Skriptbasiertes Animationssystem zur Rauchsimulation

von

Frank Pagels

Diplomarbeit am
Fachbereich Informatik
der Universität Rostock

Betreuer: Prof. Dr.-Ing. habil Dietmar Jackèl

Gutachter: Prof. Dr.-Ing. habil Dietmar Jackèl
Prof. Dr.-Ing. habil Heidrun Schumann

Institut für Computergraphik
Universität Rostock
13. Januar 2004

Wer nichts weiß, muss alles glauben.

Marie von Ebner-Eschenbach

Abstract

Die realitätsnahe Simulation von Feuer und Rauch ist immer noch eine große Herausforderung in der Computergraphik. Da es sich hier um Probleme aus der Strömungsmechanik handelt, kann auf Methoden der CFD (computational fluid dynamics) zurückgegriffen werden. In dieser Diplomarbeit wurde das Animationssystem DUSTY entwickelt, um damit, unter Benutzung des *Stable Fluids* Algorithmus von *Jos Stam*, in Echtzeit Rauch und Feuer zu simulieren. Durch ein flexibles Skriptsystem können Szenen erstellt und animiert werden. Es können zusätzlich Hindernisse in den Simulationsraum eingefügt werden, welche den Strömungsverlauf in realitätsnaher Weise beinflussen. Eine Beeinflussung der Simulation durch externe Wind- und Temperaturfelder ist ebenfalls möglich. Die vorliegende Arbeit gibt einen Überblick über die Grundlagen der Strömungsmechanik, erläutert ausführlich den benutzten *Stable Fluids* Algorithmus und das in dieser Arbeit entwickelte Animationssystem.

Realistic simulation of fire and smoke is still a challenging problem in computer graphics. Because these problems are a kind of fluid-dynamics we can use methods of CFD (computational fluid dynamics). In this diploma thesis the animation system DUSTY was developed using the *Stable Fluids* Algorithm from *Jos Stam*. It allowes to simulate smoke and fire in realtime. With a flexible scripting system one can build and animate scenes. Containing obstacles can be placed within the simulation universum to affect the fluid in a near realistic way. This paper extensivly explains the *Stable Fluids* Algorithm and gives a short introduction to the fluid-dynamics theory.

CR-Categories: I.3.5 [Computer Graphics] Computational Geometry and Object Modeling—Physically based modeling; I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism—Animation

Keywords: computational fluid dynamics, gaseous phenomena, navier-stokes, real-time simulation, smoke, stable fluids

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Fluide und Strömungen . . . 2
1.2 Geschichte der Fluidmechanik . . . 4
1.3 Numerische Simulation . . . 5

2 Grundgleichungen der Strömungsmechanik **9**
2.1 Das Kontinuum . . . 9
2.2 Erhaltung der Masse (Kontinuitätsgleichung) . . . 11
2.3 Erhaltung des Impulses . . . 12

3 Numerische Lösung der Strömungsgleichungen **15**
3.1 Diskretisierung . . . 16
3.2 Finite Differenzen Methode (FDM) . . . 16
3.2.1 Zeitdiskretisierung . . . 16
3.2.2 Raumdiskretisierung . . . 18
3.2.2.1 Praktische Anwendung auf die *Navier-Stokes* Gleichungen 19
3.2.2.2 Poisson-Gleichung . . . 20
3.2.3 Randbedingungen . . . 21
3.2.3.1 Haftbedingung . . . 22
3.2.3.2 Rutschbedingung . . . 22
3.2.3.3 Ausströmbedingung . . . 23
3.2.3.4 Einströmbedingung . . . 23
3.3 Numerische Stabilität . . . 23

4 Stable Fluids Algorithmus **25**
4.1 Grundgleichungen . . . 26
4.2 Lösen der Gleichungen . . . 28
4.3 Bewegung der Dichte . . . 28
4.3.1 Diffusion . . . 29
4.3.2 Transport . . . 31
4.3.3 Partikel-Verfolgung . . . 33
4.3.3.1 *Euler* Integration . . . 34
4.3.3.2 *Runge-Kutta* Verfahren . . . 34

4.4 Bewegung des Vektorfelds . . . 35
4.4.1 Erhaltung der Masse . . . 36
4.4.2 Vorticity Confinement . . . 37
4.5 Ablauf der Simulation . . . 38
4.6 Lösen der Gleichungen mittels FFT . . . 40

5 Dusty - Eine Skriptbasierte Rauchsimulation 41
5.1 Benutzte Komponenten und Bibliotheken . . . 41
5.1.1 TEKlib . . . 42
5.1.2 LUA . . . 42
5.1.3 FLTK . . . 42
5.2 Struktur von Dusty . . . 42
5.3 *Navier-Stokes* Solver . . . 43
5.3.1 Objekte und Quellen für eine Szene . . . 44
5.3.2 Externe Kräfte und Dichten . . . 45
5.3.3 Diffusion . . . 46
5.3.4 Transport . . . 46
5.3.5 Erzeugen der Divergenzfreiheit . . . 48
5.3.6 Vorticity Confinement . . . 49
5.4 Skriptinterpreter . . . 50
5.4.1 Aufbau eins Skriptes . . . 50
5.4.2 Implementation in DUSTY . . . 51
5.5 Grafische Ausgabe . . . 53
5.5.1 Volumenrendering . . . 53
5.5.2 Partikel . . . 54
5.5.3 Stromlinien . . . 54
5.6 Benutzungsschnittstelle . . . 54

6 Zusammenfassung und Ausblick 57
6.1 Ergebnisse . . . 57
6.2 Ausblick . . . 61

A Befehlssatz von Dusty 65
A.1 Globale Methoden . . . 65
A.2 Methoden für Container . . . 67
A.3 Methoden für Objekte und Quellen . . . 70
A.3.1 Objektmethoden . . . 70
A.3.2 Quellenmethoden . . . 71

B Lösung schwach besetzter Gleichungssysteme 73
B.1 *Gauß-Seidel* Verfahren . . . 74
B.2 SOR-Verfahren . . . 74
B.3 Verfahren der konjugierten Gradienten . . . 75

C Glossar **77**

D Abbildungen **79**

Literatur **85**

Abbildungsverzeichnis

1.1 Laminare Strömung 3
1.2 Turbulente Strömung 3
1.3 Claude Navier 1785-1836, Georg Stokes 1819-1903 5
1.4 Typischer Ablauf in der numerischen Simulation (nach [16]) 7

2.1 Ein- und ausströmende Masseströme 11

3.1 Zeitdiskretisierung 17
3.2 Räumliche Diskretisierung (nach [23]) 18
3.3 Nummerierung der Unbekannten einer Poisson-Gleichung 21

4.1 Diskretisierungsgitter 28
4.2 Schritte des Algorithmus 29
4.3 Links vor und rechts die Situation nach der Diffusion 30
4.4 Austausch der Dichte bei der Diffusion 30
4.5 Tansport der Partikel durch ein festes 2D Vektorfeld (nach [29]) 32
4.6 Partikelverfolgung 32
4.7 Interpolation der Dichte 33
4.8 Partikel-Verfolgung über die Zeit 33
4.9 Euler-Integration 34
4.10 Runge-Kutta-Verfahren 35
4.11 2D Rauchsimulation 37

5.1 Aufbau des Simulations-Systems 43
5.2 Unterteilung des Universums in 2D Gitter 53
5.3 Dusty 55

6.1 Ein Teelicht mit simulierter Flamme 58
6.2 Graph zur Tabelle 6.1 60
6.3 Zu grobes Gitter beim Zeichnen eines Logos (60x60x5) 61
6.4 Simulation von Rauch mit Raytracing und Photonmapping (aus [7]) 62

D.1 Feuersimulation (30x30x10) 79
D.2 Visualisierung durch Partikeln (30x15x10) 80

D.3 Beispiel für 2D-Rauch (70x50x3) 80
D.4 Animation einer Quelle (24x30x10) 81
D.5 Verdampfen des ICG Logos (120x120x3) 81
D.6 Feuer mit Alphakanal (30x30x10) 82
D.7 Rauchverteilung in einer komplexen Szene (35x35x35) 82
D.8 Visualisierung durch Strömungslinien (25x25x25) 83

Kapitel 1
Einleitung

Simulation und Animation sind eng verbunden. Mit einer Visualisierung durch eine Animation kann z.B. das Ergebnis einer Simulation besser veranschaulicht werden als durch den bloßen Anblick der Zahlen, die eine Simulation erzeugt.
Die Grundlage einer Computersimulation ist das mathematische Modell eines realen Systems, das aus Variablen, Gleichungen und logischen Regeln besteht. Am System selbst zu experimentieren ist oft nicht möglich oder zumindest sehr teuer. So spart etwa der virtuelle Crash im Rechner Kosten in der Herstellung und für das Testmaterial. Statt Prototypen von teuren Dummys kaputt fahren zu lassen, schaut man sich das Unfallgeschehen am Bildschirm an. Ein weiteres bekanntes Beispiel, in dem Animation und Simulation eng zusammen arbeiten, ist die Wettervorhersage. Das Wettergeschehen wird in aufwändigen Berechnungen mit Supercomputern bestimmt. Eine simple Animation der Wetterverhältnisse kann dann die Ergebnisse veranschaulichen.
Die erwähnten Simulationsbeispiele erfordern viel Rechenkraft, um genaue, sinnvolle Ergebnisse zu erhalten. In der Computergraphik, insbesondere bei Spielen, benötigt man nicht immer höchste Genauigkeit. Es soll am Ende einfach nur gut und plausibel aussehen. Schon bei den gängigen Beleuchtungsmodellen, wie das Modell von Phong, wird verallgemeinert. Diese sind dann nicht mehr physikalisch korrekt, liefern aber ausreichende Ergebnisse bei guter Geschwindigkeit.

In dieser Diplomarbeit soll das Verhalten von Rauch simuliert und dargestellt werden. Bei der Simulation von Rauch handelt es sich um eine Strömungssimulation. Exakte Strömungssimulation ist sehr aufwändig und kann selten in Echtzeit erfolgen. In dieser Arbeit wird ein Verfahren benutzt, dass die Simulation von Strömungen etwas vereinfacht, um noch eine Echtzeitdarstellung zu ermöglichen. Daraus lassen sich dann zwar keine physikalisch exakten Ergebnisse gewinnen, aber der erzeugte Rauch hat ein relativ reales Verhalten und kann für Spiele oder Animationen benutzt werden. Daraus ist, im Rahmen der vorliegenden Diplomarbeit, ein Animationssystem zum realitätsnahen Simulieren von Rauch und Gas entstanden. Mit dem System können Szenen aus Körpern erstellt und das Verhalten von Rauch in der Szene beobachtet werden. Mit einem Skriptsystem kann eine Szene erstellt und animiert werden.

1.1 Fluide und Strömungen

In der vorliegenden Arbeit wird zur Animation von Rauch eine Strömungssimulation verwendet. Bei Rauch oder Gas handelt es sich um so genannte Fluids, die sich durch Strömungen im Raum verteilen. Es muss zum weiteren Verständnis zunächst geklärt werden, um was es sich bei Fluids genau handelt.

Eine erste Berührung mit Strömungen kann jeder selbst z.B. am Wasserhahn machen. Hält man den Finger in den Wasserstrahl, so verspürt man eine **Kraft** $\vec{F}$, die die Strömung auf den Finger ausübt. Diese Kraft ist der Widerstand, den ein Körper in einer Strömung erfährt. Dieser Widerstand ist abhängig von der Geometrie des umströmten Körpers, der Oberflächenbeschaffenheit und dem strömenden Medium. Der Widerstand wird einen unterschiedlichen Wert für einen Gasstrahl bzw. für den bisher betrachteten Wasserstrahl haben. Um Gase und Flüssigkeiten nicht ständig unterscheiden zu müssen, wird der Sammelbegriff des **Fluids** eingeführt.

Fluids Fluids sind Substanzen, die in Ruhelagen Scherkräften nicht widerstehen können.

Dabei muss weiterhin nach kompressiblen und inkompressiblen Fluids unterschieden werden. Gase zählen zu den kompressiblen Fluids während Flüssigkeiten, wie Wasser, zu den inkompressiblen Fluids gehören. Das strömende Fluid wird als **Kontinuum** betrachtet. Dies bedeutet, dass die molekulare Struktur des strömenden Mediums vernachlässigt wird, da die mittlere freie Weglänge der Moleküle klein gegen die charakteristische makroskopische Abmessung des Strömungsfeldes ist. Fluids begegnen uns überall. Das bekannteste Fluid ist einfach Wasser. Aber auch die Luft, die uns umgibt, ist ein Fluid.

Als Beispiel für ein bekanntes Fluid soll der Kaffee in einer Tasse dienen. Wenn man den Kaffee umrührt, reicht eine einmalige Umdrehung des Löffels, um den gesamten Kaffee für eine gewisse Zeit in Bewegung zu halten. Wie lange ein Fluid in Bewegung bleibt, hängt einerseits von den Wechselwirkungen zwischen den Fluidpartikeln untereinander, sowie andererseits von den Kräften zwischen strömendem Fluid und ruhendem Körper bzw. zwischen bewegtem Körper und ruhendem Fluid ab. Ursache für diese Kräfte ist die *Zähigkeit* bzw. *Viskosität* der Fluide. Durch sie werden Reibungskräfte verursacht, die dafür sorgen, dass sich bewegende Fluids auch ohne Einwirkung äußerer Kräfte zur Ruhe kommen. Zum Beispiel kommt die umgerührte Tasse Kaffee nach einiger Zeit durch die Reibungskräfte zur Ruhe.

Zur Erklärung dieser Kräfte, kann man sich ein Fluid als aus Schichten zusammengesetzt vorstellen, die wie ein Stapel Spielkarten übereinander liegen. Am Anfang werden diese Schichten gleichmäßig bewegt. Wenn die Vorwärtsbewegung der unteren Schicht gestoppt wird, verursacht die *Trägheitskraft* ein Weiterrutschen der darüber liegenden Schichten. Die Reibung wirkt dieser Kraft entgegen. Es entsteht eine Situation wie in Abbildung 1.1. Durch die inneren Reibung wirkt sich die Kraft, die durch das Anhalten der unteren

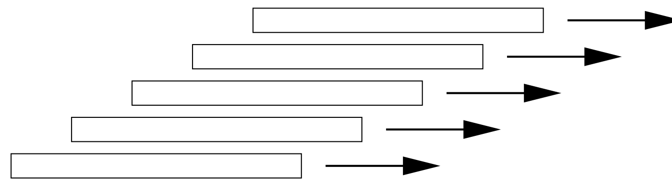

Abbildung 1.1: Laminare Strömung

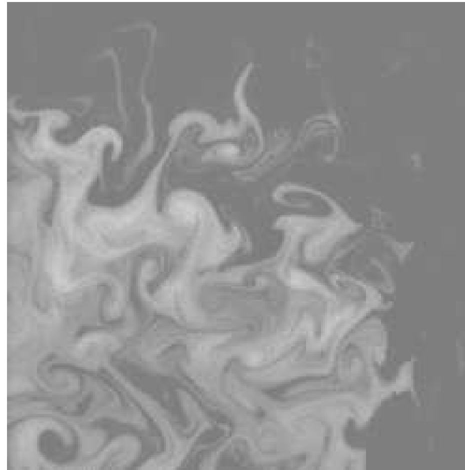

Abbildung 1.2: Turbulente Strömung

Schicht ausgeübt wird, auf die anderen Schichten aus. Dies wird als *laminare*[1] Strömung bezeichnet. Dazu gibt es noch die *turbulenten* Strömungen (Abbildung 1.2). Hier ist die Reibung so gering, dass sich die Teilchen der einzelnen Schichten vermischen können. Dies ist z.B. bei Rauch der Fall. Die Turbulenz wird daher in dem, für diese Diplomarbeit entwickelten, Animationssystem berücksichtigt.

Honig z.B. ist dagegen ein zähes Fluid. Hier ist die Reibung so stark, dass die einzelnen Schichten früher zum Stehen kommen als bei weniger viskosen Fluids, wie z.B. Wasser oder Luft. Es ist mehr Kraft nötig um einen Löffel durch den Honig zu bewegen als durch Luft. Da bei Gasen die innere Reibung sehr gering ist, kann diese häufig schon bei der Modellierung vernachlässigt werden. In diesem Idealfall werden Gase als nicht-viskose Fluids behandelt. Diese nicht-viskosen Fluids wurden in der Fluidmechanik als erstes beschrieben. Im folgenden Abschritt erfolgt eine kurze Übersicht zur Geschichte der Fluidmechanik.

[1]Von lat. *lamina:* Blech, dünne Scheibe

1.2 Geschichte der Fluidmechanik

Wie in vielen wissenschaftlichen Bereichen, besteht die Geschichte der Fluidmechanik aus einer Phase der frühen Entdeckungen, einer Zwischenära der grundlegenden Entdeckungen im 18. und 19. Jahrhundert bis zur heutigen praktischen Anwendung. Schon alte Zivilisationen hatten genug Wissen, um bestimmte Fluid- und Strömungsprobleme zu lösen. Schiffe mit Rudern und Bewässerungssysteme waren schon in frühen Zeiten bekannt.

Theoretische Vorreiter waren hier die Griechen. Archimedes und Hero von Alexandria entwickelten die Regel vom Parallelogramm zur Addition von Vektoren im dritten Jahrhundert v.u.Z. Archimedes (285–212 v.u.Z.) entdeckte das Gesetz vom Auftrieb und wendete es für schwimmende Körper an. Die Römer bauten dann im vierten Jahrhundert aufwändige Wasserleitungen (Aquädukte). Seit dem Beginn der Zeitrechnung erfolgte bis zur Renaissance eine ständige Entwicklung dieser Fluss-Systeme. Es erfolgte aber keine Aufzeichnung des erlangten Wissens. Erst Leonardo da Vinci (1452–1519) entwickelte die Gleichung zur Erhaltung der Masse im kontinuierlichen ein-dimensionalen Fluss. Leonardo war ein ausgezeichneter Experimentator und seine Aufzeichnungen enthalten genaue Bescheibungen von Wellen und Strömungen. Der Franzose Edme Mariotte (1620–1684) baute den ersten Windtunnel und testete in ihm Modelle. Impulsprobleme in Fluids konnten analysiert werden, nachdem Isaac Newton (1642–1727) seine Bewegungsgesetze und Gesetze von ideal-viskosen Flüssigkeiten vorstellte.

Die Theorie der Fluid Berechnung führte zuerst zur Annahme von perfekten reibungsfreien Flüssigkeiten. Im 18. Jahrhundert entwickelten die Mathematiker Daniel Bernoulli, Leonhard Euler, Jean d'Alembert, Joseph-Louis Lagrange und Pierre-Simon Laplace Lösungen für Probleme mit solchen Fluids. Aber solche perfekten Flüssigkeiten haben in der Praxis wenig Anwendung. In Fluids spielt die Viskosität eine Rolle, die in den Gleichungen nicht berücksichtigt wurden.

Am Ende des 19. Jahrhunderts begann eine Vereinigung der experimentellen Hydraulik und der theoretischen Hydrodynamik. William Froude (1810–1879) konstruierte als Erster Wassertanks, in denen er verschiedene Schiffsformen ausprobierte, um das beste Verhältnis von Länge, Breite und Tiefe des Rumpfs zu finden. Schließlich wies er nach, dass der Wasserwiderstand am geringsten ist, wenn ein Schiff möglichst lang gebaut wird. Seine Erkenntnisse fasste er in der Froude-Formel zusammen. Osborne Reynolds (1842–1912) begründete mit Rohrstömungsexperimenten 1883 die Turbulenzforschung und zeigte die Wichtigkeit der nach ihm benannten Reynolds Zahl.

Eine Theorie für viskose Flüssigkeiten war zwar vorhanden aber noch unerforscht bis Navier (1785–1836) und Stokes (1819–1903) den newtonschen Viskositäts-Term mit den Bewegungsgleichungen verbinden konnten. Die daraus entstandenen *Navier-Stokes* Gleichungen waren aber zu schwierig, um sie für beliebige Fluids analytisch lösen zu können. 1904 gelang Ludwig Prandtl (1875–1953) der Durchbruch, indem er zeigte, dass Fluids

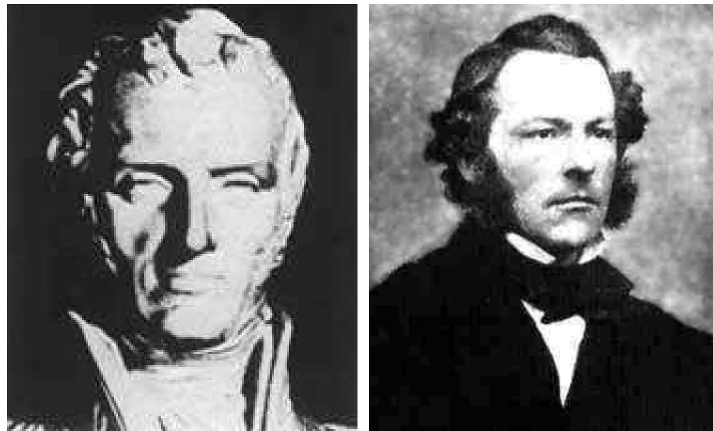

Abbildung 1.3: Claude Navier 1785-1836, Georg Stokes 1819-1903

mit geringer Viskosität wie Wasser oder Rauch in dünne Schichten zerlegt werden können. Er entwickelte seine Grenzschichttheorie. Dabei wird die Reibung in der wandnahen Schicht, der so genannten Grenzschicht, berücksichtigt, in der das Fluid durch die Haftung am Rand wesentlich langsamer als im Inneren der Strömung fließt und somit dort die Trägheitskräfte kleiner sind. Bei der Beschreibung von Strömungen mit den *Navier-Stokes* Gleichungen wird die Reibung sogar im gesamten Strömungsgebiet berücksichtigt, so dass sich damit auch zähere Fluids gut modellieren lassen. Analytisch sind diese Gleichungen aber nur noch unter stark vereinfachten Bedingungen lösbar.

1.3 Numerische Simulation

Um die Gesetzmäßigkeiten der Natur zu erforschen, musste man die Realität genau beschreiben. Dazu gab es zwei wesentliche Ansätze. Zum ersten, gibt es den *praktischen Ansatz*. Hier wird versucht, mit Experimenten und Beobachtungen, die physikalischen Gesetzmäßigkeiten zu ermitteln. Beim *theoretische Ansatz* werden die Gesetzmäßigkeiten der Natur mit Beziehungen zwischen mathematischen Größen beschrieben. Dazu wird meistens die Differential- und Integralrechnung verwendet.
Neben diesen beiden Ansätzen ist ein dritter Ansatz die *numerische Simulation*. Sie verbindet die beiden traditionellen Wege. Die *numerische Simulation* technisch-physikalischer Abläufe gewinnt in jüngster Zeit immer mehr an Bedeutung. So lassen sich mit ihrer Hilfe etwa im Automobil- und Flugzeugbau, in der elektrotechnischen und chemischen Industrie und sogar in der theoretischen Physik die Zahl teurer Experimente mit aufwändigen Versuchsaufbauten verringern. Darüber hinaus erlaubt die numerische Simulation, auch in solchen Fällen Aussagen zu treffen, in denen dies auf Grund technischer Unzulänglichkeiten sonst nicht möglich wäre oder sich ein praktisches Experiment von vornherein verbietet. Konkrete Anwendungsbeispiele der numerischen Simulation sind Strömungsvorgänge wie die Umströmung eines Flugzeugflügels, der Luftwiderstand eines Autos, Verbren-

nungsvorgänge in Wärmekraftwerken, Beschichtungsprozesse, Schmelzprozesse und Kristallwachstum bei der Halbleiterfertigung, Wasser- und Schadstofftransporte in porösen Medien, Klimamodellierung sowie eine Fülle von Anwendungen im Bereich des Umweltschutzes.

Die Entwicklung der numerischen Strömungsmechanik (*computational fluid dynamics*, CFD) ist eng mit der Entwicklung des Computers verbunden. Erste Versuche einer numerischen Lösung von partiellen Differentialgleichungen mit Bezug zu einem strömungsmechanischen Problem gab es schon 1933 [33]. Doch eine ernst zunehmende Alternative wurde sie erst mit der Entwicklung elektronischer Rechenmaschinen. Erste Arbeiten in den vierziger Jahren des 20. Jahrhunderts waren durch die geringe Rechenleistung der damaligen Rechner sehr eingeschränkt. Mitte der fünfziger Jahre wurde ein erstes effizientes Verfahren (ADI) zur Lösung parabolischer und elliptischer Probleme von *Peaceman & Rachford* [25] vorgestellt. Es begann eine stürmische Entwicklung zahlreicher Verfahren zur numerischen Lösung unterschiedlicher Strömungsmodelle. Die Entwicklung numerischer Verfahren für viskose Strömungen wurde stark von dem *Marker-and-Cell* Verfahren (MAC) von *Harlow* [6] beeinflusst. Diese Veröffentlichung ist Vorlage für das in dieser Arbeit verwendete Verfahren. Dabei handelt es sich um ein einfaches Finite Differenzen Verfahren mit einer expliziten Zeitdiskretisierung erster Ordnung. Diese Methode ist trotz ihres Alters erstaunlich flexibel und leistungsstark. Weiterhin ist sie leicht zu vermitteln. Einen schematischen Überblick über die einzelnen Teilschritte einer numerischen Simulation zeigt Abbildung 1.4. Die numerische Lösung der Strömungsgleichungen wird in Kapitel 3 genauer vorgestellt.

Realität
Physikalische Abläufe

Experimente
beobachten

mathematisches Modell z.B.
Systeme vom Differentialgleichungen
Systeme von Integralgleichungen
Eigenwertprobleme u.a.
i.a. zeitabhängig nichtlinear
kontinuierlich

Verbesserung des
kontinuierlichen
math. Modells,
Parameter verändern
andere Gleichung wählen

Diskretisierung: Gitter
endliche dimensionale Räume, Finite
Differenzen, –Elemente, –Volumen

Verbesserung der
Diskretisierung
h,p,r – adaptiv

Behandlung der Zeitabhängigkeit

Parallelisierung

Behandlung der Nichtlinearitäten

Zeitschritte

Iteration

Schnelles Lösen eines linearen
Gleichungssytems
Iterationsverfahren

Iteration

Beurteilen der diskreten Lösungen
selbst / Fehlerschätzer

Beurteilung der diskreten Lösungen
im Bezug auf das kontinuierliche
Modell und dessen Parameter

Ziel: numerische Experimente
neue Erkenntnisse z.B. durch
Abändern der Geometrie oder
der Material–Parameter

Abbildung 1.4: Typischer Ablauf in der numerischen Simulation (nach [16])

Kapitel 2 Grundgleichungen der Strömungsmechanik

In diesem Kapitel werden die für die Rauch- bzw. Strömungssimulation wichtigen Formeln, die *Navier-Stokes* Gleichungen, hergeleitet. Was ein Fluid ist, wurde im vorherigen Kapitel erläutert. Jetzt muss untersucht werden, wie man das Verhalten von Fluids beschreiben kann. Da die Behandlung der Strömungssimulation sehr umfangreich ist, soll das Thema nur so weit vertieft werden, wie es für die Implentation eines Animationssystems zur Rauchsimulation notwendig ist.

Zur Berechnung der Strömungen in einem Fluid müssen die drei Geschwindigkeitskomponenten u, v, w des Vektors $\vec{\mathbf{v}}$, die Dichte ρ, der Druck p und die Temperatur T der Strömung in Abhängigkeit von den drei kartesischen Koordinaten x, y, z ermittelt werden.
Für die weitere Betrachtung werden Fluids als Kontinuum betrachtet. Das heißt dass die molekulare Struktur nicht beachtet wird und keine Rolle spielt. Diese Annahme wird häufig bei der Beschreibung von Wärme- oder Massetransporten gemacht. Um den Vorteil dieses Vorgehens zu erkennen, muß man sich die mögliche alternative Beschreibungsmöglichkeit ansehen. Diese soll als nächstes kurz aufgezeigt werden.

2.1 Das Kontinuum

Fluids bestehen aus Molekülen. Wenn man die Startposition und die Geschwindigkeit jedes Moleküls kennt, könnte man spätere Positionen und Geschwindigkeit mit den Bewegungsgesetzen von Newton (z.B. $\vec{F} = ma$) berechnen. Die Schwierigkeit bei dieser Methode besteht darin, dass die Anzahl der Moleküle, in einem zu betrachtendem Fluid, eine genaue Berechnung unmöglicht macht. Zum Beispiel:

$$1 \text{ cm}^3 \text{ Wasser} \rightarrow 3.3 \times 10^{22} \text{ Moleküle} \rightarrow 10 \text{ Millionen Jahre}$$

Das bedeutet, dass ein Computer mit einer Rechenleistung von 10 MFLOPS für alle Multiplikationen 10 Millionen Jahre benötigen würde. Moleküle in einem Fluid kollidieren im

Durchschnitt alle 10^{-12} Sekunden. Um nun eine Sekunde realen Verhaltens zu simulieren, würde man $10^{12} \times 10$ Millionen Jahre benötigen. Dies ist natürlich eine absurd lange Rechenzeit.

