nachbarn zum Betrag des zu initialisierenden Wertes beitragen. Dabei sollte die Bewegungsrichtung des Festkörpers berücksichtigt werden. Insbesondere im Hinblick auf die Effizienz des Verfahrens, sollten dabei lediglich diejenigen Extrapolationsrichtungen erlaubt sein, die den *lattice-velocities* entsprechen. Anders ausgedrückt wird eine Festlegung der Richtung $-\beta$ benötigt, die die umgekehrte Bewegungsrichtung des Festkörpers bzw. den Geschwindigkeitsvektor $-\mathbf{u}_B$ möglichst gut annähert, vergleiche dazu Abbildung 4.8. Daher wird in dieser Arbeit die Extrapolationsrichtung so festgelegt, dass das Skalarprodukt von $\mathbf{e}_{-\beta}$ und $\mathbf{u}_B$ minimiert wird, wie in der Abbildung dargestellt. Der Einfachheit halber wird dabei zunächst davon ausgegangen, dass $\mathbf{u}_B$ für alle $z_{ij} \in Z_{SB}$ gleich sind. Mit festgelegter Richtung $-\beta$ kann die Extrapolation des Höhenfelds mit einer 3-Punkte Rückwärtsapproximation durchgeführt werden, wie Caiazzo [7] vorschlägt. Seien $\tilde{h}$ und $\tilde{\mathbf{u}}$ die extrapolierten makroskopischen Größen, dann kann man die Extrapolation wie folgt schreiben:

$$\tilde{h}(\mathbf{x}, t+\Delta t) = 3h(\mathbf{x}+\mathbf{e}_{-\beta}\Delta t, t+\Delta t) - 3h(\mathbf{x}+2(\mathbf{e}_{-\beta}\Delta t), t+\Delta t) + h(\mathbf{x}+3(\mathbf{e}_{-\beta}\Delta t), t+\Delta t) \tag{4.26}$$

Im Fall der Geschwindigkeiten kann zwischen $\mathbf{u}_B(\mathbf{b}_\beta)$ und den Geschwindigkeiten zweier Nachbarzellen in Richtung $-\beta$ interpoliert werden. Dabei muss der Abstand der Festkör-

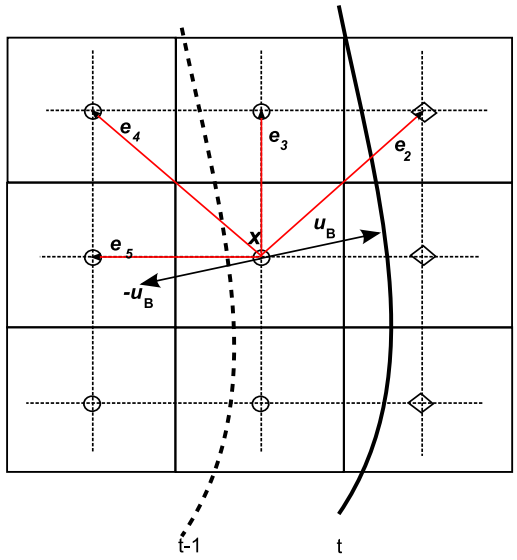


Abbildung 4.8: Bestimmung der Extrapolationsrichtung: Im mathematisch positiven Uhrzeigersinn werden die Skalarprodukte von $\mathbf{u}_b$ mit $\mathbf{e}_\alpha$ berechnet, das Minimum über diese Skalarprodukte maximiert den Winkel zwischen den beiden Vektoren und nähert $-\mathbf{u}_B$ daher am genauesten an.

perbegrenzung berücksichtigt und die Geschwindigkeit zusätzlich mit Δx skaliert werden, da sie in physikalischen und nicht in *Lattice*-Einheiten vorliegt. Damit ergibt sich:

$$\tilde{\mathbf{u}}(\mathbf{x}, t+\Delta t) = 2\Delta x \frac{\mathbf{u}_B(\mathbf{b}_\beta, t+\Delta t)}{q^2+3q+2} + 2q\frac{\mathbf{u}(\mathbf{x}+\mathbf{e}_{-\beta}\Delta t, t+\Delta t)}{q+1} - 2q\frac{\mathbf{u}(\mathbf{x}+2(\mathbf{e}_{-\beta}\Delta t), t+\Delta t)}{q+2} \tag{4.27}$$

Mit $q = \frac{1}{2}$ vereinfacht sich dies zu:

$$\tilde{\mathbf{u}}(\mathbf{x}, t+\Delta t) = \frac{8}{15}\Delta x \mathbf{u}_B(\mathbf{b}_\beta, t+\Delta t) + \frac{2}{3}\mathbf{u}(\mathbf{x}+\mathbf{e}_{-\beta}\Delta t, t+\Delta t) - \frac{2}{5}\mathbf{u}(\mathbf{x}+2(\mathbf{e}_{-\beta}\Delta t), t+\Delta t) \tag{4.28}$$

Mit den Approximationen des vorliegenden Abschnitts, können nun Probleme der Form

$$\begin{cases} \frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = S^b(Q), & \mathbf{x} \in \Omega_F, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t), & \mathbf{x} \in \Gamma, t > 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}^t, t = 0 \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t), & \mathbf{x} \in \Gamma_S, t > 0 \\ \forall i : \mathbf{u}(\mathbf{x}, t) = \mathbf{u}_B(\mathbf{x}, t), & \mathbf{x} \in \Gamma_{Si}^t, t > 0 \\ h(\mathbf{x}, t) = h_0(\mathbf{x}, t), & \mathbf{x} \in \overline{\Omega}^t, t = 0 \end{cases} \tag{4.29}$$

gelöst werden. Dazu ist der Algorithmus so zu verändern, dass die Extrapolationen (4.26) und (4.28) vor der Berechnung der Gleichgewichte durchgeführt wird (`FluidInit`). Zwischen den Zeitschritten sind die Extrapolationsrichtungen zu bestimmen (`ExtrapolationDirection`) und die *Lattice*-Zellen entsprechend zu klassifizieren (`CellFlags`).

Algorithmus 4.3 Lattice-BGK-Methode für inhomogene SWE, *Moving Boundaries* II

`Preprocessing`$\rightarrow h(\mathbf{x},0), \mathbf{u}(\mathbf{x},0)$
for $k = 1$ **to** n
`ExtrapolationDirection`$\rightarrow -\beta$
`CellFlags`$\rightarrow Z_{\mathrm{STF}}$
  for all $z_{lm} \in Z_{\mathrm{STF}}$:
    `FluidInit`$(-\beta) \rightarrow \tilde{h}(z_{lm}), \tilde{\mathbf{u}}(z_{lm})$
  for all $\mathbf{x} \in \Omega_F^{k\Delta t}$:
    for $\alpha = 0$ **to** 8:
      `EquilibriumDistribution`$\rightarrow f_\alpha^{eq}(\mathbf{x}, (k-1)\Delta t)$
      `Collision`$\rightarrow f_\alpha^{\mathrm{temp}}(\mathbf{x}, k\Delta t)$
      for all $\mathbf{x} \in \Gamma_S \cup (\bigcup_r \Gamma_{Sr}^{k\Delta t})$:
        `MBNoSlipBC`
      `Force`$\rightarrow f_\alpha^{\mathrm{temp}}(\mathbf{x}, k\Delta t) + \frac{\Delta t}{6e^2} e_{\alpha i} F_i^b$
      `Streaming`$\rightarrow f_\alpha(\mathbf{x} + \mathbf{e}_\alpha k\Delta t, k\Delta t)$
      for all $\mathbf{x} \in \Gamma \setminus (\Gamma_S \cup (\bigcup_r \Gamma_{Sr}^{k\Delta t}))$:
        `NoSlipBC`
      `Extraction`$\rightarrow h(\mathbf{x}, k\Delta t), \mathbf{u}(\mathbf{x}, k\Delta t)$

4.7 Festkörperreaktion: Impulsausstauschalgorithmus

4.7.1 Problemstellung

Die in Abschnitt 4.5 hergeleiteten diskreten Impulse, induziert durch die Bewegung des Festkörpers, verändern die lokalen Geschwindigkeiten des Fluids. Andererseits wirkt die Strömung ebenfalls auf den Festkörper und verändert dessen Geschwindigkeit durch eine Spannung, die dadurch entsteht, dass der hydrostatische Druck am *Interface* zwischen Fluid und Festkörper lokale Unterschiede aufweist. Diese lokalen Kräfte gilt es, als letzten Schritt der FSI zu berechnen. Sie werden in einer echten Koppelung von Festkörperphysik und Strömungssimulation an den Löser für die Festkörperphysik in jedm Zeitschritt übergeben. Wieder erweist sich in dieser Situation die LBM als vielversprechende Methode, da durch die mikroskopischen Verteilungsfunktionen jeder diskrete Impuls zu jedem Zeitpunkt zur Verfügung steht.

4.7.2 Diskreter Impulsaustausch

Gesucht ist die Gesamtkraft F_S, die auf einen Punkt $\mathbf{b}_\beta$ auf der *boundary* eines Festkörpergebietes einwirkt. Der ausgetauschte Impuls ist zunächst die Differenz der (bezogen auf einen Punkt $\mathbf{x}$ auf der *boundary* des Fluids, vergleiche Abbildung 4.9) von eingehender (*incoming*) und ausgehender (*outgoing*) Verteilung. Den *Incoming*-Anteil haben wir dabei

bereits mit Gleichung (4.24) berechnet. Der *Outgoing*-Anteil ist die Verteilung von Partikeln, die am Ort $\mathbf{x}$ vor dieser Korrektur anzutreffen ist. Damit kann der ausgetauschte Impuls berechnet werden als

$$f_{-\beta}^{\mathrm{MEA}}(\mathbf{x},t) = e_{\beta i} f_{\beta}^{\mathrm{temp}}(\mathbf{x},t) - e_{-\beta i} f_{-\beta}^{\mathrm{temp}}(\mathbf{x},t+\Delta t)$$

Wegen $\mathbf{e}_{\beta} = -\mathbf{e}_{-\beta}$ gilt:

$$f_{-\beta}^{\mathrm{MEA}}(\mathbf{b},t) = e_{\beta i}(f_{\beta}^{\mathrm{temp}}(\mathbf{x},t) + f_{-\beta}^{\mathrm{temp}}(\mathbf{x},t+\Delta t)) \tag{4.30}$$

Summation über alle Richtungen liefert dann die Gesamtkraft am Punkt $\mathbf{b}$,

$$F(\mathbf{b},t) = \sum_{\alpha} f_{\alpha}^{\mathrm{MEA}}(\mathbf{b},t) \tag{4.31}$$

und wir können mit hilfe des Impulsaustausches (`MomentumExchange`) den Algorithmus wie folgt erweitern:

Algorithmus 4.4 Lattice-BGK-Methode für inhomogene SWE, FSI

`Preprocessing`$\rightarrow h(\mathbf{x},0), \mathbf{u}(\mathbf{x},0)$
for $k = 1$ **to** n
`ExtrapolationDirection`$\rightarrow -\beta$
`CellFlags`$\rightarrow Z_{\mathrm{STF}}$
  for all $z_{lm} \in Z_{\mathrm{STF}}$:
    `FluidInit`$(-\beta) \rightarrow \tilde{h}(z_{lm}), \tilde{\mathbf{u}}(z_{lm})$
  for all $\mathbf{x} \in \Omega_F^{k\Delta t}$:
    for $\alpha = 0$ **to** 8:
      `EquilibriumDistribution`$\rightarrow f_{\alpha}^{eq}(\mathbf{x},(k-1)\Delta t)$
      `Collision`$\rightarrow f_{\alpha}^{\mathrm{temp}}(\mathbf{x},k\Delta t)$
      `MomentumExchange`$\rightarrow f_{\alpha}^{\mathrm{MEA}}(\mathbf{b},k\Delta t)$
      for all $\mathbf{x} \in \Gamma_S \cup (\bigcup_r \Gamma_{Sr}^{k\Delta t})$:
        `MBNoSlipBC`
      `Force`$\rightarrow f_{\alpha}^{\mathrm{temp}}(\mathbf{x},k\Delta t) + \frac{\Delta t}{6e^2} e_{\alpha i} F_i^b$
      `Streaming`$\rightarrow f_{\alpha}(\mathbf{x} + \mathbf{e}_{\alpha} k\Delta t, k\Delta t)$
      for all $\mathbf{x} \in \Gamma \setminus (\Gamma_S \cup (\bigcup_r \Gamma_{Sr}^{k\Delta t}))$:
        `NoSlipBC`
      `Extraction`$\rightarrow h(\mathbf{x},k\Delta t), \mathbf{u}(\mathbf{x},k\Delta t)$

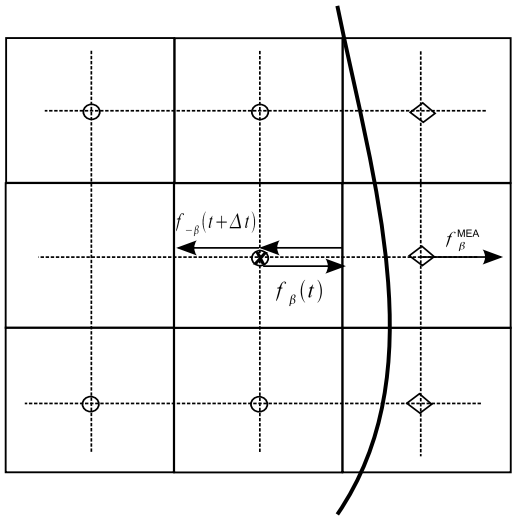


Abbildung 4.9: Mikroskopischer Impulsaustausch

5 Implementierung

Im Folgenden sollen insbesondere die Implementierungen der in Kapitel 4 vorgestellten und teilweise modifizierten Algorithmen für eine einfache Fluid-Struktur-Interaktion beschrieben werden. Die Implementierung der Erweiterungsmodule erfolgt zweistufig. Eine CPU-Referenzimplementierung ist unerlässlich, um die Korrektheit der optimierten GPU-Operationen und Löser zu gewährleisten und ermöglicht den Nachweis, dass letztere innerhalb der durch den gewählten Gleitkommatyp festgelegten Grenzen numerisch dieselbe Genauigkeit erreichen, wie auf der CPU. Da in HONEI Applikationen prinzipiell Konkatenationen von anwendungsspezifischen Operationen sind, bleibt der Datenfluss innerhalb der Anwendung (hier: des Lösers) unabhängig von der gewählten Zielarchitektur gleich: Die Operationsaufrufe werden lediglich an ein anderes *backend* weitergeleitet. Das (weitestgehend nicht optimierte) CPU-*backend* ermöglicht so auch das Testen der Partitionierung von Funktionalität einer Anwendung. Alle so spezifizierten Anwendungskernel können dann einzeln optimiert und auf die Zielhardware portiert werden.

Das HONEI *frontend*, das den in Kapitel 3 beschriebenen Basisalgorithmus 3.1 beinhaltet, wurde im Rahmen dieser Arbeit mitentwickelt und wird von Ribbrock im Detail vorgestellt [32]. Für das Verständnis der Konzeption der hier vorgestellten Implementierungen ist es jedoch unerlässlich, die Entwicklung und die Grundkonzepte dieser Bibliothek zu beschreiben.

5.1 Überblick über HONEI LBM

Aufgrund der Tatsache, dass es sich bei der Lattice-Boltzmann Methode prinzipiell um eine *gitterfreie* Methode handelt, konnte die erste Implementierung eines Lösers für die Flachwassergleichungen mit der LBM direkt ohne zusätzliche Datenstrukturen auf der Basis der HONEI-*container* erfolgen. Die Skalarfelder wurden dabei stets als dicht besetzte Matrizen des entsprechenden Gleitkommatyps repräsentiert, die beteiligten Operationen wie die `Extraction` mit Hilfe der Basisoperationen aus der Bibliothek für Operatoren der linearen Algebra realisiert. Obwohl diese Herangehensweise für einfache rechteckige Rechengebiete ausreicht, ist sie für kompliziertere Geometrie und Szenarien aus zwei Gründen unzureichend:

1. **Effizienz**: Die Komposition von komplexen Rechengebieten aus einem rechteckigen Bereich erfordert die Maskierung von *lattice*-Zellen derart, dass die aus $\overline{\Omega}$ ausgeschnit-

tenen Bereiche bei der Berechnung der unbekannten Verteilungsfunktionen f_α (bzw. f_α^{eq} und f_α^{temp}) exkludiert und zusätzlich bei der Randbehandlung berücksichtigt werden. Eine solche Unterscheidung schlägt sich unweigerlich in teuren Programmverzweigungen (`if`-Blöcken) auf der Ebene der *Elemente* der Matrizen nieder. Im Fall eines statischen Rechengebietes (im Kontext dieser Arbeit also insbesondere, wenn ausschließlich stationäre Festkörper vorhanden sind) sollte dies daher durch eine Form indirekter Indizierung verhindert werden, was möglich ist, da das invariante Rechengebiet a priori bekannt ist.

2. **Speicherbedarf:** Ein wesentlicher Nachteil der Lattice-Boltzmann Methode ist die verglichen mit anderen numerischen Verfahren relativ hohe Speicherkomplexität, da neben den makroskopischen Größen auch die mikroskopischen Verteilungsfunktionen gespeichert werden, was im hier vorliegenden Fall bei einem $N \times M$-*lattice* zu $O(31NM)$ zu speichernden Gleitkommazahlen führt. Die einfache Repräsentation des *lattice* durch maskierte Matrizen erfordert auch für kleine Fluidbereiche stets Matrizen für das vollständige, alle stationären Festkörper einschließende rechteckige Gebiet, vergleiche beispielsweise Abbildung 3.3(b).

Zusätzlich ist eine Komposition der LBM-Module aus Basisoperationen aus den in Kapitel 2 genannten Gründen nicht effizient. Man erinnere sich, dass beispielsweise die `Extraction` eine Akkumulation der mikroskopischen Verteilungsfunktionen ist, die einem wiederholten Aufruf einer komponentenweisen Matrixsumme entspricht. Damit sind viele unnötige Speichertransfers verbunden, vergleiche dazu das Beispiel in Abschnitt 2.2.2. Diese Überlegungen führten zur Entwicklung von Datenstrukturen, auf denen aufbauend die LBM-spezifischen *kernel* entwickelt wurden.

5.1.1 *Lattice*-Kompression

Die Repräsentation des zweidimensionalen *lattice* basiert auf einer Abbildung der zu $\overline{\Omega}$ gehörenden Daten auf einen kontinuierlichen Speicherbereich. Die Matrizen werden so durch Vektoren ersetzt und enthalten nur noch Daten der Teile des Rechengebietes, die mit Fluidzellen assoziiert sind, wie in Abbildung 5.1 dargestellt wird. Während dadurch bereits der Speicherbedarf je nach Beschaffenheit der Szenengeometrie stark gesenkt werden kann, ist die zentrale Idee bezüglich der Effizienz, bei dieser Transformation in einer zusätzlichen Datenstruktur zu speichern, welche Zugriffe von einer spezifischen *lattice*-Zelle aus in den den *lattice velocities* entsprechenden Richtungen gültig sind. Diese Information wird in Form von Intervallen bereitgestellt, die den Operationen angeben, innerhalb welcher Indexgrenzen ein Zugriff auf die Nachbarn in einer spezifischen Richtung gültig sind, so dass jede diesbezügliche konditionale Verzweigung innerhalb der *kernel* entfallen kann.

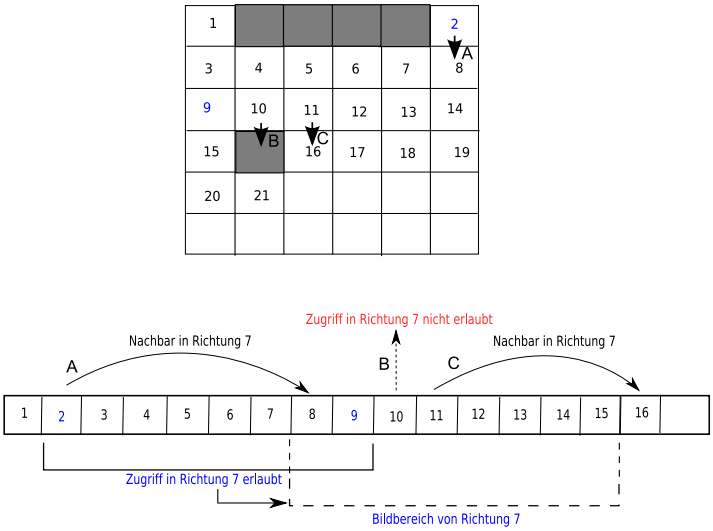


Abbildung 5.1: *Lattice*-Kompression in HONEI: Indirekte Indizierung der Nachbarschaften (hier am Beispiel von $\alpha = 7$) und Gültigkeitsintervalle.

