Flüssigkeitssimulation für Computergrafik

Christoph Groth

Januar 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung des Programmes *Falmaidan*[1], das die Bewegung von Flüssigkeit in einer Szene simuliert. Aus den Simulationsergebnissen kann mit Hilfe eines Renderers[2] eine realistisch aussehende Animation erzeugt werden. Bei der Implementierung wurde besonderer Wert auf Eleganz und Effizienz gelegt, die beide unter anderem durch eine neuartige Programmiertechnik erreicht werden.

Dem Benutzer steht eine Simulationsmethode zur Verfügung, welche die Navier-Stokes-Gleichungen löst und dadurch Phänomene wie Zähigkeit, Wirbel oder Spritzer nachbilden kann. Der Einbau weiterer Simulationsmethoden ist geplant.

Das Programm ist frei und steht unter `http://falmaidan.sourceforge.net/` zum Download bereit.

[1] „Falmaidan“ bedeutet in der Elbensprache Sindarin aus J.R.R. Tolkiens Werken „Wellenmacher“.
[2] Ein Programm, das aus einer Szenenbeschreibung Bilder der Szene erzeugt.

1 Einleitung

Lange Zeit beschränkte man sich bei der Computeranimation von Flüssigkeiten auf ruhige Wellen, die meist durch Überlagerung mehrerer Sinuskurven angenähert wurden. Für einige Anwendungen sind solche einfachen Techniken ausreichend, doch war es damit nicht möglich, beispielsweise einen Springbrunnen oder brechende Wellen glaubhaft in Szene zu setzen. Glücklicherweise haben Computer in den letzten Jahren ein Niveau an Rechenleistung erreicht, welches die sinnvolle Verwendung komplexer physikalischer Modelle in der Computergrafik ermöglicht. Filme wie „Antz“ oder „Der Sturm“ demonstrierten eindrucksvoll einige der nun möglichen Effekte.

Bislang war es aber nur Auserwählten vorbehalten, die wenigen Flüssigkeitsanimationsprogramme einzusetzen. Wenn überhaupt allgemein verfügbar, sind diese teuer und nur auf die Zusammenarbeit mit kommerziellen Renderern ausgelegt. Dabei existieren etliche sehr gute kostenlose und teilweise freie Renderer, nur fehlte bislang eine dazugehörige Flüssigkeitsanimationssoftware.

Das Ziel dieser Arbeit ist die Entwicklung eines universellen Flüssigkeitssimulationsprogramms, das diese Lücke schließen soll.

1.1 Entstehungsgeschichte

Die vorliegende Arbeit basiert auf meiner letztjährigen Jugend-Forscht-Arbeit [6]. Es ist mir damals aufgrund der extremen Komplexität des Quellcodes leider nicht gelungen, bis zum Wettbewerb alle Programmfehler der 3d-Version zu beseitigen. Nach einer längeren Pause nahm ich die Entwicklung erneut auf. Die neue Version wurde fast komplett neu geschrieben und enthält etliche Verbesserungen. Dank eines neuartigen Design-Ansatzes ist der Quellcode bei gleicher Ausführungsgeschwindigkeit deutlich einfacher und damit leichter wartbar und weniger fehleranfällig.

In der Zukunft wird das Programm weiterentwickelt werden. Es ist geplant, weitere Simulationsmethoden und Unterstützung für weitere Renderer einzubauen.

1.2 Überblick über die Abschnitte

Die Arbeit gliedert sich in einen Hauptteil und einen Anhang. Der Hauptteil selbst bleibt im Rahmen der von Jugend-Forscht vorgegebenen Seitenbeschränkung. Im Anhang wird versucht, die notwendigen physikalischen Grundlagen möglichst verständlich und kompakt zu erklären, er ist jedoch nicht zum Verständnis der Arbeit unbedingt erforderlich, sofern die Vorkenntnisse beim Leser bereits vorhanden sind.

2 Simulationsverfahren

Um Wasser oder andere Flüssigkeiten realistisch animieren zu können, müssen bei der Simulation die entsprechenden physikalischen Gesetze beachtet werden. Hierfür bieten sich die Navier-Stokes-Gleichungen an, die als eine sehr gute makroskopische Beschreibung von Flüssigkeiten gelten. Gleichzeitig dürfen die Berechnungen nicht zuviel Zeit in Anspruch nehmen, denn der Benutzer will nicht erst tagelang auf Ergebnisse warten.

Eine ausführliche Recherche ergab, daß zwei unterschiedliche Simulationsverfahren bekannt sind, die diese Bedingungen erfüllen. Andere Ansätze[3] liefern nicht ausreichend

[3] Ein Überblick findet sich in [1].

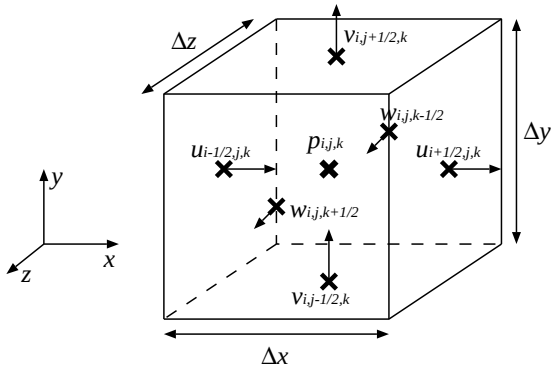

Abbildung 1: Eine Zelle beim Gitterverfahren.

realistische Ergebnisse oder sind nur auf Spezialfälle anwendbar.

2.1 Gitterverfahren

Das 1996 von Nick Foster und Dimitri Matexas vorgestellte Verfahren [5] löst[4] die Navier-Stokes-Gleichungen für ein grobes Zellengitter, so daß die Laufzeiten der Simulation noch in einem für die Computergrafik vertretbaren Rahmen bleiben. Um dennoch eine Ausgabe in hoher Auflösung zu erhalten, verwendet man massenlose Markerpartikel, die gleichmäßig in der Flüssigkeit verteilt sind und alle ihre Bewegungen mitmachen. Die Geschwindigkeit eines Teilchens wird durch lineare Interpolation der für die umliegenden Zellen berechneten Geschwindigkeiten ermittelt.

Im Folgenden wird eine verbesserte Version des Verfahrens beschrieben, so wie sie für Falmaidan implementiert wird.

2.1.1 Das Gitter

Beim Gitterverfahren sind die Größen Geschwindigkeit und Druck nicht kontinuierlich sondern, nur an bestimmten Stellen eines regelmäßigen Gitters definiert. Man unterteilt den quaderförmigen Bereich der „virtuellen Welt“, für den die Simulation durchgeführt werden soll, in $n_x \times n_y \times n_z$ quaderförmige Zellen mit den Ausmaßen $\Delta x \times \Delta y \times \Delta z$ (siehe Abbildung 1). Der Druck[5] $\tilde{p}_{i,j,k}$ ist jeweils in der Mitte jeder Zelle definiert und die Geschwindigkeiten in den Mittelpunkten der Zellenseiten. Für jeden Seitenmittelpunkt ist nur die zu dieser Seite senkrechte Geschwindigkeitskomponente gespeichert, also beispielsweise die y-Komponente für Seiten, die in der x, z-Ebene liegen. Anstatt mit v_x, v_y und v_z werden die Geschwindigkeitskomponenten im Gitter wegen der kompakteren Schreibweise mit u, v und w bezeichnet. $u_{i+1/2,j,k}$ bezeichnet somit die x-Geschwindigkeitskomponente in der Mitte der entsprechenden Seite der Zelle (i, j, k).

Da sie sich gegenseitig berühren, teilen zwei benachbarte Zellen immer eine gemeinsame Seite und für beide gilt an dieser Stelle die gleiche Geschwindigkeit. So ist beispielsweise $u_{i,j+1/2,k}$ gleich $u_{i,j+1-1/2,k}$ und $u_{i-1/2,j,k}$ gleich $u_{i-1+1/2,j,k}$.

[4] Oder nähert eher eine Lösung numerisch an, denn genaue Lösungen der Navier-Stokes-Gleichungen existieren nur für einfache Spezialfälle, wie den Fluß durch ein zylindrisches Rohr.

[5] $\tilde{p}$ ist in Wirklichkeit nicht der Druck, sondern der Quotient aus dem Druck und der konstanten Dichte.

2.1.2 Diskretisierung der Gleichungen

Die Navier-Stokes-Gleichungen sind Differentialgleichungen und daher müssen auch die Ableitungen der im Gitter definierten Größen bestimmt werden.

