



INSTITUT FÜR INFORMATIK  
AG MEDIENINFORMATIK

*Diplomarbeit*

# Echtzeit-Simulation von Windfeldern mit OpenCL zur Modellierung von Schneefall

Philipp Middendorf

November 2012

Erstgutachter: Prof. Dr. Oliver Vornberger  
Zweitgutachterin: Prof. Dr. Sigrid Knust



## **Danksagungen**

Hiermit möchte ich allen Personen danken, die mich bei der Erstellung der Arbeit unterstützt haben:

- Herrn Prof. Dr. Oliver Vornberger für die Tätigkeit als Erstgutachter und für die Bereitstellung der interessanten Thematik.
- Frau Prof. Dr. Sigrid Knust, die sich als Zweitgutachterin zur Verfügung gestellt hat.
- Frau Jana Lehnfeld und Herrn Henning Wenke für das wertvolle Feedback.



## **Zusammenfassung**

Die vorliegende Arbeit entstand in der Arbeitsgruppe Medieninformatik an der Universität Osnabrück im Bereich Computergrafik und beschäftigt sich mit der Simulation von Schneefall.

Um Schneeflocken realistische Flugbahnen zu geben, wird mit Hilfe des OpenCL-Frameworks ein Verfahren aus der numerischen Strömungsmechanik auf der Grafikkarte umgesetzt. Das Verfahren erzeugt ein Feld, welches Richtung und Stärke des Windes an jedem Punkt im Simulationsbereich enthält. Mit Hilfe dieses Feldes wird der Einfluss des Windes auf die Schneeflocken berechnet.

Die Flocken werden als Partikelsystem modelliert und mit Hilfe von OpenGL dargestellt. Geschwindigkeit und Position der Partikel werden direkt auf der Grafikkarte mit Hilfe von OpenCL verändert. In die Berechnung der auf die Flocken wirkenden Kräfte fließen Schwerkraft, Windgeschwindigkeit und Auftrieb ein.

Durch das Auftreffen der Flocken auf ein Hindernis wird zudem die dynamische Generierung einer Schneedecke mit Hilfe des Marching-Cube-Algorithmus angestoßen. Das daraus entstehende Dreiecksmesh wird mit Hilfe triplanarer Texturierung mit einer optisch ansprechenden Oberfläche versehen.



# Inhaltsverzeichnis

|   |           |
|---|-----------|
| <b>1 Einleitung</b>   | <b>8</b>  |
| 1.1 Motivation . . . . .                                    | 8         |
| 1.2 Aufbau . . . . .  | 9         |
| <b>2 Mathematische Grundlagen</b>                           | <b>11</b> |
| 2.1 Kontinuierliche Felder . . . . .                        | 11        |
| 2.2 Diskretisierung . . . . .                               | 13        |
| <b>3 Die Navier-Stokes-Gleichungen</b>                      | <b>17</b> |
| 3.1 Fluide . . . . .  | 17        |
| 3.2 Einführung . . . . .                                    | 17        |
| 3.3 Die Impulsgleichung . . . . .                           | 18        |
| 3.3.1 Lagrange und Euler . . . . .                          | 18        |
| 3.3.2 Die substantielle Ableitung . . . . .                 | 19        |
| 3.3.3 Die Kräfte in den Navier-Stokes-Gleichungen . . . . . | 20        |
| 3.4 Die Unkomprimierbarkeitsbedingung . . . . .             | 22        |
| <b>4 Methode nach Stam</b>                                  | <b>23</b> |
| 4.1 Fluideynamik in interaktiven Anwendungen . . . . .      | 23        |
| 4.2 Motivation . . . . .                                    | 24        |
| 4.3 Überblick über das Verfahren . . . . .                  | 25        |
| 4.4 Advektion . . . . .                                     | 27        |
| 4.5 Äußere Kräfte . . . . .                                 | 30        |
| 4.5.1 Gravitation . . . . .                                 | 30        |
| 4.5.2 Wirbelstärkenerhaltung . . . . .                      | 31        |
| 4.6 Projektion . . . . .                                    | 31        |
| 4.6.1 Einleitung . . . . .                                  | 31        |
| 4.6.2 Lösung des Poissonproblems . . . . .                  | 32        |
| 4.6.3 Das Jacobiverfahren . . . . .                         | 33        |
| 4.6.4 Randbedingungen in Differentialgleichungen . . . . .  | 33        |
| 4.6.5 Randbedingungen in der Simulation . . . . .           | 34        |
| <b>5 OpenCL</b>   | <b>38</b> |
| 5.1 Einleitung . . . . .                                    | 38        |
| 5.2 Geschichtliches . . . . .                               | 38        |
| 5.3 Arten der Parallelität . . . . .                        | 39        |
| 5.4 Architektur . . . . .                                   | 40        |
| 5.4.1 Platform-Model . . . . .                              | 41        |
| 5.4.2 Execution-Model . . . . .                             | 42        |
| 5.4.3 Memory-Model . . . . .                                | 43        |
| 5.4.4 OpenCL-C . . . . .                                    | 46        |
| 5.5 Beispielcode . . . . .                                  | 49        |
| <b>6 OpenGL</b>   | <b>52</b> |

|  |            |
|--|------------|
| <b>7 Windsimulation</b>                          | <b>54</b>  |
| 7.1 Einleitung . . . . .                         | 54         |
| 7.2 Wahl der Datenstruktur . . . . .             | 54         |
| 7.3 Verfahren nach Stam in OpenCL . . . . .      | 55         |
| 7.3.1 Allgemeines . . . . .                      | 55         |
| 7.3.2 Advektion . . . . .                        | 56         |
| 7.3.3 In- und Outflow . . . . .                  | 59         |
| 7.3.4 Projektion . . . . .                       | 60         |
| <b>8 Fallender Schnee</b>                        | <b>67</b>  |
| 8.1 Hintergründe . . . . .                       | 67         |
| 8.2 Schnee in interaktiven Anwendungen . . . . . | 68         |
| 8.3 Visualisierung . . . . .                     | 70         |
| 8.4 Modellierung . . . . .                       | 74         |
| 8.4.1 Einleitung . . . . .                       | 74         |
| 8.4.2 Luftwiderstand . . . . .                   | 75         |
| 8.4.3 Dynamischer Auftrieb . . . . .             | 78         |
| 8.4.4 Statischer Auftrieb . . . . .              | 80         |
| 8.4.5 Code . . . . .                             | 81         |
| <b>9 Liegengebliebener Schnee</b>                | <b>87</b>  |
| 9.1 Einleitung . . . . .                         | 87         |
| 9.2 Schneedecke mit Texturen . . . . .           | 87         |
| 9.3 Schneedecke als Mesh . . . . .               | 89         |
| 9.3.1 Einleitung . . . . .                       | 89         |
| 9.3.2 Aufbau der Schneedichte . . . . .          | 90         |
| 9.3.3 Der Marching Cubes-Algorithmus . . . . .   | 90         |
| 9.3.4 Normalen und Texturen . . . . .            | 91         |
| 9.3.5 Implementierungen . . . . .                | 95         |
| 9.3.6 GPU-Implementierung . . . . .              | 95         |
| 9.3.7 CPU-Implementierung . . . . .              | 96         |
| 9.4 Vergleich . . . . .                          | 97         |
| <b>10 Hindernisse in der Simulation</b>          | <b>99</b>  |
| 10.1 Einfacher Ansatz . . . . .                  | 99         |
| 10.2 Komplexe Hindernisse . . . . .              | 100        |
| <b>11 Reflexion</b>                              | <b>102</b> |
| 11.1 Ausblick . . . . .                          | 102        |
| 11.2 Fazit . . . . .                             | 102        |

# 1 Einleitung

## 1.1 Motivation

Die Idee, Schneefall zu modellieren, entstand im Rahmen der Masterprojektgruppe „virtueller Campus“ an der Universität Osnabrück im Wintersemester 2011/12. Die Projektgruppe befasste sich mit dem Nachbau des Campus am Westerberg der Universität Osnabrück unter Anwendung aktueller Methoden der Computergrafik.

Zur Zielsetzung der Projektgruppe gehörte auch die Visualisierung von natürlichen Phänomenen wie einem dynamischen Wechsel von Tag und Nacht sowie wechselnde Wetterverhältnisse. In diesem Zusammenhang entstand die Idee, die Modellierung von Schneefall auszulagern und separat von der Projektgruppe zu entwickeln. Daraus entstanden zwei Arbeiten: die vorliegende und die Bachelorarbeit von Manuel Schwarz ([39]).

Die Modellierung des Phänomens Schnee kann in drei Teile aufgeteilt werden:

1. Die Schneeflocken müssen möglichst realitätsnah nachgebildet werden. Dazu gehört zum einen der optische Eindruck der einzelnen Flocken von Nahem wie von Weitem, zum Anderen sollte das Verhalten der Flocken in der Luft realistisch wirken.
2. Da Schneeflocken größtenteils vom Wind fortgetragen werden, müssen innerhalb des Simulationsbereiches Informationen über die Richtung und Stärke des Windes an jedem Punkt vorhanden sein.
3. Fällt der Schnee stetig zu Boden, bildet sich bei niedrigen Temperaturen eine Schneedecke, die mit der Zeit an Umfang zunimmt.

Die Arbeit von Schwarz befasst sich mit Punkt 3, also den verschiedenen Möglichkeiten, eine dynamisch wachsende und schrumpfende Schneedecke zu modellieren. Als Resultat der Arbeit entstand ein Verfahren, welches Kollisionen von Schneeflocken mit der Umgebung zählt und daraus eine dreidimensionale Schneedecke erzeugt. Auch das Schmelzen von Schnee und die Bildung von Lawinen bei zu großen Schneehügeln ist in dem Modell berücksichtigt. Die Visualisierung der Schneedecke stand in dieser Arbeit hingegen nicht im Vordergrund.

Die vorliegende Arbeit beschäftigt sich vornehmlich mit den Punkten 1 und 2, also der Modellierung von fallendem Schnee. Es wird ein Partikelsystem verwendet, bei dem die Schneeflocken einzeln auf der Grafikkarte mit ihren Eigenschaften gespeichert sind und für sich genommen bearbeitet werden. Um den Einfluss des Windes möglichst realistisch nachzubilden, wird mit Hilfe strömungsmechanischer Verfahren ein dreidimensionales Feld berechnet, welches die Windstärke enthält.

Für die Berechnung dieses Windfeldes werden Verfahren benötigt, welche auf stark parallel arbeitende Systeme zugeschnitten sind. Sie lassen sich auf heutigen CPUs nicht in Echtzeit umsetzen. Daher wurde das OpenCL-Framework eingebunden, um die Berechnungen auf die Grafikkarte (GPU) zu verlagern, deren parallele Architektur eine effiziente Implementierung der Verfahren erlaubt.

Für die Bewegung der Schneeflocken wurde auf einen physikalisch motivierten Ansatz aus der Masterarbeit von Aaagaard ([1]) zurückgegriffen, welcher auch die Eigenarten der

Flockenbewegungen bei Windstille nachbildet. Für den optischen Eindruck wurden zweidimensionale Pointsprites verwendet, die mit Hilfe des OpenGL-Frameworks gerendert werden.

Im Laufe der Implementierung wurde die Integration der Ergebnisse von Schwarz als weiteres Ziel hinzugefügt, um die gefallenen Schneeflocken in die Bildung der Schneedecke einfließen zu lassen.

## 1.2 Aufbau

Da sowohl die Berechnungen für die Bewegung der Schneeflocken als auch die strömungsmechanischen Gleichungen mathematische Grundkenntnisse voraussetzen, werden im Abschnitt 2 einige mathematische Definitionen und Verfahren eingeführt.

Die Behandlung der strömungsmechanischen Zusammenhänge ist in drei Abschnitte aufgeteilt:

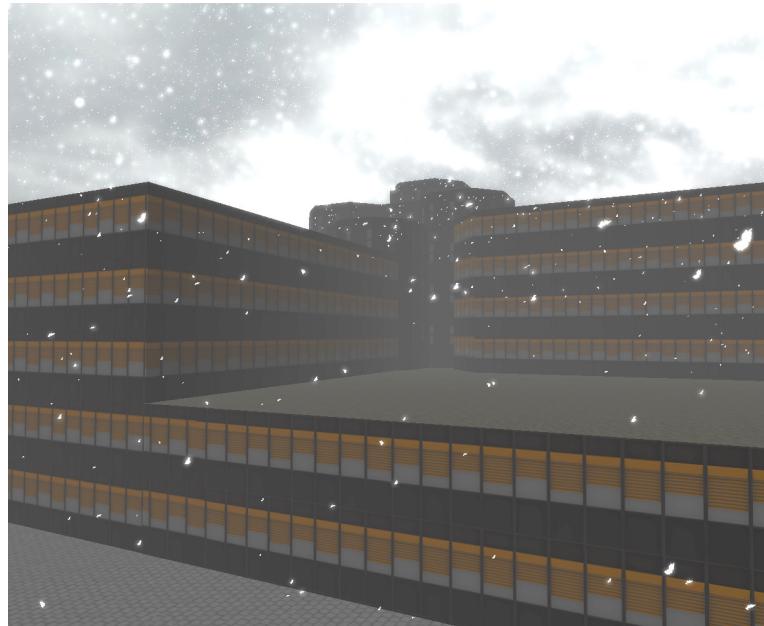
**Abschnitt 3** gibt eine Einführung in die Navier-Stokes-Gleichungen. Diese Gleichungen sind sehr allgemein gefasst und beschreiben jegliche Art von Strömungen. Um die Erklärung nicht zu kompliziert zu gestalten wird eine einfachere Form der Gleichungen erläutert. Zudem werden nur die Teile näher beleuchtet, die für die Modellierung von Wind eine Rolle spielen. Der Abschnitt soll vor allem ein intuitives Verständnis der einzelnen Bestandteile geben.

**Abschnitt 4** stellt das Verfahren von Stam vor. Zuerst wird ein Überblick über die bisherigen Ansätze in der interaktiven Strömungsmechanik gegeben. Danach wird erläutert, inwiefern sich Stams Verfahren von den bisherigen unterscheidet und was es auszeichnet.

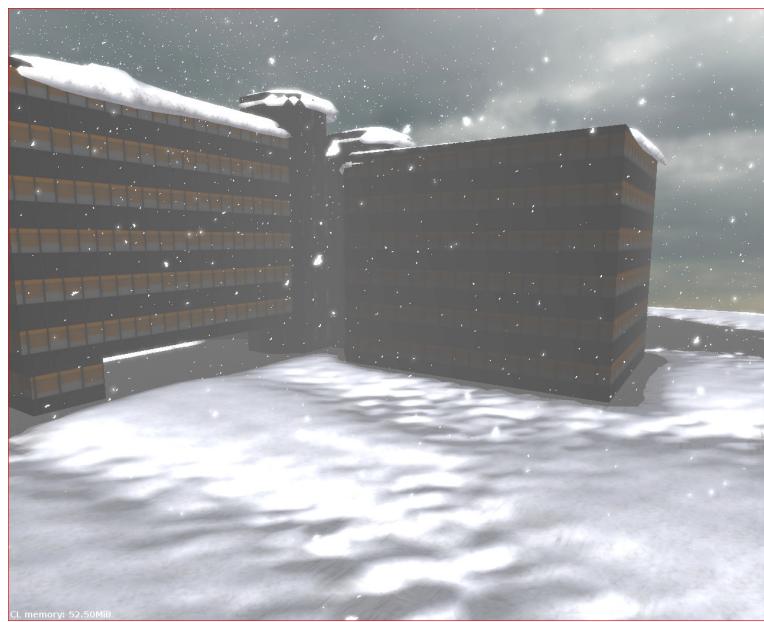
**Abschnitt 7** setzt das in Abschnitt 5 beschriebene OpenCL-Framework ein, um das Verfahren von Stam auf die Grafikkarte zu übertragen. Es wird auf die Schwierigkeiten in der Implementierung und auf Performanceoptimierungen eingegangen.

Die Implementierung des Partikelsystems findet sich in Abschnitt 8. Dort wird sowohl die Visualisierung mit Hilfe des in Abschnitt 6 eingeführten OpenGL-Frameworks besprochen als auch die physikalischen Zusammenhänge und die Implementierung in OpenCL.

Abschnitt 9 behandelt schließlich die Modellierung der Schneedecke. Neben dem Ansatz von Schwarz wird eine Alternative vorgestellt, die auf Texturen basiert. Die dreidimensionale Schneedecke wird außerdem in zwei Varianten erläutert, zum einen auf der GPU, zum anderen auf der CPU. Am Ende des Abschnitts folgt ein Vergleich der beiden Implementierungen.



**Abbildung 1.1:** Schneegestöber, hier mit etwa 32.000 Schneeflocken und hoher Windgeschwindigkeit. Eine Schneedecke hat sich noch nicht gebildet.



**Abbildung 1.2:** Bildung einer Schneedecke nach einigen Minuten.

## 2 Mathematische Grundlagen

Die Berechnungen in dieser Arbeit finden alle im reellen Raum statt, also in den Vektorräumen  $\mathbb{R}^n, n \in \{1, 2, 3\}$ . Die Gleichungen der Strömungsmechanik benutzen **Vektorfelder** und **Skalarfelder**, daher werden diese Begriffe und die zugehörigen Operationen zunächst eingeführt.

Die später betrachteten Gleichungen sind allerdings meist gar nicht oder nur sehr schwer analytisch zu lösen, daher greift man auf numerische Verfahren zurück. Dafür ist es notwendig, Operationen wie den Gradienten oder die Divergenz zu *diskretisieren*. Im letzten Teil des Abschnitts werden verschiedene Arten der Diskretisierung erläutert.

### 2.1 Kontinuierliche Felder

Für eine Funktion  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  bezeichne

$$\frac{\partial f}{\partial x_i} \quad (2.1)$$

die **partielle Ableitung** nach der  $i$ -ten Komponente.

Ein **Vektorfeld**  $\vec{u}$  ist eine Abbildung

$$\vec{u}: \mathbb{R}^n \rightarrow \mathbb{R}^n,$$

die einem Punkt  $\vec{x}$  im Raum einen Pfeil  $\vec{u}(\vec{x})$  zuordnet. So kann z. B. jedem Punkt im Raum die Windgeschwindigkeit an diesem Punkt zugeordnet werden. Abbildung 2.1a zeigt ein solches Vektorfeld.

$\text{Vect}(\mathbb{R}^n)$  bezeichne die Menge aller Vektorfelder. In dieser Arbeit betrachtete Vektorfelder sind stets differenzierbar.

Analog ist ein **Skalarfeld**  $p$  eine Abbildung

$$p: \mathbb{R}^n \rightarrow \mathbb{R},$$

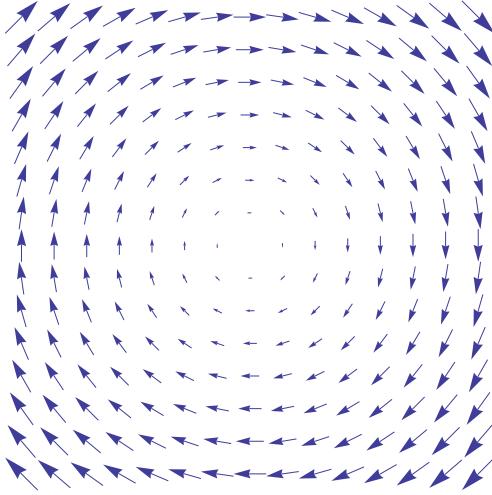
welche einem Punkt  $x$  im Raum einen skalaren Wert  $p(x)$  zuordnet. Beispielsweise könnte für jeden Punkt im Raum die dortige Temperatur in einem Skalarfeld gespeichert sein (siehe Abbildung 2.1b).  $\text{Scal}(\mathbb{R}^n)$  bezeichne die Menge aller Skalarfelder.

Auch alle Skalarfelder in dieser Arbeit sind differenzierbar.

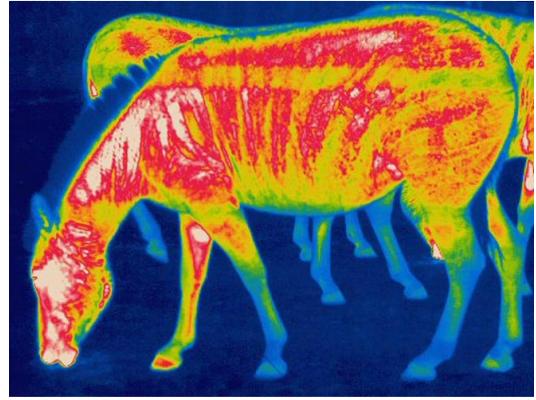
Sei  $p$  ein dreidimensionales Skalarfeld, dann ist der **Gradient**  $\text{grad } p$  definiert als:

$$\begin{aligned} \text{grad}: \text{Scal} &\rightarrow \text{Vect} \\ \text{grad } p &= \left( \frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z} \right) \end{aligned} \quad (2.2)$$

Der Gradient eines Skalarfeldes ist also ein Vektorfeld, nämlich das Vektorfeld der partiellen Ableitungen. Genau wie die Ableitung eines Graphen die Steigung an jedem Punkt angibt, so zeigt der Gradient eines Feldes in die Richtung des steilsten Anstiegs, wobei die Länge des Gradienten die Größe der Steigung in diesem Punkt angibt. Der Gradient des negativen Skalarfeldes  $-p$  zeigt folglich in Richtung des größten Gefälles. Abbildung 2.2 zeigt ein simples Beispiel für den Gradienten.



(a) Ein Vektorfeld



(b) Die Körpertemperatur eines Pferds als Skalarfeld.

Abbildung 2.1: Die betrachteten Feldtypen

Für ein differenzierbares Vektorfeld  $\vec{u}$  ist die **Divergenz** auf dem Feld definiert als

$$\begin{aligned} \text{div: Vect} &\rightarrow \text{Scal} \\ \text{div } \vec{u} &= \frac{\partial u^x}{\partial x} + \frac{\partial u^y}{\partial y} + \frac{\partial u^z}{\partial z} \end{aligned} \quad (2.3)$$

Hier sind  $u^x, u^y, u^z$  die einzelnen Komponenten des Vektorfelds. Die Divergenz eines Vektorfelds ist also ein Skalarfeld. Dieses Skalarfeld kann gedeutet werden als „Quellendichte“ des ursprünglichen Vektorfelds. Ein Feld  $\vec{u}$ , welches  $\text{div } \vec{u} = 0$  erfüllt, heißt folglich **quellenfrei**. Auf Vektorfeldern ist schließlich die **Rotation** definiert als:

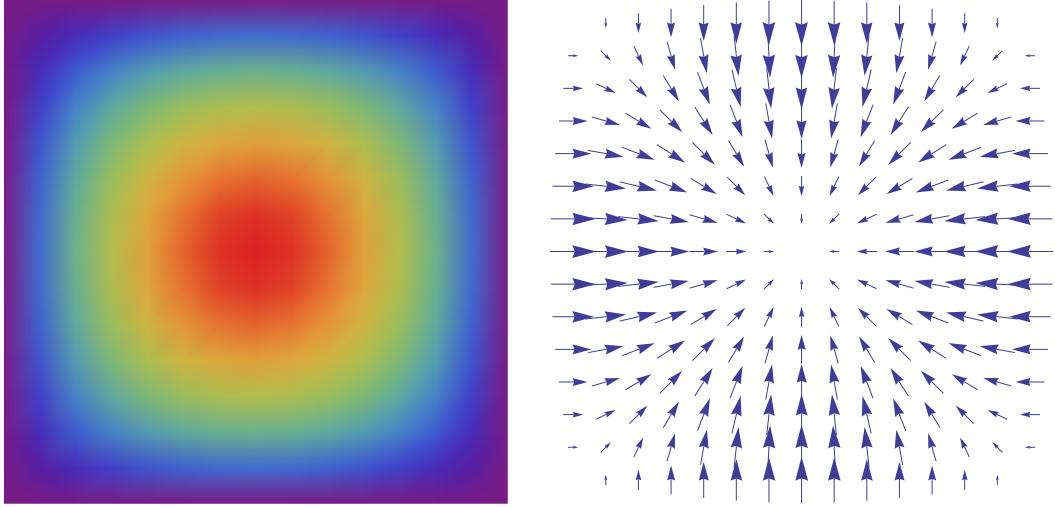
$$\begin{aligned} \text{rot: Vect} &\rightarrow \text{Vect} \\ \text{rot } \vec{u} &= \left( \begin{array}{c} \frac{\partial u^z}{\partial y} - \frac{\partial u^y}{\partial z} \\ \frac{\partial u^x}{\partial z} - \frac{\partial u^z}{\partial x} \\ \frac{\partial u^y}{\partial x} - \frac{\partial u^x}{\partial y} \end{array} \right) \end{aligned} \quad (2.4)$$

Wie der Name schon andeutet, gibt sie für jeden Punkt des ursprünglichen Vektorfelds an, wie stark und in welche Richtung das Feld in diesem Punkt rotiert. Im Zweidimensionalen ist die Rotation auch definiert, hier ergibt sich für jeden Punkt im Raum allerdings nur ein Skalar, kein Vektor. Die Rotationsachse ist immer die imaginäre „z-Achse“. Abbildung 2.3 zeigt ein Skalarfeld und die dazugehörige Rotation.

Ein Feld  $\vec{u}$ , welches  $\text{rot } \vec{u} = 0$  erfüllt, heißt **rotationsfrei**.

Schaltet man grad und div hintereinander, erhält man den **Laplace-Operator**:

$$\begin{aligned} \Delta: \text{Scal} &\rightarrow \text{Scal} \\ \Delta p &= \text{div grad } p \end{aligned} \quad (2.5)$$



(a) Das Skalarfeld zur Funktion  $f(x, y) = \sin(x)\sin(y)$ . Rötliche Färbung deutet hohe Werte an, violett und blau geringe.

(b) Der Gradient des Skalarfeldes  $f(x, y) = (\sin(y)\cos(x), \sin(x)\cos(y))$

Abbildung 2.2: Der Gradient am Beispiel

Im Dreidimensionalen ergibt sich:

$$\Delta p = \frac{\partial^2 p}{\partial^2 x} + \frac{\partial^2 p}{\partial^2 y} + \frac{\partial^2 p}{\partial^2 z} \quad (2.6)$$

Intuitiv gibt der Operator für jeden Punkt  $\vec{x}$  die Abweichung von  $p(\vec{x})$  zu Punkten in seiner Umgebung an. Die diskretisierte Variante dieses Operators wird deswegen in der Bildverarbeitung zum Erkennen von Kanten in Bildern verwendet (siehe Abbildung 2.4b).

Der Operator lässt sich auf nahe liegende Weise auf Vektorfelder erweitern, in drei Dimensionen ergibt sich:

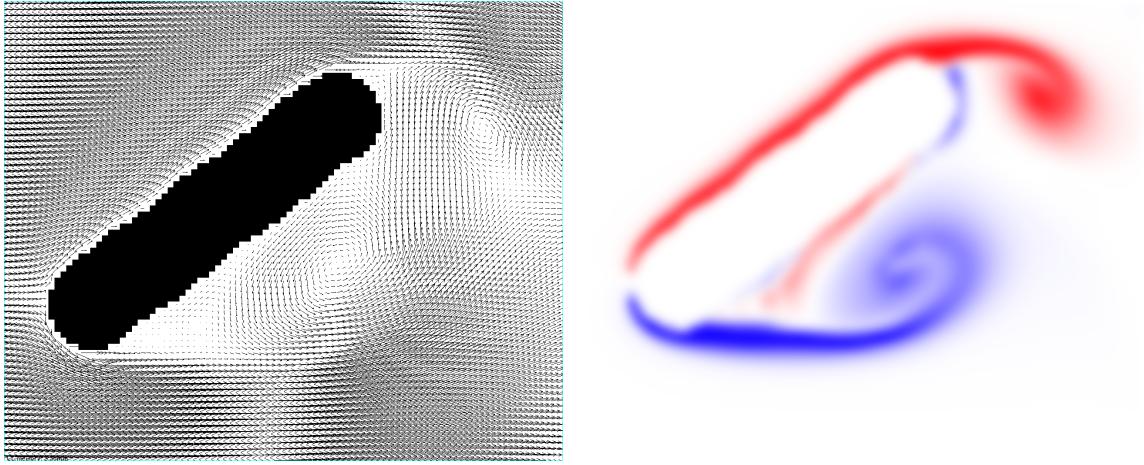
$$\Delta \vec{u} = \begin{pmatrix} \Delta u^x \\ \Delta u^y \\ \Delta u^z \end{pmatrix} \quad (2.7)$$

## 2.2 Diskretisierung

Die vorgestellten mathematischen Definitionen gehen vom unendlichen, kontinuierlichen Raum  $\mathbb{R}^n$  aus. Für die Berechnungen wird aber ein endliches, diskretes Gitter verwendet, d.h. eine Teilmenge von  $\mathbb{Z}^n$ . Eine Zelle in diesem diskreten Gitter wird auch **Voxel** (für „Volumen-Pixel“) genannt. Die vorgestellten Operatoren müssen in eine diskrete Form gebracht werden, die zunächst in einer Dimension erläutert wird.

Es sei  $f: \mathbb{R} \rightarrow \mathbb{R}$  eine reelle Funktion, von der in den späteren Berechnungen Stützpunkte in regelmäßigen Abständen  $\Delta x$  gegeben sind:

$$f_i = f(x_i), \quad x_i = n \cdot \Delta x, n \in \mathbb{Z} \quad (2.8)$$



**Abbildung 2.3:** Ein 2D-Strömungsfeld, das um ein Hindernis herumfließt mit dazugehöriger Rotation. Negative Rotation ist rot gekennzeichnet, positive in blau.

Die (kontinuierliche) Ableitung dieser Funktion ist definiert als

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x) - f(x + h)}{x - h}. \quad (2.9)$$

Je nachdem, wie genau die Simulation sein soll, kann bei der Diskretisierung dieser Ableitung unterschiedlich viel Aufwand einfließen. Grundlage für die Diskretisierung und die anschließende Abschätzung der Genauigkeit ist die **Taylorreihe** von  $f$  (um den Entwicklungspunkt  $a$ ):

$$f(x) = f(a) + f'(a)(x - a) + \frac{f''(a)}{2}(x - a)^2 + O(\Delta x^2) \quad (2.10)$$

Nimmt man als Entwicklungspunkt  $f(x_i)$  und setzt  $x = x_{i+1}$  erhält man die **Vorwärtsdifferenz**-Annäherung der Ableitung:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} + O(\Delta x) \quad (2.11)$$

Setzt man  $a = x_i, x = x_{i-1}$  erhält man analog die **Rückwärtsdifferenz**:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta x} + O(\Delta x) \quad (2.12)$$

Der Nachteil dieser beiden Näherungen ist, dass sie einseitig sind (entweder nach links oder rechts ausgerichtet) und linearen Fehler  $O(\Delta x)$  haben, da die Taylorreihe bereits nach dem ersten Glied abgeschnitten wird.

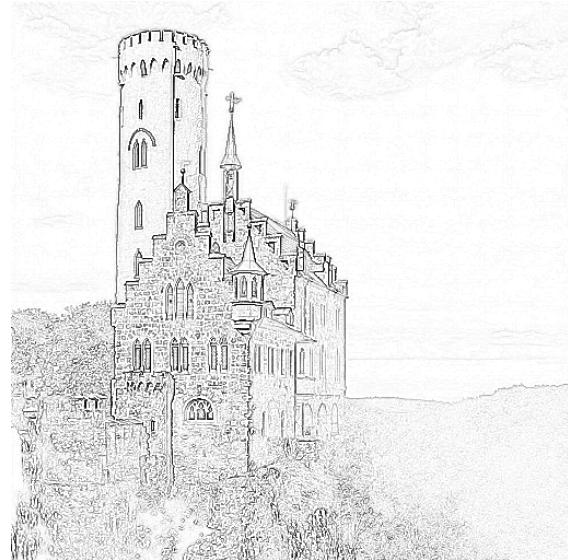
Eine bessere Näherung erhält man, wenn man Gleichung (2.12) von Gleichung (2.11) subtrahiert:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} + O(\Delta x^2) \quad (2.13)$$

Diese Näherung wird **zentrale Differenz** genannt und hat nur quadratischen Fehler  $O(\Delta x^2)$ . Zwar kann man dieses Schema fortführen, um noch bessere Näherung zu erhalten, jedoch



(a) Originalbild



(b) Das Bild nach Anwendung des Laplace-Operators

Abbildung 2.4: Der Laplace-Operator

führt dies dazu, dass eine immer größere Umgebung des aktuellen Punktes betrachtet werden muss, was in der späteren Implementierung zu viele Speicherzugriffe verursachen würde.

Mit Hilfe der zentralen Differenz lässt sich auch eine Näherung für die zweite Ableitung angeben:

$$f''(x_i) = \frac{f(x_{i+1}) - 2 \cdot f(x_i) + f(x_{i-1})}{(\Delta x)^2} \quad (2.14)$$

Da alle Operatoren auf partiellen Ableitungen beruhen, lassen sich nun diskrete Näherungen für jeden Operator angeben. Dies ist in Tabelle 2.1 für ein Skalarfeld  $p$  beziehungsweise ein Vektorfeld  $\vec{u}$  für 3 Dimensionen geschehen.

**Tabelle 2.1:** Die diskretisierten Feldoperatoren in drei Dimensionen

---

| Operator                     | Diskretisierung   |
|------------------------------|---|
| $\operatorname{grad} p$      | $\frac{1}{2\Delta x} \begin{pmatrix} p_{i+1,j,k} - p_{i-1,j,k} \\ p_{i,j+1,k} - p_{i,j-1,k} \\ p_{i,j,k+1} - p_{i,j,k-1} \end{pmatrix}$   |
| $\operatorname{div} \vec{u}$ | $\frac{u_{i+1,j,k}^x - u_{i-1,j,k}^x + u_{i,j+1,k}^y - u_{i,j-1,k}^y + u_{i,j,k+1}^z - u_{i,j,k-1}^z}{2\Delta x}$   |
| $\Delta p$                   | $\frac{p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6p_{i,j}}{(\Delta x)^2}$   |
| $\operatorname{rot} \vec{u}$ | $\frac{1}{2\Delta x} \begin{pmatrix} u_{i,j+1,k}^z - u_{i,j-1,k}^z - u_{i,j,k+1}^y + u_{i,j,k-1}^y \\ u_{i,j,k+1}^x - u_{i,j,k-1}^x - u_{i+1,j,k}^z + u_{i,j,k-1}^z \\ u_{i+1,j,k}^y - u_{i-1,j,k+1}^y - u_{i,j+1,k}^x + u_{i,j-1,k}^x \end{pmatrix}$ |

## 3 Die Navier-Stokes-Gleichungen

### 3.1 Fluide

Die Gleichungen, die im Folgenden beschrieben werden, gelten nicht nur für Gase wie Luft, sondern auch für Flüssigkeiten wie Wasser. Deshalb wird der Begriff **Fluid** verwendet, welcher beide Zustände zusammenfasst. Physikalisch gesehen sind Fluide Substanzen, die einer beliebig langsam Scherung keinen Widerstand entgegensetzen. Dieses Konzept ist allerdings in dieser Arbeit nicht von Bedeutung.

Welches Modell man für die Bewegung von Fluiden wählt, hängt davon ab, welche Art von Fluid man modelliert. Bei dickflüssigen Fluiden wie Honig muss ein Modell gewählt werden, welches die Viskosität gut modelliert. Bei Fluiden mit geringer Dichte wie Luft kann dagegen dieser Faktor ausgelassen werden. Da auch die äußeren Gegebenheiten Einfluss auf das Modell haben, werden zunächst die Rahmenbedingungen des Modells erläutert, bevor die zugehörigen Gleichungen vorgestellt werden.

Das *Volumen* von Fluiden ist im Allgemeinen nicht konstant, es wird durch Veränderung von Druck und Dichte beeinflusst. Dies geschieht z. B. beim Übertreten der Schallmauer (im Medium Luft, beispielsweise bei einer Explosion) oder bei der Ausbreitung von Tönen unter Wasser. In der Strömungsmechanik unterscheidet man daher zwischen **komprimierbaren** und **unkomprimierbaren** Fluiden, je nachdem, ob die Veränderung des Volumens eine Rolle für die Simulation spielt.

Es werden hier nur *unkomprimierbare* Fluide betrachtet. Dies stellt kein Problem dar, da nur vergleichsweise kleine Geschwindigkeiten betrachtet werden. Das Modell wird dadurch allerdings wesentlich vereinfacht.

Fluide haben im Allgemeinen an jedem Punkt  $\vec{x}$  eine unterschiedliche *Dichte*  $\rho(\vec{x})$ , welche im Wesentlichen von Druck und Temperatur beeinflusst wird. Zur weiteren Vereinfachung wird hier allerdings von einer konstanten Dichte  $\rho$  ausgegangen und beispielsweise die Temperatur nicht in die Betrachtungen einbezogen. Effekte wie Auftrieb durch Temperaturunterschiede können dem Modell allerdings optional hinzugefügt werden.