5.1.2 Module und Datenfluss

Das in HONEI zentrale Konzept der Anwendungskernel (vergleiche Abschnitt 2.2.2) führt unweigerlich zu der Überlegung, inwieweit Teile des Algorithmus (Lösers) zu solchen Operationen zusammengefasst werden können. Während dies für die `Collision` und das `Streaming` ohne Verlust der Generalität möglich und auch üblich ist, stehen einem solchen Vorgehen Überlegungen bezüglich der Wiederverwendbarkeit der entsprechenden Module im Wege. Ein immenser Effizienzgewinn würde beispielsweise durch eine Integration der Berechnung der lokalen Gleichgewichte f_α^{eq} in den Kollisionsschritt der LBM erreicht werden können, insbesondere im Hinblick auf eine drastisch verminderte Anzahl von Speichertransfers und stark erhöhte Datenwiederverwendung durch schnelle Caches. Zusätzlich könnte eine Zwischenspeicherung dieser Gleichgewichtsfunktionen entfallen, was die Speicherkomplexität der Methode um immerhin $O(9NM)$ verringern würde. Gerade dieser Schritt des Algorithmus, `EquilibriumDistribution`, bestimmt jedoch in der LBM, welche Strömungsgleichungen gelöst werden (bzw. allgemein, welches Problem gelöst wird) und ist damit ein häufig ausgewechseltes Modul. Beispielsweise enthält HONEI neben einem Modul für die Berechnung der Gleichgewichtsanteile der SWE auch eines für die Navier-Stokes Gleichungen, um die Vergleichbarkeit von Ergebnissen bezüglich Leistung und Genauigkeit mit anderen Lösern für die Strömungssimulation zu gewährleisten (vergleiche Kapitel 6). Um die Wiederverwendbarkeit des LBM-*frontends* zu erhöhen, ist es daher sinnvoll, dieses Modul separat zu implementieren. Gleiches gilt für die Implementierung der Randbedingungen, die ebenfalls häufig durch andere ersetzt bzw. erweitert werden, wie beispielsweise auch in dieser Arbeit.

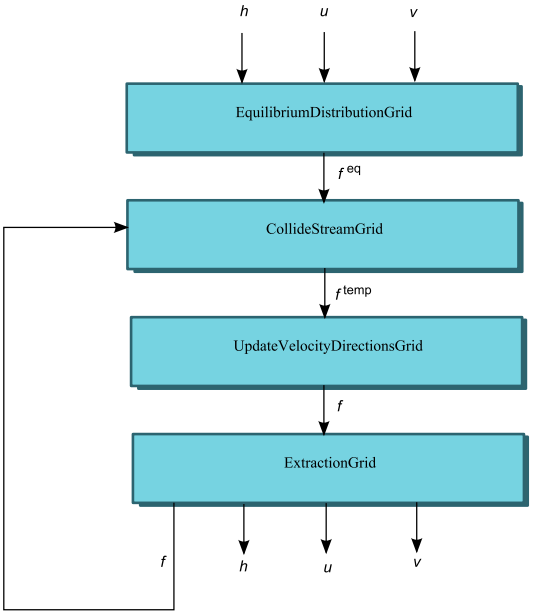


Abbildung 5.2: Datenfluss im Basisalgorithmus

Mit der Verschmelzung von Kollisions- und Transportschritt ergibt sich das in Abbildung 5.2 dargestellte Flussdiagramm für den Basisalgorithmus 3.1. Dabei werden die Module in Anlehnung an das Konzept der oben beschriebenen komprimierten Datenstruktur (in HONEI `PackedGrid` genannt) jeweils mit dem Suffix `Grid` versehen. Das Modul `UpdateVelocityDirectionsGrid` ist die verallgemeinerte Randbehandlung, von der die in den Algorithmen 3.1 bzw. 4.1 - 4.4 verwendete *bounce-back*-Regel (`NoSlipBC`) eine Instanz ist.

5.2 Konzeption der FSI-Operationen, CPU Referenzimplementierung

Alle Module für die Algorithmen 4.1 - 4.4 sind in dieser Arbeit nach den folgenden Kriterien konzipiert:

1. **Erweiterung** des Basisalgorithmus: Eine möglichst vollständige Nutzung des im letzten Abschnitt beschriebenen Lösers ist nicht nur aus softwaretechnischer Sicht effizient. Insbesondere die automatische Verwendung des `PackedGrid` erlaubt die direkte Repräsentation stationärer Festkörper mit unbeweglichen Wänden ohne jegliche Anpassung bei hoher Effizienz. Daher und aufgrund der Tatsache, dass es möglich sein soll, die in Kapitel 4 beschriebenen Erweiterungen inkrementell, d.h. beispielsweise nur die Kraftterme ohne FSI, in das Verfahren einzubinden, werden die entsprechenden Berechnung in Spezialmodulen gekapselt, die dem Algorithmus bei Bedarf hinzugefügt werden können. Invariante Basismodule bedeuten zudem, dass die erwei-

terten Algorithmen bereits teilweise auf den verschiedenen von HONEI unterstützten Hardwarearchitekturen implementiert und optimiert sind und so eine Konzentration der Optimierungen auf die tatsächlichen Erweiterungen erfolgen kann.

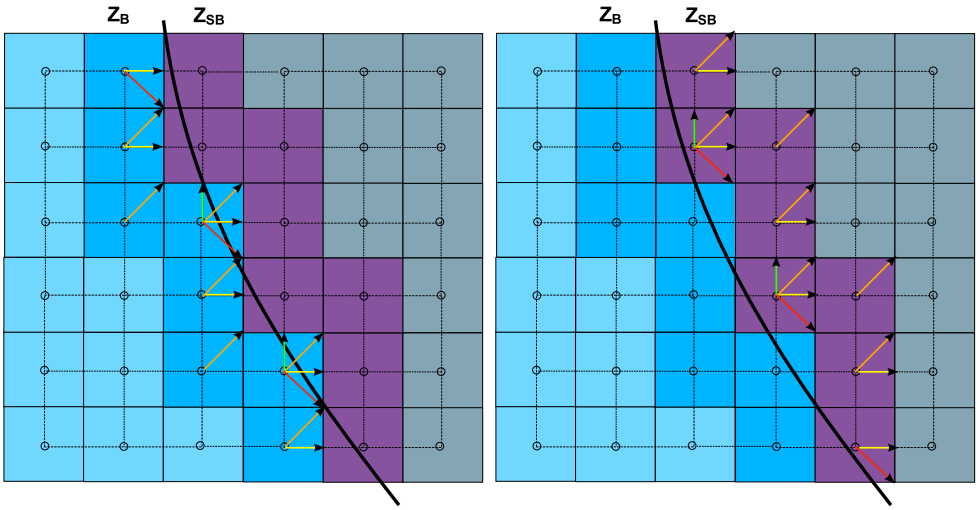
2. **Minimale Anzahl** von *kerneln* bei möglichst hoher Flexibilität: Wie bereits mehrfach angemerkt, ist es relativ schwierig, einen Kompromiss zwischen Effizienz und Wiederverwendbarkeit einer Softwarebibliothek zu erreichen. Daher erfolgt die Implementierung der Erweiterungsmodule so, dass Berechnungen zwar wann immer möglich zusammengefasst werden, jedoch die in Kapitel 4 beschriebenen Algorithmen möglichst auch einzeln ohne zu großen zusätzlichen Anteil nicht benötigter Berechnungen auskommen. Dabei wird des Weiteren darauf geachtet, dass einzelne Module auch hinsichtlich zukünftiger Erweiterungen (beispielsweise Randbedingungen höherer Ordnung) einfach zu ersetzen bzw. zu modifizieren sind.

3. **Portierbarkeit**: Bereits beim Entwurf der Algorithmen wurden Vereinfachungen vorgenommen, die die Effizienz erhöhen bei einem gleichzeitigen vertretbaren Verlust der Genauigkeit. Diese schließen die Vereinfachung des semi-impliziten Schemas bei der Auswertung der Kraftterme in Abschnitt 4.2.3, die Beschränkung der zeitlichen Veränderung der Fluid-*boundary* in Abschnitt 4.5 sowie die Beschränkung auf feste Extrapolationsrichtungen in Abschnitt 4.6 ein. Diese Vereinfachungen wurden insbesondere aufgrund von Überlegungen bezüglich der Eigenarten der Zielhardware vorgenommen (s.u.).

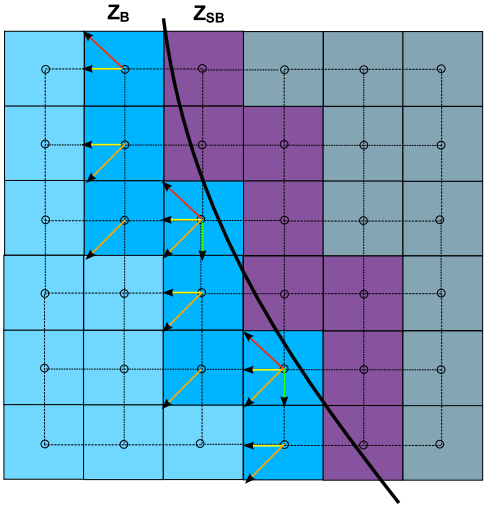
Für die Implementierung der von den Algorithmen 4.1 - 4.4 benötigten Funktionalität sind nach diesen Vorüberlegungen vier zusätzliche Module nötig. Kraftterme (für Algorithmus 4.1) sollten generell einzeln einem Löser hinzugefügt werden können. Aus diesem Grund werden die entsprechenden Module für den Gefälle- bzw. Reibungskraftterm separat durch verschiedene Spezialisierungen des *Templates* für den Kraftterm implementiert (`ForceGrid<SLOPE>` bzw. `ForceGrid<FRICTION>`). Dies ist auch insofern sinnvoll, als dass beide unterschiedliche Speicherzugriffsmuster erfordern: Während wegen der Richtungsableitungen im Gefällekraftterm auch Zugriffe auf Nachbarzellen erforderlich sind, ist die Reibung an einem Ort lediglich vom Material und den lokalen makroskopischen Größen abhängig. Lediglich zwei zusätzliche Operationen werden für die FSI Algorithmen benötigt. Ein Modul `CollideStreamFSI` dass die Randbedingungen für *Moving Boundaries* und den Impulsaustausch realisiert (Algorithmen 4.2 und 4.4) und ein weiteres, `BoundaryInitFSI` , dass für die Initialisierung von Fluidzellen zuständig ist (Algorithmen 4.3 und 4.4). Der Kompromiss bezüglich des zweiten Kriteriums besteht hier also darin, dass die diskreten Impulse durch den Impulsaustausch auch dann berechnet werden, wenn sie nicht benötigt werden, wie etwa bei einem sich schnell bewegenden, schweren Wasserfahrzeug, dessen Rumpf zwar das Fluid beeinflusst, der Einfluss auf das Fahrzeug jedoch visuell vernachlässigbar klein ist oder auch dann, wenn nur instationäre Festkörper vorhanden sind.

5.2.1 Repräsentation instationärer Festkörper, *Streaming*-Korrektur

Die Effizienz der Implementierung der FSI Module ist prinzipiell davon abhängig, wie die Veränderung des Rechengebietes durch Bewegung instationärer Festkörper realisiert wird. Obwohl die in Abschnitt 5.1.1 beschriebene Methode für statische Rechengebiete effizient ist, ist das dafür nötige Preprocessing sehr aufwändig, insbesondere da es die Allokation des Speichers für die Daten beinhaltet. Aber auch ohne letztere ist die Berechnung der Hilfs/-in/-ter/-val/-le und der nötigen Informationen für die Randbedingungen durch Verzweigungen so teuer, dass sie nicht für ein sich dynamisch veränderndes Rechengebiet geeignet ist. Für die Implementierung der Erweiterungen im Rahmen dieser Arbeit wird daher ein alternativer Ansatz gewählt. Dabei wird das `PackedGrid` normal für den zeitlich invarianten Teil des Rechengebietes verwendet. Die nötigen Informationen für sich bewegende Festkörper werden in einer zusätzlichen neuen Datenstruktur gespeichert: `PackedSolidData`. Diese ist prinzipiell analog zu der von dem Basisverfahren genutzten zentralen Datenstruktur `PackedGridData`, d.h., die Vektoren für die Daten entsprechen in der Größe exakt denen des komprimierten Gitters, enthalten also nur Daten, die mit Fluidzellen assoziiert sind. Die Hauptaufgabe dieser Datenstruktur ist die Speicherungen einer Reihe von boole'schen Variablen (*flags*) für jede *lattice*-Zelle, die die Zugehörigkeit zu je einer der in Kapitel 4 eingeführten Zellmengen anzeigt, die mit dem Inneren eines beweglichen Festkörpers (Z_{S}), dem entsprechenden *Interface* (Z_{SB} bzw. Z_{B}) oder den zu initialisierenden Fluidzellen (Z_{STF}) assoziiert sind. Dadurch kann jede Operation zu jedem Zeitschritt auf den aktuellen Zustand des nun veränderbaren *lattice* zugreifen. Die *flags* werden durch die Verrasterung des parametrisierten beweglichen Festkörpers zum aktuellen und zum letzten Zeitpunkt bestimmt, worauf genauer im nächsten Abschnitt eingegangen wird. An dieser Stelle ist jedoch zunächst wichtig, dass für die Basismodule, insbesondere `CollideStreamGrid` und `UpdateVelocityDirectionsGrid` alle im komprimierten *lattice* gespeicherten Zellen Fluidzellen sind. Dies bedeutet insbesondere, dass erstere Operation einen Transportschritt auf Festkörperzellen durchführen kann und letztere die Begrenzung des Festkörpers nicht als solche erkennt (vergleiche Abbildung 5.3). Da die Implementierung der BFL-Regel (vergleiche Abschnitt 4.5) ohnehin ein neues Modul erfordert um die von Null verschiedenen Dirichlet'schen Randbedingungen zu realisieren und diese insbesondere für in irgendeiner Form bewegliche Strukturen gilt, kann das falsche *streaming* an dieser Stelle effizient korrigiert werden. Die Idee der hier vorgestellten Operation `CollideStreamFSI` ist es daher, für die Zellen $z_{ij} \in Z_{\mathrm{S}}$ die von einem inkorrekten Transport betroffenen Zellen zu identifizieren: Zellen, die nach der Ausführung von `CollideStreamGrid` zu Z_S gehören und in einer Verteilungsfunktion einen von Null verschiedenen Wert haben, müssen korrigiert werden, was durch einen Rücktransport realisiert wird der simultan die vereinfachte BFL-Regel implementiert, wie in Abbildung 5.3 dargestellt. Beispielsweise impliziert ein von Null verschiedener Wert in $f_1^{\mathrm{temp}}(\mathbf{x}, t)$ an



(a) Vor dem regulären *Streaming* (b) Nach dem regulären *Streaming*



(c) Nach der Korrektur

Abbildung 5.3: Prinzip der *Streaming*-Korrektur

der Stelle einer Festkörperzelle, dass ein Transport vom Ort $\mathbf{x} - \boldsymbol{\Delta}\mathbf{x}$ stattgefunden haben muss. Die Reflektionsbedingung kann daher sofort angewendet werden und der Gleichung (4.24) entsprechend der neue Wert für $f_5^{\text{temp}}(x - \Delta x, t + \Delta t)$ berechnet werden. Betrachtet man zudem Algorithmus 4.4, so sieht man leicht, dass der Impulsaustausch notwendigerweise nach dem (somit korrekten) Kollisionsschritt erfolgt. Tatsächlich hat man die für die Berechnung von f_α^{MEA} nötigen Komponenten an dieser Stelle bereits berechnet. Man vergleiche dazu noch einmal Gleichung (4.30):

$$f_{-\beta}^{\text{MEA}}(\mathbf{b}, t) = e_{\beta i}(f_\beta^{\text{temp}}(\mathbf{x}, t) + f_{-\beta}^{\text{temp}}(\mathbf{x}, t + \Delta t))$$

- Der *outgoing*-Anteil $f_\beta^{\text{temp}}(\mathbf{x}, t)$ Ist der Wert der Verteilungsfunktion vor der Ausführung *streaming*-Korrektur (am Beispiel: $f_1^{\text{temp}}(\mathbf{x}, t)$).
- Der *incoming*-Anteil ist der Wert der durch die BFL-Regel korrigierten Verteilungsfunktion in umgekehrter Richtung (am Beispiel: $f_5^{\text{temp}}(x - \Delta x, t + \Delta t)$) Die zusätzliche Indexverschiebung rührt von dem fälschlicherweise zuvor durchgeführten Transport her, der damit ebenfalls korrigiert ist.
- Letztlich ist auch $\mathbf{b}$ explizit durch die Zelle $z_{ij} \in Z_{\text{S}}$ gegeben, an der die Korrektur vorgenommen wurde, da sie genau dem Punkt der Festkörper-*boundary* entspricht, der z_{ij} in dieser Richtung am nächsten liegt.

Zusätzlich zu der Berechnung der diskreten Impulse muss das Modul `CollideStreamFSI` sicherstellen, dass die mit Festkörperzellen assoziierten Daten für die Berechnung der makroskopischen Quantitäten auf Null zurückgesetzt werden, was die Zellen $z_{ij} \in Z_{\text{FTS}}$, also die Zellen, die im letzten Zeitschritt Fluidzellen waren und im aktuellen Zeitschritt Festkörperzellen sind, mit einschließt. Damit kann der Basisalgorithmus so ergänzt werden, dass `CollideStreamFSI` unmittelbar nach `CollideStreamGrid` durchgeführt wird. Das Modul `UpdateVelocityDirectionsGrid` ist dann nur noch für die *boundary* der äußeren Begrenzung des Rechengebietes und der stationären Hindernisse zuständig.

5.2.2 Verrasterung, Initialisierung von Fluidzellen

Die Verrasterung einer Parametrisierung eines Festkörpers über dem *lattice* kann effizient mit einem gemischten Verfahren aus dem bekannten Bresenham-Algorithmus für die Verrasterung von Linien und einem Füllverfahren realisiert werden, dass die von den verrasterten Linien eines geschlossenen Polygonzugs eingeschlossenen Bereiche als zu Z_{S} zugehörig markiert[1]. Mit Hilfe dieses Verfahrens ist die Menge Z_{S} für das Modul `CollideStreamFSI` bereits gegeben.

[1] Dabei wird ein modifiziertes so genanntes *Scan-Line*-Verfahren eingesetzt, dass die *lattice-Zellen* in einem lokalen Fenster, begrenzt durch die minimalen und maximalen Koordinaten der Linien des Polygonzuges, als zu Z_{S} zugehörig markiert, wenn sie zum Inneren des Polygons gehören.

Die Berechnung der initialen Werte für die makroskopischen Quantitaten für die Zellen $z_{ij} \in Z_{\mathrm{STF}}$ ist dann noch durchzuführen: Dies übernimmt das Modul `BoundaryInitFSI`: Dabei werden sowohl die Extrapolation des Höhenfelds h als auch die durch Gleichung 4.27 gegebene Interpolation der makroskopischen Geschwindigkeiten berechnet. Da die Berechnung der neu zu initialisierenden *lattice*-Zellen von der Position des Festkörpers im letzten Zeitschritt abhängt, muss `PackedSolidData` neben den diskreten Impulsen f_α^{MEA} zusätzlich die Information speichern, welche Zellen im letzten Zeitschritt zu einem Festkörper gehörten ($\overline{Z_{\mathrm{S}}}$). Für eine beliebige Zelle des *lattice* kann dann effizient festgestellt werden, ob eine Initialisierung nötig ist, indem ein boole'scher Operator verwendet wird: Eine Zelle wird initialisiert, wenn sie im vorherigen Zeitschritt zu einem Festkörper gehörte und im aktuellen Zeitschritt nicht, d.h.:

$$z_{ij} \in Z_{\mathrm{STF}} \Leftrightarrow (z_{ij} \in \overline{Z_{\mathrm{S}}}) \wedge (z_{ij} \notin Z_{\mathrm{S}})$$

5.2.3 Datenfluss im erweiterten Verfahren

Da die Kraftterme mit den temporären Verteilungsfunktionen lediglich kumuliert werden und selbst nur von den makroskopischen Größen abhängen, können sie zu einem beliebigen Zeitpunkt nach dem Kollisionsschritt berechnet werden. Die Fluid-Initialisierung ist zwingend vor der Berechnung der Gleichgewichte durchzuführen, während die erweiterte Randbehandlung mit der in Abschnitt 5.2.1 eingeführten *Streaming*-Korrektur unmittelbar nach dem Kollisions- und Transportschritt des Basisalgorithmus aufgerufen werden muss. Damit kann man den erweiterten Algorithmus wie in Abbildung 5.4 darstellen.

5.3 GPU Implementierung

5.3.1 Überblick

Durch das CUDA-*backend* ermöglicht HONEI die Konzentration der Entwicklungsarbeit auf die Operationen selbst. Insbesondere Speichertransfers werden implizit ausgeführt: Der Entwickler muss lediglich wissen, welche Daten für die Operation notwendig sind und ob auf sie im Verlauf der Berechnung nur lesend, nur schreibend oder lesend und schreibend zugegriffen wird.

Im Folgenden sollen daher grundlegende Informationen zur Implementierung der FSI-*kernel* gegeben werden. Dabei erfolgt die Darstellung von Programmcode lediglich exemplarisch. Für aktuelle Versionen des Codes sei auf die Homepage des Projektes verwiesen [9].