Für die erste Ableitung einer Größe f erhält man:

$$\frac{\partial f_{i+1/2}}{\partial x} \approx \frac{f_{i+1} - f_i}{\Delta x}.$$

Die zweite Ableitung läßt sich durch zweifaches Anwenden der obigen Formel bestimmen:

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{\frac{\partial f_{i+1/2}}{\partial x} - \frac{\partial f_{i-1/2}}{\partial x}}{\Delta x} \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2}.$$

Die Geschwindigkeit $\vec{v}'$ an einem festen Ort zum Zeitpunkt $t + \Delta t$ läßt sich aus der Geschwindigkeit $\vec{v}$ zum Zeitpunkt t und der Beschleunigung $\partial\vec{v}/\partial t$ bestimmen:

$$\vec{v}' = \vec{v} + \Delta t \frac{\partial \vec{v}}{\partial t}. \tag{1}$$

Löst man die Navier-Stokes-Gleichung (10) nach dieser Beschleunigung auf und setzt sie in (1) ein, so erhält man:

$$\vec{v}' = \vec{v} + \Delta t \left(-\vec{v} \cdot \nabla \vec{v} + \vec{g} - \nabla \tilde{p} + \nu \nabla^2 \vec{v} \right) \tag{2}$$

Die Diskretisierung ergibt für die x-Komponente:

$$\begin{aligned} u'_{i+1/2,j,k} = \; & u_{i+1/2,j,k} + \Delta t \, \Big(\tfrac{1}{\Delta x} \left(u^2_{i,j,k} - u^2_{i+1,j,k} \right) + \\ & \tfrac{1}{\Delta y} \left((uv)_{i+1/2,j-1/2,k} - (uv)_{i+1/2,j+1/2,k} \right) + \\ & \tfrac{1}{\Delta z} \left((uw)_{i+1/2,j,k-1/2} - (uw)_{i+1/2,j,k+1/2} \right) + \\ & g_x + \tfrac{1}{\Delta x} \left(\tilde{p}_{i,j,k} - \tilde{p}_{i+1,j,k} \right) + \\ & \nu \tfrac{1}{\Delta x^2} \left(u_{i+3/2,j,k} - 2u_{i+1/2,j,k} + u_{i-1/2,j,k} \right) + \\ & \nu \tfrac{1}{\Delta y^2} \left(u_{i+1/2,j+1,k} - 2u_{i+1/2,j,k} + u_{i+1/2,j-1,k} \right) + \\ & \nu \tfrac{1}{\Delta z^2} \left(u_{i+1/2,j,k+1} - 2u_{i+1/2,j,k} + u_{i+1/2,j,k-1} \right) \Big). \end{aligned} \tag{3}$$

Die u-, v- und w-Werte, die nicht direkt dem Gitter entnommen werden können, wie z. B. $u_{i,j,k}$, müssen aus ihrer Umgebung durch Interpolation bestimmt werden: $u_{i,j,k} = (u_{i+1/2,j,k} + u_{i-1/2,j,k})/2$. Analog zu (3) existieren auch Formeln für die beiden anderen Komponenten, die man durch entsprechendes „Vertauschen“ der Achsen erhält. Diese mühevolle und fehleranfällige Arbeit kann vom Compiler übernommen werden, so daß der Programmierer nur eine Version einzugeben und gegebenfalls zu ändern braucht (siehe Abschnitt 4.5.3).

Auch die Bedingung für Dichtebeständigkeit (11) kann diskretisiert werden:

$$\begin{aligned} & \tfrac{1}{\Delta x} \left(u_{i+1/2,j,k} - u_{i-1/2,j,k} \right) + \\ & \tfrac{1}{\Delta y} \left(v_{i,j+1/2,k} - v_{i,j-1/2,k} \right) + \\ & \tfrac{1}{\Delta z} \left(w_{i,j,k+1/2} - w_{i,j,k-1/2} \right) \quad = 0. \end{aligned} \tag{4}$$

2.1.3 Markerpartikel

Um die Rechenzeit in Maßen zu halten, darf die Szene nicht in allzuviele Zellen unterteilt werden. Leider muß bei einer Verdoppelung der Zellenauflösung auch der Zeitschritt Δt halbiert werden, um die Simulation nicht instabil werden zu lassen. Das führt dazu, daß in der Praxis mit Auflösungen in der Größenordnung von $40 \times 40 \times 40$ gearbeitet wird.

Würde man die Zellen einfach als Würfel darstellen, so wäre das Ergebnis sehr „pixelig“ und entspräche sicherlich nicht der menschlichen Vorstellung von Flüssigkeit. Doch mit einem Trick läßt sich der Eindruck einer viel höheren Auflösung erzeugen: Es werden Partikel eingeführt, die wie winzige Staubkörnchen überall in der Flüssigkeit schweben und ihre Bewegung mitmachen. Nach jeder Iteration wird jedes dieser Partikel entsprechend der für seinen Ort interpolierten Geschwindigkeit bewegt. Aus der Menge der Partikel kann eine glatte, realistisch aussehende Oberfläche generiert werden, die für das Rendering geeignet ist. Mehr Informationen hierzu finden sich im Abschnitt 3.2. Die Partikel werden auch verwendet, um volle von leeren Zellen zu unterscheiden. Eine Zelle ist voll, wenn sie mindestens einen Partikel enthält, ansonsten gilt sie als leer.

Da die Anzahl der Markerpartikel leicht in die Hunderttausende gehen kann, ist eine effiziente Implementierung sehr wichtig. Dies wurde durch zwei Maßnahmen erreicht:

effiziente Datenstrukturen: Pro Zelle existiert eine verkettete Liste, welche die jeweils in der Zelle befindlichen Partikel enthält, so daß sehr schnell bestimmt werden kann, wie viele Partikel sich in einer Zelle befinden. Auch das Bewegen der Partikel geht effizient von statten: Da die Partikel einer Zelle in einem Schub nacheinander abgearbeitet werden, treten weit weniger *cache misses*[6] auf, als bei einem naiven Ansatz.

Löschen nicht benötigter Partikel: Partikel, die sich innerhalb voller Zellen befinden, welche mindestens von einer weiteren Schicht voller Zellen umgeben sind, werden gelöscht, da sie keinen Einfluß auf die Flüssigkeitsoberfläche haben. „Taucht“ eine solche Zelle im Laufe der Simulation aus der „Tiefe“ auf, so werden neue Partikel für sie erzeugt.

2.1.4 Randbedingungen

Eine Zelle kann entweder voll (Fluid), leer (Atmosphäre) oder fest (Wand) sein. Da die Navier-Stokes-Gleichungen das Verhalten eines Tröpfchens beschreiben, das von allen Seiten von Flüssigkeit umgeben ist, müssen feste Wände und die Atmosphäre in der Simulation als Flüssigkeit mit besonderen Eigenschaften betrachtet werden.

Die Geschwindigkeiten an den Berührungsstellen zwischen festen und vollen Zellen betragen immer Null und der Druck in den Oberflächenzellen[7] entspricht dem in der Atmosphäre.

Es sind noch weitere Bedingungen für die Geschwindigkeiten an den Berührungsstellen zwischen Flüssigkeit und Atmosphäre erforderlich. Leider ist [5] hier und auch an

[6] „Cache misses“ nennt man Speicherzugriffe, die nicht vom schnellen Pufferspeicher abgefangen werden und deshalb deutlich langsamer sind. Um cache misses zu vermeiden, sollte ein Programm möglichst linear auf den Speicher zugreifen. Cache-optimierter Code kann durchaus drei mal schneller ablaufen als vergleichbarer nicht optimierter Code.

[7] Dies sind volle Zellen, die an mindestens eine leere Zelle angrenzen.

manch anderer Stelle ziemlich ungenau, so daß die richtigen Bedingungen für verschiedene Kombinationen von vollen, leeren und Oberflächenzellen durch eigene Versuche ermittelt werden mußten.

2.1.5 Simulationsablauf

Die Simulation wird iterativ vorangetrieben. Zunächst werden unter Anwendung von (3) aus den alten Geschwindigkeitswerten neue berechnet. Da wegen der Diskretisierung nunmehr die Dichtebeständigkeit nicht mehr überall gewährleistet ist, folgt eine Druckiteration, in der die Geschwindigkeiten und der Druck angepaßt werden. Für Zellen, in die mehr hinein- als herausfließt, wird der Druck erhöht und die Seitengeschwindigkeiten werden „nach außen“ korrigiert. Bei Zellen, aus denen mehr heraus- als hineinfließt verfährt man umgekehrt. Die Druckiteration wird so lange fortgesetzt, bis die Bedingung für Dichtebeständigkeit überall erfüllt ist. Anschließend werden die Markerpartikel bewegt und anhand ihrer neuen Positionen bestimmt, welche Zellen weiterhin voll und welche nun leer sind. Mit diesem Schritt ist die Simulationsiteration beendet und die nächste kann beginnen.