### 3.2 Einführung

Das Modell eines Fluids besteht mathematisch gesehen aus mehreren Feldern (Skalar- und Vektorfeldern), die den Zustand des Fluids zu einem *Zeitpunkt*  $t$  an einer *Position*  $\vec{x}$  angeben. In den **Navier-Stokes-Gleichungen**, die in Abbildung 3.1 dargestellt sind, werden zwei Felder betrachtet, die *Bewegungsgeschwindigkeit*  $\vec{u}(\vec{x}, t)$  des Fluids und der *Druck*  $p(\vec{x}, t)$ . Die Gleichungen wurden 1822 von Claude-Louis Navier und 1845 von George Gabriel Stokes formuliert ([25]).

Gleichung (3.1) wird **Impulsgleichung** genannt, Gleichung (3.2) nennt man **Unkomprimierbarkeitsbedingung**. Der Name und die Bedeutung der Gleichungen werden im nächsten Abschnitt näher beleuchtet.

Gleichung (3.1) ist eine Differentialgleichung, die wegen des Terms  $\vec{u} \operatorname{div} \vec{u}$  nichtlinear ist. Das macht sie besonders schwer mit klassischen Verfahren lösbar.

Die Dimension des Raums taucht in den Gleichungen nicht auf, man kann sie also in zwei oder drei Dimensionen betrachten. Zur Veranschaulichung wird im Folgenden oft von zwei Dimensionen ausgegangen.

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \operatorname{div} \vec{u} = \vec{g} + \nu \Delta \vec{u} - \frac{1}{\rho} \operatorname{grad} p \quad (3.1)$$

$$\operatorname{div} \vec{u} = 0 \quad (3.2)$$

**Abbildung 3.1:** Die unkomprimierbaren Navier-Stokes-Gleichungen

Die einzelnen Bestandteile der Gleichungen werden in den folgenden Kapiteln genauer beleuchtet, hier soll allerdings schon ein kurzer Überblick gegeben werden:

- Die Konstante  $\rho$  gibt die *Dichte* des Fluids an. Für Wasser beträgt sie etwa  $1000 \frac{\text{kg}}{\text{m}^3}$ , für Luft etwa  $1.3 \frac{\text{kg}}{\text{m}^3}$  ([5]).
- Der *Druck*  $p$  spielt bei den späteren Berechnungen eine große Rolle. Er ist dort besonders hoch, wo das Fluid auf Hindernisse trifft und diese umfließt.
- Kräfte, die von außen auf das Fluid wirken, wie z. B. die Schwerkraft oder der eingeführte Wind, werden im Vektorfeld  $\vec{g}: \mathbb{R}^3 \rightarrow \mathbb{R}^3$  zusammengefasst.
- Fluide unterscheiden sich in ihrer **Viskosität** oder „Zähflüssigkeit“, die in der Gleichung mit  $\nu$  bezeichnet ist. Dickflüssige Fluide wie Honig haben ein hohes  $\nu$ , dünnflüssige wie Luft ein niedriges.

Die Navier-Stokes-Gleichungen komplett zu erläutern geht über den Rahmen dieser Arbeit weit hinaus, daher soll vor allem ein intuitives Verständnis der einzelnen Bestandteile gegeben werden. Außerdem soll eine Beziehung zur klassischen Mechanik hergestellt werden, da die Gleichungen starke Parallelen hierzu aufweisen.

### 3.3 Die Impulsgleichung

#### 3.3.1 Lagrange und Euler

Um Fluide zu modellieren, gibt es zwei Betrachtungsweisen. Die **Euler'sche-Betrachtungsweise** und die **Lagrange'sche-Betrachtungsweise**.

Bei der *Lagrange'schen-Betrachtungsweise* modelliert man das Fluid als System von mikroskopisch kleinen *Partikeln* (man modelliert sozusagen die Moleküle des Fluids einzeln). Jedes Partikel hat ein Volumen  $V$ , eine Masse  $m$ , eine Position  $\vec{x}$  und eine Geschwindigkeit  $\vec{u}$ . In jedem Simulationsschritt berechnet man die Kräfte, die auf die Partikel wirken, und berechnet die resultierende Beschleunigung mit der Newton'schen Formel:

$$m \cdot \vec{a} = \vec{F}$$

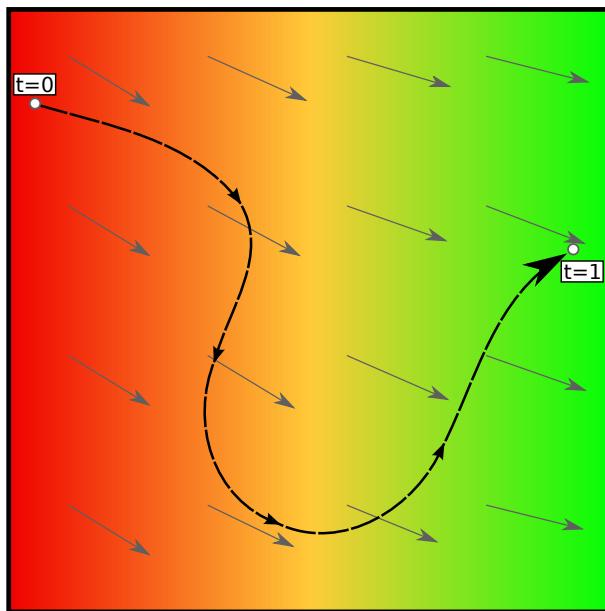
$$m \cdot \frac{\partial \vec{u}}{\partial t} = \vec{F}$$

Diese Vorgehensweise ist intuitiv und einfach umzusetzen, aber mathematisch schwer zu analysieren. Beispielsweise kann man die Frage „Welche Geschwindigkeit hat das Fluid an

Position  $\vec{x}$ ?“ nicht sofort beantworten. Bei der Bewegung der Schneeflocken ist dies aber äußerst wichtig.

Bei der *Euler'schen-Betrachtungsweise* hingegen hält man jeweils einen Punkt im Raum fest und analysiert das Strömungsverhalten (Geschwindigkeit, Temperatur) durch diesen Punkt. Diese Art der Modellierung ist weniger intuitiv, lässt sich aber mit Hilfe von Vektorfeldern und Skalarfeldern sehr gut mathematisch erfassen.

### 3.3.2 Die substantielle Ableitung



**Abbildung 3.2:** Die Bahn eines Bootes im Wasser. Die Temperatur des Fluids ist hier durch Farben kodiert. Rot bedeutet warmes Wasser, grün bedeutet kaltes Wasser. Nicht dargestellt ist die Veränderung der Wassertemperatur über die Zeit.

Sowohl die Euler'sche- als auch die Lagrange'sche Betrachtungsweise spiegeln sich in den Navier-Stokes-Gleichungen wider. Obwohl es nicht sofort erkennbar ist, bilden die Gleichungen eine Entsprechung der Gleichung  $\vec{F} = m \cdot \vec{a}$  mit dem Unterschied, dass keine Kraft auf einen einzelnen Körper berechnet wird, sondern auf einen ganzen Raumabschnitt (ein „Kontinuum“).

Was die beiden Betrachtungsweisen verbindet, ist die **substantielle Ableitung**. Zur Herleitung dieser Ableitungsform betrachten wir zunächst eine beliebige Größe  $q(\vec{x}, t)$ . Diese kann eine skalare Größe wie z. B. die Temperatur eines Gewässers sein, oder eine Vektorgröße wie die Farbe des Wassers an einem Punkt. Da  $q$  in jedem Punkt definiert ist, stellt die Funktion eine Euler'sche Größe dar.

Zudem betrachten wir ein Partikel mit einer Bewegungsbahn durch das Fluid (z. B. ein Ruderboot im Wasser). Seine Position zum Zeitpunkt  $t$  sei gegeben durch  $\vec{p}(t)$ . Dies entspricht einer Lagrange'schen Größe.

Setzen wir beide Größen zusammen, erhalten wir beispielsweise die Temperatur des Wassers auf der Bahn, die das Boot im Wasser verfolgt:

$$\text{Temperatur}(t) = q(\vec{p}(t), t) \quad (3.3)$$

Wollen wir wissen, wie sich die Umgebungstemperatur des Bootes im Lauf der Zeit verändert, bilden wir die (totale) Ableitung dieser zusammengesetzten Größe:

$$\frac{\partial \text{Temperatur}(t)}{\partial t} = \frac{\partial q}{\partial t} + \text{grad } q \cdot \frac{\partial \vec{p}}{\partial t} \quad (3.4)$$

Die Summe auf der rechten Seite besteht aus zwei Teilen. Der erste Term,  $\frac{\partial q}{\partial t}$ , gibt an, wie sich das Fluid unabhängig von der betrachteten Position verändert. Bezogen auf das Boot heißt das, wie sich die Temperatur des Gewässers über den Tag verteilt verändert (beispielsweise durch den Einfluss der Sonnenstrahlen). Der zweite Term ergänzt die globale Ableitung durch die lokale Temperaturänderung, die durch die Bahn des Bootes verursacht wird. Statt eines beliebigen Pfades durch das Fluid nimmt man zur Definition der substantiellen Ableitung nun das Geschwindigkeitsfeld des Fluids zur Hilfe:

$$\frac{Dq}{Dt} := \frac{\partial q}{\partial t} + \text{grad } q \cdot \vec{u} \quad (3.5)$$

Das bedeutet, man geht von einem Partikel aus, welches sich im Fluid „treiben“ lässt, und misst die Veränderung der Größe  $q$  auf seiner Bahn. Analog ist die substantielle Ableitung für Vektorgrößen definiert:

$$\frac{D\vec{q}}{Dt} := \frac{\partial \vec{q}}{\partial t} + \text{div } q \cdot \vec{u} \quad (3.6)$$

Die Impulsgleichung lässt sich mit Hilfe der substantiellen Ableitung schreiben als:

$$\rho \frac{D\vec{u}}{Dt} = -\text{grad } p + \Delta \vec{u} + \vec{f} \quad (3.7)$$

Dies entspricht der klassischen Newton'schen Kraftformel, wobei die Masse  $m$  durch die Dichte  $\rho$  ersetzt wird. Auf der rechten Seite der Gleichung stehen die Kräfte, die das Fluid beeinflussen. Diese Kräfte werden nachfolgend im Einzelnen beschrieben.

### 3.3.3 Die Kräfte in den Navier-Stokes-Gleichungen

Die Kraft, die überall im Fluid in gleicher Weise wirkt, ist die Schwerkraft:

$$\vec{F}_g = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} \quad (3.8)$$

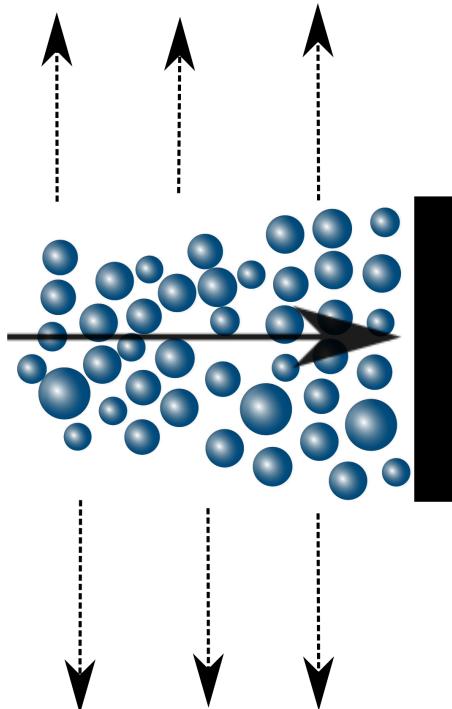
mit  $g \cong 9.81 \frac{m}{s^2}$ .

Durch den Druck  $p$  des Fluids wird eine weitere Kraft  $\vec{F}_p$  ausgeübt. Allerdings erzeugt die Anwesenheit von Druck (also  $p \neq 0$ ) allein keine Kraft. Ist der Druck beispielsweise im gesamten Fluid konstant, ist die ausgeübte Kraft gleich 0.

Stattdessen wirkt  $\vec{F}_p$  „ausgleichend“. Sie zeigt von Bereichen hohen Drucks hin zu Bereichen niedrigeren Drucks. Mathematisch gesehen ist  $\vec{F}_p$  proportional zum negativen Gradienten des Drucks.

$$F_p = - \operatorname{grad} p \quad (3.9)$$

In Abbildung 3.3 ist dies anhand eines Partikelsystems erläutert. Treffen die Partikel auf eine Wand, entsteht Druck, der tangential zur Wand wirkt und die Partikel in diese Richtung beschleunigt. In den späteren Berechnungen spielt der Druck insbesondere eine Rolle, um das Fluid in seinem unkomprimierbaren Zustand zu halten und das Hineinfließen in Hindernisse zu verhindern.



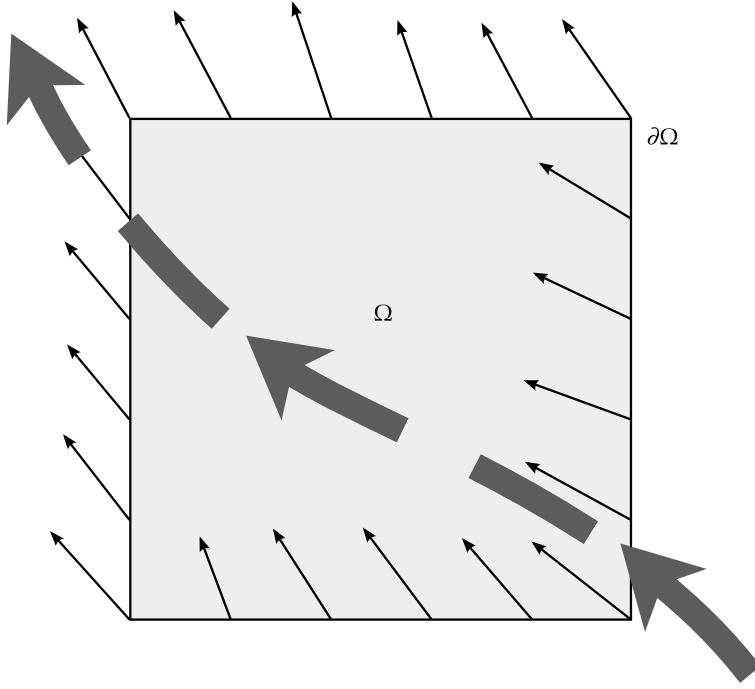
**Abbildung 3.3:** Partikelsystem, das auf ein Hindernis prallt. Der durchgezogene Pfeil deutet die Fließrichtung an, die gestrichelten Pfeile den negativen Gradienten des Drucks. Die Partikel erfahren eine Kraft nach außen.

Die Viskosität des Fluids hat ebenfalls Einfluss auf das Fluid. Bei einem viskosen Fluid wird jeder Deformation ein Widerstand entgegengesetzt. Eine Flüssigkeit wie Honig, die um ein Hindernis herumfließt, bildet hinter dem Hindernis keine Verwirbelungen; die hohe Viskosität wirkt dem entgegen. Luft hingegen hat eine niedrige Viskosität, bildet also besonders viele Wirbel.

Die Kraft, die durch die Viskosität ausgeübt wird, hat den Effekt, Geschwindigkeitsunterschiede innerhalb des Fluids auszugleichen. Sie ist daher proportional zum Laplaceoperator der Geschwindigkeit:

$$F_v = \mu \cdot \Delta \vec{u} \quad (3.10)$$

Die Konstante  $\mu$  stellt die **dynamische Viskosität** des Fluids dar.



**Abbildung 3.4:** Fluss durch ein Rechteck. Der große Pfeil deutet die Fließrichtung des Fluids an. Die Pfeile an den Rändern des Rechtecks zeigen in die dortige Fließgeschwindigkeit- und Richtung, über deren Normalenkomponente in Gleichung (3.11) integriert wird.

### 3.4 Die Unkomprimierbarkeitsbedingung

Um die Unkomprimierbarkeitsbedingung zu motivieren, sei ein beliebiges Volumen  $\Omega \subset \mathbb{R}^3$  gegeben, z. B. ein Würfel im Raum. Seine Oberfläche sei  $\partial\Omega$ .

Die Änderung des Würfelmumens über die Zeit kann berechnet werden, indem man über die Normalenkomponente entlang seiner Oberfläche integriert ([8]). Bildlich gesprochen summiert man auf diese Weise eingehende und ausgehende Strömungen an den Würfelseitenflächen (vgl. Abbildung 3.4):

$$\frac{\partial \text{Volumen}(\Omega)}{\partial t} = \iint_{\partial\Omega} \vec{u} \cdot \vec{n} \quad (3.11)$$

Damit die Flüssigkeit unkomprimierbar ist — sich das Volumen also nicht ändert — sollte dieses Integral verschwinden:

$$\iint_{\partial\Omega} \vec{u} \cdot \vec{n} = 0 \quad (3.12)$$

Mit Hilfe des Divergenzsatzes können wir dieses Integral in ein Volumenintegral umformen:

$$\iiint_{\Omega} \operatorname{div} \vec{u} = 0 \quad (3.13)$$

Da  $\Omega$  beliebig gewählt war, folgt die Unkomprimierbarkeitsbedingung  $\operatorname{div} \vec{u} = 0$ .

## 4 Methode nach Stam

### 4.1 Flüssigkeiten in interaktiven Anwendungen

Numerische Strömungsmechanik (engl. computational fluid dynamics oder CFD) war ursprünglich ein Problem, das nur mit leistungsfähigen Rechenclustern angegangen werden konnte. Diese benötigten mehrere Minuten bis mehrere Tage, um ein Ergebnis zu liefern. Deshalb wurden Probleme, die auf CFD zurückzuführen sind (wie die Simulation von Rauch, Wolken oder eben Wind) nicht physikalisch modelliert, sondern nur visuell angenähert. [41] beschreibt beispielsweise ein Partikelsystem, bei dem die Partikel sich um „Wirbelpunkte“ bewegen, die entweder zufällig generiert oder vom Designer erstellt wurden. Dadurch werden die ansonsten schwer zu berechnenden Geschwindigkeitsfelder approximiert. Mit Hilfe dieser Technik entstand der Film „Particle Dreams“ ([42]), in dem auch ein Schneesturm modelliert wurde.

Ein erstes auf die Computergrafik angepasstes, physikalisches Modell, stellten *Foster* und *Metaxas* 1997 in [12] vor. Es war allerdings numerisch nicht stabil und daher schlecht für Echtzeitanwendungen geeignet. Ein Simulationsschritt benötigte damals außerdem noch ca. 15 Sekunden zur Berechnung. 1999 erschien *Stams* wegweisendes Paper „Stable Fluids“ ([44]), welches zum ersten Mal ein numerisch stabiles Verfahren enthielt (daher „stable“), das (nach eigenen Angaben) Echtzeitsimulationen ermöglicht, wenn auch nur für zweidimensionale Felder. Das Verfahren wird im Folgenden mit SF abgekürzt.

Etwa zur gleichen Zeit wurden zwei weitere Ansätze für die schnelle Simulation von Strömungen entwickelt, die wegen ihrer Komplexität hier nicht weiter vertieft werden sollen:

- Basierend auf zellulären Automaten wurde das **Lattice-Bolzmann-Verfahren** (LBM) für die Simulation eingesetzt, welches — ähnlich Stams Ansatz — datenparallel formuliert ist und komplexe Hindernisse innerhalb der Simulation zulässt. Es wird sowohl für Gase als auch für Flüssigkeiten eingesetzt ([7]).
- Basierend auf Partikelsystemen wurde das ursprünglich in der Astronomie entwickelte Verfahren **Smoothed Particle Hydrodynamics** (SPH) vor allem für die Simulation von Flüssigkeiten eingesetzt ([25]).

Im Jahr 2003 übertrugen *Krüger et. al.* in [18] einige algebraische Operatoren auf die GPU und nutzten sie unter anderem, um SF auf der GPU umzusetzen, allerdings vorerst nur im Zweidimensionalen. Sie erreichten bei einem Feld der Größe  $1024 \times 1024$  Wiederholraten von etwa 9 Bildern pro Sekunde. Im selben Jahr implementierten *Harris et. al.* das Verfahren auf der GPU im Dreidimensionalen und erreichten bei einem Feld der Größe  $32^3$  schon Wiederholraten um 27 Bilder pro Sekunde. Strömungsmechanik in Echtzeit war zu diesem Zeitpunkt also auf Desktop-PCs möglich.

Die GPU-Implementierungen nutzten allerdings die Hardware-Grafikpipeline, die in Abschnitt 6 näher beschrieben wird. Statt Daten aus Arrays einzulesen und mittels einer universellen Programmiersprache wie C zu verarbeiten, waren von der Hardware einige Einschränkungen vorgegeben. Als Datenstruktur standen beispielsweise nur zweidimensionale Texturen zur Verfügung. Die Berechnungen mussten zudem in einer Shadersprache wie GLSL verfasst werden, in der Anweisungen zum Kontrollfluss wie z. B. Schleifen oft

nur eingeschränkt benutzbar waren. Das Spiel „Hellgate: London“ nutzte so erstmals die GPU, um dynamisch Rauch zu simulieren und zu visualisieren ([9]).

Die Grafikhardware entwickelte sich allerdings schnell und konnte immer komplexere Berechnungen durchführen. Um allgemeine (also nicht unbedingt grafikbezogene) Berechnungen aktiv zu unterstützen, veröffentlichte NVidia 2006 die erste Version des CUDA-Frameworks. Mit CUDA kann reiner C++-Code mit Abschnitten versehen werden, die auf der Grafikkarte ausgeführt werden. Ein Jahr später veröffentlichte *Goodnight* eine Implementierung von SF mit Hilfe von CUDA ([13]).

CUDA-Programme laufen allerdings nur auf NVidia-Grafikkarten. Andere Hersteller wie AMD veröffentlichten ihrerseits Frameworks für die eigene Hardware. Es existierte jedoch kein übergreifendes System, bis Ende 2008 die erste OpenCL-Spezifikation veröffentlicht wurde. Viele Hersteller implementierten daraufhin diesen Standard und GPU-Programmierung wurde hardwareübergreifend.

Bislang sind allerdings noch keine Echtzeitanwendungen bekannt, die Stams Verfahren oder OpenCL für die Berechnung von Windfeldern nutzen. Daher soll in dieser Arbeit OpenCL genutzt werden, um SF auf der GPU zu implementieren und das resultierende Windfeld für die Bewegung der Schneeflocken zu verwenden.

## 4.2 Motivation

Die Methode zur approximativen Lösung der Navier-Stokes-Gleichungen, die im Folgenden erklärt wird, wurde von *Jos Stam* im Jahr 1999 vorgestellt. Es wurde speziell für die Computergrafik entwickelt, wobei auf einige Besonderheiten eingegangen wurde:

1. Im Gegensatz zu wissenschaftlichen Simulationen ist man in der Computergrafik an einer Lösung interessiert, die in möglichst kurzen Abständen Ergebnisse produziert. Um eine flüssige Simulation zu erhalten, ist die Laufzeit des Lösungsalgorithmus auf 16 Millisekunden (für 60 Bilder pro Sekunde) bzw. 33 Millisekunden (für 30 Bilder pro Sekunde) beschränkt. Die Navier-Stokes-Gleichungen bilden als System von nichtlinearen Differentialgleichungen hier eine besondere Herausforderung.
2. Bisherige Ansätze zur Strömungsmechanik kamen aus einem wissenschaftlichen Bereich, in dem Exaktheit eine große Rolle spielte. In der Computergrafik allgemein und auch bei der Aufgabenstellung dieser Arbeit ist man allerdings nicht unbedingt an einer exakten Lösung interessiert. Ein physikalisch annähernd korrektes Ergebnis genügt.
3. Bisherige Verfahren (wie z. B. die finiten Differenzen in [12]) waren numerisch *instabil* für große Zeitschritte. Das bedeutet, die Simulation kann immer nur einen bestimmten, kleinen Zeitabschnitt  $\delta$  voranschreiten. Echtzeitanwendungen wie Spiele oder Animationssoftware können allerdings keine minimale Bildwiederholrate garantieren, da sie mit verschiedenen Umgebungen und Hardwarekonfigurationen ausgeführt werden können. Man hat im Allgemeinen also eine Simulationsschrittweite von  $\Delta$  gegeben, die auch größer sein kann als  $\delta$ .

Als Ausweg muss man einen großen Zeitschritt entweder ignorieren — was die Simulation unrealistisch macht — oder ihn in  $\lceil \Delta/\delta \rceil$  kleinere Zeitschritte unterteilen. Die

Zeit, die benötigt wird, um die Simulation ein  $\delta$  weiterzubewegen, ist aber konstant. Dies führt dazu, dass das nächste Frame erneut viel Rechenzeit benötigt und wieder entsprechend lang wird.

Dieser Effekt ist eventuell nicht mehr aufzuhalten, was eine „Explosion“ der Simulation zur Folge hat.

4. Bisherige approximative Verfahren waren auf Hilfe des Designers angewiesen, um Teile der Simulation möglichst realistisch zu modellieren. Bestimmte natürlich vor kommende Turbulenzen mussten explizit in die Simulation eingegeben werden. Idealerweise sollte die Simulation allerdings von selber fortgeführt werden, der Designer sollte nur Rahmenbedingungen wie Hindernisse und globale Eigenschaften des Fluids (wie Dichte und Viskosität) vorgeben.

SF ist also auf Geschwindigkeit ausgelegt, basiert aber nichtsdestotrotz auf den Navier-Stokes-Gleichungen und erzielt akkurate Simulationsergebnisse. Anfängliche Schwachstellen des Verfahrens, wie die zu starke Dämpfung von Wirbeln, wurden in weiteren Arbeiten ausgebessert ([11]). Diese Verbesserungen sind teilweise in diese Arbeit eingeflossen.

Wegen der numerischen Stabilität und sehr guten Parallelisierbarkeit wird Stams Verfahren bereits in einigen Spielen eingesetzt (siehe [9, 33]). Hier wird eine leicht abgewandelte Fassung beschrieben, bei der andere Randbedingungen verwendet werden.

### 4.3 Überblick über das Verfahren

In diesem Abschnitt wird ein grober Überblick über die Bestandteile des Verfahrens gegeben, die sich direkt aus den Navier-Stokes-Gleichungen ergeben. Es werden außerdem die Rahmenbedingungen erläutert.

Die Fluidsimulation findet auf einem endlichen Gitter, also einer Teilmenge von  $\mathbb{Z}^n$  statt. Gegeben sei das Geschwindigkeitsfeld zum Zeitpunkt  $t$ :  $\vec{u}^t$ . Als Anfangszustand könnten zum Zeitpunkt  $t = 0$  z. B. alle Zellen auf eine vorgegebene „Windrichtung“ gesetzt sein. Gegeben sei außerdem ein Zeitdelta  $\Delta t$  (nicht zu verwechseln mit dem Laplaceoperator). Ziel ist es, unter Zuhilfenahme der Navier-Stokes-Gleichungen ein neues Geschwindigkeitsfeld  $\vec{u}^{t+\Delta t}$  zu berechnen.

Außerdem berechnet man einmalig ein *Hindernisfeld*  $b$ . Ist  $b_{i,j,k} = 1$ , dann ist diese Zelle mit einem Hindernis ausgefüllt. Ist  $b_{i,j,k} = 0$ , ist die Zelle frei. In Abschnitt 10 wird erklärt, wie dieses Hindernisfeld gefüllt wird.

Die rechte Seite der Gleichung (3.1) besteht aus mehreren Summanden:

$$\vec{u} \operatorname{div} \vec{u} - \nu \Delta \vec{u} + \vec{g} - \frac{1}{\rho} \operatorname{grad} p \quad (4.1)$$

In Stams Verfahren wird jeder dieser Summanden als eine Operation betrachtet, die ein Geschwindigkeitsfeld sowie eventuell weitere Eingabegrößen wie  $\Delta t$  erhält, und die ein neues Geschwindigkeitsfeld zurückgibt. Das so berechnete neue Feld wird zur Eingabe der darauffolgenden Operation verwendet. So wird die Lösung der Impulsgleichung in kleinere Teile zerlegt, die für sich behandelt werden können:

- Der Term

$$\vec{u} \operatorname{div} \vec{u} \quad (4.2)$$

stellt die **Advektion** dar. Das Vektorfeld wird entlang seiner eigenen Strömungsrichtung weiterbewegt. Sie sei im Folgenden mit  $\operatorname{advection}(\vec{u}, \Delta t)$  bezeichnet.

- Der Term

$$\nu \Delta \vec{u} \quad (4.3)$$

stellt die viskose **Diffusion** dar. Selbst wenn keine Kräfte auf das Fluid wirken, bewegt es sich durch Diffusionsprozesse weiter, so wie Farbe auf einem Blatt Papier verläuft.

Dieser Term kann in SF ignoriert werden. Diffusion entsteht ohnehin „zufällig“ durch Genauigkeitsfehler während der Advektion (siehe unten). Durch das Vernachlässigen kann die Performance erhöht werden, denn die Lösung dieses Terms ist sehr aufwändig.

- Der Term  $\vec{g}$  umfasst schlicht die Summe aller äußeren Kräfte, die auf das Fluid wirken. Hierzu gehört sowohl die Schwerkraft als auch der von außen eingeführte Wind. Die Operation bekommt als Eingabe das Geschwindigkeitsfeld sowie das Zeitdelta (bei größerem Zeitunterschied zum letzten Simulationsschritt sollen die Kräfte stärker wirken). Sie sei im Folgenden mit  $\operatorname{externalForces}(\vec{u}, \Delta t)$  bezeichnet.

- Der *Druck* des Fluids wird in dem Term

$$-\frac{1}{\rho} \operatorname{grad} p \quad (4.4)$$

zusammengefasst ( $\rho$  bezeichnet die Dichte des Fluids). Er dient am Ende unter anderem dazu, die Randbedingungen und die Unkomprimierbarkeit zu erzwingen. Die Berechnung des Druckfeldes  $p$  aus dem Geschwindigkeitsfeld  $u$  sei mit  $\operatorname{calcPressure}(\vec{u})$  bezeichnet.

- Wie bei Differentialgleichungen üblich, müssen die *Rand- und Anfangsbedingungen* behandelt werden. Dadurch wird einerseits sichergestellt, dass das Fluid nicht in Hindernisse eindringt und andererseits an den Rändern der Simulation frei hinausfließen kann. Diese Operation wird im Folgenden als  $\operatorname{boundaries}(\vec{u})$  bezeichnet und wird noch vor der Berechnung des Drucks durchgeführt.

---

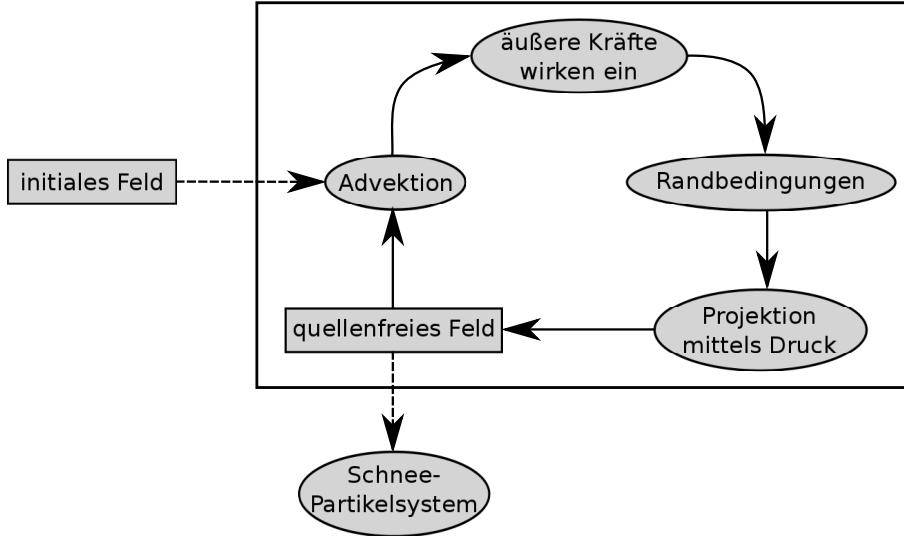
#### Algorithm 4.1 Der Lösungsalgorithmus in Pseudocode

---

```
function SIMULATE( $\vec{u}, \Delta t$ )
     $\vec{u}' = \operatorname{advection}(\vec{u}, \Delta t)$ 
     $\vec{u}'' = \operatorname{externalForces}(\vec{u}', \Delta t)$ 
     $\vec{u}''' = \operatorname{boundaries}(\vec{u}'')$ 
     $p = \operatorname{calcPressure}(\vec{u})$ 
    return  $\vec{u}''' - \operatorname{grad} p$ 
end function
```

---

Für jede Operation wird im Folgenden ein Lösungsalgorithmus vorgestellt, insgesamt erhält man aus der Impulsgleichung den Pseudocode in Algorithmus 4.1. Eine grafische Darstellung findet sich in Abbildung 4.1.



**Abbildung 4.1:** Die Simulationsschleife. Projektion bezeichnet die Berechnung des Druckgradienten und die Subtraktion des Gradienten vom Geschwindigkeitsfeld. Es entsteht ein quellenfreies Vektorfeld, welches wieder in den Advektionsalgorithmus gespeist werden kann. Dieses Feld wird zudem für die Beeinflussung der Schneepartikel verwendet.

#### 4.4 Advektion

Wie bereits in der Erklärung der Navier-Stokes-Gleichungen angedeutet, wird bei der Advektion das Geschwindigkeitsfeld entlang „sich selber“ weiterbewegt. Auf diese Weise kann sich Wind, der von einer Seite der Simulation eingeführt wird, über den gesamten Simulationsbereich ausbreiten. Manchmal spricht man auch von *Selbstadvektion*. In älterer Literatur ist auch von *Konvektion* die Rede.

Im Folgenden wird etwas allgemeiner davon ausgegangen, dass eine beliebige Größe  $q_{i,j,k}^t$  entlang des Vektorfelds  $\vec{u}$  bewegt werden soll. Die Methode  $\text{advection}(q, \Delta t)$  kann also auch verwendet werden, um Temperaturwerte oder die Dichte des Rauchs an einer Stelle entlang des Vektorfelds weiterzubewegen. Als Ausgabe erhält man ein neues Feld  $q^{t+\Delta t}$ .

Um die Advektion durchzuführen gibt es mehrere Ansätze. Der gängigste arbeitet mit der Methode der finiten Differenzen und wurde unter anderem in [11] benutzt. Finite Differenzen führen allerdings zu einem numerisch instabilen Algorithmus, wie in der Einleitung bereits erwähnt. Daher wird eine andere Vorgehensweise gewählt. Ein ebenfalls numerisch instabiler Ansatz wird leicht modifiziert, um Stabilität zu erreichen und ihn ohne Einschränkungen verwenden zu können.

Das zu berechnende neue Feld  $q^{t+\Delta t}$  sei anfangs überall 0 (bzw.  $(0, 0, 0)$ , falls es ein Vektorfeld ist). Man stelle sich nun an jedem Gitterpunkt  $(i, j, k)$  ein Partikel mit „Ladung“  $q_{i,j,k}$  vor, welches im Fluid treibt. Das Partikel wird durch das Fluid beschleunigt und dadurch um  $\Delta t \cdot \vec{u}_{i,j,k}$  verschoben. Es befindet sich im nächsten Zeitschritt bei

$$\vec{p} = (i, j, k) + \Delta t \cdot \vec{u}_{i,j,k} \quad (4.5)$$

Somit liegt es im Allgemeinen nicht mehr *exakt* auf einem Gitterpunkt, sondern zwischen

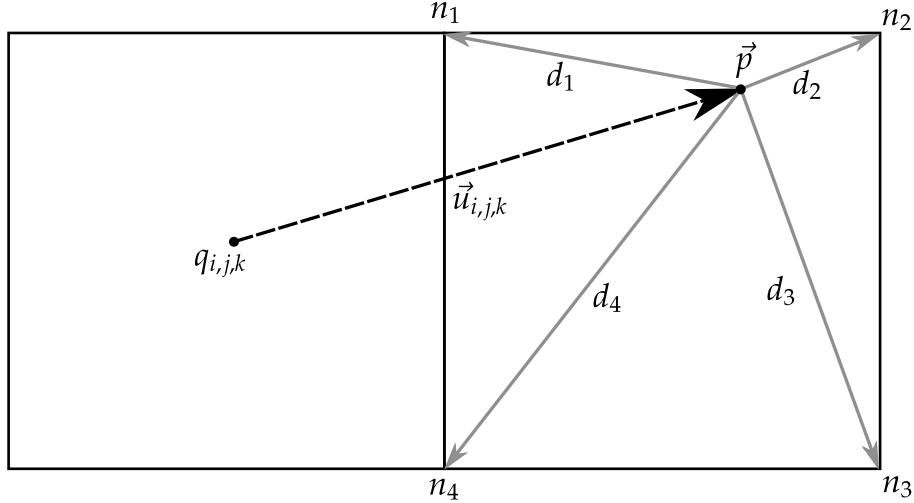


Abbildung 4.2: Das numerisch instabile Advektionsverfahren

8 Nachbarpunkten  $n_i \in \mathbb{Z}^3, i \in \{1, \dots, 8\}$  (siehe Abbildung 4.2). Zu jedem dieser Nachbarpunkte kann man den Abstand  $d_i$  zum Partikel bestimmen:

$$d_i = \|\vec{p} - n_i\|_2, i \in \{1, \dots, 8\} \quad (4.6)$$

Diese  $d_i$  dienen als Gewichtung, um die „Ladung“  $q_{i,j,k}$  anteilig auf die Nachbarknoten aufzuteilen:

$$q'_{n_i} \leftarrow q'_{n_i} + d_i \cdot q_{i,j,k} \quad (4.7)$$

Wird diese Neuverteilung für jeden Gitterpunkt durchgeführt, erhält man ein neues Feld, was um  $\Delta t$  weitergeführt wurde. Das beschriebene Verfahren ist intuitiv und einfach zu implementieren. Aber es ist numerisch nicht stabil und führt zu Oszillationen.

