UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Simulation und Visualisierung von Fluiden in einer Echtzeitanwendung mit Hilfe der GPU

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Franz Peschel

Betreuer: Stefan Rilling
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 2009

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Mithilfe von Fluiden lassen sich einige der visuell beeindruckendsten Naturphänomene wie Feuer, Wasser oder Rauch realistisch darstellen. Der Simulation und Visualisierung von Fluiden wurde daher in der Vergangenheit stets großes Interesse entgegengebracht. Fluide gelten mittlerweile in der Computergrafik als gut erforschtes Gebiet. Fluidsimulationen wurden aber bisher eher im Offline-Rendering beispielsweise für Animationsfilme oder auch in der Forschung und Entwicklung für Strömungssimulationen eingesetzt. Fluidbewegungen in Echtzeit zu simulieren galt lange Zeit als zu aufwendig. Dank der jüngsten Entwicklung der Grafikhardware ist die Echtzeitsimulation von Fluiden nicht nur möglich, Fluidsimulationen lassen sich heute sogar in Echtzeitanwendungen wie Spiele-Engines neben anderen aufwendigen Berechnungen integrieren. Die Algorithmen wurden dabei so angepasst, dass sie die hochparallele Architektur der GPU's optimal ausnutzen.

Diese Arbeit liefert einen Einblick in die Technik hinter der Echtzeitsimulation von Fluiden auf der GPU unter Verwendung von Shader Model 3.0 in OpenGL 2.1. Zusätzlich wurde ein OpenGL-Renderer als Echtzeitanwendung entworfen, um die Möglichkeit der Integration der Fluidsimulation in eine solche Anwendung zu demonstrieren. Zur Ausführung der Simulation wird mindestens eine DirectX9 GPU mit Shadermodel 3.0 benötigt.

2 State of the Art

Physikalische Prozesse wie die Entwicklung von Fluiden werden schon eine ganze Weile mithilfe von Computersystemen simuliert. Solche Simulationen haben grundsätzlich zwei verschiedene Zielstellungen. Einerseits sollen sie für wissenschaftliche Forschung physikalische korrekt sein, andererseits sollen sie für Präsentationszwecke beipielsweise für den Einsatz in Computerspielen optisch plausibel und effizient zu berechnen sein. Eine physikalisch plausible und mit aktueller Grafikhardware auch in Echtzeit verwendbare Simulation von Fluiden ist mit Hilfe der, für numerische Strömungssimulation geeigneten, Navier-Stokes-Gleichungen möglich.

Die Navier-Stokes-Gleichungen wurden im 19. Jahrhundert (1827 und 1845) von George Gabriel Stokes und Claude Louis Marie Henri Navier unabhängig voneinander formuliert. Sie gelten auch heute noch als Grundlage der Strömungsmechanik und beschreiben den Impulssatz (Impulserhaltung) für newtonsche Fluide wie Wasser, Gase oder Öle (viskose Flüssigkeiten) in differentieller Form [Wik09b]. Zur vollständigen Beschreibung einer Fluidentwicklung verwendet man noch die Gleichung zur Massenerhaltung. Zur numerischen Lösung der Navier-Stokes-Gleichungen (ein „nichlinearen partiellen Differentialgleichungssystems 2. Ordnung“ [Wik09b]) existieren unterschiedliche Ansätze.

Alle Elemente eines Volumens in einem Fluid mit Hilfe der Navier-Stokes-Gleichungen zu berechnen ist zu aufwendig, um praktikabel zu sein. Daher wählt man für den jeweiligen Zweck ein geeignetes Simulationsverfahren.

2.1 Das rasterbasierte Verfahren

Die Grundidee des rasterbasierten Ansatzes besteht in der Aufteilung der Szene in ein gleichmäßiges 3D-Raster. Die Navier-Stokes-Gleichungen werden auf die Werte in den Voxelzentren des 3D-Raster angewendet. Jedes Voxelzentrum hat seinen eigenen Geschwindigkeits-, Druck- und Dichtewert (Partikelanzahl). Die Dichtewerte können direkt gerendert werden. Die Werte der Volumenelemente zwischen den Voxelzentren werden interpoliert. Der Ablauf eines Simulationsschritts sieht dabei folgendermaßen aus:

- Wende die Advektion (Fluidbewegung) auf die Geschwindigkeitswerte und die Dichtewerte an und bewege die Partikel im Fluid.
- Wende die Beschleunigung (externe Kräfte, Temperatursimulation, Wirbelstärkenerhaltung), Hinderniserkennung und Diffusion auf die Geschwindigkeitswerte an.
- Passe den Druck im Voxel iterativ an (Inkompressibilität) [Ste08]
- Render die Dichte mittels geeignetem Volumen Rendering Verfahren.

Die rasterbasierte Methode wird hauptsächlich in der numerischen Strömungssimulation [Hei07] eingesetzt, da sie physikalisch plausibel ist und präzise Ergebnisse liefert. Die Methode ist jedoch sehr rechenaufwendig und außerdem durch die lokale Begrenzung des 3D-Rasters nur eingeschränkt verwendbar. Der Einsatz des Verfahrens in Computerspielen eignet sich daher am besten für Innenräume.

Abbildung 1: Hellgate London [TL07] und S.T.A.L.K.E.R.: Clear Sky [GSC08]

Dank der Entwicklung leistungsfähiger Grafikhardware im Consumerbereich wird der rasterbasierte Ansatz bereits vereinzelt in aktuellen Computerspielen mit DirectX10-

Unterstützung wie z.B. dem MMORPG[1] „Hellgate London“ oder dem Ego-Shooter „S.T.A.L.K.E.R.: Clear Sky“ (siehe Abb. 1) eingesetzt.

2.2 Smoothed Particle Hydrodynamics (SPH)

Die Smoothed Particle Hydrodynamics[2] (kurz SPH) wurden ursprünglich von L. B. Lucy, R. A. Gingold und J. J. Monaghan für die Simulation von astrophysikalischen Problemen entwickelt, lassen sich nach [MCG03] aber auch auf die Fluidsimulation anwenden (siehe Abb. 2). Bei den Smoothed Particle Hydrodynamics handelt es sich um eine Lagrange-Methode, d.h. die Grundidee des Verfahrens besteht darin, die Werte der Elemente im Volumen nicht in Voxeln zu speichern, sondern einzelne Elemente einfach im Fluid mitschwimmen zu lassen [Ste08]. Ein Volumenelement in einem Fluid wird als Massepunkt aufgefasst und man wählt eine Funktion, die aus dem Massepunkt wieder kleine Tropfen macht [Ste08]. Um die Beschleunigung eines Volumenelements bestimmen zu können, werden die Positionen und Beschleunigungen der Nachbarelemente verwendet. Druck entsteht durch viele relativ nahe Nachbarelemente, Reibung entsteht durch die unterschiedlichen Geschwindigkeiten der Nachbarelemente. Zwischen den in die Berechnung einbezogenen Elementen des Volumens kann man die Werte der übrigen Volumenelemente einfach interpolieren [Ste08].

Abbildung 2: Smoothed Particle Hydrodynamics [MCG03]

Der partikelbasierte Ansatz ist gut für computergrafische Simulationen in Echtzeit geeignet (z.B. Computerspiele), da man eine variable Anzahl von Partikeln bestimmen kann, die in die Berechnungen der Simulation mit einbezogen werden sollen. Bei einer geringeren Anzahl von Partikeln erreicht man eine gute Performanz, allerdings auf Kosten der physikalischen Korrektheit und damit der visuellen Qualität ($O(NlogN)$-Abhängigkeit des Rechenaufwands von der Teilchenzahl [Wik09c]). Das Verfahren ist außerdem laut [Wik09c] sehr dispersiv, d.h. es findet

[1] Massive Multiplayer Online Role Playing Game
[2] dt.: geglättete Teilchen-Hydrodynamik [Wik09c]

ähnlich wie bei der Diffusion, nur stärker, ein Ausgleich zwischen den Konzentrationsunterschieden im Fluid statt. Dennoch kann man auch mit diesem Verfahren in Kombination mit den Vortexpartikeln aus [FSWJ01] plausible visuelle Ergebnisse erzielen. Die Smoothed Particle Hydrodynamics bilden die Basis für Fluidsimulationen in Nvidias PhysX-Engine.