2.1.6 Beurteilung

Das Gitterverfahren liefert zufriedenstellende Ergebnisse, da es mit Ausnahme der Oberflächenspannung alle die realistische Bewegung von Flüssigkeiten ausmachenden Phänomene nachbildet. Zu den Nachteilen der Methode nach Foster und Matexas zählt die verhältnismäßig lange Simulationsdauer – an einigen Sekunden Animation kann ein Rechner schon mal etliche Stunden arbeiten. Außerdem stellte es sich heraus, daß die Simulationsparameter wie die Zellenauflösung oder der Zeitschritt sehr sorgfältig gewählt werden müssen, wenn die Simulation nicht instabil werden soll. Solche Instabilitäten zeigen sich darin, daß die Flüssigkeit plötzlich regelrecht explodiert, was zur Folge hat, daß man die Simulation erneut mit „vorsichtigeren“ Parametern starten muß.

2.2 Smoothed Particle Hydrodynamics

Das SPH-Verfahren entstammt der Astronomie und wurde zunächst verwendet, um Galaxien zu simulieren. In der Computergrafik wurde es als erstes von Mathieu Desbrun und Marie-Paule Gascuel angewandt [3]. Anstatt die Navier-Stokes-Gleichungen in einem Gitter zu lösen, werden hier direkt die auf die Partikel wirkenden Kräfte berechnet. Im Unterschied zu traditionellen Partikelsystemen, sind die Partikel jedoch keine unendlich kleinen Massenpunkte, sondern werden im Raum anhand einer nach Außen abnehmenden Dichtefunktion, Kernel genannt, „verschmiert“. Durch diesen Trick ist es möglich die Partikel als Flüssigkeitströpfchen aufzufassen und die Navier-Stokes-Gleichungen anzuwenden, was bei Massenpunkten nicht möglich wäre.

Die Methode verspricht vor allem eine höhere Stabilität verglichen mit dem Gitterverfahren. Allerdings ist es in einem SPH-Partikelfeld nicht ohne weiteres möglich, die zweite Ableitung einer Größe zu bestimmen, die für die Berechnung der Reibungskräfte nötig wäre. Desbrun und Gascuel verwenden daher in ihrer Implementierung eine „künstliche Viskosität“, die jedoch nur unbefriedigende Ergebnisse liefert.

Es ist geplant, SPH als eine weitere Simulationsmethode in Falmaidan aufzunehmen und das Verfahren dabei um eine „echte“ Viskosität zu erweitern. Die dazu notwendigen

zweiten Ableitungen der Geschwindigkeit sollen nach der Differenzmethode (siehe Abschnitt 2.1.2) aus den ersten Ableitungen errechnet werden. Dieser Ansatz dürfte zwar deutlich langsamer als die Verwendung „künstlicher Viskosität“ sein, doch im Gegenzug hätte man ein Verfahren, das bei vergleichbarer Realität und Ausführungsgeschwindigkeit weniger empfindlich als die Gittermethode ist.

3 Ein- und Ausgabe von Szenen

Die beste Simulation wäre nutzlos für die Computergrafik, wenn man die berechneten Flüssigkeiten nicht visualisieren könnte. Ein möglicher Ansatz hierzu wäre, einen Renderer in das Simulationsprogramm einzubauen. Dieser könnte besonders gut auf seine spezielle Aufgabe hin optimiert werden.

Bei der Entwicklung von Falmaidan wurde ein anderer, viel flexibler Weg gewählt. Das Simulationsprogramm selbst ist vom Renderer unabhängig und vom Hauptprogramm getrennte Module sorgen für das Aufbereiten von Eingabedaten in eine für das Hauptprogramm verständliche Form und die Ausgabe der Simulationsergebnisse in einem Format, das von einem Renderer verarbeitet werden kann.

Zur Zeit existieren Ein- und Ausgabe-Module für den kostenlosen Raytracer POV-Ray[8], weitere Module für andere Renderer sind geplant.

3.1 Eingabe

Bevor die Simulation beginnen kann, muß dem Programm mitgeteilt werden, was es überhaupt simulieren soll. Es bietet sich an für diese Aufgabe das gleiche, gegebenfalls geringfügig erweiterte, Datenformat zu verwenden, wie es auch der Renderer verarbeitet. Im Falle von POV-Ray ist dies die C-ähnliche POV-Ray-Szenenbeschreibungssprache. Szenen in dieser Sprache können mit Hilfe eines Texteditors oder auch unter Verwendung eines speziellen grafischen Editors erzeugt werden.

Zwar ist der Quellcode von POV-Ray offengelegt, doch leider verbieten es die Lizenzbedingungen Teile davon für eigene Projekte zu verwenden. So war es notwendig, einen eigenen „POV-Ray-Szenen-Compiler“ zu entwickeln. Bei dieser Arbeit waren die Programme *flex* und *bison* eine große Hilfe. Das Eingabemodul besteht aus drei Teilen:

Lexer: liest die Szenendatei und erkennt darin vorkommende *Tokens*, wie Anweisungen, Satzzeichen oder Zahlen. Der Lexer-C-Code wird aus einer speziellen Eingabedatei durch den „fast lexical analyzer generator“ flex erzeugt.

Parser: baut aus dem vom Lexer gelieferten Token-Strom eine hierarchische Repräsentation der Szene auf. Ähnlich wie der Lexer-Code wird auch der Parser-Code zum Großteil von einem speziellen Generator erzeugt. Der Programmierer definiert in einer Eingabedatei die Semantikregeln und mit ihnen verbundene Codefragmente, die bison zu einem funktionsfähigem Parser aufbereitet.

Analysator: beantwortet die Anfragen des Hauptprogramms, aus welchem Material die Szene an einer bestimmten Stelle besteht, indem er den vom Parser aufgebauten Szenenbaum durchläuft.

[8] `http://www.povray.org/`

Die Schnittstelle zum Hauptprogramm stellt eine kleine abstrakte C++-Basisklasse dar, so daß das Simulationsmodul vollkommen unabhängig von einer speziellen Eingabemodulimplementierung arbeitet.

3.2 Geometrieerzeugung und Ausgabe

Während der Simulation wird die Flüssigkeit durch Markerpartikel repräsentiert. Um eine realistisch aussehende Animation zu erhalten, muß aus ihren Positionen wieder ein zusammenhängendes Volumen bestimmt werden. Es ist wichtig, daß die Oberfläche dieses Volumens möglichst glatt ist, damit der gewünschte „flüssige“ Eindruck entsteht.

Eine Möglichkeit dies zu erreichen besteht darin, die Metaballs- oder Blob-Technik anzuwenden. Dabei werden die Partikel als gleichartige punktförmige Ladungen in einem Feld betrachtet. Dem Volumen zugehörig sind die Punkte, deren „Potential“ über einem bestimmten Schwellenwert liegt. Die resultierende Oberfläche ist verhältnismäßig glatt und kann mit Hilfe des in der Computergrafik häufig angewandten Marching-Cubes-Algorithmus gefunden werden.

Viele Renderer, zu denen auch POV-Ray gehört, können Blobs selbst erzeugen, so daß das Simulationsprogramm lediglich eine Textdatei mit den entsprechenden Anweisungen und Partikelkoordinaten ausgeben muß.

4 Implementierung

In diesem Abschnitt wird zunächst auf die verwendete Programmiersprache eingegangen. Im weiteren Teil werden verschiedene Aspekte des Falmaidan-Quellcodes angesprochen.

4.1 Wahl der Programmiersprache

Als Programmiersprache wurde aus mehreren Gründen C++ gewählt:

- Die Sprache unterstützt verschiedene Programmierparadigmen, darunter modulare, objekt-orientierte und generische Programmierung[9].
- Es ist eine sehr flexible und effiziente Laufzeitbibliothek vorhanden, die auch Datenstrukturen wie verkettete Listen enthält und dadurch dem Programmierer einiges an Routinearbeit erspart.
- C-Code-Generatoren wie flex und bison können auch mit C++ verwendet werden.
- C++ ist plattformunabhängig.
- Der erzeugte Maschinencode ist, abhängig vom Compiler, gut bis sehr gut.

4.2 2d- und 3d-Version

Falmaidan liegt in einer interaktiven 2d- und einer kommandozeilengesteuerten 3d-Version vor. Beide Versionen teilen den gleichen Simulationscode, bei der 2d-Version ist die Szene nur eine Zelle tief und liegt zwischen zwei reibungsfreien „Glasscheiben“. Durch diese Vorgehensweise ist es möglich, die meisten Fehler im Simulationscode anhand der einfacheren 2d-Version zu suchen.

[9] Eine gute Erklärung dieser Paradigmen findet sich in [7].

