Strömungssimulation

Seminar Programmierung von Grafikkarten
Universität Kassel

Alexander Podlich und Jens Wollenhaupt
{podlich, jewollen}@student.uni-kassel.de

16. Juli 2006

Inhaltsverzeichnis

1 Einleitung **2**

2 Physik der Fluide **2**
2.1 Darstellung eines Fluids . . . 2
2.2 Navier-Stokes Gleichungen . . . 3
2.2.1 Terme in der Navier-Stokes Gleichung . . . 3
2.2.2 Berechnung von ∇ . . . 3
2.3 Lösungsmöglichkeiten der Navier-Stokes Gleichungen . . . 4

3 Lattice-Boltzmann Modell **4**
3.1 Lattice-Boltzmann Gitter . . . 4
3.2 Simulationsablauf . . . 5
3.2.1 LBM-Algorithmus . . . 6

4 Strömungssimulationen auf der Grafikhardware **6**
4.1 LBM-Algorithmus auf der GPU . . . 6
4.2 Repräsentation der Daten im Texturspeicher . . . 7
4.3 Voxilisation . . . 8

5 Einsatzgebiete **8**

6 Zusammenfassung und Ausblick **9**

Abbildungsverzeichnis

1 Das Geschwindigkeitsvektorfeld eines Fluids [3]. . . . 2
2 Untergitter in einem dreidimensionalen Lattice-Boltzmann Gitter. . . . 4
3 Die D3Q19-Zelle des Lattice Boltzmann Gitters. . . . 5
4 LBM-Algorithmus und seine Umsetzung auf der Grafikpipeline. . . . 7
5 Abspeichern dreidimensional angeordneter LBM-Zellen in einer zweidimensionalen Textur. . . . 8

1 Einleitung

Flüssige oder gasförmige Erscheinungen wie z.B. aufsteigender Dampf spielen eine große Rolle in graphischen Simulationen. Ein Modell, welches diese beschreibt, sollte nicht nur die Strömung selbst, sondern auch die Interaktion mit der Umgebung in einer physikalisch korrekten Weise erfassen. In dieser Ausarbeitung wird eine Methode zur Simulation von Strömungen vorgestellt, die komplett auf der GPU (Graphics Processing Unit) ausgeführt wird und um Vielfaches schneller arbeitet, als auf einer CPU.
Es wird eine Einführung in die generelle Strömungslehre gemacht und es werden mögliche Ansätze vorgestellt, wie eine Strömung simuliert werden kann. Zum einen wird ein mathematisch/physikalisches Modell mittels der Navier-Stokes Gleichungen vorgestellt, zum anderen ein molekulardynamisches Modell, das Lattice-Boltzmann Modell. Der Schwerpunkt dieser Ausarbeitung liegt in der Behandlung des Lattice-Boltzmann Modells. Es wird auf die Arbeitsweise des Lattice-Boltzmann Modells und die Umsetzung auf der GPU eingegangen und abschließend gibt es einen Ausblick auf mögliche Anwendungsgebiete, in dem sowohl die graphischen als auch die simulationstechnischen Aspekte angesprochen werden.

2 Physik der Fluide

2.1 Darstellung eines Fluids

Im Allgemeinen sind Strömungen Vorgänge, die in den drei Raumdimensionen stattfinden und zeitlich variabel (in-stationär) ablaufen. Strömungen können hierbei flüssige oder gasförmige Erscheinungen, wie z.B. aufsteigender Dampf, Rauch oder Wasser, sein, welche auch als *Fluide* bezeichnet werden.
Um das Verhalten eines Fluids zu simulieren, benötigt man eine mathematische Darstellung des Zustands des Fluids zu jedem Zeitpunkt und an jedem Ort. Hierbei ist das Geschwindigkeitsfeld $\vec{u}$ des Fluids die wichtigste Größe, da durch diese die Bewegung des Fluids direkt charakterisiert wird. Das Geschwindigkeitsfeld $\vec{u}$ variiert sowohl in Raum und Zeit und wird, wie der Name schon sagt, als ein Vektorfeld dargestellt.
Das Geschwindigkeitsfeld für ein Fluid wird für die Zwecke dieser Arbeit als eine Abbildung einer vektorwertigen Funktion auf ein kartesisches Gitter definiert, sodass für ein zweidimensionales Gitter für jede Position $\vec{x} = (x, y)$ zum Zeitpunkt t die Geschwindigkeit $\vec{u}(\vec{x}, t) = (u(\vec{x}, t), v(\vec{x}, t))$ zugeordnet wird. Die Komponente $u(\vec{x}, t)$ der Geschwindigkeit $\vec{u}(\vec{x}, t)$ stellt die Geschwindigkeit in x-Richtung an der Stelle $\vec{x}$ zum Zeitpunkt t dar und die Komponente $v(\vec{x}, t)$ die Geschwindigkeit in y-Richtung. Die Abbildung 1 repräsentiert einen Zustand einer Fluid-Simulation durch ein $M \times N$ Gitter. Die Pfeile stellen die Geschwindigkeitsvektoren $\vec{u}(\vec{x}, t)$ dar.