Für die Umsetzung der Aufgabenstellung der Studienarbeit fiel die Wahl auf das rasterbasierte Verfahren trotz seiner Einschränkungen, da die Qualität der visuellen Ergebnisse gegenüber den Ergebnissen der Smoothed Particle Hydrodynamics überzeugender war.

3 Überblick

Im folgenden Abschnitt 4 zu „Dynamischen Fluiden“ werden die mathematischen Grundlagen der Fluidtheorie anhand der Navier-Stokes-Gleichungen erläutert. Außerdem wird die rasterbasierte numerische Lösungsmethode nach [Sta99] beschrieben. Im nächsten Abschnitt 5 zu „Volume Rendering“ werden dann zunächst einige Techniken zur Visualisierung eines Volumendatensatzes vorgestellt. Im Abschnitt 6 „Softwaretechnisches Konzept und Implementierung“ wird auf Kapitel 4 und 5 aufbauend die Implementierung des rasterbasierten numerischen Lösungsverfahrens und des verwendeten Visualisierungsverfahren vorgestellt. Dabei dient ein OpenGL-Renderer als Basis für die Integration der Fluidsimulation in eine Echtzeitanwendung. Die Fluidsimulation wird auf der GPU mit Hilfe von Fragmentshadern berechnet. In den beiden letzen Abschnitten werden abschließend die Ergebnisse besprochen und ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

4 Dynamische Fluide

4.1 Navier-Stokes-Gleichungen für inkompressible homogene Fluide

Bei einem inkompressiblen homogenen Fluid[3] handelt es sich um ein vereinfachten Fall des komplexen physikalischen Modells zur Simulation von Fluiden.

- Ein Fluid ist dann inkompressibel, wenn das Volumen jedes Teilbereichs des Fluids konstant über die Zeit bleibt, d.h. das Volumen vergrößert sich nicht.
- Ein Fluid ist homogen, wenn seine Dichte (Partikelanzahl) d im Raum konstant bleibt.
- Inkompressibel und homogen bedeutet in Kombination, dass die Dichte konstant in Raum und Zeit bleibt.

[3] lat. *fluidus*, fließend

Diese Vorraussetzungen sind für dynamische Fluide üblich und schränken laut [Har03] die Verwendbarkeit des mathematischen Modells für die Simulation realer Fluide wie Rauch (Gase), Wasser oder Feuer nicht ein. In den folgenden Abschnitten werden die Gleichungen im Detail beschrieben. Das Lösungsverfahren der Navier-Stokes-Gleichungen lässt sich in die Teilprobleme

- *Advektion*,
- *Diffusion*,
- *Beschleunigung* und
- *Projektion*

zerlegen [Har03]. Jeder dieser Teilschritte wird in den folgenden Abschnitten ausführlich behandelt. Zunächst werden wir aber ein paar weitere Rahmenbedingung festlegen.

Das dynamische Fluid wird in einem 3D-Raster[4] mit Hilfe kartesischer Koordinaten im Raum $\vec{x} = (x, y, z)$ und der Zeitvariable *t* simuliert. Ein Fluid mit konstanter Dichte und Temperatur kann nun mit einem Geschwindigkeits-Vektor-Feld (*velocity field*) $\vec{u}$ und einem skalaren Druckfeld (*pressure field*) *p* beschrieben werden.

$$\vec{u}(\vec{x}, t), p(\vec{x}, t)$$

Falls die Geschwindigkeit und der Druck an jedem Ort im Fluid zum Startzeitpunkt $t = 0$ bekannt sind, kann der Zustand des Fluids zu jedem zukünftigen Zeitpunkt mit Hilfe der folgenden Navier-Stokes-Gleichung für inkompressible Fluide bestimmt werden:

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

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

Die Variable $\vec{u}$ und *p* kennen wir schon als Geschwindigkeitsfeld und Druckfeld, *d* ist die Dichte (*density*), ν die Viskosität (*viscosity*), $\vec{F} = (f_x, f_y, f_z)$ beschreibt den Einfluss externer Kräfte und $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ ist der sogenannte Nabla-Operator (siehe Kap. 4.2). Bei der Gleichung 1 handelt es sich um die Impulserhaltungsgleichung und bei der Gleichung 2 um die Masseerhaltungsgleichung, d.h. um die Masseerhaltung zu gewährleisten, muss das Geschwindigkeitsfeld stets divergenzfrei (siehe Kap. 4.2) sein [Har03]. Auch die Bedingungen an den Rändern des Fluids müssen berücksichtigt werden, dazu später mehr. Betrachten wir vorerst die einzelnen Terme der ersten Gleichung genauer.

[4] engl.: *Grid*

4.2 Der Nabla-Operator

In Gleichung 1 und Gleichung 2 wird der Nabla-Operator ∇ auf drei verschiedene Weisen genutzt, als Gradient-Operator, Divergenz-Operator und Laplace-Operator.

Operator	**Definition für den kontinuierlichen Fall**
Gradient	$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}\right)$
Divergenz	$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$
Laplace	$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}$

Operator	**In Finite-Differenzen-Form für den diskreten Fall**
Gradient	$\frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\delta x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\delta y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\delta z}$
Divergenz	$\frac{u_{i+1,j,k} - u_{i-1,j,k}}{2\delta x}, \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2\delta y}, \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2\delta z}$
Laplace	$\frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{(\delta x)^2}, \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{(\delta y)^2}, \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{(\delta z)^2}$

Tabelle 1: Verschieden Operatoren für die Berechnungen in der Fluidsimulation [Har05].

Die Indizes i, j, k stehen für die Koordinaten eines Voxels in einem diskretisierten dreidimensionalen Raster. δx, δy und δz bestimmen die Abstände zwischen den Voxeln im Raster in Richtung x, y und z.

4.2.1 Der Gradient

Der Gradient eines Skalarfelds ist ein Vektor, bestehend aus allen partiellen Ableitung des Skalarfelds [Har03].

4.2.2 Die Divergenz

Die Divergenz besitzt eine physikalische Bedeutung. Sie beschreibt, zu welchem Anteil sich Partikel in einer bestimmten Region des Raumes befinden. In den Navier-Stokes-Gleichungen wird die Divergenz auf die Geschwindigkeit des Flusses angewendet, um die Veränderung der Geschwindigkeit in dem Bereich um eine kleine Region des Fluids zu messen [Har05]. Das Fluid kann nicht zusammengedrückt werden[5]. Damit diese anfangs gestellte Annahme bestand hat, wird in Gleichung 2 sichergestellt, dass das Fluid stets Null Divergenz besitzt, d.h. Divergenzfrei ist. Das Skalarprodukt des Divergenzoperators summiert die partiellen Ableitungen der einzelnen Komponenten des Geschwindigkeitsvektors auf. Der Divergenz-Operator kann also nur auf ein Vektorfeld wie das Geschwindigkeitsfeld $\vec{u} = (u, v, w)$ angewendet werden [Har05].