Auf diese Weise kann das Verhalten der Moleküle also nicht bestimmt werden. Eine Alternative ist die **Kontinuum Hypothese**. Um das makroskopische Verhalten eines Materials vorher zusagen, ist es nicht notwendig dieses auf molekularer Ebene zu beschreiben. Man muss also nicht jede einzelne Position eines Moleküls genau wissen. Es reicht die Masseverteilung, die durch ein Dichteprofil $\rho(\mathrm{r})$ der Moleküle in bestimmten Regionen (r) beschrieben wird, zu kennen.

$$\rho(\mathrm{r}) = \lim_{V \to 0} \left\{ \frac{1}{V} \sum_i m_i \right\} \tag{1}$$

Hier ist m_i die Masse von Molekül i. Die Masse wird über alle Moleküle innerhalb der Oberfläche A betrachtet. V ist das Volumen. In der Fluid-Mechanik setzen wir nun die Kontinuum Hypothese voraus. Grundlage ist hier, dass das Limit in Formel 1 konvergiert, bevor das Volumen Molekülgröße erreicht hat. Man muss nicht die Energie jedes Moleküls kennen, man braucht nur die Energie innerhalb eines Einheitsvolumen in Abhängigkeit der Position zu kennen.

Kontinuum Hypothese Eine Region kann in (infinitesimal) kleine Volumenelemente zerlegt werden die

1. klein genug sind, um als konstant zu gelten (die räumlichen Änderungen von $\rho, T, \mathbf{v}, p$ kann vernachlässigt werden)
2. groß genug sind, um genug Moleküle im statischen Sinn zu enthalten

Wir nehmen also an, dass es ein dV gibt, das die genannten Bedingungen erfüllt. Materialien, die diese Eigenschaft haben, verhalten sich wie ein **Kontinuum**. Die Kontinuum Hypothese funktioniert gut, wenn die Maße des Systems im Vergleich zur Molekülgröße groß sind. Eine Ausnahme, bei der die Hypothese nicht funktioniert, ist die Knudsen-Diffusion. Die Knudsen-Diffusion tritt in engen Poren auf, bei denen die mittlere freie Weglänge größer ist als der Porenradius. Die Moleküle stoßen häufiger mit der Porenwand zusammen als mit anderen Molekülen. Dieses Verhalten wird in den weiteren Betrachtungen vernachlässigt.

Zum Herleiten der *Navier-Stokes* Gleichungen setzten wir voraus, dass das Gas homogen ist und dass es einem Kontinuum entspricht. Es gelten die Erhaltungssätze für Masse, Impuls und Energie. Dabei wird das besprochene infinitesimal kleine Volumenelement, in Form eines Quaders (Abb. 2.1), betrachtet. Die linke obere Ecke befindet sich an beliebiger Stelle mit den Koordinaten (x, y, z) im Strömungsfeld. Die Kanten sind jeweils parallel zu den entsprechenden Koordinatenachsen ausgerichtet. Das Volumenelement soll raumfest sein, es darf sich nicht mit der Strömung bewegen.

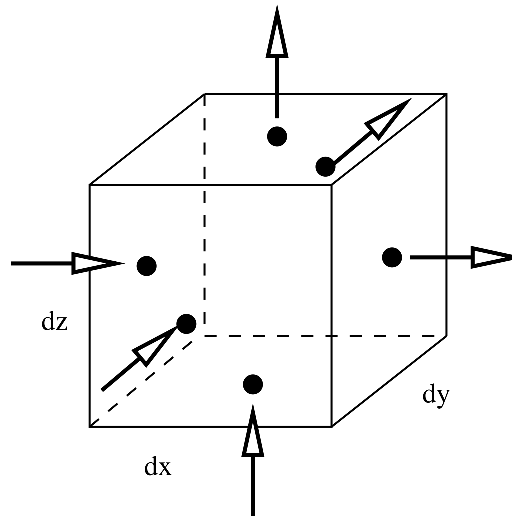

Abbildung 2.1: Ein- und ausströmende Masseströme

2.2 Erhaltung der Masse (Kontinuitätsgleichung)

Der erste Teil der *Navier-Stokes* Gleichungen beinhaltet die Bedingung, dass die Masse erhalten bleiben soll. Als Gleichung ausgedrückt soll gelten:

Die Zeitliche Änderung der Masse im Volumenelement =
$\sum$ *der einströmenden Masssseströme in das Volumenelement -*
$\sum$ *der ausströmenden Masseströme aus dem Volumenelement.*

Abbildung 2.1 zeigt ein Volumenelement aus einem Kontinuum. Die Kantenlängen des Volumenelements sind dx, dy und dz. Durch die linke Oberfläche des Volumenelements mit der Fläche $dy \cdot dz$ tritt der Massestrom $\rho \cdot u \cdot dy \cdot dz$ ein. Die Größe $\rho \cdot u$ ändert ihren Wert von der Stelle x zur Stelle $x + dx$ in x-Richtung um $(\partial(\rho \cdot u)/\partial x) \cdot dx$, so dass sich der durch die rechte Oberfläche $dy \cdot dz$ des Volumenelements austretende Massestrom mit dem Ausdruck

$$\left(\rho \cdot u + \frac{\partial(\rho \cdot u)}{\partial x} \cdot dx\right) \cdot dy \cdot dz$$

angeben lässt. Für die y- und z-Richtung gelten analoge Größen auf den entsprechenden Oberflächen $dx \cdot dz$ und $dx \cdot dy$

Die zeitliche Änderung der Masse, innerhalb des betrachteten Volumenelements, entspricht, nach der Erhaltung der Masse, der Differenz aus eintretenden und austretenden Masseströmen. Der Term

$$\frac{\partial(\rho \cdot dx \cdot dy \cdot dz)}{\partial t} = \frac{\partial \rho}{\partial t} \cdot dx \cdot dy \cdot dz$$

entspricht dem mathematischen Ausdruck für die zeitliche Änderung der Masse im Volumenelement. Gemäß der vorigen Überlegung gilt

$$\frac{\partial \rho}{\partial t} \cdot dx \cdot dy \cdot dz = \left(\rho \cdot u - (\rho \cdot u + \frac{\partial(\rho \cdot u)}{\partial x} \cdot dx)\right) \cdot dy \cdot dz + \left(\rho \cdot v - (\rho \cdot v + \frac{\partial(\rho \cdot v)}{\partial y} \cdot dy)\right) \cdot dx \cdot dz + \left(\rho \cdot w - (\rho \cdot w + \frac{\partial(\rho \cdot w)}{\partial z} \cdot dz)\right) \cdot dx \cdot dy.$$

Damit erhält man die Kontinuitätsgleichung:

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho \cdot u)}{\partial x} + \frac{\partial(\rho \cdot v)}{\partial y} + \frac{\partial(\rho \cdot w)}{\partial z} = 0.$$

Da wir nur inkompressible Fluids betrachten, vereinfacht sich die Gleichung zu

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0,$$

da deren Dichte ρ unabhängig von Ort und Zeit ist.
In Koordinaten freier Schreibweise erhält man

$$\nabla \cdot \vec{\mathbf{v}} = 0.$$

∇ ist der *Nabla*-Operator. Die Divergenz des Vektorfeldes muss also null sein. Das heißt, es dürfen keine Quellen oder Senken entstehen. Damit wurde eine Bedingung der *Navier-Stokes* Gleichungen, die Divergenzfreiheit, gefunden.

2.3 Erhaltung des Impulses

Der zweite Teil der *Navier-Stokes* Gleichungen beschreibt die Erhaltung des Impulses. Es gilt folgende Gleichung:

Die Zeitliche Änderung des Impulses im Volumenelement =
$\sum$ *der eintretenden Impulsströme in das Volumenelement -*
$\sum$ *der ausströmenden Implusströme aus dem Volumenelement +*
$\sum$ *der auf das Volumenelement wirkenden Scherkräfte, Normalspannungen +*
$\sum$ *der auf die Masse des Volumenelements wirkende Kräfte.*

Der Impuls eines Körpers ist das Produkt seiner Masse mit der Geschwindigkeit. Wenn es sich dabei um ein Fluid handelt, kann die Geschwindigkeit des Fluids vom Ort abhängen. Es wird jetzt wieder das Volumenelement aus Abbildung 2.1 betrachtet. Wie bei der Kontinuitätsgleichung wird nun die zeitliche Änderung des Impulses hergeleitet. Da der

Impuls das Produkt aus Masse und Geschwindigkeit ist, ergibt sich der Impuls innerhalb des Volumens zu $\rho \cdot dx \cdot dy \cdot dz \cdot \vec{\mathbf{v}}$. Die zeitliche Änderung beschreibt der Ausdruck

$$\frac{\partial(\rho \cdot dx \cdot dy \cdot dz \cdot \vec{\mathbf{v}})}{\partial t} = \frac{\partial(\rho \cdot \vec{\mathbf{v}})}{\partial t} \cdot dx \cdot dy \cdot dz.$$

Im weiteren wird nun nur eine Komponente des Impulsvektors $\rho \cdot dx \cdot dy \cdot dz \cdot \vec{\mathbf{v}}$ betrachtet, und zwar die Komponente, die in x-Richtung zeigt. Die zeitliche Änderung ist dann

$$\frac{\partial(\rho \cdot dx \cdot dy \cdot dz \cdot u)}{\partial t} = \frac{\partial(\rho \cdot u)}{\partial t} \cdot dx \cdot dy \cdot dz.$$

Es muss jetzt geklärt werden, wodurch sich der Impuls innhalb des betrachteten Volumenelements ändert. Ähnlich wie bei der Betrachtung der Masseströme tritt pro Zeiteinheit durch die Oberfläche des Volumenelements ein Impuls in das Volumen ein bzw. aus. Bei der Herleitung der Kontinuitätsgleichung wurde die Größe ρ (Masse pro Volumen) verwendet. Jetzt wird $(\rho \cdot u)$ (Impuls pro Volumen) verwendet. Damit können analog zu der Kontinuitätsgleichung, die ein- und ausströmenden Impulsströme angegeben werden.

Es wird in Abbildung 2.1 wieder die linke Oberfläche in x-Richtung betrachtet. Es tritt demnach durch die Oberfläche $dy \cdot dz$ der Impulsstrom

$$(\rho \cdot u) \cdot u \cdot dy \cdot dz = \rho \cdot u \cdot u \cdot dy \cdot dz$$

ein. Die Größe $\rho \cdot u \cdot u$ ändert ihren Wert in x-Richtung um

$$\frac{\partial(\rho \cdot u \cdot u)}{\partial x} \cdot dx$$

so dass sich der auf der rechten Oberfläche $dy \cdot dz$ des Volumenelements austretende Impulsstrom mit dem Ausdruck

$$\left(\rho \cdot u \cdot u + \frac{\partial(\rho \cdot u \cdot u)}{\partial x} \cdot dx\right) \cdot dy \cdot dz$$

beschreiben lässt.

Der in der x-Richtung wirkende Impuls $\rho \cdot u$ tritt auch über die verbleibenden Oberflächen $dx \cdot dz$ und $dx \cdot dy$ ein bzw. aus. Dort strömt er mit den Geschwindigkeitskomponenten v bzw. w durch die Oberflächen.
Für die y- und z-Richtung gelten analoge Überlegungen, so dass sich insgesamt auf jeder Oberfläche drei Impulsströmungen angeben lassen.

Die ein- und ausströmenden Impulsströme sind nicht die alleinige Ursache für die zeitliche Änderung des Impulses innerhalb eines Volumenelements. Der Impuls wird zusätzlich durch die am Volumen angreifenden Kräfte geändert. Zu diesen Kräften gehören

- *Oberflächenkräfte*: Diese sind z.B. Druck und innere Reibung. Ihre Größen ändern sich in x-, y- und z-Richtung.
- *Volumenkräfte*: Das sind die Kräfte, die auf die im Volumen befindlichen Massen wirken. Zu ihnen gehören z.B. die Schwerkraft, Coriolis-Kraft sowie magnetische Kräfte. Diese externen Kräfte sollen im Weiteren mit $\vec{f}$ bezeichnet werden.

Mit diesen Voraussetzungen und unter Beachtung der Kontinuitätsgleichung erhält man folgende Gleichungen für ein inkompressibles Fluid

$$\rho \cdot \left(\frac{\partial u}{\partial t} + u \cdot \frac{\partial u}{\partial x} + v \cdot \frac{\partial u}{\partial y} + w \cdot \frac{\partial u}{\partial w} \right) = f_x - \frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

$$\rho \cdot \left(\frac{\partial v}{\partial t} + u \cdot \frac{\partial v}{\partial x} + v \cdot \frac{\partial v}{\partial y} + w \cdot \frac{\partial v}{\partial w} \right) = f_y - \frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right)$$

$$\rho \cdot \left(\frac{\partial w}{\partial t} + u \cdot \frac{\partial w}{\partial x} + v \cdot \frac{\partial w}{\partial y} + w \cdot \frac{\partial w}{\partial w} \right) = f_z - \frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right).$$

ν ist die Viskosität und $\vec{\mathbf{f}}$ die externen Kräfte.
Die Koord. freie Schreibweise ergibt:

$$\rho \cdot \left(\frac{\partial \vec{\mathbf{v}}}{\partial t} + (\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}} \right) = \vec{\mathbf{f}} - \nabla p + \nu \nabla^2 \vec{\mathbf{v}}.$$

∇^2 ist die Abkürzung für $\nabla \cdot \nabla$ und entspricht dem Laplace-Operator der mit

$$\nabla^2 \vec{\mathbf{v}} = \frac{\partial^2 \vec{\mathbf{v}}}{\partial x^2} + \frac{\partial^2 \vec{\mathbf{v}}}{\partial y^2} + \frac{\partial^2 \vec{\mathbf{v}}}{\partial z^2}$$

definiert ist.
Diese Gleichung zusammen mit der Kontinuitätsgleichung

$$\nabla \cdot \vec{\mathbf{v}} = 0$$

bilden die *Navier-Stokes* Gleichungen. Diese ergeben ein Gleichungssystem, bestehend aus **vier** skalaren, partiellen, nichtlinearen Differentialgleichungen zweiter Ordnung, für die **vier** Unbekannte u,v,w und p, welches für vorgegebene Anfangs- und Randbedingungen gelöst werden muss. Auf Lösungsmethoden wird im nächsten Kapitel eingegangen.

Dieser kurze Einblick in die Strömungslehre sollte nur einen Eindruck über die notwendigen Betrachtungen in der Strömungsmechanik liefern. Der implementierte und in Kapitel 4 beschriebene Algorithmus zur Fluid Simulation geht von inkompressiblen Fluids aus. Die kompressiblen Fluids wurden deshalb hier nicht weiter betrachtet. Das nächste Kapitel zeigt, wie die hergeleiteten Gleichungen diskretisiert und mit numerischen Verfahren gelöst werden können.

Kapitel 3
Numerische Lösung der Strömungsgleichungen

Dieses Kapitel beschreibt, wie die inkompressiblen *Navier-Stokes* Gleichungen numerisch gelöst werden können. Eine analytische Lösung ist in den meisten Fällen nicht oder nur in vereinfachten, idealisierten Fällen möglich. Mit numerischen Lösungsmethoden wird daher versucht, Strömungsprobleme unter Einhaltung von Rand- und Anfangsbedingungen möglichst genau näherungsweise zu lösen, ohne dass man gravierende Vereinfachungen oder Annahmen treffen muss. In der Regel können diese Methoden für komplexe und beliebige Geometrien angewandt werden. Da Rechenleistung mittlerweile relativ preiswert ist, können immer öfter aufwändige numerische Verfahren eingesetzt werden. Ein großer Nachteil numerischer Verfahren ist allerdings, dass mit ihnen die Abhängigkeit des Ergebnisses von einer eingehenden Größe nur mit aufeinander folgenden Rechnungen bestimmt werden kann, wobei von Rechnung zu Rechnung die eingehenden Größen passend variiert werden müssen.

Betrachten wir die *Navier-Stokes* Gleichungen in der Koord. freien Schreibweise, wie sie im *Stable-Fluid* Algorithmus verwendet werden.

$$\nabla \cdot \vec{\mathbf{v}} = 0 \tag{1}$$

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

Gleichung 1 ist die in Kapitel 2 hergeleitete Kontinuitätsgleichung und 2 die Impulsgleichung. Hier ist ν die kinematische Viskosität des Fluids. ρ ist die Dichte und $\vec{\mathbf{f}}$ die externen Kräfte, die auf das Fluid einwirken. " $\cdot$ " stellt hier das Skalarprodukt zwischen zwei Vektoren dar. ∇ ist der Gradient, also die räumliche erste partielle Ableitung, genauer $\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}$. ∇^2 ist der Laplace-Operator und Abkürzung für $\nabla \cdot \nabla$. Der Teil $-(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}}$ in Gleichung 2 ist der so genannte advektive Term und sorgt dafür, dass durch das doppelte Auftreten von $\vec{\mathbf{v}}$, die Gleichung nicht-linear wird. Es müssen also numerische Verfahren zur Lösung benutzt werden.

3.1 Diskretisierung

Um die *Navier-Stokes* Gleichungen numerisch lösen zu können, müssen diese diskretisiert werden. Die Diskretisierung ist der Übergang von einem kontinuierlichen Problem zu einem, das nur in endlich vielen Punkten betrachtet wird. Hier gibt es im Wesentlichen drei Verfahren:

- Finite Elemente Methode (FEM)
- Finite Volumen Methode (FVM)
- Finite Differenzen Methode (FDM)

Die FVM erfüllt die diskretisierten Erhaltungssätze über jedes Volumenelement im Strömungsfeld während bei der FEM der numerische Fehler mit geeigneten Ansatzfunktionen und der Formulierung eines Variationsproblemes in jedem Volumenelement minimiert wird. Die FDM diskretisiert das Strömungsfeld in orthogonale Gitter und ersetzt die Differentialquotienten der Grundgleichungen durch die entsprechenden Differenzenquotienen.

Die Methode der Finiten Elemente wurde ursprünglich in der Festkörpermechanik zur Berechnung von Strukturproblemen entwickelt. Ihre Anwendung bei Strömungsproblemen wurde in Zusammenhang mit der erforderlichen Diskretisierung des Integrationsfeldes mit unstrukturierten Netzen bei komplexen Konfigurationen wie einem Flugzeug oder einem Kraftfahrzeug attraktiv.

Ähnlich wie bei der FDM wird auch bei der FVM das Integrationsgebiet mit Hilfe eines numerischen Netzes diskretisiert. Im Unterschied zum FDM werden hier jedoch nicht die Differentialquotienten in den Grundgleichungen durch Differenzenquotienten approximiert. Bei der FVM wird die Erhaltungsgleichung über das jeweilige Volumenelement in integraler Form erfüllt. Die Grundgleichungen werden also in integraler Form diskretisiert.

Für die Implementation des Stable Fluids Algorithmus aus Kapitel 4 wurde das Finite Differenzen Verfahren benutzt. Es lässt sich leicht implementieren und liefert eine ausreichende Genauigkeit. Im weiteren wird dieses Verfahren genauer vorgestellt.

3.2 Finite Differenzen Methode (FDM)

Die FDM geht im ersten Schritt von einer Diskretisierung des Integrationsbereiches aus. In einem zweiten Schritt werden die partiellen Differentialgleichungen in diskreten Gitterpunkten in Differenzengleichungen überführt. Dieses setzt ein orthogonales Rechennetz voraus. Die partiellen Ableitungen in den Gleichungen werden also durch die Differenzengleichungen ersetzt.

3.2.1 Zeitdiskretisierung

Abbildung 3.1 zeigt einen Zeitstrahl, beginnend bei $t = 0$, der in diskrete Gitterpunkte unterteilt worden ist, um an diesen Punkten den Funktionswert näherungsweise zu be-

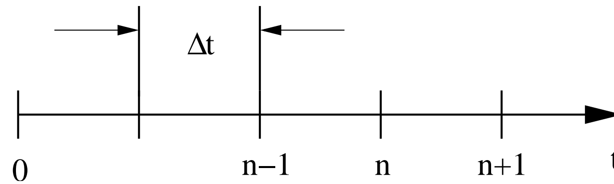

Abbildung 3.1: Zeitdiskretisierung

stimmen. Die kontinuierliche Zeit t wird also in äquidistante Zeitintervalle Δt unterteilt, an deren Intervallgrenzen die gesuchten Funktionswerte zu bestimmen sind. Ein beliebiger diskreter Zeitpunkt t^n auf der Zeitachse ist dann bestimmt durch:

$$t^n = n \cdot \Delta t \text{ mit } n = 0, 1, 2, 3, ...$$

n ist der Zählindex für die Zeit t. Δt ist das vorgegebene Zeitintervall und wird als Zeitschritt bezeichnet. t^n ist dann der n-te diskrete Zeitpunkt, an dem der Funktionswert $u(t^n)$ berechnet werden soll. Abkürzend wird dafür $u(t^n) = u^n$ geschrieben. Die Bezeichnung u^n steht für den aktuellen Funktionswert zum Zeitpunkt t^n. Die Terme u^{n-1}, u^{n-2} usw. stehen für bekannte Funktionswerte aus früheren, vergangenen Zeitpunkten. Dagegen ist der Funktionswert u^{n+1} unbekannt und muss für den Zeitpunkt t^{n+1} bestimmt werden.

Nachdem der Integrationsbereich diskretisiert wurde, kann die Approximation der Differentialquotienten durch Differenzenquotienten erfolgen. Die Approximation kann mit einer Taylor-Entwicklung in der Zeit t für einen Funktionswert $u(t_0 + \Delta t)$ erfolgen. Es gilt:

$$u(t_0 + \Delta t) = u(t_0) + \Delta t \cdot \frac{\partial u}{\partial t}|_{t=t_0} + \frac{\Delta t^2}{2!} \cdot \frac{\partial^2 u}{\partial t^2}|_{t=t_0} + \cdots$$

$$u(t_0 + \Delta t) = u(t_0) + \Delta t \cdot \frac{\partial u}{\partial t}|_{t=t_0} + O(\Delta t^2) \tag{3}$$

Der Ausdruck $O(\Delta t^2)$ besagt, wenn man die Taylor-Entwicklung nach dem dritten Summanden abbricht, erhält man einen Fehler der Ordnung 2. Löst man nun Gleichung 3 nach dem Differentialquotienten auf, den man approximieren will, erhält man:

$$\frac{\partial u}{\partial t}|_{t=t_0} = \frac{u(t_0 + \Delta t) - u(t_0)}{\Delta t} - O(\Delta t) \tag{4}$$

Gleichung 4 kann man nun für einen beliebigen Zeitpunkt t^n aufschreiben und erhält die *Vorwärtsdifferenz.*

$$\frac{\partial u(t^n)}{\partial t} = \frac{u(t^{n+1}) - u(t^n)}{\Delta t} - O(\Delta t) = \frac{\partial u^n}{\partial t} = \frac{u^{n+1} - u^n}{\Delta t} - O(\Delta t) \tag{5}$$

Die Gleichung 5 wird Vorwärts-Differenzenquotient genannt, da die Ableitung an der Stelle $t = t^n$ mit einem Wert u^{n+1} an einem zukünftigen Zeitpunkt t^{n+1} approximiert

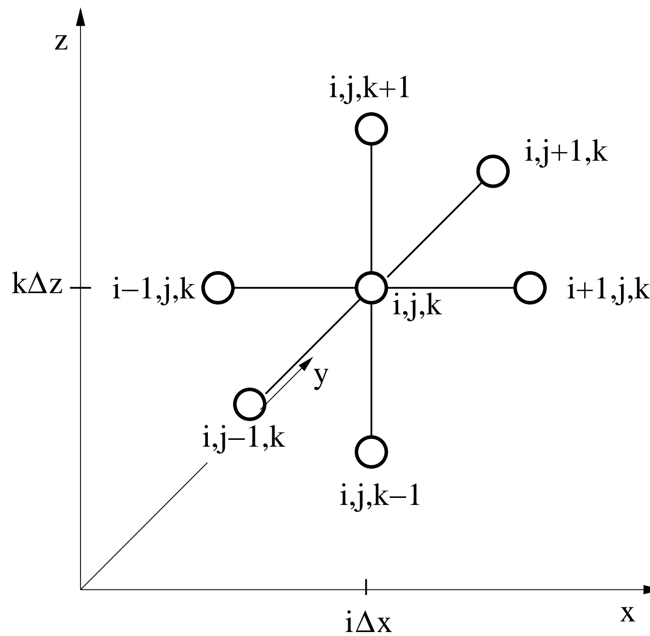

Abbildung 3.2: Räumliche Diskretisierung (nach [23])

wird. Bei bekannter Ableitung an der Stelle $t = t^n$ kann man Gleichung 5 nach dem unbekannten Wert u^{n+1} auflösen.

$$u^{n+1} = u^n + \Delta t \cdot \frac{\partial u^n}{\partial t} \tag{6}$$

Dieses entspricht dem so genannten *Euler* Verfahren.

3.2.2 Raumdiskretisierung

Eine Diskretisierung des Raumes erfolgt wie bei der Zeitdiskretisierung durch eine Unterteilung der kontinuierlichen Koordinaten in äquidistante Gitterpunkte. Die Abstände der Gitterpunkte, an denen die Funktionswerte gesucht sind, werden in räumlichen kartesischen Koordinaten x, y und z mit Δx, Δy sowie Δz bezeichnet. Die Zählindizes entlang der Koordinatenrichtungen werden mit i, j und k bezeichnet. Somit ergeben sich die diskreten unabhängigen Ortsvariablen zu

$$\begin{aligned} x_i &= i \cdot \Delta x \text{ mit } i = 0, 1, 2, 3, \cdots \\ y_j &= j \cdot \Delta y \text{ mit } j = 0, 1, 2, 3, \cdots \\ z_k &= k \cdot \Delta z \text{ mit } k = 0, 1, 2, 3, \cdots \end{aligned} \tag{7}$$

Abbildung 3.2 zeigt das Prinzip der Raumdiskretisierung. Die Gitterstelle i, j, k ist die aktuell betrachtete Position im Raum. Ihre Nachbarn erhält man durch Addition oder Subtraktion von 1 zu den Zählindizes. Eine Herleitung der Differenzenquotienten erfolgt

wieder durch eine Taylor-Reihe. Den Rückwärts-Differenzenquotienten zur Approximation der räumlichen Ableitung in x-Richtung erhält man durch eine Entwicklung von $u(x_0 - \Delta x, y_0, z_0)$.

$$u(x_0 - \Delta x, y_0, z_0) = u(x_0, y_0, z_0) - \Delta x \cdot \frac{\partial u}{\partial x}|_{x=x_0} + O(\Delta x^2)$$

Eine Vereinfachung der Gleichung liefert die *Rückwärtsdifferenz*

$$\frac{\partial u_{i,j,k}}{\partial x} = \frac{u_{i,j,k} - u_{i-1,j,k}}{\Delta x} - O(\Delta x)$$

Neben den Vorwärts- und Rückwärts-Differenzen gibt es noch die zentralen Differenzen zur Approximation der ersten Ableitung. Diese Differenzen werden durch Subtraktion der Taylor-Entwicklung für $u(x_0 - \Delta x, y_0, z_0)$ und $u(x_0 + \Delta x, y_0, z_0)$ gebildet. Die Glieder mit Ableitungen gradzahliger Ordnung heben sich gegenseitig auf. Man erhält also

$$u(x_0 - \Delta x, y_0, z_0) - u(x_0 + \Delta x, y_0, z_0) = 2 \cdot \Delta x \cdot \frac{\partial u}{\partial x}|_{x=x_0} + \frac{(\Delta x)^3}{3} \cdot \frac{\partial^3 u}{\partial x^3}|_{x=x_0} + \cdots \quad (8)$$

Aus Gleichung 8 erhält man durch Umstellen und Vereinfachen die *Zentrale Differenz*

$$\frac{\partial u_{i,j,k}}{\partial x} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2 \cdot \Delta x} - O(\Delta x)^2 \quad (9)$$

Hier ist der Fehler von zweiter Ordnung. Die zentralen Differenzen approximieren demzufolge genauer als die Vorwärts- und Rückwärtsdifferenzen.