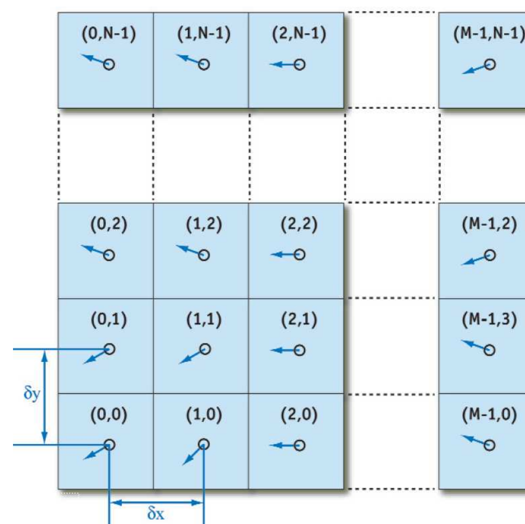

Abbildung 1: Das Geschwindigkeitsvektorfeld eines Fluids [3].

Der Kern einer Fluidsimulation besteht aus der Bestimmung des korrekten Geschwindigkeitfeldes zu jedem Zeitschritt. Dies kann durch die Lösung eines Gleichungssystems (z.B. die Navier-Stokes Gleichungen) erreicht werden, welches die Entwicklung des Vektorfelds über die Zeit und unter Einfluss verschiedener Kräfte beschreibt.

2.2 Navier-Stokes Gleichungen

Das Verhalten von Fluiden kann mittels eines mathematischen und physikalischen Modells, z.B. mit den Navier-Stokes Gleichungen, beschrieben werden. Diese Gleichungen beschreiben *inkompressible* und *homogene* Fluide. Diese vereinfachende Annahmen müssen, wie so oft in der Physik, getroffen werden, um komplexe Phänomene, wie Strömungen, überhaupt beschreiben zu können.
Inkompressible Fluide können hierbei nicht zusammengedrückt werden, *homogene* Fluide haben eine konstante Dichte ρ im Raum. Die Kombination dieser Eigenschaften bedeutet, dass die Dichte des Fluids sowohl räumlich als auch zeitlich konstant ist. Die folgende Beschreibung der Navier-Stokes Gleichungen ist an das Kapitel 38 der GPU Gems [3] angelehnt und ist gegeben durch:

$$\nabla \cdot \vec{u} = 0 \tag{1}$$

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{u} + F \tag{2}$$

Hierbei ist $\vec{u}$ das Geschwindigkeitsfeld, p das Druckfeld, ν die Zähheit des Fluids, ρ die Dichte und $\vec{F} = (f_x, f_y)$ eine beliebige externe Kraft, welche auf das Fluid wirkt.
Gleichung (1) beschreibt die Massenerhaltung und besagt, dass Einfluss und Ausfluss in ein Fluid ausgewogen sind. Gleichung (2) beschreibt die Impulserhaltung und besteht aus weiteren Gleichungen, die durch die Richtungsvektoren für Strömungen zustande kommen. Diese Richtungsvektoren geben an einer beliebigen Position $\vec{x}$ an, in welche Richtung die Strömung fließt. Im zweidimensionalen Fall besteht ein Richtungsvektor aus zwei Komponenten u und v. Pro Richtungskomponente gibt es somit eine partielle Differentialgleichung [5]:

$$\frac{\partial u}{\partial t} = -(\vec{u} \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f_x \tag{3}$$

$$\frac{\partial v}{\partial t} = -(\vec{u} \cdot \nabla)v - \frac{1}{\rho}\nabla p + \nu \nabla^2 v + f_y \tag{4}$$

Für ein zweidimensionales Fluid-Gitter ist es nun möglich mit den Gleichungen (1), (3) und (4) die drei Unbekannten u, v und p zu bestimmen.

2.2.1 Terme in der Navier-Stokes Gleichung

Wie in Abschnitt 2.2 erwähnt, beschreibt die Gleichung 2 die Impulserhaltung. Diese Gleichung kann in folgende 4 Beschleunigungsterme aufgeteilt werden:

1. **Die Advektion**. Die Geschwindigkeit eines Fluids bewirkt einen Transport von Objekten, Dichten und anderen Größen entlang der Strömung. Auch das Fluid selbst wird durch diesen Mechanismus transportiert und diese Selbstadvektion wird durch den ersten Term repräsentiert.

2. **Der Druck**. Da sich die Moleküle in einem Fluid zu einem gewissen Grad frei bewegen können, breitet sich eine Kraftaufbringung nicht sofort aus. Es baut sich ein Druckunterschied auf, welcher dann zu einer Beschleunigung führt. Diese Beschleunigung wird durch den zweiten Term ausgedrückt.

3. **Die Diffusion**. Der dritte Term stellt die Dicke, die *Viskosität*, des Fluids dar.

4. **Externe Kräfte**. Eine Beschleunigung des Fluids aufgrund von äußeren Einflüssen wird mit Hilfe des vierten Terms dargestellt. Dies können entweder lokale Kräfte (z.B. Ventilator) sein, die nur auf einen Bereich des Fluids wirken oder Kräfte, die gleichmäßig auf das gesamte Fluid wirken (z.B. Gravitationskraft).