Stam wählte einen anderen Ansatz, die sogenannte *Methode der Charakteristika*. Die Idee ist ähnlich zu der gerade vorgestellten, funktioniert aber in die „umgekehrte Richtung“.

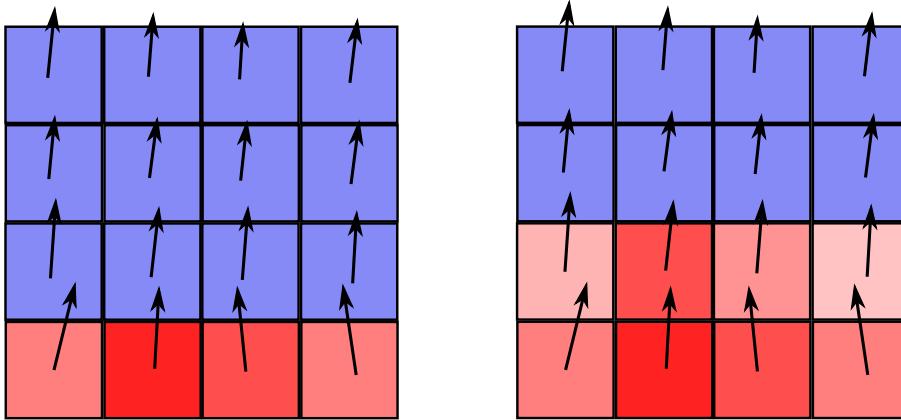
Man betrachtet wieder jeden Gitterpunkt  $(i, j, k)$  einzeln, stellt sich diesmal allerdings vor, man sei auf der Zeitachse im Punkt  $t + \Delta t$ , also bereits im nächsten Zeitschritt. Das gedachte Partikel an Position  $(i, j, k)$  habe die Geschwindigkeit  $\vec{u}_{i,j,k}^t$ .

Rechnet man auf der Zeitachse um  $\Delta t$  zurück, erhält man die „vorherige“ Position des Partikels, nämlich  $(i, j, k) - \Delta t \cdot \vec{u}_{i,j,k}^t$ . Es ergibt sich dasselbe Problem wie bei dem vorher beschriebenen Ansatz: Die Position liegt nicht genau auf dem Gitter, sondern dazwischen (siehe Abbildung 4.4)<sup>1</sup>.

Als Lösung wird zwischen den 8 Nachbarwerten des verschobenen Partikels *interpoliert* und der resultierende Wert in die Zelle  $\vec{u}_{i,j,k}^{t+\Delta t}$  geschrieben. Dies ist eine Operation, auf die Grafikkarten stark optimiert sind, und bei denen Texturen hohe Performance erreichen können.

---

<sup>1</sup>Hier wird nur ein Schritt der Partikelflugbahn zurückverfolgt. Es besteht die Möglichkeit, mehrere Schritte zurückzuverfolgen, die aber aufwändiger zu implementieren ist.

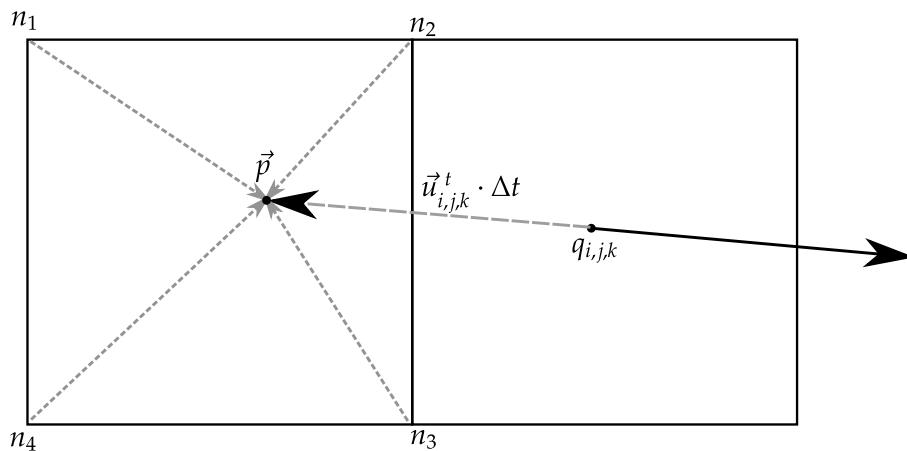


**Abbildung 4.3:** Ein Schritt des numerisch instabilen Advektionsverfahrens, angewendet auf ein Temperaturfeld. Die roten Werte deuten eine Temperaturquelle an. Der anfänglich kalte Raum erwärmt sich durch das Verfahren langsam.

Allerdings liegt hier auch eine Fehlerquelle des Verfahrens. Interpolation ist eine glättende Operation, sie liefert nur eine *Approximation* des Wertes, der zwischen den Gitterzellen angenommen würde. Ähnlich wie bei der Herleitung der diskreten Ableitung in Abschnitt 2.2 muss man in der Implementierung einen Kompromiss eingehen und ein Interpolationsverfahren wählen, welches Genauigkeit und Rechenleistung verbindet.

Auch für große Zeitschritte liefert das Verfahren ein Vektorfeld, welches in seinem Wertebereich beschränkt ist, denn die Interpolation liefert nie einen Wert zurück, der betragsmäßig größer ist als die Ausgangswerte. Das Verfahren ist deshalb stabil.

Allerdings sollte  $\Delta t$  nicht zu groß gewählt werden. [11] gibt hier als Richtlinie etwa  $5 \times$  die Gittergröße an. Diese Art der Advektion wird auch **semilagrange'sche Advektion** genannt, da zwar ein diskretes Gitter verwendet wird, aber die Flugbahn von Partikeln das Verfahren motiviert.



**Abbildung 4.4:** Stom's stabiles Advektionsverfahren.

Natürlich kann es passieren, dass man über den Rand des Simulationsbereiches „hinaus“

läuft“. Es gibt mehrere Möglichkeiten, dies zu behandeln. Beispielsweise könnte man auf die gegenüberliegenden Seite des Simulationsbereiches umbrechen (periodische Randbedingung), welches in Stams Originalarbeit getan wurde. Allerdings erhält man hier in Bezug auf ein Windfeld unrealistische Ergebnisse.

Alternativ kann man die Gerade betrachten, die durch den Mittelpunkt der aktuellen Zelle geht und als Richtungsvektor den Geschwindigkeitspfeil des Fluids besitzt. Diese Gerade schneidet man mit den Rändern der Simulation. Man wählt dann den Schnittpunkt auf dem Rand als Basis für die Interpolation (siehe Abbildung 4.5). Dieser Ansatz wurde in der Arbeit implementiert.

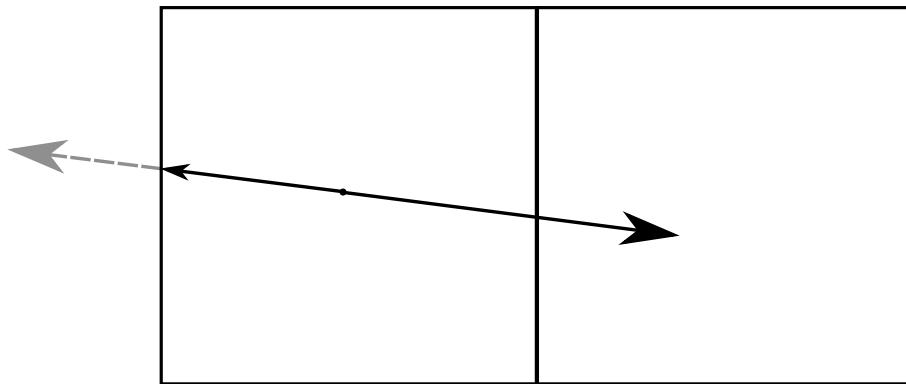


Abbildung 4.5: Abschneiden der Geschwindigkeit an den Rändern

Weiterhin muss auch der Fall, dass der zurückberechnete Vektor  $(i, j, k) - \Delta t \cdot u_{i,j,k}^t$  in einem Hindernis landet, behandelt werden. Hierfür kann man einen Geschwindigkeitswert von  $(0, 0, 0)$  annehmen, was einem stationären Hindernis wie einem Gebäude entspricht. Es ist aber auch möglich, ein weiteres Vektorfeld  $\vec{u}_{\text{boundary}}$  zu verwalten, in dem für jede Gitterzelle die Geschwindigkeit des dort vorhandenen Hindernisses notiert ist. Dies wurde in [9] umgesetzt. Für die Simulation in dieser Arbeit wurde von unbeweglichen Hindernissen ausgegangen.

## 4.5 Äußere Kräfte

Die äußeren Kräfte werden in dem Term  $g$  zusammengefasst und schlicht auf die Geschwindigkeit addiert. In diesem Term lassen sich mehrere interessante Effekte umsetzen, wie Auftrieb oder die Schwerkraft. In diesem Abschnitt werden zwei dieser Effekte vorgestellt.

### 4.5.1 Gravitation

Zu den äußeren Kräften gehört die *Gravitation*. Diese lässt sich sehr einfach ausdrücken:

$$F_g = \Delta t(0, -9.81, 0) \quad (4.8)$$

### 4.5.2 Wirbelstärkenerhaltung

Der Advektionsschritt enthält eine lineare Interpolation, um die Geschwindigkeitswerte in der Nachbarschaft des zurückverfolgten Partikels zu finden. Diese simple Interpolationsmethode führt allerdings dazu, dass Genauigkeit bei der Simulation verloren geht. Dadurch werden viele interessante Phänomene in Fluiden, wie die starke Wirbelbildung bei Rauch, gedämpft und treten nicht mehr so stark in Erscheinung.

Um dies auszugleichen, gibt es mehrere Ansätze. MacCormack hat ein Advektionsverfahren entwickelt, welches nicht mehr unbedingt stabil ist, aber eine bessere Fehlerabschätzung liefert ([40, 9]). Dadurch entstehen mehr Wirbelphänomene, allerdings bietet diese Methode keine Möglichkeit, Einfluss auf die Wirbelstärke zu nehmen.

Daher wurde statt des MacCormack-Verfahrens eine Technik namens **Wirbelstärkenerhaltung** (engl. *vorticity confinement*) umgesetzt. Sie ist einfach zu implementieren, schnell und erlaubt, die Stärke zu variieren. Entwickelt wurde sie, um die komplexen Turbulenzen in der Umgebung von Helikoptern zu modellieren ([45]).

Die Idee ist, Wirbel zu identifizieren und an den „richtigen“ Stellen zu verstärken. Dazu wird zuerst die Rotation des Geschwindigkeitsfeldes,  $\vec{R} = \text{rot } \vec{u}_{i,j,k}$ , bestimmt. Der Betrag der Rotation,  $|\vec{R}|$ , gibt an, wie stark die Wirbelbildung in einem Punkt ist. Der *Gradient* des Betrags,  $\text{grad } |\vec{R}|$ , zeigt also von Bereichen niedriger Wirbelstärke zu Bereichen hoher Wirbelstärke.

Um die Wirbel zu verstärken, bildet man eine Kraft, die senkrecht zum (normalisierten) Gradienten und zur Rotation ist:

$$\vec{F}_{\text{vorticity}} = \varepsilon \cdot \left( \frac{\text{grad } \vec{R}}{|\text{grad } \vec{R}|} \times \vec{R} \right) \quad (4.9)$$

Die Konstante  $\varepsilon$  gibt die Stärke der Kraft an.

## 4.6 Projektion

### 4.6.1 Einleitung

Die bisher vorgestellten Algorithmen vernachlässigen die Unkomprimierbarkeit des Fluids und behandeln einzig die Impulsgleichung. Nach der Advektion und der Addition der äußeren Kräfte kann die zweite Bedingung 3.2 in den Navier-Stokes-Gleichungen also verletzt sein, und die Simulation somit nicht mehr physikalisch korrekt. Außerdem sind die Randbedingungen eventuell verletzt worden. Beispielsweise wurde bei der Advektion nicht beachtet, dass das Fluid um Hindernisse herumströmen sollte. Für beide Fälle kommt der Druck  $p$  als Korrekturterm ins Spiel. In diesem Abschnitt wird  $p$  näher erläutert und ein Algorithmus zu dessen Bestimmung vorgestellt. Danach werden die Randbedingungen der Simulation näher betrachtet.

Gegeben sei ein beliebiges Vektorfeld  $\vec{u}$ , zum Beispiel das Geschwindigkeitsfeld nach der Advektion und nach Anwendung der äußeren Kräfte. Da  $\vec{u}$  nicht mehr quellenfrei ist, ist eine Funktion  $\mathbb{P}$  gesucht, die  $\vec{u}$  auf ein quellenfreies Vektorfeld abbildet ( $\text{div } \vec{u} = 0$ ). Hierzu bedient man sich eines Ergebnisses aus der Vektoranalysis:

**Satz** (Helmholtz-Zerlegung). *Sei  $\vec{u}$  ein zweimal stetig differenzierbares Vektorfeld auf einem beschränkten Definitionsbereich. Dann lässt  $\vec{u}$  sich zerlegen in ein quellenfreies Vektorfeld  $\vec{w}$  und den Gradienten eines Skalarfeldes  $p$ :*

$$\vec{u} = \vec{w} + \operatorname{grad} p \quad (4.10)$$

Gleichung (4.10) soll nach  $\vec{w}$  aufgelöst werden, denn dieses Feld ist quellenfrei. Das Feld  $p$  ist aber ebenfalls eine Unbekannte in der Gleichung. Um die Gleichung aufzulösen, wendet man die Divergenz auf beide Seiten der Gleichung an und nutzt aus, dass der Operator linear ist ( $\operatorname{div}(\vec{u} + \vec{v}) = \operatorname{div} \vec{u} + \operatorname{div} \vec{v}$ ). Als Ergebnis erhalten erhält man:

$$\operatorname{div} \vec{u} = \Delta p \quad (4.11)$$

Hier wurde ausgenutzt, dass  $\operatorname{div} \vec{w} = 0$  gilt. Es entsteht eine sogenannte **Poisson-Gleichung**. Ihre Lösung ist ein Standardproblem in der Physik, und im Folgenden wird ein Verfahren vorgestellt, welches Gleichung (4.11) nach  $p$  auflösen kann. Mit  $p$  kann man dann durch Umstellen und Bildung des Gradienten  $\vec{w}$  bestimmen:

$$\vec{w} = \vec{u} - \operatorname{grad} p \quad (4.12)$$

Darauf aufbauend kann man dann den Operator  $\mathbb{P}$  als die Funktion, die ein Vektorfeld auf den quellenfreien Anteil der Helmholtz-Zerlegung abbildet, definieren:

$$\mathbb{P}: \mathbb{R}^n \rightarrow \mathbb{R}^n \quad (4.13)$$

Diese Funktion wird in der Literatur auch oft **Projektion** genannt. Führt man  $\mathbb{P}$  am Ende des Verfahrens aus, erhält man ein unkomprimierbares Vektorfeld.

#### 4.6.2 Lösung des Poissonproblems

Um das Vektorfeld quellenfrei zu machen, muss folgende allgemeine Poisson-Gleichung nach  $p$  gelöst werden:

$$\Delta p = x, \quad (4.14)$$

wobei  $x$  ein Skalarfeld ist.

Es existieren zahlreiche Lösungsverfahren für solch eine Gleichung. Bei der Auswahl des Verfahrens muss beachtet werden, welches Verfahren sich gut auf der Grafikkarte umsetzen lässt und möglichst schnell eine akzeptable Lösung liefert.

Es haben sich mehrere **Iterationsverfahren** als günstig herausgestellt. Verfahren dieser Art beginnen mit einer initialen Lösung (beispielsweise schlicht  $p = 0$ ) und nähern sich dann in jedem Iterationsschritt weiter der eigentlichen Lösung an.

Betrachtet man große Felder, bieten sich **Mehrgitterverfahren** an. Diese wurden bereits erfolgreich auf GPUs angewendet (siehe [3, 26]). Sie sind allerdings untrivial zu implementieren, da intern ein zweites Iterationsverfahren benötigt wird (der sogenannte Restriktionsoperator), sowie Methoden zum hoch- und runterskalieren von 3D-Feldern.

Simplere Iterationsverfahren, die auch in der Praxis schon auf die GPU übertragen wurden, sind SOR ([38]), Gauss-Seidel-Relaxation ([43]) und das Jacoberverfahren ([9, 15, 33]). Letzteres ist besonders einfach zu implementieren und gleichzeitig hervorragend für die GPU geeignet. Es wurde daher für die Implementierung verwendet und soll jetzt näher erläutert werden.

### 4.6.3 Das Jacobiverfahren

Um das Jacobi-Verfahren zu veranschaulichen, soll es zunächst im Eindimensionalen erläutert werden. Wir betrachten also kein diskretes 3D-Gitter, sondern eine endliche Teilmenge der ganzen Zahlen (die Gitterbreite  $\Delta x$  ist also 1). Der Laplaceoperator entspricht in diesem Fall der zweiten Ableitung von  $p$ .

Für die exakte Lösung  $p$  des Poissonproblems muss an jeder Stelle  $i$  gelten:

$$\frac{p_{i+1} - 2 \cdot p_i + p_{i-1}}{(\Delta x)^2} = x_i \quad (4.15)$$

Löst man diese Gleichung nach  $p_i$  auf und setzt  $\Delta x = 1$  ein, erhält man:

$$p_i = \frac{p_{j+1} + p_{j-1} - x_i}{2} \quad (4.16)$$

Der Wert  $p_i$  definiert sich also durch seine direkten Nachbarn  $p_{i-1}, p_{i+1}$ , daher ist die Gleichung für sich genommen nicht aufzulösen.

Sei nun eine Anfangslösung  $p^0$  gegeben, z. B.  $p_i^0 = 0$  für alle  $i$ . Dann lässt sich eine neue, bessere Lösung bestimmen, indem man die Werte aus  $p^0$  als Näherungen für die Nachbarn in der exakten Lösung nimmt:

$$p_i^1 = \frac{p_{j+1}^0 + p_{j-1}^0 - x_i}{2} \quad (4.17)$$

Im nächsten Iterationsschritt berechnet man dann  $p_i^2$  mit Hilfe von  $p_i^1$ , usw., bis man nahe genug an die exakte Lösung herangekommen ist. In der Praxis sind mindestens 20 Iterationen nötig, da das Verfahren sehr langsam konvergiert.

Eine leichte Konvergenzverbesserung erreicht man, indem man die rechte Seite von Gleichung (4.17) nicht direkt als neue Lösung nimmt, sondern zwischen der bisherigen Lösung und der neuen Lösung mit einem Gewichtungsfaktor  $\omega$  interpoliert:

$$\bar{p}_i^{n+1} = \frac{p_{j+1}^n + p_{j-1}^n - x_i}{2} \quad (4.18)$$

$$p_i^{n+1} = (1 - \omega) \cdot p_i^n + \omega \cdot \bar{p}_i^{n+1} \quad (4.19)$$

Dieses Verfahren wird **gewichtete Jacobi-Iteration** genannt. In der Praxis hat sich ein Faktor von  $\omega = \frac{2}{3}$  als günstig erwiesen.

Das Verfahren lässt sich nun auf 3 Dimensionen (6 Nachbarn) verallgemeinern:

$$p_{i,j,k}^{n+1} = \frac{p_{i+1,j,k}^n + p_{i-1,j,k}^n + p_{i,j+1,k}^n + p_{i,j-1,k}^n + p_{i,j,k+1}^n + p_{i,j,k-1}^n - x_{i,j,k}^n}{6} \quad (4.20)$$

### 4.6.4 Randbedingungen in Differentialgleichungen

Bei der Berechnung des Drucks müssen Randbedingungen beachtet werden. Dies merkt man zum Beispiel daran, dass nicht jede Gitterzelle 6 Nachbarn hat, um die rechte Seite von Gleichung (4.20) auszuwerten. Außerdem muss der Druck in Hinderniszellen so gewählt

werden, dass das Fluid nicht in das Hindernis eindringt, wenn man den Druckgradienten im letzten Schritt des Verfahrens von der Geschwindigkeit subtrahiert. Hier sollen die umsetzbaren Randbedingungen besprochen werden.

Es gibt im Wesentlichen zwei Arten von Randbedingungen bei einer Differentialgleichung (wie der Poissons-Gleichung), zum einen die **Dirichlet-Randbedingungen** und zum anderen die **Von-Neumann-Randbedingungen**:

- Bei Dirichlet-Randbedingungen wird der Wert des Feldes am Rand fest vorgeschrieben. Beispielsweise könnte man die Druckwerte, die über den Simulationsrand hinausgehen (z. B.  $p_{-1,-1,-1}$ ), als 0 annehmen. Dies ist gewissermaßen bei der Advektion geschehen, wo die Geschwindigkeit bei Hinderniszellen als  $(0, 0, 0)$  angenommen wurde.
- Bei Von Neumann-Randbedingungen schreibt man die *Ableitung in Richtung der Normalen* an einem Punkt vor:

$$\frac{\partial \vec{u}}{\partial \vec{n}}(x) = f(x) \quad (4.21)$$

Beispielsweise könnte man  $f(x) = 0$  wählen. Wie bereits in Abschnitt 3.4 erläutert bedeutet das, dass die Geschwindigkeitspfeile nicht in ein Hindernis hinein- oder hinauszeigen dürfen, das Fluid selber darf sich aber tangential frei bewegen. Diese Randbedingung wird deshalb auch spezieller als **free-slip-Randbedingung** bezeichnet.

In der Simulation soll ein Teilbereich der echten Welt simuliert werden. Der Wind wird von einigen Seiten künstlich in die Simulation gespeist. Dies nennt man **inflow boundary** und ist eine Form der Dirichlet-Randbedingung. Aus den verbleibenden Seiten soll der Wind frei aus der Simulation „herausfließen“, als würde der Simulationsbereich dort noch weitergeführt werden. Diese Art der Randbedingung nennt man **outflow boundary**.

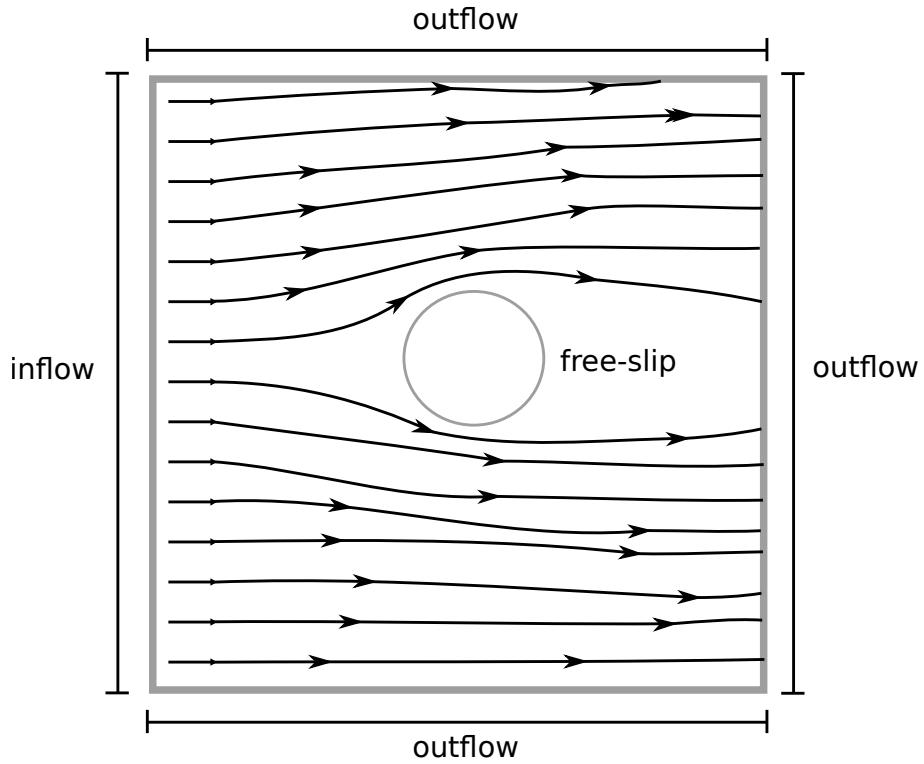
Simuliert man Rauch oder Wasser gibt es weitere Arten von Randbedingungen. Die Grenze zwischen Wasseroberfläche und Luft nennt man **free surface-Randbedingung**. Um diese umzusetzen müsste man außerhalb des Wassers  $p = 0$  setzen. Diese Randbedingung wird hier nicht weiter beachtet.

Eine Zusammenfassung aller Randbedingungen ist in Abbildung 4.6 zu sehen.

#### 4.6.5 Randbedingungen in der Simulation

Es werden zuerst die Randbedingungen für die Hindernisse betrachtet. Hier sollen *free-slip*-Bedingungen erzwungen werden. Diese soll nicht explizit hergeleitet werden, es werden nur die Veränderungen am Jacobiverfahren beschrieben.

Im bisher beschriebenen Jacobiverfahren betrachtet man jede Zelle  $p_{i,j,k}$  des Gitters und bildet die Summe über alle Nachbarzellen. Um die Randbedingungen einfließen zu lassen, testet man, ob die Nachbarzelle von einem Hindernis ausgefüllt ist (hier wird das Hindernisfeld  $b_{i,j,k}$  benutzt, welches eingangs beschrieben wurde). Gehört die Zelle zu einem Hindernis, wird nicht der Druckwert der *Nachbarzelle* zur Summe addiert, sondern der Wert der *aktuellen Zelle* (siehe Abbildung 4.8b). Auf diese Weise kommt es zu keinem Druckunterschied in



**Abbildung 4.6:** Strömungslinien einer Beispielsimulation mit 3 verschiedenen Randbedingungen.

Richtung eines Hindernisses, der Gradient in dieser Richtung ist also 0. Algorithmus 4.2 zeigt den modifizierten Jacobi-Algorithmus.

Der Jacobi-Algorithmus löst die Poissons-Gleichung allerdings nur annähernd. Folglich werden auch die Randbedingungen nur annähernd gelöst. Dies kann im nächsten Simulations-Schritt zu Problemen führen. Daher wird in der Implementierung die free-slip-Randbedingung nach dem Projektionsschritt erzwungen (also nachdem der Gradient des Drucks vom Geschwindigkeitsfeld subtrahiert wurde).

Idealerweise bräuchte man dazu ein Feld  $\vec{n}_{i,j,k}$ , welches in jedem Punkt die Normale des Hindernisses angibt (siehe Abbildung 4.7). Dies wurde in einigen Arbeiten umgesetzt (z. B. [4]), es gibt jedoch eine Vereinfachung, die dieses Feld nicht benötigt: Man iteriert erneut über alle Gitterzellen und testet für jede Gitterzelle  $(i, j, k)$ , welche Nachbarzellen von einem Hindernis ausgefüllt sind. Die zugehörige Komponente der Geschwindigkeit  $\vec{u}_{i,j,k}$  wird dann auf 0 gesetzt (siehe Algorithmus 4.3).

Für die outflow-Randbedingungen am Simulationsrand wird jede Randseite der Simulation einzeln betrachtet. Die Geschwindigkeit einer Randzelle wird ersetzt durch die Geschwindigkeit der „nächst inneren“ Zelle. Beispielsweise werden die Geschwindigkeitswerte  $\vec{u}_{0,j,k}$  (also die an der linken Simulationsseite) ersetzt durch die Werte  $\vec{u}_{1,j,k}$ . Analoges wird für die anderen Seiten des Kubus getan, mit Ausnahme dort, wo inflow-Randbedingungen bestehen, wo also Wind in die Simulation hineinfließt. Dadurch ergibt sich am Rand die Divergenz 0, der Druck ist hier also auch annähernd 0.

---

**Algorithm 4.2** Der modifizierte Jacobi-Algorithmus

---

```
function JACOBI( $p, x$ )
     $p' = 0$                                 ▷ Ergebnisfeld ist  $p'$ 
    for all  $(i, j, k)$  do
        sum  $\leftarrow 0$ 
        for all Nachbarn  $(i', j', k')$  von  $p_{i,j,k}$  do
            if  $b_{i',j',k'} = 1$  then
                sum  $\leftarrow$  sum +  $p_{i',j',k'}$ 
            else
                sum  $\leftarrow$  sum +  $p_{i',j',k'}$ 
            end if
        end for
        new = (sum -  $x_{i,j,k}$ ) / 6
         $p'_{i,j,k} \leftarrow (1 - \omega) \cdot p_{i,j,k} + \omega \cdot \text{sum}$       ▷ Gewichtete Jacobi-Iteration
    end for
    return  $p'$ 
end function
```

---

---

**Algorithm 4.3** Die abschließende Erzwingung der Randbedingungen

---

```
function FREESLIPBOUNDARY( $\vec{v}, b$ )
    for all  $(i, j, k)$  do
        if  $b_{i-1,j,k} = 1$  or  $b_{i+1,j,k} = 1$  then
             $\vec{u}_{i,j,k}^x = 0$ 
        end if
        if  $b_{i,j-1,k} = 1$  or  $b_{i,j+1,k} = 1$  then
             $\vec{u}_{i,j,k}^y = 0$ 
        end if
        if  $b_{i,j,k+1} = 1$  or  $b_{i,j,k-1} = 1$  then
             $\vec{u}_{i,j,k}^z = 0$ 
        end if
    end for
end function
```

---

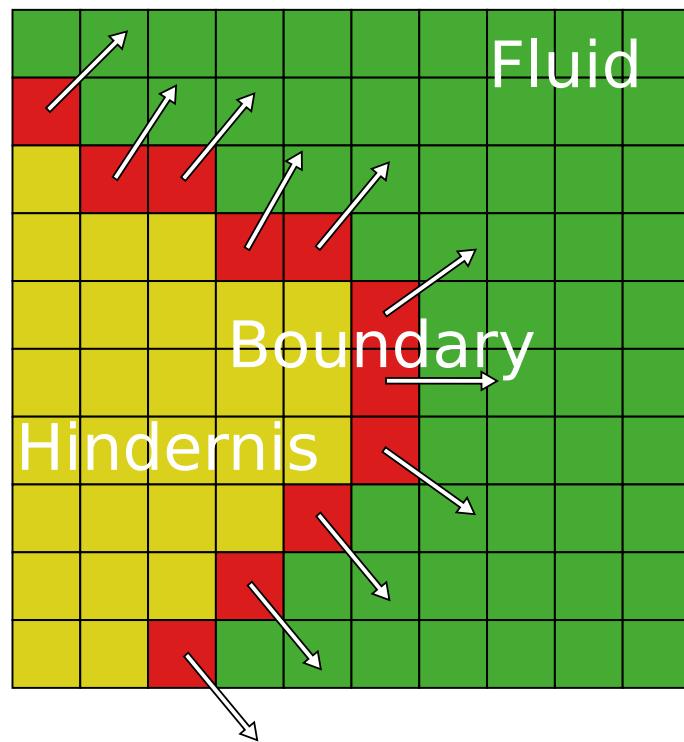


Abbildung 4.7: Hindernisse mit zugehöriger Normale

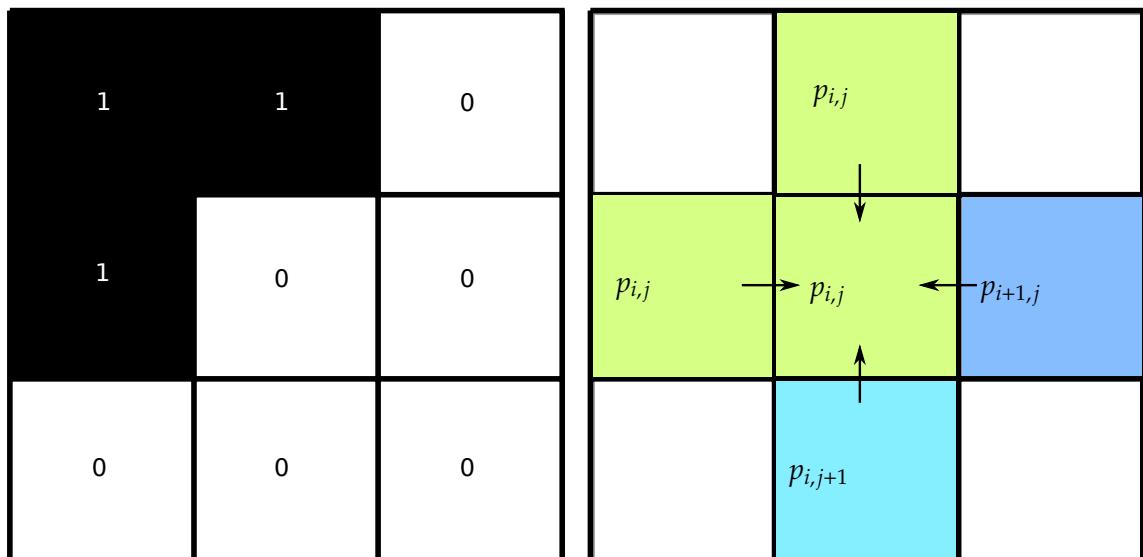


Abbildung 4.8: Druck in Zusammenhang mit Hindernisfeldern

## 5 OpenCL

### 5.1 Einleitung

In diesem Abschnitt wird OpenCL als das Framework, welches für die Berechnungen verwendet wurde, vorgestellt. Im ersten Teil wird ein kurzer geschichtlicher Abriss gegeben und die Stellung von OpenCL unter anderen Frameworks herausgestellt. Im Anschluss wird der Begriff Parallelität definiert und spezielle Ausprägungen davon im Kontext mit OpenCL besprochen. Mit diesem Wissen lässt sich abschätzen, wieso sich die vorgestellten Algorithmen besonders für eine parallele Abarbeitung eignen.

Im darauf folgenden Abschnitt wird die Architektur von OpenCL im Detail erläutert. Schließlich wird ein vollständiges Programm vorgestellt, welches eine einfache Berechnung in OpenCL durchführt. Dies soll vor allem den in der Implementierung immer wiederkehrenden Arbeitsablauf deutlich machen, der nötig ist, um Daten und Programme auf der GPU zu verwalten. Im Kapitel über die konkret implementierten Algorithmen wird dann lediglich der OpenCL-C-Code beschrieben.

### 5.2 Geschichtliches

Strömungsmechanische Berechnungen gelten als sehr rechenaufwändiges Problem, für das traditionell große Rechencluster eingesetzt werden. Diese Cluster bestehen üblicherweise aus tausenden einzelnen Prozessoren, die jeweils einen kleinen Teil des Problems lösen. Die Prozessoren werden dann zu einem 3D-Gitter zusammengefasst, welches den gesamten Simulationsbereich abdeckt. Dadurch, dass die Recheneinheiten parallel arbeiten können, erhält man schnell eine Lösung des Gesamtproblems.

Desktop-Rechner hatten bis vor einigen Jahren allerdings nur einen einzelnen Prozessor, so dass die oben beschriebene Arbeitsweise nicht übertragbar war. Tatsächlich zeigte sich bei Desktop-CPPUs bis ca. 2005 ein Anstieg in der *Taktrate* der Prozessoren, aber nicht in deren *Anzahl*. Diese Entwicklung endete, da mit der Taktrate auch der Stromverbrauch immer weiter ansteigt. In [6] findet sich der folgende Zusammenhang zwischen dem Stromverbrauch in Watt  $P$ , der Taktfrequenz  $f$ , der Spannung  $V$  und des Widerstands  $C$  eines Prozessors:

$$P = CV^2f \quad (5.1)$$

Ersetzt man einen einzelnen Prozessor mit Taktfrequenz  $f$ , Spannung  $V$  und Widerstand  $C$  durch zwei Prozessoren halber Frequenz  $f/2$ , so erhöht sich laut [6] zwar der Widerstand auf  $2.2C$ , aber die Spannung sinkt drastisch auf  $0.6V$ . Die zwei Prozessoren verbrauchen daher zusammen nur 0.396 mal so viel Leistung wie der einzelne Prozessor. Aufgrund dessen geht aktuell der Trend hin zu Prozessoren mit mehreren Kernen (momentan bis zu 8).

In den letzten Jahren ist auch die Leistungsfähigkeit von Grafikprozessoren (GPUs) rasant angestiegen (siehe Abbildung 5.1a und Abbildung 5.1b). GPUs waren anfangs spezialisierte Recheneinheiten, um die in Abschnitt 6 vorgestellte Grafikpipeline umzusetzen. Da jeder Vertex und jedes Fragment für sich bearbeitet werden kann, bot sich hier von Anfang an eine hochparallele Architektur an.