Es fehlen noch die Differenzenquotienten für die zweite Ableitung. Diese erhält man durch die Addition der Taylor-Entwicklung für $u(x_0 - \Delta x, y_0, z_0)$ und $u(x_0 + \Delta x, y_0, z_0)$. Jetzt heben sich alle Ableitungen ungradzahliger Ordnung auf und man erhält den Differenzenquotienten der 2. Ableitung

$$\frac{\partial^2 u_{i,j,k}}{\partial x^2} = \frac{u_{i+1,j,k} - 2 \cdot u_{i,j,k} + u_{i-1,j,k}}{(\Delta x)^2} - O(\Delta x)^2 \quad (10)$$

Es wurden nur die Gleichungen für die x Komponente angeben. Die Gleichungen der Ableitungen für die Variablen y, z ergeben sich einfach durch Vertauschen des jeweiligen Laufindex.

3.2.2.1 Praktische Anwendung auf die *Navier-Stokes* Gleichungen

Durch Anwendung der hergeleiteten Differenzenquotienten können die *Navier-Stokes* Gleichungen diskretisiert werden. Im folgenden wird gezeigt wie die Differenzenquotienten bei

der Implementierung des *Stable Fluids* Algorithmus aus Kapitel 4 benutzt wurden. Die Divergenz $\nabla \cdot u$ wird folgendermaßen bestimmt:

$$\begin{aligned}(\nabla \cdot u)_{i,j,k} = (&u_{i+1,j,k} - u_{i-1,j,k} + \\ &u_{i,j+1,k} - u_{i,j-1,k} + \\ &u_{i,j,k+1} - u_{i,j,k-1})/2h\end{aligned} \tag{11}$$

Für den diskreten Gradienten $\nabla p = (p_x, p_y, p_z)$ wird

$$\begin{aligned}(p_x)_{i,j,k} &= (p_{i+1,j,k} - p_{i,j,k})/h, \\ (p_y)_{i,j,k} &= (p_{i,j+1,k} - p_{i,j,k})/h, \\ (p_z)_{i,j,k} &= (p_{i,j,k+1} - p_{i,j,k})/h,\end{aligned} \tag{12}$$

genutzt. Der diskrete Laplace ∇^2 ergibt sich aus einer Kombination von Gradient- und Divergenzoperator.

3.2.2.2 Poisson-Gleichung

Bei Berechnungen in der Strömungsmechanik, trifft man häufig auf Gleichungen der Form:

$$\nabla^2 \vec{u} = f$$

Diese Art von Gleichungen werden *Poisson*-Gleichung genannt. Für den 2D-Fall erhält man folgendes Aussehen:

$$\nabla^2 \vec{u} = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y)$$

Die Diskretisierung erfolgt wieder mit Finiten Differenzen über ein Gitter und man erhält für den 2D-Fall

$$4U_{i,j} - U_{i-1,j} - U_{i+1,j} - U_{i,j-1} - U_{i,j+1} = b_{i,j}$$

Hier ist U im Allgemeinen unbekannt. Um nun ein Gleichungssystem der Form:

$$A\vec{u} = \vec{b}$$

zu erhalten, in dem $\vec{u}$ und $\vec{b}$ Spaltenvektoren sind, wird eine lineare Anordnung der Unbekannten $U_{i,j}$ benötigt. Die erforderliche Anordnung erhält man leicht mit einer Durchnummerierung der Gitterzellen wie in Abbildung 3.3 am Beispiel eines 4×4 Gitters gezeigt wird.

Mit dieser Nummerierung erhält mal folgendes Gleichungssystem:

	i=1	2	3	4
4	13	14	15	16
3	9	10	11	12
2	5	6	7	8
j=1	1	2	3	4

Abbildung 3.3: Nummerierung der Unbekannten einer Poisson-Gleichung

$$
\begin{bmatrix}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
& -1 & 4 & -1 & & & -1 & & & & & & & & & \\
& & -1 & 4 & & & & -1 & & & & & & & & \\
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
& -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
& & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
& & & -1 & & & -1 & 4 & & & & -1 & & & & \\
& & & & -1 & & & & 4 & -1 & & & -1 & & & \\
& & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
& & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
& & & & & & & -1 & & & & 4 & -1 & & & -1 \\
& & & & & & & & -1 & & & & 4 & -1 & & \\
& & & & & & & & & -1 & & & -1 & 4 & -1 & \\
& & & & & & & & & & -1 & & & -1 & 4 & -1 \\
& & & & & & & & & & & -1 & & & -1 & 4
\end{bmatrix}
*
\begin{bmatrix}
u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{4,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{4,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \\ u_{4,3} \\ u_{1,4} \\ u_{2,4} \\ u_{3,4} \\ u_{4,4}
\end{bmatrix}
=
\begin{bmatrix}
b_{1,1} \\ b_{2,1} \\ b_{3,1} \\ b_{4,1} \\ b_{1,2} \\ b_{2,2} \\ b_{3,2} \\ b_{4,2} \\ b_{1,3} \\ b_{2,3} \\ b_{3,3} \\ b_{4,3} \\ b_{1,4} \\ b_{2,4} \\ b_{3,4} \\ b_{4,4}
\end{bmatrix}
$$

Wie man sieht, wird das Gleichungssystem schon bei kleinen Gittern sehr groß. Ein Lösen dieses Systems z.B. mit einem Gauß-Algorithmus ist zu aufwändig. Der Vorteil dieses System ist, dass es nur schwach besetzt ist, da vorrangig nur einige Diagonalen besetzt sind. Der Rest der Matrix ist mit Nullen belegt. Hier können numerische Verfahren für schwach besetzte Systeme zur Lösung benutzt werden. Für solche Systeme wurden effziente numerische Verfahren entwickelt. Eine ausführliche Vorstellung dieser Algorithmen findet man im Anhang B.

3.2.3 Randbedingungen

Beim Lösen der Diffentialgleichungen müssen die Bedingungen am Rand besonders behandelt werden. Die Randzellen werden zusätzlich zum Gitter hinzugefügt. In Abbildung 3.3 erkennt man, dass die Randzellen nicht mit nummeriert werden und nicht direkt zur Bildung des Gleichungssystems beitragen. Bei der Berechnung mit den Differenzenquotienten werden diese Zellen als Nachbarzellen mit einbezogen und müssen deshalb definierte

Zustände haben. Hier muss man nun unterscheiden was mit dem Fluid passiert, wenn es einen Rand erreicht. Es lassen sich die folgenden Randbedingungen herleiten.

- Haftbedingung
- Rutschbedingung
- Ausströmbedingung
- Einströmbedinung

Die Haft- und Rutschbedingungen müssen an Objekten, die sich innerhalb des Strömungsgebietes befinden, ebenfalls betrachtet werden.

3.2.3.1 Haftbedingung

Das Fluid verläßt den Rand nicht, sondern haftet an der Wand. Die Geschwindigkeiten sollen also am Rand Null sein. Für den 2D-Fall müssen die Werte am Rand also folgendermaßen gesetzt werden

$$\begin{aligned} u_{0,j} = 0, \quad u_{i_{max},j} = 0, \quad j = 1, \ldots, j_{max}, \\ v_{i,0} = 0, \quad v_{i,j_{max}} = 0, \quad i = 1, \ldots, i_{max} \end{aligned}$$

Weiterhin sind an den senkrechten Wänden keine v-Werte und an den waagerechten keine u-Werte vorhanden. Diese werden durch Mittelung auf null gesetzt.

$$\begin{aligned} v_{0,j} = -v_{1,j}, \quad v_{i_{max}+1,j} = -v_{i_{max},j}, \quad j = 1, \ldots, j_{max}, \\ u_{i,0} = -u_{i,1}, \quad u_{i,j_{max}+1} = -u_{i,j_{max}} \quad i = 1, \ldots, i_{max} \end{aligned}$$

3.2.3.2 Rutschbedingung

Hier dringt ebenfalls kein Fluid durch die Wand. Es existieren im Gegensatz zu der Haftbedingung keine Reibungsverluste. Die Geschwindigkeiten senkrecht zu einer Wand sind hier null. Es gelten also die gleichen Bedingungen wie bei der Haftbedingung.

$$\begin{aligned} u_{0,j} = 0, \quad u_{i_{max},j} = 0, \quad j = 1, \ldots, j_{max}, \\ v_{i,0} = 0, \quad v_{i,j_{max}} = 0, \quad i = 1, \ldots, i_{max} \end{aligned}$$

Für die nicht senkrecht zur Wand liegenden Geschwindigkeiten gilt

$$\begin{aligned} v_{0,j} = v_{1,j}, \quad v_{i_{max}+1,j} = v_{i_{max},j}, \quad j = 1, \ldots, j_{max}, \\ u_{i,0} = u_{i,1}, \quad u_{i,j_{max}+1} = u_{i,j_{max}} \quad i = 1, \ldots, i_{max} \end{aligned}$$

3.2.3.3 Ausströmbedingung

Bei der Ausströmbedingung ändern sich die Geschwindigkeiten am Rand nicht. Die Randzellen werden also auf die Werte der Nachbarzellen gesetzt.

$$
\begin{aligned}
u_{0,j} &= u_{1,j}, \quad u_{i_{max},j} = u_{i_{max}-1,j}, \quad j = 1, \ldots, j_{max}, \\
v_{0,j} &= v_{1,j}, \quad v_{i_{max}+1,j} = v_{i_{max},j}, \quad j = 1, \ldots, j_{max},
\end{aligned}
$$

$$
\begin{aligned}
u_{i,0} &= u_{i,1}, \quad u_{i,j_{max}+1} = u_{i,j_{max}}, \quad i = 1, \ldots, j_{max}, \\
v_{i,0} &= v_{i,1}, \quad v_{i,j_{max}} = v_{i,j_{max}-1}, \quad i = 1, \ldots, j_{max},
\end{aligned}
$$

3.2.3.4 Einströmbedingung

Hier werden die Geschwindigkeiten explizit am Rand gesetzt. Es können also Quellen gesetzt werden, die bestimmte Geschwindigkeiten vorgeben.

3.3 Numerische Stabilität

Numerische Lösungsverfahren für partielle Differentialgleichungen sind prinzipiell von zwei verschiedenen Fehlerquellen beeinflusst:

- Rundungsfehler ε_R: Der Rundungsfehler entsteht im Rechner selbst, da Gleitkommazahlen nur mit endlicher Genauigkeit abgespeichert werden können. Zum Beispiel wird der Bruch $\frac{1}{3}$ bei einer Zahlendarstellung im Rechner nach einer endlichen Anzahl von Ziffern 3 nach dem Komma abgebrochen. Die Differenz dieser Zahl zum exakten Wert $\frac{1}{3}$ ergibt den Rundungsfehler ε_R.
- Diskretisierungsfehler ε_D: Die Differenz zwischen der exakten analytischen Lösung einer Differentialgleichung und der rundungsfehlerfreien numerischen Lösung der zugehörigen Differenzengleichung wird als Diskretisierungsfehler bezeichnet. Er entsteht folglich nicht im Rechner, sondern dadurch, dass bei einer Taylor-Entwicklung nach einer endlichen Anzahl von Summengliedern abgebrochen wird.

Ein numerisches Verfahren wird als stabil bezeichnet, wenn ein vorhandener Fehler ε bei der Berechnung der gesuchten Werte zum Zeitpunkt t^{n+1} aus einem Zeitpunkt t^n bekannten Werten nicht anwächst. Für Stabilität muss folglich folgendes gelten

$$\frac{\varepsilon^{n+1}}{\varepsilon^n} \leqslant 1.$$

Wenn bei der Auswahl der Zeitschrittweite Δt in Kombination mit der Raumschrittweite h, bestimmte Bedingen verletzt werden, stellen sich numerische Instabilitäten ein.
Diese sind die *Courant-Friedrichs-Levy* (CFL) Bedingungen welche wichtige Stabilitätskrierien in der numerischen Simulation darstellen. Sie begrenzen die Größe eines Zeitschrittes Δt während der Simulation. Im konkreten Fall der Fluid Simulation bedeutet das, dass kein Partikel des Fluids in der Zeit Δt mehr als eine Gitterweite h zurücklegen darf. Es muss also gelten:$|\mathbf{u}|\Delta t < h$.

Kapitel 4
Stable Fluids Algorithmus

Mit den *Navier-Stokes* Gleichungen und den gezeigten numerischen Lösungsverfahren, ist eine exakte Simulation des Verhaltens von Fluids möglich. Eine genaue Berechnung ist aber aufwändig und zeitintensiv. Programme, in denen *Navier-Stokes* Gleichungen Verwendung finden, werden hauptsächlich zur Lösung technisch-physikalischer Probleme eingesetzt und müssen deshalb exakt arbeiten. Dieses ist auch einsichtig wenn man z.B. die Berechnung des Strömungswiderstands eines Flugzeuges oder einer Brücke berechnen will. Hier ist eine präzise Rechnung gefordert und notwendig. Entsprechend hoch ist auch der Rechenaufwand.

In der Computergrafik oder bei Spielen dagegen, ist wichtig, dass die Simulation überzeugend wirkt und vor allem schnell ist. Die Lösungsverfahren dürfen also nicht so komplex sein, dass sie nicht mehr auf einem Standard-PC oder einer Spielekonsole schnell genug ausgeführt werden können, sollten aber trotzdem ein realistisches Verhalten zeigen. Es sind also maßgeschneiderte Algorithmen erforderlich, um dieses Ziel zu erreichen. Erste Modelle in der Computergraphik legten mehr Wert auf das Visuelle als auf die physikalische Korrektheit. Die ersten Fluid Modelle bestanden aus Partikelsystemen. Mit der Einführung von Zufalls-Turbulenzen [26] gelang eine deutliche Verbesserung in der Simulation des Fluidsverhaltens. Diese Turbulenzen sorgten automatisch für rotierende Bewegungen des Fluids. Fluids, die mit solchen Verfahren simuliert werden, haben den Nachteil, dass sie nicht auf externe Kräfte von z.B. Nutzern reagieren können. Dies ist jedoch mit Modellen, die auf den *Navier-Stokes* Gleichungen basieren, möglich. Erste Implementierungen beschränkten sich nur auf zwei Dimensionen [17]. *Foster et al.* [9] zeigten die Verwendung der drei-dimensionalen *Navier-Stokes* Gleichungen in der Computergraphik. Effekte, wie Verwirbelung oder Umfließen von Objekten, die sonst schwer zu modellieren waren, wurden damit automatisch möglich. Ihr Verfahren basierte auf den Arbeiten von *Harlow et al.* [6] aus dem Jahre 1965. Da in der Arbeit ein explizites Lösungsverfahren benutzt wurde, dürfen nur sehr kleine Zeitschritte verwenden werden, damit das Verfahren stabil bleibt. Daraus ergibt sich, dass ihr Verfahren relativ langsam ist bzw. instabil bei größeren Zeitschritten wird. *Jos Stam* präsentierte 1999 in [27] ein stabiles und schnelles Lösungverfahren. Seine Methode arbeitet implizit und verwendet dabei ein so genanntes *semi-Lagrange* Verfahren. Dieses wird in der CFD relativ wenig genutzt, da es zu einem

numerischen Energieverlust führt. Das heißt, dass das Fluid schneller in der Bewegung gedämpft wird als bei einem realen Fluid. Dies kann aber vernachlässigt werden, wenn das Fluid durch externe Kräfte, z.B. einen Animator, "am Leben" gehalten wird. Der Algorithmus von *Stam* ist auch bei großen Zeitschritten stabil und erlaubt damit eine schnellere Simulation. Man kann die Simulation quasi so schnell machen wie man möchte, ohne dass sie instabil wird. Man darf aber dann nicht erwarten, dass sie dann noch exakt ist. Allerdings wird in der Computergraphik oft einiges als gut angesehen, was gut und plausibel aussieht. Im Folgenden wird nun dieser Algorithmus von *Stam* ausführlich erläutert. Er wurde für die Rauchsimulation im Animationssystem DUSTY aus Kapitel 5 verwendet.

4.1 Grundgleichungen

Ein Fluid, dessen Dichte und Temperatur relativ konstant sind, wird durch ein Vektorfeld $\vec{\mathbf{v}}$ und den Druck p beschrieben. Diese Werte können sich im Raum über die Zeit verändern und hängen vom Rand oder den Hindernissen ab. Mit gegebenen Geschwindigkeiten im Vektorfeld und dem Druck zum Zeitpunkt $t = 0$, können die späteren Zustände mit den *Navier-Stokes* Gleichungen beschrieben werden. Diese wurden im Kapitel 2 hergeleitet. Hier die Grundgleichungen für inkompressible Fluids.

$$\nabla \cdot \vec{\mathbf{v}} = 0 \tag{1}$$

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

Dies sind die Koordinaten freien Versionen der Gleichungen und gelten sowohl für den zwei- als auch drei-dimensionalen Fall. Eine Erklärung der Gleichungen findet sich in Kapitel 2. Es können die inkompressiblen *Navier-Stokes* Gleichungen benutzt werden, da kompressible Effekte bei Rauch mit einer Geschwindigkeit unterhalb des Schalls, vernachlässigt werden können. Dieses führt zu einer Vereinfachung der Berechnung. Bei der Simulation von Explosionen [14, 35] müssen allerdings die kompressiblen Effekte berücksichtigt werden.

Zum Arbeiten mit diesen Gleichungen wäre es wünschenswert nur eine Gleichung zu haben. Das kann durch Kombination der Gleichungen erreicht werden. Hierbei hilft die so genannte *Helmholtz-Hodge* Dekomposition. Diese besagt, dass sich ein Vektorfeld $\vec{\mathbf{w}}$ in ein divergenzfreies Vektorfeld $\vec{\mathbf{v}}$ und ein skalares Feld in der Form

$$\vec{\mathbf{w}} = \vec{\mathbf{v}} + \nabla q \tag{3}$$

zerlegen lässt. Es gilt also $\nabla \cdot \vec{\mathbf{v}} = 0$ und q ist ein skalares Feld. Mit dieser Tatsache kann man sich einen Operator P definieren der jedes Vektorfeld $\vec{\mathbf{w}}$ zu einem divergenzfreien

Feld, wie in Gleichung 1 gefordert, macht. Mit $\vec{\mathbf{v}} = \mathbf{P}\vec{\mathbf{w}}$ würde man das geforderte divergenzfreie Vektorfeld erzielen. Den P Operator erhält man implizit, indem man beide Seiten der Gleichung 3 mit ∇ multipliziert. Da $\nabla \cdot \vec{\mathbf{v}} = 0$ gilt, fällt $\vec{\mathbf{v}}$ aus der Gleichung raus und es bleibt:

$$\nabla \cdot \vec{\mathbf{w}} = \nabla^2 q. \tag{4}$$

Dies ist eine so genannte *Poisson*-Gleichung mit dem unbekannten Skalarfeld q und den Neumann-Randbedingungen $\frac{\partial q}{\partial n} = 0$. ∇^2 ist der Laplace-Operator $\triangle$ der als

$$\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

definiert ist. *Poisson*-Gleichungen sind in der numerischen Mathematik ein wohl bekanntes Problem für das es effektive Lösungsverfahren gibt. Die Lösung dieser Gleichung, also das skalare Feld q, wird genutzt, um die Divergenzfreiheit zu erreichen.

$$\vec{\mathbf{v}} = \mathbf{P}\vec{\mathbf{w}} = \vec{\mathbf{w}} - \nabla q$$

Durch die Anwendung des Operators auf beide Seiten von Gleichung 2 erhält man schließlich nur eine Gleichung für die Geschwindigkeiten.

$$\frac{\partial \vec{\mathbf{v}}}{\partial t} = \mathbf{P}\left(-(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}} + \nu\nabla^2\vec{\mathbf{v}} + \vec{\mathbf{f}}\right) \tag{5}$$

Dabei wurde ausgenutzt, dass $\mathbf{P}\vec{\mathbf{v}} = \vec{\mathbf{v}}$ und $\mathbf{P}\nabla p = 0$ gilt.

Da ein reines Vektorfeld visuell für eine Rauchsimulation nicht ausreicht, müssen Rauchpartikel durch das Feld bewegt werden. Die Partikel werden durch das Vektorfeld transportiert. Im Falle von Rauch ist es aber aufwändig jedes Partikel einzeln zu transportieren. Deshalb werden die Rauchpartikel durch die Dichte ρ des Rauchs ersetzt. Eine stetige Funktion berechnet dann für jeden gewünschten Punkt im Raum den Wert für ein Rauchpartikel. Die Entwicklung des Dichtefelds im Vektorfeld kann genau berechnet werden. Dazu braucht man eine weitere Gleichung, die der Gleichung für die Geschwindigkeiten sehr ähnlich ist. Man erhält also folgende Grundgleichungen auf denen der *Stam* Algorithmus basiert.

$$\frac{\partial \vec{\mathbf{v}}}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}} + \nu\nabla^2\vec{\mathbf{v}} + \vec{\mathbf{f}} \tag{6}$$

$$\frac{\partial \rho}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\rho + \kappa\nabla^2\rho + S \tag{7}$$

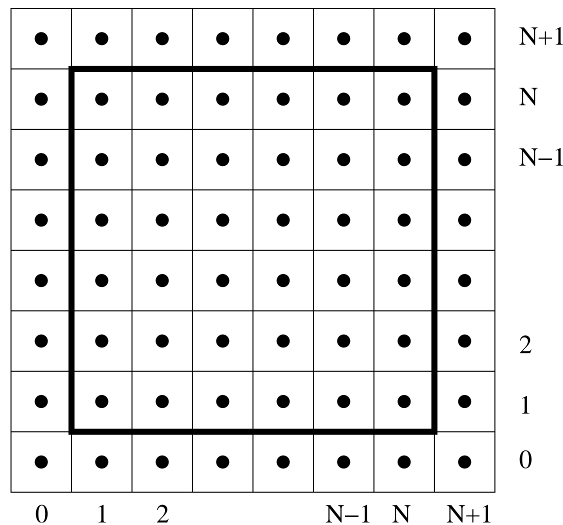

Abbildung 4.1: Diskretisierungsgitter

4.2 Lösen der Gleichungen

Ein effektives Lösen der Gleichungen 6 und 7 ist möglich, wenn man die Teile der Gleichungen einzeln löst und dann addiert. Zum Herleiten des Lösungsverfahrens, gehen wir zunächst von einem *festen* vorgegebenen Geschwindigkeitsfeld aus und betrachten die Bewegung der Dichte durch das Feld. Zum Lösen der partiellen Differentialgleichungen muss zunächst der Raum in einem Gitter diskretisiert werden. Die Vorgehensweise wurde in Kapitel 3 beschrieben. Es werden jeweils zwei Gitter pro Vektorkomponente und der Dichte benutzt. Diese werden jeweils nach einem Berechnungschritt vertauscht.
Abbildung 4.1 zeigt solche eine Diskretisierung für den zwei-dimensionalen Fall. Die Geschwindigkeiten und die Dichte werden jeweils im Zentrum einer Zelle gespeichert. Für die Randbedingungen wurde ein extra Rand verwendet, so dass das Gitter um jeweils zwei Zellen pro Achse grösser wird.

4.3 Bewegung der Dichte

Wir gehen nun von einem vorgegebenen festen Vektorfeld aus und betrachten die Entwicklung der Dichte. Im weiteren wird sich also auf Gleichung 7 bezogen. Die Terme der Gleichung müssen zum Lösen nun einzeln betrachtet werden.

$$\frac{\partial \rho}{\partial t} = \boxed{-(\vec{\mathbf{v}} \cdot \nabla)\rho} + \kappa \nabla^2 \rho + S$$

Der eingerahmte Term sorgt dafür, dass sich die Dichte ρ mit dem Vektorfeld bewegt. Der Rauch soll also der Richtung des Fluids folgen. Diesen Effekt kann man beobachten, wenn man z.B. Zigarettenrauch durch Pusten bewegt.

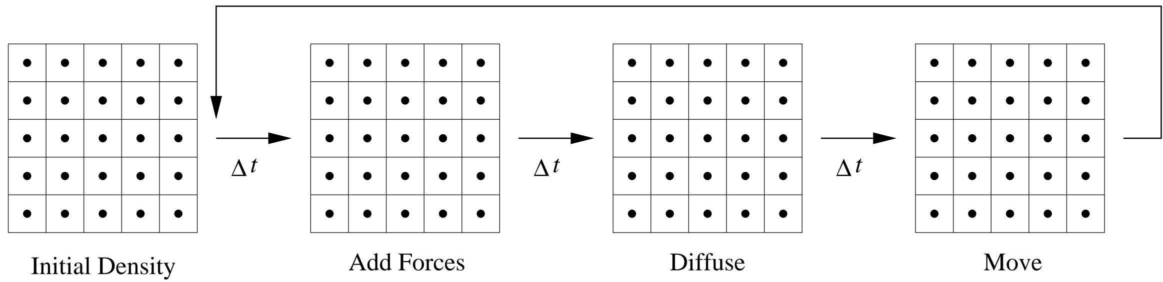

Abbildung 4.2: Schritte des Algorithmus

$$\frac{\partial \rho}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\rho + \boxed{\kappa \nabla^2 \rho} + S$$

Der zweite Term beschreibt die Diffusion der Dichte. Die Diffusion ist die räumliche Verteilung, ohne Einfluss von Strömung. Die Diffusion auf die eigentliche Strömung angewandt, sorgt für den Ausgleich der Strömungsbewegungen und stellt die Viskosität da. Die Stärke der Diffusion wird von κ bestimmt.

$$\frac{\partial \rho}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\rho + \kappa \nabla^2 \rho + \boxed{S}$$

S ist eine extern zum System zugefügte Dichte. Dies können verschiedene Dichtequellen im Raum sein und von einem Benutzer interaktiv zugefügt werden.

Abbildung 4.2 zeigt den Ablauf der Berechnung. Man startet mit einem Initalgitter das meistens leer sein wird. Die weiteren Schritte entsprechen den vorher erläuterten Termen der Gleichung 7. Der erste Schritt, das Hinzufügen der Dichte ist einfach zu realisieren. Dazu müssen die Dichtewerte der Quellen nur mit dem Zeitschritt multipliziert und dann zum Dichtegitter dazu addiert werden.

$$d_{n+1} = d_n + \Delta t S$$

4.3.1 Diffusion

Der nächste Schritt, nachdem die externe Dichte hinzugefügt wurde, ist die Berechnung der Diffusion. Abbildung 4.3 zeigt den Effekt der Diffusion.

$$D_n \xrightarrow{\Delta t} D_{n+1}$$

Wir gehen wieder von dem zwei-dimensionalen Fall aus. Zwischen den benachbarten (adjazenten) Zellen findet ein Austausch der Dichte statt. Abbildung 4.4 zeigt wie der Austausch erfolgt. Dichte verlässt eine Zelle und es strömt aus der Nachbarzelle Dichte hinein. Diese Tatsache wurde schon im Kapitel 2, bei der Herleitung der *Navier-Stokes* Gleichung, erkannt.

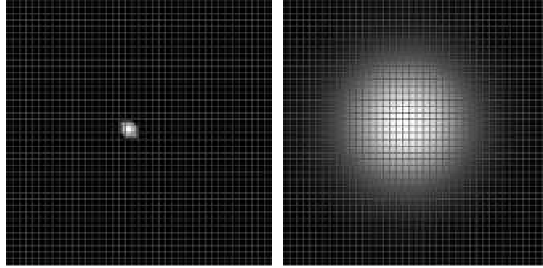

Abbildung 4.3: Links vor und rechts die Situation nach der Diffusion

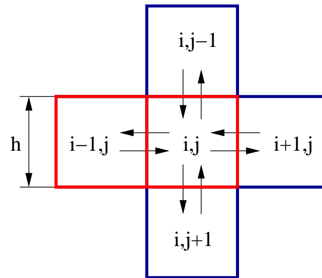

Abbildung 4.4: Austausch der Dichte bei der Diffusion

Die Diffusion wird durch den zweiten Term der Gleichung 7, $\kappa\nabla^2\rho$, beschrieben. Betrachtet wird jetzt nur die linke und die mittlere Zelle. Die Änderung zwischen den Zellen ist die Differenz zwischen den einfließenden und ausströmenden Dichtewerten unter Berücksichtung des Zeitschritts, der Diffusionsrate Kappa und dem Quadrat der Zellgrösse h.

$$\Delta D = \kappa \Delta t (D_{i-1,j} - D_{i,j})/h^2$$

Der Dichtefluss steigt also bei einem großen Zeitschritt, einer hohen Diffusionsrate oder einer kleinen Zellengröße. Werden alle Zellen zusammen betrachtet, kommt man zu folgender Gleichung:

$$D_{n+1_{i,j}} = D_{n_{i,j}} + \kappa \Delta t (D_{n_{i-1,j}} + D_{n_{i+1,j}} + D_{n_{i,j-1}} + D_{n_{i,j+1}} - 4D_{n_{i,j}})/h^2$$

Dies entspricht der Diskretisierung des Laplace-Operators ∇^2 aus dem Diffusionsterm. Eine Implementierung dieser Gleichung wäre sehr einfach. Leider wird diese Implementierung nicht stabil funktionieren. Das System wird instabil, wenn der Zeitschritt oder die Diffusionsrate zu groß sind oder die Zellengröße zu klein wird. Bei zu großen Diffusionsraten, fangen die Dichtewerte an zu oszillieren, bekommen negative Werte und divergieren letztendlich. Dies führt dazu, dass die Simulation sinnlos wird. Die Simulation wird instabil wenn

$$\Delta t \kappa / h^2 > 1/2$$

gilt. Dieses Verhalten ist typisch für instabile Methoden. Die Lösung für ein stabiles Verfahren ist in einer impliziten Methode zu suchen.