2.2.2 Berechnung von ∇

Die Gleichungen (1), (2), (3) und (4) gebrauchen auf drei unterschiedliche Arten das Symbol ∇, welcher auch als der *Nabla*-Operator bekannt ist. ∇p ist hierbei der Gradient des Druckfeldes p, $\nabla \cdot \vec{u}$ stellt die Divergenz (= „das Auseinanderströmen") des Geschwindigkeitsfeldes $\vec{u}$ dar und $\nabla^2 \vec{u}$ beschreibt den Laplace Operator des Geschwindigkeitsfeldes $\vec{u}$. In dieser Ausarbeitung ist die genauere Analyse nicht notwendig und wird aus diesem Grund nicht weiter beschrieben. Genaueres lässt sich in Kapitel 38 der GPU Gems [3] nachlesen.

2.3 Lösungsmöglichkeiten der Navier-Stokes Gleichungen

Eine analytische Lösung der Navier-Stokes Gleichungen ist nur in ein paar wenigen physikalischen Spezialfällen möglich und ist daher für die Berechnung von Strömungen nicht relevant. Die naheliegendste Möglichkeit ist üblicherweise eine Lösung durch das Anwenden von numerischen Integrationsverfahren, bei der die Advektions-, Diffusions- und Kraftterme ausgerechnet werden. Eine genauere Beschreibung dieser Methode lässt sich in Kapitel 38 der GPU Gems [3] nachlesen. Im Folgenden wird mit dem Lattice-Boltzmann Modell ein anderer Ansatz vorgestellt.

3 Lattice-Boltzmann Modell

Ein Fluid besteht aus vielen kleinen Partikeln, deren kollektives Verhalten die makroskopischen Eigenschaften (siehe Unterkapitel 3.2) des Fluids bestimmt. In diesem Kapitel wird das Lattice-Boltzmann Modell (LBM) vorgestellt, mit dem es zwar nicht möglich ist, die Gleichungen (1) und (2) direkt zu lösen, aber die Möglichkeit bietet das Verhalten dieser Partikel auf einem diskreten Gitter zu diskreten Zeitschritten zu simulieren, sodass die Navier-Stokes Gleichungen erfüllt sind.

3.1 Lattice-Boltzmann Gitter

Das Fluidverhalten kann als ein aus der Bewegung und Kollision kleiner Partikel resultierender Ablauf betrachtet werden. Das Lattice-Boltzmann Modell kann hierzu auf einem 2D oder 3D Gitter definiert werden. Die verwendeten Gitter unterscheiden sich in ihrer Dimensionalität x und der Anzahl der Kollokationspunkte y und können mit der Schreibweise DxQy beschrieben werden. Kollokationspunkte stellen in diesem Zusammenhang den Ort dar, wo ein Austausch von Partikeln zwischen den Zellen erfolgen kann. Um den Aufbau eines dreidimensionalen Gitters besser verstehen zu können, kann man 4 symmetrische Untergitter definieren (siehe Abbildung 2):

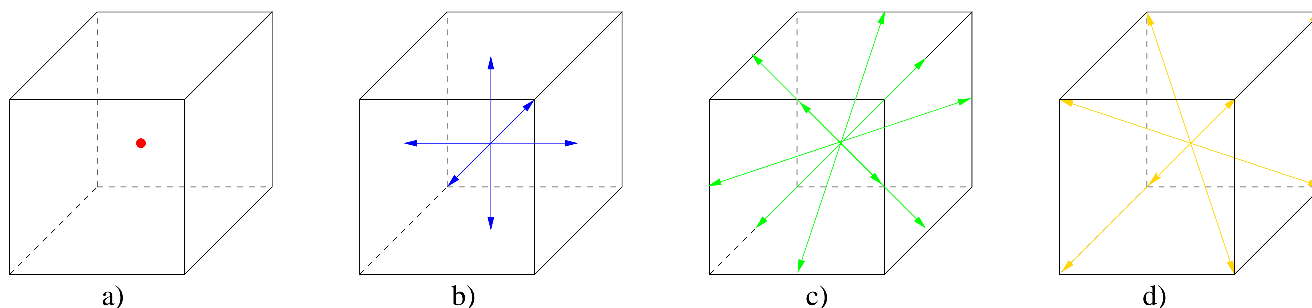

Abbildung 2: Untergitter in einem dreidimensionalen Lattice-Boltzmann Gitter.

a) Die Zelle $(0, 0, 0)$ mit dem Nullgeschwindigkeitsvektor

b) Die sechs nächsten Nachbarn $(\pm1, 0, 0)$, $(0, \pm1, 0)$, $(0, 0, \pm1)$

c) Die zwölf zweitnächsten Nachbarn $(\pm1, \pm1, 0)$, $(0, \pm1, \pm1)$, $(\pm1, 0, \pm1)$

d) Die acht drittnächsten Nachbarn $(\pm1, \pm1, \pm1)$

Durch Kombination dieser Gitter erhält man ein geeignetes Endgitter für die Simulation. Die üblichsten Gitter sind das D3Q15, D3Q19 und D3Q27 Modell. D3Q15 besteht aus den Untergittern a), b) und d) und hat am wenigsten Kollokationspunkte, was unter Umständen zu einer numerischen Instabilität führen kann. D3Q27 besteht aus allen vier Untergittern und benötigt so den höchsten Speicher- und Rechenaufwand. Ein guter Kompromiss ist das D3Q19, bestehend aus a), b) und c), zu sehen in Abbildung 3.