4.2.1 2d-Version

Die 2d-Version ist als ein grafisches Programm mit Maussteuerung realisiert. Dadurch, daß die Flüssigkeit nur in zwei Dimensionen simuliert wird, kann der Benutzer den Verlauf der Simulation ohne lange Wartezeiten mitverfolgen und es läßt sich das innere Geschehen, wie Wirbel und Strömungen, beobachten. Die 2d-Version ist zwar nicht in der Lage, räumliche Animationen zu erzeugen, ist aber eine große Hilfe beim Entwickeln und Debuggen.

Ein Beispiellauf der 2d-Version ist auf Abbildung 2 zu sehen. Die Partikel sind als kurze Linien dargestellt, die parallel zur Bewegungsrichtung liegen, wodurch die Strömungslinien erkennbar werden. Die Farbe der Partikel entspricht dem lokalen Druck (je rötlicher, desto höher), sowie dem Betrag der Geschwindigkeit (je grünlicher, desto höher). Die Trennschicht zwischen der Flüssigkeit und dem Feststoff, die an manchen Stellen sichtbar wird, ist eine Folge davon, daß die eigentliche Simulation in einem diskreten Zellengitter stattfindet. Dieser Effekt ist bei einer 3d-Animation weit weniger auffällig.

Als Quelldatei diente hier die gleiche Szenendatei wie in 3, nur wurde sie mit der 2d-Version von Falmaidan verarbeitet. Die 2d-Szene stellt einen Schnitt durch die 3d-Szene dar.

4.2.2 3d-Version

Die 3d-Version ist ein Kommandozeilenprogramm. Aus der Szene und den Simulationsparametern, die ihr als Kommandozeilenoptionen übergeben werden, erzeugt sie für jedes Einzelbild der Animation eine für den Renderer geeignete Szenenbeschreibung. Man kann mit dem Rendern bereits beginnen, bevor alle Szenen berechnet worden sind und auf diese Weise kontrollieren, ob die Simulation den Vorstellungen entsprechend voranschreitet.

Nachdem alle Einzelbilder gerendert worden sind, können sie mit Hilfe eines geeigneten Encoders zu einer Animation in einem der verbreiteten Formate aufbereitet werden.

Abbildung 3 enthält drei Einzelbilder einer einfachen Animation. In der Zukunft wird es auch möglich sein, durchscheinendes „Wasser'mit Lichtbrechung und Kaustiken zu erzeugen, was den Realismus der Bilder noch deutlich erhöhen sollte.

4.3 Modularität

Das Programm ist modular aufgebaut. Es besteht aus den beiden Hauptprogrammen (2d- und 3d-Version), den Ein- und Ausgabemodulen und einem Simulationsmodul. Die Module sind in *namespaces*[10] gekapselt und die Schnittstellen zwischen Modulen sind möglichst einfach gehalten und wurden als *abstrakte Basisklassen*[11] definiert.

Der modulare Aufbau macht den Code übersichtlicher und erleichtert Erweiterungen, wie beispielsweise den Einbau eines weiteren Simulationsverfahrens oder eines Ein-/Ausgabemoduls für einen anderen Renderer. Er ermöglicht es auch, eine 2d-Version und eine 3d-Version des Programmes zu erstellen, die den gleichen Simulationscode nutzen.

[10] Namespaces gliedern den Code in mehrere Teile mit jeweils einem eigenen Namensraum, so daß beispielsweise in zwei Namespaces zwei Klassen mit gleichem Namen definiert werden können, ohne daß ein Konflikt auftritt.

[11] Von einer abstrakten Basisklasse werden eine oder mehrere Klassen abgeleitet, welche die eigentliche Funktionalität enthalten. Diese Technik ermöglicht es explizit und durch den Compiler überprüfbar Schnittstellen von der Implementierung zu trennen.

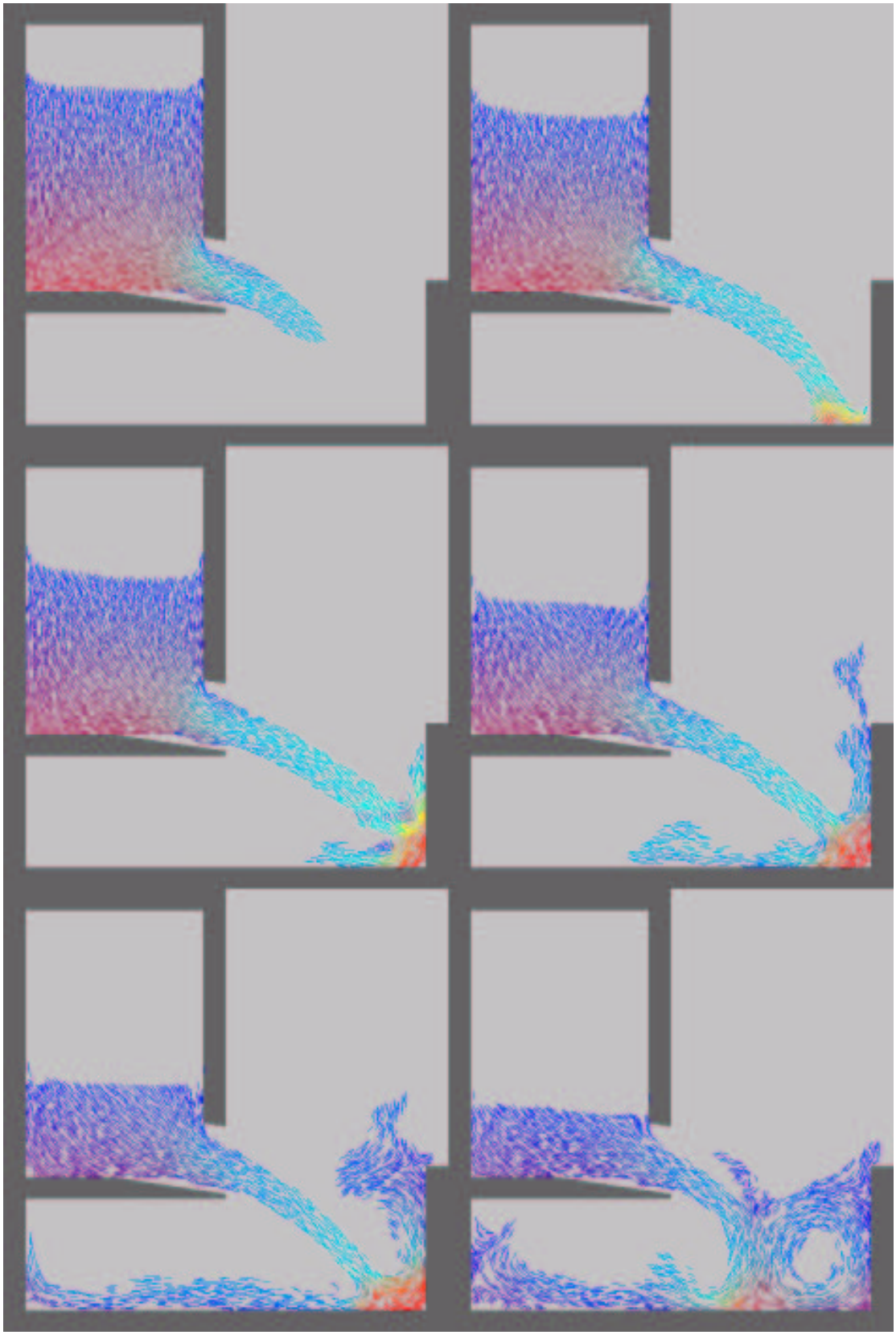

Abbildung 2: Ein Beispiellauf der 2d-Version.