[5] inkompressibel

4.2.3 Der Laplace-Operator

Der Gradient eines Skalarfelds ist ein Vektorfeld. Die Divergenz eines Vektorfelds ist wieder ein Skalarfeld. Wenn der Divergenz-Operator auf das Ergebnis eines Gradientoperators angewendet wird, erhält man den Laplace-Operator $\nabla \cdot \nabla = \nabla^2$ als Ergebnis. Falls die Rastervoxel des Volumens an jeder Seite die gleiche Kantenlänge besitzen (kubisch), kann man den Laplace-Operator auf folgende Notation vereinfachen:

$$\nabla^2 p = \frac{p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6p_{i,j,k}}{(\delta x)^2} \tag{3}$$

Der Laplace-Operator findet häufiger Anwendung in der Physik, beispielsweise in Form einer Diffusions-Gleichung wie der Gleichung für Hitzeentwicklung zur Beschreibung der Hitzeverteilung in einer Region. In der Form $\nabla^2 x = b$ wird sie auch als Poisson Gleichung bezeichnet. Im Fall $b = 0$ handelt es sich um die Laplace Gleichung, welche den Ursprung des Laplace-Operators darstellt. In der zweiten Gleichung $\nabla \cdot \vec{u} = 0$ wird der Laplace-Operator auf ein Vektorfeld angewandt. Dabei handelt es sich um eine Vereinfachung der Notation, die Operation wird eigentlich für jede Komponente des Vektorfelds einzeln ausgeführt [Har05].

4.3 Advektion

Der erste Term auf der rechten Seite der Gleichung 1 beschreibt die sogenannte (selbst-) Advektion[6] des Geschwindigkeitsfeldes. Das Geschwindigkeitsfeld des Fluids sorgt dafür, dass das Fluid Substanzen wie Partikel, Objekte oder andere Dinge entlang des Strömungsflusses transportiert. Wenn man z.B. blaue Füllertinte in Wasser tropft, wird die Tinte entlang des Geschwindigkeitsfeldes des Wassers transportiert bzw. „geströmt“. Die Geschwindigkeit eines Fluids transportiert sich sogar selbst mit beim Transport der Substanz. Der beschriebene Advektions-Term sorgt also auch für die selbst-Advektion des Geschwindigkeitsfeldes [Har03].

4.4 Druck

Im zweiten Term $-\frac{1}{d}\nabla p$ wird die Beschleunigung, die durch den Druck-Gradienten verursacht wird, beschrieben. Da Moleküle in einem Fluid sich relativ frei umeinander herum bewegen können, prallen sie ständig zusammen oder werden einfach aneinander gedrückt. Wenn auf die Moleküle in einem lokalen Bereich eines Fluids eine Kraft ausgeübt wird, fliegen diese nicht gleich durch das gesamte Volumen des Fluids, sondern prallen auf die benachbarten Moleküle, wodurch wiederum Druck ensteht, da sich nun eine höhere Anzahl an Molekülen in einem Teilbereich des

[6] lat. *advehi*, heranbewegen

Fluids befindet. Die Moleküle werden an dieser Stelle aneinander gedrückt. Genauso entsteht Unterdruck in dem Bereich, von dem die Moleküle abgezogen werden. Das verursacht dann ebenfalls eine Beschleunigung mit anderem Vorzeichen. Da Druck gleich Kraft pro Raumeinheit ist, führt jeglicher Druck nach Newtons zweiten Gesetz $\vec{F} = m\vec{a}$ zu Beschleunigung [Har03].

4.5 Diffusion

Der dritte Term $\nu\nabla^2\vec{u}$ beschreibt die Diffusion des Fluids, deren Wirkungsgrad mit Hilfe der Viskosität skaliert wird. Die Viskosität ist ein Maß dafür, wie „zähflüssig“ oder dickflüssig eine Flüssigkeit fließen soll. Sirup oder Honig haben z.B. eine höhere Viskosität als Wein oder Wasser. Der so beschriebene Widerstand in einem Fluid beinflusst also, wie stark sich ein Impuls in einer Flüssigkeit ausbreiten bzw. diffundieren kann.

4.6 Beschleunigung

Im vierten Term $\vec{F}$ wird die Beschleunigung durch externe Kräfte, die auf ein Fluid wirken, ausgedrückt. Es gibt dabei verschiedene Arten von Kräften. Globale Kräfte wie die Schwerkraft unterscheiden sich von lokale Kräfte wie beispielsweise einem Ventilator oder einer anderen Strömungsquelle. Damit kleine Details wie Verwirbelungen in der Strömungssimulation erhalten bleiben, wird das Fluid durch eine Wirbelstärkenkraft beschleunigt (Siehe Kap. 4.6.3). Für die Simulation von Rauch eignet sich zudem eine Temperatursimulation (siehe Kap. 4.6.2). Dabei wird das Fluid mit Hilfe von einer durch Wärmequellen erzeugten Auftriebskraft beschleunigt, die zusätzlich von der Schwerkraft, die auf die Rauchpartikel wirkt, beeinflusst wird und für sogenannte Konvektionsströmungen sorgt [Har03].

4.6.1 Simulation von Flüssigkeiten und Gasen

Die direkteste Anwendung der Fluidsimulation besteht aus der Simulation von Flüssigkeiten und Gasen. Um eine Fluidsimulation sichtbar machen zu können, benötigt man lediglich ein Feld mit skalaren Werten, welches die zu transportierenden Partikel enthält. Dabei kann es sich z.B. um Staub, Wassertropfen oder Rußpartikel für Rauch handeln. Wir wenden den Advektions-Operator auf dieses Partikelfeld *d* an, um den Fluss der Partikel durch das Fluid zu simulieren.

$$\frac{\partial\vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{d}$$

Dazu verwenden wir den selben Teil aus Gleichung 1, den wir für die Advektion der Geschwindigkeit nutzen. Anstelle der Geschwindigkeit $\vec{u}$ wird der Advektions-Operator aber auf die Dichte *d* angewendet [Har05].

4.6.2 Simulation von Rauch - Auftrieb und Konvektionsströme

Temperatur spielt eine wichtige Rolle für das Flussverhalten vieler Arten von Fluiden. Die Veränderung der Dichte und der Temperatur im Zusammenspiel erzeugen Konvektionsströme [Har03], die z.B. das Verhalten des Wetters oder auch das unseres Kaffees in der Kaffeetasse beeinflussen. Um die dabei entstehenden Effekte simulieren zu können, müssen wir, wie in [Har03] beschrieben, den Auftrieb[7] berechnen und ihn zu unserer Fluidsimulation hinzufügen. Dazu verwenden wir einfach ein neues Skalarfeld für die Temperatur T und einen Auftriebs-Operator, der dem Geschwindigkeitsfeld an der Stelle einen Impuls hinzufügt, an der die durch eine Wärmequelle erzeugte lokale Temperatur höher ist, als die Umgebungstemperatur T_u.

$$\vec{F}_{buo} = \sigma\,(T - T_u)\,\vec{k} \tag{4}$$

$\vec{k}$ zeigt die vertikale Richtung an und σ ist ein konstanter Skalierungsfaktor. Für die Simulation von Rauch benötigen wir zusätzlich noch die Dichte aus dem Skalarfeld d. Die Auftriebskraft $\vec{F}_{buo}$ wird dabei von der Gravitationskraft, die auf die Rauchpartikel einwirkt, beeinflusst [Har03].

$$\vec{F}_{buo} = (-\mu d + \sigma\,(T - T_u))\,\vec{k}$$

Bei μ handelt es sich um einen konstanten Skalierungsfaktor für die Masse der Partikel. Durch das Hinzufügen von Rauchpartikeln und Temperaturquellen (z.B. in Form einer Feuerquelle oder einer glühenden Zigarettenspitze) an eine beliebige Position im Fluidvolumen können wir nun Rauch simulieren [Har05].

4.6.3 Wirbelstärkenerhaltung

Der Rauch einer Zigarette oder die Tinte in einem Wasserglas bilden hochdetailierte faszinierend anzuschauende Verwirbelungen. Durch numerischen Verluste der Simulation verschwinden diese detailreichen Turbulenzen. Neben künstlichen Ansätzen, welche die Wirbel wieder pseudo-zufallsgesteuert hinzufügen, existiert eine Ansatz von [FSWJ01] zur Wirbelstärkenerhaltung[8], der die Wirbel an ihrer korrekten Position wieder hinzufügt. Das Verfahren besitzt drei Teilschritte.