Wieviele Teilchen sich in eine Zelle (Knoten im Gitter) im Zeitpunkt t am Ort $\vec{x}$ mit einer bestimmten Geschwindigkeit bewegen, wird durch die Geschwindigkeitsverteilung $f_i(\vec{x}, t)$ repräsentiert. Der Index i beschreibt die Richtungsvektoren einer D-dimensionale Zelle. Bei D3Q19 gibt es also insgesamt 19 mögliche Flussrichtungen. Jede Flussrichtung bekommt eine Verteilungfunktion $f_i(\vec{x}, t)$ mit dem Index $i \in \{0, ..19\}$ zu dem jeweiligen Kollokationspunkt. Jede dieser Verteilungsfunktionen genügt nach Arie Kaufmann und Thomas Schiwietz [4, 1] an jedem diskreten Punkt den Navier-Stokes Gleichungen (1) und (2).

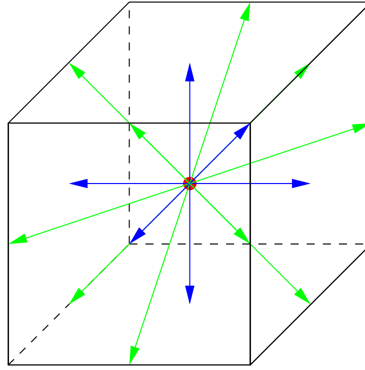

Abbildung 3: Die D3Q19-Zelle des Lattice Boltzmann Gitters.

3.2 Simulationsablauf

Bei dem Lattice-Boltzmann Modell wird die Geschwindigkeitsverteilung $f_i(\vec{x}, t)$ an jeder Gitterzelle anhand von zwei einfachen Regeln berechnet:

- **Propagation** (Die Ausbreitung von Teilchen)
- **Kollision** (Das aufeinandertreffen von Teilchen)

Die dazugehörigen Gleichungen für die Propagation (5) und für die Kollision (6) lauten:

$$f_i^{neu}(\vec{x}, t) - f_i(\vec{x}, t) = \Omega_{qi} \tag{5}$$

$$f_i(\vec{x} + \vec{e}_i, t + 1) = f_i^{neu}(\vec{x}, t) \tag{6}$$

Ω_i beschreibt hierbei den so genannten Kollisionsoperator, $\vec{e}_i$ den Richtungsvektor der Geschwindigkeit und f_i die Geschwindigkeitsverteilung am Kollokationspunkt i. Im Zeitschritt t werden die Geschwindigkeitsverteilungen in jeder Zelle anhand des gewählten Kollisionsoperators Ω ausgerechnet. Vom Zeitpunkt t zu $t+1$ bewegen sich die Teilchen anschließend entlang der Richtung $\vec{e}_i$ in die jeweilige Nachbarzelle. Die Dichte ρ und die Gesamtgeschwindigkeit $\vec{u}$ in der Zelle i sind die makroskopischen Eigenschaften einer Zelle und werden anhand der Verteilungen ausgerechnet:

$$\rho = \sum_i f_i \tag{7}$$

$$\vec{u} = \frac{1}{\rho} \sum_i f_i \vec{e}_i \tag{8}$$

Die Summe über die Geschwindigkeitsverteilungen in Gleichung (7) liefert die Dichte ρ, die Gesamtanzahl der Teilchen in der Zelle i. Die Dichte ρ und die Gesamtgeschwindigkeit $\vec{u}$ sind antiproportional zueinander. Eine große Dichte ρ bewirkt, dass die Geschwindigkeit kleiner wird. Zurückzuführen ist dieser Sachverhalt auf die mögliche Kollision und Behinderung der Teilchen untereinander in der Zelle i.
Setzt man Gleichung (5) in (6) ein, so erhält man $f_i(\vec{x} + \vec{e}_i, t + 1) - f_i(\vec{x}, t) = \Omega_i$. Ein geeigneter Operator ist der so genannte BGK-Kollisionsoperator, welcher von einer Gleichgewichtsverteilung ausgeht, die pro Zelle nur von der makroskopischen Dichte ρ und Geschwindigkeit $\vec{u}$ abhängig ist. Setzt man diesen Operator ein, so erhält man:

$$f_i(\vec{x} + \vec{e}_i, t + 1) - f_i(\vec{x}, t) = \frac{1}{\tau}(f_i(\vec{x}, t) - f_i^{eq}(\rho, \vec{u})) \tag{9}$$

τ stellt hierbei die Relaxationszeit und beschreibt die Zeit nach einer Kollision, die gebraucht wird, um Teilchen in den Normalzustand zurück zu führen. Die Gleichgewichtsverteilung ist gegeben durch:

$$f_i^{eq}(\rho, \vec{u}) = \rho(A_q + B_q(\vec{e}_i \cdot \vec{u}) + C_q(\vec{e}_i \cdot \vec{u})^2 + D_q(\vec{u})^2) \tag{10}$$

Die Koeffizienten A_q, B_q, C_q und D_q sind abhängig vom jeweiligen Gittertyp. Mit den Gleichungen (7),(8),(9) und (10) ergibt sich der in 3.2.1 beschriebene Algorithmus für die Strömungsberechnung anhand des Lattice-Boltzmann Modells.