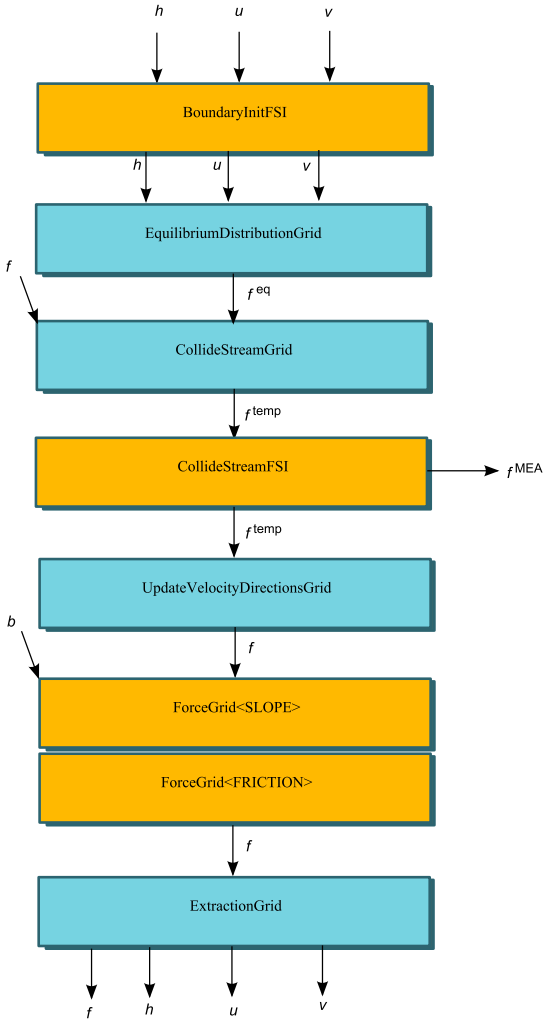


Abbildung 5.4: Datenfluss im erweiterten Algorithmus

5.3.2 Repräsentation von Nachbarschaften zwischen *lattice*-Zellen

Die Speicherzugriffsmuster der Lattice-Boltzmann-Operationen im Allgemeinen und der FSI-Operationen im Speziellen sind so beschaffen, dass für eine Zelle des *lattice* deren Nachbarn bezüglich der *lattice-velocities* bekannt sein müssen. Dies wird durch acht Indexvektoren realisiert, die zusätzlich zu den Daten für die jeweilige Operation (einmalig) zur GPU transferiert werden. So enthält beispielsweise der Indexvektor für die Richtung eins den rechten Nachbarn der dem aktuellen Index entsprechenden Zelle. Nicht zulässige Zugriffe werden mit einer Konstanten kodiert, die größer als die Gesamtzahl der Zellen im *lattice* ist. Diese Vektoren werden durch das Preprocessing bereitgestellt und beziehen sich lediglich auf den invarianten Teil des Rechengebietes. Dies bedeutet, dass für die FSI-Operationen zusätzlich die boole'schen Vektoren für die von der Operation benötigten *flags* auf der GPU zur Verfügung gestellt werden müssen.

5.3.3 FSI-Module

`ForceGrid`

Am Beispiel der Kraftterme lässt sich gut die generelle Vorgehensweise bei der Entwicklung von Operationen für das HONEI CUDA-*backend* demonstrieren, wobei wir uns hier auf die Implementierung des CUDA-*device-code*s konzentrieren und insbesondere von Details bezüglich der Programmierinfrastruktur abstrahieren.

Der erste wesentliche Schritt hierbei ist die Weiterleitung an das CUDA-*backend*, die den `MemoryArbiter` (vergleiche Kapitel 2) anweist, alle benötigten Speichertransfers durchzuführen, wie in Listing 5.1 gezeigt. Dabei wird zunächst der in einer zentralen Konfigurationsdatei eingetragene Wert für die Blockgröße bestimmt, der dem *backend* für diese Operation mitgeteilt werden soll. Dann weden die für die Berechnungen nötigen Daten und Hilfsvektoren *gelockt*. Dies teilt dem `MemoryArbiter` mit, dass die entsprechenden Daten auf der GPU in dem angegebenen Modus (`read_only`, `read_and_write` bzw. `write_only`) verfügbar sein müssen, der diese dann bei Bedarf transferiert, beispielsweise, wenn sie bisher noch nie auf die GPU hochgeladen wurden oder (seltener) zwischendurch von einem anderen *device* verändert wurden. Resultat dieser Locks sind GPU-Speicheradressen, die der Operation zusätzlich zu skalaren Daten und anderen benötigten Informationen wie der Blockgröße übergeben werden. Die Methode `cuda_force_grid_float` ist es dann schließlich, die die einzelnen *device*-seitig auszuführenden CUDA-Methoden entsprechend oft startet, vergleiche Listing 5.2. Hierbei werden zunächst die Größe des *grid* bzw. eines *block* bestimmt, vergleiche hierzu Abschnitt 2.1.3. Die GPU-Adressen werden dann in Pointer des entsprechenden Datentyps umgewandelt. Die eigentlichen (*device*-seitigen) CUDA-Operationen werden dann mit den entsprechenden Größen für *grid* und *block* auf diesen Daten aufgerufen. Um die Betrachtung der Implementierung der Kraftterme abzuschließen, sollen diese Implementierungen im Folgenden beschrieben werden. Da die Imple-

Listing 5.1: Implizite Transfers am Beispiel des Gefällekraftterms

```
void ForceGrid<CUDA, ...>::value(PackedGridData<D2Q9, float> & data, ...)
{
    .
    .
    .
    unsigned long blocksize(Configuration::instance()->get_value("cuda::force_grid_float", 128ul)
        );

    void * cuda_dir_1_gpu(info.cuda_dir_1->lock(lm_read_only, tags::GPU::CUDA::memory_value));
    void * cuda_dir_2_gpu(info.cuda_dir_2->lock(lm_read_only, tags::GPU::CUDA::memory_value));
    .
    .
    .
    .
    void * h_gpu(data.h->lock(lm_read_only, tags::GPU::CUDA::memory_value));
    void * b_gpu(data.b->lock(lm_read_only, tags::GPU::CUDA::memory_value));

    void * f_temp_1_gpu(data.f_temp_1->lock(lm_read_and_write, tags::GPU::CUDA::memory_value));
    void * f_temp_2_gpu(data.f_temp_2->lock(lm_read_and_write, tags::GPU::CUDA::memory_value));
    .
    .
    .

    cuda_force_grid_float(
            cuda_dir_1_gpu, cuda_dir_2_gpu, cuda_dir_3_gpu, cuda_dir_4_gpu,
            cuda_dir_5_gpu, cuda_dir_6_gpu, cuda_dir_7_gpu, cuda_dir_8_gpu,
            h_gpu, b_gpu,
            f_temp_1_gpu, f_temp_2_gpu,
            f_temp_3_gpu, f_temp_4_gpu, f_temp_5_gpu,
            f_temp_6_gpu, f_temp_7_gpu, f_temp_8_gpu,
            (*data.e_x)[1], (*data.e_y)[1],
            (*data.e_x)[2], (*data.e_y)[2],
            (*data.e_x)[3], (*data.e_y)[3],
            (*data.e_x)[4], (*data.e_y)[4],
            (*data.e_x)[5], (*data.e_y)[5],
            (*data.e_x)[6], (*data.e_y)[6],
            (*data.e_x)[7], (*data.e_y)[7],
            (*data.e_x)[8], (*data.e_y)[8],
            g, d_x, d_y, d_t,
            data.h->size(),
            blocksize);

    info.cuda_dir_1->unlock(lm_read_only);
    info.cuda_dir_2->unlock(lm_read_only);
    .
    .
    .

}
```

Listing 5.2: Aufruf des *device*-seitigen CUDA Programms durch das *host*-Programm

```
extern "C" void cuda_force_grid_float(...)
{
    dim3 grid;
    dim3 block;
    block.x = blocksize;
    grid.x = (unsigned)ceil(sqrt(size/(double)block.x));
    grid.y = grid.x;

    unsigned long * dir_1_gpu((unsigned long *)dir_1);
    unsigned long * dir_2_gpu((unsigned long *)dir_2);
    .
    .
    .

    float * h_gpu((float *)h);
    float * b_gpu((float *)b);

    float * f_temp_1_gpu((float *)f_temp_1);
    float * f_temp_2_gpu((float *)f_temp_2);
    .
    .
    .

    honei::cuda::force_grid_gpu_x<<<grid, block>>>(
            dir_1_gpu, f_temp_1_gpu,
            h_gpu, b_gpu,
            e_x_1,
            g, d_x, d_y, d_t,
            size);
    honei::cuda::force_grid_gpu_xy<<<grid, block>>>(
            dir_2_gpu, dir_1_gpu, dir_3_gpu, f_temp_2_gpu,
            h_gpu, b_gpu,
            e_x_2, e_y_2,
            g, d_x, d_y, d_t,
            size);
    .
    .
    .

    CUDA_ERROR();
}
```

Listing 5.3: Krafttermimplementerung am Beispiel des Gefällekraftterms: einfacher Fall

```
__global__ void force_grid_gpu_x(
        unsigned long * dir,
        float * f_temp,
        float * h, float * b,
        float e_x,
        float g, float d_x, float d_y, float d_t,
        unsigned long size)
{
    unsigned long idx = (blockDim.y * blockIdx.y * gridDim.x * blockDim.x) + (blockDim.x
        * blockIdx.x) + threadIdx.x;
    .
    .
    .
    if (idx < size)
    {
        unsigned long i(idx);
        if (dir[i] < size)
        {
            f_temp[i] += force_times_gravity * e_x * (h[i] + h[dir[i]]) * (b[dir[i]] - b[
                i]) / dx_by_two;
        }
    }
}
```

mentierung der Operationen stets nach dem oben beschriebenen Muster verläuft, werden wir uns bei den anderen Operationen auf die Implementierung des *device*-seitigen CUDA-Codes beschränken.

Während der Reibungskraftterm nur elementweise Zugriffe auf die makroskopischen Größen erfordert, ist die Implementierung des Gefällekraftterms dadurch schwieriger, als dass abhängig von α bis zu drei verschiedene Nachbarn in die Berechnung mit einfließen. Beispielsweise wird für $\alpha = 2$ auch in Richtung $\mathbf{e}_2$ interpoliert und die Richtungsableitungen werden anhand von $\mathbf{e}_1$ bzw. $\mathbf{e}_3$ berechnet. Für ungerade α kann die Berechnung entweder für die x-Richtung oder für die y-Richtung entfallen. Daher ergeben sich zwei CUDA-Methoden, je eine für die Richtungen mit geradem α und einen vereinfachten, der $e_{\alpha x} = 0$ bzw. $e_{\alpha y} = 0$ entsprechend berücksichtigt. So kann das CUDA-*device*-Programm für die Richtungen, bei denen lediglich die x-Richtung relevant ist, beispielsweise wie in Listing 5.3 dargestellt implementiert werden. Dabei werden zunächst die Thread-Nummer wie in Abschnitt 2.2.3 beschrieben und die für Gleichung (4.17) nötigen Konstanten berechnet, inklusive der D2Q9-spezifischen Gewichtung. Die Interpolation und die Richtungsableitung erfolgt dann mit Hilfe der Thread-Nummer als Index für die transferierten Daten für h und b. Für die Interpolationsrichtungen mit geradem α muss dies sowohl für die x- als auch die y-Richtung erfolgen - die Ergebnisse werden dann kumuliert.

Für die Berechnung des Reibungskraftterms, d.h. genauer die Wurzelterme (in Gleichung (4.2)) werden in dieser Arbeit die von CUDA zur Verfügung gestellten *built-ins* für mathematische Standardberechnungen, `sqrtf` und `powf` , verwendet.

Listing 5.4: *Streaming*-Korrektur und Impulsaustausch

```
__global__ void collide_stream_fsi_1_gpu(...)
{
    .
    .
    .
        unsigned long i(idx);
        if(f_temp_i[i] != 0. && line_flags[i])
        {
            f_temp_m_i[dir[i]] = f_temp_i[dir[i]] + 2./3. * (d_xu * e_x[1] + d_yv * e_y
                [1]);
            f_mea_m_i[i] = (e_x[5] + e_y[5]) * (f_temp_m_i[dir[i]] + f_temp_i[i]);
            f_temp_i[i] = 0.;
        }
    }
}
```

`CollideStreamFSI`

Bei der Berechnung der Terme für die BFL-Regel, sowie der Berechnung des Impulsaustausches bei gleichzeitiger Korrektur des Transportschritts, werden sowohl f_{α}^{temp} und $f_{-\alpha}^{\text{temp}}$ als auch $f_{-\alpha}^{\text{MEA}}$ benötigt. Zusätzlich muss das CUDA-Programm auf die Indexvektoren für beide Richtungen zugreifen können. Für die *Streaming*-Korrektur ist letztlich der *flag*-Vektor für Z_{S} nötig. Daher ist eine CUDA-Methode bei dieser Operation für die Bearbeitung einer Richtung α zuständig, wie Listing 5.4 beispielhaft für $\alpha = 1$ zeigt.

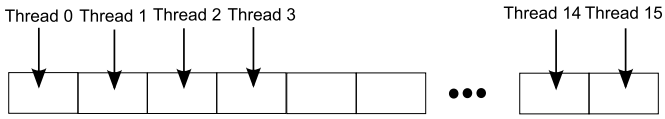
`BoundaryInitFSI`

Wie in Abschnitt 4.6.2 beschrieben, wird die Extrapolationsrichtung bei der Fluid-Initialisierung durch die Bewegungsrichtung eines Festkörpers festgelegt. Man benötigt allerdings nur eine Unterscheidung diesbezüglich für den Aufruf der Operation, die dann jeweils denselben CUDA-*device*-Code auf verschiedenen Daten verwendet.

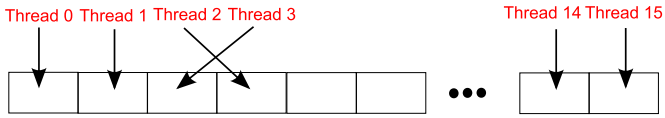
Die *flag*-Vektoren für Z_{S} und $\overline{Z_{\text{S}}}$ werden wie oben beschrieben für die Berechnung von Z_{STF} verwendet. Dabei wird unmittelbar, wenn die entsprechende Information nicht mehr benötigt wird, die *flags* für $\overline{Z_{\text{S}}}$ mit dem aktuellen Wert überschrieben, so dass sie für die nächste Iteration zur Verfügung stehen. Durch die Berechnung der Extrapolationen und insbesondere die Gültigkeitsberechnungen für die entsprechenden Zugriffe, sind die *device-seitigen* Implementierungen für diese Operation vergleichsweise aufwändig. Listing 5.5 zeigt den CUDA-Code, dem der Hauptteil der Arbeit zufällt. Die Zugehörigkeit zu den Zellmengen, sowie die Gültigkeit der Zugriffe auf Nachbarzellen für die Interpolation werden anhand der Daten für $\overline{Z}_{\text{S}}$ und Z_{S} bzw. der Indexvektoren für die Extrapolationsrichtung (hier `dir_mb`) anhand von boole'schen Ausdrücken berechnet. Zugriffe finden so stets auf den tatsächlich benötigten Wert statt, oder auf eine Zelle, die in der Extrapolationsrichtung vor dem benötigten Wert liegt. Die so berechneten Indizes werden dann für die Extrapolationen verwendet.

Listing 5.5: Initialisierung von Fluidzellen

```
__global__ void boundary_init_fsi_i_set_gpu(
        unsigned long * dir_mb,
        bool * boundary_flags,
        bool * solid_flags,
        bool * solid_old_flags,
        float * h,
        float * u,
        float * v,
        float c_u,
        float c_v,
        unsigned long size)
{
    unsigned long idx = (blockDim.y * blockIdx.y * gridDim.x * blockDim.x) + (blockDim.x
        * blockIdx.x) + threadIdx.x;
    if (idx < size)
    {
        unsigned long i(idx);

        bool stf(solid_old_flags[i] & !solid_flags[i]);
        bool fts(!solid_old_flags[i] & solid_flags[i]);

        bool prev((dir_mb[i] < size) ? (!solid_flags[dir_mb[i]] ? true : false) : false);
        unsigned long prev_index(prev ? dir_mb[i] : i);

        bool pre_prev(prev ? ((dir_mb[prev_index] < size) ?
                    (!solid_flags[dir_mb[prev_index]] ? true : false) : false) : false);
        unsigned long pre_prev_index(pre_prev ? dir_mb[prev_index] : prev_index);

        bool pre_pre_prev(pre_prev ? (dir_mb[pre_prev_index] < size ?
                    (!solid_flags[dir_mb[pre_prev_index]] ? true : false) : false) :
                        false);
        unsigned long pre_pre_prev_index(pre_pre_prev ? dir_mb[pre_prev_index] :
             pre_prev_index);

        h[i] = stf ? (float(3.) * (h[prev_index] - h[pre_prev_index]) + h[
             pre_pre_prev_index]) : (fts ? float(0) : h[i]);

        u[i] = (boundary_flags[i] & !stf) ? c_u  :
            (stf ? (float(8./15.) * c_u) + (float(2./3.) * u[prev_index]) - (float(2./5.)
                 * u[pre_prev_index]) :(fts ? float(0) : u[i]));

        v[i] = (boundary_flags[i] & !stf) ? c_v  :
            (stf ? (float(8./15.) * c_v) + (float(2./3.) * v[prev_index]) - (float(2./5.)
                 * v[pre_prev_index]) :(fts ? float(0) : v[i]));

    }
}
```



(a) Reguläres Speicherzugriffsmuster: *coalesced access*



(b) *Uncoalesced access*

Abbildung 5.5: Mögliche Speicherzugriffsmuster

5.3.4 Speicherzugriffsmuster

Die Speicherbandbreite von GPUs kann dann am besten ausgenutzt werden, wenn die benötigten Speichertransfers der Threads eines *half warp* (vergleiche Abschnitt 2.1.3) durch den Speicherkontroller der Grafikkarte in einem gemeinsamen Transfer kombiniert werden können. Dies wird *coalescing* genannt und wird bei den meisten GPUs[2] dadurch bedingt, dass das Zugriffsmuster aller Threads gleich ist. Während diese Bedingung für die Basisoperationen der LBM erfüllt sind, insbesondere aufgrund der Tatsache, dass die Berechnungen stets für jede Richtung separat durchgeführt werden, so dass sich die Zugriffsmuster der Threads einer Operation selten unterscheiden, ist dies bei den hier vorgestellten Erweiterungen nicht der Fall. Ein einfaches Beispiel ist die Extrapolation, die auf einer 3-Punkte Rückwärtsapproximation beruht: Hier hängt der Zugriff des i-ten Threads auf die Daten für h beispielsweise maßgeblich davon ab, ob sich in der entsprechenden Richtung ein Hindernis befindet oder nicht. Dadurch kann es also passieren, dass der i-te Thread auf eine andere Stelle zugreift, als die anderen Threads des *half warp*. Abbildung 5.5(b) zeigt diesen Fall. Obwohl mit einer Festlegung auf feste Extrapolationsrichtungen dem bereits entgegengewirkt wird, ist dies ein bestehendes Problem für die Operationen im Blickfeld dieser Arbeit. Die Speicherkontroller aktuellerer GPUs können allerdings auch solche irregulären Muster entsprechend behandeln.

[2] Beispielsweise die GeForce 8800, vergleiche für Details zum *Coalescing* das CUDA Programming Guide [27]

6 Ergebnisse

6.1 Vorgehensweise

Im Folgenden sollen die Implementierungen der im Rahmen dieser Arbeit vorgestellten und erweiterten Algorithmen zunächst numerisch validiert werden. Dazu werden in den Abschnitten 6.2 - 6.5 verschiedene Aspekte der Fluidsimulation analysiert und die Ergebnisse mit analytischen Lösungen bzw. numerischen Referenzlösungen anderer Bibliotheken verglichen. Die Algorithmen (bzw. Löser und FSI-Module) werden dann in Abschnitt 6.6 auf ihre Leistungsfähigkeit hin untersucht. Dabei ist die zentrale Fragestellung, wie hoch die Auflösung des *lattice* sein kann, so dass die Simulation noch hinreichend schnell ist, d.h. mit einer *interaktiven* Rate Lösungen für die Zeitschritte erzeugen kann. Schließlich werden die Algorithmen in Abschnitt 6.7 auf ihre Einsetzbarkeit in 3D-Echtzeitumgebungen hin untersucht. Dabei werden möglichst komplexe Szenarien verwendet, um die Vielseitigkeit der in Kapitel 4 vorgestellten und modifizierten Erweiterungen der Lattice-BGK Methode zu demonstrieren und gleichzeitig den Nachweis zu erbringen, dass Echtzeit-Strömungssimulation in interaktiven 3D-Szenen auch ohne dreidimensionale Berechnungen brauchbare visuelle Ergebnisse erzielen kann.