1. Im ersten Schritt wird ein Wirbelstärkefeld aus dem Geschwindigkeitsfeld berechnet. Wirbelstärke ist der Grad der Zirkulations-Rotationen in einer Partikelwolke, dargestellt mit Hilfe eines Vektorfeldes. Der Betrag eines Vektors im Vektorfeld beschreibt dabei den Grad der Rotation und die Richtung des Vektors beschreibt die Rotationsachse der Rotation an dieser Stelle.

$$\vec{\Phi} = \nabla \times \vec{u}$$

[7] engl.: *buoyancy*

[8] engl.: *vorticity confinement*

2. Im zweiten Schritt werden die normalisierten Wirbelstärke-Positions-Vektoren berechnet. Sie zeigen lokal von der Region mit geringer Wirbelstärke zu der Region mit höherer Wirbelstärke.

$$\vec{X} = \frac{\vec{\omega}}{|\vec{\omega}|}, \vec{\omega} = \nabla|\vec{\Phi}|$$

3. In einem dritten Schritt wird die Richtung und die Skalierung der Wirbelstärkenkraft $\vec{F}_{vc}$ bestimmt.

$$\vec{F}_{vc} = \epsilon \left(\vec{X} \times \vec{\Phi} \right) \tag{5}$$

   Wie in Abb. 3 gezeigt, kontrolliert ϵ den Umfang der Details, die wieder zum Strömungsfeld hinzugefügt werden [FSWJ01].

Nach der Berechnung des Wirbelstärkenkraftfelds kann es wie jede andere externe Kraftquelle einfach zu dem Geschwindigkeitsfeld wieder hinzugefügt werden [Har03]. Das Hinzufügen der Wirbelstärkenerhaltung ist optional und kann ausgelassen werden.

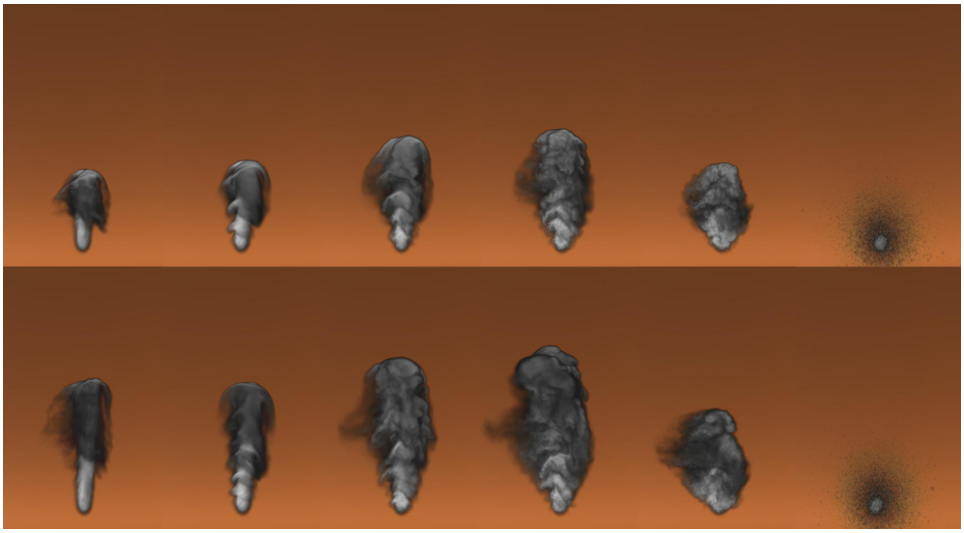

Abbildung 3: Hinzufügen der Wirbelstärkenerhaltung zur Simulation mit unterschiedlichem ϵ-Werten. Von links nach rechts: $\epsilon = -0.25$, $\epsilon = 0$, $\epsilon = 0.25$, $\epsilon = 0.5$, $\epsilon = 0.75$, $\epsilon = 1.3$. Ein zu hoher Wert verursacht eine instabile Simulation und Artefakte in der Darstellung [FSWJ01].

Da sich die Qualität der Visualisieren durch das Hinzufügen der Wirbelstärkenerhaltung erheblich verbessert und die Performanz nicht signifikant verschlechtert, ist das Verwenden der Wirbelstärkenerhaltung jedoch sehr zu empfehlen.

4.7 Lösen der Navier-Stokes-Gleichungen

Navier-Stokes-Gleichungen sind für einige vereinfachte physikalische Gegebenheiten analytisch lösbar. Eine inkrementelle numerische Lösung erlaubt uns z.B. die Entwicklung eines Flusses über die Zeit darzustellen. Der in [Sta99] dazu vorgestellte Algorithmus zum Lösen der Navier-Stokes-Gleichungen besteht aus mehreren Schritten, in denen die einzelnen Gleichungen gelöst werden. Die Navier-Stokes-Gleichungen zur Impulserhaltung enthalten im dreidimensionalen Fall vier skalare Gleichungen mit vier Unbekannten: (u, v, w, p), oder einfach $(\vec{u}, p)$ [Har05].

4.7.1 Helmholtz-Hodge Zerlegung

Ein Vektor $\vec{v}$ lässt sich in einzelne Basisvektoren zerlegen. Summiert man die Basisvektoren auf, erhält man wieder $\vec{v}$. Ein Vektor wird üblicherweise als ein n-Tupel von Distanzen in einem n-dimensionalen Raum dargestellt. Unter Verwendung normalisierter Basisvektoren $\vec{a}_1, \vec{a}_2, \vec{a}_3, ..., \vec{a}_n$, die parallel zu den Achsen eines kartesischen Koordinatensystems ausgerichtet sind, kann man einen Vektor als Summe darstellen: $\vec{v} = x\vec{a}_1 + y\vec{a}_2 + z\vec{a}_3$ [Har05].

Genau wie ein Vektor lässt sich auch ein Vektorfeld in eine Summe von einzelnen Vektorfeldern zerlegen. Da die Geschwindigkeits- und Druckfelder zusammenhängen, können wir zur Vereinfachung den Druck zunächst mit Hilfe der Helmholtz-Hodge Zerlegung eliminieren, um die Anzahl der Gleichungen von vier auf drei zu reduzieren [Har05]. Das Ergebniss der drei Gleichungen ist jedoch vorerst divergent. Laut Gleichung 2 zur Erhaltung der Masse muss das Geschwindigkeitsfeld aber divergenzfrei sein. Dieses Problem löst man wie folgt.

Der Helmholtz-Hodge Satz besagt, dass jedes Vektorfeld $\vec{w}$ in eine Summe von zwei Vektorfeldern zerlegt werden kann, in ein divergenzfreies Vektorfeld $\vec{u}$ und dem Gradienten eines Skalarfeldes p. Die Vektoren des divergenzfreien Vektorfelds werden dabei an den Rändern Richtung Null angenähert [Har05].

$$\vec{w} = \vec{u} + \nabla p \tag{6}$$

Das Lösen der Navier-Stokes-Gleichungen beinhaltet also nach [Sta99] für jeden Zeitschritt drei Berechnungsschritte, die Advektion, Diffusion und das Hinzufügen eines Impulses. Das Ergebnis ist ein Geschwindigkeitsfeld $\vec{w}$, dessen Divergenz wie bereits festgestellt noch nicht Null beträgt und damit nicht die in der Gleichung 2 geforderte Masseerhaltung erfüllt. Ziehen wir den Gradienten des resultierenden Druckfeldes in einem späteren Projektion genannten Schritt von dem Geschwindigkeitsfeld $\vec{w}$ ab, können wir laut Helmholtz-Hodge Satz die Divergenz wieder korrigieren. Die Divergenz von $\vec{w}$ beträgt dann wieder Null und die Masseerhaltung ist gewährleistet [Har03].