Mit der Zeit stieg jedoch die Nachfrage nach mächtigeren Shaderprogrammen, um aufwendigere Grafikeffekte umzusetzen. Daher wurden die Recheneinheiten der GPUs immer

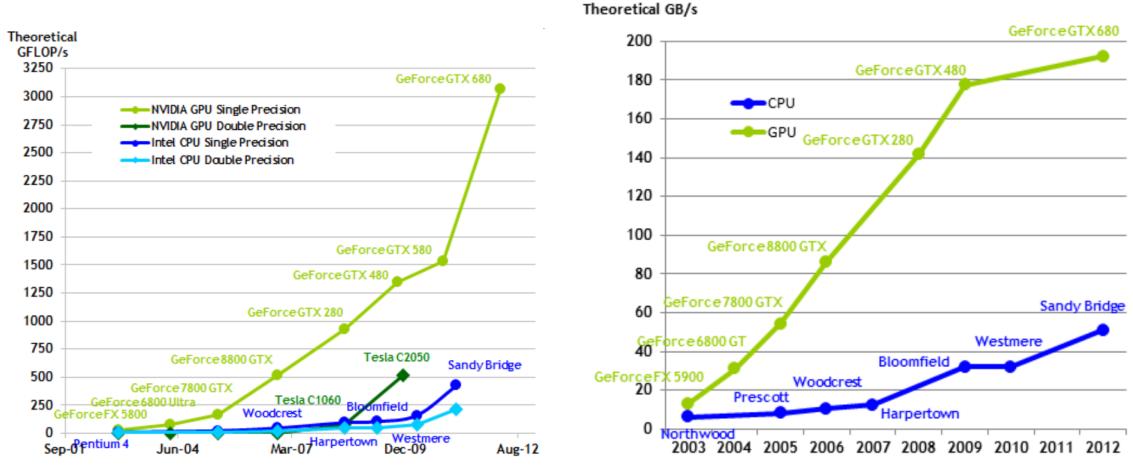


Abbildung 5.1: Vergleich zwischen CPUs und GPUs

weiter verallgemeinert. Mit Hilfe dieser leistungsfähigen Shaderprogramme begannen auch Wissenschaftler, Grafikkarten für Rechenaufgaben zu verwenden und es entstanden ca. 2004 erste Umsetzungen von Stams Methode auf Grafikkarten ([49]).

Allerdings war es bis zu diesem Zeitpunkt weder möglich, rohen Speicher auf der Grafikkarte zu reservieren, noch beliebige Programme zu starten, die auf diesem Speicher arbeiten. Vielmehr mussten die bestehenden Mechanismen der Grafikpipeline „missbraucht“ werden. So wurden Simulationsergebnisse in zweidimensionalen Texturen gespeichert, die man als „Rendertargets“ angeben konnte. Dann wurde ein Rechteck auf den Bildschirm gezeichnet, um die Ausführung der Berechnungs-Shader künstlich anzustoßen.

Um allgemeine Berechnungen komfortabler auf der Grafikkarte bzw. auf Mehrkernsystemen zu tätigen, wurden mehrere Frameworks entwickelt, unter anderem CUDA von NVidia und AMDs Stream Framework. Diese waren aber nicht formal spezifiziert und oft herstellerabhängig. Ein CUDA-Programm ist z. B. nur auf GPUs von NVidia lauffähig.

Als Lösung für dieses Problem wurde Ende 2008 das OpenCL-Framework in der Version 1.0 veröffentlicht, unter anderem Apple, NVidia, AMD und IBM unterstützt. OpenCL ermöglicht das Erstellen von parallel laufenden Programmen auf GPUs, CPUs und sogar FPGAs. Man spricht allgemein von **heterogenen Systemen**. Inzwischen ist OpenCL bei Version 1.2 angelangt, wobei diese noch nicht von allen Herstellern unterstützt wird.

### 5.3 Arten der Parallelität

Ob ein Problem mit Hilfe einer Grafikkarte gut lösbar ist, hängt vor allem davon ab, ob es gut *parallelisierbar* ist. Es wird daher in diesem Abschnitt der Begriff Parallelität definiert. Außerdem werden die verschiedenen Ausprägungen von Parallelität, die OpenCL unterstützt, erläutert. Schließlich wird begründet, wieso sich Stams Methode gut für die Berechnung mit OpenCL eignet.

Parallelität oder auch *Nebenläufigkeit* liegt im Allgemeinen vor, wenn mehrere Operationss-

tröme unabhängig voneinander in einem Zeitschritt fortschreiten können ([27]). Ein klassisches Beispiel für nebenläufige Operationsströme bieten *Threads* in Programmiersprachen wie Java. Während ein Thread beispielsweise auf Daten von einem Netzwerksocket wartet, kann ein anderer einen Ladebalken anzeigen. Mit Hilfe von Threads können sowohl vollkommen verschiedene als auch gleiche Operationen (z. B. eine bestimmte Funktion mit verschiedenen Eingaben) mehrfach parallel ausgeführt werden.

In OpenCL werden insgesamt zwei Arten von Parallelität unterschieden: **Taskparallelität** und **Datenparallelität**. Der Unterschied liegt darin, dass entweder mehrere *Aufgaben* (Tasks) auf die nebenläufigen Operationsströme verteilt werden oder mehrere *Datenblöcke*.

Die oben genannten Threads werden meistens für *taskparallele* Probleme eingesetzt. Ein Problem wird in mehrere Aufgaben unterteilt, die möglichst unabhängig voneinander auf den Recheneinheiten ausgeführt werden können und eventuell auch unabhängig voneinander Ergebnisse produzieren. Die Operationsströme und die Eingangsdaten der verschiedenen parallel laufenden Einheiten sind eventuell sehr unterschiedlich voneinander.

Bei einem *datenparallelen* Problem hingegen wendet man denselben Operationsstrom (das-selbe Programm) auf jeweils unterschiedliche Eingabedaten an. Das einfachste Beispiel für Datenparallelität ist *Single Instruction Multiple Data* (SIMD), wo das Programm meist aus einer einzigen Operation besteht. Die Addition zweier Arrays ist beispielsweise ein Problem, das auf diese Weise lösbar ist. Für jedes Arrayelement wird ein Prozessor verwendet, der folglich nur Logik für die Addition zweier Zahlen benötigt.

Die nebenläufig arbeitenden Prozesse beeinflussen sich in diesem Beispiel nicht gegenseitig. Dies ist aber nicht zwingend für ein datenparalleles Problem. OpenCL unterstützt Synchronisationsmechanismen, mit denen mehrere Operationsströme untereinander kommunizieren können.

Aufwändiger datenparallele Probleme enthalten *if*-Abfragen und können so unterschiedliche Ausführungspfade je nach Eingabedaten nehmen. Diese Art der Parallelität nennt man *Single Program Multiple Data* (SPMD, [22]).

Grafikkarten sind aus historischen Gründen sehr gut für datenparallele Probleme geeignet, denn die Umsetzung der Grafikpipeline stellt ebenfalls ein derartiges Problem dar.

Die Funktionen in Stams Verfahren sind alle datenparallel gestaltet. Als Eingabe dient immer ein dreidimensionales Vektor- oder Skalarfeld, bei dem jeder Voxel für sich bearbeitet werden kann (unter Zuhilfenahme seiner Nachbarschaft im Eingabefeld), wodurch wieder ein Vektor- bzw. Skalarfeld entsteht. Auch das später vorgestellte Partikelsystem ist ein datenparalleles Problem, denn jede Schneeflocke kann für sich bearbeitet werden (im hier gewählten Modell beeinflussen sich Flocken nicht gegenseitig).

## 5.4 Architektur

Die OpenCL-Architektur ist in 4 Teile geteilt, die nun separat erläutert werden sollen:

1. Das **Platform-Model** bietet eine abstrakte Beschreibung eines heterogenen Systems. Dazu gehört der **Host** als Steuereinheit sowie die **Devices** des Systems wie Grafikkarten, CPUs und integrierte Chips.
2. Das **Execution-Model** beschreibt den Begriff **Kernel** als ein Programm, das auf dem

Device ausgeführt wird, sowie die Mechanismen um Kernel zu starten und auszuführen.

3. Das **Memory-Model** beschreibt die unterschiedlichen Speichertypen, die OpenCL bereitstellt.
4. **OpenCL-C** stellt momentan die einzige Sprache dar, mit der parallele Programme in OpenCL verfasst werden können.

#### 5.4.1 Platform-Model

OpenCL soll sowohl auf CPUs als auch auf GPUs und sogar „exotischeren“ Systemen wie FPGAs eingesetzt werden. Daher müssen als Basis allgemeine Begriffe zur Beschreibung von Systemen mit mehreren Prozessoren und Komponenten definiert werden. Dies geschieht im Platform-Model.

OpenCLs Platform-Model ist in Abbildung 5.2 dargestellt. Es ist sehr allgemein gehalten, sodass es sich auf eine Vielzahl von Hardwaresystemen sinnvoll übertragen lässt. Im Modell existiert genau einen **Host**, der mehrere **Devices** verwaltet. Ein Device ist zum Beispiel eine Grafikkarte oder eine CPU. Ein typisches Desktopsystem als Host könnte beispielsweise drei Devices enthalten: eine CPU und zwei Grafikkarten.

Ein Device hat mehrere **Compute Units** (CU), die wiederum **Processing Elements** (PE) enthalten. Die eigentlichen Berechnungen finden auf den Processing Elements statt. Auf einer CU laufen meistens mehrere PE parallel und kommunizieren gegebenenfalls untereinander über CU-eigene Speicher und Synchronisationspunkte. Die Unterscheidung zwischen CU und PE muss auf Hardware-Ebene allerdings nicht vorhanden sein, sie ist aber für das Execution-Model wichtig (siehe unten).

Eine Grafikkarte besitzt im Allgemeinen wesentlich mehr Compute Units und Processing Elements als eine CPU.

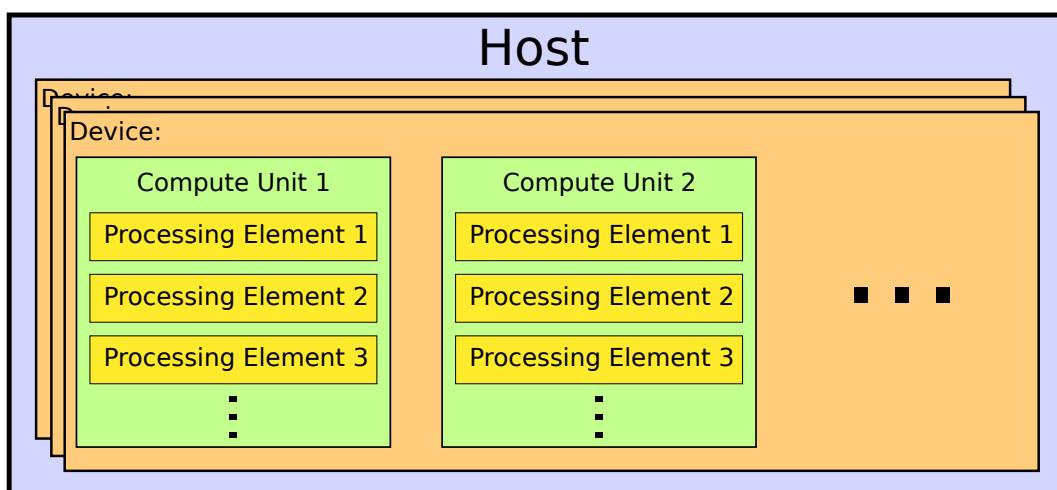


Abbildung 5.2: Platform-Model in OpenCL

### 5.4.2 Execution-Model

Eine OpenCL-Anwendung wird gesteuert durch das Host-Programm (das auf dem Host läuft). Das Host-Programm ist in einer Programmiersprache wie Java oder C++ geschrieben und ist zuständig für die Reservierung von Speicher auf den Devices, das Abfragen von Device-Eigenschaften und dem Starten der sogenannten **Kernel**.

Kernel sind Programme, die auf dem Device ausgeführt werden und somit die eigentliche parallel ausgeführte Logik enthalten. Sie erhalten vom Host-Programm die vorher reservierten Speicherbereiche auf dem Device sowie weitere Parameter. Ansonsten verhalten sie sich ähnlich einer Funktion in einer Programmiersprache. Sie können den Speicher des Devices, auf dem sie ausgeführt werden, sowohl lesen als auch schreiben und sind so in der Lage, Ergebnisse zu produzieren.

Listing 1 zeigt einen in OpenCL-C geschriebenen Kernel, der zwei Elemente aus den Arrays *a* und *b* addiert und das Ergebnis in ein Array *result* zurückschreibt. An diesem Beispiel zeigt sich bereits, dass OpenCL auf datenparallele Probleme zugeschnitten ist: Der Kernel schreibt nur ein einzelnes Arrayelement mit Index *i* zurück. Das gesamte Array wird dadurch bearbeitet, dass man den Kernel mehrfach (parallel) mit unterschiedlichen Indizes instanziert. Das Execution-Model legt fest, wie Kernel parallel auf einem Device *ausgeführt* und *gestartet* werden. Wie Kernel *geschrieben* werden, wird hingegen im Abschnitt 5.4.4 erklärt.

---

**Listing 1** Ein Beispiel-Kernel zum Addieren zweier Arrays.

---

```
kernel void add_arrays(
    global float const *a,
    global float const *b,
    global float *result)
{
    // Gibt die ID der aktuellen Instanz des Kernels zurueck,
    // siehe unten.
    size_t i = get_global_id(0);
    result[i] = a[i] + b[i];
}
```

---

Bei der folgenden Erklärung werden alle Mechanismen weggelassen, die mit Taskparallelität zusammenhängen, da diese Art der Parallelität in dieser Arbeit keine Rolle spielt.

Um einen Kernel wie den obigen auszuführen, werden neben dem Namen des Kernels die folgenden Daten angegeben:

- Das Gerät, auf dem der Kernel ausgeführt werden soll.
- Eine *n*-dimensionale **globale Arbeitsgröße**  $\mathcal{G}$ .
- Eine optionale *n*-dimensionale **lokale Arbeitsgröße**  $\mathcal{L}$ .
- Benutzerdefinierte Eingabedaten (reservierte Speicherbereiche, Parameter).

Die globale Arbeitsgröße bestimmt, wie viele Instanzen des gegebenen Kernels *insgesamt* gestartet werden. Es wird damit allerdings noch nichts darüber ausgesagt, wie viele Instanzen gleichzeitig *parallel* ausgeführt werden. Man kann ein-, zwei-, oder dreidimensionale Arbeitsgrößen angeben.

Um mit dem Kernel aus Listing 1 das `result`-Array komplett zu füllen, müsste man also  $G = \text{Länge}(\text{result})$  wählen (also eine eindimensionale Arbeitsgröße). Das Device startet dann entsprechend viele *Instanzen* des Kernels, die auch **Workitems** genannt werden.

Jedes Workitem steht für sich und führt den Code aus, den der Kernel vorgibt. Zusätzlich zu den Eingabedaten, die vom Benutzer angegeben werden, erhält ein Workitem weitere Parameter, darunter seine eindeutige *globale ID*. Sie beginnt im Beispiel bei 0 und geht bis einschließlich  $\text{Länge}(\text{result}) - 1$ . Im allgemeinen ist die ID  $n$ -dimensional und richtet sich nach der Angabe für  $G$ . Die globale ID wird im Code häufig als Index in ein Array benutzt. Da sie eindeutig ist, gibt es keine Konflikte mit anderen parallel laufenden Workitems, die auch in das Array schreiben.

Oft reicht diese Form der Ausführung aus, grade wenn die Workitems untereinander nicht miteinander kommunizieren müssen. Intern fasst das Device allerdings mehrere Workitems zu **Workgroups** zusammen. Die Besonderheit bei Workgroups ist, dass die darin enthaltenen Work-Units auf derselben *Compute Unit* ausgeführt werden. Dies hat zweierlei Konsequenzen:

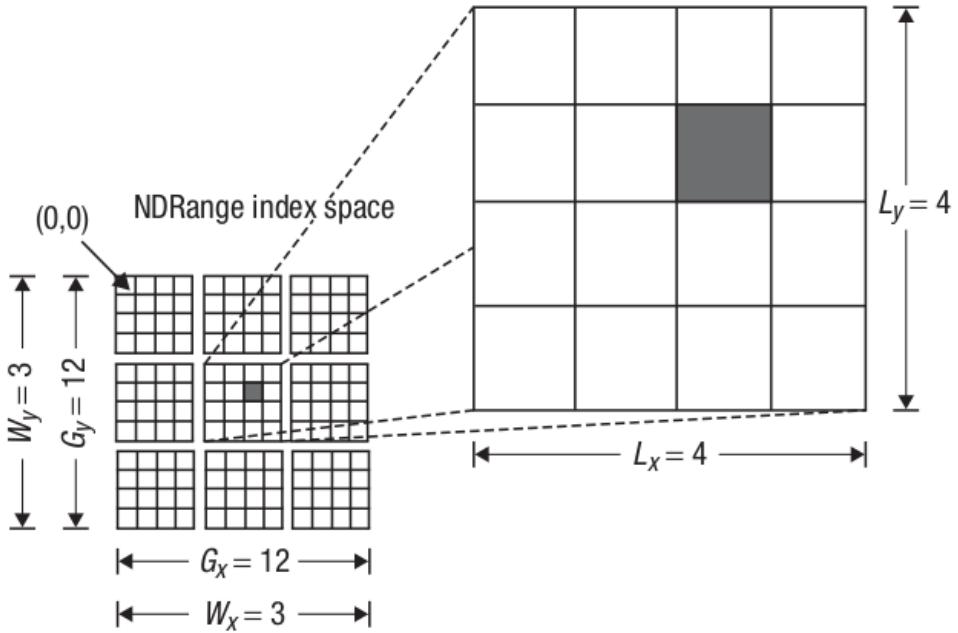
1. Die Workitems einer Workgroup können über **lokalen Speicher** auf der CU Daten untereinander austauschen (siehe Abschnitt 5.4.3). Dies kann als Cache-Mechanismus sehr nützlich sein, denn der CU-interne Speicher ist wesentlich schneller als der sonst zur Verfügung stehende globale Speicher.
2. Die Workitems können sich mit Hilfe von Barrieren beeinflussen. Beispielsweise könnten an einer Stelle im Code alle Workitems aufeinander warten und dann erst gemeinsam fortfahren.

Man kann die Größe der Arbeitsgruppe bei der Ausführung auch explizit angeben, muss aber darauf achten, dass die Arbeitsgruppengröße in jeder Dimension durch die globale Arbeitsgröße teilbar ist. Gibt man die Arbeitsgruppengröße nicht an, wählt die OpenCL-Implementierung eine angemessene Größe aufgrund einer Heuristik. Jedes Workitem erhält vom Device zusätzlich zu seiner globalen ID immer eine *lokale ID*, die das Item innerhalb seiner Workgroup eindeutig identifiziert. Außerdem kann ein Workitem seine *group ID* abfragen. Im Übrigen ist nicht garantiert, dass die Workitems einer Workgroup parallel ausgeführt werden. Abbildung 5.3 zeigt ein zweidimensionales Beispiel einer Kernelausführung.

#### 5.4.3 Memory-Model

Da OpenCL auf vielen verschiedenen Systemen lauffähig sein soll, müssen — ähnlich wie beim Platform-Model — allgemeine Begriffe geschaffen werden, welche die für den Benutzer verfügbaren Speicherbereiche beschreiben.

Es gibt zwei Arten von Speicherobjekten in OpenCL: **buffer objects** und **image objects**. Buffer objects stellen rohe Speicherblöcke einer bestimmten Größe dar, die der Anwender



**Abbildung 5.3:** Beispiel zur Ausführung eines Kernels. Die globale Arbeitsgröße ist  $(12, 12)$ , die Arbeitsgruppengröße ist  $(4, 4)$ , es entstehen also  $3 \times 3$  Gruppen. Jedes Workitem erhält eine lokale ID (im Bild markiert  $(2, 1)$ ) und eine globale ID (im Bild markiert  $(6, 5)$ ) ([27]).

mit beliebigem Inhalt füllen kann. Sie sind in allen Implementierungen nutzbar und nur durch den verfügbaren Speicher begrenzt.

Image objects hingegen sind zur Speicherung von zwei- und dreidimensionalen Bildern gedacht. Sie haben ihren Platz in OpenCL gefunden, da GPUs spezielle Hardware zur Adressierung von Bildern enthalten, auf die man in OpenCL aus Performancegründen eventuell zurückgreifen will. Beim Erstellen eines image object gibt man seine Größe und das gewünschte Format an, wobei nicht alle theoretisch verfügbaren Formate von allen Geräten unterstützt werden und die Größe beschränkt ist. Beim Zugriff auf ein Bild muss man den Adressierungs-, und den Filtermodus angeben, sowie die Koordinaten des Bildpunktes, den man laden will.

Es ist auch möglich, mit einer Fließkommakoordinate auf ein Bild zuzugreifen. In diesem Fall bestimmt der Filtermodus, wie Werte zwischen den Bildpunkten interpretiert werden. Ein Device muss nicht zwingend image objects unterstützen. Außerdem sind dreidimensionale Texturen nicht aus Kernels heraus beschreibbar, im Gegensatz zu Zweidimensionalen. Im OpenCL gibt es 5 verschiedene Speicherbereiche:

**Hostspeicher** Diese Art Speicher ist nur für das Host-Programm verfügbar.

**Globaler Speicher** Auf diesen Speicherbereich können alle Workitems sowohl lesend als auch schreibend zugreifen. Der Zugriff kann zudem wahlfrei geschehen, wobei Lesezugriffe je nach Device gecached sein können. Die Zugriffszeit ist ansonsten stark davon abhängig, wie auf den Speicher innerhalb einer Workgroup zugegriffen wird.

**Konstanter Speicher** Dieser Speicherbereich bleibt während die Ausführung eines Kernels konstant. Er wird am Anfang einmal geschrieben und steht allen Workitems danach nur lesend zur Verfügung. Zugriff auf den konstanten Speicher ist allerdings sehr performant.

**Lokaler Speicher** Auf diesen Speicherbereich können alle Workitems innerhalb einer Workgroup lesen und schreibend zugreifen. Im Allgemeinen ist der Zugriff sehr schnell, da der Speicher oft auf der CU verbaut ist.

**Privater Speicher** Dieser Speicherbereich ist nur für das aktuelle Processing Element verfügbar und enthält unter anderem lokale Variablen.

Abbildung 5.4 zeigt alle Speichertypen im Überblick. Es ist erkennbar, dass das Host-Programm lediglich den konstanten und den globalen Speicher lesen und schreiben kann. Da der Transfer von Host-Speicher zu Device-Speicher allerdings vergleichsweise langsam vonstatten geht, wird im Allgemeinen versucht, wenige Speichertransfers durchzuführen und auf dem Device zu bleiben.

Um ein buffer object zu erzeugen, ruft das Host-Programm die Funktion `clCreateBuffer` auf und übergibt:

- die Größe des Buffers in Bytes
- das Device, auf dem der Buffer erstellt werden soll<sup>2</sup>
- Flags, die angeben, ob der Buffer für die Kernel nur lesbar oder auch schreibbar sein soll
- Optional der initiale Inhalt des Buffers

Um einen Buffer vom Host-Programm aus zu schreiben oder auszulesen, kann man den Inhalt des Buffers mit dem Kommando `clEnqueueMapBuffer` auf einen Speicherbereich des Host-Speichers übertragen. Nachdem man aus diesem Bereich gelesen oder in den Bereich geschrieben hat, muss man ihn mit `clEnqueueUnmapBuffer` wieder freigeben. Bei einer Schreiboperation wird der Buffer daraufhin wieder zum Device übertragen.

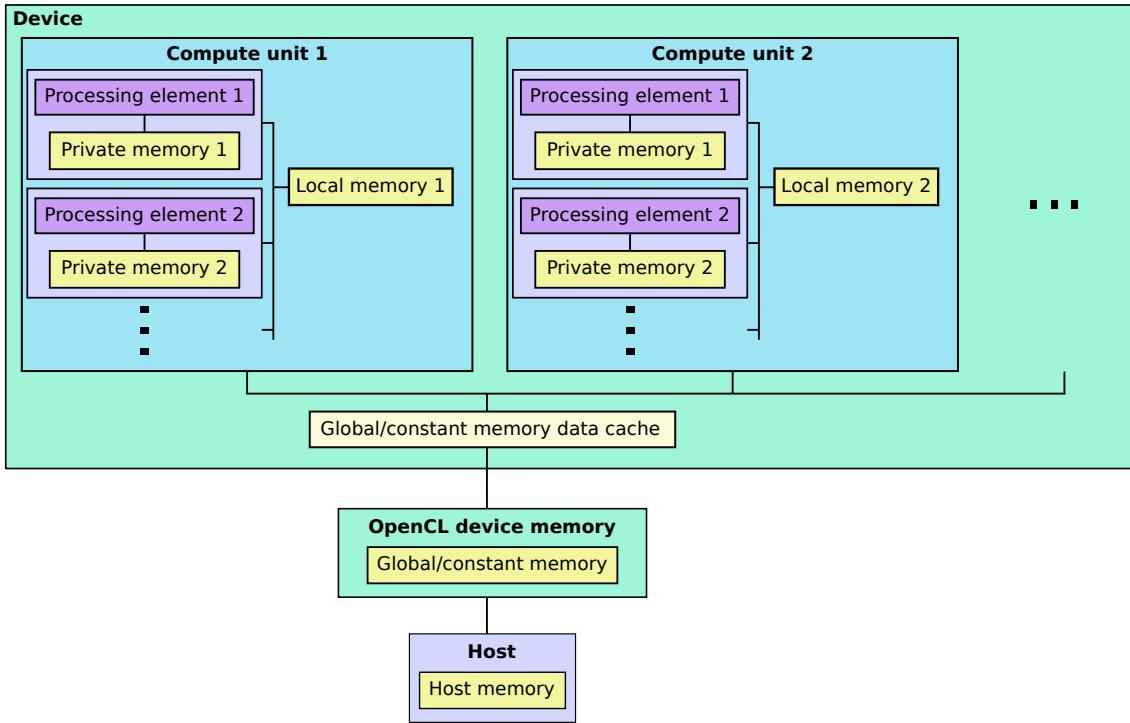
Um ein (zweidimensionales) image object zu erstellen, ruft man die Funktion `clCreateImage2D` auf und übergibt die folgenden Parameter:

- die Breite und Höhe des Bildes
- das Format, zum Beispiel `CL_RGB`, `CL_FLOAT` für ein Bild mit drei float-Farbkanälen
- Flags, die angeben, ob das Bild nur lesbar oder auch schreibbar sein soll (analog zu buffer objects)
- Optional der initiale Inhalt des Bildes

Analog zu buffer objects gibt es Kommandos, um den Inhalt von und zum Host zu übertragen.

---

<sup>2</sup>hier wird auf die Einführung des **Kontext** verzichtet und nur vom Device gesprochen.



**Abbildung 5.4:** Ein Überblick über die verschiedenen Speicherbereiche, die OpenCL vorgibt, und der Zusammenhang mit dem Execution-Model.

Es ist außerdem möglich, Speicher zwischen OpenGL und OpenCL zu teilen, sowohl für image objects als auch buffer objects. Dies wird unter anderem beim Partikelsystem für die Schneeflocken dazu verwendet, Eigenschaften der Schneeflocke wie die Position direkt im OpenGL-Vertexbuffer zu verändern. So werden Speicherplatz und Kopierkosten gespart. Allerdings muss sichergestellt sein, dass alle OpenCL-Operationen auf dem entsprechenden Buffer abgeschlossen sind, bevor der Buffer von OpenGL verwendet werden kann. Der umgekehrte Fall, also der Abschluss aller OpenGL-Operationen auf dem Buffer, muss ebenfalls sichergestellt werden. Dies ist momentan nur möglich, indem man die Kommandos `glFinish` und `c1Finish` verwendet. Diese schließen allerdings global alle noch laufenden Operationen auf *allen* Buffern ab und blockieren in dieser Zeit das Hauptprogramm. Es kann daher zu Performanceproblemen kommen.

#### 5.4.4 OpenCL-C

In diesem Abschnitt wird die Programmiersprache OpenCL-C eingeführt, sowie die Zusammenhänge zu den vorher besprochenen Abschnitten deutlich gemacht. Es soll dabei zunächst der Weg vom fertig geschriebenen OpenCL-C-Programmtext zu seiner Ausführung auf einem Device deutlich gemacht werden. Bei den Erklärungen wird zwischen dem Host-Programm und dem OpenCL-C-Programm hin- und hergesprungen. Dabei ist zu beachten, dass die von der OpenCL-Runtime bereitgestellten Funktionen immer mit „cl“ beginnen. Diese Funktionen werden also immer auf dem Host ausgeführt. Im Anschluss werden einige Besonderheiten von OpenCL-C gegenüber C99 herausgestellt.

OpenCL-C ist momentan die einzige Programmiersprache, welche die OpenCL-Spezifikation zur Erstellung von OpenCL-Kerneln formal beschreibt. Sie basiert auf der ISO-genormten Sprache C99 (genauer ISO/IEC 9899:1999), wobei einige sprachliche Erweiterungen und eine große Standardbibliothek ergänzt wurden. Im Folgenden wird minimales Grundwissen über C99 vorausgesetzt.

OpenCL-C-Programme werden entweder in separaten Textdateien gespeichert und im Host-Programm in einen String geladen, oder sie werden direkt im Host-Programm als Stringliteral eingebunden. Der String mit dem Programmcode wird dann für ein bestimmtes Device<sup>3</sup> mit Hilfe der Funktionen `clCreateProgramWithSource` und `clBuildProgram` kompiliert und es entsteht ein **program object**. Beim Kompilieren lassen sich Optimierungsflags angeben. Ein Beispiel soll die folgenden Erklärungen verdeutlichen:

---

```

1 // Hilfsfunktion zum Quadrieren eines Werts.
2 // Achtung: Nicht von aussen aufrufbar!
3 float square(float x)
4 {
5     return x*x;
6 }
7
8 // Quadriert ein Array und addiert eine Konstante.
9 kernel void square_and_add(
10     global float *a,
11     private float b)
12 {
13     private float current_value = a[get_global_id(0)];
14     a[get_global_id(0)] = square(current_value) + b;
15 }
16
17 constant float4 null_value = (float4)(0.0f);
18
19 // Setzt alle Pixel eines 2D-Bildes auf 0 zurueck.
20 kernel void null_image(
21     global write_only image2d_t bild)
22 {
23     write_imagef(
24         bild,
25         (int2)(
26             get_global_id(0),
27             get_global_id(1)),
28         (float4)(
29             0.0f));
30 }
```

---

Aus dem **program object** lassen sich nun mittels `clCreateKernel` ein oder mehrere **kernel**-

<sup>3</sup>Streng genommen ist es eine Liste von Devices.

**nel object** erzeugen. In OpenCL-C sind Kernel normale C99-Funktionen mit Rückgabetyp `void`, die mit dem Attribut `kernel` ausgestattet sind. Nur diese Funktionen können von außen als „Einsprungpunkte“ für die OpenCL-Runtime benutzt werden. `square_and_add` und `null_image` im obigen Code sind Kernel, `square` hingegen nicht. Die Funktion `c1CreateKernel` erhält das `program object` und den Namen des Kernels als Parameter.

Die Kernelfunktionen in OpenCL-C erhalten Parameter, die vom Host-Programm vor der Ausführung des Kernels gesetzt werden müssen, z. B. die Parameter `a, b` der Funktion `square_and_add`. Die Funktion `c1SetKernelArg` erhält dazu das `kernel object`, den Index des Parameters, den man setzen will, sowie den Parameter selber.

Buffer objects werden in OpenCL-C als *Pointer* repräsentiert. Die Sprache sieht keine mehrdimensionalen Strukturen vor. Diese müssen mittels Indextransformationen selber geschrieben werden. Image objects haben den speziellen Typ `image2d_t` bzw. `image3d_t` und müssen mit `read_only` bzw. `write_only` qualifiziert werden (gleichzeitiger Lese- und Schreibzugriff ist aufgrund von Hardwarebeschränkungen nicht möglich). Zum Schreiben steht die Funktion `write_imagef`, zum Lesen die Funktion `read_imagef` zur Verfügung.

Sind alle Parameter gesetzt, kann man der Funktion `c1EnqueueNDRangeKernel` das `kernel object` sowie die in Abschnitt 5.4.2 beschriebenen Parameter übergeben und der Kernel wird auf dem Device ausgeführt.

In OpenCL ist es wichtig, jede Variable mit einem Attribut auszustatten, die den Speicherbereich der Variable angibt. Mit Ausnahme des Hostspeichers gibt es für jeden der in Abschnitt 5.4.3 beschriebenen Bereiche ein Schlüsselwort: `private`, `constant`, `local` und `global`. Schreibt man den Speicherbereich nicht explizit hin, wird `private` angenommen. Da Speicherbereiche nicht gemischt werden dürfen, ist es nicht möglich, einem `global float *p` einen Pointer `local float *u` zuzuweisen.

OpenCL-C definiert neue Datentypen für Vektoren. Diese heißen genauso wie die skalaren Typen in C und haben ihre Dimension angehängt (z. B. `float4` im Beispiel oben). Für die Dimension sind allerdings nur 1, 2, 4, 8 und 16 möglich. Mit OpenCL-1.1 ist die Dimension 3 hinzugekommen. Für Vektoren sind die arithmetischen Operatoren in natürlicher Weise überladen, sodass man zwei Vektoren beispielsweise einfach mit `a+b` addieren kann. Vektorliterale sind von der Form `(Typ)(a,b,c,...)`. Als Kurzform steht auch `(Typ)(a)` zur Verfügung, was den kompletten Vektor mit dem Wert `a` initialisiert. Auf diese Form der Initialisierung wird in der Implementierung häufiger zurückgegriffen. Auf die einzelnen Komponenten eines Vektors kann man mit dem `.`-Operator wie auf die Komponenten einer Struktur zugreifen, also z. B. `v.x`.

OpenCL-C bietet außerdem eine umfangreiche Standardbibliothek. Sie enthält vor allem mathematische Funktionen, die allerdings auch für die Vektortypen überladen sind. Die Funktion `sin` zur Berechnung des Sinus steht beispielsweise auch für Vektoren zur Verfügung und arbeitet dort komponentenweise. Auch „echte“ Vektorfunktionen wie das Kreuz-, und das Skalarprodukt (`cross, dot`) stehen zur Verfügung und können auf manchen Systemen in schnelle SIMD-Instruktionen übersetzt werden. Statt die verwendeten Bibliotheksfunctionen komplett aufzuführen, werden sie an passender Stelle erklärt der Implementierung erklärt.

Die Sprache listet allerdings einige Einschränkungen, damit sie auch auf minimalen Prozessoren lauffähig ist:

- Rekursive Funktionen sind nicht erlaubt.

- Funktionspointer sind nicht erlaubt.
- Pointer auf Pointer sind zwar erlaubt, aber nicht als Kernelparameter.
- Dynamische Speicherverwaltung (normalerweise mit Hilfe der Funktionen `malloc` und `free`) ist nicht vorgesehen.
- Die C-Standardbibliothek steht nicht zur Verfügung.

## 5.5 Beispielcode

Um das Zusammenspiel aller OpenCL-Komponenten zu verdeutlichen, soll hier der Programmcode vorgestellt werden um zwei Arrays einzulesen, sie mit Listing 1 auf Seite 42 zu addieren und das Ergebnis auszugeben. Es wird leicht vereinfachter C-Code verwendet und z. B. Fehlerbehandlung weggelassen.

Wir nehmen an, die Variable `device` enthielt das Gerät, auf dem der Code ausgeführt werden soll. Außerdem sei die Länge der Arrays auf 100 festgelegt. Das Host-Programm beginnt damit, die nötigen buffer objects zu erstellen:

---

```
// Diese beiden Arrays koennten vom Benutzer auf der
// Kommandozeile eingelesen werden
float erstes_array[100], zweites_array[100];

cl_mem a = clCreateBuffer(
    device,
    CL_MEM_READ_ONLY,
    100 * sizeof(float),
    erstes_array);

cl_mem b = clCreateBuffer(
    device,
    CL_MEM_READ_ONLY,
    100 * sizeof(float),
    zweites_array);

cl_mem result = clCreateBuffer(
    device,
    CL_MEM_READ_WRITE,
    100 * sizeof(float),
    // Der Inhalt dieses Buffers wird nicht vorgegeben.
    NULL);
```

---

Wie bereits angesprochen, erhält die Funktion `clCreateBuffer` die Größe des Buffers in Bytes (daher das `sizeof(float)`) und zudem ein Flag, welches angibt, ob der Buffer nur gelesen (`CL_MEM_READ_ONLY`) oder auch geschrieben werden kann (`CL_MEM_READ_WRITE`).

Die Buffer, die als Eingabe für den Algorithmus dienen, werden direkt mit den vom Benutzer vorgegebenen Arrays gefüllt und sind im weiteren nur lesbar. Der `result`-Buffer bleibt uninitialisiert (es wird `NULL` als Inhalt übergeben), denn er wird vom Kernel gefüllt. `clCreateBuffer` liefert ein Handle vom Typ `cl_mem` zurück, welches an andere Funktionen weitergereicht werden kann.