6.1.1 Testhardware

Wir konzentrieren uns im Folgenden auf die Zielhardware GPU. Dabei werden zwei verschiedene Grafikkarten verwendet: Zum einen eine NVIDIA GeForce GTX 285, eine Grafikkarte der aktuellen Generation, und eine etwas ältere GeForce GTX 8800. Dadurch kann insbesondere gemessen werden, inwieweit die Entwicklung der GPUs bei gleichbleibendem Applikationscode die Leistungsfähigkeit desselben steigert. Gerade für die in dieser Arbeit implementierten Erweiterungsmodule ist die Verbesserung der GPU-Speicherkontroller von einer Hardwaregeneration zur nächsten bedeutsam, da es sich als sehr schwierig erwiesen hat, die einzelenen Operationen hinsichtlich der Speicherzugriffsmuster zu optimieren, vergleiche Abschnitt 5.3.4. Als CPU-Referenzsystem für die numerischen und *Performance*-Benchmarks wird ein Intel Core i7 920 -System verwendet. Die technischen Kenndaten Speicherbandbreite, Kerntakt, Arbeits- (bzw. *device*)-Speicher, sowie die Anzahl der Rechenkerne (*cores*) für die drei Evaluationsplatformen sind in Tabelle 6.1 zusammengefasst. Die Entwicklung der GPUs zeigt sich, wie diese Tabelle zeigt, in einer etwaigen Verdoppelung von Multiprozessoren und Speicherbandbreite bei etwa gleichbleibendem Kerntakt.

	GTX 8800	GTX 285	Core i7
# Rechenkerne	16	30	4
Kerntakt (GHz)	1.3	1.5	2.66
Arbeitsspeicher (GB)	0.768	2	12
Speicherbandbreite (GB/s)	86	159	38

Tabelle 6.1: Ausgewählte technische Daten der Evaluationshardware

Die Speicherbandbreite der GTX 8800 ist dabei bereits mehr als doppelt so groß, wie bei dem modernen Intel-Vierkernsystem.

6.1.2 Bemerkung zur Genauigkeit und Regressionstests

Die Korrektheit einzelner Operationen wird in HONEI und damit insbesondere auch bei den hier implementierten Operationen durch eine Verbindung von Vergleichstest mit analytischen Ergebnissen (sogenannte *unittests*) und Vergleichen der numerischen Ergebnisse aller *backend*-Instanzen einer Operation mit dem CPU-*backend* (*regressiontests*) realisiert. Damit ist sichergestellt, dass die hardwarespezifischen Implementierungen bis auf eine vom entsprechenden Gleitkommasystem abhängige Toleranzgrenze numerisch gleiche Ergebnisse liefern. Die in den Folgenden Abschnitten dargestellten numerischen Ergebnisse sind daher stets mit der GTX 285 erzeugt, falls nicht explizit etwas anderes angegeben wird, wobei gewährleistet ist, dass diese bis auf sehr kleine Abweichungen mit der CPU-Referenzlösung übereinstimmen.

Alle Ergebnisse werden mit einfacher Genauigkeit berechnet, da im Blickfeld dieser Arbeit die numerische Genauigkeit in der Bedeutung hinter den visuellen Ergebnissen und der Leistung steht. Eine Ausnahme dazu sind die numerischen Tests in Abschnitt 6.4, bei denen ein Vergleich mit externen Referenzergebnissen eine Berechnung mit doppelter Genauigkeit auf der CPU erfordert.

6.2 Kontinuität

6.2.1 Vorgehensweise

Eine wesentliche Eigenschaft, die ein numerisches Verfahren zur Strömungssimulation haben muss, ist die Kontinuität der makroskopischen Masse (Masseerhaltung). Der erste numerische Test für die in dieser Arbeit implementierten Algorithmen ist daher die Untersuchung der Lösungen auf die Größenordnung des Fehlers nach komplexen Simulationen bezüglich der analytischen Masse des Fluids und das Verhalten des Fehlers bei steigender Auflösung des zu lösenden Problems. Durch die Inkompressibilität der hier betrachteten Fluide ist die Dichte konstant, weshalb hier die makroskopische Masse äquivalent ist mit

dem *Volumen* des Fluids über dem Rechengebiet. Daher kann man bei diesem Test so vorgehen, dass man zunächst hinreichend komplizierte Test-Szenarien definiert, so dass das analytische Volumen a priori bekannt ist. Diese werden dann bis zu einem *Steady-State* gelöst (d.h., der Löser entsprechend eines vorgegebenen Konvergenzkriteriums iteriert).

6.2.2 Analytisches Volumen, numerische Integration, Volumenfehler und Konvergenz

Im nicht-diskreten Fall ist bei den Flachwassergleichungen das (initiale) Volumen über dem Rechengebiet gegeben durch

$$V_{\text{ana}} = \int_0^{\overline{x}} (\int_0^{\overline{y}} h(x,y) - b(x,y)dy)dx \tag{6.1}$$

wobei $\overline{x}$ bzw. $\overline{y}$ die Dimensionen des rechteckigen Rechengebietes sind. Für kompliziertere Rechengebiete sind zusätzlich die Festkörpergebiete sowohl bei der Berechnung des analytischen Volumens, als auch von einem numerischen Verfahren entsprechend zu berücksichtigen. Wir verwenden für diesen Test das ebenfalls von Becker [4] benutzte und in der HONEI-*math* Bibliothek implementierten Verfahren zur numerischen Integration, dem Gauss-Quadraturen mit bilinearer Interpolation zugrunde liegen. Da alle Lattice-Boltzmann basierten Löser Festkörper in den Skalafeldern für die makroskopischen Quantitäten durch Null-Werte repräsentieren, werden sie bei der numerischen Volumenberechnung automatisch berücksichtigt. Der relative Volumenfehler ist definiert als:

$$E_{\text{vol}} = \left| \frac{V_{\text{ana}} - V_{\text{res}}}{V_{\text{ana}}} \right| \tag{6.2}$$

Dabei ist V_{res} das Volumen der diskreten Lösung im *Steady State*, d.h., wenn keine Veränderung der makroskopischen Größen mehr zu verzeichnen ist. Wir messen dies mit Hilfe der Differenzen zwischen den makroskopischen Geschwindigkeiten im aktuellen und im letzten Zeitschritt, d.h. das Verfahren terminiert, wenn

$$||u_i(\mathbf{x}, t - \Delta t) - u_i(\mathbf{x}, t)|| < \epsilon, \text{ für } i \in \{x, y\} \tag{6.3}$$

gilt. Da die Daten für $\mathbf{u}$ in separaten Vektoren gespeichert sind, ist $||u(x,y)|| : \mathbb{R}^{NM} \rightarrow \mathbb{R}$ eine Vektornorm und die Schranke ϵ ist dem Gleitkommasystem entsprechend auf die kleinste messbare Differenz zweier Zahlen gesetzt und bewegt sich bei einfacher Genauigkeit in einer Größenordnung von 1E-8.

6.2.3 Testszenarien

Die im Folgenden beschriebenen Löserkonfigurationen und Anfangsbedingungen sind im Hinblick auf das Gesamtspektrum der Simulationen gewählt, das mit den zu testenden Erweiterungen der LBM möglich ist. Dabei wird jedoch prinzipiell für jedes Szenario der

vollständige Algorithmus (Algorithmus 4.4) angesetzt, da demonstriert werden soll, dass es keine Seiteneffekte durch die Erweiterungsmodule gibt, d.h., dass auch einfache Szenarien wie mit dem Basisalgorithmus 3.1 ohne Verlust der Korrektheit berechnet werden können. Die Parametrisierung des Lösers ist dabei zur besseren Vergleichbarkeit bei allen Szenarien gleich: Das quadratische Rechengebiet hat eine Kantenlänge von $50m$ und wird schrittweise in Abhängigkeit von der Diskretisierungsstufe d so aufgelöst, dass $\Delta x = \Delta y = \frac{1}{2^{d-1}}$ und $\Delta t = \tau = 1$ für $d \in \{1, ..., 6\}$ ist. D.h., dass die Anzahl der *lattice*-Zellen in Abhängigkeit von d $(50 \cdot 2^{d-1})^2$ beträgt, bis zu einem Maximum von 1600^2. Es werden stets die in Kapitel 3 beschriebenen *No-Slip*-Randbedingungen verwendet.

Die Anfangsbedingungen für die einzelnen Szenarien werden im Folgenden beschrieben.

Volldammbruch

Ausgangspunkt für die einfachste Simulation ist eine initiale konstante Wasserhöhe von $h_0 = 5m$, wobei wie in Abbildung 3.1(a) dargestellt ein quaderförmiges Wasservolumen mit den Abmessungen $5m \cdot 5m \cdot 3m$ kollabiert (ein sogenannter quaderförmiger Volldammbruch, *full cuboidal dam break*, FCuDB). Die Bodentopographie ist dabei eben und konstant Null.

Teildammbruch

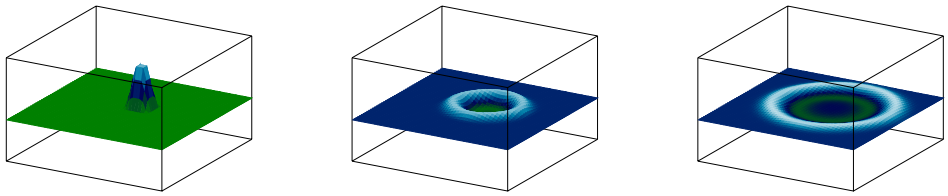
Wir erweitern das FCuDB-Szenario durch Erweiterung der Szenengeometrie mit zwei $5m$ breiten und $20m$ langen Wänden, die einen teilweise gebrochenen Damm und für die durch ein höher gelegenes Wasserreservoir induzierte Strömung durch den $10m$ breiten Durchlass Hindernisse in Form von stationären Festkörpern darstellen (PCuDB).

Volldammbruch mit unebenem Bodenprofil

Um die Masseerhaltung und Stabilität bei einem Bodenprofil mit Gefälle zu testen, wird die Volldammbruchsimulation mit einer durch die Funktion $b_\gamma(x, y) = \gamma(x^2 + y^2)$ gegebenen Topographie versehen und für subkritische Werte für γ, d.h. $b_\gamma(x, y) < 5$ evaluiert ($\text{FCuDB}_{\text{slope}}$). Die numerischen Eigenschaften der Krafttermimplementierungen werden in Abschnitt 6.3 genauer behandelt.

Kritischer Teildammbruch, *Dry-States*

Modifikation des Teildammbruchszenarios PCuDB derart, dass die Wasserhöhe außerhalb des Reservoirs Null beträgt, ergibt eine Situation, in der die Strömung am Durchlass steil auf trockenen Untergrund trifft ($\text{PCuDB}_{\text{dry}}$). Dazu wird der Löser so konfiguriert, dass Zellen mit einer Wassertiefe von weniger als 1E-02m als trocken definiert werden, vergleiche Abschnitt 4.3.2. Durch dieses Abschneiden ist generell ein durchscnittlich höherer Masseverlust zu erwarten, als bei den anderen Konfigurationen, vergleiche hierzu auch Abschnitt 6.3.



(a) Nach dem Preprocessing (b) Nach 10 Zeitschritten (c) Nach 20 Zeitschritten

Abbildung 6.1: Prinzip und ausgewählte Zeitschritte der Volldammbruchsimulation (FCuDB)

Festkörper mit eigenem Antrieb I

Die Masseerhaltung bei Reaktion des Fluids auf einen instationären Festkörper wird mit Hilfe der zweidimensionalen Repräsentation eines Würfels mit Kantenlänge $5m$ durchgeführt, der sich mit konstanter Geschwindigkeit $30sec$ durch das Fluid bewegt und dann anhält (BoxFSI).

Festkörper mit eigenem Antrieb II

Schließlich werden in einer letzten Simulation mehrere Effekte kombiniert: Wir erweitern das Szenario BoxFSI zu $\text{BoxFSI}_{\text{adv}}$, indem wir

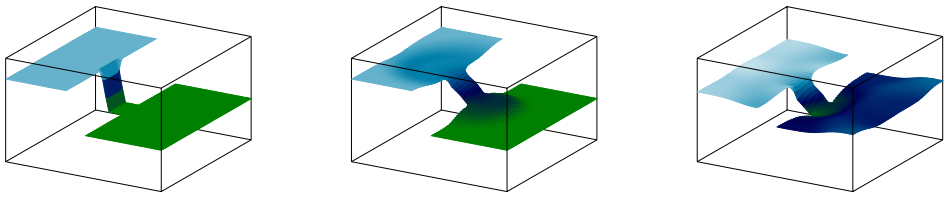
1. zwei stationäre Hindernisse hinzufügen,
2. einen Volldammbruch erzeugen, durch den sich der Festkörper bewegt und
3. die Bodentopographie wie in $\text{FCuDB}_{\text{slope}}$ definieren.

Prinzip und Zwischenergebnisse der Simulationen sind in den Abbildungen 6.1 - 6.6 in Form von 3D-Ansichten bzw. Höhenkarten für $h + b$ dargestellt.

6.2.4 Ergebnisse

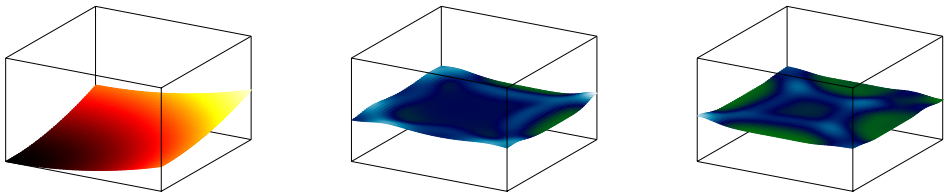
Die Werte für den Volumenfehler für die unterschiedlichen Szenarien bei den verschiedenen Diskretisierungsstufen sind in Tabelle 6.2 zusammengefasst.

Dabei wird zunächst deutlich, dass sich der relative Fehler bei fast allen Szenarien auch bei niedrigster Auflösung innerhalb vertretbarer Grenzen von 0.002 Prozent bis 2 Prozent bewegt, die visuell praktisch nicht wahrnehmbar sind. Eine Ausnahme bildet erwartungsgemäß das *Dry-States*-Szenario, das bei niedriger Auflösung des *lattice* von 50^2 Zellen einen nicht-tolerierbaren Fehler von 10% des analytischen Gesamtvolumens aufweist, was sich auch visuell durch eine nicht glatte Fluidoberfläche zeigt, vergleiche Abbildung 6.4 und was auf die eingesetzte Technik des Abschneidens von Höhenwerten zurückzuführen ist.



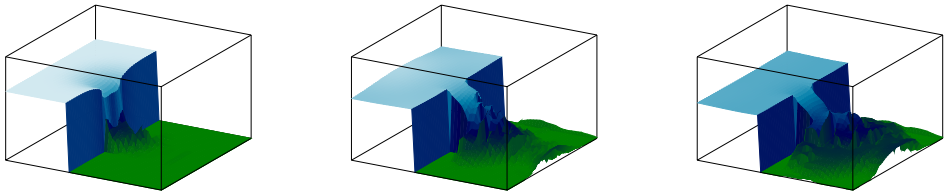
(a) Nach dem Preprocessing (b) Nach 20 Zeitschritten (c) Nach 60 Zeitschritten

Abbildung 6.2: Prinzip und ausgewählte Zeitschritte der Teildammbruchsimulation (PCuDB)



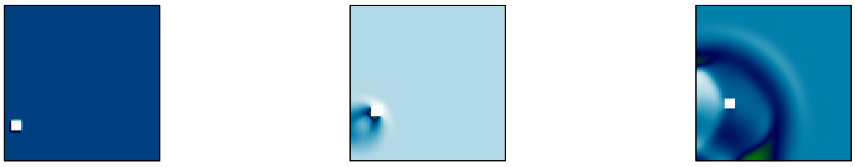
(a) Bodenprofil: $f_\gamma(x, y) = \gamma(x^2 + y^2)$ (b) Nach 40 Zeitschritten (c) Nach 50 Zeitschritten

Abbildung 6.3: Prinzip und ausgewählte Zeitschritte der Volldammbruchsimulation mit unebenem Untergrund ($\text{FCuDB}_{\text{slope}}$)



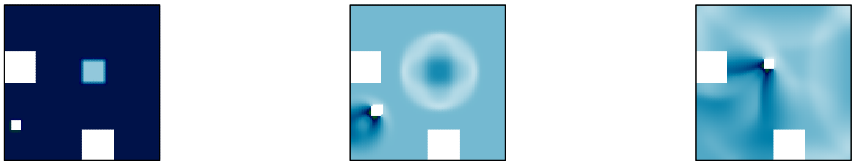
(a) Nach 10 Zeitschritten (b) Nach 40 Zeitschritten (c) Nach 50 Zeitschritten

Abbildung 6.4: Prinzip und ausgewählte Zeitschritte der Teildammbruchsimulation mit *Dry-States* ($\text{PCuDB}_{\text{dry}}$)



(a) Nach dem Preprocessing (b) Nach 20 Zeitschritten (c) Nach 80 Zeitschritten

Abbildung 6.5: Prinzip und ausgewählte Zeitschritte der Fluidreaktions-Simulation (BoxFSI)



(a) Nach dem Preprocessing (b) Nach 20 Zeitschritten (c) Nach 80 Zeitschritten

Abbildung 6.6: Prinzip und ausgewählte Zeitschritte der erweiterten Fluidreaktions-Simulation ($\text{BoxFSI}_{\text{adv}}$)

d	FCuDB	PCuDB	$\text{FCuDB}_{\text{slope}}$	$\text{PCuDB}_{\text{dry}}$	BoxFSI	$\text{BoxFSI}_{\text{adv}}$
1	1.83E-04	9.33E-04	2.75E-02	1.11E-01	1.74E-02	1.66E-02
2	6.37E-05	5.56E-04	2.74E-02	5.05E-02	3.41E-03	1.94E-02
3	7.96E-06	4.45E-06	2.74E-02	5.70E-03	9.98E-04	1.50E-02
4	1.11E-06	1.48E-06	1.01E-02	5.01E-03	2.58E-06	3.00E-03
5	7.97E-07	7.40E-06	8.96E-03	5.00E-03	4.00E-07	2.60E-03
6	7.96E-07	0.96 E-06	4.70E-04	4.98E-03	2.01E-07	1.01E-04

Tabelle 6.2: Relativer Fehler E_{vol} des Volumens für die verschiedenen Testszenarien bei steigender Diskretisierungsstufe

Diese ist jedoch notwendig, um die Stabilität des Algorithmus zu gewährleisten, wie in Abschnitt 6.3 gezeigt wird, da ansonsten zu viele Oszillationen resultieren würden, die auch mit dem verwendeten Limiter-Ansatz zu einem unbrauchbaren Ergebnis führen.

Bei allen Szenarien sinkt der Volumenfehler mit steigender Auflösung soweit, dass bei einfachen Szenarien ein zu vernachlässigender Fehler von 0.00008% bis 0.005% des Volumens resultiert. Der Fehler bei dem Szenario $\mathrm{PCuDB_{dry}}$ kann durch eine höhere Diskretisierungsstufe auf 0.5% gesenkt werden. Wiederum wie erwartet steigt der Fehler und sinkt die Rate der Fehlerreduktion mit der Komplexität der Simulation: Insbesondere wird durch innere Ränder das Ergebnis bereits (wenn auch nur schwach) beeinflusst. Bei Simulation von instationären Festkörpern entsteht kein nachweisbarer Verlust der Genauigkeit: Man betrachte dazu die Daten der BoxFSI-Simulation. Obwohl bei niedriger Auflösung durchaus Unterschiede zu einfachen Szenarien feststellbar sind, ist die Genauigkeit bei mittlerer und höherer Auflösung mindestens gleich hoch, obwohl durch die zusätzliche Rechenintensität und insbesondere Extrapolationsfehler eine höhere Abweichung zu erwarten wäre.

6.3 Kraftterme, *Dry-States*

6.3.1 Vorgehensweise

Um die Korrektheit der Krafttermimplementierungen zu demonstrieren, werden im Folgenden zwei Tests verwendet. Ein wesentlicher Funktionstest für die in dieser Arbeit verwendete Kombination von Krafttermen assoziiert mit der Bodentopographie ist es, diese auf die Fähigkeit hin zu untersuchen, physikalisch korrekte Ergebnisse bei einem unebenen Bodenprofil zu erzeugen. Genauer bedeutet dies sicherzustellen, dass das numerische Schema der Kraftterme hinreichend genau ist, um Unebenheiten entsprechend auszugleichen, so dass die absolute Wassertiefe $h + b$ bzw. die Wasseroberfläche im *Steady-State* keine Anomalien aufweist. Dazu wird ein Szenario verwendet, bei dem das Bodenprofil anschaulich durch einen steilen „Hügel“ dominiert wird, der vollständig unterhalb der Wasseroberfläche liegt. Es wird dann ein Volldammbruch sowohl mit (Algorithmus 4.1) als auch ohne Kraftterme (Algorithmus 3.1) simuliert und die Ergebnisse im *Steady-State* miteinander und mit der analytischen Referenzlösung verglichen. Dieser Test wurde auch von Zhou für seine Implementierung des semi-impliziten Schemas für den Gefälle-Kraftterm durchgeführt [42]. Er konnte zeigen, dass dabei andere numerische Schemata (wie beispielsweise das Basisschema in Abschnitt 4.2.3) nicht in der Lage waren, das korrekte Ergebnis zu berechnen.