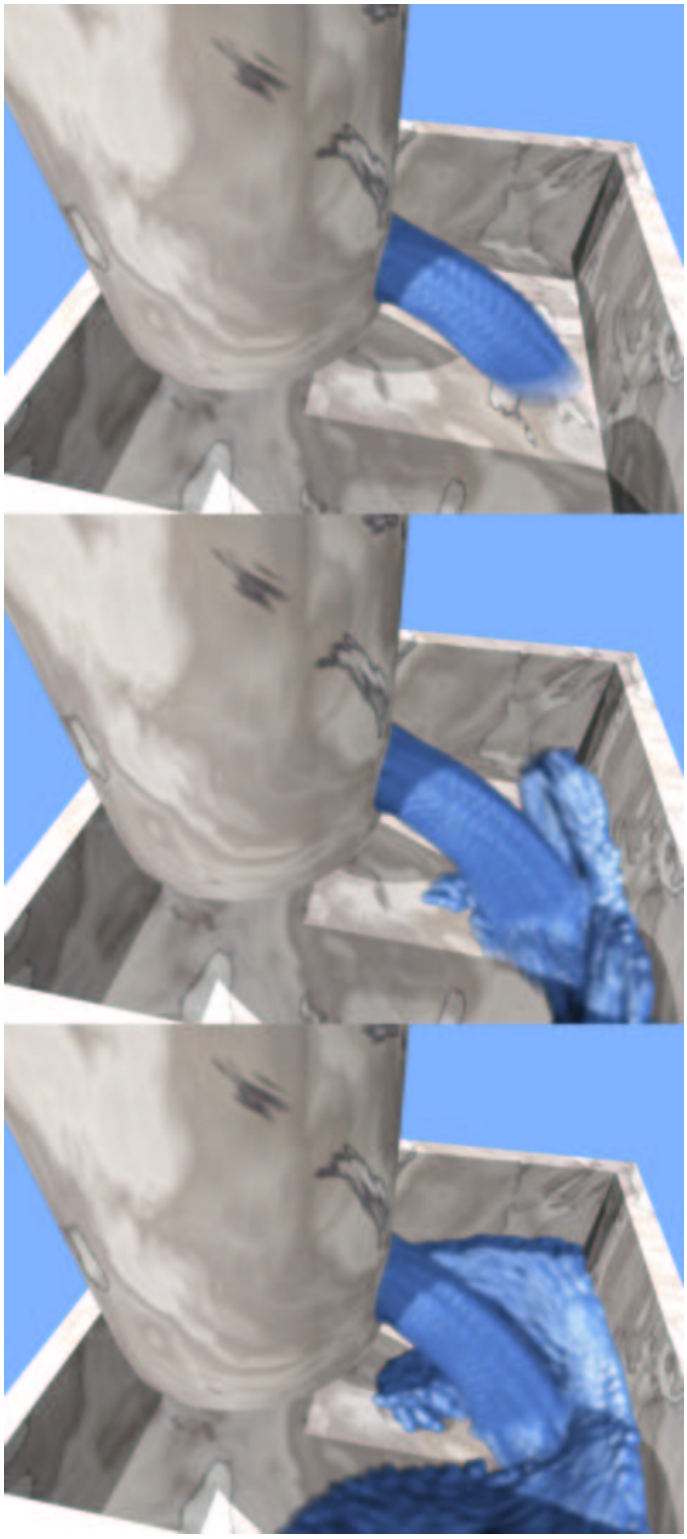

Abbildung 3: Ein Beispiellauf der 3d-Version.

4.4 Objekt-orientierte Funktionsweise

Die von C++ gebotene Möglichkeit, durch Vererbung Hierarchien von Datentypen aufzubauen, hat es unter anderem ermöglicht, das Eingabemodul für POV-Ray-Szenen elegant zu realisieren. POV-Ray-Szenen setzen sich aus geometrischen Grundkörpern wie Kugeln, Quadern oder Kegeln zusammen, die miteinander durch Mengenoperatoren verknüpft werden. So ist es beispielsweise möglich, einen Kegel aus einem Quader „herauszuschneiden“ und das so entstandene Teilobjekt mit einer Kugel zu „verschmelzen“. Nach diesem Prinzip lassen sich auch sehr komplexe Szenen beschreiben.

Die vom Parser aufgebaute interne Szenenrepräsentation besteht aus einem Baum von Körper-Objekten, die alle von der gleichen Basisklasse abgeleitet wurden. Diese Objekte können Primitive wie Kugeln sein oder komplexe Objekte, die mehrere Unter-Körper miteinander verknüpfen.

Der Vorteil einer solchen Implementierung besteht darin, daß weitere Geometrieobjekt-Arten in das Programm eingebaut werden können, ohne bereits vorhandenenen Code ändern zu müssen.

4.5 Design des Simulationsmoduls

Mein erster Versuch, ein Flüssigkeitssimulationsprogramm zu schreiben [6] resultierte in einem sehr unübersichtlichem Quellcode. Das Problem lag darin, daß während der Entwicklung dem berühmten Satz von Donald Knuth „Premature optimization is the root of all evil“ zu wenig Beachtung geschenkt wurde.

4.5.1 Alte Designfehler

Im Nachhinein betrachtet wurden beim letztjährigen Simulationsmodul folgende Designfehler begangen:

- Der Simulationscode war monolithisch innerhalb einer einzigen Klasse zusammengefaßt.
- In der Hoffnung auf eine gewisse Geschwindigkeitssteigerung und etwas geringeren Speicherverbrauch wurden Werte wie der Druck oder die Geschwindigkeitskomponenten in getrennten Arrays gehalten, die noch dazu leicht unterschiedliche Dimensionen hatten. Die Folge war, daß innerhalb von Schleifen jeweils bis zu fünf Zeiger „mitgeschleppt“ und auch bei jeder Schleifeniteration aktualisiert werden mußten. Es mußte penibel darauf geachtet werden, daß die einzelnen Zeiger nicht außer Synchronisation geraten.

Diese Fehler wirkten sich wegen der relativen Komplexität der Simulationsberechnungen, wie beispielsweise (3), verstärkt aus. Auf die Spitze getrieben wurden die Probleme aber erst dadurch, daß der meiste Simulationscode in dreifacher Ausführung, jeweils eine für jede Raumachse, vorliegen mußte. Die drei Ausführungen waren zwar von der Funktion her gleich, doch mußten, um eine Ausführung aus einer anderen zu erhalten, nach einem nicht ganz einfachen Schema alle Variablennamen geändert und Zähler und Begrenzungen geändert werden – eine mühevolle und fehleranfällige Kleinarbeit.

Um nur ein Beispiel zu nennen, lag der unmittelbare Grund für die Instabilität des Programmes aus [6] zum Zeitpunkt der Vorführung darin, daß an einer Stelle innerhalb von rund 1000 Zeilen Simulationscode ein „y“ statt eines „x“ stand. Der Code war an

sich korrekt, nur wurde bei der Erzeugung der y-Version einer Berechnung aus der x-Version der Name einer Variable nicht verändert. Solche Fehler lassen sich wegen der enormen Menge an Simulationsdaten und der Komplexität der Berechnungen kaum mit einem Debugger finden, so daß man auf mühseliges Herumprobieren angewiesen ist.

4.5.2 Neue Version

Bei der Entwicklung der neuen Version wurde versucht, die im vorangegangenem Abschnitt beschriebenen Fehler zu vermeiden. Anstatt die Größen in unabhängigen Arrays zu speichern, existiert nur noch ein Array von Objekten des Typs `Cell`. Ein `Cell`-Objekt kapselt die Geschwindigkeitskomponenten von drei Zellenseiten[12], den Druck, den Zellentyp und die darauf anwendbaren Operationen.

4.5.3 Halbautomatische Simulationscodegenerierung

Damit es nicht mehr nötig ist, den Simulationscode in dreifacher Ausführung zu schreiben, wurde eine neuartige Technik angewandt: Sie bedient sich des Templatemechanismus[13] von C++ und ermöglicht es, mehrere Versionen aus einer automatisch generieren zu lassen.

Die Technik basiert auf Familien von untereinander austauschbaren Klassen, die verschiedene Konzepte repräsentieren und als Templateparameter Verwendung finden. Es existieren beispielsweise die Klassen `P` und `N`, die für Subtraktion und Addition oder auch „positive Richtung“ und „negative Richtung“ stehen. Man definiert auch die Klassen `X`, `Y` und `Z`, welche die Raumachsen und ihre Beziehungen untereinander abstrahieren.

Beim Aufruf dafür vorbereiteter Templatefunktionen werden einige dieser speziellen Klassen als Templateparameter angegeben. Um beispielsweise bei einer Zelle `c` auf die Nachbarzelle in negativer x-Richtung zugreifen zu können, schreibt man `n<N,X>(c)`. Letztendlich erzeugt der Compiler daraus Code, der zu `(c - 1)` äquivalent ist, doch „weiß“ die Funktionsdefinition von `n` nicht, mit welcher Kombination aus Achse und Richtung sie aufgerufen werden wird. Der Programmierer braucht statt sechs nur eine Version von `n` zu schreiben und später gegebenenfalls zu ändern.

Man kann aus Aufrufen einfacher Templatefunktionen komplizierte Templatefunktionen zusammensetzen, die nach dem gleichen Prinzip funktionieren. Der Compiler erzeugt für jede Kombination von Templateparametern eigenen Maschinencode, so daß die Laufzeiteffizienz mit „handgeschriebenem“ Code vergleichbar ist.

Dieses einfache Verfahren hat sich in der Programmierpraxis als eine kaum zu überschätzende Hilfe bewährt: Der Quellcode ist deutlich kleiner und übersichtlicher. Der Hauptvorteil liegt aber darin, daß nun die 3 Versionen einer Routine immer einander exakt entsprechen – vorbei sind die Tage mysteriöser Programmfehler, die dadurch entstanden, daß eine Änderung nicht konsistent in allen drei Versionen durchgeführt wurde.