$$\vec{u} = \vec{w} - \nabla p$$

Der Helmholtz-Hodge Satz führt auch zu einer Methode, die es uns ermöglicht, das Druckfeld zu berechnen. Dazu wenden wir den Divergenz-Operator auf beiden Seiten der Gleichung $\vec{w} = \vec{u} \cdot p$ an.

$$\nabla \cdot \vec{w} = \nabla \cdot (\vec{u} + \nabla p)$$

$$\nabla \cdot \vec{w} = \nabla \cdot \vec{u} + \nabla^2 p$$

Da aber die Masseerhaltungsgleichung $\nabla \cdot \vec{u} = 0$ fordert, vereinfacht sich die Gleichung zu

$$\nabla^2 p = \nabla \cdot \vec{w} \tag{7}$$

mit Hilfe dieser Poisson-Gleichung lässt sich der Druck eines Fluids bestimmen. Die Gleichung wird daher auch Poisson-Druck Gleichung genannt. Nachdem wir unser divergentes Geschwindigkeitsfeld $\vec{w}$ bestimmt haben, können wir die Gleichung 7 für den Druck p lösen, um mit $\vec{w}$ und p anschließend am Ende jedes Zeitschritts wieder ein neues divergenzfreies Geschwindigkeitsfeld $\vec{u}$ zu berechnen [Har05].

Als nächstes müssen wir einen Weg finden $\vec{w}$ zu bestimmen. Zum besseren Verständnis schauen wir uns den Schritt zunächst wieder an einem Vektor an und übertragen ihn anschließend auf ein Vektorfeld. Das Skalarprodukt eines Vektor $\vec{v}$ und eines Einheitsvektors $\vec{e}$ liefert uns die Projektion von $\vec{v}$ auf $\vec{e}$. Das Skalarprodukt arbeitet also als Projektionsoperator für Vektoren. Der Operator bildet in unserem Beispiel den Vektor $\vec{v}$ auf seine Komponenten in Richtung des Vektors $\vec{e}$ ab. mit Hilfe des Helmholtz-Hodge Satzes können wir einen solchen Projektions-Operator P definieren, der ein Vektorfeld $\vec{w}$ auf seine divergenzfreie Komponente $\vec{u}$ projeziert. Wende wir P auf die Gleichung 6 an, erhalten wir

$$P\vec{w} = P\vec{u} + P\vec{u} + P(\nabla p)$$

P ist laut Definition $P\vec{w} = P\vec{u} = \vec{u}$, daher ist $P(\nabla p) = 0$. Wenden wir nun unseren Projektions-Operator P auf beide Seiten der Navier-Stokes-Gleichung 1 zur Impulserhaltung an, erhalten wir

$$P\frac{\partial \vec{u}}{\partial t} = P\left(-(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{d}\nabla p + \nu \nabla^2 \vec{u} + \vec{F}\right)$$

Da $\vec{u}$ divergenzfrei ist, ist auch die Ableitung $\frac{\partial \vec{u}}{\partial t}$ divergenzfrei, wir erhalten daher $P\left(\frac{\partial \vec{u}}{\partial t}\right) = \frac{\partial \vec{u}}{\partial t}$. Der Druck-Term kann aus der Gleichung eliminiert werden, da

$P(\nabla p) = 0$, um die Masseerhaltung zu gewährleistet [Har05]. Wir erhalten die Gleichung

$$\frac{\partial \vec{u}}{\partial t} = P\left(-(\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u} + \vec{F}\right) \tag{8}$$

Den Wert der Funktion nach der Zeit des Geschwindigkeitsfelds $\vec{u}(t)$ erhalten wir für jeden Zeitpunkt *t* mit dem erhöhen von *t* um δt. Die Lösung der projezierten Gleichung 1 zur Impulserhaltung besteht aus mehreren Schritten [Dep08]. Das Geschwindigkeitsfeld $\vec{u}(t)$ zum Zeitpunkt t erhalten wir, indem wir auf der rechten Seite der Gleichung jeden Term einzeln auswerten und anschließend addieren. Von links nach rechts starten wir mit dem ersten Term der (selbst-) Advektion und erhalten das Geschwindigkeitsfeld $\vec{w}_1$. $\vec{w}_1$ wird im zweiten Term der Diffusion verändert und wir erhalten $\vec{w}_2$. $\vec{w}_2$ wird beschleunigt durch das Hinzufügen externer Kräfte, Temperatur oder der Wirbelstärken-Erhaltung, wir erhalten das divergente Geschwindigkeitsfeld $\vec{w}_3$ Zum Schluss wird der Projektionsoperator P auf $\vec{w}_3$ angewendet. Das resultierende Geschwindigkeitsfeld ist jetzt divergenzfrei und die Masseerhaltung ist erfüllt. Wir lösen dazu die Gleichung 7, um den Druck *p* zu ermitteln und den Gradienten von *p* von $\vec{w}_3$ abzuziehen [Dep08].

$$\vec{u}(t) \overset{\text{Advektion}}{\longrightarrow} \vec{w}_1 \overset{\text{Diffusion}}{\longrightarrow} \vec{w}_2 \overset{\text{Beschleunigung}}{\longrightarrow}{}^{9} \vec{w}_3 \overset{\text{Projektion}}{\longrightarrow} \vec{u}(t + \delta t) \text{ [Dep08]}$$

Jede Komponente der Gleichung 8 ist als separater Schritt zu verstehen, der ein Vektorfeld als Eingabe und eine verändertes Vektorfeld als Ausgabe besitzt. Die gesamte Gleichung 8 lässt sich wie P als Operator S [Har03] auffassen, der die Gleichung 8 für jeden Zeitschritt löst. Dieser Operator wird laut [Har05] als Komposition der Operationen für

- *Advektion A*,
- *Diffusion D*,
- *Beschleunigung F*
- *Projektion P*

zu

$$S(\vec{u}) = P \circ F \circ D \circ A(\vec{u}) \text{ [Har03]}$$

Im folgenden Abschnitt werden die einzelnen Schritte einzeln erläutert und das Lösungsverfahren der Poisson-Gleichung beschrieben.

4.7.2 Advektion

Die Advektion beschreibt, wie ein Geschwindigkeitsfeld eines Fluids sich selbst oder andere Dinge, z.B. Partikel im Fluidvolumen transportiert. Um die Advektion von Partikeln zu berechnen, müssen die Partikel an jeder Stelle in einem Raster aktualisiert werden. Wir könnten die Position $\vec{s}$ eines jeden Partikels entlang des Geschwindigkeitsfelds über eine Zeitspanne δt bewegen.

$$\vec{s}(t + \delta t) = \vec{s}(t) + \vec{u}\delta t \tag{9}$$

Dieses explizite Euler-Verfahren[10] ermöglicht das explizite Integrieren von gewöhnlichen Differentialgleichungen und ist gleichzeitig „das einfachste und schnellste Verfahren für das numerische Lösen eines Anfangswertproblems“[11] [Wik09a]. Allerdings ist es auch das ungenaueste mit der höchsten Fehlerquote. Die Differentialgleichung gehört zur Klasse der Anfangswertaufgaben, da wir sie aus vorgegebenen Anfangsdaten $\vec{s}_0$ und einem Zeitpunkt t_0 mit Hilfe einer Differentialgleichung $\vec{s}$ berechnen wollen. Natürlich gilt dabei $\vec{s}(t_0) = \vec{s}_0$. Neben dem Euler-Verfahren existieren auch noch genauere Verfahren wie die Runge-Kutta-Verfahren oder die linearen Mehrschrittverfahren [Wik09a], diese sind aber etwas langsamer [Har05].