Dies wird von Zhou allerdings lediglich für subkritische Szenarien getestet, d.h. für Wassertiefen deutlich größer als Null. Der zweite Test zielt daher auf die Korrektheit der Kraftterme in Kombination mit *Dry-States* ab. Dazu wird ein kritisches Szenario verwendet, bei dem das Bodenprofil (s.o) aus der Wasseroberfläche ragt. Eine Strömung, die auf die so entstehende Erhebung trifft, muss sich dann so verhalten, wie in Abbildung 3.1(c) in Kapitel

3 dargestellt. Dabei ist zu untersuchen, wie sich die in dieser Arbeit vorgestellte experimentelle Kombination von Krafttermen, dem Einsatz von Limitern und der Definition von *Dry-States* über minimale Wassertiefen und dem damit verbundenen lokalen Masseverlust verhalten.

6.3.2 Testszenarien

Als Bodentopographie für beide oben beschriebenen Tests wird die Funktion

$$b_\gamma(x, y) = \gamma \cdot exp(-5(x-1)^2 - 50(y-0.5)^2) \tag{6.4}$$

verwendet mit $\gamma = 4cm$ für den subkritischen Fall und $\gamma = 8cm$ für den Test mit *Dry-States*. Die initiale Wassertiefe beträgt in beiden Fällen $h(\mathbf{x}, 0) + b(\mathbf{x}) = 5cm$ und das *lattice* hat eine Auflösung von 200×100 *lattice*-Zellen, bei $\Delta x = 1cm$ und $\Delta t = 0.01sec$, die Relaxationszeit beträgt $\tau = 1.1$, d.h wir diskretisieren ein $2m \cdot 1m$ großes Rechengebiet. Die Volldammbrüche werden mit verschiedenen Intensitäten (d.h. mit verschieden großen Volumina) durchgeführt. Damit ist die analytische Wasserhöhe im *Steady-State* stets durch die initiale Wasserhöhe zuzüglich zu einem durch das Dammbruchvolumen induzierten Wert gegeben.

Für beide Tests wird das im letzten Abschnitt definierte Konvergenzkriterium verwendet.

6.3.3 Ergebnisse

Das resultierenden absoluten Wassertiefen für die subkritische Löserkonfiguration mit und ohne Kraftterm sowie das Bodenprofil sind in Abbildung 6.7 dargestellt. Dabei handelt es sich um die entsprechenden Daten entlang der Mittellinie parallel zur x-Achse, so dass die Höhenfelder über der durch Gleichung (6.4) beschriebenen symmetrischen Funktion etwa mittig abgetastet werden. Dies verdeutlicht den Unterschied zwischen Flachwassersimulationen ohne und mit Kraftterm und zeigt insbesondere die Korrektheit der hier vorgestellten Modifikationen und der Implementierung auf der GPU. Diese Ergebnisse decken sich mit denen von Zhou, wobei im Rahmen der hier vorliegenden Arbeit verifiziert werden konnte, dass bei geeigneter Parametrisierung des Algorithmus auch wesentlich komplexere Szenarien mit unebenem Bodenprofil visuell und numerisch korrekte Ergebnisse liefern. Diese umfassen das gesamte Spektrum der hier vorgestellten Szenarien und schließen insbesondere auch bewegliche Festkörper mit ein, siehe auch Abschnitt 6.7.

Im Fall der Simulation mit *Dry-States* kann verifiziert werden, dass der Löser stabilisiert werden kann und dann auch ansprechende visuelle Ergebnisse liefert. Dazu zeigt Abbildung 6.8 die absolute Wassertiefe $h + b$ für verschiedene Einstellungen bezüglich der minimalen von Null verschiedenen Wassertiefe (im Folgenden Abschneidelimit genannt), so dass der Löser die entsprechenden Zellen noch als Fluidzellen behandelt. So ist in der Mitte der Abbildung praktisch nur noch das Bodenprofil sichtbar, da die entsprechende Funktion dort

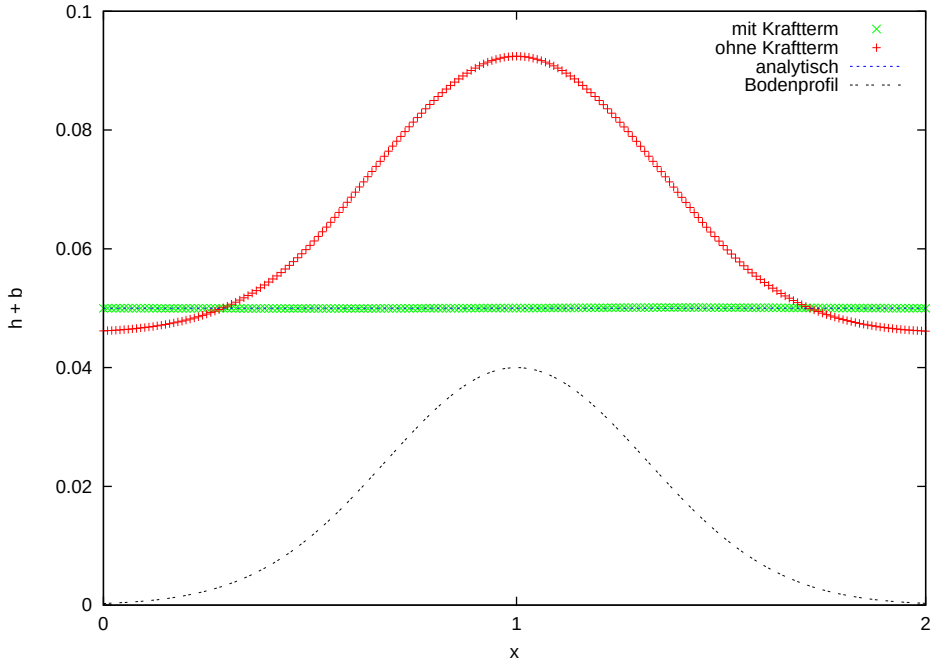


Abbildung 6.7: Zur Korrektheit der Kraftterme: Absolute Wassertiefe $h + b$ mit und ohne Kraftterm bei einem subkritischen Volldammbruch mit unebenem Bodenprofil

größer ist, als die Fluidtiefe ($h = 0 \Rightarrow h + b = b$). Die Grenze des Gebietes mit *Dry-States* zur Flüssigkeit erkennt man an einem abrupten Abfall der absoluten Wassertiefe, die im Fall kleinerer Werte für das Abschneidelimit immer weniger deutlich sichtbar sind. Dieser Wert beeinflusst daher sowohl die Stabilität, als auch das visuelle Ergebnis maßgeblich. Die dargestellten Kurven repräsentieren die Werte für $h+b$ zu einem Zeitpunkt, an dem die sich von rechts nach links bewegende Strömung gerade etwa auf die Insel trifft, so dass zuvor trockene Zellen ihren Zustand ändern. Während bei hohen Werten für das Abschneidelimit eine deutlich sichtbare Grenze zwischen trockenem Gebiet und Fluid entsteht, kommt es bei niedrigeren Werten häufiger zu numerischen Störungen, die durch große Geschwindigkeiten entstehen, in der Abbildung sichtbar durch die Fluktuationen an der linken Flanke des Bodenprofils bei Simulationen mit einem Abschneidelimit von 1E-3 und kleiner. Die Grenze zwischen Fluid- und Nicht-Fluidgebiet ist in der Abbildung zusätzlich durch vertikale Linien markiert, was verdeutlicht, wie das Abschneidelimit den Volumenfehler beeinflusst: Ein kleiner Wert zieht eine geringere Differenz zwischen $h + b$ und b an der Fluidgrenze nach sich, als ein größerer. Es sei an dieser Stelle angemerkt, dass diese Simulationen nur durch den Einsatz des hier vorgestellten adaptiven Limiters stabil sind. Ein deutlich kleinerer Wert als 1E-4 destabilisiert den Löser jedoch unabhängig von Einsatz und Parametrisierung der Limiterfunktion. Dennoch können auf die in dieser Arbeit vorgestellten Weise in vielen Szenarien *Dry-States* mit ansprechendem visuellen Ergebnis simuliert werden,

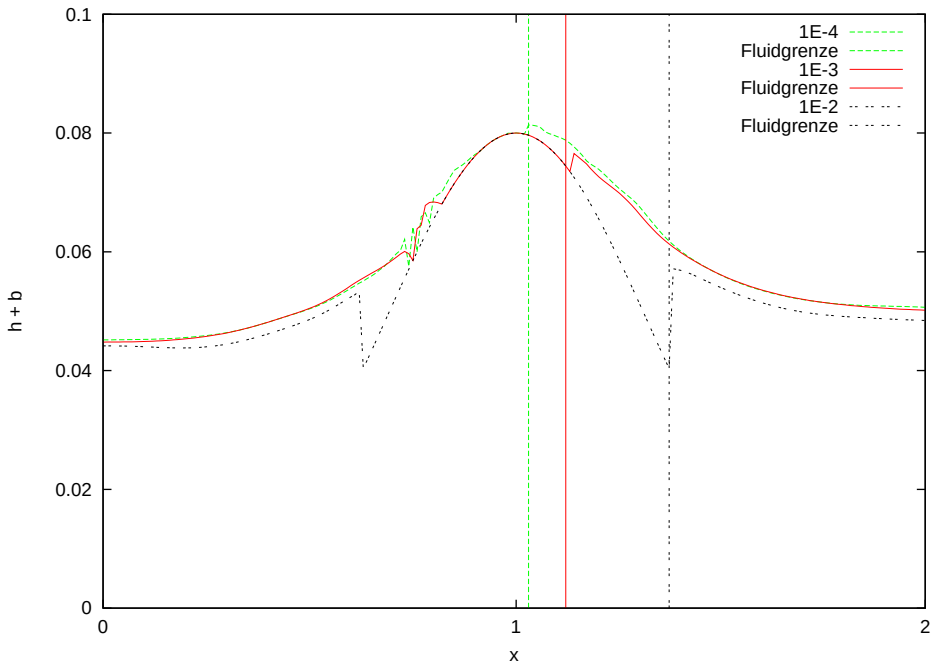


Abbildung 6.8: *Dry-States:* Absolute Wassertiefe bei stabilen Simulationen mit verschiedenen Werten für das Abschneidelimit

vergleiche Abschnitt 6.7. Da die Stabilität jedoch so stark von den globalen Einstellungen abhängig ist, ist die Methode in ihrem gegenwärtigen Entwicklungsstadium nur begrenzt für den interaktiven Bereich einsetzbar.

6.4 *Driven-Cavity* mit mikroskopischen Randbedingungen

6.4.1 Vorgehensweise

Die Korrektheit der modifizierten BFL-Regel für *Moving-Boundaries* (Algorithmus 4.2) kann durch das in Abschnitt 4.5 beschriebene *Lid-Driven-Cavity* Szenario nachgewiesen werden. Anhand dieses Tests weist Ribbrock in seiner Diplomarbeit die Korrektheit des Basisalgorithmus (Algorithmus 3.1) nach [32]. Dabei werden die Lösungen mit numerischen Ergebnissen für die makroskopische Geschwindigkeit von Hübner [19] verglichen. Dazu wird die im Rahmen der hier vorliegenden Arbeit entwickelte Implementierung der Berechnung der Gleichgewichtsfunktionen für die Navier-Stokes Gleichungen verwendet. Ribbrock zeigt mit Hilfe dieses Moduls, dass der Grundalgorithmus als valide gelten kann, da keine signifikanten Unterschiede zu den Referenzlösungen bestehen.

Da der Basislöser keine beweglichen Wände simulieren kann, werden die Dirichlet'schen Randbedingungen lediglich auf makroskopischer Ebene berücksichtigt. Die Idee des folgenden Testverfahrens ist es, die Korrektheit von Algorithmus 4.2 zu zeigen, indem die bewegliche Wand im *Driven-Cavity* Szenario entsprechend repräsentiert wird, vergleiche

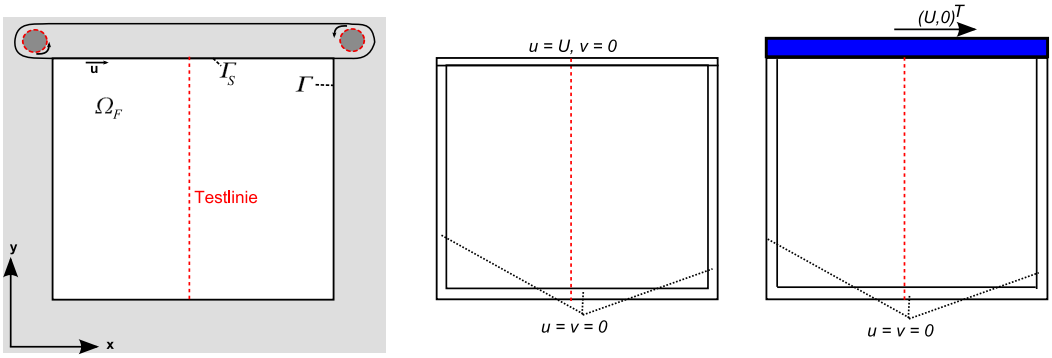


Abbildung 6.9: Prinzip des modifizierten *Driven-Cavity*-Tests: Das Integrationsgebiet (links), wird längs der y-Achse mittig entlang einer Testlinie analysiert. Ohne BFL-Regel werden die Dirichlet'schen Randbedingungen zwischen den Zeitschritten für die makroskopischen Quantitäten gesetzt (Mitte). Mit Algorithmus 4.2 kann man die bewegliche Wand durch einen Festkörper mit der entsprechenden Momentangeschwindigkeit simulieren.

Abbildung 6.9. Dabei wird das Rechengebiet am oberen Rand erweitert und ein stationärer Festkörper mit der Momentangeschwindigkeit $\mathbf{u}_B = (U, 0)^T$ eingesetzt. Die Ergebnisse für verschiedene Parametrisierungen der Löser für die Algorithmen 3.1 und 4.2 werden dann jeweils mit den numerischen Ergebnissen von Hübner verglichen. Bei Korrektheit des Algorithmus für *Moving-Boundaries* müssen die entsprechenden Lösungen bezogen auf die Abweichung zu den Referenzergebnissen mindestens gleichwertig sein. Dadurch, dass die modifizierte BFL-Regel jedoch die Randbedingungen auch auf mikroskopischer Ebene, d.h. im entsprechenden Nichtgleichgewichtsanteil berücksichtigt, ist jedoch eine Verbesserung gegenüber dem Basisalgorithmus zu erwarten.

6.4.2 Dynamisch äquivalente Parametrisierungen des Lösers

Die Vergleichbarkeit von Ergebnissen verschiedener Löser für die Strömungssimulation basiert auf dem Prinzip der dynamischen Äquivalenz von Strömungen. Dabei wird die Ähnlichkeit von Testszenarien (bzw. allgemein die Zustände von Fluidkörpern) etwa durch die Reynolds-Zahl gemessen:

$$R_e = \frac{U \cdot L}{\nu} \tag{6.5}$$

Dabei ist U die charakteristische Geschwindigkeit des Fluids, im Fall des *Driven-Cavity*-Problems also die Geschwindigkeit der den Strudel induzierenden beweglichen Wand, L die charakteristische Länge des Fluidkörpers und ν die bereits in Kapitel 3 eingeführte kinematische Viskosität. Da im Fall der Lattice-Boltzmann Gleichung im LBGK Modell mit einem D2Q9 *lattice* $\nu = \frac{1}{6}\frac{\Delta x^2}{\Delta t}(2\tau - 1)$ ist und wir bei diesem Test im Folgenden stets

Parametrisierung	Δx	Δt	τ
DC_1	1	1	1
DC_2	1	1	1.5
DC_3	1	1	0.6
DC_4	1	1	0.9
DC_5	1	1	1.1
DC_6	1	0.9	1
DC_7	1	1.1	1
DC_8	0.9	1	1

Tabelle 6.3: Löserparametrisierungen für die *Driven-Cavity*-Tests

eine Auflösung von 129×129 verwenden, ist $L = 129 \cdot \Delta x$ und damit die Reynolds-Zahl gegeben als:

$$R_e = \frac{6 \cdot 129 \cdot U}{\frac{\Delta x}{\Delta t}(2\tau - 1)} \tag{6.6}$$

Wir vergleichen die Werte für die makroskopische Geschwindigkeit entlang der vertikalen Mittellinie des Rechengebiets mit denen von Hübner bei einer Reyonolds-Zahl von 100. Dabei wird der Löser wie bisher in diesem Kapitel über die Werte für τ, Δx und Δt parametrisiert. Anhand von Gleichung (6.6) wird dann die charakteristische Geschwindigkeit für das Szenario berechnet, so dass $R_e = 100$. Dabei werden die in Tabelle 6.3 zusammengefassten Einstellungen (DC_i)verwendet und die Güte der Lösung mit

$$E_H = ||u_H - u_{\text{res}}||$$

gemessen, wobei u_H die x-Komponente der makroskopischen Geschwindigleit in der Referenzlösung und u_{res} das entsprechende Ergebnis der Simulation.

6.4.3 Ergebnisse

Der Fehler bzw. die Abweichung zum Referenzergebnis, E_H, wird sowohl für den Basisalgorithmus (GRID) als auch für den erweiterten Algorithmus (FSI) für die jeweiligen Löserparametrisierungen in Tabelle 6.4 dargestellt. Zusätzlich wird für jeden der beiden Löser die Anzahl der jeweils benötigten Iterationen dargestellt. Die Tabelle zeigt zunächst, dass bei allen dargestellten Einstellungen bei diesem Test die Abweichung zu dem als wesentlich genauer als beide Algorithmen geltenden Verfahren von Hübner stets bei dem FSI-Löser kleiner ist, als bei dem Basisverfahren. Dabei bewegt sich die durch die Berücksichtigung der Dirichlet-Randwerte im Nichtgleichgewichtsanteil erwartete Verbesserung gegenüber Algorithmus 3.1 zwischen 20% und 85%. Es hat sich gezeigt, dass sich die optimale Parametrisierung im Bereich von $\tau = \Delta t = \Delta x$ bewegt. In diesem Fall beträgt die Verbesserung

Szenario	Fehler (GRID)	Fehler (FSI)	Fehlerreduktion	k_{grid}	k_{fsi}
DC_1	2.00E-03	1.55E-03	22.5%	26250	26120
DC_2	2.12E00	8.90E-01	58%	17369	17119
DC_3	1.57E00	1.02E-00	35%	74665	64434
DC_4	5.0E-01	8.50E-02	83%	30031	29257
DC_5	2.52E-01	1.08E-01	57%	23654	23580
DC_6	1.95E-03	1.55E-03	20.5%	26700	26347
DC_7	1.95E-03	1.55E-03	20.5%	26025	25894
DC_8	1.95E-03	1.55E-03	20.5%	26024	25669

Tabelle 6.4: Fehler zur numerischen Referenzlösung E_H für den Basisalgorithmus (GRID) und mit *Moving-Boundaries* (Algorithmus 4.2, FSI), Fehlereduktion und Iterationszahlen k.

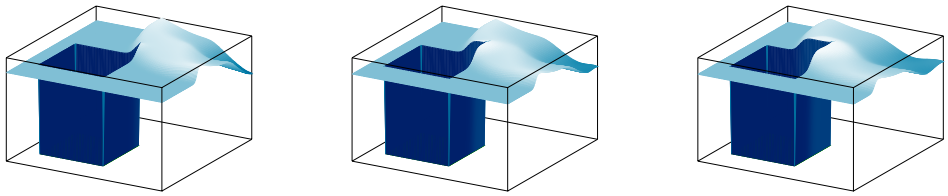
der numerischen Ergebnisse von Algorithmus 4.2 gegenüber Algorithmus 3.1 etwa 20%, wodurch ersterer als validiert gelten kann. Zusätzlich ist in den dargestellten (und allen im Rahmen dieser Arbeit durchgeführten) Simulationen die Anzahl der benötigten Iterationen bis zum *Steady-State* bei dem FSI-Löser immer kleiner als beim Grundalgorithmus, was die Annahme der Korrektheit des Verfahrens zusätzlich unterstützt.

6.5 Impulsaustausch und Festkörperreaktion

6.5.1 Vorgehensweise

Die korrekte Arbeitsweise der Kraftterme sowie der abgewandelten BFL-Regel konnten direkt in den letzten beiden Abschnitten gezeigt werden. Zusätzlich hat bereits Abschnitt 6.2 visuell und numerisch gezeigt, dass die Extrapolationen (Algorithmus 4.3) bei sich mit eigenem Antrieb durch das Fluid bewegenden Festkörpern korrekt arbeiten.

Durch das Fehlen eines Lösers für die Strukturmechanik, ist es jedoch schwierig, den Impulsaustauschalgorithmus (bzw. dessen Implementierung in dieser Arbeit, insbesondere mit den eingeführten Vereinfachungen) direkt zu evaluieren und zu validieren. Daher soll an dieser Stelle die Korrektheit des Impulsaustausches in Algorithmus 4.4 indirekt gezeigt werden. Dazu betrachten wir ein Szenario, bei dem ein an den Achsen des Koordinatensystems ausgerichteter (theoretisch instationärer) Würfel mit einer konstanten Momentangeschwindigkeit von Null bzw. das entsprechende Quadrat im $\mathbb{R}^2$ von einer durch einen Volldammbruch induzierten Welle exakt an einer seiner Ecken zuerst getroffen wird, vergleiche die Visualisierungen der Simulation in Abbildung 6.10. Hintergrund dieser Vorgehensweise ist folgender: In einer solchen symmetrischen Situation müssen die Beträge der diskreten Kräfte an den Kanten des Quadrates (den Seitenflächen des Würfels), die *adjazent* zu der von der Strömung getroffenen Ecke sind, gleich sein, was numerisch gemessen werden kann.