3.2.1 LBM-Algorithmus

1. Anfangswerte, wie z.B. Dichte und Geschwindigkeit, in allen Zellen setzen.
2. Dichte und Gesamtgeschwindigkeit für jede Zelle mit den Gleichungen (7) und (8) ausrechnen.
3. Dichte und Geschwindigkeit in Gleichung (10) einsetzen, um die Gleichgewichtsverteilung zu bekommen.
4. Neue Verteilungen mit (9) durch Einsetzen der alten Verteilungen und der Gleichgewichtsverteilung ausrechnen.
5. Das Ergebnis der Gleichung (9) in die Nachbarzellen schreiben. Das entspricht der Propagation von Teilchen.

4 Strömungssimulationen auf der Grafikhardware

Strömungssimulationen lassen sich sowohl nach dem in Kapitel 3 beschriebenen Lattice-Boltzmann Modell als auch durch numerische Integration der Navier-Stokes Gleichungen [3] mit großen Performancegewinn gegenüber reiner CPU-Verarbeitung auf einer GPU ausführen, z.B. haben Xiaoming Wei und Wei Li [4] einen Speedup von über 50 im Vergleich zu einer CPU-Strömungssimulation gemessen. Die große Performancesteigerung ist vorallen daher möglich, weil die Berechnungen für eine Strömungssimulation für nahezu alle Daten immer gleich und ohne Programmsprünge auf der GPU umgesetzt werden können. Die Grafikhardware kann hier den Vorteil ihrer durch Shaderprogramme programmierbaren SIMD-Architektur für Vektordaten voll ausspielen, die Programme und Datenstrukturen müssen jedoch an die spezielle Hardware der Grafikkarte, z.B. die Organisation des GPU-Grafikspeichers oder die Zahlendarstellung, angepasst werden.
Im Folgenden wird die Umsetzung einer Strömungssimulation nach dem Lattice-Boltzmann-Modell auf der Grafikhardware beschrieben. Die Implementation nach numerischer Integration der Navier-Stokes-Gleichungen unterscheidet sich hauptsächlich durch andere Shaderprogramme, die abzuspeichernden Daten und ihre Darstellung im Grafikspeicher ist ähnlich umgesetzt.

4.1 LBM-Algorithmus auf der GPU

Um die Strömungssimulation auf der Grafikkarte ausführen zu können, wird für jede Zelle des Lattice-Boltzman-Gitters ein *Voxel* erstellt. Voxel ist die Bezeichnung für eine Volumeneinheit. Auf der Grafikhardware können Voxel z.B. durch Vertices repräsentiert werden. Für diese Voxel werden dann Shaderprogramme mit den Berechnungen für eine Zelle nach dem LBM-Modell ausgeführt. Jede Berechnung bestimmt die Veränderungen der Strömung innerhalb eines Zeitintervalls, sodass für die Simulation über einen längeren Beobachtungszeitraum mehrere Berechnungsdurchläufe nötig sind.
Damit alle Zellen unabhängig voneinander berechnet werden können, wird die Partikelpropagation (siehe Kapitel 3) für jede Zelle nur nach „innen“ berechnet, d.h. die Partikelverteilungen der Nachbarn werden in die aktuell berechnete Zelle hineinkopiert. Berechnungsergebnisse eines Durchlaufs werden an einem anderen Ort gespeichert, sodass keine race conditions entstehen. Die Arbeit kann daher parallel auf den Shadereinheiten der Grafikkarte ausgeführt werden.
An Rändern und Hindernissen sind spezielle Berechnungen nötig. Partikel können hier Abprallen, aus dem Gitter austreten oder bei *verbundenen* Rändern zur anderen Seite wechseln. Um diese Berechnungen ohne Programmsprünge und Fallunterscheidungen mit Shaderprogrammen ausführen zu können, werden die Betroffenen Voxel identifiziert und einfach mit angepassten Shaderprogrammen berechnet. Betroffene Voxel für fest stehende Hindernisse und Ränder können im Vorfeld der LBM-Simulation bestimmt werden und ändern sich nicht mehr während des Ablaufs der Simulation. Bei sich verformenden oder bewegenden Hindernissen müssen die Voxel, für die gesonderte Shaderprogrammme auszuführen sind, nach jedem Durchlauf neu gesucht werden. In der Literatur wird dies mit *Voxelisieren* bezeichnet. Ein dafür verwendetes Verfahren wird im Abschnitt 4.3 vorgestellt.
Die Daten für die Berechnung einer Zelle können in Pixeln/Texeln einer Textur gespeichert werden. Texel einer Textur können von einem Shaderprogramm beliebig interpretiert und von Fragmentshader aus mit wahlfreiem Zugriff gelesen werden. Da z.B. für die Propagationsberechnungen wie oben beschrieben Zugriff auf die Daten anderer Zellen notwendig ist und mehr Fragmentshadereinheiten als Vertexshadereinheiten auf der Grafikkarte zur Verfügung stehen, bietet sich die Verwendung von Fragmentshadern an. Die Verarbeitung, welche die Grafikkarte pro Durchlauf und Zelle ausführt, ist in der Abbildung 4 dargestellt. Angestoßen wird die Verarbeitung durch die Eingabe des Vertex-Array und der Daten-Textur.