Hierbei sucht man nach der Dichte, die vor der Diffusion aktuell ist. Das erreicht man mit einem negativen Zeitschritt.

$$D_n \xleftarrow{-\Delta t} D_{n+1}$$

Es muss jetzt rückwärts in der Zeit geguckt werden. Als Gleichung wird das folgendermaßen ausgedrückt.

$$(\mathbf{I} - \kappa \Delta t \nabla^2) D_{n+1} = D_n$$

Hier ist $\mathbf{I}$ die Identitätsmatrix. Diskretisiert erhält man folgene Gleichung.

$$D_{n_{i,j}} = D_{n+1_{i,j}} - \Delta t \kappa (D_{n+1_{i-1,j}} + D_{n+1_{i+1,j}} + D_{n+1_{i,j-1}} + D_{n+1_{i,j+1}} - 4 D_{n+1_{i,j}})/h^2.$$

Das Problem hierbei ist, dass die rechte Seite, also D_{n+1}, unbekannt ist. Da der Dichtewert jeder Gitterzelle eine Unbekannte ist, erhält man ein lineares Gleichungssystem aus, für den zwei dimensionalen Fall, $M \times N$ Unbekannten. Dieses lineare Gleichungssystem muss gelöst werden. Das System kann allerdings sehr groß werden und hängt von der Größe des Diskretisierungsgitters ab. Dieses stellt aber kein großes Problem dar, da ein schwach besetztes System vorliegt. Es sind hauptsächlich die Diagonale und Umgebung der Matrix besetzt. Der Rest der Matrix ist mit Nullen aufgefüllt. Das Entstehen des Gleichungssystems wurde in Kapitel 3 genauer erklärt. Ein Lösen dieser Gleichungssysteme durch einen einfachen Gauß-Algorithmus, wäre hier unsinnig. Es würde zu viel Zeit benötigen und unnötig viele Multiplikationen mit Null enthalten. Für diese speziellen Matrizen existieren schnelle numerische Lösungsverfahren. Hier seien das SOR- und das konjugierte Gradienten Verfahren (CG-Verfahren) erwähnt. Eine genaue Erläuterung zu diesen Verfahren findet man im Anhang B

4.3.2 Transport

In diesem Schritt werden die Dichtepartikel durch das Vektorfeld bewegt. Verantwortlich hierfür ist der Term $-(\mathbf{v} \cdot \nabla)\rho$. Abbildung 4.5 zeigt diesen Vorgang. Man kann gut erkennen wie sich die Rauchpartikel entlang dem Vektorfeld ausbreiten.Wir gehen dabei wieder von einem bekannten festen Vektorfeld aus. Hier unterscheidet sich der Algorithmus von *Stam* von der üblicher Vorgehensweise in der Strömungsmechanik. *Foster et al.* [9] verwendeten zum Lösen des Terms Finite Differenzen was zu Instabilität führen kann, wenn die Gitterauflösung nicht fein genug ist oder der Zeitschritt zu groß gewählt wurde.

Wie beim Diffusionsschritt betrachten wir wieder die Strömungen zwischen den Zellen. Hier gibt das Vektorfeld die Richtung vor in der sich die Stöme bewegen. Eine naive Implementation führt wieder zu einem Instabilen Verfahren, wenn der Zeitschritt, die Geschwindigkeiten oder die Viskosität zu groß werden. Man könnte sich mit einem stabilen impliziten Verfahren behelfen. Dieses führt aber zu einer nicht symmetrischen Matrix mit

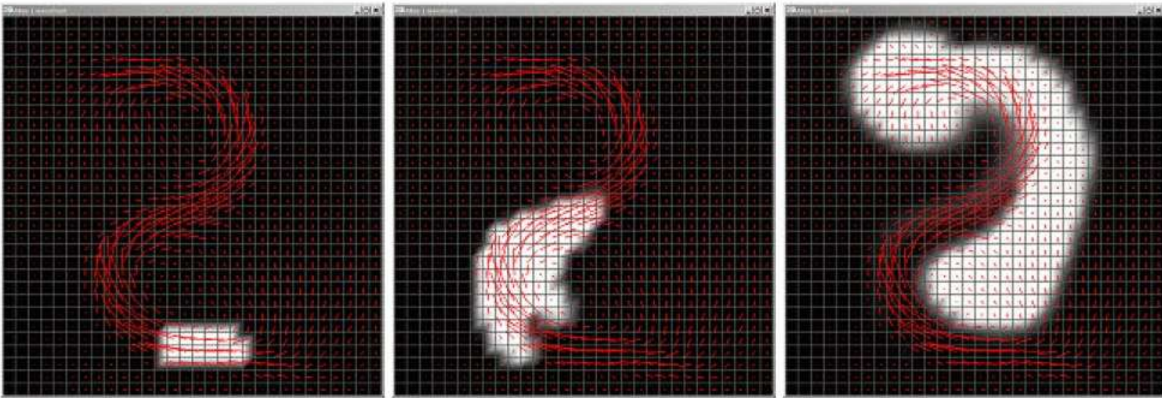

Abbildung 4.5: Tansport der Partikel durch ein festes 2D Vektorfeld (nach [29])

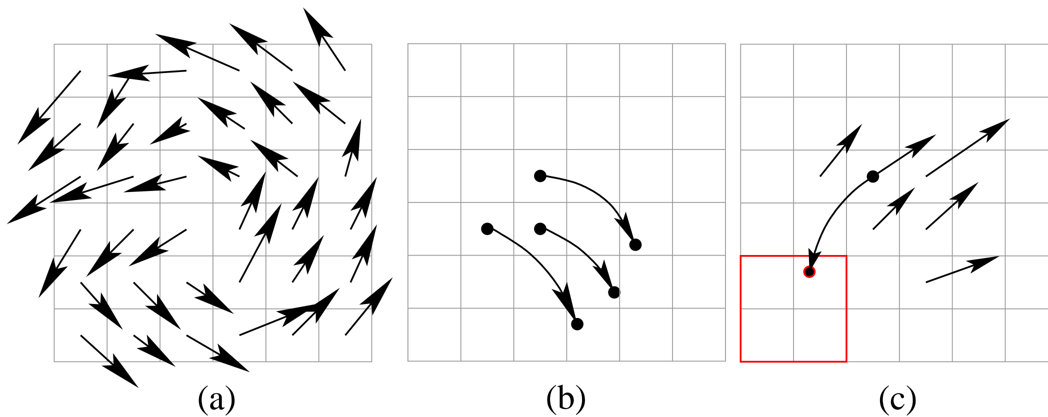

Abbildung 4.6: Partikelverfolgung: (a) Vektorfeld, (b) Partikelverfolgung endet nicht in Zellenmittelpunkt, (c) von einem Zellenmittelpunkt rückwärts in der Zeit gucken

unterschiedlichen Konstanten. Hierfür gibt es keine geeigneten schnellen Lösungsverfahren die nicht zur Instabilität führen.

Die Lösung des Problems besteht darin, dass die Dichte als Partikel betrachtet wird und eine Partikelverfolgung durch das Feld erfolgt, um die neue Position des Partikels zu erhalten. Dazu wird die Mitte einer Zelle als Partikel angenommen und mit dem Vektorfeld bewegt. Dieses Partikel repräsentiert die Rauchdichte in der Mitte der Zellen und ist damit eine Diskretisierung der Dichte.

Zur Umsetzung werden wieder zwei Gitter verwendet. Ein Quellgitter und ein Zielgitter in dem die neuen Dichtewerte eingetragen werden. Verfolgt man das Partikel nun vorwärts in der Zeit, wird es nicht unbedingt in einem Zellmittelpunkt enden (siehe Abbildung

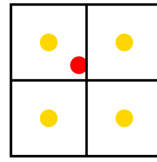

Abbildung 4.7: Interpolation der Dichte

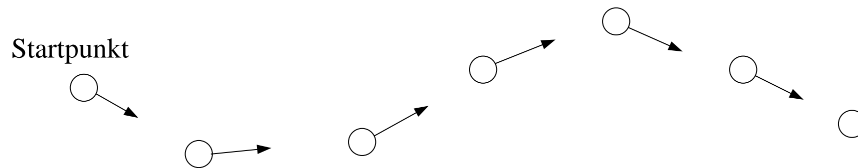

Abbildung 4.8: Partikel-Verfolgung über die Zeit

4.6(b)). Für die weiteren Berechnungen wird der Dichtewert in den Zellmittelpunkten benötigt. Eine Bestimmung des Dichtewertes in der Zellmitte ist jedoch auf diese Weise nicht möglich. Besser ist es, die Partikel im Quellgitter zu finden, die genau in einem Zellmittelpunkt nach dem Transport enden. Diese kann man finden, indem man rückwärts in der Zeit geht. Man betrachtet ein Partikel im Zellmittelpunkt und verfolgt es zu dem Punkt von wo es vor dem Zeitschritt hergekommen sein muss (Abbildung 4.6(c)). Der Dichtewert dieses Partikels wird dann in der neuen Position eingetragen. Der Ursprung des Partikels wird sich nicht in einem Zellenmittelpunkt befinden von dem man sofort den Dichtewert übernehmen könnte. Es ist hier eine Interpolation durch die am nächsten liegenden Nachbarzellen erforderlich.
Abbildung 4.7 verdeutlicht dieses Vorgehen. Der gesuchte Wert kann durch lineare Interpolation gefunden werden, da die vier umgebenden Werte bekannt sind. Diese Vorgehensweise wird auch *semi-Lagrange* Verfahren genannt. Diese Verfahren haben den Vorteil, dass sie nicht von den CFL Bedingungen abhängen und große Schrittweiten ermöglichen. Die CFL Bedingungen besagen, dass kein Partikel des Fluids in der Zeit Δt mehr als eine Gitterweite h zurücklegen darf. Siehe dazu das Glossar auf Seite 77.

4.3.3 Partikel-Verfolgung

Wie gezeigt wurde, ist die Partikel-Verfolgung eine wichtige Grundlage zur Berechnung des Transports der Rauchpartikel durch das Vektorfeld.
Abbildung 4.8 zeigt ein Beispiel dafür wie sich ein Partikel über mehrere Zeitschritte bewegen könnte. Das entspricht der Gleichung

$$dx = \vec{\mathbf{V}}dt$$

oder in Integralform

$$\vec{x}(t) = \int_t \vec{\mathbf{V}}dt.$$

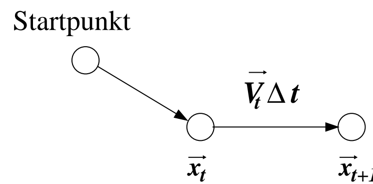

Abbildung 4.9: Euler-Integration

Kritisch ist hier die Wahl von dt. Wird der Zeitschritt zu groß gewählt, können kleinere Details übersprungen werden. Zum Lösen ist eine numerische Integration erforderlich. Hierfür kommen im wesentlichen zwei Verfahren in Frage. Diese sind:

- Euler Integration
- Kutta-Runge Verfahren

Diese beiden Verfahren sollen nun genauer betrachtet werden.

4.3.3.1 *Euler* Integration

Die *Euler* Integration ist das mit Abstand einfachste Verfahren. Dieses Verfahren ist sehr populär, da man mit ihm schnell erste Ergebnisse erhält.

$$\vec{x}_{t+1} = \vec{x}_t + \vec{\mathbf{V}}\Delta t$$

Die Position ergibt sich aus der alten Position plus eine um dem Zeitschritt verringerte Geschwindigkeiten des Vektorfelds $\vec{\mathbf{V}}$. In Abbildung 4.9 wird dieser Vorgang gezeigt. Dies lässt sich schnell und einfach berechnen. Da man in unserer Partikel-Verfolgung rückwärts in der Zeit geht, muss die Gleichung wie folgt aussehen:

$$\vec{x}_{t+1} = \vec{x}_t - \vec{\mathbf{V}}\Delta t.$$

Der numerische Fehler ist

$$\mathbf{O}(\Delta t^2).$$

4.3.3.2 *Runge-Kutta* Verfahren

Ein besseres numerisches Verhalten hat das so genannte *Runge-Kutta* Verfahren der zweiten Ordnung. Hier wird mittels des *Euler* Verfahrens ein erster Schritt durchgeführt, der durch einen zweiten verfeinert wird.

$$\vec{x}_{t+1} = \vec{x}_t + \frac{\Delta t}{2}(\vec{\mathbf{V}}_t + \vec{\mathbf{V}}_{t+1})$$

$\vec{\mathbf{V}}_{t+1}$ wird hier zunächst mittels der *Euler* Methode berechnet.
Der numerische Fehler dieses Verfahrens ist

$$\mathbf{O}(\Delta t^3).$$

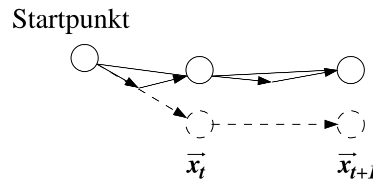

Abbildung 4.10: Runge-Kutta-Verfahren

Mit dem *Runge-Kutta* Verfahren erreicht man eine größere Genauigkeit. Es erlaubt größere Zeitschritte bei gleichem Fehler im Vergleich zum *Euler* Verfahren. Die Berechnung ist allerdings aufwändiger. Neben dem Verfahren zweiter Ordnung ist auch das Verfahren vierter Ordnung sehr verbreitet. Zum Zwecke einer Rauchsimulation genügt das Verfahren zweiter Ordnung, da es guter Kompromiss zwischen Qualität und Rechenzeit ist.

4.4 Bewegung des Vektorfelds

Bisher wurde von einem festen Vektorfeld ausgegangen, in dem sich Materie bzw. deren Dichte bewegte. Das Vektorfeld selber verändert sich bei einem Einfluss von externen Kräften ebenfalls. Das soll nun untersucht werden.

$$\frac{\partial \vec{\mathbf{v}}}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}} + \nu \nabla^2 \vec{\mathbf{v}} + \vec{\mathbf{f}} \tag{8}$$

$$\frac{\partial \rho}{\partial t} = -(\vec{\mathbf{v}} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{9}$$

Wenn man nochmal die beiden Gleichungen für das Vektorfeld (Gleichung 8) und die Dichte (Gleichung 9) vergleicht, fällt eine starke Ähnlichkeit auf. Auch bei der oberen Gleichung gibt es einen Diffusionsterm $\nu\nabla^2\mathbf{v}$, mit der Viskosität ν. Die Berechnung dieses Terms wurde schon für die Dichte hergeleitet. Für die Geschwindigkeiten im Vektorfeld erfolgt die Berechnung auf ähnliche Art und Weise. Dabei muss jedoch der Term für jede Vektor-Komponente berechnet werden. Es müssen also zwei Diffusionsgleichungen für den 2D- und drei für 3D-Fall gelöst werden. Das Hinzufügen externer Kräfte zu dem System beschreibt der letzte Ausdruck in der Gleichung. Hier geht man ebenfalls wie bei dem Hinzufügen externer Dichte vor.

Am Interessantesten ist nun der erste Term $(\vec{\mathbf{v}} \cdot \nabla)\vec{\mathbf{v}}$. Er hat große Ähnlichkeit mit dem entsprechenden Term für die Dichte-Gleichung. Der Term ist durch das doppelte Auftreten der Geschwindigkeiten $\vec{\mathbf{v}}$ dafür verantwortlich, dass die Gleichung nicht-linear wird. In dem Fall für die Dichte konnte man diesen Ausdruck so interpretieren, dass die Dichte dem Vektorfeld folgt. Hier müsste man nun sagen, dass das Vektorfeld sich selbst folgt und bewegt. Dieses kann wirklich angenommen werden. Das Vektorfeld bewegt sich quasi selber weiter. Das hat den großen Vorteil, dass man die gleichen Lösungsverfahren benutzen kann wie sie für die Dichte hergeleitet wurden. Es wird das beschriebene *semi-Lagrange*

Verfahren benutzt. Bei der Berechnung werden wieder zwei Gitter benötigt. Man geht zunächst einen Zeitschritt rückwärts in der Zeit und bestimmt, wie bei der Dichte, mittels Interpolation im ersten Gitter den dortigen Vektor. Dieser Wert wird dann in die aktuelle Position im zweiten Gitter eingetragen. Man benötigt für jede Vektorkomponente zwei Gitter. Die Bestimmung des neuen Vektors muss dann für jede Komponente einzeln berechnet werden. Die Position ist jeweils die gleiche, nur die Interpolation zur Bestimmung des Vektors muss komponentenweise erfolgen.

4.4.1 Erhaltung der Masse

In den Grundgleichungen der *Navier-Stokes* Gleichungen ist gefordert, dass die Masse erhalten bleibt, d.h. dass das Vektorfeld divergenzfrei ist. Das besagt die folgende Gleichung, die Teil der *Navier-Stokes* Gleichung ist.

$$\nabla\cdot \vec{\mathbf{v}} = 0$$

Betrachten wir Abbildung 4.4 auf Seite 30 die den Dichteaustausch benachbarter Zellen zeigt. Sie kann genauso für den Austausch der Vektorgeschwindigkeiten benutzt werden. Wenn obige Gleichung gelten soll, muss also für den 2D-Fall

$$U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1} = 0$$

erfüllt sein. U und V sind die Komponenten eines 2D-Vektors. Diese Gleichung wird nach der Durchführung der beschriebenen Teilschritte zum Lösen der *Navier-Stokes* Gleichungen nicht erfüllt sein. Das Vektorfeld muss korrigiert werden, damit die Gleichung erfüllt ist. Wie das geschehen kann, wurde schon in Abschnitt 4.1 hergeleitet. Dabei wurde die *Hodge-Dekompostion* benutzt die besagt dass sich jedes Vektorfeld in ein divergenzfreies Feld und ein Gradientenfeld zerlegen lässt.

$$\vec{\mathbf{w}} = \vec{\mathbf{v}} + \nabla q$$

Bei der Herleitung in Abschnitt 4.1 kam man schließlich zu der Gleichung

$$\nabla\cdot \vec{\mathbf{w}} = \nabla^2 q \tag{10}$$

womit sich das divergenzfreie Vektorfeld bestimmen lässt.

$$\vec{\mathbf{v}} = \vec{\mathbf{w}} - \nabla q \tag{11}$$

Es muss also das skalare Feld q bestimmt werden, damit sein Gradient vom Vektorfeld subtrahiert werden kann, um Divergenzfreiheit zu erreichen. q kann man durch Lösen der Gleichung 10 erhalten. Diese ist eine so genannte *Poisson*-Gleichung und kann für den 2D-Fall wie folgt diskretisiert werden:

$$(U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1})/h = (Q_{i+1,j} + Q_{i-1,j} + Q_{i,j+1} + Q_{i,j-1} - 4Q_{i,j})/h^2.$$

Q ist hier also unbekannt und kann durch Bildung eines Gleichungssystems, wie beim Diffusions Schritt, bestimmt werden. Das Gleichungssystem ist wiederum schwach besetzt

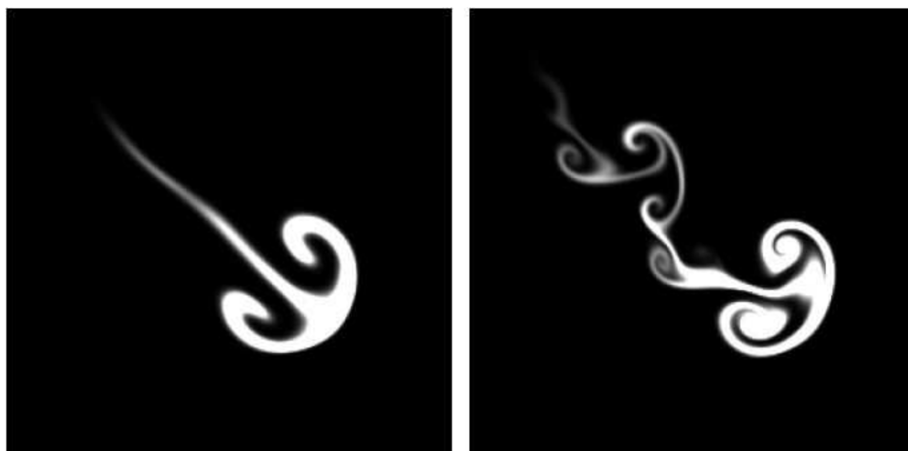

Abbildung 4.11: 2D Rauchsimulation: links ohne und rechts mit *Vorticity Confinement*(aus[21])

und kann mit den gängigen Methoden wie SOR oder das konjugierte Gradienten Verfahren effektiv gelöst werden. Erläuterungen dazu finden sich in Anhang B. Nachdem das skalare Feld nun bestimmt wurde, kann sein Gradient vom Vektorfeld, wie in Gleichung 11, einfach subtrahiert werden,um die geforderte Divergenzfreiheit zu erreichen.

4.4.2 Vorticity Confinement

Rauch erscheint realistisch, wenn dieser rotierende und turbulente Strukturen enthält. Durch die numerische Dämpfung des *Stable Fluids* Algorithmus werden diese Eigenschaften schnell unterdrückt. Eine Lösung um diese Turbulenzen wieder zu erhalten, wäre eine zufällige Verwirbelung des Fluids. Dieses Vorgehen garantiert aber nicht, das die Turbulenzen auch an den Stellen wieder zugefügt werden wo sie verloren gegangen sind. Das kann zu einem unrealistischen Verhalten führen.
Eine bessere Methode mit dem Namen *Vorticity Confinement* wurde bereits 1994 von *Steinhoff et al.* [30] beschrieben. Die Grundidee dieser Methode ist, die durch die numerische Dissipation verloren gegangene Energie dem Vektorfeld wieder zuzuführen. *Fedkin et al.* [7] verwendeten 2001 diese Methode erstmalig in der Computergraphik. Als Beispiel für den 2D-Fall zeigt Abbildung 4.11 deutlich die Verbesserung einer Rauchsimulation durch das *Vorticity Confinement.* Im linken Bild sind kaum Verwirbelungen zu erkennen, während sie rechts deutlich hervortreten.

Statt der Geschwindigkeiten im Vektorfeld wird nun die Verwirbelung betrachtet. Der erste Schritt zur Berechnung, ist also die Bestimmung der Stellen an denen Rotationen auftreten. Die Gebiete mit Verwirbelung werden mit

$$\vec{\omega} = \nabla \times \vec{\mathbf{v}}$$

bestimmt. Man erhält wieder ein Vektorfeld. Im 2D-Fall stehen die Vektoren senkrecht auf der Fluid-Ebene. Das Ergebnis kann als Maß genommen werden, wie hoch die Rotation an jeder Stelle im Fluid ist. Diese Rotation wird allerdings numerisch beim Stable Fluid Algorithmus gedämpft. Um das zu verhindern, wird eine externe Kraft senkrecht zum Gradientenfeld des Wirbelfeldes hinzugefügt. Das normierte Gradientenfeld ist

$$\mathbf{N} = \frac{\eta}{|\eta|} \quad (\eta = \nabla|\vec{\omega}|)$$

Der Gradientenvektor jeder Gitterzelle zeigt auf den größten Anstieg der Stärke der Wirbelbildung. Senkrecht zu diesem Gradienten muss nun eine externe Kraft, die proportional zur Verwirbelung ist, hinzugefügt werden.

$$\vec{\mathbf{f}}_{conf} = \varepsilon(\mathbf{N} \times \vec{\omega})$$

Der Faktor ε bestimmt die Stärke mit der $\vec{\mathbf{f}}_{conf}$ wieder dem System zugefügt wird. Wie Abbildung 4.11 zeigt, ist *Vorticity Confinement* eine gute Möglichkeit realistische Wirbel zu erzeugen.

4.5 Ablauf der Simulation

Nachdem die einzelnen Schritte zum Lösen der Gleichungen entwickelt wurden, kann daraus der nötige Ablauf der Simulation angegeben werden. Als erstes muss der Raum in Zellen unterteilt werden. Diese ergeben die erwähnten Gitter zur Diskretisierung. Es müssen je zwei Gitter für jede Komponente des Vektorfeldes und des Dichtefeldes bereitgestellt werden. Die Geschwindigkeiten des Vektorfelds und die Dichtewerte werden im Zentrum jeder Zelle definiert. In der Strömungsmechanik werden oft versetzte Gitter, so genannte *staggered Grids*, benutzt. Hier befinden sich die Geschwindigkeiten am Rand der Zellen. Das garantiert ein besseres numerisches Verhalten, ist aber umständlicher zu benutzen und zeitaufwändiger bei der Differenzenbildung. Für eine einfache Rauchsimulation in der Computergraphik reicht es, die Geschwindigkeiten im Zentrum jeder Zelle anzunehmen. Für eine hohe Qualität oder Verwendung des Algorithmus zur Simulation von Flüssigkeiten (siehe *Foster et al.* [8]), eignet sich dagegen das *staggered Grid* besser.

Zu Beginn der Simulation sind die Gitter im Allgemeinen leer. Man könnte jedoch ein Vektorfeld oder eine Dichteverteilung vorgeben. Das Ergebnis einer Berechnung in jedem Schritt wird im jeweils anderen Gitter gespeichert. Die Gitter werden danach getauscht. Für den 2D-Fall kann man folgende Simulationschleife angeben.

```
while(simulation)
{
      swap(v1,v0); swap(s1,s0);
      vstep(u1,u0,visc,F,dt);
      sstep(s1,s0,vS,aS,u1,source,dt);
}
```

`v` sind die Gitter für das Vektorfeld und `s` die Gitter für das Dichtefeld. Die Viskosität wird mit `visc` angegeben und `vS` ist die Diffusion der Materie. `F` sind die externen Kräfte, die dem Vektorfeld hinzugefügt werden und `source` ist externes Material, welches in die Simulationsumgebung eingefügt wird. Der Zeitschritt, der die Geschwindigkeit der Simulation steuert, wird mit `dt` angegeben. Dieser Wert wird im Allgemeinen zwischen 0.1 und 1.0 liegen. Bei instabilen Verfahren müsste hier ein wesentlich kleinerer Wert benutzt werden. Ein Wert der noch nicht erwähnt wurde ist `aS`. Er gibt die *dissipation* an, die festlegt wie sich die Materie im Raum verteilt und weniger wird. Ohne den *dissipations* Schritt würde sich ein Raum während der Simulation ständig mit Materie füllen. Der Raum würde quasi "voll gequalmt" werden. Deshalb wird jedes Element aus dem Dichtefeld nach den Simulationsschritten mit $1 + aS * dt$ dividiert. Dadurch verschwindet älterer Rauch langsam aus dem Simulationsraum.
Die einzelnen Schritte für den `vstep()` also für das Vektorfeld sehen so aus:

```
vstep(u1, u0, visc, F, dt)
{
        addforce(u0,F,dt);
        diffuse(u1,u0,visc,dt);
        transport(u0,u1,u1,dt);
        project(u1,u0,dt);
}
```

Die `project()` Prozedur sorgt für die, von den *Navier-Stokes* Gleichungen geforderte, Divergenzfreiheit des Vektorfeldes. Da für die Behandlung der Dichte und des Vektorfeldes die gleichen Verfahren benutzt worden sind, können für den `sstep()` die selben Routinen des `vstep()` verwendet werden.

```
sstep(s1, s0, k, a, u, source, dt)
{
        addforce(s0,source,dt);
        diffuse(s1,s0,k,dt);
        transport(s0,s1,u,dt);
        dissipate(s1,s0,a,dt);
}
```

Die Routine ist also sehr ähnlich zum `vstep()`. Es wird hier keine Prozedur `project()` benötigt. Dafür gibt es die `dissipate()` Funktion die für eine „Auflösung“ des Rauches sorgt. Nachdem ein neues Dichtefeld berechnet wurde, kann dieses in geeigneter Weise ausgeben werden. Welche Möglichkeiten es dafür gibt, wird in Kapitel 5 anhand der Implementation erläutert.