Danach wird das `program object` aus dem Quellcode erstellt und kompiliert. Das zurückgegebene `cl_program`-Handle kann dann zur Erstellung eines `kernel object` verwendet werden. Es wird angenommen, dass der Quelltext des Programms in einem String `program_source_code` gespeichert ist:

---

```
cl_program program = clCreateProgramWithSource(
    device,
    program_source_code);

clBuildProgram(
    program,
    // Optionale Parameter fuer den Compiler
    "");
```

---

```
cl_kernel kernel = clCreateKernel(
    program,
    "add_arrays");
```

---

Dem so konstruierten `cl_kernel`-Handle können nun mittels der Funktion `clSetKernelArg` die Buffer übergeben werden. Die OpenCL-Runtime unterstützt die Zuweisung von Kernelparametern nur mittels Index, nicht via Name:

---

```
clSetKernelArg(
    kernel,
    0,
    a);
clSetKernelArg(
    kernel,
    1,
    b);
clSetKernelArg(
    kernel,
    2,
    result);
```

---

Das Programm ist jetzt bereit zur Ausführung:

---

```
clEnqueueNDRangeKernel(
    device,
```

```
kernel,  
// Globale Arbeitsgroesse  
100,  
// Keine lokale Arbeitsgroesse  
NULL);
```

---

Wenn die Funktion zurückgibt, hat das Device seine Arbeit verrichtet und das Ergebnis liegt in `result`. Dieses buffer object liegt allerdings im globalen Device-Speicher. Um die Werte z. B. auf der Konsole auszugeben, wird der Inhalt in den Host-Speicher übertragen.

```
float result_host_memory[100];  
  
float *result_array = clEnqueueMapBuffer(  
    result);  
  
copy(result_array, result_host_memory);  
  
clEnqueueUnmapBuffer(  
    result);
```

---

Die Variable `result_host_memory` enthält  $a + b$ , wobei komponentenweise addiert wurde.

## 6 OpenGL

In diesem Abschnitt wird das OpenGL-Framework vorgestellt, das in der Implementierung verwendet wurde. Es wird nur ein grober Überblick gegeben und speziell auf die programmierbaren Shader eingegangen, da diese im Weiteren noch von größerer Bedeutung sind. Es werden bei der Erklärung teilweise OpenCL-Begriffe verwendet, da sich gewisse Unterschiede und Gemeinsamkeiten zwischen den Frameworks ergeben.

OpenGL stellt eine Programmierschnittstelle bereit, welche abstrakte Operationen zur Darstellung von Grafik mit der GPU zur Verfügung stellt. Der Aufbau der Komponenten, die zum Rendering nötig sind, wird in dem Modell der Grafik- bzw. Rendering-Pipeline zusammengefasst (siehe Abbildung 6.1).

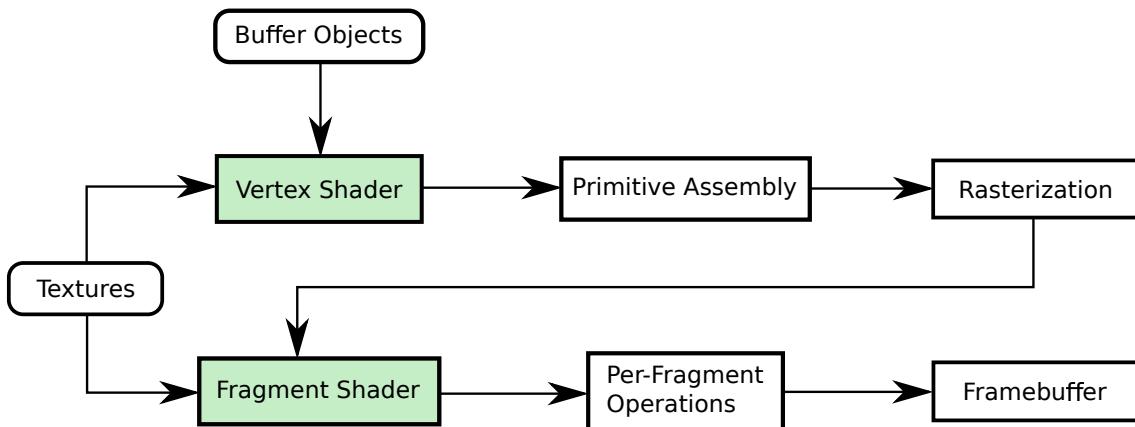


Abbildung 6.1: Die OpenGL-Grafikpipeline. Programmierbare Schnittstellen sind hervorgehoben.

Wie in OpenCL gibt es grundsätzlich zwei Datenstrukturen, die in OpenGL als Eingabe fungieren können: Buffer und Texturen. Buffer werden im Allgemeinen zur Speicherung geometrischer Daten verwendet, d. h. Arrays von Vertizes und Primitiven. Texturen gleichen image objects in OpenCL. Es sind mehrdimensionale Datenstrukturen mit speziell definierten Attributen und Operationen, die z. B. verschiedene Arten von Interpolation erlauben. Sie können ein-, zwei- oder dreidimensionale Bilddaten repräsentieren.

Die zu verarbeitenden Objekte liegen als Vertizes in einer definierten Reihenfolge in Form von Vertex- und Indexbuffern vor. Beim Zeichnen gibt man eine Topologie für diese Vertizes an. In dieser Arbeit werden Punkte (Point Sprites) und Dreiecke als Topologien verwendet. Dreiecksnetze nennt man auch **Meshe**s. Sie werden z. B. aus einer Modelldatei geladen oder algorithmisch erstellt (siehe Abschnitt 9.3).

Die einzelnen Vertizes werden über einen **Vertex Shader** verarbeitet, der als Ausgabe Vertexattribute setzt oder modifiziert, wie etwa Position, Texturkoordinaten und Normalen. Die transformierten Vertizes werden dann über fest eingegebene, d. h. nicht programmierbare, Verarbeitungsschritte zu Grafikprimitiven zusammengesetzt und gerastert.

Dabei wird bestimmt, welche Teile der Geometrie sichtbar sind und welche Primitive welche Bereiche des Bildes (z. B. Framebuffer) überdecken. Das Ergebnis der Rasterung sind Fragmente, die im **Fragment Shader** manipuliert werden können. Ein Fragment ist eine Datenstruktur, welche die zu einem einzelnen Pixel gehörigen Daten eines Primitivs enthält.

Mehrere Fragments können zu einem Pixel gehören, etwa bei überlappender Geometrie, aber ein Fragment beeinflusst immer nur höchstens ein Pixel des fertigen Bildes.

Jedes Fragment besteht aus festen Datenfeldern, wie Tiefeninformation und Bildschirmkoordinaten. Zusätzlich kann das Shaderprogramm beliebige weitere vertexbezogene Daten vom Vertexshader übergeben bekommen. Die Transformierung von Vertexdaten in Fragmentdaten geschieht im Rasterungsschritt durch Interpolation. Die Ausgabe eines Fragmentshaders besteht aus Farb- und Tiefeninformation, die am Ende der Pipeline in eine spezielle Datenstruktur, den sogenannten Framebuffer, geschrieben werden. Im einfachsten Fall ist dies der Puffer, der das am Bildschirm dargestellte Bild im Speicher der Grafikkarte repräsentiert. Es sind aber über sogenannte Framebuffer Objects auch andere benutzerdefinierte Ausgaben möglich, die geschrieben und z. B. als Texturen in weiteren Renderdurchgängen durch die Pipeline wiederum als Input für andere Shader verwendet werden können. Die in dieser Arbeit verwendeten Shader sind in der OpenGL Shading Language (GLSL) geschrieben, einer höheren Programmiersprache mit C-ähnlicher Syntax, die zur Ausführung auf der Grafikkarte mit OpenGL zu binärem Shadercode kompiliert wird.

## 7 Windsimulation

### 7.1 Einleitung

In diesem Abschnitt wird die Umsetzung der Windsimulation vorgestellt. Da dieser Teil der Implementierung nicht mit der Darstellung zusammenhängt, wird nur OpenCL verwendet. Der Abschnitt knüpft direkt an Abschnitt 4 an.

Zuerst wird die Wahl der Datenstrukturen besprochen. Es kommt hier darauf an, eine möglichst performante Struktur zu wählen, die optimale Zugriffszeiten liefert. Danach wird das Verfahren von Stam in OpenCL-Code übertragen, dessen Bestandteile detailliert erläutert werden.

### 7.2 Wahl der Datenstruktur

Für die Simulation des Winds ist es wichtig, eine gute Datenstruktur zur Speicherung der verwendeten Felder zu verwenden, um die Speicherzugriffszeiten zu minimieren.

Wie bereits angesprochen bietet OpenCL zur Speicherung die Wahl zwischen buffer objects und image objects (sofern image objects auf der Zielplattform unterstützt werden). Beide Speicherarten haben leicht unterschiedliche Anwendungsgebiete, daher müssen erst die Anforderungen erarbeitet werden, die an die Datenstruktur gestellt werden.

In vielen Kernen ist man daran interessiert, den Wert für die „aktuelle“ Zelle im Gitter sowie den Wert aller Nachbarzellen zu bestimmen. Der „Wert“ kann je nach betrachtetem Feldtyp entweder Druck, Geschwindigkeit oder etwas anderes bedeuten. Bei der Advektion wird außerdem linear zwischen den Voxeln in einer Nachbarschaft *interpoliert*.

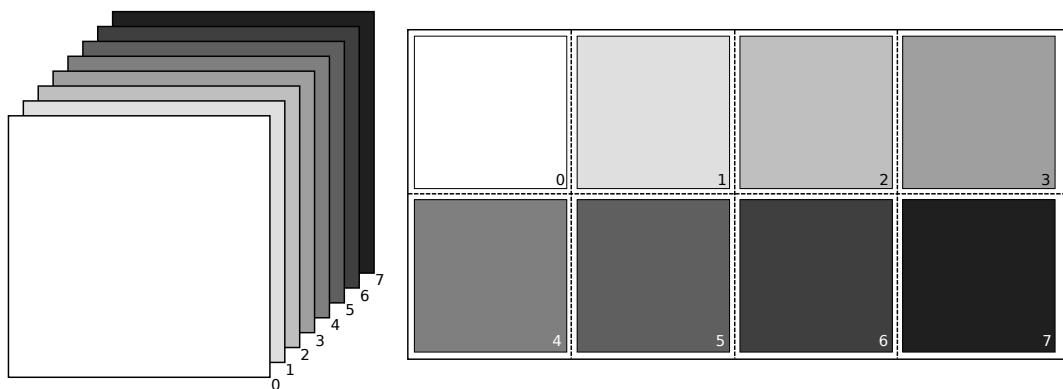


Abbildung 7.1: Veranschaulichung einer Flat-3D-Textur

Schnelle Zugriffe auf einen Voxel samt seiner Nachbarn sind ein Performancemerkmal von image objects. Sie würden sich daher für viele Kernel anbieten, und viele GPU-Implementierungen von strömungsmechanischen Verfahren nutzen in der Tat image objects (wenn auch oft aus historischen Gründen). Allerdings ist das Schreiben von 3D-Texturen in OpenCL-1.1 nur mit einer Extension möglich. Als Hilfsmaßnahme greift man daher üblicherweise zu sogenannten **Flat-3D-Texturen** ([16]). Hier werden die einzelnen „Scheiben“ der dreidimensionalen Struktur nebeneinander in eine zweidimensionale Textur geschrieben (siehe Abbildung 7.1).

Für die Umrechnung zwischen 3D- und 2D-Koordinaten ist etwas Rechenaufwand nötig. 2D-Texturen erlauben außerdem natürlicherweise nur zweidimensionale Interpolation zwischen den Zellen. Benötigt man hingegen dreidimensionale Interpolation, ist auch hier etwas Mehraufwand zu verrechnen.

Image objects werden in einer optimierten Weise gespeichert, sodass Leseoperationen beschleunigt werden ([34]). Diese optimierte Speicherung führt aber dazu, dass das Schreiben in Texturen relativ langsam ist. Selbst durchgeführte Benchmarks zeigen, dass buffer objects sogar im Zweidimensionalen schneller sind als image objects. Daher wurden image objects als mögliche Speicherform verworfen und es wurden einzig Buffer verwendet. Diese erweisen sich außerdem als kompatibler, denn OpenCL-Implementierungen müssen image objects nicht unterstützen.

Da Buffer eindimensional sind, muss zwischen drei- und eindimensionalen Koordinaten konvertiert werden. Sei  $(x, y, z)$  eine dreidimensionale Koordinate und seien  $(w, h, d)$  die Dimensionen eines Buffers (er enthält also  $w \cdot h \cdot d$  Elemente). Dann erhält man einen eindimensionalen Index  $i$  mittels:

$$i = w \cdot h \cdot z + w \cdot y + x \quad (7.1)$$

Umgekehrt erhält man die  $(x, y, z)$ -Koordinaten eines Index mittels

$$\begin{aligned} x &= i \bmod w \\ y &= (i/w) \bmod h \\ z &= i/(w \cdot h). \end{aligned} \quad (7.2)$$

## 7.3 Verfahren nach Stam in OpenCL

### 7.3.1 Allgemeines

Es werden nun die einzelnen Schritte des Verfahrens in OpenCL-Kernel umgewandelt. Der Teil des Programms, der auf dem Host läuft, soll hier nicht in Gänze erarbeitet werden. Stattdessen soll deutlich werden, in welcher Reihenfolge die Kernel aufgerufen werden und welche Daten beim Aufruf gelesen und geschrieben werden. Aus Gründen der Lesbarkeit werden einige Bezeichner abgekürzt, aber immer so, dass aus der Erläuterung bzw. den Kommentaren hervorgeht, was sich dahinter verbirgt.

Die Simulation benötigt einige persistente Daten. Das sind solche, die zwischen den Simulationsschritten beibehalten werden und nicht nur temporär für Berechnungen verwendet werden:

- ein Buffer **boundaries** vom Typ **float**, der 1.0 enthält, wenn die zugehörige Gitterzelle ein Hindernis enthält und 0.0 wenn nicht. Das Befüllen dieses Buffers wird in Abschnitt 10 beschrieben.
- ein Buffer **velocity** vom Typ **float3**, der das aktuelle Geschwindigkeitsfeld enthält. Dieses Feld wird anfangs auf  $(0, 0, 0)$  gesetzt. Der Wind wird manuell von einer Seite in die Simulation gespeist (siehe Abschnitt 7.3.3) und breitet sich dann aus.
- Viele der Algorithmen benötigen außerdem das aktuelle „Zeitdelta“ **float dt**. Es gibt die Zeit an, die seit dem letzten Simulationsdurchlauf vergangen ist. Mit Hilfe

dieser Größe wird die Simulation zeitlich normiert. Lange Zeitabstände zwischen den diskreten Frames sollen zu größeren Veränderungen im Windfeld führen.

Die meisten Kernel werden mit einer dreidimensionalen Arbeitsgröße initialisiert, die der Größe des Gitters entspricht. Ein Work-Item erhält also einen dreidimensionalen Vektor als globale ID, welcher die Gitterposition angibt, die das Work-Item bearbeiten soll.

Da die buffer objects mit den Feldern eindimensional sind, bietet es sich an, den Übergang von einem drei- zu einem eindimensionalen Index in eine Funktion auszulagern. Diese Funktion kann gleichzeitig testen, ob die gegebene Position den Rand des Gitters verlässt, und in dem Fall den nächstgelegenen Voxel am Rand zurückliefern. So lassen sich auf einfache Weise alle Nachbarn eines Voxels aus dem Buffer extrahieren. Die zuständige Funktion `i4p` ist im Folgenden dargestellt:

---

```
// Bekommt die Gittergroesse s und eine 3D-Position p.
// Liefert einen Index.
uint i4p(uint3 p, uint3 s)
{
    // Komponentenweise in gueltigen
    // Bereich [0,n - 1] transformieren
    p = clamp(
        p,
        (uint3)(0),
        s - 1);

    return s.w * s.h * p.z + s.w * p.y + p.x;
}
```

---

Für die Indextransformation wurde Gleichung (7.1) verwendet. Die Standardbibliotheksfunktion `clamp` stellt sicher, dass ein Wert innerhalb eines Intervalls bleibt:

$$\text{clamp}(x, a, b) = \max\{a, \min\{b, x\}\} \quad (7.3)$$

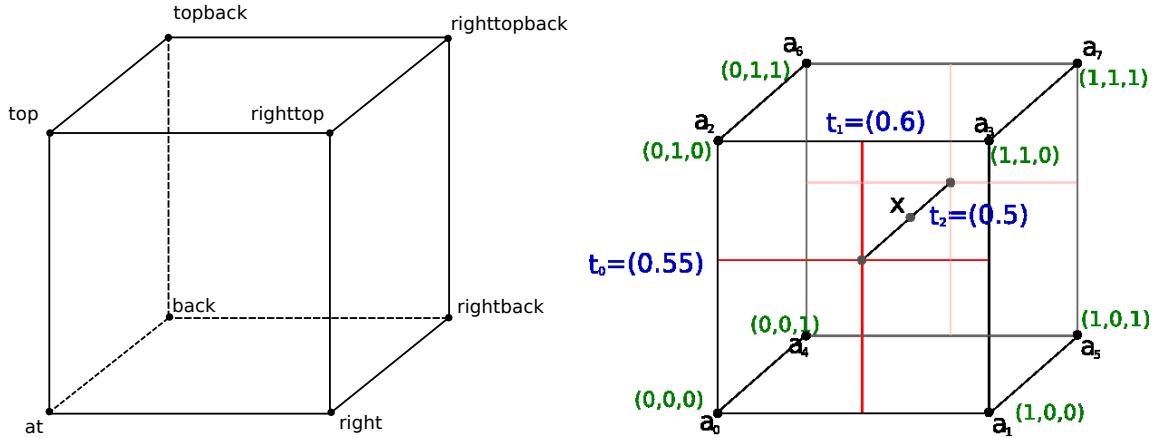
### 7.3.2 Advektion

Der erste Schritt im Verfahren von Stam ist die Advektion des Vektorfeldes `velocity` (vgl. Abschnitt 4.4). Der dazugehörige Kernel ist in mehrere Funktionen und Strukturen aufgeteilt, die nun im Einzelnen erklärt werden.

Vom aktuellen Voxel aus verfolgt man den Geschwindigkeitsvektor in `velocity` zurück und landet dabei, weil der Vektor aus Fließkommawerten besteht, im Allgemeinen zwischen 8 Gitterzellen. Diese 8 Zellen nutzt man für eine lineare Interpolation um die neue Geschwindigkeit zu finden. Es bietet sich an, zunächst eine Struktur zu definieren, um einen Geschwindigkeitswert sowie die direkten Nachbarn zu speichern (siehe Abbildung 7.2a).

---

```
typedef struct
{
```



(a) Veranschaulichung der Nachbarschaftsstruktur neighbors.

(b) Dreidimensionale Interpolation mit dem Beispieldatenvektor  $(0.55, 0.6, 0.05)$

Abbildung 7.2: Die dreidimensionale Nachbarschaft neighbors

---

```

float3 at;
float3 right,top,back,righttop;
float3 rightback,topback;
float3 righttopback;
} neighbors;
```

---

Eine Instanz dieser Struktur wird von der zugehörigen Funktion `neighbors_for_pos` zurückgegeben, die eine beliebige (diskrete) Position im Gitter sowie die Größe des Gitters als Eingabe erhält. Die Funktion greift auf `i4p` zurück und ist so gegen Zugriffe außerhalb des Gitters geschützt:

---

```

// Parameter genau wie i4p, zusätzlich "b"
neighbors neighbors_for_pos(
    global float3 *b,
    uint3 p,
    uint3 s)
{
    neighbors result;
    result.at = b[i4p(p,s)];
    result.right = b[i4p(p+(uint3)(1,0,0),s)];
    result.top = b[i4p(p+(uint3)(0,1,0),s)];
    result.back = b[i4p(p+(uint3)(0,0,1),s)];
    result.righttop = b[i4p(p+(uint3)(0,1,1),s)];
    result.rightback = b[i4p(p+(uint3)(1,0,1),s)];
    result.topback = b[i4p(p+(uint3)(0,1,1),s)];
    result.righttopback = b[i4p(p+(uint3)(1,1,1),s)];
    return result;
}
```

---

---

Mit Hilfe einer Instanz von `neighbors` und einem beliebigen Vektor im Einheitsintervall  $[0, 1]^3$  soll ein interpolierter Vektor erzeugt werden. Hier hilft die OpenCL-C-Funktion `mix`, die anhand eines dritten Wertes linear zwischen zwei Werten interpoliert:

$$\text{mix}(a, b, x) = (1 - x) \cdot a + x \cdot b \quad (7.4)$$

Diese Funktion wird speziell auf Grafikkarten in hoch effektiven Code übersetzt, da sie sich auf sogenannte „multiply-add-Operationen“ (auch „MAD“ genannt) zurückführen lässt. Darunter versteht man skalare Operationen der Form  $a \cdot x + b$ , also eine Multiplikation gefolgt von einer Addition. Da dies in der Grafikpipeline häufig auftritt, sind die Arithmetikchips einer GPU speziell darauf ausgelegt. Auf `mix` wird in der unten stehenden `interpolate_neighbors` mehrfach zurückgegriffen:

---

```
float3 interpolate_neighbors(
    neighbors n,
    // v ist ein Vektor im Intervall [0,1]^3
    float3 v)
{
    float3
    v1 = mix(
        mix(n.at, n.right, v.x),
        mix(n.top, n.righttop, v.x),
        v.y),
    v2 = mix(
        mix(n.back, n.rightback, v.x),
        mix(n.topback, n.righttopback, v.x),
        v.y);

    return mix(v1,v2,v.z);
}
```

---

Das Funktionsprinzip ist am Beispiel in Abbildung 7.2b verdeutlicht. Es wird erst vier mal auf der  $x$ -Achse interpoliert, dann zweimal auf der  $y$ -Achse und schließlich einmal auf der  $z$ -Achse.

Der eigentliche Advektionskernel erhält als Eingabe ein quellenfreies Geschwindigkeitsfeld `input` und schreibt die Ausgabe in das Feld `output`:

---

```
kernel void advection(
    global float *boundary,
    global float3 *input,
    global float3 *output,
    // Das Zeitdelta
    float dt,
```

```

// Gibt die Gittergroesse an
uint3 size)
{
    uint3 position = (uint3)(
        get_global_id(0),
        get_global_id(1),
        get_global_id(2));

    uint index = i4p(position, size);

    if(boundary[index] != 0.0f)
    {
        output[index] = (float3)(0.0f);
        return;
    }

    float3
    v = input[index],
        // Konvertiere mit convert_float3 die diskrete Position
        // in eine Fliesskommposition. Fuehre dann Advektion durch.
        advected = convert_float3(position) - dt * v,
        // fract liefert den Fliesskommanteil eines Vektors.
        fractions = fract(advectecd_vector);

    neighbors n = neighbors_for_pos(
        input,
        position,
        size);

    output[index] = interpolate_neighbors(
        n,
        fractions);
}

```

---

### 7.3.3 In- und Outflow

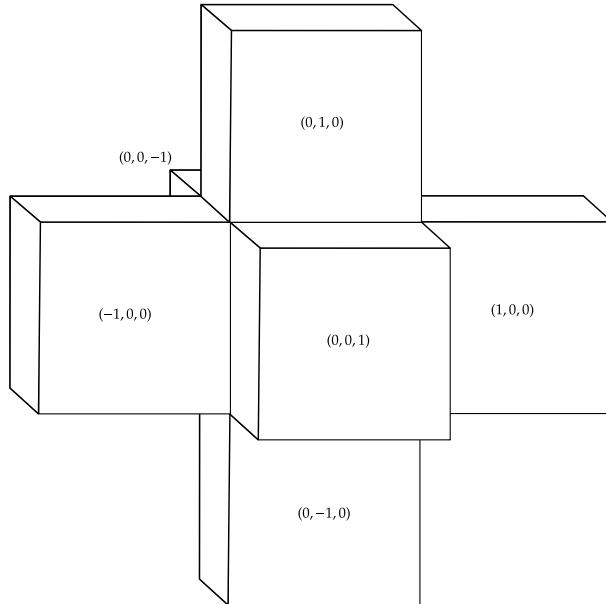
Nach der Advektion werden die In- und Outflowboundaries behandelt (vgl. Abschnitt 4.6.4). Es wird hier, im Gegensatz zu den anderen Kerneln, nicht das gesamte Feld als globale Arbeitsgröße gewählt, sondern nur eine Teilmenge des Gitters. Die Kernel sind so einfach gehalten, dass sie hier nur erläutert, aber nicht im Code vorgestellt werden.

Die Inflow-Boundaries gestalten sich sehr simpel. Es wird ein Vektor vorgegeben, der die momentane Windstärke und -richtung angibt. Dann wird eine Seite des Simulationswürfels mit diesem Vektor gefüllt. Stams Verfahren sorgt dafür, dass sich der Wind langsam im Simulationsbereich ausbreitet.

Bei den Outflow-Boundaries werden die 6 Randseiten des Würfels separat behandelt. Jede

Seite wird in die jeweils benachbarte „Scheibe“ kopiert; für die linke Seite in die Scheibe rechts daneben, die Oberseite in die Scheibe einen darunter, und so weiter.

### 7.3.4 Projektion



**Abbildung 7.3:** Die dreidimensionale Von Neumann-Nachbarschaft

Nach der Advektion ist das Vektorfeld  $\vec{u}$  nicht mehr quellenfrei und der Projektionsoperator muss angewendet werden (vgl. Abschnitt 4.6). Dazu bestimmt man zuerst die Divergenz  $\text{div } \vec{u}$  des Vektorfeldes und löst dann das Gleichungssystem  $\Delta x = \text{div } \vec{u}$  nach  $x$  mit Hilfe des Jacobiverfahrens. Der Gradient von  $x$  wird schließlich von  $\vec{u}$  subtrahiert und das Feld ist wieder quellenfrei.

Man benötigt also zunächst die Divergenz des Vektorfeldes. Analog zur Advektion wird eine Struktur definiert um die Von Neumann-Nachbarschaft eines Gitterpunktes zu speichern (siehe Abbildung 7.3):

---

```
typedef struct vn_neighbors
{
    float3 at;
    float3 left, right;
    float3 top, bottom;
    float3 front, back;
    float boundary_at;
    float boundary_left, boundary_right;
    float boundary_top, boundary_bottom;
    float boundary_front, boundary_back;
};
```

---

Die Struktur enthält zusätzlich die zu den Nachbarn gehörigen Hinderniswerte. Die Bestimmung dieser Werte und die der Vektoren verläuft nach demselben Schema wie bei der Advektion und sei im Folgenden ohne den zugehörigen Code mit `vn_neighbors_for_pos` bezeichnet.

Bei der Bestimmung der Divergenz müssen die Randbedingungen beachtet werden. Ist einer der Nachbarn des aktuellen Voxels von einem Hindernis ausgefüllt, wird statt des Vektors an dieser Stelle der *Nullvektor* angenommen:

---

```
vn_neighbors n = vn_neighbors_for_pos(...);
if(n.boundary_left != 0.0f)
    n.left = (float3)(0.0f);
```

---

Statt dies jedoch als eine `if`-Bedingung umzusetzen, wird eine Multiplikation mit dem Randwert verwendet.

Dies röhrt daher, dass sich GPUs suboptimal verhalten, wenn mehrere Work-Units in einer Work-Group unterschiedliche Ausführungspfade durch den Code nehmen. Bei einer `if`-Abfrage, die beispielsweise für nur ein Work-Item `true` ergibt, durchlaufen dennoch *alle* Work-Units der Work-Group den Rumpf des `if`-Statements, wobei die Ergebnisse der „toten“ Ausführungspfade verworfen werden. Natürlich geschieht dies transparent für den Anwender, sodass keine unerwünschten Nebeneffekte eintreten.

Die obige Abfrage lässt sich wie folgt umschreiben:

---

```
vn_neighbors n = vn_neighbors_for_pos(...);
n.left = (1.0f - n.boundary_left) * n.left;
```

---

Hier liegt wieder eine MAD-Operation vor, der Code ist also sehr effizient.

Der zur Divergenz gehörige Kernel hat die folgende Form:

---

```
kernel void divergence(
    global float3 *input,
    global float *output,
    global float *boundary,
    uint3 size)
{
    uint3 position = (uint3)(
        get_global_id(0),
        get_global_id(1),
        get_global_id(2));

    uint index = i4p(position, size);

    vn_neighbors n = vn_neighbors_for_pos(
        input,
        position,
```

---

```

    boundary,
    size);

float3 z = (float3)(0.0f);

// Hier geschieht die Behandlung der Randbedingungen.
// Es wird ausgenutzt, dass die Randwerte die Werte 0 oder 1
// annehmen.
n.left = mix(n.left, z, n.boundary_left);
n.right = mix(n.right, z, n.boundary_right);
n.top = mix(n.top, z, n.boundary_top);
n.bottom = mix(n.bottom, z, n.boundary_bottom);
n.front = mix(n.front, z, n.boundary_front);
n.back = mix(n.back, z, n.boundary_back);

// Da die Gittergroesse 1 betraegt, vereinfacht sich die Formel
// aus dem ersten Abschnitt ein wenig.
output[index] =
    (n.right - n.left).x +
    (n.bottom - n.top).y +
    (n.front - n.back).z;
}

```

---

Um die Ähnlichkeit mit den restlichen Kerneln aufzuzeigen, wurde hier wieder die Funktion `mix` verwendet.

Anschließend wird das Jacobiverfahren durchgeführt. Das Verfahren ist iterativ, der Jacobi-Kernel muss also mehrfach gestartet werden. Um Speicherplatz zu sparen, bietet sich ein „Ping-Pong-Schema“ an. Dazu werden zwei zusätzliche buffer objects `s1`, `s2` benötigt, welche jeweils ein Skalarfeld speichern. Anfangs enthält `s1` die initiale Lösung für die Poisson-Gleichung  $\Delta x = \text{div } \vec{u}$ , also einfach das Nullfeld  $x = 0$ . Eine Anwendung des Jacobi-Kernels füllt `s2` mit der ersten Iteration des Verfahrens. Diese wird zur Eingabe der nächsten Iteration, die `s1` überschreibt, und so weiter. Nach einer geraden Anzahl von Iterationsschritten enthält `s1` die endgültige Lösung. Algorithmus 7.1 zeigt das Verfahren im Pseudocode.

Die Randbedingungen im Jacobi-Kernel werden ähnlich denen im Divergenz-Kernel behandelt. Ist einer der Nachbarn von einem Hindernis ausgefüllt, wird statt des Vektors an dieser Stelle der Vektor im *Zentrum der Nachbarschaft* angenommen. Dies lässt sich erneut mit der Operation `mix` durchführen.

---

```

kernel void jacobi(
    global float *divergence,
    global float *boundary,
    global float3 *input,
    global float *output,
    uint3 size)
{

```

---

**Algorithm 7.1** Der Ping-Pong-Algorithmus für das Jacobiverfahren

---

```
function JACOBI(div u,b)
    s1 ← clCreateBuffer
    s2 ← clCreateBuffer
    fill_buffer(s1,0)
    current_source ← s1
    current_destination ← s2
    clSetKernelArg(jacobi,0,div u)
    clSetKernelArg(jacobi,1,b)
    for i = 0; i < iterations; i++ do
        clSetKernelArg(jacobi,2,current_source)
        clSetKernelArg(jacobi,3,current_destination)
        clEnqueueNDRangeKernel(jacobi)
        swap(current_source,current_destination)
    end for
    return s1
end function
```

---

```
uint3 position = (uint3)(
    get_global_id(0),
    get_global_id(1),
    get_global_id(2));

uint index = i4p(position,size);

vn_neighbors n = vn_neighbors_for_pos(
    input,
    position,
    boundary,
    size);

n.left = mix(n.left, n.at, n.boundary_left);
n.right = mix(n.right, n.at, n.boundary_right);
n.top = mix(n.top, n.at, n.boundary_top);
n.bottom = mix(n.bottom, n.at, n.boundary_bottom);
n.front = mix(n.front, n.at, n.boundary_front);
n.back = mix(n.back, n.at, n.boundary_back);

float sum     = n.left + n.right + n.top + n.bottom + n.front + n.back,
      result = (sum - divergence[index]) / 6.0f;

output[index] = mix(
    result,
    center,
```

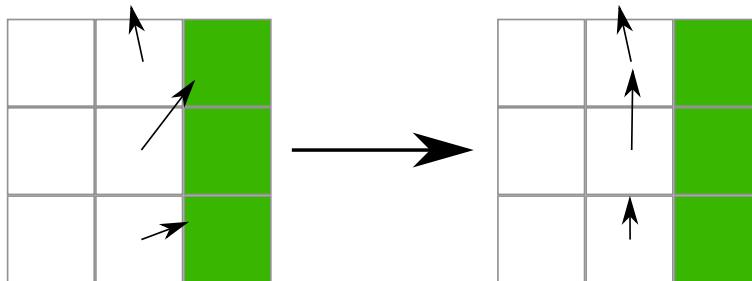
```
1.0f/3.0f);  
}
```

---

Der Code zeigt die gewichtete Jacobi-Iteration mit Wichtungswert  $\frac{1}{3}$ , der sich als günstig herausgestellt hat.

Die Zahl der Iterationen kann von der Leistung der Grafikkarte abhängig gemacht werden. Sie sollte jedoch mindestens 20 betragen. Gute Ergebnisse werden bei 40 Iterationen erzielt. Die Berechnung des Gradienten von  $x$  und die darauf folgende Subtraktion von  $\vec{u}$  lässt sich (auch aus Performancegründen) in einem einzigen Kernel behandeln. In diesem Kernel werden auch die vom Jacobi-Verfahren nur unvollständig gelösten Randbedingungen erzwungen. Der Kernel erfüllt folgende Aufgaben:

1. Er bestimmt den Gradienten des Druckfeldes. Es wird für einen Nachbarwert wieder der Wert im Zentrum genommen, falls die Nachbarzelle von einem Hindernis ausgefüllt ist.
2. Der Gradient wird vom Geschwindigkeitsfeld abgezogen.
3. Die free-slip-Randbedingungen werden erzwungen.



**Abbildung 7.4:** Die Korrektur der nach dem Projektionsschritt eventuell noch unvollständig gelösten Randbedingungen. Die markierten Felder enthalten Hindernisse, die  $x$ -Komponente der Geschwindigkeit wird daher auf 0 gesetzt.

In Schritt 3 wird für jede Koordinatenachse gespeichert, ob sich in der Von Neumann-Nachbarschaft auf der Achse ein Hindernis befindet. Ist dies der Fall, wird die Geschwindigkeitskomponente auf dieser Achse auf 0 gesetzt. Abbildung 7.4 verdeutlicht das Prinzip. Der Koeffizient, der die entsprechende Komponente entweder behält oder gleich 0 setzt, kann als Funktion der beiden Randwerte auf der Achse ausgedrückt werden:

---

```
float boundary_mask(float boundary_left, float boundary_right)  
{  
    return 1.0f - min(1.0f, boundary_left + boundary_right);  
}
```

---

Offenbar ist die Funktion genau dann 0, wenn einer der beiden Werte 0 ist. Die Funktion enthält zudem keine (expliziten) bedingten Anweisungen. Der gesamte Kernel ergibt sich wie folgt:

---

```

kernel void subtract_pressure_gradient(
    global float3 *velocity,
    global float *boundary,
    global float *pressure,
    uint3 size)
{
    // Schritt 1: Bestimme Gradient des Druckfeldes
    uint3 position = (uint3)(
        get_global_id(0),
        get_global_id(1),
        get_global_id(2));

    uint index = i4p(position,size);

    vn_neighbors n = vn_neighbors_for_pos(
        input,
        position,
        boundary,
        size);

    if(boundary[index] == 1.0f)
    {
        velocity[current_index] =
            (float3)(
                0.0f,
                0.0f,
                0.0f);
        return;
    }

    float const center =
        pressure[index];

    // Bei Hinderniszellen wird das Zentrum gewaehlt
    n.left = mix(n.left, n.at, n.boundary_left);
    n.right = mix(n.right, n.at, n.boundary_right);
    n.top = mix(n.top, n.at, n.boundary_top);
    n.bottom = mix(n.bottom, n.at, n.boundary_bottom);
    n.front = mix(n.front, n.at, n.boundary_front);
    n.back = mix(n.back, n.at, n.boundary_back);

    float3 const pressure_gradient =
        (float3)(
            n.right.x - n.left.x,

```

```

    n.bottom.y - n.top.y,
    n.front.z - n.back.z);

// Schritt 2 und 3: Gradient subtrahieren und Randbedingungen erzwingen
float3 const vmask =
(float3)(
    boundary_mask(n.boundary_left,n.boundary_right),
    boundary_mask(n.boundary_top,n.boundary_bottom),
    boundary_mask(n.boundary_front,n.boundary_back));

velocity[current_index] =
    vmask * (velocity[index] - pressure_gradient);
}

```

---

Das so erzeugte `velocity`-Feld kann in der nächsten Iteration wieder in die Advektionsroutine gegeben werden.