Ergebnisse einer Berechnung werden in den Frame Buffer bzw. eine Ergebnistextur geschrieben und müssen für den nächsten Durchlauf der Berechnung über die *copy-to-texture*, oder direkt per *render-to-texture*-Funktion der

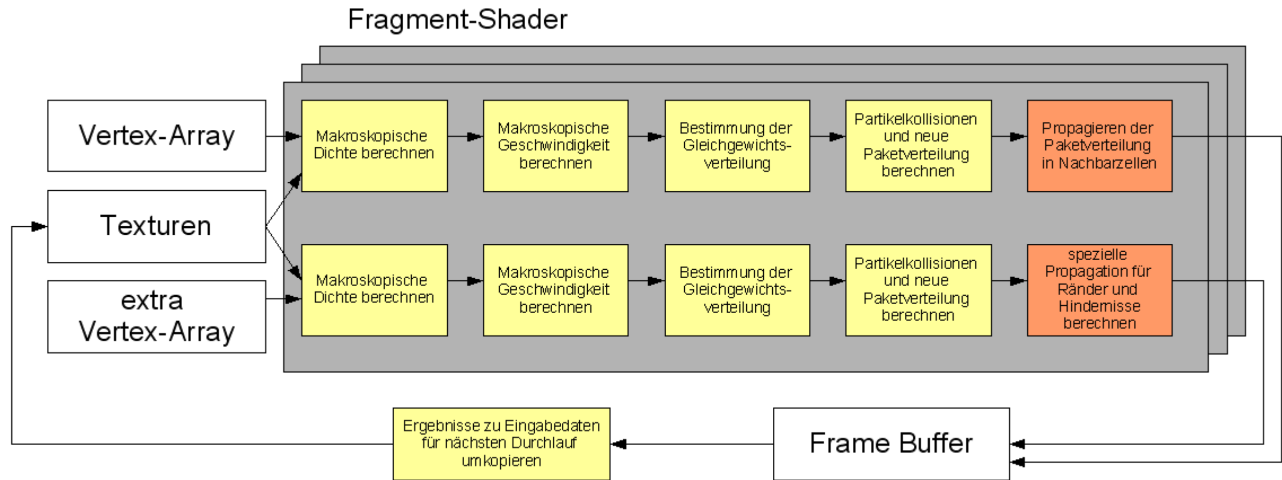

Abbildung 4: LBM-Algorithmus und seine Umsetzung auf der Grafikpipeline.

Grafikkarte in die Anfangstextur geschrieben werden. Auf die Darstellung der Daten im Texturspeicher wird im folgenden eingegangen.

4.2 Repräsentation der Daten im Texturspeicher

Wie bereits erwähnt, laden die Shaderprogramme die für die Berechnungen notwendigen Daten aus einer Textur. Für eine D3Q19 LBM-Simulation müssen pro Zelle 19 Partikelverteilungen, eine pro Nachbar der Zelle sowie eine für die Zelle selbst (siehe Kapitel 3), die makroskopische Dichte ρ und die makroskopische Geschwindigkeit $\vec{u}$ der Zelle abgespeichert werden. Die makroskopische Geschwindigkeit $\vec{u}$ ist im D3 ein dreidimensionaler Vektor, so das insgesamt $19 + 1 + 3 = 23$ Skalardaten gespeichert werden müssen.
Jeder Pixel/Texel einer Textur besitzt vier 8-Bit Kanäle für Rot, Grün, Blau und den Alphakanal, auf welche Shaderprogramme zugreifen können. Um alle Daten einer Zelle zu speichern sind daher 6 Texel nötig, wobei der letzte Kanal des letzten Texels ungenutzt bleibt (Siehe Tabelle 4.2). Pro Zelle entsteht so ein Speicherbedarf von 24 Byte. Eine dreidimensionale LBM-Simulation mit $256 * 256 * 128$ Zellen benötigt daher bereits 192 MB Speicher. Die Daten aller Zellen können in einer großen Textur gespeichert werden.