4.6 Lösen der Gleichungen mittels FFT

In seiner Veröffenlichung [27] zeigt *Stam* eine alternative Lösungsmöglichkeit für die Diffusion und den Schritt zum Erreichen der Divergenzfreiheit basierend auf einer *Fourier* Transformation. Dieses funktioniert aber nur, wenn man einen periodischen Rand hat und sich keine Objekte im Simulationsgebiet befinden. Ausführlich beschreibt *Stam* in [28] das Verfahren und zeigt dort, dass ein *Navier-Stokes* Solver in 70 Zeilen C Code möglich ist. Wenn das Vektorfeld periodisch ist, kann man es in den Fourier-Raum transformieren.

$$\mathbf{v}(\mathbf{x}, t) \longrightarrow \widehat{\mathbf{v}}(\mathbf{k}, t)$$

Im Fourier-Raum ist der Gradientenoperator ∇ äquivalent zu einer Multiplikation mit $i\mathbf{k}$, wobei $i = \sqrt{-1}$ gilt. Das vereinfacht wesentlich den Diffusions-Schritt sowie den Projektions-Schritt zum Erzeugen der Divergenzfreiheit, da nun keine Gleichungssysteme mehr gelöst werden müssen. Die Berechnung der Schritte ergibt sich also folgendermaßen:

$$\mathbf{I} - \nu\Delta t\nabla^2 \longrightarrow 1 + \nu\Delta t k^2$$

$$\mathbf{P}\mathbf{w} \longrightarrow \widehat{\mathbf{P}}\widehat{\mathbf{w}}(\mathbf{k}) = \widehat{\mathbf{w}}(\mathbf{k}) - \frac{1}{k^2}(\mathbf{k} \cdot \widehat{\mathbf{w}}(\mathbf{k}))\mathbf{k}$$

Es gilt weiterhin $k = |\mathbf{k}|$. Der Operator $\widehat{\mathbf{P}}$ projiziert den Vektor $\widehat{\mathbf{w}}(\mathbf{k})$ auf eine Ebene die senkrecht zu der Wellennummer $\mathbf{k}$ ist. Die Fourier-Transformation eines divergenzfreien Vektorfeldes liegt senkrecht zu den Wellennummern. Die Komplexität der Fourier-Transformation ist $O(NlogN)$. Dieses ist theoretisch aufwändiger als eine Lösung mit den schnellsten Verfahren zum Lösen schwachbesetzter Gleichungssysteme. Diese sind aber umständlich und kompliziert zu implementieren. Eine Implementation mittels der FFT ist entscheidend einfacher. Für die Berechnung der Fourier-Transformation verwendet *Stam* die **FFTW** Bibliothek [15]. Die FFTW ist eine populäre C Biliothek für eine schnelle Fourier-Transformation. Es muss also abgewogen werden, ob sich eine Berechnung mittels FFT lohnt. Für den 2D-Fall kann es eine lohnende Alternative sein. Nachteilig ist hier, dass von einem periodischem Rand ausgegangen wird. Fluid was z.B. den rechten Rand verlässt erscheint am linken Rand wieder. Es ist weiterhin mit dieser Methode nicht möglich Objekte im Fluid zu behandeln. *Stam* nutzte dieses Verfahren deshalb vorrangig zur Erzeugung von Texturen.

Kapitel 5
Dusty - Eine Skriptbasierte Rauchsimulation

Dieses Kapitel beschreibt den Aufbau und Implementation eines Animationssystems zur Rauchsimulation. Grundlage der Implementation ist der in Kapitel 4 beschriebenen *Stable Fluids* Algorithmus von *Stam* [27]. Ziel des Projektes ist eine realitätsnahe Simulation von Rauch in Echtzeit.
Das Ergebnis der Implementierung ist das Simulationssystem Dusty. Die wesentlichen Merkmale von Dusty sind:

- realitätsnahe Echtzeit Simulation von Rauch und Gasen
- flexibles Skriptsystem
- Benutzer Interaktion während der Laufzeit
- Alphakanal Unterstützung zur Montage in Filmsequenzen
- einfach zu Bedienen
- auf drei Betriebssystemen verfügbar (Linux, Windows, MacOS)

Auf die Implementierung soll in diesem Kapitel näher eingegangen werden. Vor den weiteren Erläuterungen, werden kurz die verwendeten externen Bibliotheken vorgestellt, die die Entwicklung von Dusty in dieser Form möglich gemacht haben.

5.1 Benutzte Komponenten und Bibliotheken

Für Dusty wurden verschiedene externe Komponenten verwendet, wie z.B. für das GUI oder den Skriptinterpreter, verwendet. Vor der weiteren Beschreibung von Dusty soll kurz auf diese eingegangen werden.

5.1.1 TEKlib

Eine tragende Rolle bei der Entwicklung von DUSTY spielt die *TEKlib*[18]. Sie ist eine plattform unabhängige Multimedia-Bibliothek, virtuelles OS, Middleware und Entwicklungspaket. *TEKlib* steht unter der BSD-Lizenz und kann damit einfach für eigene Projekte verwendet werden. Sie vereinfacht das Programmieren z.B. durch einfachere Handhabung von Systemfunktionen. Diese werden auf allen portierten Systemen gleich benutzt. *TEKlib* bildet diese dann auf das benutzte Betriebssystem ab. Weiterhin ist *TEKlib* modular aufgebaut und kann dadurch leicht erweitert werden. Wichtig war, das *TEKlib* eine gute Unterstützung für Threads und Semaphoren, welche für die Integration des Skript-Interpreters in DUSTY wichtig sind, bietet. Da *TEKlib* auf mehreren System verfügbar ist, sind mit der *TEKlib* entwickelte Programme meistens sofort portabel und können für das entsprechende Zielsystem einfach neu kompiliert werden. Eine Besonderheit von *TEKlib* ist die Verwendung eigener Typen. So können auf jedem System die gleichen Typen benutzt werden. Diese werden auf die korrespondierenden Typen des Systems gemappt. Statt *int* oder *float* wird z.B. in *TEKlib TINT* und *TFLOAT* benutzt.

5.1.2 LUA

LUA [32] ist eine Skriptsprache die hauptsächlich zur Erweiterung von Applikationen entwickelt wurde. *LUA* hat einen kleinen eigenen Sprachsatz. Der Vorteil besteht darin, dass leicht eigene Funktionen implementiert werden können. So konnte der Befehlssatz um die Funktionen, die für DUSTY nötig sind, erweitert werden. *LUA* wurde als Modul in *TEKlib* implementiert und konnte so leicht in DUSTY integriert werden.

5.1.3 FLTK

Die Benutzungsschnittstelle von DUSTY wurde mit der Bibliothek *FLTK* (The Fast Light Toolkit) entwickelt. Diese Bibliothek ist auf drei Betriebssystemen verfügbar, basiert auf OpenGL und enthält eine GLUT Emulation. Da die Ausgabe der Simulationsergebnisse von DUSTY mit OpenGL erfolgen sollte, war sie eine gute Wahl für eine Implementation.

5.2 Struktur von Dusty

Abbildung 5.2 zeigt einen groben Überblick des internen Aufbaus von DUSTY. Die wichtigen Teile sind das Benutzerinterface mit der Ausgabe der Simulationsergebnisse, der *Navier-Stokes* Solver und der Skriptinterpreter.

Der Skriptinterpreter läuft in einem eigenen *Thread*. Um den Ablauf mit dem Hauptprogramm zu synchronisieren, ist die Benutzung von *Semaphoren* notwendig. Wie das realisiert wurde, wird in Abschnitt 5.4 beschrieben. Der Aufruf des *Navier-Stokes* Solvers zur Ausführung eines Simulationsschrittes, erfolgt entweder intern oder kann von dem gestarteten Skript gesteuert werden. Dies ist sinnvoll, wenn das Skript nicht nur zum Aufbau der Szene, sondern auch zur Animation der Szene verwendet wird. In den nächsten Sektionen werden nun die einzelnen Teile von DUSTY genauer beschrieben.

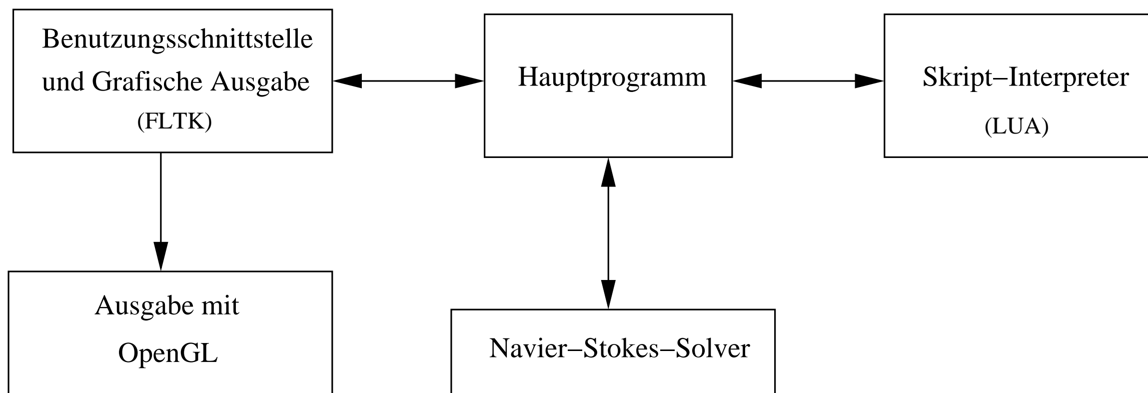

Abbildung 5.1: Aufbau des Simulations-Systems

5.3 *Navier-Stokes* Solver

Das Lösen der *Navier-Stokes* Gleichung ist die Grundlage der Strömungssimulation in DUSTY. Dieses kann effizient mit dem in Kapitel 4 beschriebenen *Stable Fluids* Algorithmus realisiert werden. Die Abarbeitung des Algorithmus wurde optimiert, so dass eine Simulation in Echtzeit möglich ist. Die nötigen Schritte und ihre Umsetzung in DUSTY werden nun genauer erläutert. Die einzelnen Schritte der Simulation sind:

1. Füge externe Kräfte und Dichte zu
2. Diffusion
3. Advektion bzw. Bewegung des Vektor- und Dichtefeldes
4. Divergenzfreiheit herstellen
5. Vorticity Confinement
6. gehe zu Schritt 1

Der letzte Schritt, das *Vorticity Confinement*, ist optional und dient dazu Verwirbelungen zu erhalten oder zu verstärken. Bevor die Simulationsschritte ausgeführt werden können, muss Speicher für die nötigen Gitter zur Diskretisierung bereitgestellt werden. Wie aus dem Algorithmus hervorgeht, benötigt man jeweils zwei Gitter zwischen denen nach jedem Schritt gewechselt wird. Die einzelnen Komponenten der Geschwindigkeiten, u, v und w werden jeweils in einem separaten Gitter gespeichert. Für das Dichtefeld werden drei Gitter benötigt, wenn der COLOR-Modus aktiviert wird. Das heißt, das farbiger Rauch erzeugt werden soll. In diesem Fall wird je ein Gitter für die R, G, B Komponente benutzt. Für die Temperatur wird ebenfalls ein Gitter bereitgestellt. Alle Gitter müssen doppelt vorhanden sein. Ihre Größe berechnet sich durch $(wx+2)*(wy+2)*(wz+2)$. Die Erweiterung um 2 ist für den Rand notwendig. Siehe dazu auch Abbildung 4.1 und Erklärung

auf Seite 28. Es wird ein eindimensionales Feld benutzt, um den Zugriff auf die einzelnen Elemente auf Geschwindigkeit zu optimieren. Der Index für die Elemente wird mit einem Makro berechnet.

```
#define IX(i,j,k) ((i) + (c->wx+2)*(j) + (c->wx+2)*(c->wy+2)*(k))
```

Es kann so z.B. mit `u[IX(i,j,k)]` auf ein Element im Feld zugegriffen werden. Da es öfter vorkommt, dass man den gleichen Index für die verschiedenen Gitter benötigt, reicht es wenn der Index mit dem Makro nur einmal berechnet wird. Dann kann dieser Wert in einer Variablen gespeichert und direkt für die weiteren Gitter verwendet werden. Dieses spart Rechenzeit.

5.3.1 Objekte und Quellen für eine Szene

Bevor die Simulation ausgeführt werden kann, muss festgelegt werden, wo im System sich Objekte und Quellen für externe Kräfte befinden. Diese werden von dem geladenen Szenenskript bestimmt. Welche Möglichkeiten es dabei gibt, wird in Abschnitt 5.4 beschrieben. Die in der Szene vorhandenen Objekte und Quellen werden in einer verketteten Liste verwaltet. Um Objekte bei der Simulation als Hindernisse zu berücksichtigen, müssen diese in Voxel überführt werden. Das heißt, es muß bestimmt werden, welche Gitterzelle sich in einem Hindernis befindet. Dazu wird nach dem Laden der Szene (oder wenn die Objekte ihre Position verändert haben) die Prozedur *voxelize()* aufgerufen. Hier wird dann für jede Art von Objekt (z.B. Kugel, Quader) geprüft, welche Zellen durch das Objekt belegt werden. Diese Information wird in einem zusätzlichen Feld *bounds* vom Typ *voxel* gespeichert. Ein Voxel hat folgene Struktur:

```
typedef struct _vox
{
     TBOOL taken;
     TFLOAT temp;
     TINT  type;
} voxel;
```

Mit dem Element *taken* der Struktur wird bestimmt, ob die Gitterzelle von einem Hindernis belegt ist oder nicht. Hat das Objekt eine bestimmte Temperatur, wird diese in *temp* gespeichert. Im Element *type* wird der Typ der Zelle festgelegt. Das heißt, es wird unterschieden ob sich die Zelle im Inneren des Hindernisses oder am Rand befindet. Dabei muss unterschieden werden, wo sich der Rand relativ zu der Zelle befindet. Es wird z.B. unterschieden ob sich die Zelle oben, unten, links, rechts, vorne oder hinter dem Objekt befindet. Dies ist nötig, um die in Abschnitt 3.2.3 auf Seite 21 erläuterten Randbedingungen auch an den Kanten der Hindernisse einzuhalten.
Für die Quellen sind diese Bestimmungen nicht notwendig. Sie werden für den ersten Schritt der Simulation, das Hinzufügen der externen Kräfte und Dichten, benötigt. Dieser Schritt wird als nächstes beschrieben.

5.3.2 Externe Kräfte und Dichten

Die Simulation beginnt mit dem Hinzufügen von externen Kräften und Materie in Form von Dichte zu dem System. Dieses ist sehr einfach, da nur

$$\vec{v}_{n+1} = \vec{v}_n + \Delta t \vec{f}$$

berechnet werden muss. Der folgende Code-Ausschnitt zeigt, wie das in DUSTY umgesetzt wurde. Es wird durch die Liste der Objekte iteriert. Wenn ein Objekt eine Quelle ist, werden an der Position der Quelle im Gitter ihre Geschwindigkeiten und Materie, multipliziert mit dem Zeitschritt, in das Gitter übertragen.

```
/* iterate objects */
onode = c->objects.tlh_Head;
while ((onext = onode->tln_Succ))
{
  TINT idx;
  obj *tmp = (obj *)onode;

  if(tmp->type==SOURCE)
  {
    x = tmp->int_pos[0];
    y = tmp->int_pos[1];
    z = tmp->int_pos[2];

    idx = IX(x,y,z);
    c->rdens[idx] += c->dt*tmp->rdens;
    c->gdens[idx] += c->dt*tmp->gdens;
    c->bdens[idx] += c->dt*tmp->bdens;

    c->u[idx] += c->dt*tmp->uforce;
    c->v[idx] += c->dt*tmp->vforce;
    c->z[idx] += c->dt*tmp->wforce;

    c->temp[idx] = tmp->temp;
  }
  onode = onext;
}
```

Wenn die Rauchpartikel dem Auftrieb der Temperatur folgen sollen, müssen zusätzlich die erzeugten Aufstiegskräfte addiert werden. *Fedkin et al.* schlagen in [7] folgendes Modell zur Berechnung der Kräfte vor.

$$\vec{f} = -\alpha \rho \vec{v} + \beta (T - T_{amb}) \vec{v}$$

Hierbei ist $\vec{v} = \{0, 1, 0\}$. Die Parameter α und β müssen so gewählt werden, dass sich ein realistisches Verhalten ergibt. Das Temperaturgitter wird vorher mit einer Umgebungstemperatur T_{amb} initialisiert. Wird nun eine externe Temperatur in das System gegeben, enstehen mit der obigen Gleichung Kräfte, die für einen Aufstieg sorgen. Ist die Temperatur kälter als T_{amb} fallen die Rauchpartikel nach unten. Partikel können sich also an kalten Hindernissen abkühlen und damit absinken.

5.3.3 Diffusion

Der zweite Schritt im Algorithmus, ist die Diffusion, die Verteilung der Rauchpartikel. Die Diffusion auf Geschwindigkeiten angewendet, bestimmt die Viskosität des Rauches. Da Gase und Rauch im Allgemeinen nur eine sehr geringe Viskosität besitzen, haben *Fedkin et al.* in [7] völlig auf den Diffusions-Term verzichtet. Man erhält die so genannten inkompressiblen *Euler* Gleichungen. Wird in DUSTY die Viskosität auf Null gesetzt, wird dieser Teil übersprungen. Das spart Rechenzeit, da in diesem Teil ein Gleichungssystem gelöst werden muss.
Wie in Abschnitt 4.3.1 hergeleitet wurde, muss folgendes Gleichungssystem für die Unbekannten D_{n+1} gelöst werden:

$$\begin{aligned} D_{n_{i,j,k}} = & D_{n+1_{i,j,k}} - \Delta t \kappa (D_{n+1_{i-1,j,k}} + D_{n+1_{i+1,j,k}} \\ & + D_{n+1_{i,j-1,k}} + D_{n+1_{i,j+1,k}} + D_{n+1_{i,j,k-1}} + D_{n+1_{i,j,k+1}} - 6 D_{n+1_{i,j,k}})/h^2 \end{aligned}$$

Das System ist nur schwach auf den Diagonalen besetzt (siehe Seite 20). Für DUSTY wurde deshalb ein numerisches Lösungsverfahren zur Bestimmung der Unbekannten benutzt. Konkret wird eine *Gauß-Seidel* Relaxation verwendet. Diese läßt sich in wenigen Zeilen realisieren und liefert brauchbare Ergebnisse. Für beste Qualität empfiehlt *Stam* in [7] die Verwendung eines konjugierten Gradienten Verfahren mit unvollständiger Choleski Zerlegung. Das CG-Verfahren beinhaltet mehrere Matrix-Multiplikationen und ist daher zeitaufwändiger. Im Anhang B findet sich eine Beschreibung dieses Verfahrens.

5.3.4 Transport

Im dritten Teil des Algorithmus erfolgt die Bewegung der Dichte durch die Vektoren der Geschwindigkeiten. Im Gegensatz zu den herkömmlichen Verfahren, die auch bei diesem Teil des Algorithmus auf Finite Differenzen setzen, verwendet *Stam* ein *semi-Lagrange* Verfahren. Eine ausführliche Erläuterung wurde in Sektion 4.3.2 auf Seite 31 gegeben. Für dieses Verfahren ist eine Partikel-Verfolgung notwendig. Als in Frage kommende Verfahren, wurden das *Euler* und das *Runge-Kutta* Verfahren angegeben. In DUSTY kann zwischen den beiden Verfahren gewählt werden. Die Berechnung mit dem *Euler*-Verfahren ist geringfügig schneller als mit dem *Runge-Kutta* Verfahren. Das der Unterschied nur gering ist, begründet sich damit, dass andere Teile des *Stable Fluids* Algorithmus einen größeren Einfluss auf die Gesamtrechenzeit haben. Das Verhalten der Strömung unterscheidet etwas zwischen den Verfahren, was auf die niedrige Genauigkeit des *Euler* Verfahren zurückzuführen ist. Dieses kann aber auch von dem Animator gewünscht sein, so das man

nicht sagen kann, welches Verfahren nun besser geignet. Welches Verfahren benutzt werden soll, kann über das Szenenskript festgelegt werden. Standardmäßig wird das *Euler* Verfahren benutzt.
Die Partikel-Verfolgung wird zunächst mit einem negativen Zeitschritt rückwärts ausgeführt, um zu bestimmen von welcher Position ein Partikel starten muss, damit es in einem Zellenmittelpunkt endet. Die gefundene Position wird sich im Allgemeinen nicht im Mittelpunkt einer Zelle befinden, aus dem man sofort die Geschwindigkeiten oder Dichte an der Stelle entnehmen könnte. Um die Geschwindigkeiten bzw. die Materie an der Startposition zu bestimmen, müssen diese aus den umgebenden Zellen mit einer trilinearen Interpolation berechnet werden (siehe Abbildung 4.7 auf Seite 33). Ausgehend von der berechneten Startposition im Gitter, werden die Positionen der acht umgebenden Zellenmittelpunkte ermittelt, mit denen eine trilineare Interpolation erfolgen kann. Man kann sich das so vorstellen, dass man einen Funktionswert an einer Position innerhalb eines Würfels sucht, bei dem die Funktionswerte an den Ecken bekannt sind. Wenn die Ecken mit V_{000} bis V_{111} bezeichnet sind, ergibt sich folgende Rechenvorschrift zur Bestimmung des Funktionswertes an der Position x, y, z:

$$
\begin{aligned}
V_{xyz} = &V_{000}(1-x)(1-y)(1-z)+ \\
&V_{100}x(1-y)(1-z)+ \\
&V_{010}(1-x)y(1-z)+ \\
&V_{001}(1-x)(1-y)z+ \\
&V_{101}x(1-y)z+ \\
&V_{011}(1-x)yz+ \\
&V_{110}xy(1-z)+ \\
&V_{111}xyz
\end{aligned}
$$

Der folgende Quellcode zeigt die Implementation für die u Komponente des Vektorfeldes in DUSTY.

```
c->u[idx] = q0*(s0*(t0*c->u0[idx000]+t1*c->u0[idx010])+
                s1*(t0*c->u0[idx100]+t1*c->u0[idx110]))+
                q1*(s0*(t0*c->u0[idx001]+t1*c->u0[idx011])+
                s1*(t0*c->u0[idx101]+t1*c->u0[idx111]))*diss;
```

Diese Interpolation muss für alle Komponenten des Geschwindigkeitsfeldes, des Dichtefeldes und des Temperaturfeldes erfolgen. Da immer die gleiche Startposition benötigt wird, wurden die Indexwerte der acht umgebenen Punkte, in den Variablen idx000–idx111 vorberechnet. Das Ergebnis der Interpolation wird in das neue Gitter eingetragen.
Als Optimierung erfolgt der *Dissipations*-Schritt an dieser Stelle mit der Multiplikation der Variablen *diss*. Diese Variable hat im Allgemeinen einen Wert kleiner als 1 und sorgt dafür, dass Geschwindigkeiten und Dichte im Laufe der Simulation weniger werden. Die Variable kann natürlich auch auf 1 gesetzt werden, womit die Dichte im Simulationsraum

ständig zunimmt und diesen mit der Zeit völlig ausfüllt.
Für eine maximale Qualität der Interpolation, schlagen *Fedkiw et al.* in [7] statt der linearen eine kubische Interpolation vor. Da Interpolationen höherer Ordnung instabil werden können, zeigen sie dort eine Lösung wie dieses vermieden werden kann. Kubische Interpolationen erfordern weiterhin einen größeren Rechenaufwand als lineare Interpolationen und sind deshalb für eine Echtzeitberechnung nicht geeignet. In [7] zeigen *Fedkiw et al.* an einem Beispiel, dass die Berechnung einer Szene mit kubischer Interpolation etwa 18 mal länger dauert.

5.3.5 Erzeugen der Divergenzfreiheit

Die *Navier-Stokes* Gleichungen fordern mit

$$\nabla \cdot \vec{\mathbf{v}} = 0$$

Divergenzfreiheit. Das heißt, dass eine Erhaltung der Masse gewährleistet werden muss. Es dürfen keine Quellen oder Senken entstehen. Der Zustand der Divergenzfreiheit ist nach den vorherigen Berechnungen zunächst nicht erfüllt. Ein Operator, der die Divergenzfreiheit wieder herstellt, wurde im Kapitel 4 zum *Stable-Fluids* Algorithmus entwickelt. Er basiert darauf, dass sich ein Vektorfeld $\vec{\mathbf{w}}$ in ein divergenzfreies Feld $\vec{\mathbf{v}}$ und ein Gradientenfeld zerlegen lässt.

$$\vec{\mathbf{w}} = \vec{\mathbf{v}} + \nabla q$$

Durch Subtraktion des Gradientenfeldes vom Vektorfeld $\vec{\mathbf{w}}$, lässt sich dann das divergenzfreie Feld erzeugen. Das Problem ist, dass das Gradientenfeld nicht bekannt ist. Wie in Kapitel 4 gezeigt wurde, kann das skalare Feld q durch Lösen der folgenden *Poisson*gleichung ermittelt werden.

$$\nabla \cdot \vec{\mathbf{w}} = \nabla^2 q$$

Das Vektorfeld ist gegeben und man muss q bestimmen. Die Gleichung lässt sich wieder als lineares Gleichungssystem schreiben. Es ist, wie bei der Diffusion, schwach besetzt. Deshalb wurde ebenfalls eine *Gauß-Seidel* Relaxation verwendet. So eine Relaxation funktioniert gerade bei groben Gittern sehr gut. Für eine Echtzeitsimulation werden solche weitmaschigen Gitter benutzt. Im *Stable-Fluids* Algorithmus werden die sonst auftretenden Instabilitäten vermieden.
Zum Bestimmen von q muss zunächst die Divergenz von $\vec{\mathbf{w}}$ berechnet werden. Das erfolgt mit zentralen Differenzen (siehe Kapitel 3).

$$\begin{aligned}(\nabla \cdot w)_{i,j,k} = (&w_{i+1,j,k} - w_{i-1,j,k} + \\ &w_{i,j+1,k} - w_{i,j-1,k} + \\ &w_{i,j,k+1} - w_{i,j,k-1})/2h\end{aligned} \tag{1}$$

Es muss darauf geachtet werden, dass für Zellen die von einem Objekt belegt sind, keine Divergenz berechnet wird und diese auf Null gesetzt wird. Nach der Berechnung der

Divergenz ist die linke Seite der Gleichung bekannt und es kann q durch Lösen des Gleichungssystems bestimmt werden. Die *Gauß-Seidel* Relaxation ist ein iteratives Verfahren. Vor jedem Iterationsschritt müssen die Bedingungen an den Rändern angepasst werden. Das heißt, dass die Zellen an den Rändern den Wert von den jeweiligen Nachbarzellen zugewiesen bekommen.
Nachdem das skalare Feld q bestimmt wurde, kann sein Gradient einfach von dem Vektorfeld subtrahiert werden. Man erhält das divergenzfreie Vektorfeld $\vec{\mathbf{v}}$.

5.3.6 Vorticity Confinement

Das *Vorticity Confinement* ist optional und nicht Bestandteil des originalen *Stable Fluids* Algorithmus. Mit ihm kann für ein besseres Hervortreten von Verwirbelungen gesorgt werden, da sie durch den *Stable Fluids* Algorithmus numerisch verloren gehen.
Als erstes muss die sogenannte *Rotation* (siehe Glossar) berechnet werden. Sie ist wie folgt definiert:

$$\vec{\omega} = \nabla \times \vec{\mathbf{v}}$$

In DUSTY wird das für jede Zelle in einer Schleife folgendermaßen berechnet:

```
tx = 0.5*(z[IX(i,j+1,k)]-z[IX(i,j-1,k)] - v[IX(i,j,k+1)]+v[IX(i,j,k-1)]);
ty = 0.5*(u[IX(i,j,k+1)]-u[IX(i,j,k-1)] - z[IX(i+1,j,k)]+z[IX(i-1,j,k)]);
tz = 0.5*(v[IX(i+1,j,k)]-v[IX(i-1,j,k)] - u[IX(i,j+1,k)]+u[IX(i,j-1,k)]);

  /* norm von w berechnen */
c->ws[idx] = sqrt(tx*tx+ty*ty+tz*tz);
c->ux[idx] = tx;
c->uy[idx] = ty;
c->uz[idx] = tz;
```

Im Feld `ws` wird die Norm $|\omega|$ gepeichert. Sie wird für die weiteren Berechnungen benötigt. Anschließend wird

$$\eta = \nabla|\omega|$$

mit folgendem Code berechnet

```
ex = 0.5*(c->ws[IX(i+1,j,k)]-c->ws[IX(i-1,j,k)]);
ey = 0.5*(c->ws[IX(i,j+1,k)]-c->ws[IX(i,j-1,k)]);
ez = 0.5*(c->ws[IX(i,j,k+1)]-c->ws[IX(i,j,k-1)]);
```

Danach kann

$$N = \frac{\eta}{|\eta|}$$

bestimmt werden. Schließlich wird

$$f_{conf} = \varepsilon(N \times \omega)$$

berechnet und zu dem aktuellen Geschwindigkeitsfeld addiert. Mit ε wird bestimmt, wie stark sich das *Vorticity Confinement* auswirken soll.

```
idx = IX(i,j,k);
u[idx] += (ny*c->uz[idx]-nz*c->uy[idx])*c->vor;
v[idx] += (nz*c->ux[idx]-nx*c->uz[idx])*c->vor;
z[idx] += (nx*c->uy[idx]-ny*c->ux[idx])*c->vor;
```

5.4 Skriptinterpreter

Ein wichtiger Teil in Dusty ist der Skriptinterpreter, der nun beschrieben werden soll. Szenen werden in Dusty mit einem Skript erstellt. Das Skript dient weiterhin zum Animieren einer Szene. Für Dusty konnte auf einen fertigen Skriptinterpreter zurückgegriffen werden. *TEKlib* bietet Support für den *LUA* Interpreter [32]. Dieser ist eine vollständige Skriptsprache, die man leicht um eigene Befehle erweitern kann. *LUA* wurde primär zum Erweitern von Applikationen entwickelt. Eine Einführung in die Programmierung von *LUA* findet man unter [2]. Weiterhin werden zu Dusty Beispielskripte geliefert, aus denen leicht eigene Skripte erstellt werden können.