Für gewöhnliche Differentialgleichungen würden wir beispielsweise eine diskrete Zeitspanne für die Schritte wählen. Allerdings ist der Eulersche Ansatz laut [Har05] besonders für größere Zeitschritte numerisch instabil. Der Grund für die Instabilität des Verfahrens ist die Vorraussage der Zukunftswerte basierend auf aktuellen Werten, wobei in jedem Schritt Ungenauigkeitsfehler aufaddiert werden bis der Fehler zu groß wird und die Werte jeden Rahmen sprengen. Umso größer der Zeitschritt gewählt wird, desto größer wird der resultierende Fehler. Fehler treten auf, wenn der Betrag von $\vec{u}(t)\,\delta t$ größer ist als eine Zelle des Rasters. Ausserdem lässt sich das Verfahren nicht auf der GPU implementieren, da die Positionen der Werte in den Fragmenten nicht verändert werden können, während auf ihnen geschrieben wird. Die Partikel müssen aber für die Realisierung des Verfahren beweglich sein [Har05].

Anstelle eines expliziten Verfahrens nutzen wir daher, wie [Sta99] vorschlägt, ein invertiertes implizites rückwärtiges Verfahren, welches uns die notwendige Stabilität garantiert. Im Gegensatz zum Euler-Verfahren, indem wir Partikel über einen Zeitschritt advektieren, verfolgen wir jetzt die Flugbahn der Partikel jeder Rasterzelle zurück zu ihren früheren Positionen. Anschließend werden die Partikel an dieser Position zu der Position ihrer Start-Rasterzelle kopiert. Dieser Rückschau-Ansatz ist laut [Sta99] für beliebige Zeitschritte und Geschwindigkeiten stabil, da

[10] auch Eulersches Polygonzugverfahren
[11] auch Cauchy-Problem

wärend der Vorraussage der zukünftigen Werte gleichzeitig die früheren Werte korrigiert werden und der Gesamtfehler an einen endlichen Wert angenähert wird. Um die Positionen der Menge q, die z.B. aus Partikeln, Geschwindigkeits- oder Temperaturwerten bestehen kann, im Fluid aktualisieren zu können, verwenden wir folgende Gleichung

$$q\left(\vec{x}, t+\delta t\right) = q\left(\vec{x} - \vec{u}\left(\vec{x}, t\right)\delta t, t\right) \tag{10}$$

Die Gleichung beschreibt, wie wir ein Partikel entlang seines Weges zurückverfolgen [Har05].

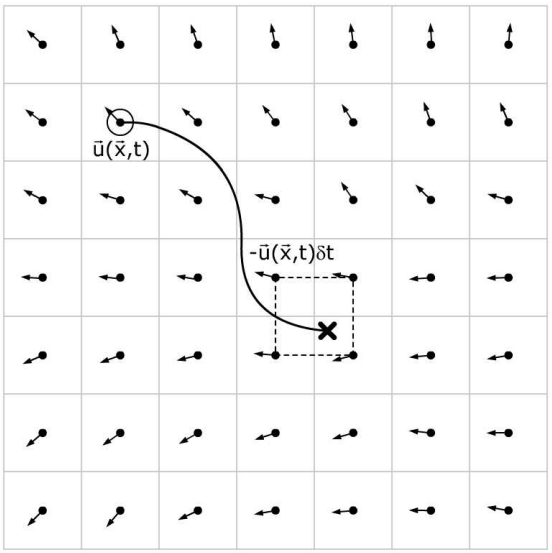

Abbildung 4: 2D-Advektion nach [Har03]

Abb. 4 zeigt die Advektion eines Partikels in der markierten Zelle. Das Ergebnis „x“ wird trilinear interpoliert und in die Startzelle geschrieben [Har05].

4.7.3 Viskose Diffusion

Viskose Fluide besitzen einen Widerstand gegen den Fluss. Dieser führt zur Diffusion[12] der Geschwindigkeiten. Der Grad der Dämpfung der Bewegung des Flusses wird von der Viskosität ν bestimmt. Ein hoher Wert lässt das Fluid wie dickflüssiges Öl fließen, ein niedriger Viskositätswert dagegen lässt das Fluid wie ein Gas strömen. Bei

$$\frac{\partial \vec{u}}{\partial t} = \nu \nabla^2 \vec{u} \tag{11}$$

handelt es sich um eine partielle Differentialgleichung, die die viskose Diffusion beschreibt. Genau wie bei der Advektion verwenden wir eine Rückschau-Methode. Dazu nutzen wir die Werte des Laplace-Operator vom künftigen noch unbekannten Geschwindigkeitsfeld. Wir gehen zunächst wieder von einer expliziten diskreten Gleichung aus

$$\vec{u}\left(\vec{x}, t + \delta t\right) = \vec{u}\left(\vec{x}, t\right) + \nu \delta t \nabla^2 \vec{u}\left(\vec{x} t\right)$$

Der Laplace-Operator ∇^2 steht hier in seiner diskreten Form. Ähnlich zur Advektion ist die Methode in dieser Form für große Werte von δt und ν instabil. Wir nutzen daher erneut eine in [Sta99] vorgeschlagene implizite Form der Gleichung 11:

$$\left(I - \nu \delta t \nabla^2\right) \vec{u}\left(\vec{x}, t + \delta t\right) = \vec{u}\left(\vec{x} t\right) \tag{12}$$

oder auch in der Form

$$\left(I - \nu \delta t \nabla^2\right) \vec{w}_2\left(\vec{x}\right) = \vec{w}_1\left(\vec{x}\right)$$

I ist eine Einheitsmatrix. Die implizite Form ist wieder für beliebige Zeitschritte δt und Viskositätswerte ν stabil [Har03]. Da Diffusions-Effekte durch die Ungenauigkeit der Berechnung im Adektionsschritt ohnehin auftreten und die Berechnung der Diffusion nur mit höheren Viskositätswerten bemerkbar ist, kann der Diffusionsschritt bei niedrigen Werten einfach ausgelassen werden. Die Berechnung der Diffusion ist außerdem sehr rechenaufwändig und verbessert die Qualität der Visualisierung nicht, da sie Details verschwinden lässt. Wenn das Verhalten von stark viskosen Flüssigkeiten in der Simulation von Interesse ist, sollte der Diffusionsschritt verwendet werden. In allen anderen fällen ist von seiner Verwendung eher abzuraten.

4.7.4 Beschleunigung

Das Fluid bzw. das Geschwindigkeitsfeld des Fluids wird durch Hinzufügen von externen Kräften $\vec{F}$, z.B. durch einen Ventilator, durch Auftriebskräfte $\vec{F}_{buo}$ bei einer Temperatursimulation oder durch die Wirbelstärkenkraft $\vec{F}_{vc}$ (siehe Kap. 4.6.3)

[12] Zerstreuung

beschleunigt. Das Geschwindigkeitsfeld verändert sich proportional zu den hinzugefügten Kräften.

$$\vec{w_3} = \vec{w_2} + \left(\vec{F} + \vec{F}_{buo} + \vec{F}_{vc}\right) \delta t \tag{13}$$

Um das Strömen eines Fluids überhaupt beobachten zu können, muss eine Kraftquelle die Strömung erstmal erzeugen. Eine Partikelwolke sollte für die Visualisierung der Strömung ebenfalls hinzugefügt werden (siehe nächstes Kapitel).

4.7.5 Projektion - Lösen der Poisson-Gleichung

Der letzte Schritt der Simulation ist die Projektion P.

$$\vec{u}\,(t + \delta t) = P\,(\vec{w_3})$$

Dazu müssen zwei Poisson-Gleichungen gelöst werden:

1. Die Poisson-Druck Gleichung
2. Die Gleichung für viskose Diffusion

Wir verwenden dafür ein iteratives Verfahren, welches mit einer angenäherten Lösung beginnt und diese Schritt für Schritt verbessert. Die Poisson-Gleichung hat die Form eines Matrix-Gleichungssystems

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

$\vec{x}$ ist der Vektor der zu ermittelnden Werte Druck p und Geschwindigkeit $\vec{u}$. $\vec{b}$ ist ein Vektor von Konstanten. A ist die Matrix, die hier den Laplace-Operator ∇^2 repräsentiert. Die iterative Lösungstechnik beginnt mit einer initialen Startschätzung x_0^0 der Lösung. In jedem Schritt k wird die Lösung weiter angenähert und verbessert zu x_k^n. n gibt die Anzahl bereits getätigter Iterationen an. Eine einfache iterative Technik ist die Jacobi-Iteration. Sie ist leicht zu implementieren und daher geeignet [Har05].