[12] Da jeweils zwei Zellen eine gemeinsame Seite besitzen, speichert jedes Zellenobjekt nur drei Geschwindigkeitskomponenten und erfragt die Fehlenden von angrenzenden Zellenobjekten

[13] Templates ermöglichen generische Programmierung in C++. Das heißt, daß der vom Programmierer geschriebene Code nicht auf einen bestimmten Datentyp bezogen sein muß, sondern erst der Compiler aus einer Templatedefinition die gewünschten Templateinstanzen erzeugt. Beispielsweise ist es möglich, eine `List`-Templateklasse zu schreiben. Aus dieser Templateklasse erzeugt der Compiler bei Bedarf zum Beispiel eine Liste von Ganzzahlen `List<int>` oder eine Liste von Mitarbeitern `List<Empolyee>`. Genauere Informationen hierüber finden sich in [7].

5 Die mitgelieferte CD

Auf der mitgelieferten CD findet sich der Quellcode der letztjährigen und der neuen Version von Falmaidan in gzip-komprimierten tar-Archiven. Ferner enthält die CD einige Beispielsanimationen im MPEG- und FLI-Format. Nähere Hinweise können der Datei LIESMICH im Wurzelverzeichnis der CD entnommen werden.

Falmaidan wurde auf einem GNU/Linux-System unter Verwendung des GNU-C++-Compilers entwickelt, es sollte jedoch auch auf anderen Unix-Arten und vielleicht auch unter Windows compilierbar sein. Zum Compilieren der 2d-Version ist die Qt-Klassenbibliothek von Troll Tech[14] notwendig, die für das X Window System frei verfügbar ist.

6 Ausblick

Mittelfristig sind verschiedene Erweiterungen denkbar. Es ist geplant, das Programm um Unterstützung von farbigen Flüssigkeiten, die sich miteinander vermischen können, zu erweitern. Sicherlich wären damit sehr ansprechende Effekte möglich. Das Problem liegt in diesem Fall eher beim Renderer, denn eine solche verschiedenfarbige Flüssigkeit müßte an ihn als eine räumliche Textur übergeben werden. Da meines Wissens nach weder POV-Ray noch ein anderer gängiger Raytracer so etwas unterstützt, müßte die Flüssigkeit aus einer Unmenge von winzigen verschiedenfarbigen Körpern aufgebaut werden.

Bisher muß die Szenengeometrie unbeweglich sein, was in vielen Fällen eine große Einschränkung ist. Man könnte kleine Objekte (beispielsweise Blätter in einem Teich) mit der Flüssigkeit mitschwimmen lassen ohne sie zu beeinflussen, was noch recht einfach zu realisieren wäre. Eine realistische Simulation der Bewegung von Fluiden und festen Körpern und ihrer Interaktion untereinander wäre zwar sehr kompliziert und aufwendig, aber langfristig sicherlich eine interessante Herausforderung.

7 Danksagung

Mein Dank gilt Matthias Baas für die interessanten und kompetenten E-Mail-Diskussionen. Ich bin auch Jochen Winkler und Hans-Georg Friedmann dafür dankbar, daß sie mir ihre eigene Hardware ausliehen und somit die weitere Arbeit möglich machten. Nicht zuletzt danke ich Claudia Knake, Stefanie Öhrlein, Sebastian Götschel und Matthias Görner für das Korrekturlesen und ihre Verbesserungsvorschläge.

[14] `http://www.trolltech.com/`

A Physikalische Grundlagen

Der Anhang soll die physikalischen Grundlagen der in dieser Arbeit vorgestellten Simulationsmethoden erläutern. Er stellt keine vollwertige Einführung in die Mechanik der Fluide dar, hierfür sind Fachbücher, wie [4] oder [2], die das Thema ausführlich behandeln, besser geeignet. Vielmehr ist er als ein „Hydrodynamik-Crashkurs für Programmierer“ gedacht, in dem die für das Verständnis der Arbeit wichtigen Eigenschaften von Flüssigkeiten kompakt besprochen und daraus auf möglichst einfache Weise die Navier-Stokes-Gleichungen hergeleitet werden.

A.1 Was sind Flüssigkeiten?

Für Flüssigkeiten[15] gelten die gleichen grundlegenden physikalischen Gesetze wie für feste Körper. Die beiden Stoffklassen unterscheiden sich aber darin, daß zwischen benachbarten Feststoffteilchen starke Bindungen bestehen, die Bindungen zwischen benachbarten Flüssigkeitsteilchen jedoch viel schwächer sind. Folglich kann man einen festen Körper nur begrenzt verformen – irgendwann wird die verformende Kraft zu groß und der Stoff reißt. Eine Flüssigkeit dagegen kann auch mit einer geringen Kraft unbegrenzt verformt werden, lediglich die Geschwindigkeit der Verformung hängt von der Kraft ab.

Der Zustand einer Flüssigkeit wird an jedem zu ihr gehörigem Punkt durch mehrere Größen bestimmt. Die Wichtigsten davon sind:

- Dichte ϱ
- Druck p
- Geschwidigkeitsvektor $\vec{v}$
- Viskosität (Zähigkeit) η
- Temperatur T

Eine Erhöhung der Temperatur bewirkt bei Flüssigkeiten hauptsächlich eine Erniedrigung der Viskosität, daher kann der Effekt einer Temperaturänderung durch Ändern der Viskosität erreicht werden und somit kann die Temperatur in den folgenden Betrachtungen außer Acht gelassen werden. Die Bedeutung der Größen Dichte, Druck und Geschwindigkeit wird als bekannt vorausgesetzt.

Die Dichte wird als konstant angenommen, die Flüssigkeiten sind also *dichtebeständig*, das heißt sie können (wie beispielsweise Wasser) nur vernachlässigbar weit komprimiert werden.

A.2 Viskosität

Auch wenn die Teilchen eines Fluids frei beweglich sind, treten bei gegenseitiger Verschiebung Reibungskräfte auf. Mißt man die Kraft, die notwendig ist, um zwei gleichgroße, parallele, in geringem Abstand zueinander liegende Platten mit dem Geschwindigkeitsunterschied Δv, zwischen denen sich die zu untersuchende Flüssigkeit befindet, zu verschieben (Abb. 4), so stellt man fest, daß die Kraft für die meisten Flüssigkeiten

[15] Flüssigkeiten werden hier immer makroskopisch betrachtet. Eine Berücksichtigung mikroskopischer Effekte, wie beispielsweise der Brownschen Molekularbewegung, würde eine andere Betrachtungsweise erfordern und wäre für die Computergrafik eher uninteressant.

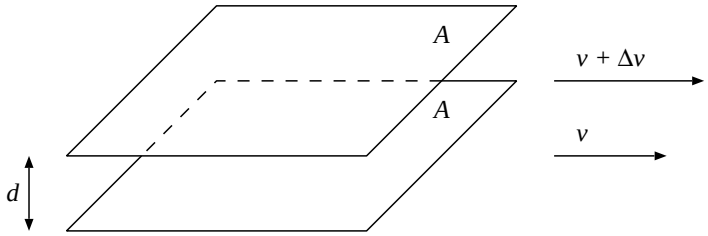

Abbildung 4: Bestimmung der Viskosität.

proportional zu Δv und umgekehrt proportional zum Abstand d ist, oder anders gesagt proportional zu $\Delta v/d$ – dem sogenannten Geschwindigkeitsgradienten. Es ist außerdem einleuchtend, daß bei einer Verdopplung der Fläche auch die Kraft verdoppelt werden muß, so daß auch hier eine Proportionalität vorliegt. Es gilt also:

$$\frac{F}{A} \sim \frac{\Delta v}{d}.$$

Die Proportionalitätskonstante dieser Beziehung

$$\eta = \frac{F}{A}\frac{d}{\Delta v} \tag{5}$$

nennt man Viskosität oder Zähigkeit eines Stoffes und sie läßt sich für einen Stoff und eine bestimmte Temperatur experimentell bestimmen. Flüssigkeiten, bei denen die auftretenden Reibungskräfte sich mit diesem Gesetz beschreiben lassen, werden *Newtonsche Flüssigkeiten* genannt. Es fallen hierunter anorganische Verbindungen wie Wasser, Glyzerin oder Quecksilber, aber auch Öle.

Viele organische Fluide, beispielsweise Blut, verschiedene Polymere oder Ketchup zählen jedoch nicht zu den Newtonschen Flüssigkeiten, denn ihre Viskosität ist auch von den inneren Spannungen abhängig. Da man ihr Verhalten aber trotzdem für die Zwecke der Computergrafik, bei der es ja letztendlich nur auf den visuellen Eindruck ankommt, recht gut durch Newtonsche Flüssigkeiten annähern kann, werden sie hier nicht weiter beachtet.