5.4.1 Aufbau eins Skriptes

Ein einfaches Skript für Dusty sieht folgendermaßen aus:

```
1  -- create an new container
2  -- x, y, z, colormode, enable temperature
3  c = dusty.newcontainer(25,25,25,COLOR,0)
4
5  -- create sourceobject
6  -- x, y, z, name
7  s = c:newsource(12,1,12,"source1")
8  s:set_force(0,4,0)
9  s:set_dens(20,10,0)
```

Mit `--` werden Kommentare eingeleitet. Das Skript verfolgt einen Objektorientierten Ansatz. So muss als erstes in Zeile 3 ein Container erzeugt werden, auf den sich alle weiteren Operationen beziehen. Die ersten drei Argumente sind die Dimension, die der Container haben soll. Die Dimension wird hier in Voxel angegeben, also die Anzahl der Gitterzellen pro Achse. Als nächstes muss bestimmt werden, ob der Container im `COLOR` oder `MONO` Modus eingestellt wird. Das heißt, ob die Quellen farbige oder graue Materie emittieren. Das letzte Argument gibt an, ob die Temperatur berücksichtigt werden soll und muss für diesen Fall auf `USETMP` gesetzt werden.

Nachdem ein Container erzeugt wurde, können Quellen und Objekte der Szene zugefügt werden. In dem obigen Beispiel wird in Zeile 7 eine neue Quelle kreiert. Die ersten Argumenten bestimmen die Position im Container. Danach wird ein Name für die Quelle angegeben. Eine Quelle stellt wiederum verschiedene Methoden zur Verfügung. Die wichtigsten sind `set_force()` und `set_dens()`. In Zeile 8 werden der Quelle Geschwindigkeiten zugeordnet. Diese werden jeweils für die x, y, z Richtung angegeben. Eine Quelle dient

auch gleichzeitig zum Einbringen von Dichte in das Universum. Dies erfolgt in Zeile 9 mit dem Befehl `set_dens()`. Hier kann die Stärke in Form von R,G,B Werten angegeben werden. Eine komplette Übersicht der möglichen Befehle, wird in Anhang A gegeben.

Soll die Szene animiert werden, muss eine Schleife dem Skript zugefügt werden. In dieser muss mit `dusty.delay()` abgefragt werden, ob das Hauptprogram beendet wurde. Gleichzeitig kann damit eine Pause, in Sekunden, angegeben werden, welche angibt wie lange an der Stelle im Skript gewartet werden soll bevor es weiter abgearbeitet wird. Dieses erfolgt in dem folgenden Skriptauschnitt in Zeile 9. Die Pause wird mit der Variablen `DELAY` bestimmt.

```
1  -- main loop
2  ---------------
3  repeat
4
5  for i=4,20 do
6      s1:set_place(i,1,5)
7      dusty.dostep()
8      dusty.saveframe()
9      abort = dusty.delay(DELAY)
10     if abort then break end
11 end
12 until abort == true
```

Um die Animation im Skript mit dem Hauptprogram zu synchronisieren, kann mit `dusty.dostep()` (Zeile 7) explizit der Schritt zur Berechnung des neuen Zustandes ausgeführt werden. Normalerweise ruft Dusty intern den *Navier-Stokes* Solver auf. Dies kann unterbunden werden, indem am Anfang eines Skriptes `dusty.internstep(false)` aufgerufen wird. Damit wird der interne Schritt deaktiviert und kann nun von dem Skript aus aufgerufen werden. Möchte man Bilder der Animation abspeichern, wird mit `dusty.saveframe()` das jeweils aktuelle Bild gespeichert. Aus den Einzelbildern kann dann z.B. einen MPEG Film erzeugt werden.

5.4.2 Implementation in Dusty

Nachdem Dusty gestartet wurde, wird ein, eventuell als Argument übergebenes, Skript geladen und der Interpreter als eigener Thread gestartet. Das heißt, er läuft völlig eigenständig von Dusty.
Damit es zu keinen Problemen bei Zugriffen auf gemeinsame Speicherstellen kommt, ist der Einsatz von *Semaphoren* und Flags notwendig. Das bedeutet, dass der Zugriff auf Speicher exklusiv angefordert werden muss. *TEKlib* bietet die Befehle `TLock()` und `TUnlock()`, um das zu realisieren. Mit `TLock()` wird gewartet, bis eine vorher bestimmte *Semaphore* frei wird. Dann kann auf den gemeinsamen Speicher zugegriffen werden. Ist der Zugriff beendet, muss die *Semaphore* mit `TUlock()` wieder frei gegeben. Eine *Semaphore* muss

vorher mit `TCreateLock()` bereit gestellt werden. Der folgende Quellcode der Implementation des neuen *LUA* Befehls `set_visc()` zeigt die Anwendung dieser Vorgehensweise. Der Befehl setzt eine neue, im Skript angegebene Viskosität.

```
1  /*
2  **   con:set_visc(float)
3  **   set viscosity
4  */
5  static LUACFUNC TINT con_setvisc(lua_State *L)
6  {
7    TAPTR *o = (TAPTR *)luaL_checkudata(L, 1, "Container*");
8    container *c;
9    TINT numargs;
10
11   numargs = lua_gettop(L); /* number of arguments */
12
13   if (o == TNULL) luaL_argerror(L, 1, "bad handle");
14
15   c = (container *)*o;
16
17   TLock(dw.exec, dw.lock);
18   c->visc = (TFLOAT)lua_tonumber(L, 2);
19   dw.flags |= GUIDIRTY;
20   TUnlock(dw.exec, dw.lock);
21
22  return 0;
23 }
```

In Zeile 7 wird zunächst das Objekt, auf das die Methode `set_visc()` angewendet werden soll, von dem LUA-Stack geholt werden. In diesem Fall kann es nur ein Container sein. Um jetzt die neue Viskosität in die Container-Struktur zu schreiben, wird in Zeile 17 der Exklusivzugriff angefordert. Danach kann in Zeile 18 der neue Wert in die Container-Struktur eingetragen werden. Da der Viskositätswert in der GUI angezeigt wird, muss diese neu gezeichnet werden. Damit das Hauptprogramm dieses tut, wird ihm über das Flag `GUIDIRTY` in Zeile 19 mitgeteilt, dass es eine Änderung gab. Nachdem das Hauptprogramm die Benutzungsschnittstelle angepasst hat, wird dieses Flag von ihm wieder gelöscht. Abschließend wird die *Semaphore* in Zeile 20 mit `TUlock()` wieder frei gegeben. Nach diesem Schema sind alle Befehle von DUSTY implementiert worden.

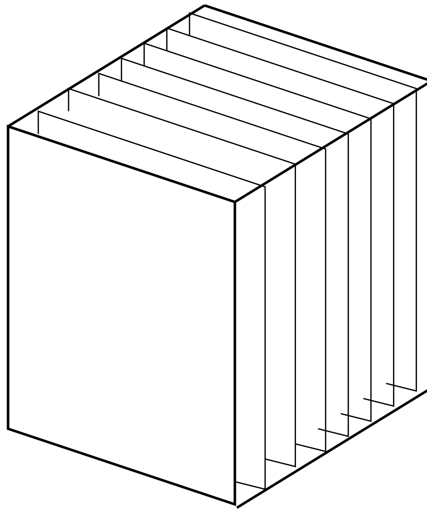

Abbildung 5.2: Unterteilung des Universums in 2D Gitter

5.5 Grafische Ausgabe

Bis jetzt wurde nur auf die Berechnung des Geschwindigkeitsfeldes und der Dichte eingegangen. Jetzt soll die Darstellung des Rauches beschrieben werden. Dazu kommen in Dusty verschiedene Verfahren zur Anwendung. Mit einem Volumenrendering Verfahren und einem Partikelsystem kann der Verlauf der Dichte visualisiert werden. Mittels *Stromlinien* kann das Geschwindigkeitsfeld sichtbar gemacht werden.

5.5.1 Volumenrendering

Prinzipiell gibt es zwei Möglichkeiten das Volumen zu visualisieren. Entweder erfolgt die Visualisierung direkt über die Hardware oder über Software durch Raytracing. *Fedkiw et al.* zeigen in [7] die Visualisierung mittels Photonmaps. Damit können qualitativ hochwertige Bilder erzeugt werden.
Für Dusty sollte eine Echtzeitdarstellung möglich sein. Deshalb erfolgt die Visualisierung mit *OpenGL*. Dazu wird das Universum zunächst in Scheiben aus 2D-Gittern unterteilt. Siehe Abbildung 5.2. Die Unterteilung erfolgt entlang der Achse, die der View Achse am dichtesten ist. Dies erfolgt durch Auswertung der z Komponenten der Rotationsmatrix. Beim Drehen des Universums muss die Unterteilungsrichtung schließlich angepasst werden. Danach werden in jeder Schicht `GL_QUADS` im `BLEND` Modus gezeichnet. Dies sind Rechtecke deren Eckpunkte vier Zellenmittelpunkte bilden. Die Grafikhardware sorgt dann für das Blending also das Vermischen der überlagerten QUADS. Dazu muss für jeden Eckpunkt ein Alphawert angegeben werden, der sich aus der Dichte an der jeweiligen Position ergibt. Jeder Eckpunkt bekommt damit einen anderen Farb- und Alphawert aus denen Farbverläufe durch die Hardware interpoliert werden.
Dieses Vorgehen liefert bereits gute Bilder um den Rauchverlauf zu beobachten. Nachtei-

lig ist hier, dass man bei einem ungünstigen Drehwinkel des Universums die Schichten erkennen kann. Für höchste Qualität muss also Raytracing benutzt werden.

5.5.2 Partikel

Eine weitere interessante Möglichkeit der Visualisierung von Strömungen, ist die Verwendung von Partikel. Abbildung D.2 auf Seite 80 zeigt ein Beispiel für die Verwendung von Partikel. Partikel werden in einer verketteten Liste verwaltet. Jedes Partikel hat als Attribute die aktuelle Position und die Zeit die es noch zu leben hat. Die Zeit wird in `ttl` gespeichert und nach jedem Simulationsschritt verringert. Ist der Wert Null, wird das Partikel aus der Liste gelöscht. Die Struktur eines Partikel sieht folgendermaßen aus:

```
typedef struct _par
{
  THNDL handle;
  TBOOL linked;
  TFLOAT x,y,z;
  TINT   ttl;    /* time to life */
}particle;
```

In `handle` befinden sich die Strukturen zur Verwaltung der Verketteten Liste. Um ein Partikel zu bewegen, müssen die Geschwindigkeitswerte an der Postion des Partikels ermittelt werden. Da sich die Partikel auch zwischen Gitterzellen befinden können, müssen diese Werte trilinear interpoliert werden. Hier wird das gleiche Verfahren benutzt wie es auf Seite 47 für den Transportteil beim Lösen der *Navier-Stokes* Gleichungen beschrieben wurde. Wurden die Geschwindigkeiten gefunden, kann die neue Position $\vec{x}_{n+1}$ des Partikels wie folgt gefunden werden:

$$\vec{x}_{n+1} = \vec{x}_n + dt \cdot \vec{v}$$

Dies entspricht also der *Euler*-Integration.

5.5.3 Stromlinien

Die Geschwindigkeiten können mit sogenannten Stromlinien visualisiert werden. Dabei wird die Strömung ausgehend von einer Gitterzelle, über eine gewisse Zeit verfolgt und als Linie dargestellt. Dabei handelt es sich im Prinzip um eine Partikelverfolgung mit *Euler* Integration. In Dusty wird eine Stromlinie durch ein `GL_LINE_STRIP` dargestellt. Ausgehend vom Geschwindigkeitswert einer Gitterzelle wird, wie bei der Partikeldarstellung, ein imaginäres Partikel über vier Stufen verfolgt. Diese Stufen ergeben die Segmente für ein `GL_LINE_STRIP`. Dadurch erhält man Linien, die dem Geschwindigkeitsfeld folgen. Abbildung D.8 zeigt ein Beispiel für die Stromlinien.

5.6 Benutzungsschnittstelle

Abbildung 5.3 zeigt die Benutzungsschnittstelle von Dusty nach dem Starten. Sie wurde mit der C++ Bibliothek **FLTK** [4] implementiert und konnte damit portabel gehalten werden. Es wurde darauf geachtet, dass sie weitgehend selbsterklärend ist und trotzdem

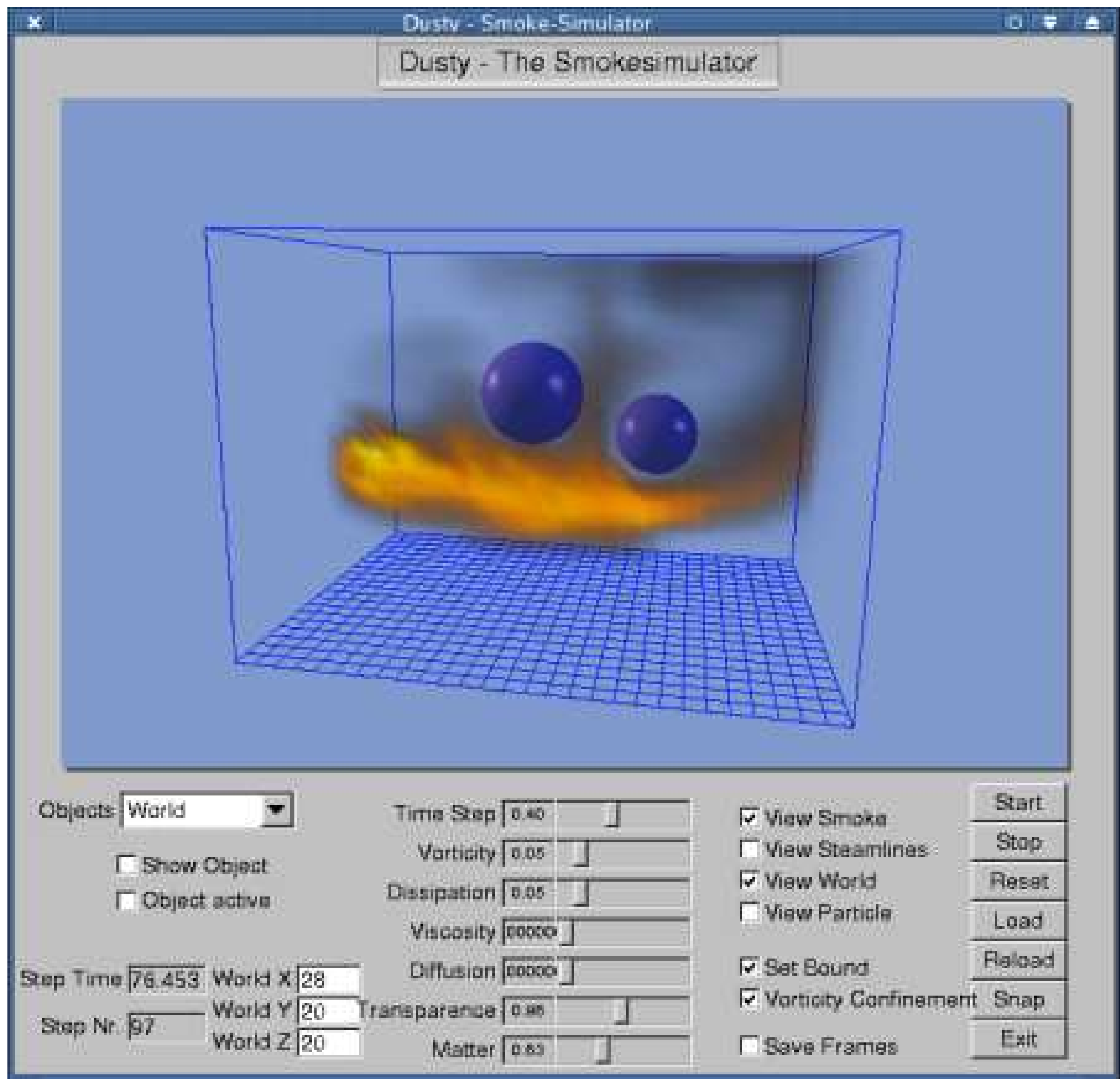

Abbildung 5.3: Dusty

alle nötigen Bedienelemente enthält. Rechts findet man die Elemente zum Starten und Stoppen der Simulation bzw. zum Laden von Szenen. Wird DUSTY ohne Angabe eines Szenenskripts als Argument gestartet, wird zunächst eine Defaultszene erzeugt. So kann man schnell erste Versuche mit DUSTY unternehmen. Durch Anklicken des *Start* Buttons wird die Simulation begonnen. Sie kann mit *Stop* jederzeit angehalten werden. Die Simulation spielt sich in einem Universum ab, welches als Quader dargestellt wird. Mit der Maus kann das Universum frei im Raum gedreht werden. Dazu wurde der Trackball-Algorithmus von *Crawfis* [5] implementiert. Mit *Reset* kann wieder der Startzustand der Szene hergestellt werden. Das heißt, das gesamte Universum bzw. das Vektorfeld werden mit Null initialisiert. *Reload* lädt die Szene erneut ein, so dass eventuell verschobene Objekte wieder ihre Anfangsposition erhalten. Mit dem *Snap* Button kann schließlich ein Screenshot gemacht werden. Es wird ein Bild der aktuellen Szene in das aktuelle Verzeichnis gespeichert.
Die kontroll Buttons, links daneben, legen fest, was dargestellt wird. So kann bestimmt werden, ob das Universum angezeigt wird oder ob der Strömungsverlauf durch *Strömungslinien* visualisiert werden soll. Weiterhin kann das *Vorticity Confinement* aktiviert werden und ob während der Simulation die aktuell angezeigten Frames gespeichert werden sollen. Diese erhalten dann als Namenszusatz die jeweilige Framenummer. Dadurch können aus den einzelnen Frames leicht Animationen, z.B. als MPEG, erzeugt werden.
Mit den Reglern in der Mitte kann interaktiv auf Simulationsparameter wie Zeitschritt oder *Dissipation* zugegriffen werden, um die Auswirkung der Änderung sofort zu sehen. So können leicht Parameter gefunden werden, mit denen eine Szene gut aussieht.
Links befinden sich die Elemente zum Verändern der Szene. Zunächst gibt es einen *Objects*-Listbutton mit dem ein Objekt in der Szene ausgewählt werden kann. Danach kann es mit der Maus bei gedrückter rechter Maustaste verschoben werden. Das angewählte Objekt ändert seine Farbe dabei nach Grün, so dass es leicht in der Szene zu erkennen ist. Darunter kann die Dimension des Universums verändert werden. Die Änderung erfolgt nach der Eingabe der neuen Größen. Der größte der drei Werte, wird zum Unterteilen eines Einheitswürfel mit der Kantenlänge Eins, benutzt. Damit wird die Größe einer Zelle festgelegt. Eine größere Anzahl von Zellen sorgt also dafür, dass die Auflösung feiner wird. Die Zeit, zur Berechnung einer Szene, steigt dafür mit einer höheren Auflösung. Links unten wird die Zeit zum Berechnen einer Szene in Millisekunden, sowie die aktuelle Bildnummer ausgegeben. DUSTY kann mit dem *Exit* Button oder durch Drücken der Escape Taste beendet werden.

Kapitel 6
Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde ein Animationssystem zur Simulation von Rauch und Feuer entworfen und implementiert. Grundlage ist der *Stable Fluids* Algorithmus von *Stam*, der erstmals 1999 in [27] veröffentlicht wurde. Mit diesem Algorithmus ist es möglich die *Navier-Stokes* Gleichungen, mit einem groben Diskretisierungsgitter und relativ großen Zeitschritten, in Echtzeit zu lösen. Zum Verständnis wurde in der vorliegenden Arbeit ein Überblick zur Strömungsmechanik gegeben. Es wurden die *Navier-Stokes* Gleichungen hergeleitet und auf Lösungsmöglichkeiten eingegangen. Dabei wurde insbesondere das Finite Differenzen Verfahren, zum Diskretisieren der Gleichungen, ausführlich erläutert. Anschließend wurde umfassend auf den *Stable Fluids* Algorithmus eingegangen und seine Funktionsweise beschrieben. Die Umsetzung des Algorithmus erfolgte im Animationssystem Dusty, das in Kapitel 5 vorgestellt wurde. Zusätzlich wurde ein Skriptsystem entworfen, mit dem Szenen erstellt und animiert werden können.

6.1 Ergebnisse

Der *Stable Fluids* Algorithmus basiert auf den, in der Strömungsmechanik benutzten, *Navier-Stokes* Gleichungen. Eine genaue Lösung ist im Allgemeinen sehr Zeitaufwändig. Der Algorithmus von *Stam* erlaubt die Verwendung eines groben Diskretisierungsgitters und relativ große Zeitschritte ohne das der Algorithmus instabil wird. Hierdurch wird eine Simulation in Echtzeit möglich. Statt der benutzen Finite Differenzen Methode, wird in der Strömungsmechanik oft die Finite Elemente oder Finite Volumen Methode zum Diskretisieren der Gleichungen benutzt. Weiterhin muss ein feineres Gitter und ein kleinere Zeitschritt benutzt werden. Das alles garantiert exakte Ergebnisse wird aber mit einem größeren Rechenaufwand erkauft. Die Verwendung großer Zeitschritte (z.B. 0.5 statt 0.05) und eines *semi Lagrange* Verfahren beim *Stable Fluids* Algorithmus sorgen für eine schnelle Ausführung aber auch für eine numerische Dissipation, das heißt, dass das die Bewegung des Fluids schneller gedämpft wird als es in der Realität geschehen würde. Dieses kann bei der Simulation von Rauch in der Computergraphik im Allgemeinen vernachlässigt werden, da oft externe Kräfte für eine ständige Bewegung sorgen.

In dieser Diplomarbeit wurde ein Animationssystem, basierend auf dem *Stable Fluids* [27] Algorithmus, entworfen und implementiert. Mit dem System Dusty können in Echtzeit

Abbildung 6.1: Ein Teelicht mit simulierter Flamme

die drei-dimensionalen *Navier-Stokes* Gleichungen auf einem Standard-PC gelöst werden. Diese Gleichungen bilden die Grundlage einer Stömungssimulation, die hier zum Erzeugen von Feuer und Rauch benutzt wurden. In das Simulationsgebiet können beliebig viele Hindernisse eingefügt werden, die den Strömungsverlauf beeinflussen.

Der Grund, warum mit dem *Stable Fluids* Algorithmus eine Echtzeitsimulation möglich ist, besteht darin, dass der Algorithmus grobe Gitter und große Zeitschritte erlaubt. Die implementierte Visualisierung mit OpenGL erzeugt auch bei groben Gittern relativ weiche Rauchverläufe (Abbildung 5.3). Wenn das Gitter allerdings zu grob gewählt wurde, werden die Zellen als Strukturen sichtbar.

Zum Erstellen und Animieren von Szenen wurde in Dusty ein Skriptsystem integriert. Hierzu wurde die Skriptsprache *LUA* [32] verwendet, die um eigene Dusty spezifische Befehle erweitert wurde. Mit diesem Befehlssatz (siehe Anhang A) können schnell Szenen erstellt und animiert werden. Da das Skriptsystem eine vollständige Sprache mit Kontrollkonstrukten wie `if` oder `while` enthält und eine Änderung aller Parameter von Dusty über das Skipt zur Laufzeit möglich ist, können komplexe Skripte zur Animation erstellt werden. Eine Erweiterung, um zusätzliche Befehle ist im Quellcode von Dusty mit wenigen Zeilen möglich.

Anhand der folgenden Beispiele soll gezeigt werden, dass mit Dusty durchaus eine realitätsnahe Simulation möglich ist.

Da in Dusty eine Simulation und Anzeige in Echtzeit erfolgen sollte, wurde ein Hardware Volumenrendering mittels OpenGL benutzt. Dieses Verfahren liefert Ergebnisse die sogar für kleine Filmeffekte benutzt werden können. Abbildung 6.1 zeigt ein Bild aus einer erstellten Filmsequenz. Hier wurde eine von Dusty erzeugte Flamme auf ein Teelicht montiert.

Nr.	Gitterdimension	Anzahl Zellen	Athlon 1.2Ghz	P4 2.66Ghz
1	20x14x14	3920	12ms	11ms
2	28x20x20	11200	78ms	53ms
3	35x25x25	21875	215ms	153ms
4	40x30x30	36000	510ms	335ms
5	50x37x37	68450	1170ms	850ms
6	65x45x45	131625	2660ms	2000ms
7	70x52x52	189280	4800ms	3500ms
8	80x60x60	288000	8700ms	5400ms

Tabelle 6.1: Simulationszeit für verschiedene Auflösungen des Universums

Dazu musste zunächst eine Animation der Flamme aus Einzelbildern erstellt werden. Um die Bilder für Filmsequenzen zu benutzen, wird zusätzlich ein Alphakanal erzeugt, welcher die Transparenz der Flamme angibt. Dieses wird von DUSTY unterstützt und kann im Szenenskript eingeschaltet werden. Abbildung D.6 zeigt einen solchen Alphakanal am Beispiel eines Feuers. Das Teelicht wurde zunächst mit einer Videokamera einige Sekunden aufgezeichnet. Mit einer Filmkomposing Software wurde die Bewegung des Teelicht, welche durch das Wackeln der Kamera entstand, getrackt. Anhand der Bewegungsvektoren montierte die Software anschließend die Animation der Flamme an die vorgesehende Stelle am Teelicht. Die Rechenzeit zur Simulation des Feuers in Abbildung D.6, betrug ca. 80ms pro Bild bzw. etwa 12 Bilder pro Sekunde auf einem Rechner mit 1.2 GHz.

Die Einbringung von Hindernissen in eine Szene wird in Abbildung 5.3 auf Seite 41 gezeigt. In Abbildung 5.3 steigt der Rauch aufgrund seiner Temperatur nach oben und umfließt die Kugeln. Interaktiv kann in die Szene eingegriffen werden, indem die Kugeln während der Simulation bewegt werden können. Diese Bewegungen beeinflussen sofort den Verlauf der Strömungen in der Art, dass der Rauch mit der Bewegung transportiert wird. Hier betrug die Rechenzeit pro Bild etwa 78ms, was etwa 13 Bilder pro Sekunde entspricht. Mit solchen Zeiten ist es noch möglich, in Echtzeit die erwähnten Bewegungen von Hindernissen von einem Benutzer ausführen zu lassen.
Die Zeit zur Berechnung einer Szene hängt im Wesentlichen von der Dimension des Universums ab. Diese Zeit steigt sehr schnell an. In Tabelle 6.1 wurden die Zeiten für verschiedene Auflösungen der Szene aus Bild 5.3 aufgelistet. Die Zellenzahl steigt kubisch an. Das heißt, dass bei einer Verdopplung der Gitterdimension sich die Anzahl der Zellen im Universum verachtfacht. Die Rechenzeit dagegen verhält sich anders. Sie steigt schneller an, z.B. bei der Verdopplung der Ausmaße von Test Nr. 1 zu Nr. 4 steigt sie um das 42fache an. Dieser Anstieg nimmt bei höheren Dimensionen ab. So steigt die Rechenzeit bei Verdopplung der Gittergröße von Test 4 auf Test 8 nur noch um das 17fache an. Abbildung 6.2 zeigt diesen Verlauf als Kurve.