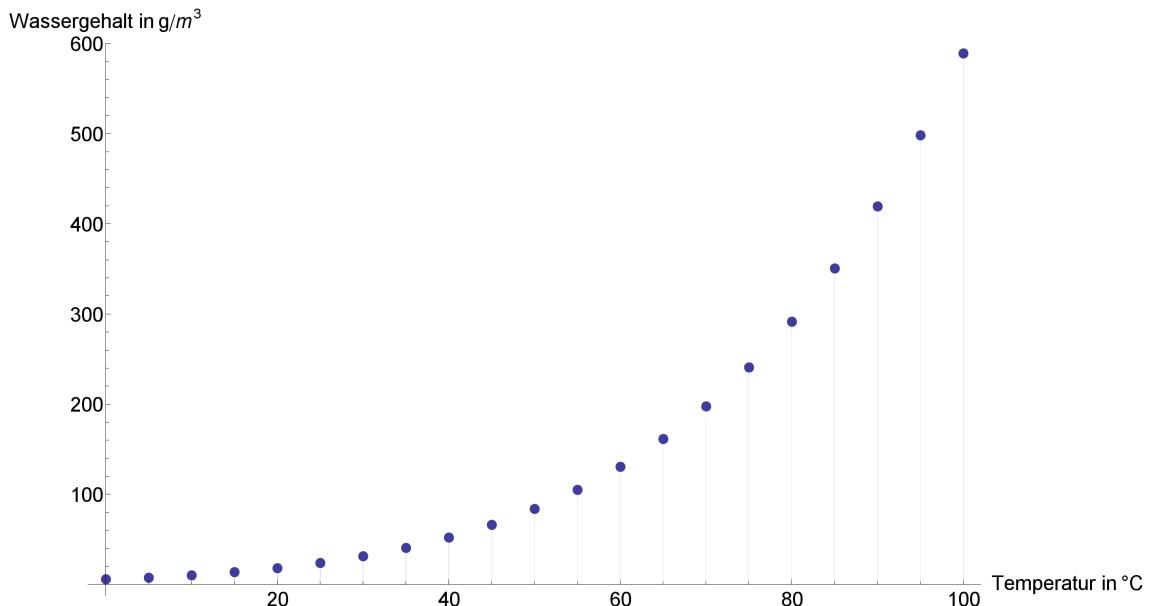
## 8 Fallender Schnee

### 8.1 Hintergründe

Schnee ist, genau wie Regen, eine Spezialform von **Niederschlag**. Hier soll kurz auf die Entstehung von Niederschlag und die Bildung von Schneeflocken eingegangen werden. Der Erklärungsansatz orientiert sich an [47].

Überall dort, wo sich eine freie Wasseroberfläche wie z. B. ein See befindet und die Temperatur einen bestimmten Wert übersteigt, lösen sich Moleküle der Wasseroberfläche von ihrem Verbund und verdunsten in die Luft. Umgekehrt treffen verdunstete Wassermoleküle wieder auf die Wasseroberfläche und kondensieren dort. Die *Kondensationsrate* und die *Verdunstungsrate* geben an, wie viel Wasser in einem Bereich zu einem Zeitpunkt verdunstet und wie viel kondensiert.

Stellt man sich eine Wasseroberfläche bei trockener Luft vor, so ist die Kondensationsrate anfangs 0, da keine Wassermoleküle in der Luft enthalten sind, die kondensieren können. Die Verdunstungsrate ist bei einer ausreichend hohen Temperatur allerdings ungleich 0. Mit der Zeit steigt so die Anzahl der Wassermoleküle in der Luft, die Luft „sättigt“ sich mit Wasser. Dadurch steigt wiederum die Kondensationsrate. Bei gleich bleibenden Bedingungen gleichen sich nach einiger Zeit die Kondensation und Verdunstung aneinander an. Die in diesem Gleichgewichtszustand vorliegende Konzentration von Wassermolekülen in der Luft ist die **Sättigungskonzentration** oder **maximale Luftfeuchtigkeit**. Sie ist umso höher, je wärmer es ist, siehe Abbildung 8.1. Umgangssprachlich sagt man daher, warme Luft könne „mehr Wasser aufnehmen“.



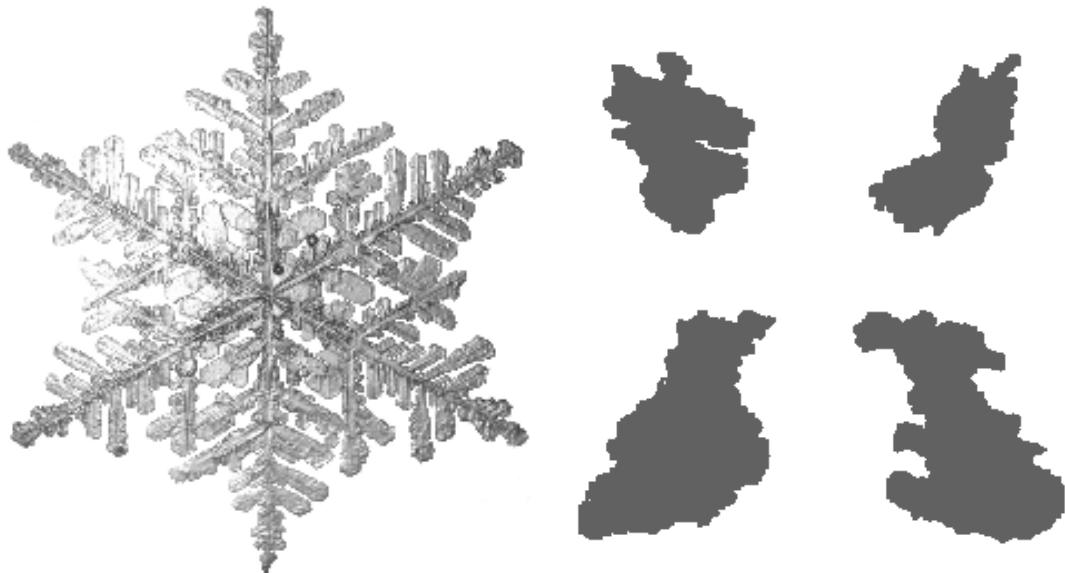
**Abbildung 8.1:** Der exponentielle Zusammenhang zwischen Temperatur und Sättigungsmenge von Wasserdampf in der Luft ([48])

Steigt warme Luft mit hoher Wassersättigung vom Boden in die Atmosphäre auf, kann die Sättigungsmenge in den höheren Luftsichten über den eigentlich maximalen Wert

steigen — es entsteht wieder ein Ungleichgewicht. Um dieses Ungleichgewicht zu kompensieren, kondensiert das überschüssige Wasser an sogenannten **Kondensationskernen** (beispielsweise Staubpartikeln in der Luft) und es bilden sich kleine Wassertröpfchen in Mikrometergröße. Passiert dies großflächig, entstehen Wolken am Himmel. Verdichten sich die kondensierten Tropfen weiter, werden sie irgendwann zu schwer und fallen aufgrund der Schwerkraft herunter, es entsteht *Regen*.

Eine Spezialform der eben beschriebenen Kondensationskerne sind die *Kristallisationskerne*. Bei Temperaturen unter  $0^{\circ}\text{C}$  bilden sich an diesen Kernen keine Tropfen sondern *Kristalle*. Bei starken Minustemperaturen (ab  $-40^{\circ}\text{C}$ ) müssen sie jedoch nicht vorhanden sein, es bilden sich auch so Kristalle.

Diese Kristalle sind anfangs noch sehr klein, sie wachsen erst während ihres Falles zu Boden weiter an. Dabei entstehen aufgrund spezieller Eigenschaften des Wassers charakteristische Formen (siehe Abbildung 8.2a). Durch die Verkettung mehrerer Kristalle entstehen Schneeflocken, die man beim Zubodenfallen beobachten kann. Sie können eine Größe von bis zu 10cm erreichen ([28]) und haben ebenfalls vielfältige Formen, die allerdings keine Symmetrie mehr aufweisen, (vgl. Abbildung 8.2b).

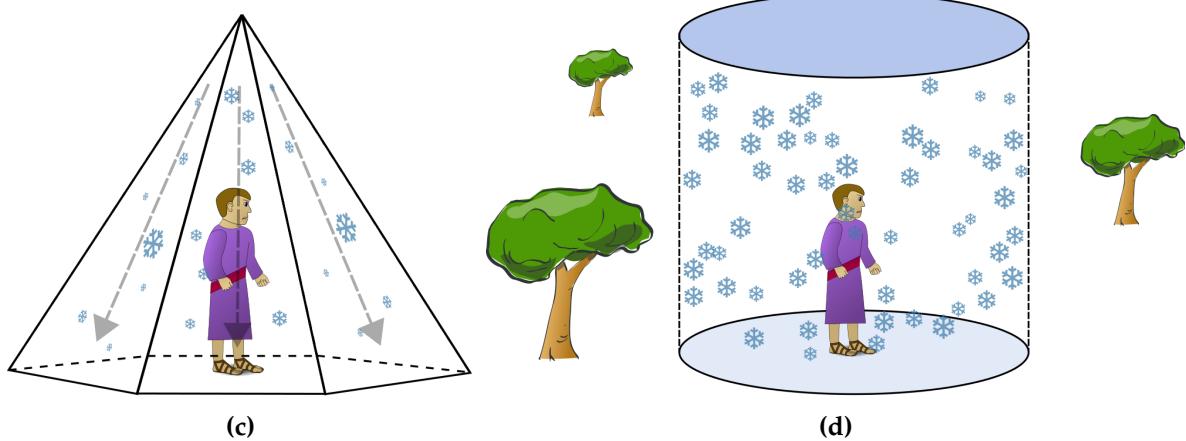


(a) Ein einzelner „Farn-artiger“ Schneekristall ([50]). (b) Kameraaufnahmen einzelner Schneeflocken mit etwa 1–2cm Durchmesser ([14]).

## 8.2 Schnee in interaktiven Anwendungen

Aktuelle interaktive Anwendungen wie Spiele simulieren typischerweise weder die Bildung einer Schneedecke noch den Fall von Schneeflocken in physikalisch motivierter Art und Weise.

Für die Modellierung von Schneeflocken werden in Filmen und Spielen oft lokale Modelle verwendet, die nicht auf ein (globales) Windfeld zurückgreifen, sondern die Flocken lediglich in der Umgebung der Kamera unter Zuhilfenahme von Zufallsvariablen simulieren. Es folgen Beispiele der eingesetzten Verfahren:



**Abbildung 8.2:** (a) Schnee mittels Texturen, hier mit Hilfe eines Kegels um den Betrachter herum. Die Pfeile deuten an, dass die Texturen stetig nach unten gescrollt werden. (b) Ein lokales Modell für die Visualisierung von Schneeflocken. Außerhalb des gezeigten Zylinders werden keine Flocken erstellt.

1. Es werden ein oder mehrere Texturen direkt vor dem Auge des Betrachters eingeblendet. Die Texturen stellen Bilder verschiedener Schneeflocken dar und werden stetig nach unten bewegt um den Eindruck zu erwecken, die Flocken fielen zu Boden.

Die zu den Texturen gehörige Geometrie ist entweder ein Rechteck direkt vor dem Auge des Betrachters (eine texturierte „Glasscheibe“, die sich mit dem Betrachter mitbewegt), oder eine komplexere Form wie ein Kegel oder ein Doppelkegel (siehe [46]). Dieser Ansatz ist extrem eingeschränkt, denn der Blickwinkel darf sich nicht signifikant von einem Horizontalen unterscheiden. Er bietet sich dort an, wo die synthetische Kamera ohnehin nicht frei ist, z. B. bei einem Flugsimulator.

2. Es wird ein **Partikelsystem** verwendet ([37]), also jede Schneeflocke einzeln modelliert und weiterbewegt, was physikalisch wesentlich akkurater ist. Dazu wird meist eine feste Anzahl an Schneeflocken vorgegeben. Die Anzahl richtet sich nach der Rechnerleistung und der Stärke des Schneefalls.

Die Partikel haben Eigenschaften wie eine Position, eine Masse und eine Momentangeschwindigkeit, die am Anfang der Simulation zufällig festgelegt werden. Als Position wird allerdings kein beliebiger, zufälliger Punkt im Raum verwendet. Vielmehr wird ein Vektor generiert, der sich innerhalb eines eingeschränkten Bereiches um den Betrachter herum befindet, z. B. ein Punkt in einem Zylinder um den Betrachter (vgl. Abbildung 8.2d).

Die Flocken werden aufgrund ihrer Geschwindigkeit weiterbewegt. Treffen sie auf dem Boden auf, werden sie neu in die Simulation gesetzt, wobei die Position wieder aus dem Zylinder gewählt wird. Die Geschwindigkeit wird anhand der Schwerkraft verändert. Hat man zudem ein globales Schneemodell zur Verfügung, kann man die Flugbahn der Flocken im Zylinder der Windgeschwindigkeit anpassen.

Problematisch ist die Behandlung von Hindernissen. Bewegt sich der Betrachter beispielsweise in ein Gebäude, sollte bei geschlossener Decke kein Schnee mehr um die

Person generiert werden. Dies muss gesondert behandelt werden. Eine ähnliche Situation ergibt sich, wenn der Betrachter auf der windabgewandten Seite eines Gebäudes steht. Hier sollten ebenfalls keine Schneeflocken generiert werden (oder sehr wenige). Dies ist nur sehr schwer umzusetzen, da der Zylinder in seiner Größe begrenzt ist und sonst ad hoc keine Informationen über die Verteilung von Schnee in der Welt zur Verfügung stehen.

Mit dieser Simulationsmethode lässt sich zudem keine Schneedecke generieren. Es ist zwar möglich, den Auftreffpunkt der Schneeflocken auf den Hindernissen weiterzuverarbeiten. Die so generierte Schneedecke existiert aber nur in einer Umgebung um den Bewegungspfad des Betrachters. Weiter entfernte Objekte sind stets frei von Schnee.

Wegen der besseren physikalischen Modellierbarkeit und wegen des beliebig wählbaren Blickwinkels bei einer Ego-Perspektive wurde entschieden, ein *globales* Partikelsystem umzusetzen. Die Flocken werden also anfänglich im gesamten Simulationsbereich verteilt. Ihre Geschwindigkeit wird aufgrund der Windgeschwindigkeit verändert. Das Auftreffen auf Hindernisse wird für die Schneedecke weiterverarbeitet und die Flocke wird wieder an einen beliebigen, zufälligen Ort gesetzt. Im Folgenden wird zuerst die Visualisierung der Partikel erläutert, danach wird auf die Modellierung eingegangen.

### 8.3 Visualisierung

Bei der Wahl der Visualisierungsmethode müssen mehrere Faktoren in Betracht gezogen werden. Die Simulation soll sowohl ruhige Wetterverhältnisse mit wenigen Schneeflocken als auch Szenen mit starkem Wind und sehr vielen Schneeflocken behandeln können. Daher ist einerseits wichtig, dass einzelne Schneeflocken einen gewissen Detailgrad besitzen, aber die Performance gleichzeitig so wenig wie möglich beeinträchtigen. Es sollen mindestens

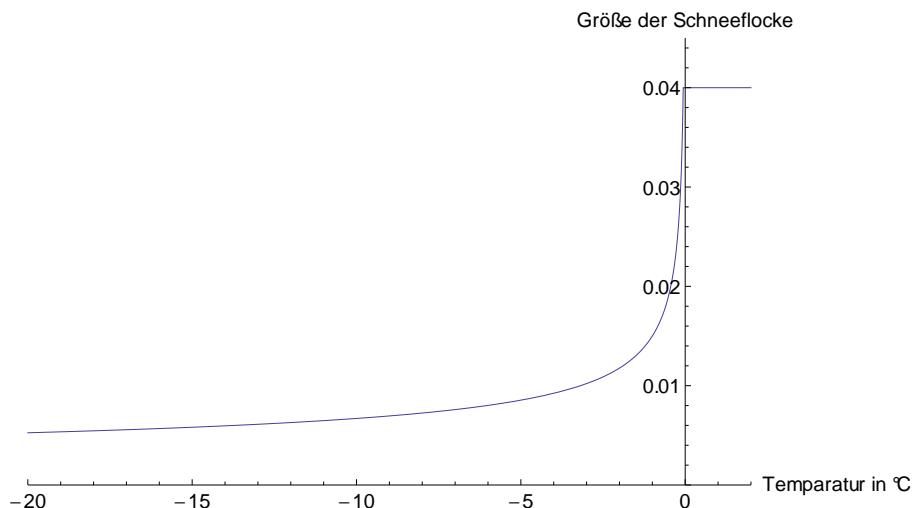


Abbildung 8.3: Der Zusammenhang von Schneeflockengröße und Temperatur nach [17].

die zwei wichtigsten Eigenschaften einer Schneeflocke modelliert werden: die Größe und

das Aussehen. Beide Eigenschaften hängen von der Umgebungstemperatur ab. Für den Durchmesser  $D$  einer Schneeflocke in Abhängigkeit von der Temperatur  $T$  wurde in [17] folgende Relation aufgestellt (siehe Abbildung 8.3):

$$D = \begin{cases} 0.015 \cdot |T|^{-0.35} & \text{für } T \leq -0.061 \\ 0.04 & \text{für } T > -0.061 \end{cases} \quad (8.1)$$

Das Aussehen einer Schneeflocke bestimmt sich primär dadurch, ob *trockener* oder *feuchter* Schnee vorliegt. Nahe bei  $0^\circ\text{C}$  ist der Schnee feucht und hat eine hohe Dichte. Bei kälteren Temperaturen werden die Schneeflocken kleiner und sehen zarter aus.

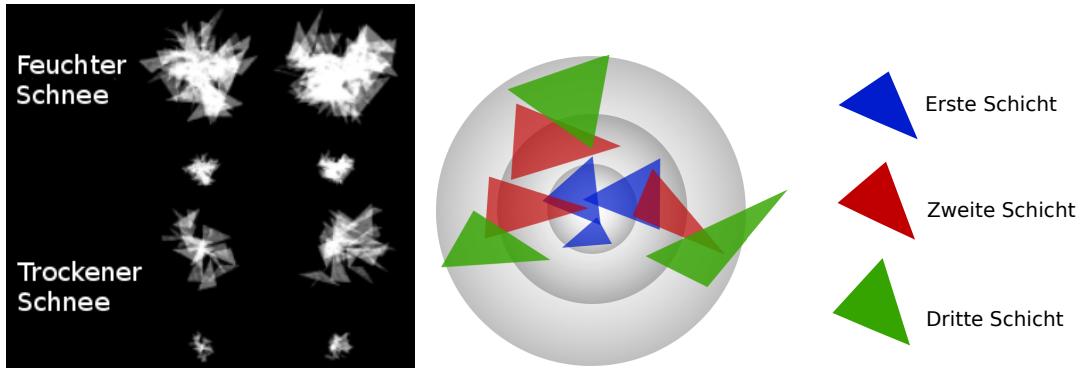
Aagaard hat in [1] ein Modell entwickelt, welches die tatsächliche Form der Schneeflocke abhängig von Dichte und Größe möglichst genau abbildet. Dazu verwendet er zufällig generierte, weiß gefüllte Dreiecke, welche die Eiskristalle darstellen sollen. Anhand des (randomisierten) Durchmessers der Schneeflocke wird eine Anzahl von *Schichten* ermittelt, in der die Dreiecke dann angeordnet werden, siehe Abbildung 8.4a und Abbildung 8.4b. Auf diese Art können vielfältige zufällige Formen generiert und an die aktuellen Wetterbedingungen angepasst werden. Durch die dreidimensionale Modellierung kann man den Flocken außerdem eine Rotation geben und sie von allen Seiten betrachten.

Allerdings ist das Modell sehr laufzeitintensiv. Dies hat zwei Ursachen:

1. Die einzelnen Dreiecke einer Schneeflocke sind halbtransparent. Dies führt dazu, dass weit mehr Fragments erzeugt werden als es Pixel auf dem Bildschirm gibt. Diesen Effekt nennt man *Overdraw*.
2. Es werden außerdem sehr viele kleine Dreiecke für eine einzelne Flocke benötigt (Aagaard geht von mindestens 10 und bis zu 100 aus). Für 1.000 Flocken erhält man so schon 10.000 Dreiecke. Die angestrebte Flockenzahl bei heftigem Schneefall liegt aber bei weit über 1.000 Flocken. Aagaard schlägt daher vor, ein „Level of detail“-System einzubauen, bei dem weiter entfernte Flocken durch simplere Geometrie (z. B. mit weniger Dreiecken) angenähert werden. Er implementiert dies jedoch selber nicht, da Performance kein entscheidender Faktor für die Implementierung ist.
3. Die Eigenschaften einer Schneeflocke wie etwa ihre Geschwindigkeit und die Darstellung der Flocke müssen getrennt werden, da OpenGL es nicht direkt erlaubt, Daten für einen „Block“ von Primitiven anzugeben wie etwa die Dreiecke der Schneeflocke. Dies macht die Implementierung schwieriger und langsamer.

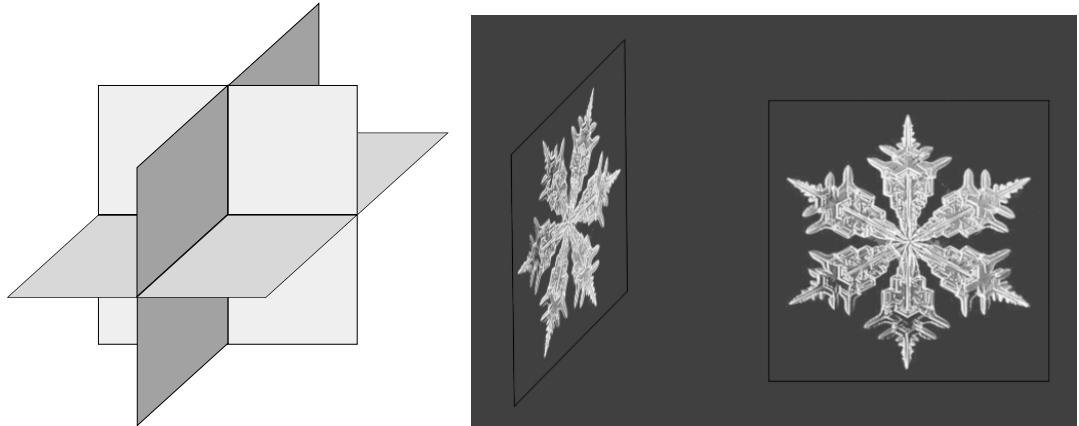
In [38] findet sich ein etwas simplerer Ansatz, der nur drei transparente Rechtecke pro Schneeflocke verwendet (siehe Abbildung 8.5a). Dadurch wird massiv Performance gespart, die Daten für die Flocken müssen allerdings trotzdem in einem separaten Buffer gespeichert werden.

In dieser Arbeit wird daher eine simplere Visualisierung mit Hilfe von sogenannten **Point-sprites** gewählt. Statt mehrerer Dreiecke repräsentiert nur ein einzelner Punkt eine Schneeflocke. Dadurch wird es möglich, alle Daten für eine Flocke im OpenGL-Buffer selber zu speichern. Zudem sind Punkte in OpenGL keine mathematischen Punkte, denn sie besitzen eine *Ausdehnung*. Dadurch wird es möglich, die Flockengröße zu variieren.



(a) Das Resultat von Aagaards 3D-Modell für die Schneeflocken.  
 (b) Der schichtweise Aufbau einer Schneeflocke in Aagaards Modell.

Abbildung 8.4: Aagaards Schneemodell



(a) Eine Schneeflocke aus drei Rechtecken zusammengesetzt.  
 (b) Links ein Schneekristall aus zwei Dreiecken zusammengesetzt, rechts als Pointsprite.

Abbildung 8.5: Einfache Schneemodelle

Diese Ausdehnung ist jedoch nicht in Weltkoordinaten angegeben, sondern in Pixeln auf dem Bildschirm. Das bedeutet, dass die Flocken immer zum Betrachter zeigen, man kann sich nicht um sie herum bewegen (siehe Abbildung 8.5b). Außerdem muss — im Gegensatz zu Dreiecken — eine Formel für die Größe erdacht werden, damit weiter entfernte Punkte kleiner werden (für Dreiecke wird dies mit Hilfe einer perspektivischen Projektionsmatrix sichergestellt).

Um Pointsprites in OpenGL zu zeichnen, gibt man bei der OpenGL-Funktion `glDrawArrays` den Primitivtyp `GL_POINTS` an. Listing 2 zeigt den dazugehörigen Vertexshader, der als Input die Projektionsmatrix, den Augenpunkt, die Größe der Schneeflocke und deren Position bekommt. Mit Hilfe der Variable `gl_PointSize`, der man einen `float`-Wert zuweist, kann man die Größe des Punktes setzen. Die Funktion `determine_point_size` wird gleich erläutert. Der zu Listing 2 gehörige Fragmentshader Listing 3 verwendet den zweidimensionalen Vektor  $gl_FragCoords \in [0, 1]^2$ . Dieser Vektor gibt an, wo sich das Fragment innerhalb des

---

**Listing 2** Der zu Pointsprites gehörige Vertexshader

---

```
uniform mat4 model_view_projection;
uniform vec4 eye_position;

in vec4 position;
in float point_size;

float determine_point_size(float distance_to_camera)
{
    // ...
}

void main()
{
    float distance_to_camera = distance(eye_position,position);
    gl_PointSize = determine_point_size(distance_to_camera);
    gl_Position = model_view_projection * position;
}
```

---

aktuellen Punktes (respektive, der aktuellen Flocke) befindet. OpenGL verwaltet also ein eigenes Koordinatensystem innerhalb des Punktes. Mit Hilfe dessen kann man den Punkten im Fragmentshader nicht nur eine einheitliche Farbe geben, sondern auch eine Textur. Die Textur der Punkte wird in der Implementierung zufällig aus einer Menge von Texturen ausgewählt, die zur aktuellen Außentemperatur passen.

Um den Vertexshader zu komplettieren, muss die Größe der Pointsprites bestimmt werden. Verwendet man für seine Szene eine perspektivische Projektion, besteht zwischen dem Abstand von Objekten und ihrer dargestellten Größe ein nichtlinearer Zusammenhang. Daher sollte auch für die Größe der Pointsprites eine ähnliche Proportionalität bestehen. Microsofts DirectX-Framework verwendet hierfür folgende Formel ([35]):

$$S = S' \sqrt{\frac{1}{A + B \cdot D_e + C \cdot D_e^2}} \quad (8.2)$$

$S'$  bezeichnet hierbei die vorgegebene Punktgröße, die Konstanten  $A, B, C$  sind frei wählbar und  $D_e$  bezeichnet den Abstand des Punktes von der Kamera. Die Werte  $A = 0.96, B = 0.19, C = 0.06$  haben sich als visuell ansprechend herausgestellt.

Für die Simulation von heftigem Schneefall reicht es allerdings nicht aus, nur die Anzahl der Flocken zu erhöhen und deren Größe zu randomisieren. Das menschliche Auge interpretiert die Flocken dann eher als unrealistisches „Rauschen“. Um dies auszumerzen wird in [19] ein Verfahren vorgestellt, mit dem in der Bildebene aus einigen wenigen Schneeflocken weitere synthetisiert werden, sodass ein realistischer Eindruck einer großen Menge Schneeflocken entsteht. Dieses Verfahren ist sehr performant, allerdings relativ schwierig umzusetzen. Daher wurde ein anderer Weg gewählt: die *Transparenz* der Schneeflocken variiert ebenfalls abhängig von der Entfernung zur Kamera (mit Hilfe von Gleichung (8.2)), allerdings mit

---

**Listing 3** Der zu Pointsprites gehörige Fragmentshader

---

```
uniform sampler2D snowflake_texture;

out vec4 fragment_color;

void main()
{
    // Einfarbige Punkte (hier weiss)
    // fragment_color = vec4(1.0, 1.0, 1.0, 1.0);
    // Mit Textur:
    fragment_color = texture(snowflake_texture, gl_FragCoord);
}
```

---

den Konstanten  $A = 0.1$ ,  $B = 0.15$ ,  $C = 0$  und  $S' = 1$ .

Mit Hilfe von Pointsprites ist es möglich, auf einem aktuellen System weit über 100.000 Schneeflocken gleichzeitig anzuzeigen. Es ist jedoch nicht möglich, die Flocken rotieren zu lassen, was allerdings visuell unkritisch ist.

## 8.4 Modellierung

### 8.4.1 Einleitung

Nach der visuellen Beschreibung der Schneeflocken soll nun deren Bewegung erläutert werden. Dazu wird zuerst ein physikalisch motiviertes Modell eingeführt und danach die Implementierungsdetails des Partikelsystems erklärt.

Physikalisch gesehen wird eine Schneeflocke als *starrer Körper* mit einer sehr geringen Masse modelliert. Kennt man die aktuelle Geschwindigkeit  $\vec{v}$  des Körpers, sowie seine momentane Position  $\vec{p}$  und hat man ein Zeitdelta  $\Delta t$  gegeben, dann kann seine Bahn  $\vec{s}$  in diesem Zeitschritt durch das *Weg-Zeit-Gesetz* bestimmt werden, sofern man die momentane Beschleunigung  $\vec{a}$  kennt:

$$\vec{s} = \frac{1}{2}\vec{a}\Delta t^2 + \vec{v}t + \vec{p} \quad (8.3)$$

Dabei nimmt man natürlich an, die Beschleunigung sei in dem Zeitabschnitt  $\Delta t$ , den man grade betrachtet, konstant. Für die Beschleunigung selber gilt Newtons Formel:

$$\vec{F} = m \cdot \vec{a} \quad (8.4)$$

$$\vec{a} = \frac{\vec{F}}{m} \quad (8.5)$$

Man muss also — wie schon beim Windfeld — berechnen, welche Kräfte auf die Schneeflocke wirken. Daraus ergibt sich dann die Flugbahn. Es bietet sich manchmal auch an, eine Kraft nicht als Beschleunigung zu modellieren, sondern sie in Gleichung (8.3) direkt

auf die Geschwindigkeit zu addieren (die Geschwindigkeit der Schneeflocke also nicht zu verändern). Im Folgenden sollen die einzelnen Kräfte näher beleuchtet werden.

In [1] wurden vier Kräfte zusammengetragen, welche zusammen die Bewegung einer Schneeflocke beeinflussen:

1. Die Schwerkraft  $\vec{F}_g$  ist eine konstante Kraft, die — genau wie beim Windfeld — in negativer  $y$ -Richtung wirkt.
2. Der Wind übt eine Kraft  $\vec{F}_w$  auf die Schneeflocke aus, die den **Luftwiderstand** widerspiegelt. Aufgrund der geringen Masse der Flocken ist dies die Kraft, welche die Flocke am meisten in ihrer Bahn beeinflusst.
3. Die aerodynamische Form der Flocke führt dazu, dass sie selbst bei Windstille keine direkte Bahn zum Boden verfolgt, sondern in kleinen Spiralbahnen zur Erde fällt. Dieser Effekt wird **dynamischer Auftrieb** genannt. Er wird jedoch nicht als Kraft modelliert, sondern direkt als Geschwindigkeitsänderung (siehe unten).
4. Auch Schneeflocken haben einen gewissen *statischen Auftrieb*  $F_{\text{Auftrieb}}$  innerhalb der umgebenden Luft.

Diese Kräfte werden nun einzeln besprochen. Es werden OpenCL-C-Funktionen aufgestellt, die die Kräfte abhängig von den Eigenschaften einer Schneeflocke umsetzen. Diese Funktionen werden schließlich in dem Kernel, der die Schneeflocken weiterbewegt, benutzt.

#### 8.4.2 Luftwiderstand

Die Methode von Stam liefert ein dreidimensionales Vektorfeld von Windgeschwindigkeiten. Das bedeutet, man kann in jedem Punkt innerhalb der Simulationsgrenzen eine Windgeschwindigkeit angeben (wenn der Punkt nicht auf dem Gitter liegt, wird stattdessen ein Geschwindigkeitswert interpoliert).

Es ist aber noch nicht klar, welchen Einfluss die Windgeschwindigkeit auf die Geschwindigkeit der Flocke ausübt, welche Kraft also aus der Windgeschwindigkeit resultiert. Im Folgenden wird eine einzelne Schneeflocke an Position  $\vec{p}_f$  mit Geschwindigkeit  $\vec{v}_f$  betrachtet. Mit  $\vec{v}_w$  sei die Geschwindigkeit des Windes an Position  $\vec{p}_f$  bezeichnet.

Der Luftwiderstand — oder allgemeiner der **Strömungswiderstand** in Fluiden — bezeichnet eine Kraft, die ein Fluid der Bewegung eines Körpers entgegensetzt. Bei der Berechnung dieser Kraft kommt es auf die *Relativgeschwindigkeit* von Schneeflocke und Wind an. Herrscht beispielsweise Windstille (ist also  $\vec{v}_w = 0$ ), sorgt die Schwerkraft für eine konstante Beschleunigung nach unten, die Flocke fällt immer schneller zu Boden.

Der Luftwiderstand wirkt dieser Beschleunigung entgegen, und das umso stärker, je schneller die Flocke ist. Dadurch stellt sich irgendwann ein Gleichgewicht zwischen Schwerkraft und Luftwiderstand ein und die Flocke erreicht ihre **Endgeschwindigkeit**  $v_{y,\max}$ . Im Vakuum hingegen könnte eine Schneeflocke theoretisch beliebig schnell fallen.

Streng genommen ist der Luftwiderstand aufgeteilt in **Druckwiderstand** und **Reibungswiderstand**. Deren Anteile hängen von der Form des umströmten Körpers ab. Bei einer Kreisscheibe überwiegt der Druckwiderstand, bei einer Tragfläche eines Flugzeugs beispielsweise der Reibungswiderstand (siehe Abbildung 8.6). Die Unterscheidung spielt hier aber keine Rolle.

| Körper | Widerstand |         |
|--------|------------|---------|
|        | Druck      | Reibung |
|        | 100%       | 0%      |
|        | 70%        | 30%     |
|        | 30%        | 70%     |
|        | 0%         | 100%    |

**Abbildung 8.6:** Der Zusammenhang von Druck- und Reibungswiderstand in Abhängigkeit der Form des umströmten Objekts.

$\vec{v}_r$  bezeichne die Relativgeschwindigkeit von Flocke und Wind:

$$\vec{v}_r = \vec{v}_w - \vec{v}_f \quad (8.6)$$

Der Luftwiderstand kann für Fluide mit hohen Geschwindigkeiten (die Windgeschwindigkeit fällt in diese Kategorie) mit der folgenden Luftwiderstandsformel angegeben werden, die auf *Rayleigh* zurückgeht:

$$\vec{F}_w = \frac{1}{2} \rho_{\text{Luft}} \vec{v}_r^2 C_d A \quad (8.7)$$

In der Formel bezeichnet  $\rho_{\text{Luft}}$  die Dichte der Luft, die bei  $0^\circ\text{C}$  etwa  $1.293 \frac{\text{kg}}{\text{m}^3}$  beträgt.  $A$  bezeichnet die *Bezugsfläche* des Körpers, den man betrachtet. Nähert man die Schneeflocke als Kugel mit Radius  $r$  an, so kann  $A$  als  $\pi r^2$  gewählt werden, wenn man die Projektion der Kugel auf die Ebene als Referenz wählt, oder  $2\pi r^2$ , wenn man die „Vorderfläche“ der Kugel als Referenz wählt.  $C_d$  bezeichnet den dimensionslosen **Strömungswiderstandkoeffizienten**. Der Luftwiderstand wächst also quadratisch zur Geschwindigkeit des Objekts. Dadurch, dass die Relativgeschwindigkeit  $\vec{v}_r$  in die Formel eingesetzt wurde, entsteht auch bei Windstille eine zur Geschwindigkeit der Flocke entgegengesetzte Kraft.

Die Größe  $C_d$  ist noch unbekannt. Sie wird normalerweise experimentell ermittelt. Da zu diesem Thema jedoch keine Ergebnisse zu finden sind, muss ein anderer Weg gewählt werden, um den Koeffizienten abschätzen zu können.

Angenommen, eine Schneeflocke fällt mit ihrer Endgeschwindigkeit  $v_{y,\text{max}}$  gerade nach unten. Dann entspricht die Kraft, die durch den Luftwiderstand hervorgerufen wird, genau

der Schwerkraft:

$$|\vec{F}_g| = |\vec{F}_w| \quad (8.8)$$

$$m \cdot |\vec{g}| = \frac{1}{2} \rho v_{y,\max}^2 C_d A \quad (8.9)$$

Löst man die untere Gleichung nach  $C_d$  auf erhält man:

$$C_d = \frac{2 \cdot m \cdot |\vec{g}|}{\rho \cdot v_{y,\max}^2 \cdot A} \quad (8.10)$$

Dies eingesetzt in die Luftwiderstandsformel 8.7 ergibt:

$$\vec{F}_w = \frac{\vec{v}_w^2 \cdot m \cdot |\vec{g}|}{v_{y,\max}^2} \quad (8.11)$$

Die Formel hängt nur noch von  $v_{y,\max}^2$  ab. Diese Größe ist — im Gegensatz zu  $C_d$  — gut bestimmt worden, z. B. in [14] für Temperaturen zwischen  $-2^\circ\text{C}$  und  $0^\circ\text{C}$ . Dabei wurde festgestellt, dass die Größe der Schneeflocke kaum Einfluss auf die Fallgeschwindigkeit hat. Es wurden Geschwindigkeiten im Bereich  $0.5\text{m/s}$  bis  $2.0\text{m/s}$  gemessen. In [36] wurde festgestellt, dass zwischen feuchtem und trockenen Schnee ein Faktor 2 bezüglich der Geschwindigkeit besteht. Aagaard wählt in [1] daher die folgenden Endgeschwindigkeiten:

$$0.5 \frac{m}{s} \leq v_{y,\max,\text{trocken}} \leq 1.5 \frac{m}{s} \quad (8.12)$$

$$1 \frac{m}{s} \leq v_{y,\max,\text{feucht}} \leq 2 \frac{m}{s} \quad (8.13)$$

Um  $\vec{F}_w$  zu bestimmen, muss nun lediglich die Masse  $m$  der Schneeflocke untersucht werden. In [36] wurde festgestellt, dass sich die *Dichte* einer Schneeflocke etwa umgekehrt proportional zu ihrem Durchmesser verhält:

$$\rho_{\text{Flocke}} = \frac{1}{D} \quad (8.14)$$

Der Durchmesser einer Flocke wurde bereits behandelt. Zwischen Dichte und Masse besteht die Beziehung  $\rho = m \cdot V$ , wobei  $V$  das Volumen des betrachteten Körpers ist. Es wird daher  $\rho = m$  gesetzt.