R	G	B	A
v_x	v_y	v_z	ρ
$f(1,0,0)$	$f(-1,0,0)$	$f(0,1,0)$	$f(0,-1,0)$
$f(1,1,0)$	$f(-1,-1,0)$	$f(1,-1,0)$	$f(-1,1,0)$
$f(1,0,1)$	$f(-1,0,-1)$	$f(1,0,-1)$	$f(-1,0,1)$
$f(0,1,1)$	$f(0,-1,-1)$	$f(0,1,-1)$	$f(0,-1,1)$
$f(0,0,1)$	$f(0,0,-1)$	$f(0,0,0)$	*leer*

Tabelle 1: Simulationsdaten einer Zelle, gespeichert in 6 Texeln

Die Einträge $f(x,y,z)$ der Tabelle stehen dabei für die Partikelverteilung am Kollokationspunkt (x,y,z) einer Zelle. Vor dem Speichern der Daten muss berücksichtigt werden, dass die GPU herstellerspezifische Festkomma-Zahlen verwendet. Z.B. 8-Bit Festkomma-zahlen im Intervall von $[-1,1]$ bei einer NVIDIA 5900 GPU [4]. Um Buffer-Overflows zu vermeiden müssen die Anfangswerte für die Partikelverteilung eines Lattice daher entsprechend skaliert werden.
Da die GPU für die Verarbeitung von zweidimensionalen Texturen optimiert ist und in jedem Renderingdurchlauf nur eine geringe Anzahl von Texturen geschrieben werden kann, bietet sich für das Abspeichern der Daten aller Zellen eine große zweidimensionale Textur an. So können außerdem häufige Texturewechsel vermieden werden.
Die dreidimensionalen Koordinaten einer Zelle (x,y,z) müssen daher auf zweidimensionale Koordinaten der Textur (u,v) umgerechnet werden. Ein dreidimensionales Volumen mit den Abmessungen W und H wird in d Scheiben aufgeteilt. Die Umrechnung kann dann mit den Gleichungen (11) und (12) vorgenommen werden (siehe Abbildung

5).

$$u = (z \ mod \ d) * W + x \tag{11}$$

$$v = floor(\frac{z}{d}) * H + y \tag{12}$$

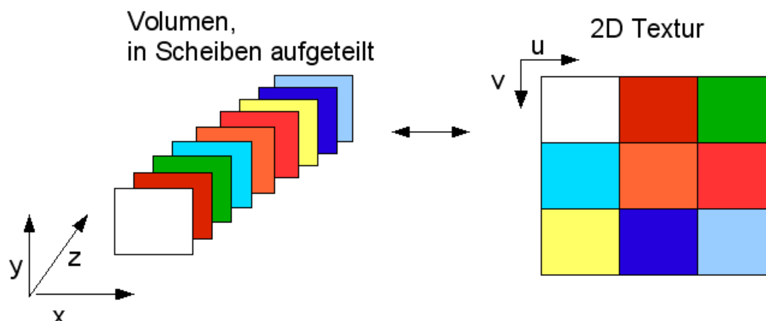

Abbildung 5: Abspeichern dreidimensional angeordneter LBM-Zellen in einer zweidimensionalen Textur.

4.3 Voxilisation

In Kapitel 47 der GPU GEMS [2] wird für die Voxilisation ein Algorithmus basierend auf *depth peeling* vorgestellt, mit dem die Voxel, die Teile eines Hindernisses oder dem Simulationsrand enthalten, identifiziert werden können. Der Algorithmus wird vollständig auf der GPU durch Fragment- und Vertextshaderprogramme ausgeführt und muß insbesondere für sich bewegenden oder verformende Objekte vor jedem Berechnungsdurchlauf der LBM-Simulation angewendet werden.
Die Idee des Algorithmus ist, dass Oberflächen von Objekten „geschält“ werden, und so jede einzelne Schicht identifiziert werden kann. Dazu wird die Szene zunächst so durch ein Fragmentshaderprogramm gerendert, dass für alle sichtbaren Oberflächen Pixel in den Frame Buffer geschrieben werden. In den Pixeln sind die Attribute wie z.B. Position für die betroffenen Voxel, gespeichert. Zusätzlich zum Rendern von Pixeln in den Frame Buffer werden die Tiefeninformation der sichtbaren Objekte in einer Textur abgespeichert. In der folgenden Phase wird die am nächstem liegende Schicht „abgeschält“ indem die Szene erneut gerendet wird. Dabei werden aber nur noch Objekte berücksichtigt, deren Tiefeninformation größer als die in der Tiefentextur gespeicherten Daten sind. Das Abschälen wird solange fortgesetzt, bis keine Pixel mehr den Frame Buffer geschrieben werden. Die Pixel können nun in ein Vertex-Array kopiert und von einem Vertexshaderprorgamm verarbeitet werden, dass daraus neue Vertices für die Berechnung der Simulation erstellt, wie in 4.1 dargestellt. Damit keine Voxel ausgelassen werden, wendet man den oben beschriebenen Algorithmus auf alle drei Ebenen an. Durch das dreifache Render können für einzelne Voxel vielfache Vertices erstellt werden, wodurch die Berechnungen für die entsprechende Zelle der LBM-Simulation ebenfalls mehrfach ausgeführt wird. Das mehrfache Berechnen einer Zelle verfälscht die Ergebnisse der Simulation jedoch nicht, da in dem beschriebenen Verfahren die Eingangsdaten während eines Durchlaufs der Simulation nicht überschrieben werden.