Die Gleichungen 7: $\nabla^2 p = \nabla \cdot \vec{w}$ und 12: $\left(I - \nu \delta t \nabla^2\right) \vec{u}\,(\vec{x}, t + \delta t) = \vec{u}\,(\vec{x}t)$ lassen sich beide mit Hilfe von Gleichung 3 diskretisieren und umschreiben zu der Form

$$\vec{x}_{i,j,k}^{n+1} = \frac{\vec{x}_{i-1,j,k}^{n} + \vec{x}_{i+1,j,k}^{n} + \vec{x}_{i,j-1,k}^{n} + \vec{x}_{i,j+1,k}^{n} + \vec{x}_{i,j,k-1}^{n} + \vec{x}_{i,j,k+1}^{n} + \alpha \vec{b}_{i,j,k}}{\beta} \tag{14}$$

α und β sind Konstante. Die Werte für $\vec{x}, \vec{b}, \alpha$ und β sind für die zwei Poisson-Gleichungen verschieden. In die Poisson-Druck Gleichung wird das Druckfeld p für $\vec{x}$ und das Divergenzfeld $\nabla \cdot \vec{w}$ für $\vec{b}$ eingesetzt.

1. Poisson-Druck Gleichung

   - $\vec{x} = p$
   - $\vec{b} = \nabla \cdot \vec{w}$
   - $\alpha = -(\delta x)^2$
   - $\beta = 6$

   Das Ergebnis der Poisson-Druck Gleichung ist zwar $p\delta t$ statt p, der Druck wird aber nur zur Berechnung des Gradienten im Projektionsschritt genutzt. Da δt über dem gesamten Raster konstant bleibt, beeinflusst δt den Gradienten nicht. In die Gleichung für viskose Diffusion wird das Geschwindigkeitsfeld $\vec{u}$ für $\vec{x}$ und $\vec{b}$ eingesetzt.

2. Gleichung für viskose Diffusion

   - $\vec{x} = \vec{u}$
   - $\vec{b} = \vec{u}$
   - $\alpha = \frac{(\delta x)^2}{\nu \delta t}$
   - $\beta = 6 + \alpha$

Um die Gleichung möglichst genau zu lösen, werden eine Anzahl von Iterationen durchlaufen. Dabei wird die Gleichung 14 auf jeder Rasterzelle ausgeführt und das Ergebnis als Eingabe für den nächsten Iterationsschritt verwendet [Har03].

$$\vec{x}^{(n+1)} \longrightarrow \vec{x}^{(n)}$$

Da die Jacobi-Iteration nur langsam konvergiert, müssen mindestens 20 Schritte ausgeführt werden. Meine Implementation verwendet ca. 30-40 Schritte.

4.8 Start- und Randbedingungen

Für die Berechnung der Fluidentwicklung benötigen wir den Startzustand des Fluids. Als Startzustand nehmen wir das Geschwindigkeitsfeld und Druckfeld im gesamten Volumen als Null an. Jedes Differentialgleichungsproblem eines endlichen Bereichs benötigt Randbedingungen. Sie werden für das Geschwindigkeitsfeld und das Druckfeld separat definiert. Das Fluid wird in einem Raster simuliert und besitzt eine Grenze. Die Grenze soll in unserem Fluid durchlässig simuliert werden. Es existeren jedoch verschiedene Randbedingungen:

- no-slip Bedingung
  Will man eine undurchlässige Begrenzung simulieren wie z.B. bei einem Aquarium, nutzt man die no-slip Bedingung. Dafür wird der Wert der nächsten Nachbarzelle einer Randzelle mit -1 skaliert und in die Randzelle kopiert. Die Geschwindigkeit zwischen den beiden Zellen beträgt jetzt Null.

- free-slip Bedingung
  Für die Randbehandlung in einem Geschwindigkeitsfeld mit einem Hindernisobjekt verwendet man die free-slip Bedingung. Die Voxel in der Nähe der Ränder des Hindernisobjekts nehmen die Geschwindigkeit des Hindernisobjekts entlang seiner Oberflächennormalen an, d.h. die Oberflächennormale wird in das jeweils nächstgelegene Voxel kopiert.

- Neumann-Randbedingung
  Für das Druckfeld wählen wir die Neumann-Randbedingung. Der Skalierungsfaktor beträgt hier 1. Das führt zu einem Null-Gradienten zwischen der Randzelle und der nächsten inneren Nachbarzelle.

In dem Volumen, in dem die Rauchpartikel gespeichert werden, setzen wir in jedem Zeitschritt am Rand den Wert Null. Dadurch können wir in beständig neue Rauchpartikel hinzufügen, ohne dass das ganze Volumen nach ein paar Sekunden komplett mit Rauchpartikeln ausgefüllt ist. Trifft ein Partikel auf den Rand, verschwindet es. Das Geschwindigkeitsfeld wird ebenfalls in den Randzellen stetig auf Null gesetzt. So wird der Eindruck einer unsichtbaren Barriere an den Grenzen des Fluidvolumens verhindert, an der der Rauch abzuprallen scheint.

5 Volume Rendering

Das Volumenrendern ermöglicht die Visualisierung der Strukturen im Inneren eines Objekts. Es unterscheidet sich damit vom gebräuchlichen Rendern von Polygonoberflächen hohler Objekte ohne innerer Struktur. Ohne Berücksichtigung dieser inneren Strukturen ist es z.B. nicht möglich, Beleuchtungsphänomene korrekt darzustellen, die nur durch diese Innenstruktur entstehen können. Besonders im Bereich der medizinischen Bildverarbeitung, in dem die tatsächlichen Aufnahmedaten für die medizinische Analyse nicht manipuliert werden dürfen, benötigt man verfäschungsfreie Darstellungsverfahren. Beim Polygonrendern treten aber durch die Triangulierung der Daten Verfälschungen auf. Daher ist dieses Verfahren nicht geeignet. mit Hilfe des Volumenrenderverfahrens lässt sich dieses Problem lösen, denn die Verfahren verändern die Datensätze auf dem Weg zur Darstellung nicht. Neben der medizinischen Bildverarbeitung sind Volumenrenderverfahren auch für eine realistische Visualisierung von volumentrischem Nebel, Rauch oder Feuer geeignet.

Es existieren zwei verschiedene Arten von Volumenrenderverfahren

- Raycasting
- Volume Slicing

Da diese Arbeit ein Volume Slicing Verfahren verwendet, wird hier nicht näher auf das in [CLT07] beschriebene Raycasting-Verfahren eingegangen.

5.1 Volume Slicing - texturbasiertes Volumenrendering

Beim Volume Slicing wird ein Volumen innerhalb einer Bounding Box mit Hilfe von Slices[13] dargestellt. Diese Slices sind Quad-Primitive, die zu einem Stapel geordnet durch das Volumen in der Bounding Box gelegt werden. Die einzelnen Schichten in dem Volumen, welches beispielsweise als 3D-Textur vorliegt, werden als 2D-Textur auf die Quads gelegt. mit Hilfe einer Alpha-Blending Funktion lassen sich unwichtige oder durchscheinende Regionen[14] ausblenden bzw. transparent machen. Ein Volumeneffekt wird erzielt.