(a) nach 50 Zeitschritten (b) nach 60 Zeitschritten (c) nach 70 Zeitschritten

Abbildung 6.10: 3D-Ansicht der Testsimulation für den Impulsaustauschalgorithmus: Der Festkörper wird an der nord-östlichen Ecke von der Strömung zuerst getroffen, die Welle bewegt sich gleich schnell an beiden betroffenen Kanten entlang. Da der stationäre Festkörper im Ergebnishöhenfeld durch Null-Werte repräsentiert wird, sind seine Begrenzungen durch (fast) senkrechte Wände dargestellt.

Zusätzlich müssen die diskreten Kräfte entlang der Kanten des Quadrats bei einer glatten zeitabhängigen Lösung für das Höhenfeld der Simulation die Eigenschaft haben, (als diskrete Funktion betrachtet) ebenfalls glatt zu sein, was sich relativ einfach überprüfen lässt. Zuletzt müssen selbige Kräfte offensichtlich proportional zur Fluidhöhe (Masse) sein, da sie maßgeblich durch den hydrostatischen Druck in ihrer Größenordnung bestimmt werden. Letzteres bedeutet insbesondere, dass sich für jede Kante des Kontrollquadrats für den Festkörper überprüfen lässt, ob sich die durch den Impulsaustauschsalgorithmus berechneten Kräfte im Einklang mit der zeitabhängigen Wasserhöhe an der Fluid-Festkörpergrenze befindet.

6.5.2 Testszenario

Für diesen Test wird das in Abschnitt 6.2.3 beschriebene Volldammbruch-Szenario verwendet, wobei ein Festkörper mit quadratischer zweidimensionaler Repräsentation so plaziert wird, dass sein Mittelpunkt mit dem Mittelpunkt des den Volldammbruch darstellenden Quaders und einer Ecke des ebenfalls quadratischen Rechengebietes eine Linie bilden.

6.5.3 Ergebnisse

Die resultierende Gesamtkraft nach Gleichung (4.31) an der oberen Kante des Festkörpers für verschiedene Zeitschritte (k) ist beispielhaft für eine relativ geringe Auflösung von 100×100 *lattice*-Zellen in Abbildung 6.11 dargestellt. Dabei bewegt sich die Welle Abbildung 6.10 entsprechend also von rechts nach links. Es ist festzuhalten, dass

1. sich die Kraft proportional zur Wassertiefe verhält und

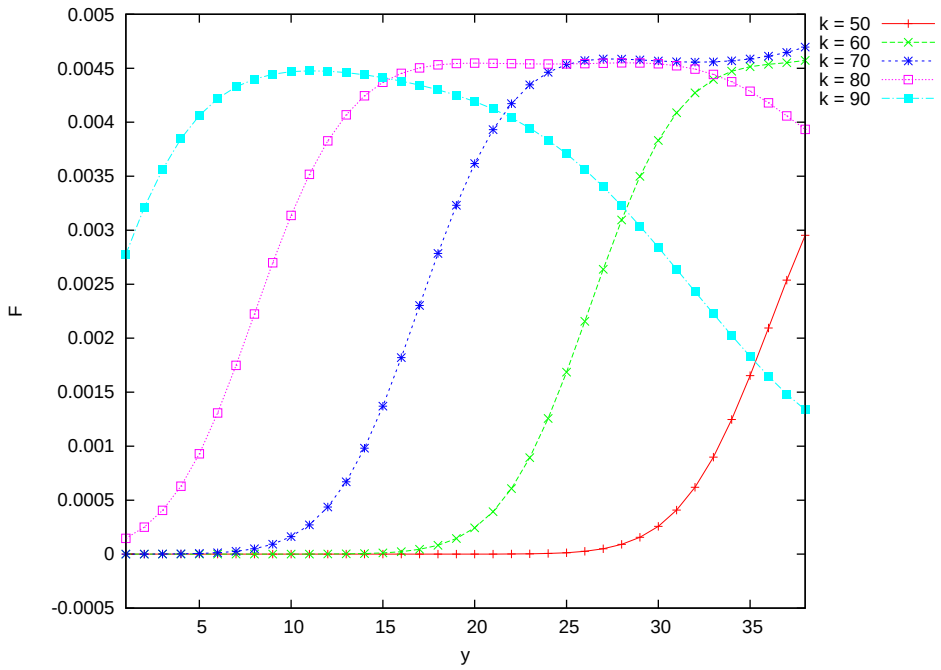


Abbildung 6.11: Gesamtkraft durch den Impulsaustausch auf eine Seite des Festkörpers zu verschiedenen Zeitpunkten: Die Strömung bewegt sich von rechts nach links.

2. in jedem Zeitschritt die Werte für die auf den Festkörper an der Kante einwirkenden Kräfte hinreichend glatt sind, d.h. eine an diese Daten gekoppelte Physiksimulation nicht durch fehlerhafte Ausreißer gestört wird.

Die Beträge der Kraft entlang dieser Kante sind dabei numerisch identisch zu denen an der anderen mit der oberen rechten Ecke des Kontrollquadrats adjazenten Kante. Daher würde in einer solchen symmetrischen Situation eine Festkörperbewegung mit der Strömung exakt in Richtung der unteren linken Ecke des Rechengebiets resultieren, wobei keine Rotation des Körpers um die z-Achse vorhanden wäre, da der Drehimpuls wegen der symmetrischen Kräfte an den beiden besagten Kanten Null wäre. Im Rahmen dieser Arbeit wurden verschiedene Szenarien mit verschiedenen Diskretisierungsstufen auf diese Weise untersucht, mit dem Ergebnis, dass die oben beschriebenen Beobachtungen stets reproduzierbar sind. Daher kann auch Algortihmus 4.4 als validiert gelten.

6.6 *Performance*-Benchmarks und Echtzeitsimulation

6.6.1 Echtzeitbegriff

Typischerweise wird die Leistungsfähigkeit von CFD-Lösern mit Hilfe von gängigen Metriken bezüglich der Gleitkommaleistung ((G)Flop/s) bzw. des Datendurchsatzes ((G)B/s)

gemessen. Für CFD-Code, der auf Lattice-Boltzmann Methoden basiert, werden oft die Anzahl der *lattice*-Zellen, die pro Sekunde verarbeitet werden können (*(Mega) Lattice Updates per second*, (M)LUP/s) herangezogen. Diese Vorgehensweise ist im Fall der hier vorliegenden Arbeit jedoch nur begrenzt sinnvoll, da solche Metriken nur indirekt einen Rückschluss auf die praktische Einsetzbarkeit in interaktiven 3D Applikationen zulassen. So kann beispielsweise die reine Gleitkommaleistung aufgrund der Problemgröße sehr hoch sein, obwohl die benötigte Ausführungszeit für einen Zeitschritt bereits zu groß ist, als dass man noch von einer Echtzeitsimulation sprechen kann. Anders ausgedrückt ist die praktische *Peak-Performance* der Algorithmen weniger relevant als die tatsächliche Ausführungszeit selbiger, bzw. die Anzahl der Zeitschritte, die bei gegebener Problemgröße pro Sekunde berechnet werden können. Im Folgenden wird daher von *frames per second* gesprochen, wenn dieser Wert gemeint ist, obwohl wir uns hier auf die Anzahl der pro Sekunde erzeugten Lösungen der Algorithmen beschränken, alle Messungen also insbesondere unabhängig von der gewählten Visualisierung durchführen.

Nach Möller und Haines [24] ist eine Applikation dann echtzeitfähig, wenn sie mit einer Rate von mindestens 15fps Ergebnisse bereitstellt und diese darstellt. Obwohl die Interaktionsfähigkeit bei einer solchen Bildrate (*framerate*) sicherlich gegeben ist, ist die Darstellung dabei unbefriedigend. Erst ab einer Rate von deutlich mehr als 20fps ist die Darstellung für das menschliche Auge nahezu übergangslos, ab einer *framerate* von über 70fps kann kein qualitativer Unterschied mehr wahrgenommen werden. Daher wird in dieser Arbeit die untere Schranke für die *framerate*, so dass der entsprechende Algorithmus noch in Echtzeit rechnet, auf einen moderaten Wert von 30fps festgelegt. Dieser Wert ist sicherlich nur dann aussagekräftig, wenn man unterstellt, dass das gewählte Darstellungsverfahren keine wesentlich höhere Auflösung hat, als die Auflösung der Simulation durch räumliche Diskretisierung, was in dieser Arbeit vorausgesetzt wird.

6.6.2 Benchmarks der Algorithmen

Aufgrund der in Kapitel 5 beschriebenen Implementierung der FSI-Module wird der Impulsaustausch auch dann durch die Operation `CollideStreamFSI` berechnet, wenn nur stationäre Festkörper in der Simulation vorhanden sind. Zudem ist der übliche Anwendungsfall für die Flachwassergleichungen durch die möglichst genaue Berücksichtigung des Bodenprofils charakterisiert. Daher werden im Folgenden zunächst drei verschiedene Algorithmen betrachtet:

1. Der Grundalgorithmus 3.1,
2. der Algorithmus für die inhomogenen Flachwassergleichungen unter Berücksichtigung beider Kraftterme (Algorithmus 4.1) und
3. der vollständige Algorithmus für die FSI, inklusive der Kraftterme (Algorithmus 4.4).

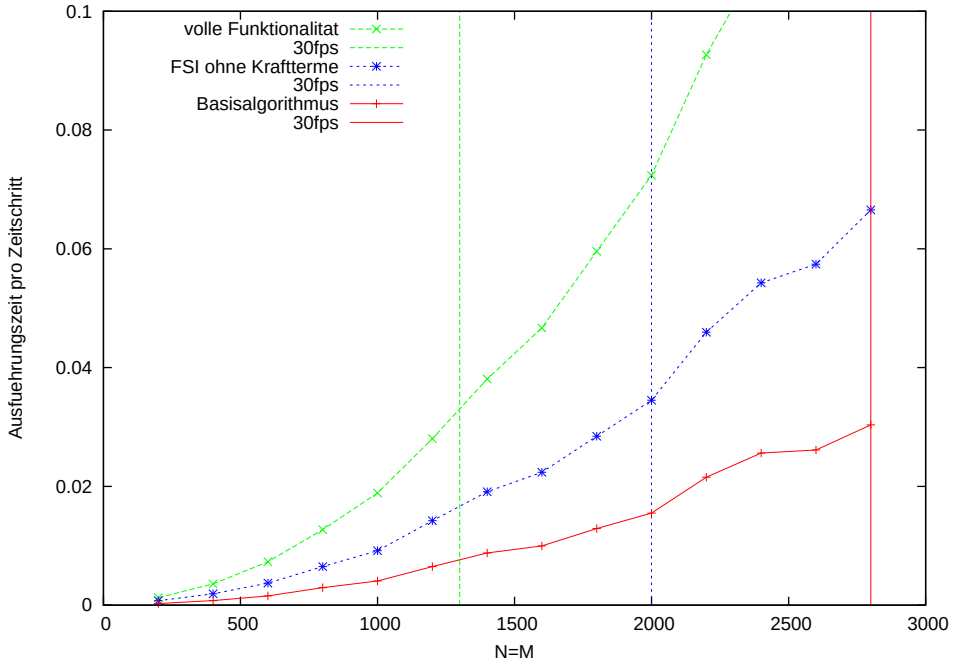


Abbildung 6.12: Ausführungszeit pro Zeitschritt und Echtzeitgrenze für den Basisalgorithmus, den Algorithmus mit beiden Bodenkrafttermen und den vollständigen FSI-Algorithmus.

Die Benchmark-Szenarios entsprechen dabei denen, die bereits in Abschnitt 6.2 verwendet werden, d.h. im Einzelnen, dass für das Benchmark des Grundalgorithmus das Teildammbruchszenario PCuDB, für das Benchmark mit Krafttermen $\text{FCuDB}_{\text{slope}}$ und schließlich $\text{BoxFSI}_{\text{adv}}$ für den vollständigen Algorithmus 4.4 eingesetzt wird. Die entsprechenden Rechenzeiten bei unterschiedlicher Auflösung des *lattice*, $N \times M$ sind in Abbildung 6.12 visualisiert, wobei die Problemgröße auf der Ordinate des Diagramms die Kantenlänge $N = M$ des Rechengebietes zeigt. Zusätzlich visualisieren die vertikalen Linien die jeweiligen Echtzeitgrenzen gemäß obiger Definition, d.h. alle Problemgrößen links von diesen Grenzen kann der Algorithmus jeweils noch mit einer *framerate* von mindestens 30fps lösen. Als Testsystem dient wie bisher die NVIDIA GeForce GTX 285.

Der Basisalgorithmus kann ein mit über 7.8 Millionen *lattice*-Zellen diskretisiertes Rechengebiet noch mit einer *framerate* von 30fps behandeln (d.h. ein Rechengebiet der Dimension 2800×2800), was die Effizienz der derzeitigen HONEI-Implementierung auf der GPU eindrucksvoll demonstriert. Bei Zuschaltung der Kraftterme resultiert eine Reduktion dieser Zahl auf etwa 4 Millionen (2000×2000) Zellen. Diese Verringerung der *Performance* ist sicherlich zum Teil den zusätzlichen Gleitkommaoperationen geschuldet, da der Grundalgorithmus abgesehen von der `EquilibriumDistribution` keine Operation mit den Krafttermen vergleichbarer Rechenintensität aufweist. Im Wesentlichen sind jedoch die bei Krafttermen anders als bei allen Basismodulen benötigten Programmverzweigungen häu-

figer: Sowohl die durch das semi-implizite numerische Schema nötigen Interpolationen, als auch die Berücksichtigung von *Dry-States* (die auch bei subkritischen Szenarien erfolgt), sowie die dadurch nötigen Verzweigungen in der `Extraction` durch Einsatz des adaptiven Limiters erhöhen die Komplexität des Codes gegenüber dem Grundalgorithmus beträchtlich. Dadurch werden auf der GPU Speicherzugriffe wesentlich häufiger sequentiell pro Thread durchgeführt, vergleiche Abschnitt 5.3.4.

Bei vollständiger Funktionalität, d.h. bei zusätzlicher Verwendung der Module `CollideStreamFSI` und `BoundaryInitFSI` kann der Algorithmus immerhin noch bei einer Problemgröße von 1.7 Millionen *lattice*-Zellen (einem Rechengebiet der Dimension 1300×1300) eine Echtzeit-*framerate* erreichen. Auch dabei gilt, dass die zum Einsatz kommenden Zusatzmodule die des Grundalgorithmus in Rechenintensität und Komplexität bezüglich nötiger konditionaler Programmverzweigungen um ein Vielfaches übersteigen, was die Verringerung der *Performance* erklärt. Des Weiteren befinden sich alle hier vorgestellten Implementierungen derzeit noch auf einem wesentlich geringeren Optimierungsstand, als die Basismodule des `PackedGrid`-basierten Algorithmus. Das Ergebnis, ein derart hoch aufgelöstes Rechengebiet auf handelsüblicher Hardware in Echtzeit behandeln zu können muss als Erfolg gewertet werden, insbesondere bei der in den vorangegangenen Abschnitten dieses Kapitels demonstrierten Genauigkeit und wenn man die Implementierung auf der GPU mit einer „*straight-forward*“ Implementierung auf für die CPU vergleicht, wie im nächsten Abschnitt dargestellt. Es sei jedoch noch einmal betont, dass diese Erweiterungen sowohl numerisch, als auch vom Standpunkt der Implementierung aus, experimentellen Charakter haben. Beispielsweise ist der Einsatz eines Korrekturmoduls für das *Streaming* sicherlich aus softwaretechnischer Sicht gerechtfertigt. Die Ersetzung des `CollideStreamGrid` Moduls durch eine neue Operation, die die Funktionalität beider Algorithmen vereint ist jedoch aus Sicht der *Performance* vorzuziehen. Es ist die Überzeugung des Authors dieser Arbeit, dass die Rechenzeit der hier vorgstellten Implementierungen insbesondere durch Verschmelzen der Operationen noch weiter signifikant gesenkt werden kann, was den Rahmen dieser Arbeit jedoch weit überschritten hätte. Der *Performance*-Verlust der hier implementierten Verfahren gegenüber dem Basisalgorithmus ist im Hinblick auf die gesteigerte Funktionalität und die daraus resultierenden Möglichkeiten absolut vertretbar.

6.6.3 Hardwarevergleich

In diesem Abschnitt soll der Unterschied in der Rechenleistung bezogen auf die im letzten Abschnitt verwendete Metrik zur Echtzeitfähigkeit durch den Einsatz verschiedener Hardware, insbesondere den Unterschied zwischen GPUs verschiedener Hardwaregenearationen, demonstriert werden. Dazu verwenden wir das $\text{BoxFSI}_{\text{adv}}$ Szenario für Algorithmus 4.4. Obwohl die CPU-Referenzimplementierung der FSI-Module vornehmlich der numerischen Validierung dient, soll im Folgenden auch der Löser auf der Core i7 CPU evaluiert werden.

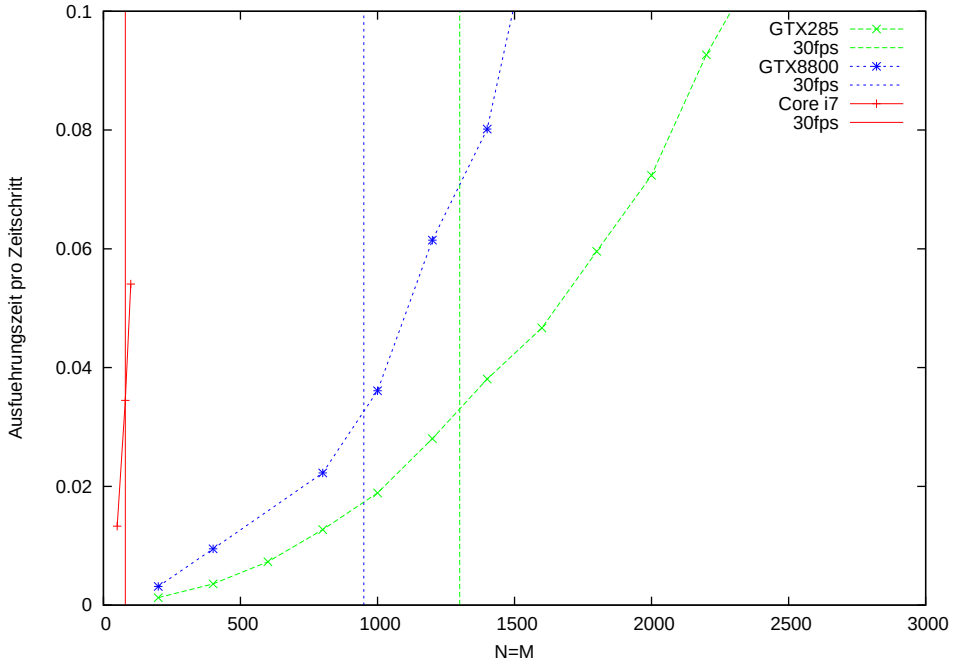


Abbildung 6.13: Vergleich der Ausführungszeiten für einen Zeitschritt bei steigender Auflösung des Rechengebietes auf verschiedener Hardware.

Dabei werden die entsprechenden Basismodule mit dem HONEI SSE-*backend* ausgeführt, d.h. manuell optimierter Code wird verwendet, falls dieser vorhanden ist. Damit verbleiben nur die FSI-Operationen in nicht manuell für SSE optimierter Form bei dieser Simulation, so dass zwar kein direkter Vergleich zwischen CPU- und GPU-Implementierung zulässig ist, man jedoch einen Eindruck davon erhält, inwieweit die GPU Varianten einen *Performance*-Gewinn gegenüber einer Erweiterung des Grundalgorithmus abschneidet, die die Vektorisierung des CPU-Codes dem Compiler überlässt. Des Weiteren wird der CPU-Code lediglich mit einem Thread ausgeführt. Das Ergebnis dieses Benchmarks ist in Abbildung 6.13 dargestellt. Dabei wird deutlich, dass die GTX 8800 bei 30fps ein Rechengebiet mit den Abmessungen 950×950, d.h. mit mehr als 900000 *lattice*-Zellen verarbeiten kann, also etwas mehr als halb so viele, wie die GTX 285, was etwa den Erwartungen entspricht, wenn man die Verdoppelung der Multiprozessoren (und der Speicherbandbreite) der GTX 285 gegenüber dem älteren Modell betrachtet (vergleiche Tabelle 6.1). Die CPU-Implementierung kann auf dem derzeitigen Entwicklungsstand gerade ein Gebiet mit $80 \times 80 = 6400$ *lattice*-Zellen unter Einhaltung des angelegten Echtzeitkriteriums integrieren, fällt also selbst im Vergleich mit der GTX 8800 weit zurück. Ribbrock zeigt in seiner Diplomarbeit, dass der Basisalgorithmus auf der CPU bei einem Thread und manueller SSE-Optimierung etwa um den Faktor 16 langsamer ist, als der entsprechende GPU-Code. Im Optimalfall kann dieser Faktor bei Einsatz von 4 Threads auf etwa 6 reduziert werden. Mit den bisherigen Ergebnissen der hier vorliegenden Arbeit ist daher zu erwarten, dass auch eine vollständig

implementierte SSE Variante des FSI-Lösers auf dem Vierkernsystem bei optimaler Anzahl von Threads und einer *framerate* von 30fps nur etwa 300000 *lattice*-Zellen (im Vergleich zu 900000 bzw. 1.7 Millionen auf den GPUs) bewältigen können wird.