A.3 Kräfte in Flüssigkeiten

Betrachten wir einen sehr kleinen, würfelförmigen „Tropfen“ mit der Seitenlänge δs, dem Volumen $\delta V = \delta s^3$ und der Masse $\delta m = \varrho \delta V$, der sich irgendwo in einer größeren Menge Flüssigkeit befindet. Auch für diesen Tropfen gilt der Hauptsatz der Mechanik:

$$\delta m \vec{a} = \vec{F}.$$

Auf den Tropfen wirken verschiedene Kräfte, wovon folgende für sein Verhalten maßgeblich sind:

- Volumenkräfte
- Druckkräfte
- Reibungskräfte
- Durch Oberflächenspannung hervorgerufene Kräfte

Der Einfluß der Oberflächenspannung wird im weiteren vernachlässigt, weil ihre Berechnung zu aufwendig wäre. Da das Verhältnis von Oberfläche zu Volumen mit größerem Maßstab der Szene sinkt, wird auch die Oberflächenspannung mit zunehmender Größe der simulierten Szene glücklicherweise immer unwichtiger.

Die übrigen drei Kräfte ergeben addiert die auf den Tropfen wirkende Gesamtkraft:

$$\delta m\vec{a} = \varrho\delta V\vec{a} = \vec{F} = \vec{F}_{\text{Volumen}} + \vec{F}_{\text{Druck}} + \vec{F}_{\text{Reibung}}. \tag{6}$$

A.3.1 Volumenkräfte

Volumenkräfte sind der Oberbegriff für „äußere Kräfte“, wie zum Beispiel Gravitationskräfte oder Scheinkräfte in beschleunigten Systemen. Sie sind im Allgemeinen proportional zur Masse des Tröpfchens und so gilt:

$$\vec{F}_{\text{Volumen}} = \delta m\vec{g} = \varrho\delta V\vec{g}.$$

A.3.2 Druckkräfte

Druckkräfte entstehen durch Druckunterschiede innerhalb der Flüssigkeit. Der Druck in der Mitte M des Tröpfchens soll p heißen. Um auch die Druckwerte in der nächsten Umgebung ausdrücken zu können, wird die „Steigung“ des Druckes benötigt. Da es sich hierbei aber um eine räumliche Funktion handelt, ist diese „Steigung“ ein Vektor, dessen drei Komponenten aus den „Steigungen“ in jeweils x-, y-, und z-Richtung bestehen. Diese lassen sich durch partielles Ableiten der Druckfunktion bestimmen. Der so definierte Vektor

$$\nabla p = \begin{pmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \\ \frac{\partial p}{\partial z} \end{pmatrix}$$

wird Druckgradient genannt, wobei $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$ der sogenannte Nablaoperator ist.

Da das Tröpfchen sehr klein ist, kann man annehmen, daß die Druckfunktion in der Umgebung linear verläuft. Daher läßt sich der Druck in einem genügend nahe an der Mitte M gelegenen Punkt T wie folgt berechnen:

$$p_T = p + \overrightarrow{MT} \cdot \nabla p.$$

Um die weiteren Betrachtungen zu vereinfachen, wird im Folgenden angenommen, daß vier der Kanten des Tröpfchens parallel zum Druckgradienten verlaufen (Abb. 5). Diese Annahme schränkt die Allgemeingültigkeit der Betrachtungen jedoch nicht ein – man kann nach dem Bausteinprinzip jeden Körper durch genügend kleine „parallele Tröpfchen“ beliebig genau annähern.

Da die Kräfte, die auf die vier zum Gradienten parallelen Seiten wirken, betragsgleich und jeweils paarweise entgegengesetzt sind, heben sie sich auf. So müssen nur noch die auf die beiden übrigen Seiten wirkenden Kräfte betrachtet werden. Die auf die in Richtung des Gradienten liegende Seite nach innen wirkende Kraft

$$F_{\text{vorne}} = \delta s^2 \left(p + \frac{\delta s}{2} |\nabla p| \right)$$

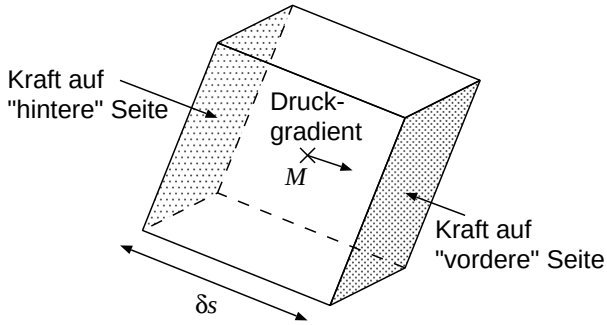

Abbildung 5: Auf einen „Tropfen“ wirkende Druckkräfte.

und die auf die gegenüberliegende Seite ebenfalls nach innen wirkende Kraft

$$F_{\text{hinten}} = \delta s^2 \left(p - \frac{\delta s}{2} |\nabla p| \right)$$

ergeben zusammen die in Richtung des Gradienten wirkende Kraft

$$F_{\text{Druck}} = F_{\text{hinten}} - F_{\text{vorne}} = -\delta s^3 |\nabla p| = -\delta V |\nabla p|. \tag{7}$$

Um den Druckkraftvektor zu erhalten, multipliziert man (7) mit dem Einheitsvektor $\nabla p/|\nabla p|$, welcher die Richtung der Druckkraft, die ja der Richtung des Druckgradienten entspricht, angibt:

$$\vec{F}_{\text{Druck}} = -\delta V \nabla p.$$

A.3.3 Reibungskräfte

Druckkräfte wirken auf eine Fläche immer senkrecht. Reibungskräfte dagegen können in jede beliebige Richtung wirken, weshalb ihre Herleitung auch am aufwendigsten ist.

Betrachten wir eine kleine Fläche δA, die parallel zur y/z-Ebene liegt. Man kann sie sich als die Seite eines kleinen Würfels, der ins Wasser getaucht wurde, vorstellen. Wenn Wasser (oder eine andere Flüssigkeit) den Würfel umspült, wird durch Reibung eine Kraft auf die Seiten verursacht. Der Kehrbruch des Flächeninhalts der Seite multipliziert mit der Kraft $\vec{\sigma}_x = \frac{1}{\delta A}\vec{F}$ wird Spannung genannt. Die drei Komponenten des Spannungsvektors sollen nun hergeleitet werden.

Die y-Komponente σ_{xy} der Spannung, die auf die kleine Fläche wirkt, läßt sich aus der Definition der Viskosität (Abschnitt A.2) ableiten. Gleichung (5) läßt sich auch schreiben als:

$$\frac{F}{A} = \eta \frac{\Delta v}{d}.$$

Setzt man für F/A die Spannung σ_{xy} und für $\Delta v/d$ den entsprechenden Geschwindigkeitsgradienten ein, so ergibt sich

$$\sigma_{xy} = \eta \frac{\partial v_y}{\partial x},$$

womit die y-Komponente der Spannung bestimmt wäre.

Da die Reibungskräfte isotrop[16] sind, gilt für die z-Komponente der Spannung entsprechend:

$$\sigma_{xz} = \eta \frac{\partial v_z}{\partial x}.$$

[16] Nicht von der Richtung abhängig.

Es läßt sich experimentell belegen, daß nicht nur die Tangentialspannungen σ_{xy} und σ_{xz}, sondern auch die Normalspannung σ_{xx} auf analoge Weise vom entsprechenden Geschwindigkeitsgradienten abhängt, woraus folgt:

$$\sigma_{xx} = \eta \frac{\partial v_x}{\partial x}.$$

Wegen der Isotropie lassen sich die Spannungen $\vec{\sigma}_y$ und $\vec{\sigma}_z$ durch entsprechende Überlegungen bestimmen. Die drei Spannungsvektoren bilden zusammen den Spannungstensor:

$$\sigma = \begin{pmatrix} \sigma_{xx} & \sigma_{yx} & \sigma_{zx} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{zy} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix} = \eta \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{pmatrix}. \tag{8}$$

In der linken Spalte befinden sich die Spannungen, die auf eine zur y/z-Ebene (Normalvektor zeigt in x-Richtung.) parallele Fläche wirken. Die übrigen Spalten enthalten dementsprechend die Spannungen auf Flächen mit Normalvektoren in y- und z-Richtung.

Nachdem der Spannungstensor bestimmt ist, soll untersucht werden, wie sich die Spannungskräfte auf das kleine, würfelförmige Tröpfchen auswirken. Die Seiten des Tröpfchens sollen diesmal parallel zu den Achsen liegen, was jedoch, wie im Abschnitt A.3.2 gezeigt wurde, keine Einschränkung der Allgemeingültigkeit bedeutet.