5 Einsatzgebiete

Strömungssimulationen wie oben beschrieben können beispielsweise für folgende Bereiche verwendet werden:

- Simulation von Frischluftströmungen oder Giftstoffausbreitung in bebauten Gebieten
- Temperaturströmungen für die Wettervorhersage
- Fahrtluftströmungssimulation beim Fahr- und Flugzeugentwurf
- Simulation von Strömungen in Filmen und Computerspielen

Dabei kann zwischen Anwendungen mit Echtzeitanforderungen, wie in Computerspielen, und Anwendungen bei denen eine schnellere Simulationen zu effizienteren Arbeiten führt, unterschieden werden. Durch höhere Verarbeitungsgeschwindigkeit können Entwickler und Ingenieure z.B. Fahrtlufströmungssimulationen häufiger durchführen. So können die Auswirkungen kleinerer Änderungen öfter geprüft, und Fehlentscheidungen vermieden werden. Computerspiele profitieren von einer Leistungssteigerung bei Strömungssimulationen durch realistischere Effekte.
Allgemein erschließen sich mit der Geschwindigkeit der Simulation auch weitere Anwendungsgebiete, z.B. Wetterberechnungen für detailliertere Karten. In der Arbeit von Xiaoming Wei und Wei Li [4] wird auf einem Einprozessorsystem ein Speedup von über 50 im Vergleich zu einer Softwarelösung gemessen. Um Strömungssimulationen mit größerem Volumen berechnen zu können haben Arie Kaufman und Zhe Fan [1] ein Cluster mit leistungsfähigen Grafikkarten verwendet. Dafür wurde das LBM-Gitter in Bereiche aufgeteilt und zur Berechnung auf die einzelnen Knoten verteilt. Im Vergleich zur der reinen CPU-Verarbeitung auf dem Cluster erzielte die Berechnung auf der GPUs ein Speedup von über 6. Zum Zeitpunkt des Aufbaus des Clusters standen nur AGP-Bus Grafikkarten zur Verfügung. Da nach jedem Durchlauf des LBM Daten zwischen den Knoten ausgetauscht werden müssen ist hier vorallem die Übertragungsbandbreite von nur 133 MB/s von der GPU zum Speicher auf dem AGB-Bus ein Flaschenhals gewesen. Mit heutigen Grafikkarten mit PCI-Express-Anschluss wären sicher höhere Speedups möglich. Allgemein sind die Simulationen jedoch noch durch den hohen Speicherbedarf beschränkt. Um möglichst genaue LBM-Simulationen durchzuführen, sollte ein zu simuliertes Volumen durch möglichst vielen Zellen eines LBM-Gitters dargestellt werden. Pro Zelle werden im einfachsten Fall 6 Texel mit zusammen 24 Byte benötigt. Ein $400x400x80$ Gitter benötigt demnach ca. 300 MB Texturspeicher. Der Größe und dem Einsatzgebiet einer Simulation sind daher momentan noch enge Grenzen gesetzt, welche aber sicher durch die weitere rasante Entwicklung von GPUs immer mehr aufgeweicht wird.

6 Zusammenfassung und Ausblick

Diese Ausarbeitung hat einen kurzen Überblick über die physikalischen Grundlagen von Strömungen und der Implementation von Strömungssimulationen nach dem Lattice-Boltzmann Modell auf der GPU gegeben.
GPU-Implementierungen verwenden Texturen zur Speicherung der Daten und Vertices, die mit Fragmentshaderprogrammen gerendert werden, für die Durchführung der Berechnung. Bereits heute sind so beeindruckende Speedups möglich.
Es ist zu erwarten, dass der Geschwindigkeitsvorteil von GPU Implementierungen solcher Verfahren gegenüber CPUs immer größer wird. Für Anwendungen mit geringeren Genauigkeitsanforderungen, z.B. graphischen Darstellung von Strömung in Filmen, lohnt sich der Einsatz bereits heute. Falls im Rahmen der fortschreitenden Hardwareentwicklung die momentan niedrige Fließkommagenauigkeit und Speicherknappheit beseitigt werden, ist auch der Einsatz von GPUs in physikalisch korrekten Simulation,z.B. Strömungssimulation für Fahrzeugbau, denkbar.

Literatur

[1] Arie Kaufman Zhe Fan and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. 2004.

[2] Xiaoming Wei Wei Li Zhe Fan and Arie Kaufman. *GPU GEMS Chapter 47, Flow Simulation with Complex Boundaries.* 2005.

[3] Mark J. Harris. *GPU GEMS Chapter 38, Fast Fluid Dynamics Simulation on the GPU.* 2004.

[4] Xiaoming Wei Wei Li and Arie Kaufman. Implementing lattice boltzmann computation on graphics hardware. 2003.

[5] Thomas Schiwietz. Echtzeitfähige Simulation von Wasser auf Grafikhardware. Master's thesis, Technische Universität München, Fakultät für Informatik, 2003.