Zusammenfassend lässt sich zu den Ergebnissen hinsichtlich der Leistungsfähigkeit sagen, dass hinreichend fein aufgelöste komplexe Strömungsprobleme auf der GPU mit den hier vorgestellten Implementierungen möglich sind. Dabei wurde bisher nicht berücksichtigt, dass auch die Visualisierung und ein Löser für den Strukturmechanik-Teil der Simulationen einen Teil der Ressourcen beanspruchen. Das *rendering* nimmt normalerweise, je nach Darstellungsart, zwischen 10 und 30 Prozent der Rechenzeit ein (vergleiche beispielsweise Fierz [11]). Unabhängig von der Festkörperphysikseite eines gekoppelten Gesamtverfahrens, verbleibt für die Fluidseite der Simulation daher etwa eine *lattice*-Größe von 1.4 bis 1.5 Millionen *lattice*-Zellen.

6.7 Visuelle Ergebnisse der Echtzeitsimulation

In den bisherigen Abschnitten dieses Kapitels wurden sowohl die numerische Korrektheit als auch die Echtzeitfähigkeit der vorgestellten Implementierungen demonstriert. Zuletzt sollen an dieser Stelle Beispiele für die visuellen Ergebnisse des Codes präsentiert werden. Zur Visualisierung der Höhenfelder für die Fluidoberfläche und das Bodenprofil sowie die Festkörper wurde im Rahmen dieser Diplomarbeit eine auf OpenGL basierende Applikation implementiert. Für Details zur OpenGL-Programmierung sei an dieser Stelle auf das OpenGL Redbook verwiesen [34]. Für die Entwicklung der grafischen Benutzeroberfläche wurde dabei die plattformunabhängige Bibliothek Qt verwendet [30]. Abbildungen 6.14 bis 6.24 zeigen die Ergebnisse von einigen Simulationen, die die Möglichkeiten der Erweiterungen im Blickfeld dieser Arbeit demonstrieren sollen.

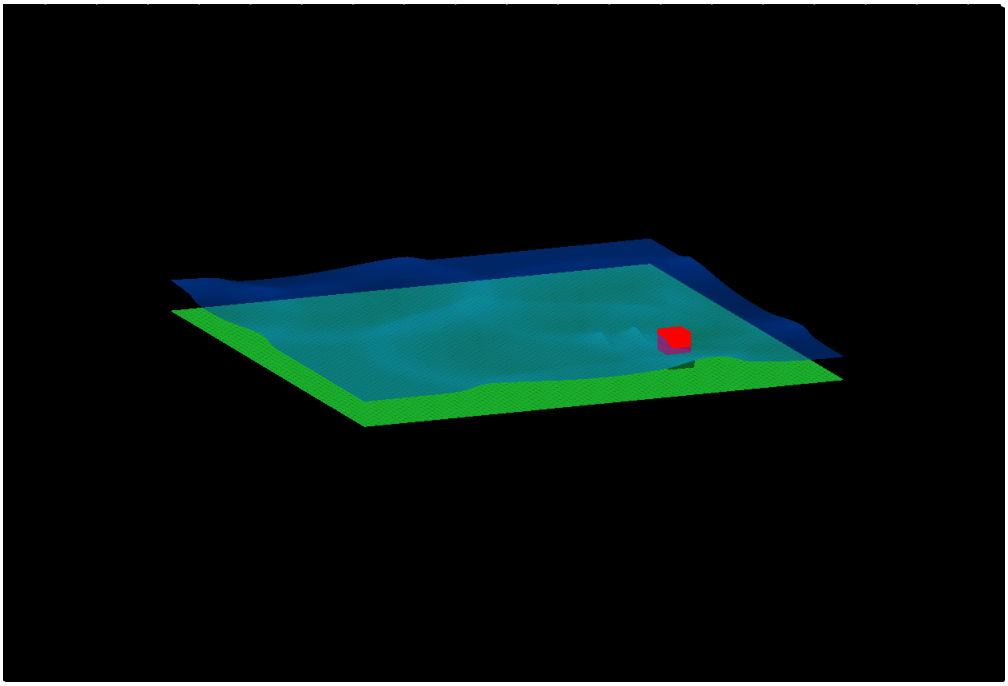


Abbildung 6.14: Bewegung einer Box parallel zur x-Achse durch bewegtes Wasser nach multiplen Dammbrüchen

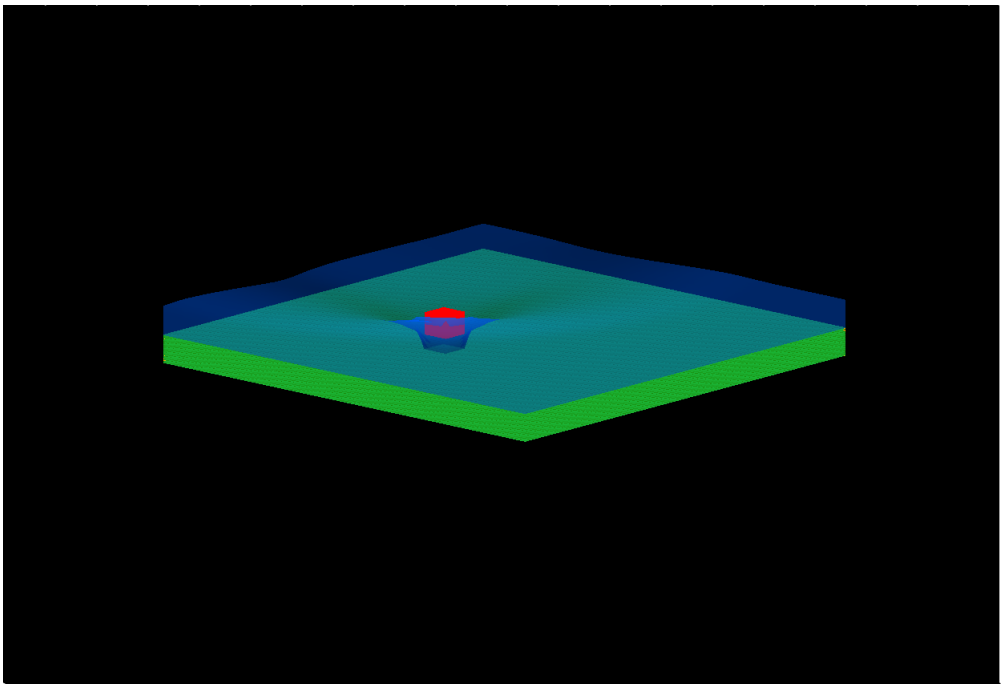


Abbildung 6.15: Bewegung einer Box in nicht-kartesischer Richtung...

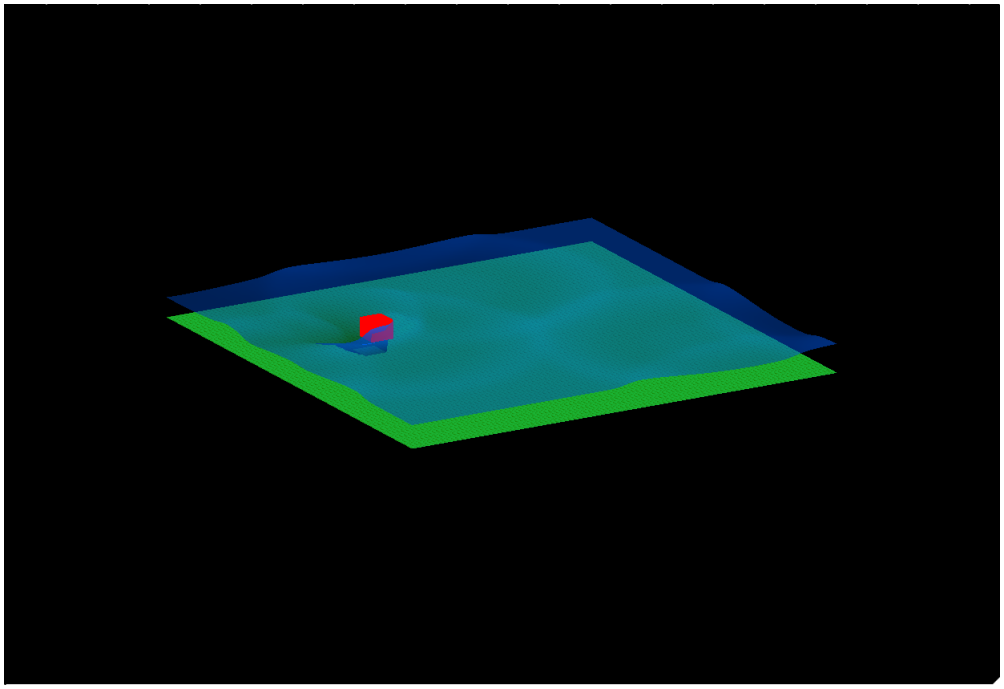


Abbildung 6.16: ...nach multiplen Dammbrüchen...

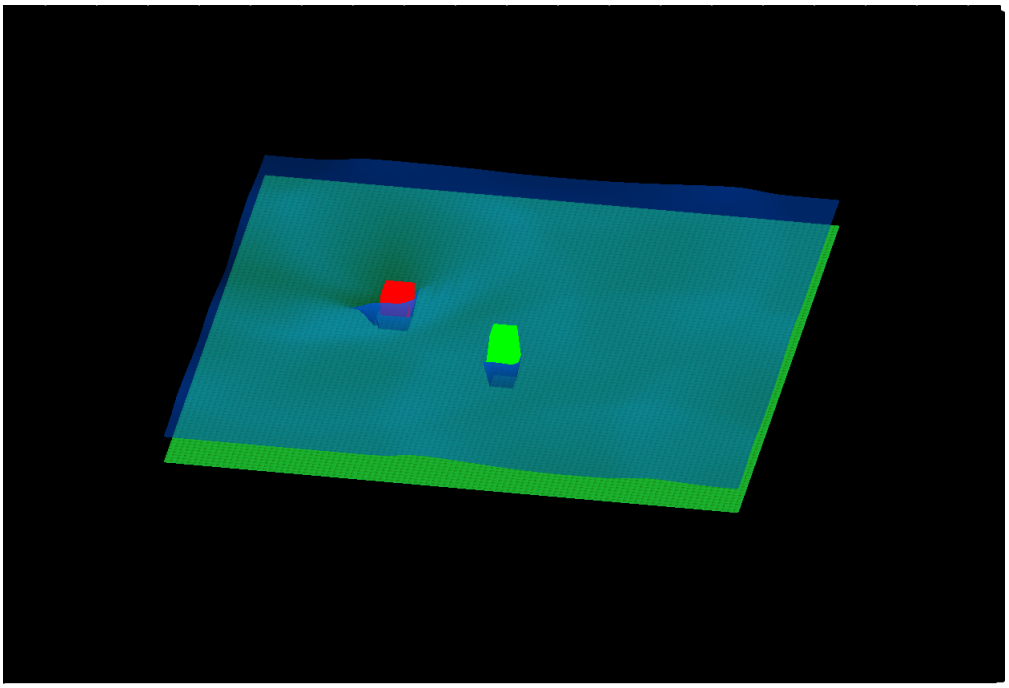


Abbildung 6.17: ...mit einem stationären Hindernis...

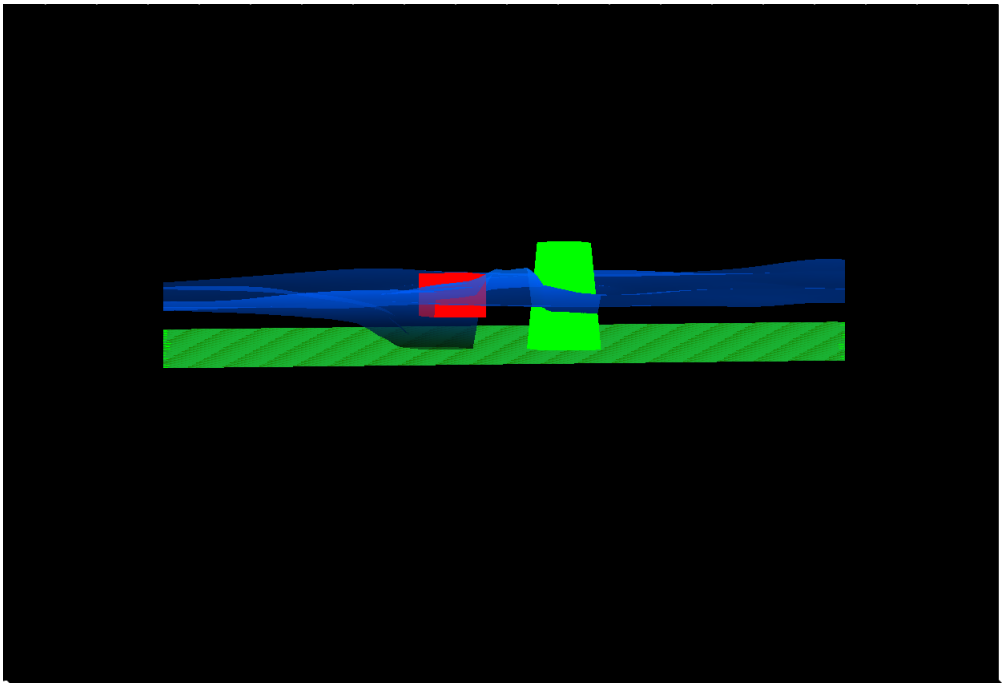


Abbildung 6.18: ...aus einer anderen Perspektive.