Das Tröpfchen hat sechs Seiten, jeweils mit der Fläche δs^2. Auf jede davon wirken durch Spannungen verursachte Kräfte, die zusammen die Reibungskraft auf das Tröpfchen ergeben. Ist in der Mitte des Tröpfchens die Spannungtensor σ und die partiellen Ableitung seiner Komponenten bekannt, so lassen sich die Spannungen für die Seitenflächen berechnen, da für den sehr kleinen Tropfen angenommen werden kann, daß die Spannungen in seiner Umgebung linear verlaufen. So ergibt sich beispielsweise

$$F_x(+x) = \delta s^2 \sigma_{xx}(+x) = \delta s^2 \left(\sigma_{xx} + \frac{\partial \sigma_{xx}}{\partial x} \frac{\delta s}{2} \right)$$

für die x-Komponente der Kraft auf die in positiver x-Richtung liegende Seite. Die anderen Kraftkomponenten[17], lassen sich analog bestimmen.

Um die Reibungskraft zu bestimmen, müssen nur noch die auf die unterschiedlichen Seiten wirkenden Kraftkomponenten zusammengefaßt werden. Fr die x-Komponente der Reibungskraft gilt somit:

$$\begin{aligned} F_{\text{Reibung},x} &= F_x(+x) + F_x(-x) + F_x(+y) + F_x(-y) + F_x(+z) + F_x(-z) \\ &= \delta s^2 \left(\sigma_{xx} + \tfrac{\partial \sigma_{xx}}{\partial x} \tfrac{\delta s}{2} \right) - \delta s^2 \left(\sigma_{xx} - \tfrac{\partial \sigma_{xx}}{\partial x} \tfrac{\delta s}{2} \right) \\ &\quad + \delta s^2 \left(\sigma_{yx} + \tfrac{\partial \sigma_{yx}}{\partial y} \tfrac{\delta s}{2} \right) - \delta s^2 \left(\sigma_{yx} - \tfrac{\partial \sigma_{yx}}{\partial y} \tfrac{\delta s}{2} \right) \\ &\quad + \delta s^2 \left(\sigma_{zx} + \tfrac{\partial \sigma_{zx}}{\partial z} \tfrac{\delta s}{2} \right) - \delta s^2 \left(\sigma_{zx} - \tfrac{\partial \sigma_{zx}}{\partial z} \tfrac{\delta s}{2} \right) \\ &= \delta V \left(\tfrac{\partial \sigma_{xx}}{\partial x} + \tfrac{\partial \sigma_{yx}}{\partial y} + \tfrac{\partial \sigma_{zx}}{\partial z} \right). \end{aligned}$$

Setzt man für die Spannungen ihre Definitionen aus (8) ein, so erhält man:

$$F_{\text{Reibung},x} = \delta V \eta \left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} \right).$$

[17] Es sind insgesamt 18 – sechs Seiten mit jeweils drei Komponenten.

Die x- und y-Komponenten der Kraft lassen sich analog bestimmen, so daß man den Vektor der Reibungskraft

$$\vec{F}_{\text{Reibung}} = \delta V \eta \begin{pmatrix} \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} \\ \frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2} + \frac{\partial^2 v_y}{\partial z^2} \\ \frac{\partial^2 v_z}{\partial x^2} + \frac{\partial^2 v_z}{\partial y^2} + \frac{\partial^2 v_z}{\partial z^2} \end{pmatrix} = \delta V \eta \nabla^2 \vec{v}$$

erhält, wobei $\nabla^2 = \nabla \cdot \nabla = (\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2)$ der sogenannte Laplaceoperator ist.

A.3.4 Navier-Stokes

Nachdem die drei unterschiedlichen Kräfte hergeleitet wurden, können ihre Definitionen in die Bewegungsgleichung des Tröpfchens (6) eingesetzt werden:

$$\varrho \delta V \vec{a} = \vec{F}_{\text{Volumen}} + \vec{F}_{\text{Druck}} + \vec{F}_{\text{Reibung}} = \varrho \delta V \vec{g} - \delta V \nabla p + \delta V \eta \nabla^2 \vec{v}.$$

Beidseitiges Dividieren durch $\varrho \delta V$ ergibt die sogenannten Navier-Stokes-Gleichungen für dichtebeständige Fluide:

$$\vec{a} = \frac{D\vec{v}}{Dt} = \vec{g} - \frac{1}{\varrho} \nabla p + \nu \nabla^2 \vec{v},$$

wobei $\nu = \eta/\varrho$ die sogenannte kinematische Viskosität ist.

Die Schreibweise

$$\frac{Df}{Dt} = \underbrace{\frac{\partial f}{\partial t}}_{1} + \underbrace{v_x \frac{\partial f}{\partial x} + v_y \frac{\partial f}{\partial y} + v_z \frac{\partial f}{\partial z}}_{2} = \underbrace{\frac{\partial f}{\partial t}}_{1} + \underbrace{\vec{v} \cdot \nabla f}_{2} \tag{9}$$

bezeichnet die *material derivative*[18] einer skalaren oder vektoriellen Größe f. Die „gewöhnliche“ Ableitung einer Größe nach der Zeit $\partial f/\partial t$ entspricht der zeitlichen Veränderung von f an einer bestimmten Stelle. Bei der material derivative dagegen wird f für einen Punkt, der „mitschwimmt“ abgeleitet. Der Term 1 von (9) berücksichtigt hierbei die Änderung von f wegen der verstreichenden Zeit und Term 2 wegen der Bewegung des „Meßpunktes“.

Die endgültige Form der Navier-Stokes-Gleichungen für dichtebeständige Fluide ist demnach:

$$\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} = \vec{g} - \frac{1}{\varrho} \nabla p + \nu \nabla^2 \vec{v}, \tag{10}$$

A.4 Dichtebeständigkeit

Wenn eine Flüssigkeit dichtebeständig ist, muß zu jedem Zeitpunkt genauso viel Masse in jedes festgelegte Volumen hineinfließen, wie herausfließt: $\partial m/\partial t = 0$. Für ein kleines, würfelförmiges Volumenelement $\delta V = \delta s^3$, dessen Kanten parallel zu den Raumachsen

[18] Der entsprechende deutsche Begriff ist mir leider unbekannt.

liegen, gilt für die Bilanz der Masse, die pro Zeiteinheit die Elementgrenze überschreitet:

$$\begin{aligned}\frac{\partial m}{\partial t} &= \varrho\delta s^2\left(v_x + \frac{\partial v_x}{\partial x}\frac{\delta s}{2}\right) - \varrho\delta s^2\left(v_x - \frac{\partial v_x}{\partial x}\frac{\delta s}{2}\right)\\ &\quad +\varrho\delta s^2\left(v_y + \frac{\partial v_y}{\partial y}\frac{\delta s}{2}\right) - \varrho\delta s^2\left(v_y - \frac{\partial v_y}{\partial y}\frac{\delta s}{2}\right)\\ &\quad +\varrho\delta s^2\left(v_z + \frac{\partial v_z}{\partial z}\frac{\delta s}{2}\right) - \varrho\delta s^2\left(v_z - \frac{\partial v_z}{\partial z}\frac{\delta s}{2}\right)\\ &= \varrho\delta V\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right)\\ &= \varrho\delta V\nabla\cdot\vec{v}\end{aligned}$$

Um die Dichtebeständigkeit zu garantieren muß somit zu jedem Zeitpunkt und an jedem Punkt der Flüssigkeit

$$\nabla\cdot\vec{v} = 0 \tag{11}$$

erfüllt sein.

Literatur

[1] Baas, Matthias: *Fotorealistische Wasseranimation bei Brunnenanlagen.* Diplomarbeit, Universität Karlsruhe, 2000.
`http://i31www.ira.uka.de/~baas/wasser/`.

[2] Chorin, Alexandre J. und Jerrold E. Marsden: *A Mathematical Introduction to Fluid Mechanics.* Springer-Verlag, 1993.

[3] Desbrun, Mathieu und Marie-Paule Gascuel: *Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies.* In: *6th Eurographics Workshop on Animation and Simulation*, 1996.
`http://www.multires.caltech.edu/~mathieu/smoothed.html`.

[4] Fay, James A. und Nishikant Sonwalkar: *A Fluid Mechanics Hypercourse.* Massachusetts Institute of Technology, 1991.

[5] Foster, Nick und Dimitri Matexas: *Realistic Animation of Liquids.* Graphical models and image processing, 1996.
`http://www.cis.upenn.edu/~fostern/liquid.html`.

[6] Groth, Christoph: *Realistische Animation von Flüssigkeiten für Zwecke der Computergrafik.* Jugend-Forscht-Arbeit, 2000.

[7] Stroustrup, Bjarne: *The C++ Programming Language, Special Edition.* Addison-Wesley, 2000.