Damit lässt sich  $\vec{F}_w$  nun vollständig bestimmen. Wie später erläutert wird, werden für den Querschnitt  $A$  der Flocken, sowie für  $v_{y,\max}$  zufällige Werte bestimmt, um ein natürlicheres Muster zu erzeugen und um die Annäherung der Flocken durch Kugeln auszugleichen. Die folgende Funktion setzt die Kraft um, die durch den Luftwiderstand erzeugt wird:

---

```
float3 drag_force(
    float3 terminal_velocity,
    float diameter,
    float3 wind_velocity,
```

```

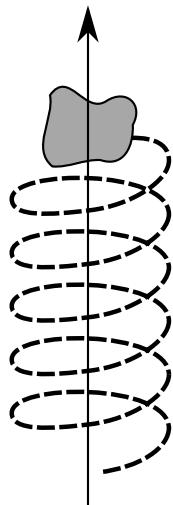
float3 flake_velocity,
float gravity)
{
    return
        // Quadrierte Relativgeschwindigkeit
        pow(wind_velocity - flake_velocity,2.0f) *
        // Annäherung fuer die Masse
        (1.0f / diameter) *
        gravity /
        pow(terminal_velocity,2.0f);
}

```

---

#### 8.4.3 Dynamischer Auftrieb

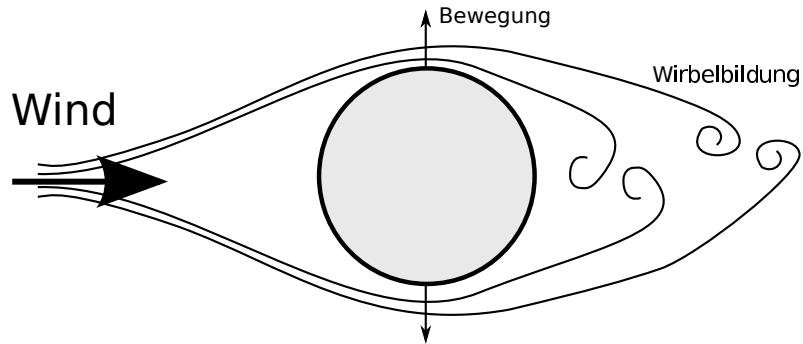
Beobachtungen an Schneeflocken bei ruhigem Wetter haben ergeben, dass die Flocken selten in einer geraden Bahn nach unten fallen, sondern eher Spiralbewegungen machen, siehe Abbildung 8.7.



**Abbildung 8.7:** Die ungefähre Bahn einer Schneeflocke ohne den Einfluss von Wind.

Diese Spiralbahn kann nicht durch den Luftwiderstand hervorgerufen werden, denn dieser wirkt bei Windstille nur entgegengesetzt der Schwerkraft, also gerade nach oben. Stattdessen wird eine Kraft dadurch erzeugt, dass sich hinter der Schneeflocke Verwirbelungen bilden und sie dadurch orthogonal zur Flugbahn ablenken, siehe Abbildung 8.8. Dasselbe Phänomen tritt auch makroskopisch in der Atmosphäre auf, siehe Abbildung 8.9.

Die Wirbel entstehen durch niedrigen Druck, der hinter dem Hindernis entsteht. Je nach Form des Objekts hat er unterschiedliche Auswirkungen. Der dynamische Auftrieb auf dünnen, länglichen Objekten wie den Tragflächen von Flugzeugen sorgt dafür, dass das Flugzeug abhebt. Er ermöglicht Vögeln und Flugzeugen das Fliegen. Bei stumpfen Körpern wie Golfbällen oder Schneeflocken hingegen bilden sich die besagten Wirbel und dadurch eine Seitwärtsbewegung. Diese Wirbel sind im Dreidimensionalen nur sehr schwer zu be-



**Abbildung 8.8:** Hinter der Schneeflocke bilden sich während des Flugs Wirbel, die eine Kraft seitwärts ausüben.



**Abbildung 8.9:** Die Insel Rishiri-to an der nordwestlichen Küste von Hokkaido, Japan erzeugt Wirbel in der Atmosphäre, die sich über weite Strecken ausbreiten.

schreiben. Statt einem genauen Modell wird also ein einfacheres Modell angesetzt und der Effekt nur „nachgebaut“.

Damit sich die Schneeflocke auf einer kreisförmigen Bahn in der  $xz$ -Ebene bewegt, wird folgende Ablenkungsgeschwindigkeit zugrunde gelegt:

$$\vec{v}_a = \omega \cdot r \begin{pmatrix} -\sin \omega t \\ 0 \\ \cos \omega t \end{pmatrix} \quad (8.15)$$

Hier bezeichnet  $\omega$  die Winkelgeschwindigkeit,  $r$  den Kreisradius der Bahn und  $t$  den Zeitparameter (in diesem Fall kein Zeitdelta, sondern ein absoluter Wert). Die Flocken sollen allerdings nur dann eine Kreisbahn einnehmen, wenn ihre Geschwindigkeit relativ wenig durch den Wind beeinflusst wird. Deshalb wird die Gleichung (8.15) mit einem  $C_a$  gewichtet:

$$C_a = \frac{|\vec{v}_w - \vec{v}_f|}{|\vec{v}_f|} \quad (8.16)$$

Es wurden keine Experimente bezüglich des Bahnradius  $r$  und der Winkelgeschwindigkeit  $\omega$  gefunden. Experimentelle Werte aus [1] ergaben aber für diese Größen folgende Einschränkungen:

$$-\frac{\pi}{4} \leq \omega \leq -\frac{\pi}{3} \quad (8.17)$$

$$0 \leq r \leq 2 \quad (8.18)$$

Daher wurden auch diese Größen randomisiert. Die folgende Funktion gibt die Geschwindigkeitskorrektur durch den dynamischen Auftrieb an:

---

```

float3 lift_force(
    float3 wind_velocity,
    float3 flake_velocity,
    float angular_velocity,
    float radius,
    float t)
{
    return
        length(wind_velocity - flake_velocity) / length(flake_velocity) *
        angular_velocity * radius *
        (float3)(
            -sin(angular_velocity * t),
            0.0f,
            cos(angular_velocity * t));
}

```

---

#### 8.4.4 Statischer Auftrieb

Auftrieb bezeichnet eine Kraft, die der Schwerkraft entgegengesetzt ist. Sie wirkt also in Richtung  $(0, 1, 0)$ . Sie entsteht durch die Verdrängung des Fluids durch einen Körper und sie ist indirekt abhängig von der Luftdichte. Sei daher  $\rho_{\text{Luft}}(x, y, z)$  das Vektorfeld der Dichte in jedem Punkt. Bei  $(x, y, z)$  ergibt sich der folgende (betragsmäßige) Auftrieb:

$$|F_A| = \rho_{\text{Luft}}(x, y, z) \cdot V \cdot g \quad (8.19)$$

Hier ist  $V$  das Volumen der Schneeflocke und  $g$  die Gravitationskonstante. Dieser Zusammenhang ist als *Archimedisches Prinzip* bekannt.

Aus folgenden Gründen wurde sich allerdings *dagegen* entschieden, den statischen Auftrieb zu modellieren:

1. Die Dichte der Luft ist abhängig vom Druck und von der Temperatur. Die Temperatur wird allerdings nicht explizit modelliert, man kann sie also überall als konstant annehmen. Der Druck variiert verhältnismäßig wenig. Daher kann insgesamt die Dichte der Luft als konstant angenommen werden. Die Auftriebskraft variiert so nur mit dem (ebenfalls sehr kleinen) randomisierten Volumen der Schneeflocken.
2. Die Dichte der Luft ist nicht nur konstant, sie ist auch sehr klein im Vergleich zu anderen Fluiden wie z. B. Wasser, was den Effekt von Auftrieb vernachlässigbar macht.

#### 8.4.5 Code

**Tabelle 8.1:** Die zu einer Flocke gehörigen Daten und ihr Ursprung

---

| E            | Beschreibung                                   | Berechnung  |
|--------------|--|---|
| $\vec{p}$    | Position                                       | Zufälliger Anfangswert innerhalb der Simulationsgrenzen   |
| $\vec{v}$    | Geschwindigkeit                                | Anfangs gleich $v_{y,\max}$   |
| $D$          | Durchmesser der Flocke                         | Gemäß Gleichung (8.1)   |
| $r$          | Rotationsradius für dynamischen Auftrieb       | Zufällig, gleichverteilt aus dem Intervall $[0, 2]$   |
| $\omega$     | Winkelgeschwindigkeit für dynamischen Auftrieb | Zufällig, gleichverteilt aus dem Intervall $[-\pi/4, -\pi/3]$ oder $[\pi/4, \pi/3]$                   |
| $v_{y,\max}$ | Endgeschwindigkeit                             | Zufällig, gleichverteilt aus $[0.5, 1.5]$ für feuchten Schnee bzw. $[1.0, 2.0]$ für trockenen Schnee. |

---

Nun wird die konkrete Umsetzung der Partikelsimulation besprochen. Tabelle 8.1 zeigt die Eigenschaften, die für jede Flocke zu verwalten sind. Viele dieser Eigenschaften wie die Größe oder der Rotationsradius können einmalig beim Erstellen des Buffers auf dem Host gesetzt werden und bleiben dann konstant, sofern die Außentemperatur  $T$  nicht verändert wird. Dynamisch sind einzige die Eigenschaften Position und Geschwindigkeit. Für jede Eigenschaft wird ein Buffer auf der Grafikkarte reserviert.

Die Bewegung der Schneeflocken ist in zwei Teile geteilt, den *Bewegungssteil* und den *Kollisionsteil*, die allerdings aus Performancegründen später in einem einzigen Kernel umgesetzt sind.

Die Methode für die Bewegung der Flocken bekommt als weitere Eingabe das Vektorfeld der Geschwindigkeitspfeile. Daraus wird die Geschwindigkeit des Fluids an der Position der Flocke interpoliert. Weiterhin wird ein fortlaufender Zeitparameter  $t$  zusätzlich zum Zeitdelta  $dt$  an den Kernel gegeben. Mit  $t$  wird die Rotation der Flocke berechnet.

Nach Ausführung der Bewegungsmethode kann es allerdings vorkommen, dass eine Flocke entweder in einem Hindernis oder außerhalb des Simulationsbereiches landet. In diesem

Fall muss eine neue Position für die Schneeflocke gefunden werden. Hier ergeben sich zwei Schwierigkeiten:

1. Die neue Position sollte zufällig gewählt werden, damit ein minimaler Grad an sich wiederholenden Mustern beim Schneefall auftritt. Die Grafikkarte bietet jedoch keinen Mechanismus hierfür, weder für Pseudozufallszahlen noch für echten Zufall.
2. Die neue Position sollte idealerweise nicht so gewählt werden, dass sie wieder in einem Hindernis liegt.

Da kein Mechanismus in OpenCL vorhanden ist, um Zufallszahlen zu generieren, müssen die im Kernel schon verfügbaren Daten als Zufallsquelle herangezogen werden. OpenCL schreibt vor, dass Zahlen vom Typ `float` im IEEE-754-Format vorliegen müssen. Dies erlaubt es, Eigenschaften dieser Kodierung auszunutzen, um für einen gegebenen Fließkommawert einen pseudozufälligen Wert zurückzugeben, der aus den einzelnen Bits der Eingabezahl entsteht. Dies wurde in [10] in CUDA umgesetzt, um die Positionen von Partikeln zu randomisieren. Die folgende OpenCL-C-Funktion gibt einen Wert im Intervall [0, 1] zurück.

---

```
float
random_float_value(
    float f)
{
    return
        (as_float(
            as_int(f) & 0x007fffff | 0x40000000)
        - 3.0f
        + 1.0f) /
        2.0f;
}
```

---

Die Eingabe `f` wird zeitweise als Ganzzahl interpretiert, um die Bit-Operatoren `&` und `|` auszunutzen.

Diese Funktion kann man jetzt nutzen, um gegeben der alten Position der Schneeflocke eine neue Position zu finden. Der folgende Code enthält die Funktion, um eine neue Position zu berechnen sowie um eine gegebene Flocke zurückzusetzen.

---

```
float3
random_position(
    int4 bounding_rect,
    float3 f)
{
    float3 result;

    result.x =
        random_float_value(
```

```

        (f.x + f.z) / f.y);
result.z =
    random_float_value(
        f.x / f.z + f.y);
result.y =
    result.x +
    (1.0f - result.x) *
    random_float_value(
        f.x - f.z - f.y);

return
    result *
    convert_float3(
        bounding_rect);
}

void
reset_flake(
    global float3 *positions,
    global float3 *velocities,
    global float *terminal_velocities,
    int3 bounding_volume,
    size_t my_id)
{
    positions[my_id] = random_position(bounding_volume, positions[my_id]);
    velocities[my_id] = (float3)(0.0f, -terminal_velocities[my_id], 0.0f);
    return;
}

```

---

Statt nur der alten Position der Schneeflocke könnte man weitere Daten wie ihre Geschwindigkeit in die Berechnungen einfließen lassen.

Durch diese Funktion ist allerdings nicht sichergestellt, dass die resultierende Position außerhalb eines Hindernisses liegt. Dies ist ebenfalls umsetzbar, zum Beispiel mit folgendem Verfahren

1. Erstelle Hindernisfeld (dies wird in Abschnitt 10 beschrieben).
2. Erstelle zusätzlichen Buffer `boundary_indices` bestehend aus dreidimensionalen Vektoren. Dieser Buffer enthält die Positionen aller Zellen, die von einem Hindernis belegt sind.
3. Beim Reset einer Flocke, wähle ein zufälliges Element aus `boundary_indices` und generiere einen zufälligen Vektor innerhalb der so gewählten Zelle.

Da der Effekt einer fehlplatzierten Flocke allerdings vernachlässigbar ist, wurde dieses Verfahren nicht umgesetzt.

Der folgende Kernel benutzt alle eben definierten Funktionen, um die Flockenpositionen und die Geschwindigkeiten zu updaten.

---

```
bool out_of_bounds(int3 position, int3 size)
{
    return any(position < (int3)(0)) || any(position > size);
}

float3 interpolated_wind_velocity(
    global float3 *wind_velocity,
    float3 position,
    int3 discrete_position,
    int3 bounding_volume)
{
    return interpolate_neighbors(
        neighbors_for_pos(
            wind_velocity,
            discrete_position,
            bounding_volume),
        fract(
            position));
}

float3 gravity_force(
    float gravity,
    float diameter)
{
    return
        float3(
            0.0f,
            -gravity * diameter),
            0.0f);
}

kernel void move_flakes(
    global float3 *positions,
    global float3 *velocities,
    global float3 *wind_velocity,
    global float *boundary,
    global float *diameters,
    global float *rotation_radiuses,
    global float *angular_velocities,
    global float *terminal_velocities,
    float gravity,
    float dt,
```

```

float t,
int3 bounding_volume)
{
    size_t my_id = get_global_id(0);
    float3 const current_position = positions[my_id];
    int3 const discrete_position = convert_int3(current_position);
    uint index = i4p(discrete_position, size);
    if(
        out_of_bounds(discrete_position, bounding_volume) ||
        boundary[index] > 0.5f)
    {
        reset_flake(
            positions,
            velocities,
            terminal_velocities,
            bounding_volume,
            my_id);
        return;
    }

    float3 wind_velocity = interpolated_wind_velocity(
        wind_velocity,
        current_position,
        discrete_position,
        bounding_volume);

    float3 current_forces =
        gravity_force(gravity, diameters[my_id]) +
        drag_force(
            terminal_velocities[my_id],
            diameters[my_id],
            wind_velocity,
            velocities[my_id],
            gravity);

    velocities[my_id] +=
        dt * current_forces +
        lift_force(
            wind_velocity,
            velocities[my_id],
            angular_velocities[my_id],
            rotation_radiuses[my_id],
            t);

    positions[my_id] += dt * velocities[my_id];
}

```

}

---

Die Funktion `out_of_bounds` erhält eine diskrete Gitterposition und die Gittergröße und gibt zurück, ob die Position innerhalb des Gitters liegt.

`interpolated_wind_velocity` arbeitet nach demselben Prinzip wie die Advektionsroutine aus Abschnitt 7.3.2.

Der Kernel testet zuerst, ob sich die Flocke außerhalb des Gitters befindet oder in ein Hindernis geflogen ist. Ist dies der Fall, wird sie mittels `reset_flake` neu gesetzt. Ansonsten werden erst die Gravitation und der Luftwiderstand berechnet. Das Ergebnis wird verwendet, um die Geschwindigkeit der Flocke zu verändern. Hier geht auch `lift_force` ein. Schließlich wird die Position neu gesetzt. Das Weg-Zeit-Gesetz wird also in indirekter Form umgesetzt.

## 9 Liegengebliebener Schnee

### 9.1 Einleitung

In Abschnitt 8 wurden Techniken vorgestellt, um die Bewegungen einer großen Anzahl Schneeflocken zu simulieren und diese Flocken auf dem Bildschirm darzustellen. Es wird ein realistischer Eindruck von fallendem Schnee geschaffen, der auf physikalischen Ansätzen beruht, statt — wie sonst üblich — lediglich den visuellen Eindruck nachzubauen.

Treffen die Flocken auf dem Boden oder einem Hindernis auf, werden sie an eine neue, zufällige Position zurückgesetzt. In der Realität würde sich allerdings (bei kaltem Untergrund) nach und nach eine *Schneedecke* bilden. Dieser Aspekt wurde bisher nicht behandelt, das Schneemodell ist also noch unvollständig.

An welchen Stellen wie viel Schnee liegenbleibt, lässt sich nicht allein aus dem Windfeld und der Hindernisgeometrie bestimmen. Es ist auch abhängig von der Anzahl der Schneeflocken, sowie deren Flugeigenschaften, welche wiederum aufgrund von Temperaturparametern und Zufallsvariablen variieren. Da die Schneeflocken als Partikelsystem modelliert werden, also jede Flocke einzeln betrachtet wird, kann man einen statistischen Ansatz verwenden, um die Schneemenge abzuschätzen: Man erweitert den Code, der beim Auftreffen einer Schneeflocke auf einem Hindernis ausgeführt wird. Statt an dieser Stelle nur Position und Geschwindigkeit der Flocke zurückzusetzen, erhöht man die Schneemenge am Auftreffpunkt.

In diesem Abschnitt werden zwei Ansätze besprochen, um mit Hilfe dieses Mechanismus eine Schneedecke zu erzeugen. Der erste Ansatz basiert darauf, die Oberfläche der Objekte an den Stellen mit einer Schneetextur einzufärben, an denen die Schneeflocken auftreffen. Er erlaubt keine „Schneogeometrie“ — also Schneehügel, die sich auftürmen. Dafür trägt jede Schneeflocke sichtbar zur Schneedecke bei.

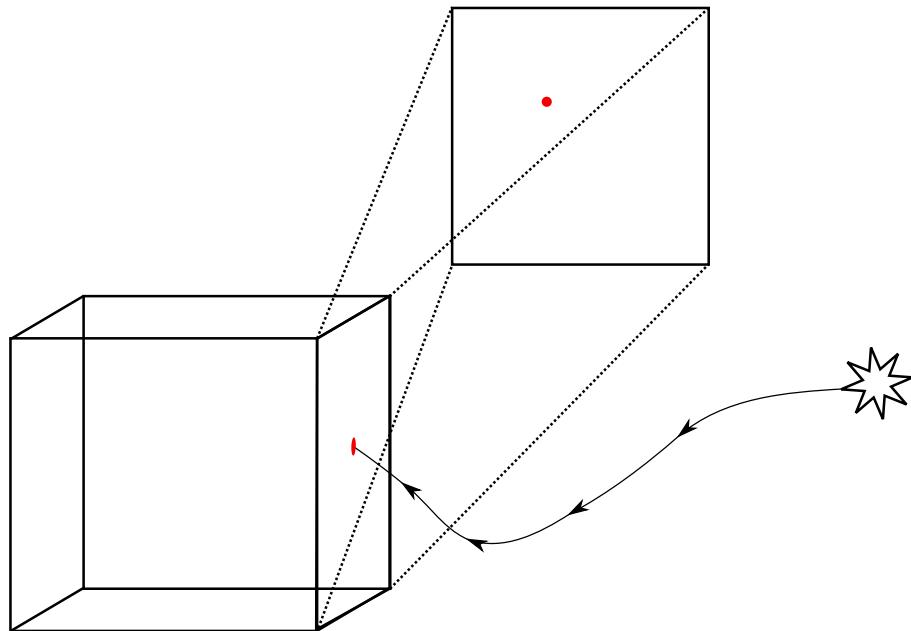
Wegen gravierender Nachteile dieses Verfahrens wurde danach ein anderer Ansatz verfolgt, der ein dreidimensionales Gitter verwaltet. In den Zellen des Gitters ist die dortige Schneemenge gespeichert. Beim Auftreffen einer Schneeflocke auf einem Hindernis wird die Schneemenge an der Flockenposition erhöht. Mit Hilfe des Marching Cubes-Algorithmus wird daraus ein *Mesh* für den Schnee erzeugt. Dieses Verfahren wurde zuerst auf der GPU und später auf der CPU implementiert. Es sollen die Unterschiede in den Implementierungen herausgestellt werden.

### 9.2 Schneedecke mit Texturen

Die moderne Computergrafik nutzt für die realistische Einfärbung von Oberflächen **Texture Mapping**, also die Projektion eines Bildes auf die einzufärbende Oberfläche. Heutige Grafikkarten unterstützen auch **Multitexturing**, um mehrere Texturebenen gleichzeitig auf eine Oberfläche abzubilden. Der erste Ansatz eine Schneedecke zu visualisieren bestand darin, sie als zusätzliche Textur auf den Oberflächen zu modellieren, welche anfangs komplett durchsichtig ist und mit der Zeit eine weiße Färbung annimmt, wenn Flocken auf der Oberfläche auftreffen.

Es ist aus OpenCL heraus möglich, in (zweidimensionale) Texturen zu schreiben. Daher kann man im Kernel, der die Bewegung der Schneeflocken durchführt, folgendes Verfahren für die Generierung einer Schneedecke umsetzen (vgl. Abbildung 9.1):

1. Reserviere für jede Oberfläche jedes Objekts neben der Farbtextur eine zweite Textur für die Schneedecke. Diese Texturen sollten anfangs komplett durchsichtig sein.
2. Beim Auftreffen einer Schneeflocke auf ein Hindernis, bestimme das getroffene Objekt und die Oberfläche auf dem Objekt, die getroffen wurde. Bestimme daraus die Textur, die zu der Oberfläche gehört.
3. Durch einen Koordinatensystemwechsel, bestimme den Pixel auf der Textur, der von der Flocke getroffen wurde.
4. Erhöhe den Wert des Pixels um einen vorher gewählten Wert, um es ein wenig weißer und untransparenter zu machen.
5. Vermische beim Rendern die Oberflächentextur und die Schneetextur, sodass die Schneedecke sichtbar wird.



**Abbildung 9.1:** Das Funktionsprinzip des Algorithmus für die Texturschneedecke verdeutlicht.

Der Ansatz liefert relativ gute Ergebnisse, es ergeben sich aber signifikante Hürden:

1. Es können keine Schneehügel umgesetzt werden, die sich auftürmen, da die Geometrie nicht verändert wird, nur die Oberflächeneigenschaften.
2. Es muss wesentlich mehr Speicherplatz aufgewendet werden, um für jede Oberfläche eine zweite Textur zu reservieren. Zudem ist unklar, wie groß die Schneetextur sein sollte. Bei kleineren Flächen bietet sich auch eine kleinere Textur an, aber eine Normierung zu finden, ist ein schwierigeres Problem. Um Platz zu sparen, könnte man die Textur zu einer Oberfläche auch erst erzeugen, wenn wirklich Schnee auf ihr aufgetroffen ist.

3. Die zu einem Auftreffpunkt gehörige Textur in angemessener Zeit zu bestimmen, benötigt eine Struktur, die den Raum effizient aufteilt.
4. Stabilitätseigenschaften wie die Bildung von Schneelawinen sind nicht einfach zu integrieren.



**Abbildung 9.2:** Die Schneedecke mit Texturen. Nur der Boden verwendet das Verfahren.

Aufgrund dieser Nachteile wurde lediglich eine Form des Verfahrens umgesetzt, die für den Boden der Simulation eine Schneedecke erzeugt. Abbildung 9.2 zeigt das Ergebnis.

### 9.3 Schneedecke als Mesh

#### 9.3.1 Einleitung

Statt die Schneedecke als reine Oberflächeneigenschaft zu modellieren, wird jetzt ein Ansatz vorgestellt, der die *Schneedichte* an jedem Punkt im Raum betrachtet. Diese Dichte wird mit Hilfe des Marching Cubes-Algorithmus (MC) in ein Mesh verwandelt, welches dann texturiert und beleuchtet wird.

Auf diese Weise können auch Schneehügel entstehen und Stabilitätsbedingungen lassen sich einfacher integrieren. Viele der Nachteile des Texturansatzes werden mit diesem Vorgehen behoben. Aber das Verfahren hat auch Nachteile, wie beispielsweise der hohe Rechen- und Speicherbedarf.

Es wird zuerst besprochen, wie das Dichtefeld aufgebaut wird und welche Eingabe der Marching Cubes-Algorithmus erhält. Danach wird der Algorithmus selbst grob erklärt. Schließlich werden die zwei umgesetzten Implementierungen für GPU und CPU verglichen und die endgültige Integration in die Anwendung erläutert.

### 9.3.2 Aufbau der Schneedichte

Trifft eine Schneeflocke auf ein Hindernis, also einen Voxel mit Boundarywert 1, so wird sie auf eine neue, zufällige Position zurückgesetzt. Dies ist der Zeitpunkt, an dem die abgelagerte Schneemenge in einer entsprechenden Datenstruktur erhöht werden muss. Man legt ein weiteres Skalarfeld `snow_density` an, welches die Schneemenge in den Gitterzellen der Simulation enthält (z. B. gemessen in Gramm). Beim Auftreffen einer Schneeflocke an einem Voxel wird der Eintrag an dieser Stelle des Skalarfeldes um das Gewicht der Schneeflocke erhöht. Mit dieser einfachen Herangehensweise ergeben sich zwei Probleme:

1. Der Test auf Kollision geschieht ein Voxelschicht „zu spät“. Testet man, ob die Schneeflocke *in* einem Hindernis ist und erhöht *dort* die Schneemenge, steckt auch der Schnee im Hindernis, nicht an seiner Oberfläche.
2. Es können sich ohne weitere Stabilitätsbedingungen keine Schneehaufen oder dickere Schneedecken bilden, denn der Kollisionscode für die Schneeflocke beachtet nur das Feld `boundary`, nicht die Schneedichte. Voxel, die keine Hindernisvoxel sind, können keinen Schnee ansammeln.

Zur Lösung dieser Probleme wird ein weiteres Skalarfeld eingeführt, das **Aktivitätsfeld** `activity`. Dieses Feld wird im Kollisionscode der Schneeflocken verwendet. Es ist ein binäres Feld — ein Voxel kann nur die Werte 0 oder 1 annehmen.

Das Aktivitätsfeld wird in jedem Simulationsdurchlauf abgeleitet aus den Feldern `boundary` und `snow_density`. Es ist an einer Gitterzelle ( $x, y, z$ ) genau dann 1, wenn eine der folgenden Bedingungen zutrifft:

1. Ein Nachbarfeld von ( $x, y, z$ ) im Feld `boundary` hat den Wert 1. Auf diese Weise wird das erste oben besprochene Problem gelöst, denn die Flocken kollidieren eine „Schicht“ vor dem Hindernis, und dort wird auch die Schneedichte erhöht. Es spielt in diesem Fall keine besondere Rolle, welche Nachbarschaft gewählt wird.
2. Die Schneedichte in `snow_density` ist größer als ein bestimmter Schwellenwert. So wird das zweite Problem gelöst, denn bereits gefallener Schnee wirkt sich jetzt auf den Kollisionscode aus. Man könnte alternativ an dieser Stelle das Feld `boundary` auf 1 setzen, damit sich Schneehaufen sogar auf die Windsimulation auswirken.

Als Eingabe für den Marching Cubes-Algorithmus dient allerdings weiterhin das Feld `snow_density`, das die Schneedichte enthält.

### 9.3.3 Der Marching Cubes-Algorithmus

Der Marching Cubes-Algorithmus wurde 1987 von *William E. Lorensen* und *Harvey E. Cline* als Ergebnis einer Forschungsarbeit für die Forschungsabteilung des Unternehmens General Electric in der Zeitschrift Computer Graphics vorgestellt ([21]). Der Algorithmus stand 20 Jahre lang unter einem Patent, seit 2005 kann er frei verwendet werden.

Ursprünglich war MC für bildgebende Verfahren in der Medizin gedacht. Medizinische Geräte wie ein MRT-Scanner geben dreidimensionale Punktwolken zurück, die dann entweder „scheibenweise“ als 2D-Bilder betrachtet werden können oder als dreidimensionales

Modell. Bei dem Modell handelt es sich natürlich nur um eine Annäherung, die stark von der Qualität der Punktfolge abhängt.

Als Eingabe erhält der Algorithmus in dieser Arbeit statt einer Punktfolge ein regelmäßiges Gitter mit skalaren Werten — z. B. die Schneidichte —, sowie einen **Isowert**, der angibt, ab wann eine Gitterzelle als „fest“ angesehen wird. Benachbarte, feste Zellen werden vom Algorithmus in ein zusammenhängendes Mesh verwandelt.

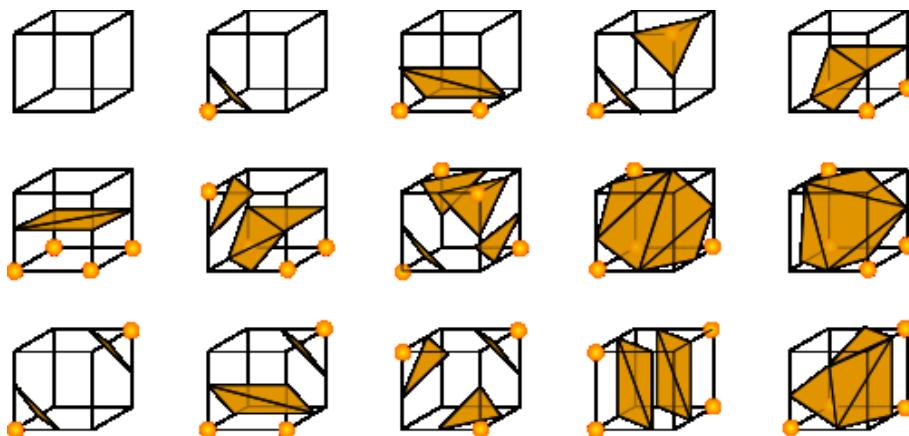
Die Ausgabe des Algorithmus ist eine Liste von Dreiecken, deren Eckpunkte zunächst nur durch ihre Position gegeben sind. Der Algorithmus wird später um Normalen und Texturkoordinaten erweitert.

Das Eingabefeld wird zellenweise bearbeitet. Für jede Zelle  $n_1$  werden die unmittelbaren Nachbarzellen  $n_2, \dots, n_8$  bestimmt, genau wie in Abschnitt 7.3.2 beschrieben. Der Dichtewert an jeder Ecke des durch die  $n_i$  beschriebenen Würfels wird mit dem vorgegebenen Isowert verglichen. Es entsteht eine 1, wenn der Wert über dem Isowert liegt, ansonsten eine 0. Als Ausgabe für eine Zelle erhält man auf diese Weise eine 8-Bit-Zahl und  $2^8 = 256$  mögliche Kombinationen von Werten für einen Würfel.

Die 8-Bit-Zahl wird als Index für eine Lookuptabelle verwendet, in der die *Dreiecke* stehen, die zu dieser Kombination von Isowerten gehören (siehe Abbildung 9.3). Diese Liste ist leer für den Fall, dass keiner der Werte über dem Isowert liegt. Ansonsten werden die Dreiecke zur Ausgabe hinzugefügt und die nächste Zelle wird betrachtet.

Wahlweise werden die Eckpunkte der Dreiecke in diesem letzten Schritt leicht verschoben um der Tatsache gerecht zu werden, dass einige Werte ggf. weit über dem Isowert liegen und andere nur leicht darüber.

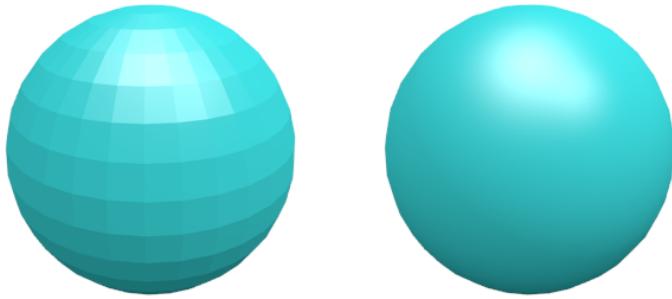
Außerdem kann die Lookuptabelle aufgrund von Symmetrieeigenschaften von 256 auf 15 Fälle reduziert werden.



**Abbildung 9.3:** Die durch Symmetrieeigenschaften auf 15 Fälle reduzierte Lookuptabelle für die Dreiecke in einem Würfel

#### 9.3.4 Normalen und Texturen

Der eben beschriebene Algorithmus gibt eine Liste von Dreiecken aus, an deren Eckpunkten lediglich Informationen über die Position enthalten sind. Diese Dreiecksmenge kann nicht ad hoc texturiert werden, denn es fehlen Texturkoordinaten. Auch die Beleuchtung fällt



**Abbildung 9.4:** Vergleich zwischen Flat Shading (links), welches durch Flächennormalen entsteht, und Smooth Shading (rechts)

ohne Normaleninformationen schwer. In diesem Abschnitt werden Lösungen für beide Probleme besprochen.

Um Normalen für das Mesh zu erhalten, gibt es mehrere Möglichkeiten. Die einfachste besteht darin, für jedes Dreieck  $\vec{p}_1, \vec{p}_2, \vec{p}_3$  eine Flächennormale mit Hilfe des Kreuzprodukts zu bestimmen:

$$\vec{n} = (\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1) \quad (9.1)$$

Dies ist ohne Weiteres umsetzbar, es ermöglicht allerdings nur **Flat Shading** als Beleuchtungsmodell, also keine sanften Übergänge zwischen benachbarten Flächen (siehe Abbildung 9.4).

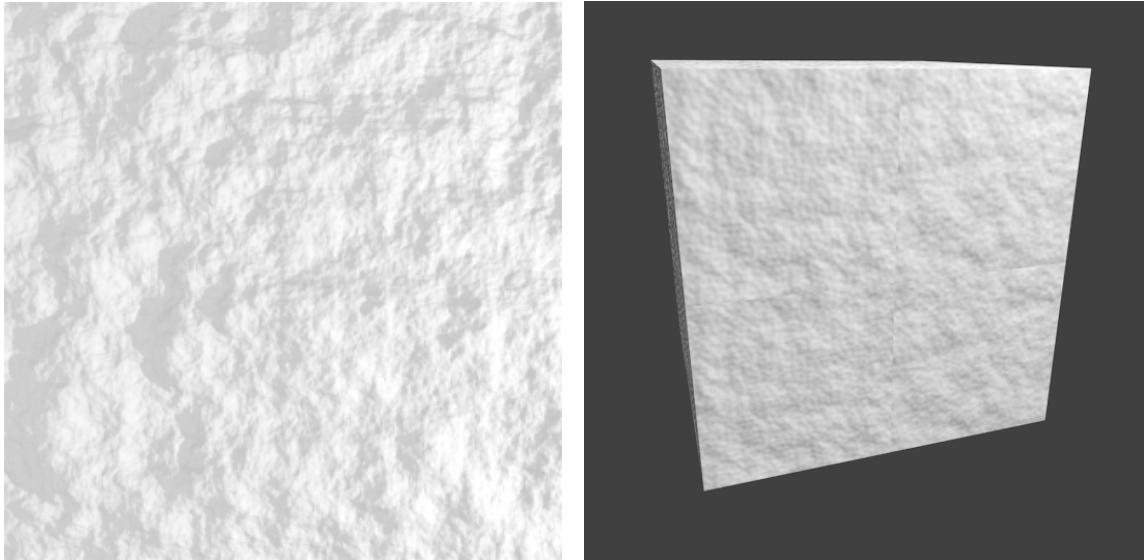
Um zwischen den Flächen interpolierte Normalen zu erhalten könnte man für jeden Punkt jeder Fläche alle benachbarten Flächen suchen und mitteln. Da die Dreiecksliste aus der MC-Ausgabe aber keine hilfreichen topologischen Informationen enthält, benötigt dieses Vorgehen quadratische Laufzeit.