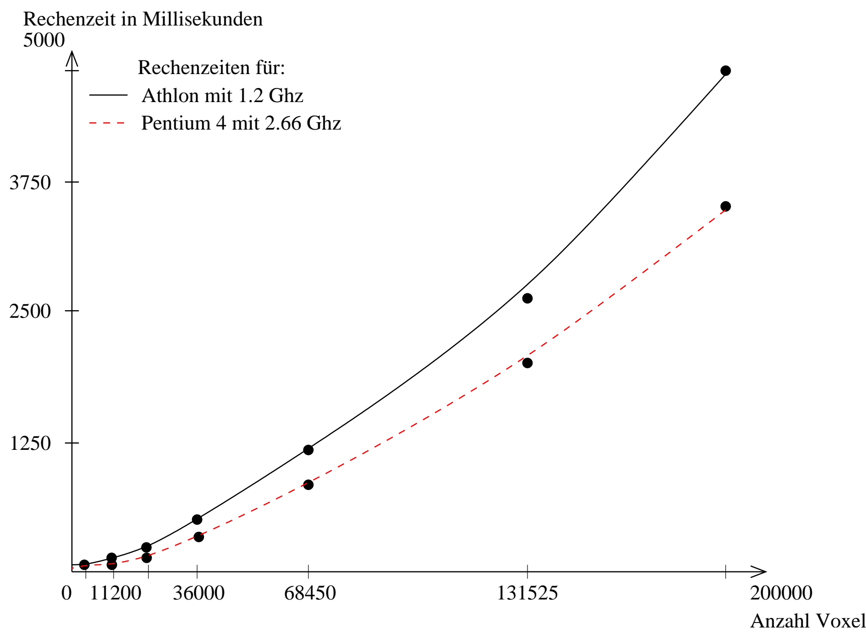

Abbildung 6.2: Graph zur Tabelle 6.1

Ein komplexerer Verlauf einer Strömung wird in Abbildung D.7 gezeigt. Man erkennt wie der Rauch jede Nische ausfüllt. Oben links wurde ein Loch in die Wand modelliert durch welches der Rauch erwartungsgemäß entweicht.
Eine Erkenntnis beim Experimentieren mit komplexen Szenen ist, dass die Dauer der Simulation pro Zeitschritt im Wesentlichen von der Dimension des Universums und nicht von der Anzahl der Objekte abhängt. Das ist darin begründet, dass zur Beachtung aller Hindernisse immer sämtliche Voxel des in Abschnitt 5.3.1 eingeführten Feldes *bounds* durchlaufen werden. Dieses Feld wird nur einmal vor Beginn der Simulation oder wenn sich die Position von Objekten verändert erstellt. Es wird getestet, ob ein Voxel von einem Hinderniss ausgefüllt wird. Die nötige Behandlung dieses Falles erfordert nur wenig Aufwand und fällt in der Gesamtzeit nicht in das Gewicht. Eine Untersuchung mit einem Profiler ergab, dass 41% der Rechenzeit für den Projektschritt, also die Herstellung der Divergenzfreiheit, zusammen mit dem *Vorticity Confinement* benötigt wird. Dicht darauf folgte der Transportschritt mit 39%.

Als Beispiel für eine Animation einer Quelle dient die Bilderfolge D.4. Hier sorgt das Szenenskript für die Bewegung der Quelle. In dem Skript wurden zwei Schleifen programmiert, die für diese Bewegungen verantwortlich sind. Da durch einfache Befehle das Skript leicht zu programmieren ist, können so beliebige Bewegungen erzeugt werden. Eine Erläuterung der von Dusty zur Verfügung gestellten Befehle findet sich in Anhang A. Eine allgemeine Einführung zur Skriptsprache wird in [2] gegeben. Eine Bewegung von Objekten mittels einer einfache Schleife, wurde in Abschnitt 5.4.1 gezeigt.

Abbildung 6.3: Zu grobes Gitter beim Zeichnen eines Logos (60x60x5)

Für die Abbildungen D.5 wurden die Zeichenroutinen von Dusty zum Erzeugen des Logos des Instituts für Computergraphik der Uni-Rostock benutzt. Dieses besteht aus Kurven und Geraden. Mit diesen Routinen kann Materie mit einer bestimmten Temperatur in das Universum gezeichnet werden. Beim Start der Simulation verdampft quasi die Materie und löst sich auf. Das ist sehr gut in der Bilderfolge zu erkennen. Diese sind Einzelbilder aus einer erzeugten Videosequenz. Zum Generieren des Films wurden die einzelnen Bilder der Simulation mit Dusty gespeichert und mit dem freien MPEG Encoder *mpeg2enc* [3] zu einer Sequenz konvertiert. Die Auflösung wurde mit 120x120x5 relativ hoch gewählt, damit die Zeichnung nicht zu grob wird und die Zellen des Gitters sichtbar werden (Abbildung 6.3). Die Simulationszeit betrug ca. 800ms pro Bild. Da hier nur eine Simulation im zwei dimensionalen gefordert ist, konnte die Tiefe des Universums klein gewählt werden und damit die Rechenzeit relativ niedrig gehalten werden. Zum Erstellen der Bilder für eine Animation ist allerdings auch keine Berechnung in Echtzeit nötig.

Als zusätzliche Möglichkeit zur Visualisierung der Strömungen, wurde ein Partikelsystem in Dusty integriert. Abbildung D.2 zeigt ein Beispiel mit dieser Möglichkeit zur Visualisierung der Strömung. Mit dem Partikelsystem kann der Verlauf der Strömung besser als mit Rauch beobachtet werden, da der Weg einzelner Partikel verfolgt werden kann. Es können beliebig viele Partikelquellen in einem Universum verteilt werden. So kann man den Strömungsverlauf für jede Position im Universum genau untersuchen.
Eine weitere Möglichkeit zur Strömungsvisualisierung bietet Dusty mit der Anzeige von Stromlinien. Mit ihnen wird der Verlauf der Strömung durch Linien dargestellt. Abbildung D.8 zeigt ein Beispiel für diese Visualierungsart. Die Stromlinien können während der Simulation beliebig zu- oder abgeschaltet werden.

6.2 Ausblick

Mit Dusty wurde ein flexibles Grundgerüst zum Experimentieren mit Fluids in der Computergraphik geschaffen. Dieses kann jedoch noch ausgebaut werden. Der erste Punkt ist die Darstellung des erzeugten Rauches. Das Hardwarerendering und die Visualisierung mit Partikeln geben bereits einen guten Einblick in die Strömungsmechanik. Die Ergebnisse sind aber nur bedingt für professionelle Videoproduktionen geeignet. Um die Qualität der

Abbildung 6.4: Simulation von Rauch mit Raytracing und Photonmapping (aus [7])

Bilder zu verbessern, sind Verfahren wie Raytracing und Photonenmapping notwendig. Mit diesen Verfahren kann die Streuung des Lichtes durch den Rauch berechnet werden. Zusätzlich ist mit Raytracing das Erzeugen von Schatten möglich. In [7] verwendeten *Fedkiw et al.* Photonenmapping nach *Jensen* [12] zum Rendern des simulierten Rauches. Damit ist eine qualitativ hochwertige Produktion möglich. Ein Verfolgen der Simulation in Echtzeit ist dann natürlich nicht mehr möglich. Abbildung 6.4 zeigt ein Beispiel für, von *Fedkiw* erzeugten, Rauch mit Photonenmapping.

Beim Betrachten der Messwerte aus Tabelle 6.1, stellt sich die Frage, ob es nicht möglich ist, ein großes Universum aus z.B. vier kleineren zu bilden. Eine Verdopplung der Maße von z.B. Test 3 würde dann theoretisch $4 \cdot 215 = 860$ms statt, wie in Test 7, 4800ms benötigen. Man hätte also eine über fünf mal schnellere Berechnung der Szene.

Wünschenswert wäre es auch, beliebige Polygonobjekte als Hindernisse zu verwenden. Dazu sind Routinen nötig die diese Objekte in Voxel konvertieren können. Weiterhin wird das Berechnen der Randbedingungen an den Objekträndern erschwert.
Die Simulation in Dusty ist sehr allgemein gehalten. Um spezielle Effekte besser erzeugen zu können, wurde der *Stable Fluids* Algorithmus mehrfach erweitert. In Dusty ist das Erzeugen von Feuer durch eine geeignete Farbwahl der zugefügten Dichte möglich. *Fedkiw et al.* beschreiben in [22] eine genauere Simulation von Feuer. Grundlage ist dabei wieder der *Stable Fluids* Algorithmus. Sie modellieren dort eine Flamme aus mehreren Teilen die

unterschiedliche physikalische Eigenschaften haben. Zum Beispiel wird der bläuliche Kern einer Kerzenflamme einzeln simuliert.

Interessant ist die Simulation von Explosionen. Für DUSTY wurden die inkompressiblen *Navier-Stokes* Gleichungen verwendet. Der Effekt der Viskosität ist unbedeutend bei Gasen die mit groben Gittern, wie beim *Stable Fluids* Algorithmus, simuliert werden, da die numerische Dissipation diese überlagert. Sind die Geschwindigkeiten des Rauches weit weniger als die Schallgeschwindigkeit können kompressible Effekte ebenfalls vernachlässigt werden und es können die Gleichungen für inkompressibles Fluid verwendet werden. Das vereinfacht die Berechnung erheblich. Anders sieht das bei Explosionen aus. Hier müssen die kompressiblen *Navier-Stokes* Gleichungen benutzt werden. *Yngve et al.* zeigen in [35] die Verwendung dieser Gleichungen zum Animieren von Explosionen. Eine Darstellung von Explosionen mit Partikeln geben *Feldman et al.* in [14].

Wie gezeigt, steigt mit der Größe des Simulationsgitters die Rechenzeit schnell an. Um Explosionen mit einem Gitter von 2000x2000x2000 effizient berechnen zu können, entwickelten *Rasmussen et al.* [19] ein Verfahren, um nur die zwei dimensionalen *Navier-Stokes* Gleichungen lösen zu müssen. Aus den berechneten Werten konnte durch Interpolation ein 3D Bild erzeugt werden. In diesem Fall reduzierte sich die Rechenzeit um den Faktor 2000. Die erzeugten Atomexplosionen wurden für dem Film *Terminator 3* verwendet.

Ein weiteres Anwendungsfeld des (modifizierten) *Stable Fluids* Algorithmus ist die Simulation von Flüssigkeiten. *Foster et al.* zeigen in [8] diese Simulation. Dort werden die inkompressiblen *Navier-Stokes* Gleichungen benutzt. Die bei *semi-Lagrange* Verfahren üblichen großen Zeitschritte sorgen allerdings für eine numerische Dissipation welche für Gase visuell noch akzeptabel sind, dagegen bei Flüssigkeiten störend wirken. Es entsteht ein Rauschen. Deshalb verwendeten sie für den Transport der Flüssigkeitspartikel Methoden die den CLF Bedigungen (siehe Glossar) genügen. Für die Berechnung der Geschwindigkeiten wurden dagegen wieder *semi-Lagrange* Verfahren benutzt. Sie verwendeten ihr Verfahren zum Simulieren von Milch und Schlamm in dem Animationsfilm *Shrek*.

Die größte, noch anstehende, Aufgabe wurde vom *Clay Mathematics Institute* of Cambridge, Massachusetts ausgeschrieben. In Anlehnung an die von David Hilbert im Jahr 1900 aufgestellten größten Mathematischen Probleme, wurden 100 Jahre später am 24.5.2000 sieben neue Probleme aufgestellt. Eine der Aufgaben ist die Erforschung einer genauen Theorie zum exakten Lösen der *Navier-Stokes* Gleichungen. Das ist bis jetzt nicht möglich und man behilft sich, wie auch in dieser Arbeit, mit Annäherungen durch Approximation. Als Anreiz sind für die Lösung dieses Problems 1 Million Dollar ausgeschrieben.

Anhang A
Befehlssatz von Dusty

In dieser Sektion wird der, für Dusty erweiterte, Befehlssatz des Interpreters *LUA* vorgestellt. Die Befehle sind Methoden die auf ein bestimmes Objekt angewendet werden. Sie können in drei Typen eingeteilt werden.

1. Globale Methoden
2. Methoden für Container
3. Methoden für Objekte und Quellen

A.1 Globale Methoden

Globale Methoden haben die Form `dusty.befehl()`. Sie sind Methoden der Instanz `dusty`. Diese steht dem Skript sofort zur Verfügung.

---

abort

`dusty.abort()`

Funktion:
Erzwingt eine Beendigung des Skriptinterpreters

---

delay

`abort = dusty.delay(time)`

time	float Wert für Zeit in Sekunden

Funktion:
Festlegen der Zeit die das Skript warten soll, bis es weiter abgearbeitet wird.

---

dostep

`dusty.dostep()`

Funktion:
Fordert explizit die Ausführung eines Simulationsschrittes an. Vorher muss `interstep` auf `false` gesetzt werden

---

internstep

```
dusty.interstep(step)
```

step	`true` oder `false`

Funktion:
Hiermit wird festgelegt, ob ein Simulationsschritt Dusty-Intern erfolgt oder explizit in einem Skript aufgerufen wird.

---

newcontainer

```
container = dusty.newcontainer(x, y, z, colormode, usetemp)
```

x,y,z	Dimension des Containers in Zellen als Integerwerte
colormode	`COLOR` oder `MONO`
usetemp	`USETEMP` wenn die Temperatur beachtet werden soll, sonst null

Funktion:
Erzeugt einen neuen Container (Universum), in dem eine Szene aufgebaut werden kann. Zur Zeit kann nur ein Container verwendet werden.

---

saveframe

```
dusty.saveframe()
```

Funktion:
Speichert das aktuelle Bild

---

voxelize

```
dusty.voxelize()
```

Funktion: Führt explizit dazu, das die Szene in Voxel umgewandelt wird. Das heißt, Hindernisse werden der Szene bekannt gemacht.

A.2 Methoden für Container

Die Methoden für einen Container setzen vorraus, dass vorher mit `dusty.newcontainer()` ein solcher erzeugt wurde. Die nachfolgenden Methoden werden mit `con:methode()` aufgerufen. `con` ist ein vorher erzeugter Container.

---

drawcurve

`con:drawcurve(p1,p2,p3,p4,seg,r,g,b,t)`

p1–p4	Punkte aus denen die Bézierkurve gezeichent werden soll.
seg	Anzahl der Segmente der Kurve
r,g,b	Farbe der Kurve
t	Temperatur

Funktion:
Zeichnet eine Béziercurve aus den vier Punkten p1–p4. Ein Punkt wird folgendermaßen im Skript definiert: `p = {x,y,z}`. Es sind nur Ganzzahlwerte zugelassen. Die Koordinaten beziehen sich auf die Zellen des Containers.

---

drawline

`con:drawline(x1,y1,z1,x2,y2,z2,r,g,b,t)`

x1–z2	Anfangs- und Endpunkt der Linie. Koordinaten sind Ganzzahlig und beziehen sich auf die Zellen des Containers
r,g,b	Farbe der Linie
t	Temperatur

Funktion: Zeichnet eine Linie mit der angebenen Farbe.

---

enable_bound

`con:enable_bound(bound)`

bound	`true` oder `false`

Funktion: Legt fest, ob der Rand des Containers berücksichtigt werden soll.

---

enable_vort

`con:enable_vort(vort)`

vort	`true` oder `false`

Funktion:
Hiermit kann das *Vorticity Confinement* ein- oder ausgeschaltet werden.

---

newobject

`con:newobject(type,x,y,z,name)`
`con:newobject(type,p1,p2,name)`

type	`SPHERE` oder `BOX`
x,y,z	Ganzzahlige Koordinaten des Objektes im Container wenn der type `SPHERE` ist
p1,p2	Eckkoordinaten eines Quaders
name	Eindeutiger Name des Objektes

Funktion:
Erzeugt ein neues Objekt mit den Koordinaten x,y,z im Universum. Wird als type `BOX` angegeben, werden die Koordinaten der Eckpunkte eines Quaders verlangt. Ein Punkt wird mit `p = {x,y,z}` angegeben. Über den Namen kann das Objekt in Dusty direkt ausgewählt werden.

---

newsource

`con:newsource(x,y,z,name)`

x,y,z	Ganzzahlige Koordinaten der Quelle im Container
name	Eindeutiger Name der Quelle

Funktion:
Erzeugt eine neue Quelle mit den Koordinaten x,y,z im Universum. Über den Namen kann die Quelle in Dusty direkt ausgewählt werden.

---

rotate

`con:rotate(a,x,y,z)`

a	Winkel in Rad
x,y,z	Rorationsachse

Funktion:
Rotiert den Container um den Winkel `a`. Die Rotationsachse wird mit den Fließkommazahlen `x,y,z` angegeben. Z.B mit `1,0,0` wird um die x Achse rotiert.

---

set_alphascale

`con:set_alphascale(alpha`

alpha	Alphawert als Fließkommazahl, 0..1

Funktion:
Bestimmt die Durchsichtigkeit des Rauches. Null ist absult durchsichtig, ein Wert von eins macht den Rauch undurchsichtig.

---

set_densscale

`con:set_densscale(dens)`

dens	Skalierungswert für die Materie, 0..1

Funktion:
Mit dieser Funktion kann die Dichte der Materie skaliert werden.

---

set_diff

`con:set_diff(diff)`

diff	Fließkommazahl, die die Diffusion festlegt

Funktion:
Explizites Setzen der Diffusion.

---

set_dissipate

`con:set_dissipate(diss)`

diss	Fließkommazahl, die die Dissipation festlegt

Funktion:
Explizites Setzen der Dissipation.

---

set_matter

`con:set_matter(x,y,z,r,g,b)`

x,y,z	Ganzzahlkoordinanten im Container
r,g,b	Farbwert der Materie

Funktion:
Explizites Setzen von Materie.

---

set_pathtrace

`con:set_pathtrace(mode)`

mode	`EULER` oder `RUNGE`

Funktion:
Mit der Funktion wird bestimmt, welches Verfahren für die Partikelverfolgung im Advektions Teil benutzt werden soll. Es kann zwischen dem *Euler*- und dem *Runge-Kutta*-Verfahren gewählt werden. *Runge-Kutta* ist genauer als *Euler* aber zeitaufwändiger.

---

set_timestep
`con:set_timestep(time)`

time	Fließkommazahl für den Zeitschritt, 0 ...1

Funktion:
Explizites Setzen des Zeitschrittes.

---

set_visc
`con:set_visc(visc)`

visc	Fließkommazahl für die Viskosität, 0 ...1

Funktion:
Explizites Setzen der Viskosität.

---

set_vort
`con:set_vort(vort)`

vort	Fließkommazahl für das Vorticity Confinement, 0 ...1

Funktion:
Explizites Setzen des Vorticity Confinement.

A.3 Methoden für Objekte und Quellen

Die Methoden für Objekte und Quellen setzen vorraus, dass vorher mit `c:newsource()` oder `c:newobjekt()` neue Quellen und Objekte erzeugt wurden. Die nachfolgenden Methoden werden mit `o:methode()` oder `s:methode()` aufgerufen. `o` ist ein vorher erzeugtes Objekt und `s` eine vorher erzeugte Quelle.

A.3.1 Objektmethoden

---

set_place
`o:set_place(x, y, z)`

x,y,z	Ganzzahlkoordinanten im Container

Funktion:
Bewegt ein Objekt an die angegeben Position.

---

set_scale

`o:set_scale(s)`

s Scalefaktor

Funktion:
Vergrössern oder Verkleinern eins Objektes

---

set_show

`o:set_show(sh)`

sh Boolean

Funktion:
Bestimmung ob ein Objekt sichtbar sein soll.

A.3.2 Quellenmethoden

---

set_dens

`s:set_dens(r, g, g)`

r,g,b Werte für Dichte

Funktion:
Legt die Dichtewerte fest, mit der Materie in einen Container eingefügt wird.

---

set_force

`s:set_force(u, v, w)`

u,v,w Geschwindigkeitswerte für die x,y,z-Achse

Funktion:
Legt die Geschwindigkeiten für jede Achse fest. Es kann also bestimmt werden in welche Richtung eine Quelle Kräfte abgibt.

---

set_place

`s:set_place(x, y, z)`

x,y,z Ganzzahlkoordinanten im Container

Funktion:
Bewegt ein Quelle an die angegeben Position.

---

set_show

`s:set_show(sh)`

sh	Boolean

Funktion:
Bestimmung ob ein Quelle sichtbar sein soll.

---

set_settemp

`s:set_settemp(t)`

t	Temperatur

Funktion:
Bestimmt die Temperatur einer Quelle.

Anhang B
Lösung schwach besetzter Gleichungssysteme

Wie schon in Kapitel 3 gezeigt, entstehen beim Diskretisieren von *Poisson*-Gleichungen große Gleichungssysteme. Diese Gleichungssysteme sind nur schwach besetzt. Das heißt, es sind Systeme der Form $Ax = b$, bei denen der Großteil der Koeffizienten der Matrix A gleich Null ist. Im Allgemeinen sind nur einige Diagonalen mit Werten ungleich Null belegt. Standardalgorithmen zur Lösung linearer Gleichungssysteme sind für Berechnungen mit schwach besetzten Matrizen nur bedingt geeignet. Dafür gibt es folgende Gründe:

1. Der Speicheraufwand von $O(n^2)$ ist für schwach besetzte Matrizen nicht notwendig; es genügt, die Elemente ungleich Null abzuspeichern. Diesem Punkt kommt deshalb eine so große Bedeutung zu, da die Dimensionen bei dieser Art von Matrizen meist deutlich größer sind (z.B. $n = 100.000$) als bei voll besetzten Systemen.

2. Die Algorithmen sind ineffizient, da z.B. bei den Matrix-Vektorprodukten viele Teile durch Multiplikation mit Null verschwinden (Aufwand normalerweise bei $O(n^3)$ kann deutlich gedrückt werden).

Solche Systeme werden daher effizient numerisch mit iterativen Verfahren gelöst. Die Matrizen, die man üblicherweise in der Strömungsmechanik erhält, sind symmetrisch und positiv definit. Dieses begünstigt die Anwendung iterativer Verfahren. Im Weiteren sollen folgende Verfahren vorgestellt werden:

- *Gauß-Seidel* Verfahren
- SOR-Verfahren
- Verfahren der konjugierten Gradienten

Dieses sind die gängigsten Verfahren, die zum Lösen der *Poisson*-Gleichungen in der Strömungsmechanik benutzt werden.

B.1 *Gauß-Seidel* Verfahren

Wir nehmen an, alle x-Werte (bis auf den zu berechnenden i-ten) sind schon bekannt. Löst man die i-te Gleichung des Systems $Ax = b$ nach der i-ten Komponente von x auf, so erhält man:

$$x_i^{k+1} := \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k}{a_{ii}} \tag{1}$$

Mit k wird der Iterationsschritt angegeben. Zur Berechnung der k-ten Annäherung der i-ten Variablen verwendet man die (in diesem k-ten Iterationsschritt schon berechneten) x-Werte $x_1 \dots x_{i-1}$ und die im letzten Iterationsschritt $(k-1)$ berechneten x-Werte $x_{1+1} \dots x_n$. Angewendet auf eine 2 dimensionale *Poisson*-Gleichung sieht Gleichung 1 folgendermaßen aus:

$$x_{ij}^{k+1} = \frac{1}{4}[x_{i-1,j}^{k+1} + x_{i+1,j}^{k} + x_{i,j-1}^{k+1} + x_{i,j+1}^{k} - (\Delta x)^2 b_{ij}].$$

Wie man sieht, muß nur der Unbekanntenvektor gespeichert werden, da die Koeffizientenmatrix nur mit den Werten 4 und 1 gefüllt ist.

B.2 SOR-Verfahren

Das SOR (successiv overrelaxation) Verfahren ist eine Erweiterung des *Gauß-Seidel* Verfahren. Der Grundgedanke besteht darin, dass eine einmal eingeschlagene Richtung zur wahren Lösung hin nicht komplett falsch sein kann und man daher gleich einen etwas größeren Schritt wagen kann. Außerdem läßt man den vorhergehenden Iterationsschritt in die Berechnung der neuen Richtung mit einfließen. Man möchte mit diesem Vorgehen die Konvergenzgeschwindigkeit des *Gauß-Seidel* Verfahrens deutlich steigern.
Die Differenz zwischen zwei Iterationsschritten wird als Residuum bezeichnet:

$$r = x_{ij}^{k+1} - x_{ij}^{k}$$

Man versucht nun dieser Korrektur der Lösung, zwischen zwei Iterationsschritten, einen größeren Einfluß zur Beschleunigung der Iteration zu geben. Dieses erreicht man durch eine Multiplikation mit einem Faktor ω. Man erhält:

$$x_i^{k+1} := \omega \cdot \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k}{a_{ii}} + (1-\omega) \cdot x_i^k \tag{2}$$

Für die 2 dimensionale *Poisson*-Gleichung ergibt sich:

$$x_{ij}^{k+1} = (1-\omega) x_{ij}^k + \frac{\omega}{4}[x_{i-1,j}^{k+1} + x_{i+1,j}^{k} + x_{i,j-1}^{k+1} + x_{i,j+1}^{k} - (\Delta x)^2 b_{ij}].$$

Kritisch ist hier die Wahl von ω. Dieser Wert kann zwischen $0 \dots 2$ liegen. Für $\omega = 1$ erhält man das *Gauß-Seidel* Verfahren. Die Wahl von ω kann von dem konkreten Problem abhängig sein. Für $1 < \omega < 2$ spricht man von einer Überrelaxation und für $0 < \omega < 1$ von einer Unterrelaxation. Als Richtwert kann man $\omega = 1,7$ benutzen. Letztendlich gilt folgender Satz:

Satz Ist A eine symmetrische und positiv definite Matrix, so konvergiert das SOR-Verfahren für jedes ω aus dem Intervall $(0, 2)$ und für jeden beliebigen Startvektor.

Weitere Informationen zum SOR-Verfahren findet man in [11, 16].

B.3 Verfahren der konjugierten Gradienten

Verfahren der konjugierten Gradienten (CG-Verfahren) sind effiziente Lösungsalgorithmen für symmetrische, positiv definite Systeme. Sie basieren auf der Verwendung von *Korrekturrichtungen* bei der Aktualisierung der Iterationen und Residuen. Dabei muß jeweils nur eine geringe Anzahl von Vektoren gespeichert werden.
Damit bestimmte Orthogonalitätsbedingungen der Vektorfolgen erfüllt bleiben, werden bei jeder Iteration zwei innere Produkte zur Berechnung von skalaren Korrekturgrößen gebildet. Bei symmetrischen, positiv definiten linearen Systemen bewirken diese Bedingungen, daß der Abstand zur exakten Lösung in der verwendeten Norm minimiert wird.
Anstelle des linearen Gleichungssystems $Ax = b$ wird iterativ die äquivalente Minimierungsaufgabe gelöst. Es gilt

$$Ax = b$$

und minimiere

$$F(x) = \frac{1}{2}x^T Ax - x^T b$$

sind äquivalent. Den Beweis erhält man mit der Hilfsfunktion

$$E(x) = \frac{1}{2}(Ax - b)^T A^{-1}(Ax - b).$$

Da auch A^{-1} positiv definit ist, gilt $E(x) > 0$. Somit ist $E(x)$ genau dann minimal, wenn gilt $Ax - b = 0$.
Die Berechnung von $E(x)$ ergibt unter Verwendung von $A^T = A$

$$E(x) = F(x) + \frac{1}{2}b^T A^{-1}b$$

Daraus folgt, dass $E(x)$ und $F(x)$ an derselben Stelle minimal sind, d.h.

$$F(x) \stackrel{!}{=} \text{Min.} \;\Leftrightarrow Ax - b = 0.$$

Die Richtung des stärksten Abstiegs einer Funktion ist durch ihren negativen Gradienten gegeben. Für F gilt:

$$\nabla F(x) = Ax - b.$$

Es besteht also die Aufgabe, jenen Punkt zu suchen, in dem der Gradient verschwindet.
Gegeben ist eine symetrische, positv definite $n \times n$ Matrix A. Gesucht ist die Lösung x des linearen Gleichungssystems $Ax = b$. Es lässt sich folgender Algorithmus für das CG-Verfahren angeben:

1. Startpunkt $x_0 \in \mathbb{R}^n$ (beliebig) $d_0 = -g_0 = -(Ax_0 - b)$. Falls $g_0 = 0$, kann abgebrochen werden, x_0 ist dann die Lösung.