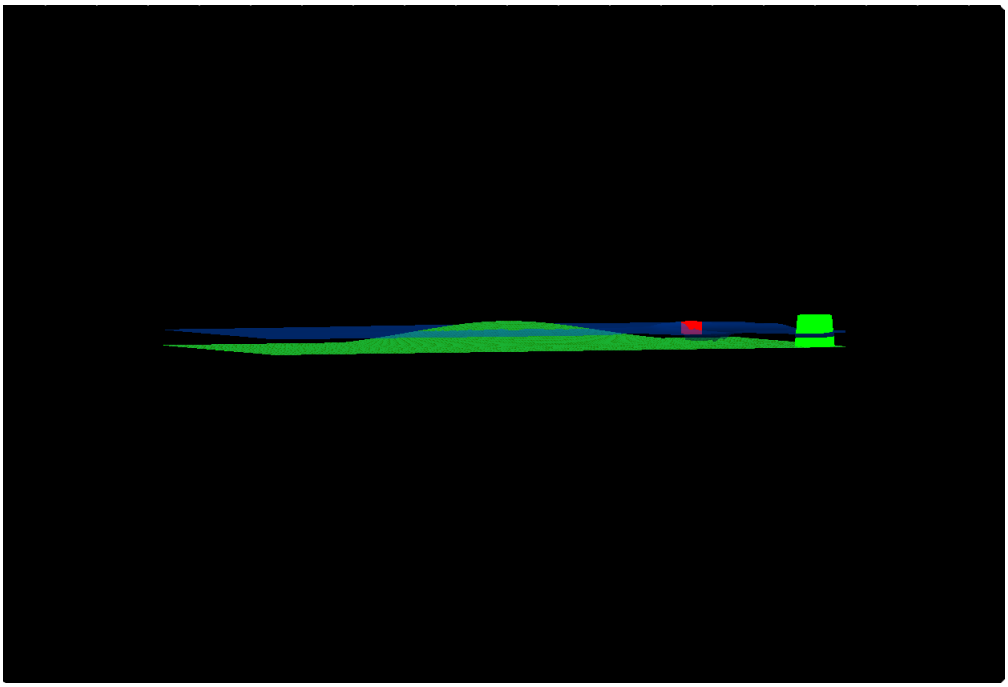


Abbildung 6.19: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* I

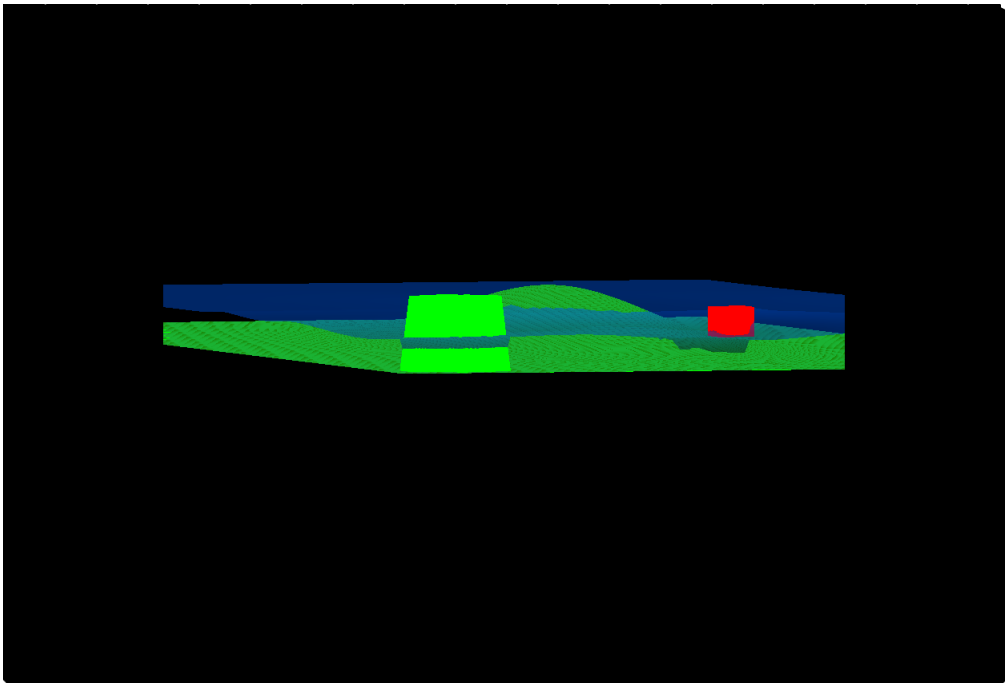


Abbildung 6.20: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* II

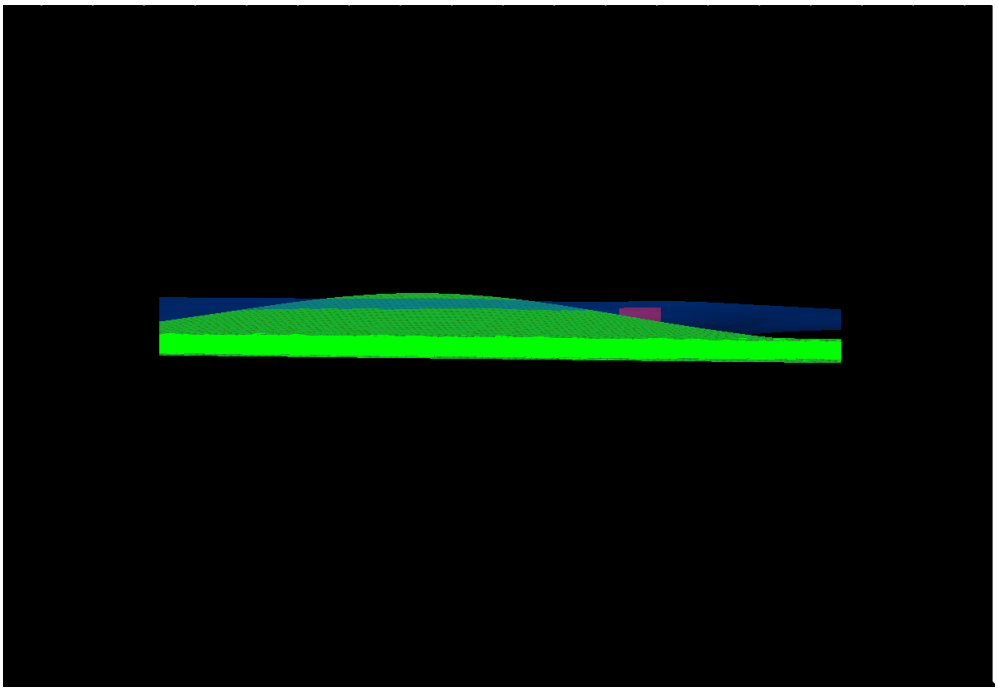


Abbildung 6.21: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* III

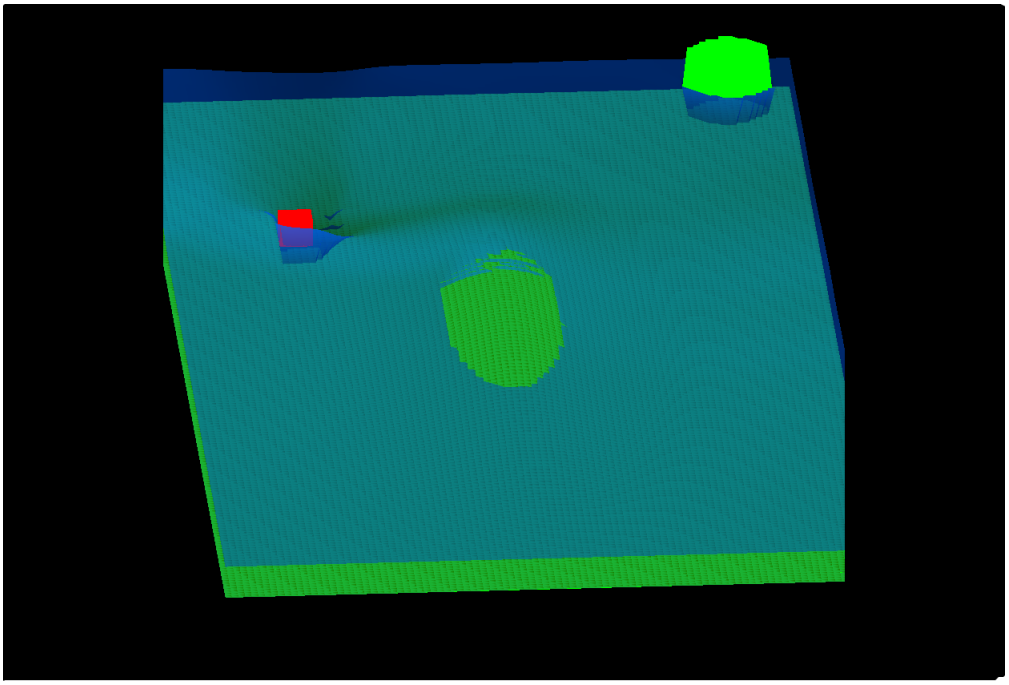


Abbildung 6.22: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* IV

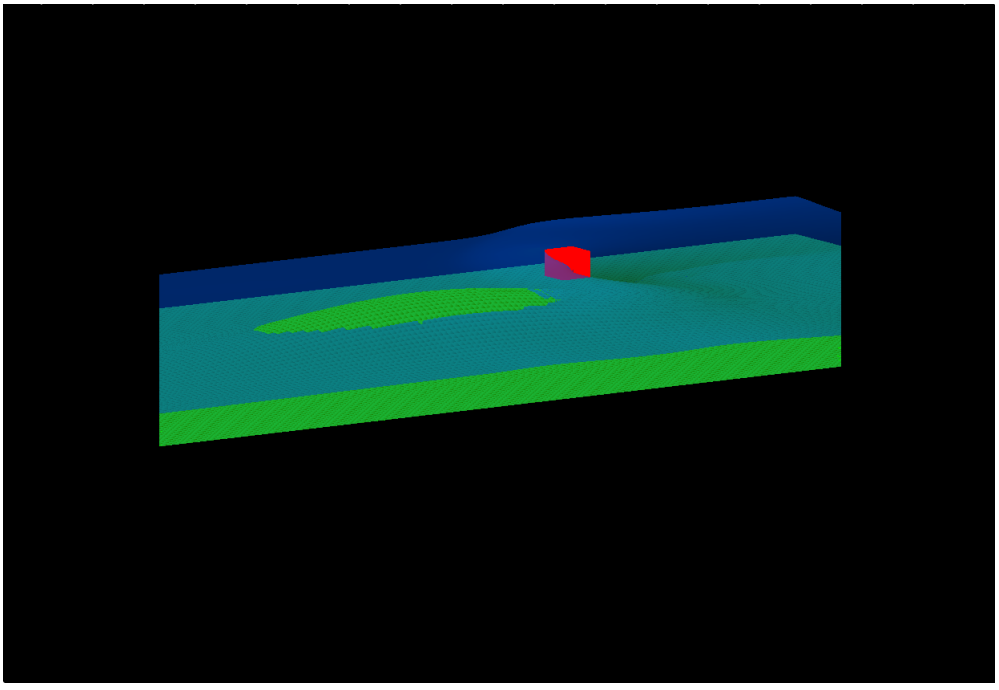


Abbildung 6.23: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* V

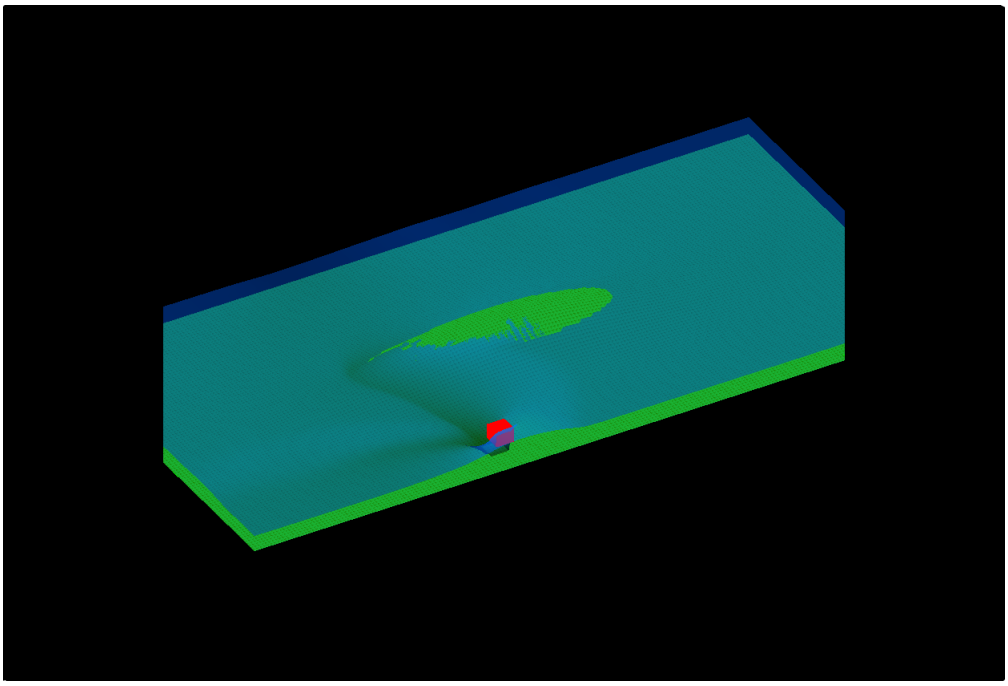


Abbildung 6.24: Diagonalbewegung einer Box, stationäres Hindernis, unebenes Bodenprofil und *Dry-States* VI

7 Zusammenfassung und Ausblick

Viele Löser für die Strömungssimulation sind auf solche Szenarien beschränkt, bei denen die Flüssigkeit innerhalb eines Gebietes in Bewegung versetzt wird. Im Rahmen dieser Arbeit wurde die Lattice-Boltzmann Methode für komplexere Flachwassersimulationen beschrieben, hinsichtlich ihres Einsatzes in der 3D-Echtzeitcomputergrafik erweitert und auf der GPU implementiert. Dabei lag der Fokus auf möglichst effizienten Algorithmen zur echtzeitfähigen Simulation einfacher Fluid-Struktur Interaktionen (FSI), die maßgeblich für eine Interaktion der Fluidsimulation mit einer dreidimensionalen Szene sind. Die FSI wurde dabei ebenfalls in zwei Dimensionen modelliert. Grundlage dafür waren die inhomogenen Flachwassergleichungen, die mit der Einbeziehung von Kräften, die durch die Bodentopographie induziert werden, bereits mit einer dreidimensionalen Szene interagieren können. Die entsprechende Implementierung des numerischen Lösungsverfahrens wurde aufbauend auf einem effizient implementierten Basisalgorithmus daher zunächst um entsprechende Kraftterme erweitert, mit denen das Gefälle und die Reibung durch die Bodentopographie in das Modell internalisiert werden können. Zusätzlich wurde ein auf Limiter-Funktionen basierendes experimentelles Verfahren vorgestellt, mit dem es möglich ist, auch *Dry-States* zu behandeln, was normalerweise nicht innerhalb der Möglichkeiten der Methode liegt.

Grundlage für die Einbeziehung einer Reaktion der Flüssigkeit auf nicht-deformierbare Festkörper waren die Erweiterung der Lattice-BGK Methode um eine experimentell vereinfachte BFL-Regel für die Behandlung von *Moving Boundaries* und Extrapolationsmethoden für die makroskopischen Größen Fluidtiefe und -geschwindigkeit bei einer Bewegung des Festkörpers. Schließlich wurde das Verfahren hinsichtlich einer Koppelung mit einer Festkörperphysiksimulation um den Impulsaustauschalgorithmus erweitert.

Die zentrale Idee dieser Modellierung, die komplette Flüssigkeitssimulation mit der Lattice-BGK Methode für die zweidimensionalen Flachwassergleichungen durchzuführen, war durch den sehr hohen Rechenaufwand von dreidimensionalen Verfahren zur Strömungssimulation motiviert. Mit der hier verwendeten Kombination von Methoden sollte es möglich sein, diese auch in zwei Dimensionen durchzuführen und trotzdem mit visuell überzeugendem Ergebnis in der interaktiven Computergrafik einzusetzen.

Die Implementierung der beschriebenen Algorithmen mit den vorgestellten Erweiterungen erfolgte mit dem Fokus auf die GPU, da Verfahren zur Strömungssimulation auch im zweidimensionalen Bereich einen enormen Ressourcenbedarf haben, wenn bestimmte Qualitätskriterien, wie eine möglichst hohe Auflösung des Rechengebietes angelegt wer-

den. Dass Standard-PC Prozessoren für diese Art von Simulationen nur begrenzt geeignet sind, wurde zu Beginn der Arbeit gezeigt und damit der Einsatz der GPU motiviert. Es wurde die Numerik Bibliothekssammlung HONEI vorgestellt, die im Rahmen dieser Arbeit auf die oben beschriebene Funktionalität erweitert wurde und grundlegende Prinzipien zur Programmierung von GPUs für wissenschaftliche Aufgaben, sowie ihre Architektur beschrieben und die Implementierung der vorgestellten Erweiterungsoperationen diskutiert.

Die einzelnen Komponenten der beschriebenen Erweiterungen wurden durch numerische Tests validiert. Dabei ist festzuhalten, dass alle Erweiterungsmodule diesen Tests standgehalten haben. Im Einzelnen konnte gezeigt werden, dass alle Löser unter Erhaltung der Masse arbeiten und insbesondere visuell keine Unterschiede zu analytischen und externen Referenzlösungen zu verzeichnen sind. Die experimentellen Erweiterungen erfüllen außerdem ihren Zweck: Unter bestimmten Umständen ist es sogar möglich *Dry-States* zu simulieren, obwohl die Stabilität der Simulation nur durch vorsichtige Feineinstellungen bei der Parametrisierung sichergestellt werden kann, wodurch die vorgeschlagene Methode im gegenwärtigen Stadium nur begrenzt in der Echtzeit-Computergrafik einsetzbar ist. Weiterhin konnte gezeigt werden, dass sowohl die Extrapolationsmethoden, als auch der Impulsaustauschalgorithmus in der vorliegenden Implementierung korrekte Ergebnisse liefert. Insbesondere erzielt die BFL-Regel durch Repräsentation von Randbedingungen auf mikroskopischer Ebene eine numerische Verbesserung gegenüber dem Grundalgorithmus, der diese lediglich auf makroskopischer Ebene berücksichtigt.

Durch den Einsatz der GPU als Implementierungsplatform konnte gezeigt werden, dass die Simulation von interaktiven Flüssigkeiten mit einer Auflösung von bis zu 1300×1300 *lattice*-Zellen bei einer Bildrate von 30fps möglich ist. Schließlich zeigen die Visualisierungen der Simulationen, dass damit tatsächlich der Eindruck einer dreidimensionalen Szene entsteht.

Die hier zusammen- und vorgestellte Methode ist als Einstiegspunkt zu verstehen: Die einzelnen Operationen können auf vielfältige Weise erweitert und an die individuellen Bedürfnisse einer Applikation angepasst werden. Insbesondere die FSI-Module sollen zukünftig auf allen Architekturen, die HONEI unterstützt implementiert werden. Dies schließt auch eine Neu-Konzeptionierung der Operationen, wie das Verschmelzen der *Streaming*-Korrektur und der erweiterten BFL-Regel mit dem Transportschritt der Grundmethode mit ein. Zusätzlich kann die Methode in ihrer Genauigkeit verbessert werden, indem die Operationen den Abstand zu einem inneren Rand exakt berücksichtigen. Die Stabilität der Algorithmen ist indes insbesondere im Hinblick auf *Dry-States* künftig zu verbessern, was eine eingehendere Untersuchung der experimentellen Methode erfordert, als im Rahmen dieser Arbeit möglich war. Schließlich sind viele Flachwassersimulationen dadurch gekennzeichnet, dass in vielen Teilen des Rechengebietes eine wesentlich weniger große Auflösung benötigt wird. Dies soll zukünftig durch ein adptives *lattice* berücksichtigt werden, was zu einem zusätzlichem *Performance*-Gewinn führt.

Literaturverzeichnis

[1] AMD Corporation: *ATI Stream Computing - Technical Overview.* `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`, 2009.

[2] Baboud, Lionel und Xavier Décoret: *Realistic Water Volumes in Real-Time*, 2006.

[3] Barz, Till, Thorsten Deinert, Danny van Dyk, Markus Geveler, David Gies, Nina Harmuth, Volker Jung, Mathias Kadolsky, Sven Mallach, Andre Matuschek, Joachim Messer und Dirk Ribbrock: *Endbericht der Projektgruppe SmartCell.* Technischer Bericht, Technische Universität Dortmund, Fakultät für Informatik, Fakultät für Mathematik, 2008.

[4] Becker, Hendrik: *GPU-Basierte Echtzeit-Simulation und Visualisierung von Shallow Water Equations*, 2006. Diplomarbeit, Technische Universität Dortmund.

[5] Bogner, Simon: *Simulation of Floating Objects in Free-Surface Flows*, 2009. Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Informatik.

[6] Bouzidi, M'hamed, Mouaouia Firdaouss und Pierre Lallemand: *Momentum transfer of a Boltzmann-lattice fluid with boundaries.* Physics of Fluids, 13(11):3452–3459, 2001.

[7] Caiazzo, Alfonso: *Asymptotic Analysis of lattice Boltzmann method for Fluid-Structure interaction problems.* Doktorarbeit, Technische Universität Kaiserslautern, Scuola Normale Superiore Pisa, 2007.

[8] Delis, Anargiros I. und Theodoros D. Katsaounis: *Numerical solution of the two-dimensional shallow water equations by the application of relaxation methods.* Applied Mathematical Modelling, 29(8):754–783, 2005.

[9] Dyk, Danny van, Markus Geveler, Sven Mallach und Dirk Ribbrock: *HONEI Projekt Homepage.* `http://www.honei.org`.

[10] Dyk, Danny van, Markus Geveler, Sven Mallach, Dirk Ribbrock, Dominik Göddeke und Carsten Gutwenger: *HONEI: A collection of libraries for numerical computations targeting multiple processor architectures*, Mai 2009. to appear.

[11] Fierz, Basil: *Real-time Fluid Simulations with Wavelet Turbulence*, 2008. Dasterarbeit.

[12] Geller, S., J. Tölke, M. Krafczyk, S. Kollmannsberger, A. Düster und E. Rank: *A COUPLING ALGORITHM FOR HIGH ORDER SOLIDS AND LATTICE BOLTZMANN FLUID SOLVERS*. European Conference on Computational Fluid Dynamics ECCOMAS CFD, 2006.

[13] Geller, S., J. Tölke, M. Krafczyk, E. Scholz, A. Düster und E. Rank: *SIMULATION OF BIDIRECTIONAL FLUID-STRUCTURE INTERACTION BASED ON EXPLICIT COUPLING APPROACHES OF LATTICE BOLTZMANN AND P-FEM SOLVERS*. Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering COUPLED PROBLEMS, 2005.

[14] Göddeke, Dominik und Robert Strzodka: *Performance and accuracy of hardware-oriented native, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double Precision GPUs)*. Technischer Bericht, Fakultät für Mathematik, Technische Universität Dortmund, 2008. Nummer 370, invited talk at NVISION 2008 - The World of Visual Computing.

[15] Göddeke, Dominik, Robert Strzodka und Stefan Turek: *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*. International Journal of Parallel, Emergent and Distributed Systems, 22(4):221–256, Januar 2007.

[16] Götz, J., K. Iglberger, C. Feichtinger, S. Donath, F. Deserno und U. Rüde: *Comparing strategies for parallelizing large scale uid structure interaction using lattice Boltzmann methods and a physics engine*. Technischer Bericht, Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Informatik, 2009.

[17] Hänel, Dieter: *Molekulare Gasdynamik*. Springer, Berlin, Heidelberg, New York, 2004.

[18] Higuera, F. J. und J. Jimenez: *Boltzmann Approach to Lattice Gas Simulations*. EPL (Europhysics Letters), 9(7):663–668, 1989.

[19] Hübner, Th. und S. Turek: *Efficient simulation techniques of the Lattice Boltzmann equation on unstructured meshes*. Technischer Bericht, Fakultät für Mathematik, TU Dortmund, November 2007. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 355.

[20] Laan, Wladimir J. van der, Simon Green und Miguel Sainz: *Screen space fluid rendering with curvature flow*. In: *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, Seiten 91–98, New York, NY, USA, 2009. ACM.

[21] Lallemand, Pierre und Li-Shi Luo: *Lattice Boltzmann method for moving boundaries*. Journal of Computational Physics, 184(2):406 – 421, 2003.

[22] Li, Wei, Xiaoming Wei und Arie Kaufman: *Implementing Lattice Boltzmann Computation on Graphics Hardware*, 2003.

[23] Mei, R., L.-S. Luo und W. Shyy: *An Accurate Curved Boundary Treatment in the Lattice Boltzmann Method*. Journal of Computational Physics, 155:307–330, November 1999.

[24] Möller, Thomas und Eric Haines: *Real-Time Rendering*. A K Peters, Ltd., Natick, 1999.

[25] Network, Forest Science Labs Research: *Hydraulics reference Web page: Table for Manning's n values*, 2009. `http://www.fsl.orst.edu/geowater/FX3/help/8_Hydraulic_Reference/Mannings_n_Tables.htm`.

[26] NVIDIA Corporation: *An Introduction to NVIDIA PhysX*. `http://developer.nvidia.com/object/nvision08-PhysX_X.html`, 2008.

[27] NVIDIA Corporation: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide (Version 2.2)*. `http://www.nvidia.com/cuda`, 2008.

[28] Peter, Mark Carlson, Peter J. Mucha und Greg Turk: *Rigid Fluid: Animating the Interplay Between Rigid Bodies and Fluid*. In: *ACM Trans. Graph*, Seiten 377–384, 2004.

[29] Pohl, Thomas: *High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method*. Doktorarbeit, Universität Erlangen-Nürnberg, 2008.

[30] Project, QT: *QT Projekt Homepage*. `http://www.qtsoftware.com`.

[31] Qian, Yue Hong: *Lattice Gas and lattice kinetic theory applied to the Navier-Stokes equations*. Doktorarbeit, Universite Pierre et Marie Curie, Paris, 1990.

[32] Ribbrock, Dirk: *Entwurf einer Softwarebibliothek zur Entwicklung portabler, hardwareorientierter HPC Anwendungen am Beispiel von Strömungssimulationen mit der Lattice Boltzmann Methode*, 2009. Diplomarbeit.

[33] Schwanenberg, Dirk: *Die Runge-Kutta-Discontinuous-Galerkin-Methode zur Lösung konvektionsdominierter tiefengemittelter Flachwasserprobleme*. Doktorarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, 2003.

[34] Shreiner, Dave, Mason Woo, Jackie Neider und Tom Davis: *OpenGL Programming Guide: The Official Guide To Learning OpenGL Version 2.1*. Adison Wesley, Upper Saddle River, 2006.

[35] Succi, Sauro: *The Lattice Boltzmann equation. For Fluid Dynamics and Beyond.* Clarendon Press, Oxford, 2004.

[36] Thürey, N.: *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method.* Doktorarbeit, Mar 2007.

[37] Thürey, N., K. Iglberger und U. Rüde: *Free Surface Flows with Moving and Deforming Objects for LBM.* Proceedings of Vision, Modeling and Visualization 2006, Seiten 193–200, Nov 2006.

[38] Thürey, N., U. Rüde und M. Stamminger: *Animation of Open water Phenomena with coupled Shallow Water and Free Surface Simulation.* Proceedings of the 2006 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, Seiten 157–166, Jun 2006.

[39] Weidendorfer, J. und C. Trinitis: *Off-loading application controlled data prefetching in numerical codes for multi-core processors.* Int. J. Comput. Sci. Eng., 4(1):22–28, 2008.

[40] Wolf-Gladrow, Diether A.: *Lattice Gas Cellular Automata and Lattice Boltzmann Methods - An Introduction.* Springer, Berlin, Heidelberg, New York, 2000.

[41] Yu, Qizhi, Fabrice Neyret, Eric Bruneton und Nicolas Holzschuch: *Scalable Real-Time Animation of Rivers*, mar 2009. to appear.

[42] Zhou, Jian Gou: *Lattice Boltzmann Methods for Shallow Water Flows.* Springer, Berlin, Heidelberg, New York, 2004.