Stattdessen benutzt man nicht die *Ausgabe* des Algorithmus, sondern das *Eingabefeld*, um die Normalen zu berechnen. Es wird der *Gradient* des Eingabefeldes als Quelle für die Normalen verwendet. Dies fällt besonders einfach aus, da der Code zum Berechnen des Gradienten schon existiert — er wurde in Abschnitt 7.3.4 besprochen.

Um das Mesh zu texturieren wurden zwei Ansätze verfolgt, die beide erläutert werden. Der erste basiert auf Noise aus 3D-Texturen und benötigt keine weiteren Daten außer der Eckpunkten der Dreiecke. Der zweite wird als **triplanare Texturierung** bezeichnet und benötigt die Flächennormalen der Dreiecke. Die Technik wurde unter anderem in [29] vorgestellt und liefert visuell ansprechende Ergebnisse.

Hat man nur die Positionen der Dreieckspunkte zur Verfügung und will Farben bestimmen, sodass die Oberfläche der einer Schneedecke ähnelt, liegt es nahe, die Farbdaten direkt aus einer dreidimensionalen Textur zu laden. Diese 3D-Textur repräsentiert einen „Block“ aus Schnee, in den man mittels einer Koordinate hineingreifen kann und die Schneefarbe innerhalb des Volumens erhält. Es stellt sich die Frage, woher man diese Textur bekommt und wie groß sie sein muss.

Die Größe der Textur ist davon abhängig, wie groß der Simulationsbereich gewählt wird. Simuliert man einen 100 Meter großen Bereich und möchte Details erkennen, die 10cm groß



**(a)** Eine mit Perlin-Noise generierte Schneedecke **(b)** Artefakte, die bei einer nichtkachelbaren 3D-Texture auftreten

**Abbildung 9.5:** Generierung der Schneetextur aus Perlin-Noise

sind, benötigt man bereits eine  $1000 \times 1000 \times 1000$  große Textur, die bei 8 Bit Graustufen schon annähernd einen Gigabyte Platz verbraucht. Heutige Grafikkarten haben allerdings *insgesamt* nur einen Gigabyte Videospeicher. Es ist daher sinnvoll, eine kleinere Volumen-textur zu verwenden, die aber an allen Seiten nahtlos anschließt (**kachelbar** ist), sodass sie mehrfach auf jeder Achse wiederholt werden kann. Es bleibt zu klären, wie man eine solche Textur herstellt.

Bei zweidimensionalen Texturen reicht es oft, Fotos der entsprechenden Oberfläche zu machen und in eine Textur zu konvertieren. Bei Volumendaten hingegen greift man häufig auf prozedurale Generierung zurück, also der algorithmischen Erstellung von Farbwerten. Abbildung 9.5a zeigt beispielsweise eine 2D-Textur für eine Schneedecke, die algorithmisch mittels Perlin-Noise ([32]) generiert wurde.

Die Generierung von dreidimensionalem Noise ist jedoch an sich schon sehr aufwändig. Die Anforderung, dass die Textur kachelbar sein muss, erhöht den Konstruktionsaufwand immens. Der Ansatz wurde daher nur mit nichtkachelbaren Texturen umgesetzt, was zu deutlichen Bildfehlern führte (vgl. Abbildung 9.5b).

In der finalen Implementierung wurde daher eine triplanare Texturierung verwendet. Diese Methode ist im Vergleich sehr platzsparend, dafür aber etwas rechenaufwändiger als eine 3D-Textur. Sie erlaubt es, steilen Oberflächen eine andere Textur zu geben als annähernd waagerechten, was den Realismusgrad beträchtlich erhöht. Das Verfahren lässt sich komplett in GLSL umsetzen und benötigt keine weiteren Daten von außen.

Die Idee hinter dem Verfahren ist, dass man drei 2D-Texturen vorgibt, jeweils für die  $xy$ -Ebene, die  $xz$ -Ebene und die  $yz$ -Ebene. Die Normale an einem Dreieckspunkt entscheidet, welche Textur wie stark zur Färbung des Dreiecks beiträgt. Ein horizontales Dreieck mit Normale  $(0, 1, 0)$  verwendet beispielsweise nur die zur  $xz$ -Ebene gehörige Textur.

Für den Schnee wird später eine relativ glatte Textur auf der  $xz$ -Ebene verwendet und zwei etwas „rauere“ Texturen für die beiden anderen Ebenen. Dadurch, dass man die Normalen an den Eckpunkten interpoliert, kann man auch gekrümmte Flächen gut einfärben, sofern sich die Texturen nicht zu stark unterscheiden. Die folgende Funktion erhält die Oberflächennormale und berechnet daraus die Gewichte für die drei Texturen. Sie werden als dreidimensionaler Vektor zurückgegeben:

---

```
vec3
calculate_blend_weights(
    vec3 normal_vector)
{
    vec3 blend_weights =
        max(
            (abs(normal_vector) - 0.2) * 7.0,
            0.0);

    float blend_sum =
        blend_weights.x + blend_weights.y + blend_weights.z;

    // Achtung: Eventuell Division durch 0!
    if(blend_sum < 0.001)
        return vec3(1.0,0.0,0.0);

    blend_weights /=
        blend_sum;

    return
        blend_weights;
}
```

---

Der Normalenvektor wird zunächst verschoben und gewichtet. Die Werte 0.2 und 7 sind Erfahrungswerte, um das Überblenden zwischen den Texturausrichtungen etwas schöner zu gestalten. Sie wurden direkt aus [29] übernommen. Danach wird der Vektor durch die Summe seiner Komponenten geteilt, damit sich alle Komponenten zu 1 aufaddieren.

Für jede der drei Texturen berechnen sich die Texturkoordinaten, indem man aus der Position des Dreieckspunktes jeweils die beiden Achsen nimmt, die die Ebene aufspannen. Die folgende Funktion erhält die Position des Punktes in Weltkoordinaten, die errechneten Gewichte und die Texturen und berechnet die endgültige Farbe:

---

```
vec4
calculate_blended_color(
    vec3 blend_weights,
    vec3 world_position,
    uniform sampler2D steep_texture,
```



Abbildung 9.6: Die triplanare Texturierung einer Anhöhe mit zwei Texturen.

```
uniform sampler2D flat_texture)
{
    vec2 coord1 = world_position.yz,
          coord2 = world_position.zx,
          coord3 = world_position.xy;

    vec4 col1 = texture(stEEP_texture,coord1),
          col2 = texture(flat_texture,coord2),
          col3 = texture(stEEP_texture,coord3);

    return col1 * blend_weights.x +
           col2 * blend_weights.y +
           col3 * blend_weights.z;
}
```

---

### 9.3.5 Implementierungen

In diesem Abschnitt werden die MC-Implementierungen auf der GPU und der CPU besprochen. Dabei soll deutlich werden, was die jeweiligen Vor- und Nachteile sind und wieso die endgültige Anwendung schließlich die CPU-Implementierung einsetzt.

### 9.3.6 GPU-Implementierung

Der Marching Cubes-Algorithmus ist im Wesentlichen datenparallel. Jede Zelle des Gitters kann unabhängig vom Rest bearbeitet werden. Daher liegt es auch hier nahe, die GPU für die Konvertierung der Schneedecke zu verwenden. Dies erweist sich jedoch in der Praxis als schwierig, denn obwohl die Zellen an sich unabhängig zu bearbeiten sind, werden für jede Zelle unterschiedlich viele Dreiecke erzeugt. Bei den bisher behandelten datenparallelen

Problemen war die Anzahl der Elemente der Eingabe stets gleich der Anzahl der Elemente der Ausgabe. Hierbei konnte sich die Größe der Eingabe- und Ausgabeelemente jedoch unterscheiden (wie bei der Divergenz, wo ein Vektorfeld zu einem Skalarfeld wurde). Der MC-Algorithmus passt nicht in dieses Schema.

In einer solchen Situation verwendet man üblicherweise eine **Reduktion** (auch **Scan** genannt, siehe [2]). Reduktionen sind algorithmische Grundbausteine auf parallelen Systemen. Mit ihnen können scheinbar nur sequentiell (und damit schlecht) lösbarer Operationen wie die Summenbildung effizient parallel umgesetzt werden. Reduktionen sollen hier nicht besprochen werden, da sie den Rahmen der Arbeit übersteigen.

Als Basis für MC auf der GPU wurde ein Programm aus der Beispielsammlung von NVidia gewählt ([31]). Das Programm besitzt eine hohe Komplexität, implementiert den Algorithmus allerdings in sehr effizienter Weise unter Benutzung mehrerer Reduktionsschritte. In seiner Rohform setzt es jedoch nur Flächennormalen und die maximale Feldgröße ist auf  $64^3$  beschränkt. Beide Probleme konnte ohne großen Aufwand behoben werden, die maximale Feldgröße wurde auf  $128 \times 64 \times 128$  erhöht.

Neben der hohen Komplexität spricht gegen eine Umsetzung auf der GPU zudem, dass die Zwischenergebnisse des Algorithmus' relativ viel Speicherplatz verbrauchen. Der Grafikkartenspeicher ist jedoch stark begrenzt.

### 9.3.7 CPU-Implementierung

Der MC-Algorithmus ist auf der CPU weit weniger effizient als auf der GPU, da die Unabhängigkeit der einzelnen Würfel bei der Bearbeitung nur wenig ausgenutzt werden kann. Dies wiegt allerdings nicht so schwer, da die Schneedecke nicht in jedem Simulationsschritt neu berechnet werden muss. Für einen visuell angenehmen Eindruck reicht es, die Schneedecke jede Sekunde neu zu erstellen, solange die Simulation dadurch nicht sichtbar ins Stocken gerät.

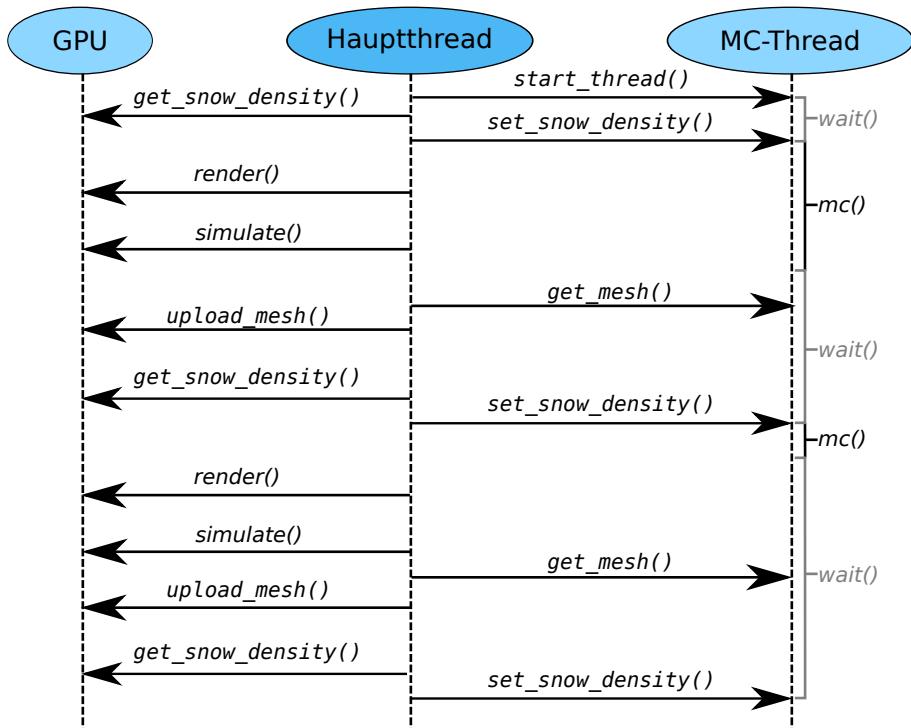
Die MC-Implementierung aus [20] konnte direkt übernommen werden, da sie bereits interpolierte Normalen ausgibt und beliebig große Felder zulässt. Die Implementierung verwendet einen leicht modifizierten Algorithmus, der stärkere topologische Garantien liefert und so im Allgemeinen glattere Meshes erzeugt. Intern wird kein Multithreading genutzt, sondern eine sequentielle Schleife über alle Gitterzellen.

Um die Simulation und die Visualisierung weiterlaufen zu lassen und trotzdem ohne sichtbare Aussetzer die Schneedecke zu updaten, werden *Threads* eingesetzt. Es ist zu beachten, dass Multithreading in den zwei heute gängigen Grafik-Frameworks OpenGL und DirectX erst in den neuesten Versionen korrekt unterstützt wird. Um mit den älteren Versionen dieser Frameworks kompatibel zu sein, wurde das Verfahren so modifiziert, dass nur ein Thread Zugriff auf die GPU hat.

Es gibt zwei Threads: Der MC-Thread erhält vom Hauptthread die aktuelle Schneedichte und nutzt diese, um daraus die Dreiecke des Schnee-Meshs zu erzeugen. Der Hauptthread stößt währenddessen kontinuierlich die Kernel für die Simulation an und tätigt Renderaufrufe. Wenn der MC-Thread eine Dreiecksmenge produziert hat, wird diese vom Hauptthread zur GPU hochgeladen. Danach wird die aktuelle Schneedichte von der GPU heruntergeladen und an den MC-Thread übergeben.

Dies ist ein **Producer-Consumer-Problem**, wobei nur ein Producer (der MC-Thread) und

ein Consumer (der Hauptthread) existiert. Abbildung 9.7 verdeutlicht den Ablauf.



**Abbildung 9.7:** Das Producer-Consumer-Problem in der Anwendung. Der MC-Thread wartet solange, bis ihm der Hauptthread Daten zur Verfügung stellt. Er verarbeitet die Daten und wartet dann. Der Hauptthread prüft nur an bestimmten Synchronisationspunkten, ob der MC-Thread Daten geliefert hat.

## 9.4 Vergleich

Die Arbeit von Manuel Schwarz ([39]) befasst sich ebenfalls mit der Generierung und Anzeige einer Schneedecke. In der Arbeit werden zwei Ansätze vorgeschlagen. Der erste basiert auf sogenannten *Displacement-Maps*. Damit bezeichnet man Texturen, welche die Geometrie des Meshes verändern, zu dem sie gehören. Mit dieser Methode können sich auch ohne ein separates Mesh Schneehügel bilden.

Der zweite Ansatz, der in der Arbeit vorgestellt wird, entspricht dem MC-Ansatz, der auch hier erläutert wird. Es sind allerdings einige Unterschiede herauszustellen:

1. Schwarz lässt ebenfalls beliebige Hindernisse zu, indem er sie aus Modeldateien einliest und in eine Gitterstruktur verwandelt (siehe Abschnitt 10). Dadurch, dass relativ kleine Gitter (kleiner als  $32 \times 32 \times 32$ ) verwendet werden, ist er aber in der Lage, den Punkt-in-Polygon-Algorithmus zu verwenden, der für die vorliegende Arbeit zu zeitaufwändig ist. Schwarz ist also nicht auf ein Programm wie binvox angewiesen, welches in Abschnitt 10.2 vorgestellt wird.
  2. Da der Fokus bei Schwarz' Arbeit nicht auf der realistischen Visualisierung des Schnees selber liegt, verwendet er keine Texturen. Die Schneedecke wird lediglich mit Ver texnormalen versehen und beleuchtet.

3. Beide Arbeiten ermöglichen die Bildung von Schneeanhäufungen. Mit dem vorliegenden, einfachen Modell kann es jedoch passieren, dass diese Anhäufungen unnatürliche Formen annehmen. Schnee, der mit einer gewissen horizontalen Geschwindigkeit auf eine Wand prallt, erzeugt einen „Klecks“ auf der Wand, der dort verharrt. In Wirklichkeit würde dieser Klecks wegen seines Eigengewichts zu Boden fallen. Um dies zu erzwingen, enthält die Arbeit von Schwarz Ansätze und Implementierungen für sogenannte **Stabilitätsbedingungen**. Es ergibt sich also eine weit realistischere Schneedecke bei starkem Schneefall.

Stabilitätsbedingungen wurden wegen der hohen Rechenkomplexität bei größeren Gittern in dieser Arbeit nicht weiter verfolgt.

## 10 Hindernisse in der Simulation

Bisher wurde das Hindernisfeld `boundary` als gegeben angenommen. Nun sollen Möglichkeiten vorgestellt werden, das Feld mit Hinderniswerten zu füllen. Es wird zuerst ein Ansatz beschrieben, bei dem Hindernisse aus primitiven Formen zusammengesetzt werden. Danach werden Verfahren vorgestellt, um direkt aus einer Meshdatei ein Hindernisfeld zu erzeugen.

### 10.1 Einfacher Ansatz

Oft ist die Geometrie von Hindernissen relativ einfach gestaltet oder sie wird speziell für die Simulation vereinfacht. Das angezeigte Mesh hat dann gegebenenfalls wesentlich mehr Details als das diskretisierte Hindernisfeld. Bewegen sich die Partikel schnell genug bzw. löscht man die Partikel erst eine gewisse Zeit nach der Kollision mit den Hindernissen, ist der Detailunterschied nicht zu merken.

Es bietet sich daher an, die Hindernisse aus einfachen geometrischen Formen wie Kugeln oder Quadern zusammenzusetzen. Diese Formen lassen sich sehr einfach diskretisieren, was im Folgenden besprochen werden soll.

Gegeben eine Menge von geometrischen Formen und ein anfangs mit 0 gefülltes Hindernisfeld, kann das endgültige Hindernisfeld inkrementell konstruiert werden, indem einzelne Hindernisse hinzugefügt werden (sprich, Werte auf 1 gesetzt werden). Für jedes Primitiv benötigt man einen Kernel, der die geometrischen Eigenschaften (z. B. Ausdehnung) sowie `boundary` erhält.

Für einen Quader  $Q$ , gegeben durch die drei Intervalle  $[x_1, x_2], [y_1, y_2], [z_1, z_2]$ , ist dieser Kernel sehr einfach gestaltet. Für jeden Voxel muss er testen, ob dessen Position innerhalb des Quaders ist.

Für eine Kugel, gegeben durch einen Radius  $r$  und eine Position  $(x, y, z)$  muss für jeden Voxel mit Position  $(v_x, v_y, v_z)$  die Kugelungleichung ausgewertet werden:

$$(x - v_x)^2 + (y - v_y)^2 + (z - v_z)^2 \leq r^2 \quad (10.1)$$

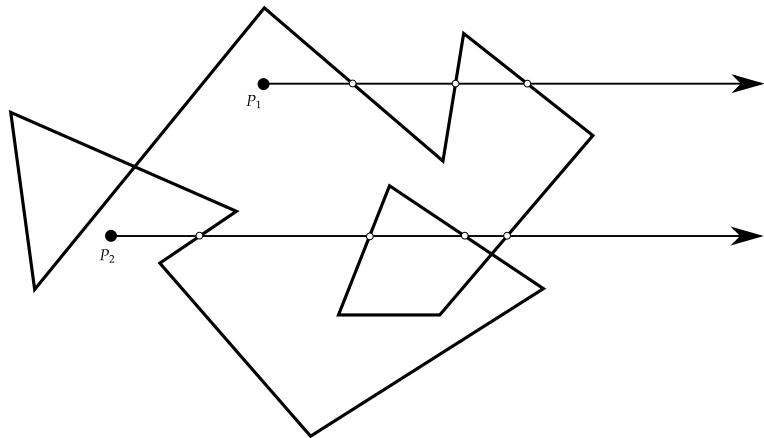
Ist sie wahr, wird der Voxel auf 1 gesetzt. Ähnlich einfach lässt sich ein Ellipsoid behandeln, welcher durch drei Radien  $r_x, r_y, r_z$  und eine Position  $(x, y, z)$  gegeben ist. Seine Ungleichung lautet:

$$\frac{(x - v_x)^2}{r_x} + \frac{(y - v_y)^2}{r_y} + \frac{(z - v_z)^2}{r_z} \leq 1 \quad (10.2)$$

Diese Art, Hindernisse einzubringen, ist allerdings sehr zeitaufwändig, da stets zwei Modelle zu erstellen sind — eins zur Anzeige, bestehend aus einer Menge von Dreiecken, und eins für die Simulation, bestehend aus geometrischen Formen. Die Konvertierung dieser Formen zu Dreiecksnetzen ist zwar ohne Probleme möglich, es ist jedoch ein sehr unnatürlicher Arbeitsablauf und es existieren keine Tools, die diesen vereinfachen könnten.

## 10.2 Komplexe Hindernisse

Es wäre günstiger, aus dem Mesh, das später angezeigt wird, das Hindernisfeld zu gewinnen — oder notfalls aus einem leicht vereinfachten Mesh, welches rein visuell interessante Objekte nicht enthält. Diesen Vorgang nennt man **Voxelization**. Im Weiteren wird ein simpler Algorithmus vorgeschlagen, um diese Aufgabe zu lösen. Für größere Felder ist dieser Algorithmus allerdings ungeeignet, daher werden Alternativen besprochen. Gegeben sei



**Abbildung 10.1:** Der einfache Test, ob ein Punkt in einem Körper enthalten ist, hier in zwei Dimensionen.  $P_1$  hat 3 Schnittpunkte und ist innerhalb,  $P_2$  hat 4 und ist somit außerhalb.

eine Menge von Dreiecken, definiert durch ihre Eckpunkte (weitere Eigenschaften wie Normalen und Texturkoordinaten sind für die Umwandlung nicht interessant). Die Menge von Dreiecken soll einen geschlossenen Körper bilden, Löcher oder Überschneidungen sind nicht erlaubt. Unter diesen Voraussetzungen lässt sich für jeden Gitterpunkt in `boundary` entscheiden, ob dieser innerhalb des Körpers ist oder außerhalb: Man betrachtet eine Halbgerade vom Gitterpunkt aus und zählt die Schnittpunkte der Geraden mit allen Dreiecken des Körpers. Bei einer geraden Anzahl von Schnittpunkten ist der Punkt außerhalb, sonst innerhalb (siehe Abbildung 10.1).

Der Test ist arithmetisch sehr einfach gestaltet und performant. Allerdings beträgt seine Laufzeit  $O(n^3 \cdot k)$ , wobei  $n$  die Größe des Gitters angibt und  $k$  die Anzahl an Dreiecken. Bei einem  $64^3$  großen Feld und einem hinreichend komplexen Modell mit 3.000 Dreiecken kommt man bereits auf 786.432.000 Tests.

Die Implementierung eines effizienteren Verfahrens übersteigt den Rahmen dieser Arbeit allerdings deutlich, daher wurde auf das Tool *binvox* zurückgegriffen ([24]). Es verwendet den Algorithmus aus [30], der auf „Ray Stabbing“ basiert, einer Abwandlung des eben beschriebenen Verfahrens. Das Tool arbeitet parallel auf der GPU und liefert in Sekunden Ergebnisse. Als Eingabe erhält es die Modelldatei im obj-Dateiformat, welche vom Programm auch direkt zur Anzeige verwendet wird. Die Ausgabe erfolgt in einem eigens geschriebenen Dateiformat, dem *binvox*-Format ([23]). Eine *.binvox*-Datei hat den folgenden, einfachen Aufbau:

```
#binvox 1
dim 128 128 128
```

```

translate -0.120158 -0.481158 -0.863158
scale 7.24632
data
...

```

Die erste Zeile gibt die Version des Dateiformats an. Zum jetzigen Zeitpunkt gibt es nur Version 1. Die nächste Zeile gibt die Dimensionen des Gitters an. Die Werte für **translate** und **scale** geben an, wie das Gitter verschoben und skaliert werden muss, um mit dem vorgegebene Model übereinzustimmen.

Danach folgen die eigentlichen Daten als rohe Bytes. Um Platz bei großen Gittern zu sparen, sind sie *laufängenkodiert*. Das Datensegment enthält eine Folge von Byte-Paaren. Das erste Byte in einem Paar gibt an, ob eine Folge von Nullen oder Einsen folgt. Das zweite Byte gibt an, wie viele Nullen oder Einsen folgen.

Die lineare Folge von Gitterpunkten wird mit der Festlegung zum dreidimensionalen Gitter, dass sich die  $y$ -Koordinate am schnellsten verändert, dann die  $x$ -Koordinate, dann die  $z$ -Koordinate.

Abschließend lässt sich sagen, dass sich mit dem Programm bivox komplexe Hindernisse in beliebiger Größe und Detailstufe in die Simulation einbringen lassen.



**(a)** Das Modell einer Lampe

**(b)** Die von bivox diskretisierte Version der Lampe

**Abbildung 10.2:** Das Tool bivox bei der Arbeit.

## 11 Reflexion

### 11.1 Ausblick

Es war im Rahmen dieser Arbeit nicht möglich, weitere Verfahren der numerischen Strömungssimulation neben dem Verfahren von Stam umzusetzen und zu vergleichen. Allerdings erwies sich das gewählte Verfahren als hinreichend performant und auch auf schwächere Systeme skalierbar.

Sowohl Stams Verfahren als auch der Marching-Cubes-Algorithmus, wie sie in dieser Arbeit behandelt wurden, basieren auf einer konstanten, vor dem Start des Programms festgelegten Gittergröße. Um größere Simulationsbereiche effizient zu unterstützen, müssen entweder mehrere Gitter zusammengeschaltet werden oder das Gitter darf nicht mehr regulär sein. Beide Ansätze übersteigen den Umfang der Arbeit jedoch erheblich, sie wurden daher nicht weiter verfolgt.

Die dynamische Schneedecke wurde lediglich in Teilen aus der Arbeit von Schwarz übernommen. Da sich bei der Implementierung der Stabilitätsbedingungen große Performanceprobleme ergaben, wurden sie weggelassen. Bei geringen Windgeschwindigkeiten und mäßigem Schneefall ergibt sich auf diese Weise zwar eine realitätsgestreue Schneedecke. Der liegengebliebene Schnee hat jedoch kein Eigengewicht und „haftet“ so an Hindernissen, welches besonders bei starkem Schneefall deutlich wird.

Das Hochladen der Schneedecke auf die Grafikkarte geschieht synchron im Hauptthread, welches ab einer bestimmten Datenmenge zu spürbaren Aussetzern der Simulation führt. Um dem entgegenzuwirken bieten neuere Versionen von OpenGL und DirectX Mechanismen für das Aufteilen von Ressourcen auf mehrere Threads. Dies wurde aus Kompatibilitätsgründen nicht weiter verfolgt.

Sowohl für die Strömungssimulation als auch für die Stabilitätsbedingungen sind weitere Arbeiten denkbar, welche die noch offenen Probleme behandeln.

### 11.2 Fazit

Die im Zuge dieser Arbeit entstandene Anwendung erfüllt die anfangs gestellten Ziele in vollem Umfang. Mit Hilfe von Stams Verfahren wird in Echtzeit ein Windfeld auf der GPU berechnet. Dazu ist lediglich eine OpenCL-fähige Grafikkarte notwendig; die Genauigkeit der Simulation kann zur Laufzeit an die Leistung der Grafikkarte angepasst werden. Die Stärke und Richtung des Windes ist dynamisch und kann beispielsweise aus einer Wetterdatenbank gespeist werden.

Das Windfeld wird von einem Partikelsystem dazu genutzt, den Einfluss des Windes auf die einzelnen Schneeflocken zu berechnen. In die Flugbahn der Partikel gehen weitere Kräfte wie beispielsweise der Auftrieb ein, welche für natürlichere Bewegungen sorgen. Es lässt sich gegebenenfalls eine große Anzahl an Partikeln simulieren, wodurch verschieden starker Schneefall simuliert werden kann. Visuell sind die Schneeflocken trotz der vereinfachten Darstellung gerade durch ihre realistischen Flugeigenschaften zufriedenstellend umgesetzt. Das Auftreffen der Schneeflocken auf Hindernissen sorgt für die dynamische Bildung einer Schneedecke, welche durch den Einsatz triplanarer Texturierung selbst bei geringer Gitterauflösung optisch ansprechend wirkt.

Zusammenfassend lässt sich sagen, dass jeder Aspekt von fallendem Schnee zufriedenstellend modelliert und visualisiert wurde und durch den Einsatz von OpenCL und OpenGL auch in Echtzeit simuliert werden kann.

## Literatur

- [1] M. Aagaard und D. Lerche. „Realistic modelling of falling and accumulating snow“. Diss. Aalborg University, 2004. URL: <http://www.cvmt.dk/Snow/>.
- [2] Guy Bleloch. „Scans as Primitive Parallel Operations“. In: *IEEE Transactions on Computers* 38 (1987), S. 1526–1538.
- [3] Jeff Bolz u. a. „Sparse matrix solvers on the GPU: conjugate gradients and multigrid“. In: *ACM SIGGRAPH 2003 Papers*. New York, New York, USA, 2003, S. 917–924.
- [4] Alex Laier Bordignon und Geovan Tavares. „Real Time Fluid Dynamics on Programmable GPU with Arbitrary Boundaries and Vorticity Confinement“. Rio de Janeiro, 2006.
- [5] Robert Bridson und Matthias Müller-Fischer. „Fluid simulation: SIGGRAPH 2007 course notes“. In: *ACM SIGGRAPH 2007 courses*. San Diego, California: ACM, 2007, S. 1–81.
- [6] a.P. Chandrakasan u. a. „Optimizing power using transformations“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.1 (1995), S. 12–31. ISSN: 02780070.
- [7] Shiyi Chen und G.D. Doolen. „Lattice Boltzmann method for fluid flows“. In: *Annual review of fluid mechanics* 30.1 (1998), S. 329–364.
- [8] a. J. Chorin, J. E. Marsden und A. Leonard. „A Mathematical Introduction to Fluid Mechanics“. In: *Journal of Applied Mechanics* 47.2 (1980), S. 460. ISSN: 00218936.
- [9] Keenan Crane, Ignacio Llamas und Sarah Tariq. „Real-time simulation and rendering of 3d fluids“. In: *GPU Gems*. Hrsg. von Hubert Nguyen. Bd. 3. Addison-Wesley Professional, 2007. Kap. 30, S. 633–675. ISBN: 0321515269.
- [10] Robin Eidissen. „Utilizing GPUs for Real-Time Visualization of Snow“. Diss. Norwegian Institute of Science and Technology, 2009.
- [11] Nick Foster, PDI Dreamworks und Ronald Fedkiw. „Practical Animation of Liquids“. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, S. 1–8.
- [12] Nick Foster und Dimitris Metaxas. „Modeling the motion of a hot, turbulent gas“. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. New York, New York, USA: ACM Press/Addison-Wesley Publishing Co., 1997, S. 181–188. ISBN: 0897918967.
- [13] Nolan Goodnight. „CUDA/OpenGL Fluid Simulation“. 2007.
- [14] Monika Hanesch. „Fall Velocity and Shape of Snowflakes“. Diss. Swiss Federal Institute of Technologies, Zürich, 1999.
- [15] Mark Harris. „Parallel Prefix Sum (Scan) with CUDA“. In: *GPU Gems 3*. January. 2008. Kap. 39.
- [16] Mark Harris u. a. „Simulation of cloud dynamics on graphics hardware“. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. San Diego, California, 2003, S. 92–101.

- [17] Norman Junker. *Winter Weather Forecasting*. 2000. URL: <http://www.hpc.ncep.noaa.gov/research/snow2a/index.htm>.
- [18] J. Krüger und R. Westermann. „Linear algebra operators for GPU implementation of numerical algorithms“. In: *ACM SIGGRAPH 2005 Courses* (2005).
- [19] M. S. Langer u. a. „A spectral-particle hybrid method for rendering falling snow“. In: *Eurographics Symposium on Rendering*. 2004.
- [20] Thomas Lewiner u. a. „Efficient implementation of marching cubes cases with topological guarantees“. In: *Journal of Graphics Tools* 8.2 (2003), S. 1–15.
- [21] William E. Lorensen und Harvey E. Cline. „Marching cubes: A high resolution 3D surface construction algorithm“. In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), S. 163–169. ISSN: 0097-8930.
- [22] Timothy Mattson, Beverly Sanders und Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004. ISBN: 0321228111.
- [23] Patrick Min. *bivox file format*. 2012. URL: <http://www.cs.princeton.edu/~min/bivox/bivox.html>.
- [24] Patrick Min. *bivox voxelization tool*. 2012. URL: <http://www.google.com/search?q=bivox>.
- [25] Matthias Müller, David Charypar und Markus Gross. „Particle-based fluid simulation for interactive applications“. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. San Diego, California, 2003, S. 154–159.
- [26] Matthias Müller und Nuttapong Chentanez. „Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid“. In: *ACM SIGGRAPH 2011 papers*. 2011, 82:1–82:10.
- [27] A. Munshi, B. Gaster und T.G. Mattson. *OpenCL Programming Guide*. OpenGL Series. Addison-Wesley, 2011. ISBN: 9780321749642. URL: [http://books.google.de/books?id=M-Sve\\\_\\\_KItQwC](http://books.google.de/books?id=M-Sve\_\_KItQwC).
- [28] Brigid Irene Naughton. *Diameter of a Snowflake*. 1996. URL: <http://hypertextbook.com/facts/BrigidNaughton.shtml>.
- [29] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.
- [30] F.S. Nooruddin. „Simplification and repair of polygonal models using volumetric techniques“. In: *IEEE Transactions on Visualization and Computer Graphics* 9.2 (2003), S. 191–205.
- [31] NVIDIA *OpenCL SDK Code Samples*. 2012. URL: [http://developer.download.nvidia.com/compute/cuda/4\\_2/rel/sdk/website/OpenCL/html/samples.html](http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/website/OpenCL/html/samples.html).
- [32] Ken Perlin. „Improving noise“. In: *ACM Trans. Graph.* 21.3 (Juli 2002), S. 681–682. ISSN: 0730-0301.
- [33] Franz Peschel. „Simulation und Visualisierung von Fluiden in einer Echtzeitanwendung mit Hilfe der GPU“. Studienarbeit. Universität Koblenz, 2009.
- [34] Matt Pharr und Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005. ISBN: 0321335597.

- [35] *Point Sprites (DirectX 9)*. 2012. URL: [http://msdn.microsoft.com/en-us/library/windows/desktop/bb147281\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb147281(v=vs.85).aspx).
- [36] Roy Rasmussen. *Theoretical Considerations in the Estimation of Snowfall Rate Using Visibility*. November. Montreal: Transportation Development Centre, 1998.
- [37] W. T. Reeves. „Particle Systems – a Technique for Modeling a Class of Fuzzy Objects“. In: *ACM Trans. Graph.* 2.2 (Apr. 1983), S. 91–108. ISSN: 0730-0301.
- [38] Ingar Saltvik. „Parallel Methods for Real-Time Visualization of Snow“. Master thesis. Norwegian University of Science und Technology, 2006.
- [39] Manuel Schwarz. „Prozedurale Modellierung von Schneedecken“. Bachelorarbeit. Universität Osnabrück, 2012.
- [40] Andrew Selle u. a. „An unconditionally stable MacCormack method“. In: *Journal of Scientific Computing* 35.2 (2008), S. 350–371.
- [41] Karl Sims. „Particle animation and rendering using data parallel computation“. In: *SIGGRAPH Comput. Graph.* 24.4 (Sep. 1990), S. 405–413. ISSN: 0097-8930.
- [42] Karl Sims. „Particle Dreams“. In: *SIGGRAPH Video Review* (1988).
- [43] Jos Stam. „Real-time fluid dynamics for games“. In: *Proceedings of the Game Developer Conference*. Bd. 18. Citeseer, 2003.
- [44] Jos Stam. „Stable fluids“. In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. New York, New York, USA: ACM Press/Addison-Wesley Publishing Co., 1999, S. 121–128. ISBN: 0201485605.
- [45] John Steinhoff und David Underhill. „Modification of the Euler equations for "vorticity confinement": Application to the computation of interacting vortex rings“. In: *Physics of Fluids* 6.8 (1994).
- [46] Niniane Wang und Bretton Wade. „Rendering falling rain and snow“. In: *ACM SIGGRAPH 2004 Sketches* (2004).
- [47] Wikipedia. *Luftfeuchtigkeit — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 3. Juli 2012]. 2012. URL: <http://de.wikipedia.org/w/index.php?title=Luftfeuchtigkeit&oldid=105102238>.
- [48] Wikipedia. *Sättigung (Physik) — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 19. September 2012]. 2012. URL: [http://de.wikipedia.org/w/index.php?title=S%C3%A4ttigung\\_\(Physik\)&oldid=106312931](http://de.wikipedia.org/w/index.php?title=S%C3%A4ttigung_(Physik)&oldid=106312931).
- [49] Enhua Wu, Youquan Liu und Xuehui Liu. „An improved study of real-time fluid simulation on GPU“. In: *Computer Animation and Virtual Worlds* 15.34 (Juli 2004), S. 139–146. ISSN: 1546-4261.
- [50] Satoshi Yanagi. *Notes of the Snow Crystal Observation*. 2011. URL: [http://www1.odn.ne.jp/snow-crystals/English\\_index.html](http://www1.odn.ne.jp/snow-crystals/English_index.html).

## **Erklärung**

Ich versichere, dass ich die eingereichte Diplomarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, November 2012