2. Für $k = 0, 1, \ldots, n-1$ werden nacheinander berechnet:

   (a) $\alpha_k = -\frac{d_k^T g_k}{d_k^t A d_k}$

   (b) $x_{k+1} = x_k + \alpha_k d_k$ mit $\{x_{k+1} \in x_0 + U_{k+1}(d_0, \ldots, d_k)\}$

   (c) $g_{k+1} = g_k + \alpha_k A d_k$ mit $\{g_{k+1} \perp d_0, \ldots, d_k, g_{k+1} \notin U_{k+1}(d_0, \ldots, d_k)\}$ Gilt zu vorgegebenen $\varepsilon > 0 \quad ||g_{k+1}||_\infty < \ \varepsilon$, kann dann abgebrochen werden mit x_{k+1} als Lösung.

   (d) $\beta_k = \frac{g_{k+1}^T A d_k}{d_k^T A D_k}$

   (e) $d_{k+1} = -g_{k+1} + \beta_k d_k$ mit $\{d_{k+1} \perp A d_k, d_{k+1} \notin U_{k+1}(d_0, \ldots, d_k)\}$

Vorteile des Verfahrens sind:

- leichte Vektorisierbarkeit und Parallelisierbarkeit, sowie
- rasche Konvergenz. Diese wird aber mit einem größeren Rechenaufwand pro Schritt erkauft.

Nachteil des Verfahrens ist die große Empfindlichkeit gegen Rundungsfehler. Für höchste Qualität, empfiehlt *Stam* jedoch dieses Verfahren. Zur Implementation kann das C++ Template von [1] benutzt werden.

Anhang C
Glossar

Alphakanal Im Alphakanal werden bei verschiedenen Bildformaten (z.B. PNG, PSD oder TIFF) Transparenzinformationen zusätzlich zu den eigentlichen Bilddaten gespeichert. In der Regel wird hierfür ein zusätzliches Byte pro Pixel verwendet, dessen Wert die Durchlässigkeit des Bildes gegenüber dem Hintergrund beschreibt.

CFD Die CFD (computational fluid dynamics) beinhaltet Verfahren zur numerischen Berechnung strömender Medien.

CFL Bedingungen Die *Courant-Friedrichs-Levy* Bedingungen sind wichtige Stabilitätsbedigungen in der numerischen Simulation. Sie begrenzen die Größe eines Zeitschrittes Δt während der Simulation. Die CFL Bedingungen hängen von dem verwendeten physikalischen System zur Modellierung der Simulation und den benutzten Diskretisierungsmethoden ab. Im konkreten Fall der Fluid Simulation bedeutet das, dass kein Partikel des Fluids in der Zeit Δt mehr als eine Gitterweite h zurücklegen darf. Es muss also gelten:$|\mathbf{u}|\Delta t < h$. Hierdurch wird eine Simulation langsam. Der im *Stable Fluids* Algorithmus verwendete *semi-Lagrange* Ansatz ist nicht von den CFL Bedingungen abhängig und erlaubt damit große Schrittweiten die für eine schnellere Simulation sorgen.

Diffusion Diffusion ist der Mechanismus, mittels dessen sich mikroskopische Partikel in Flüssigkeiten oder Gasen frei oder durch (semi)permeable Barrieren hindurch ausbreiten. Grundlage der Diffusion ist die Brownsche Molekularbewegung. Die Diffusion ist eine Räumliche Verteilung ohne Einfluss von Strömungen.

Divergenz Die Anwendung des Nabla-Operators ∇ auf ein Vektorfeld f ergibt über das Skalarprodukt $\nabla \cdot f$ ein skalares Feld, das in jedem Punkt des Raumes angibt, ob dort Feldlinien entstehen oder verschwinden. Am Ort einer positiven Punktladung wäre die Divergenz des elektrischen Feldes beispielsweise größer als Null, da an diesem Punkt Feldlinien entstehen. Punkte mit positiver Divergenz werden Quellen genannt, Punkte mit negativer Divergenz dagegen Senken.

Fluid Ein Fluid ist ein Material, das Scherkräften nicht widerstehen kann.

Gradient Ein Feld, dem eine skalare Funktion f zugrunde liegt, ordnet jedem Punkt des Definitionsraumes von f eine Zahl zu. Ein Beispiel für ein solches skalares Feld im dreidimensionalen Raum wäre die Temperatur, die Dichte, oder das Potential, die an jedem Ort durch eine Zahl (plus Einheit) beschrieben werden können. Die Anwendung des Nabla-Operators ∇ auf f ergibt ein Vektorfeld, das Gradient genannt wird. Der Gradient zeigt an jedem Punkt des Raumes in die Richtung des stärksten Anstiegs, sein Betrag gibt die Steigung in diese Richtung an. Ist das skalare Feld ein Potential, so gibt der negative Gradient des Feldes das zugehörige Kraftfeld an. Anschaulich klar ist das im Fall des Gravitationsfeldes: Ein Körper fällt in die Richtung, in der die Änderung seines Potentiales maximal ist.

laminare Strömung Die laminare Strömung ist die Bewegung von Flüssigkeiten und Gasen, bei der keine Turbulenzen (Verwirbelungen) auftreten. Bei der laminaren Strömung verhält sich der Fliesswiderstand proportional zur Strömungsgeschwindigkeit.

Rotation Neben dem Skalarprodukt können zwei Vektoren auch über das Kreuzprodukt miteinander verknüpft werden. Bildet man $\nabla \times f$ erhält man eine Vektorfunktion, die Rotation genannt wird und die die Wirbel von f charakterisiert.

Semaphore Semaphoren dienen der Synchronisation von Prozessen. Jede einzelne Semaphore besitzt Speicherplatz für einen nicht-negativen Ganzzahlwert. Das besondere an diesen Strukturen ist, dass sie zum Blockieren und Freigeben von Prozessen benutzt werden können. Das heißt, ein Prozeß kann einen bestimmten Wert in eine Semaphore schreiben und dann seine Weiterausführung solange unterbrechen, bis ein erwarteter anderer Wert in der Semaphore steht.

Thread Eine Instanz der Ausführung eines Programmes, aber mit etwas weniger Gewicht als ein Prozeß, da zu einem Prozeß mehrere Threads gehören können, die alle Ressourcen des Prozesses gemeinsam nutzen.

turbulente Strömung Die turbulente Strömung ist die Bewegung von Flüssigkeiten und Gasen, bei der Turbulenzen (Verwirbelungen) auftreten. Bei der turbulenten Strömung verhält sich der Fliesswiderstand proportional zum Quadrat der Strömungsgeschwindigkeit.

Viskosität Die Viskosität gibt die Zähigkeit von Flüssigkeiten und Gasen an. Bewegt sich ein fester Körper mit der Geschwindigkeit $\mathbf{v}$ durch eine ruhende Flüssigkeit, dann ist im Allgemeinen zur Aufrechterhaltung der Bewegung eine Kraft $\mathbf{F}$ erforderlich, die von der Größe und Form des Körpers und einer Eigenschaft der Flüssigkeit, der dynamischen Viskosität, ν, abhängt.

Anhang D
Abbildungen

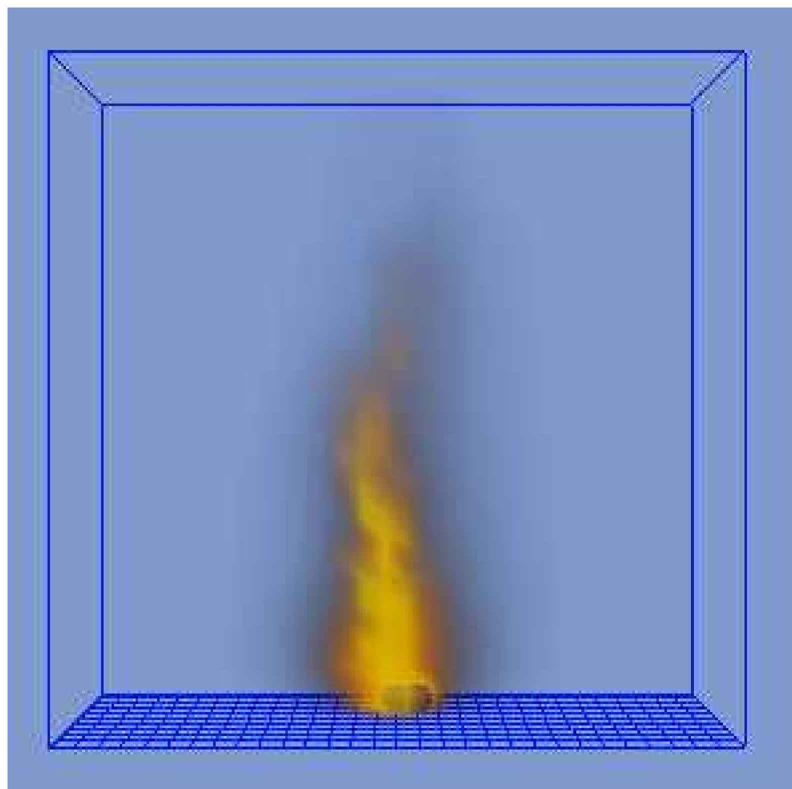

Abbildung D.1: Feuersimulation (30x30x10)

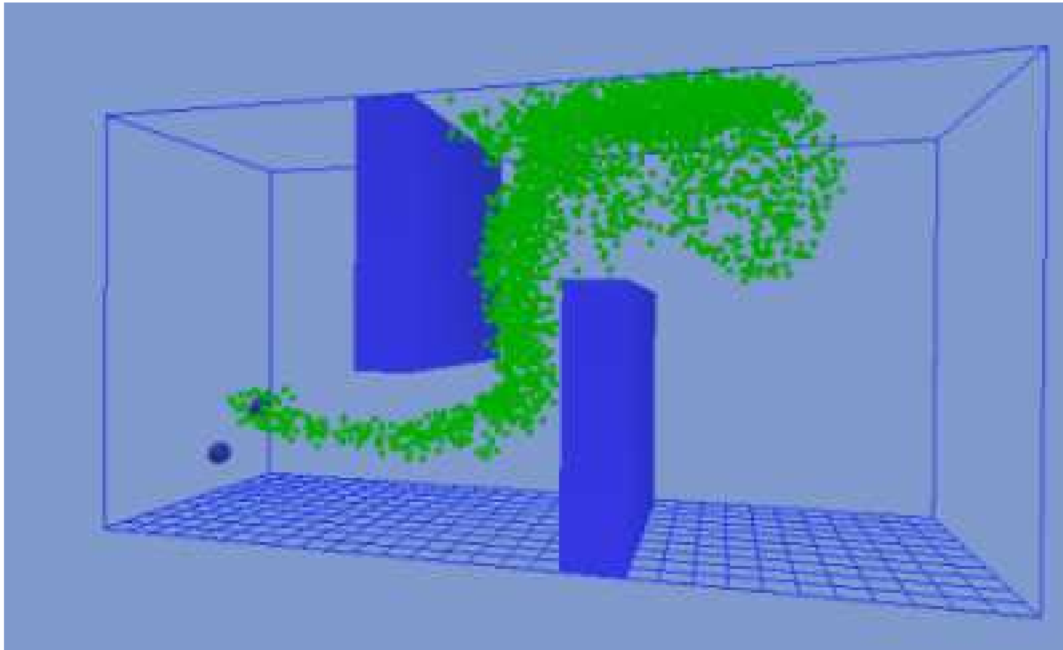

Abbildung D.2: Visualisierung durch Partikeln (30x15x10)

Abbildung D.3: Beispiel für 2D-Rauch (70x50x3)

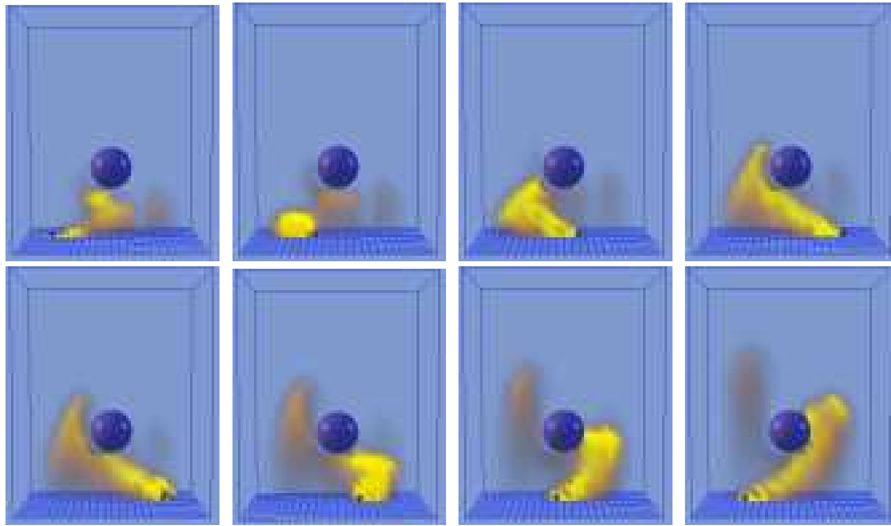

Abbildung D.4: Animation einer Quelle (24x30x10)

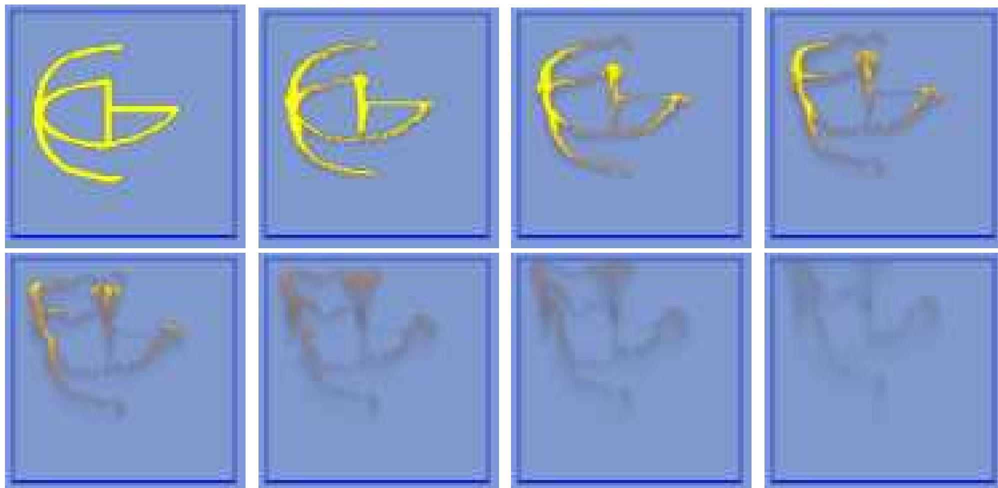

Abbildung D.5: Verdampfen des ICG Logos (120x120x3)

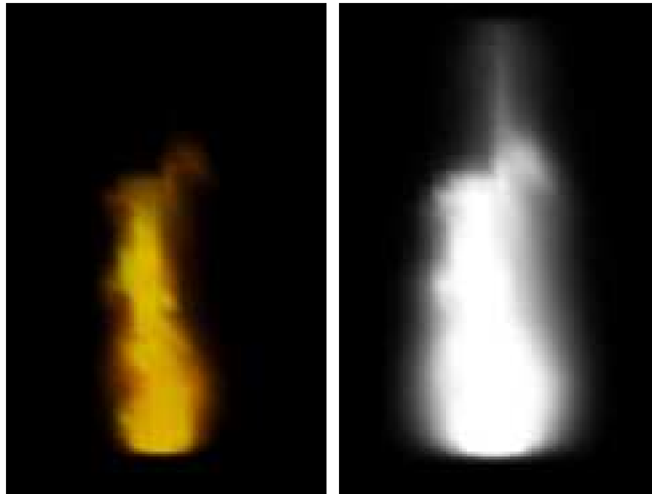

Abbildung D.6: Feuer mit Alphakanal (30x30x10)

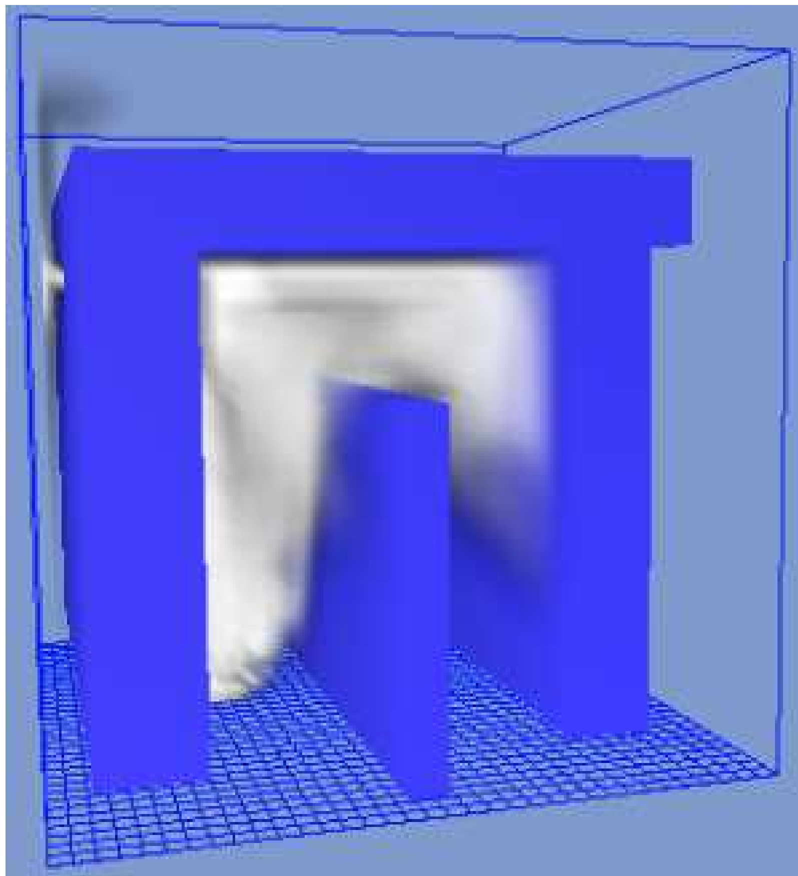

Abbildung D.7: Rauchverteilung in einer komplexen Szene (35x35x35)

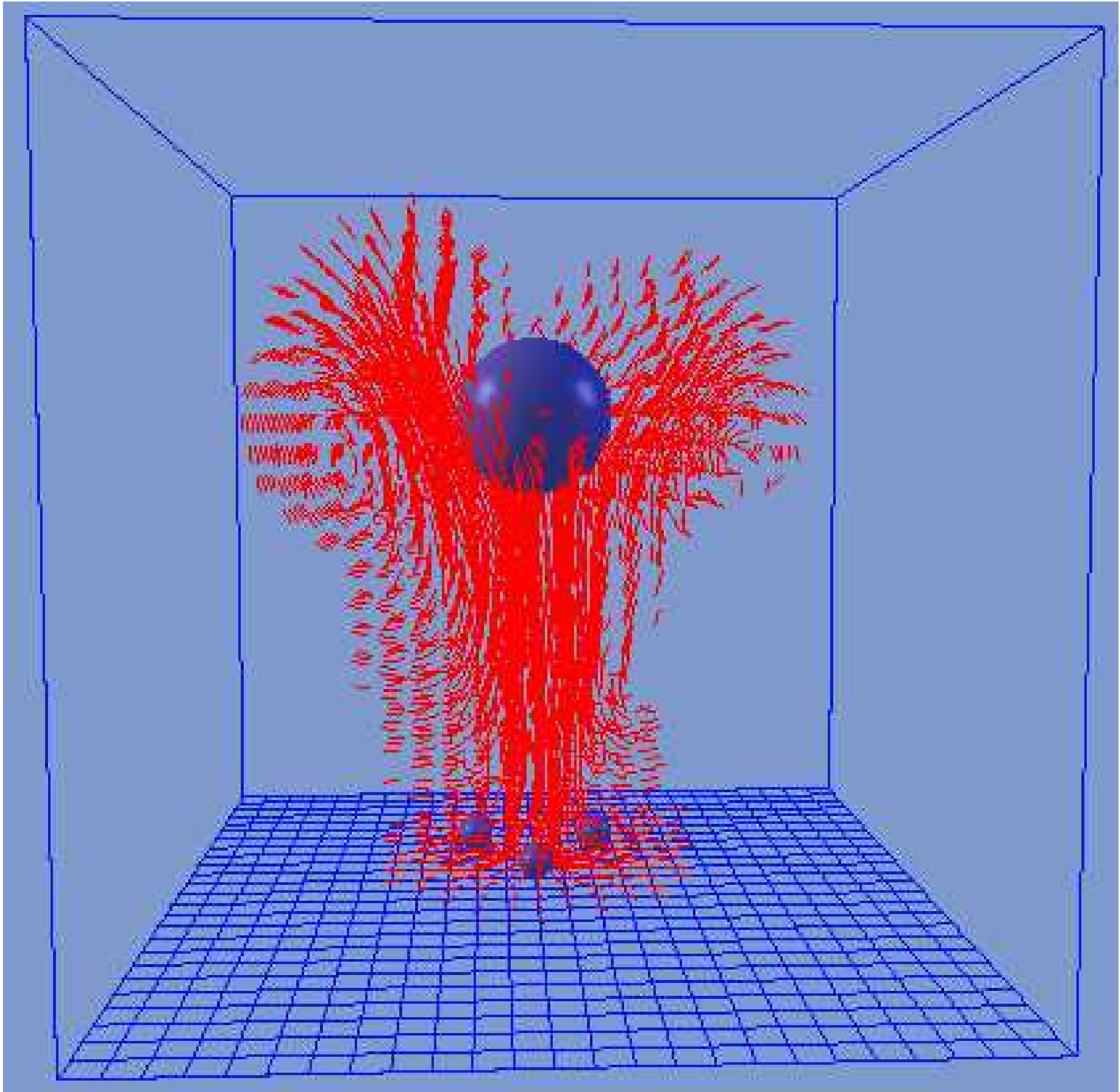

Abbildung D.8: Visualisierung durch Strömungslinien (25x25x25)

Literatur

[1] Iterative Methods Library (IML++). http://math.nist.gov/iml++.

[2] Lua Tutorial. http://lua-users.org/wiki/LuaTutorial.

[3] MJPEG Tools. http://mjpeg.sourceforge.net.

[4] The Fast Light Toolkit. http://www.fltk.org.

[5] Roger Crawfis. Implementing a Virtual Trackball. http://www.cis.ohio-state.edu/∼crawfis/cis781/Slides/VirtualTrackball.html.

[6] J. Welch F. Harlow. Numerical calculation of time-dependet viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, (8):2182–2189, 1965.

[7] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual Simulation of Smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[8] Nick Foster and Ronald Fedkiw. Practical Animations of Liquids. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 23–30. ACM Press / ACM SIGGRAPH, 2001.

[9] Nick Foster and Dimitris Metaxas. Modeling the Motion of a Hot, Turbulent Gas. *Computer Graphics*, 31(Annual Conference Series):181–188, 1997.

[10] Fritz Reutter Gisela Engeln-Müllges. *Numerik-Algorithmen*. VDI Verlag, Düsseldorf, 1996.

[11] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssyteme*. B.G. Teubner, Stuttgart, 1993.

[12] P.H. Christiansen H.W. Jensen. Effecient Simulation of Light Transport in Scenes with Participating Media using Photon Maps. In *SIGGRAPH 98 Conference Proceedings, Annual Conference Series*, pages 311–320, 1998.

[13] Clay Mathematics Institute. Millenium Prize Problem . http://www.claymath.org/Millennium_Prize_Problems.

[14] Bryan Feldman James. Animating Suspended Particle Explosions. *ACM Transactions on Graphic*, (22):707–715, 2003.

[15] Steven G. Johnson. Matteo Frigo. Fastest Fourier Transform in the West. `http://www.fftw.org`.

[16] Tilman Neunhoeffer Michael Griebel, Thomas Dornseifer. *Numerische Simulation in der Strömungsmechanik*. Vieweg Verlag, Braunschweig, 1995.

[17] P.F. Lopes M.N. Gamito and M.R. Gomes. Two-dimensional Simulation of Gaseous Phenomena Using Vortex Particles. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15. Springer Verlag, 1995.

[18] Timm S. Müller. TEKlib – Virtual operating system and middleware project. `http://teklib.neoscientists.org`.

[19] Willi Geiger Ronald Fedkiw N. Rasmussen, Duc Quang Nguyen. Smoke Simulation For Large Scale Phenomena. In *SIGGRAPH 2003, Computer Graphics Proceedings*, pages 703–707. ACM Press / ACM SIGGRAPH, 2003.

[20] L. Navier. Memoire dur les lois du mouvements des fluids. *Memoir. de l'Academ. Royale des Sci.*, 6, 1827.

[21] Andrew Nealen. Physical Based Simulation and Animation of Gaseous Phenomena in a Periodic Domain. Technical report, Department of Computer Science University of British Columbia, 2002.

[22] D. Nguyen, R. Fedkiw, and H. Jensen. Physically Based Modeling and Animation of Fire. In *Siggraph 2002 Annual Conference*, pages 721–728, 2002.

[23] Herbert Oertel. *Strömungsmechanik*. Vieweg Verlag, Braunschweig, 1999.

[24] Dennis C. Prieve. *A Course in Fluid Mechanics with Vector Field Theory*. Department of Chemical Engineering Carnegie Mellon University Pittsburgh, PA 15213, 2000.

[25] D. Peacemanm H. Rachford. The numrical solution of parabolic and elliptic differential equations. *Indus. Appl. Math.*, 3:28–41, 1955.

[26] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *Proceedings of SIGGRAPH '93*, pages 369–376. Addison-Wesley Publishing Company, 1993.

[27] Jos Stam. Stable fluids. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 121–128, Los Angeles. Addison Wesley Longman.

[28] Jos Stam. A simple fluid solver based on the FFT. *Journal of Graphics Tools: JGT*, 6(2):43–52, 2001.

[29] Jos Stam. Real-Time Fluid Dynamics for Games. In *Proceedings of the Game Developer Conference*, March 2003.

[30] J. Steinhoff and D. Underhill. Modification of the euler equations for "vorticity confinement":Application to the computation of interacting vortex rings. *Physics of Fluids*, 6(8):2738–2744, 1994.

[31] George Stokes. On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastics solids. *Transact. Cambridge Philos. Soc.*, 9, 1851.

[32] Tecgraf. Lua scriptinterpreter. `http://www.lua.org`.

[33] A. Thom. *The flow past circular cylinders at low speeds*, chapter A 141, pages 651–666. Proc. R. Soc. London, 1933.

[34] Frank M. White. *Fluid Mechanics*. MCGRAW-HILL PUBL.COMP., 1998.

[35] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins. Animating explosions. In *SIGGRAPH*, pages 29–36, 2000.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 13.01.2004 Frank Pagels

Thesen

1. Die realitätsnahe Simulation von Feuer und Rauch ist immer noch eine große Herausforderung in der Computergraphik.

2. Bei der Simulation von Rauch handelt es sich um eine Strömungssimulation. Exakte Strömungssimulation ist sehr aufwändig und kann selten auf einem Standard-PC in Echtzeit erfolgen.

3. Der *Stable Fluids* Algorithmus basiert auf den, in der Strömungsmechanik benutzten, *Navier-Stokes* Gleichungen. Der Algorithmus erlaubt die Verwendung eines groben Diskretisierungsgitters und relativ große Zeitschritte ohne das der Algorithmus instabil wird. Hierdurch wird eine Strömungssimulation in Echtzeit möglich.

4. Für die vollständige analytische Lösung der *Navier-Stokes* Gleichungen, existieren noch keine Verfahren. Daher erfolgt die Lösung durch Approximation mittels der Finite Differenzen Methode.

5. In dieser Diplomarbeit wurde ein Animationssystem, basierend auf dem *Stable Fluids* Algorithmus, entworfen und implementiert. Mit dem System DUSTY können in Echtzeit die drei-dimensionalen *Navier-Stokes* Gleichungen auf einem Standard-PC gelöst werden. In das Simulationsgebiet können beliebig viele Hindernisse eingefügt werden, die realitätsnah den Strömungsverlauf beeinflussen.

6. Zum Erstellen und Animieren von Szenen wurde in DUSTY ein Skriptsystem integriert. Hierzu wurde die Skriptsprache *LUA* verwendet, die um eigene DUSTY spezifische Befehle erweitert wurde. Mit diesem Befehlssatz können schnell Szenen erstellt und animiert werden.

7. Ein Volumenrendering erfolgt durch OpenGL. Die Ergebnisse für Feuereffekte sind mittels Alphakanal für kleine Filmeffekte nutzbar. In komplexen Szenen kann die Ausbreitung von Rauch verfolgt werden. Das Strömungsfeld kann durch ein Partikelsystem und Stromlinien visualisiert werden.