Beim Rendern des Stapels kommt es auf die Position der Kamera an. Bei ungünstigem Blickwinkel könnte man durch die einzelnen Schichten hindurchschauen. Da es beim Alpha-Blending auch auf die Reihenfolge der zu rendernden Objekte ankommt, kann es passieren, dass die Schichten bei einer falschen Renderreihenfolge nicht mehr sichtbar sind. Ein falsche Renderreihenfolge tritt z.B. bei einer Position der Kamera hinter dem Stapel auf, wenn die Schichten des Stapels von hinten nach vorn gezeichnet werden. Zur Vermeidung dieser unerwünschten Effekte gibt es zwei Lösungsansätze.

5.1.1 Object-aligned Volume Slicing

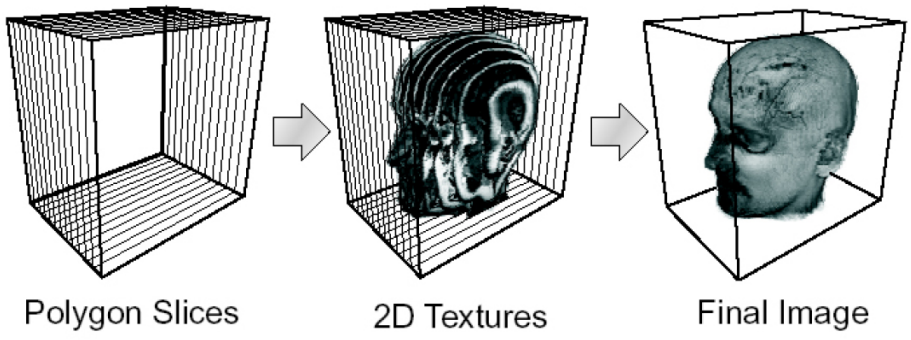

Abbildung 5: Object-aligned Volume Slicing [Wil]

[13] Scheiben
[14] Regionen ohne Partikel

Beim Object-aligned Volume Slicing (Abb. 5) wird für jede Achse $\vec{x}, \vec{y}, \vec{z}$ eines kartesischen Koordinatensystems ein Texturstapel durch das Volumen gelegt (siehe Abb 7). Die Normalen der Ebenen eines solchen Texturstapels verlaufen parallel zur jeweiligen Achse. Der Texturstapel, dessen Normale am ehesten parallel zur Blickrichtung der Kamera verläuft, wird gerendert. In Abb. 6 wird beispielsweise der Texturstapel zwischen dem Schritt C und D umgeschaltet. Zusätzlich wird für korrekt funktionierendes Alpha-Blending die Reihenfolge berücksichtigt, in der die einzelnen Schichten eines Stapels gerendert werden. Befindet sich die Kamera beispielsweise hinter dem Stapel, werden die Schichten von vorn nach hinten gezeichnet. Befindet sie sich vor dem Stapel, wird dieser von hinten nach vorn gezeichnet. Das Object-aligned Volume Slicing hat den Vorteil, einfach implementierbar zu sein.

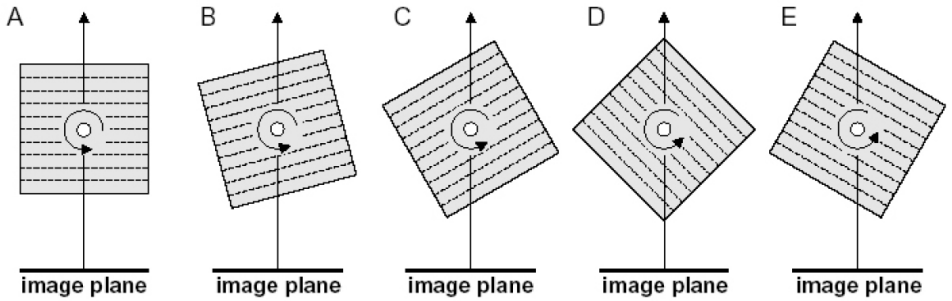

Abbildung 6: Je nach Kamerasicht kann zwischen den drei Texturstapeln umgeschaltet werden [Wil]

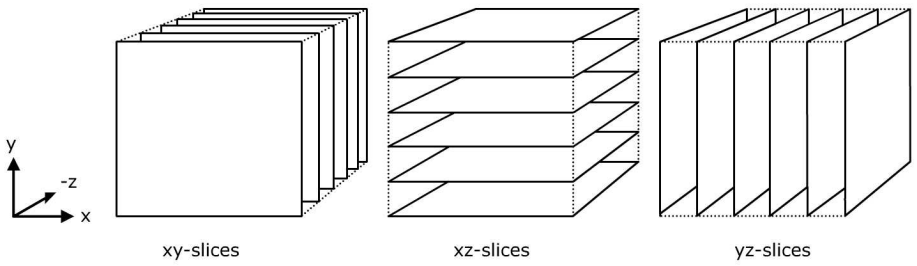

Abbildung 7: Ein Stapel für jede Achse x,y,z

5.1.2 View-aligned Volume Slicing

Das Object-aligned Volume Slicing ist einfach zu implementieren und verhindert störende Effekte wie z.B. das Hindurchschauen zwischen den Schichten oder das Verschwinden des kompletten Volumes durch eine falsche Renderreihenfolge. Das Hindurchschauen zwischen die Schichten wird jedoch nur verhindert, solange der

Öffnungswinkel[15] der Kamera auf ca. 45°-50° Grad eingestellt ist. Wird jedoch ein größerer Öffnungswinkel für eine bessere Übersicht wie z.B. 90° Grad verwendet, kann es wieder zu dem Hindurchschauen zwischen den einzelnen Schichten eines Stapels kommen, z.B. wenn sich das Volumen am Rand des Blickfeldes befindet. Im Moment des Umschaltens zwischen den Stapeln entsteht beim Object-aligned Volume Slicing ausserdem ein störender visuell-wahrnehmbaren „Sprung“ in der Darstellung (Poppingeffekt). Diese Effekte lassen sich mit dem in Abb. 8 veranschaulichten View-aligned Volume Slicing vermeiden und die visuelle Qualität steigern. Allerdings ist dieses Verfahren aufwändiger zu implementieren. Die Slices werden dabei wie in Abb. 8 so durch das Volumen gelegt, dass ihre Normalen immer parallel zur Blickrichtung der Kamera ausgerichtet sind.

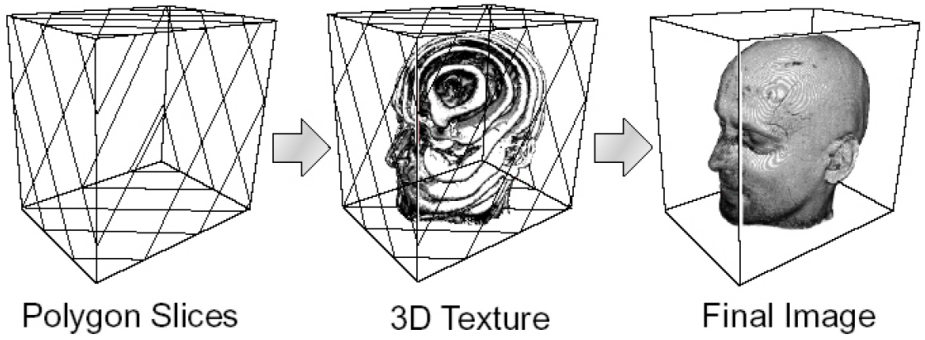

Abbildung 8: View-aligned Volume Slicing [Wil]

[15] fov

6 Softwaretechnisches Konzept und Implementierung

Da die gesamte Systemarchitektur des verwendeten Renderers sehr komplex ist, werden nur die für die Fluid-Simulation wichtigen Teile des Systems beschrieben. mit Hilfe eines in Abb. 9 dargestellten Prozess-Ablaufplans der Initialisierungen und Funktionsaufrufe soll das softwaretechnische Konzept erläutert werden.

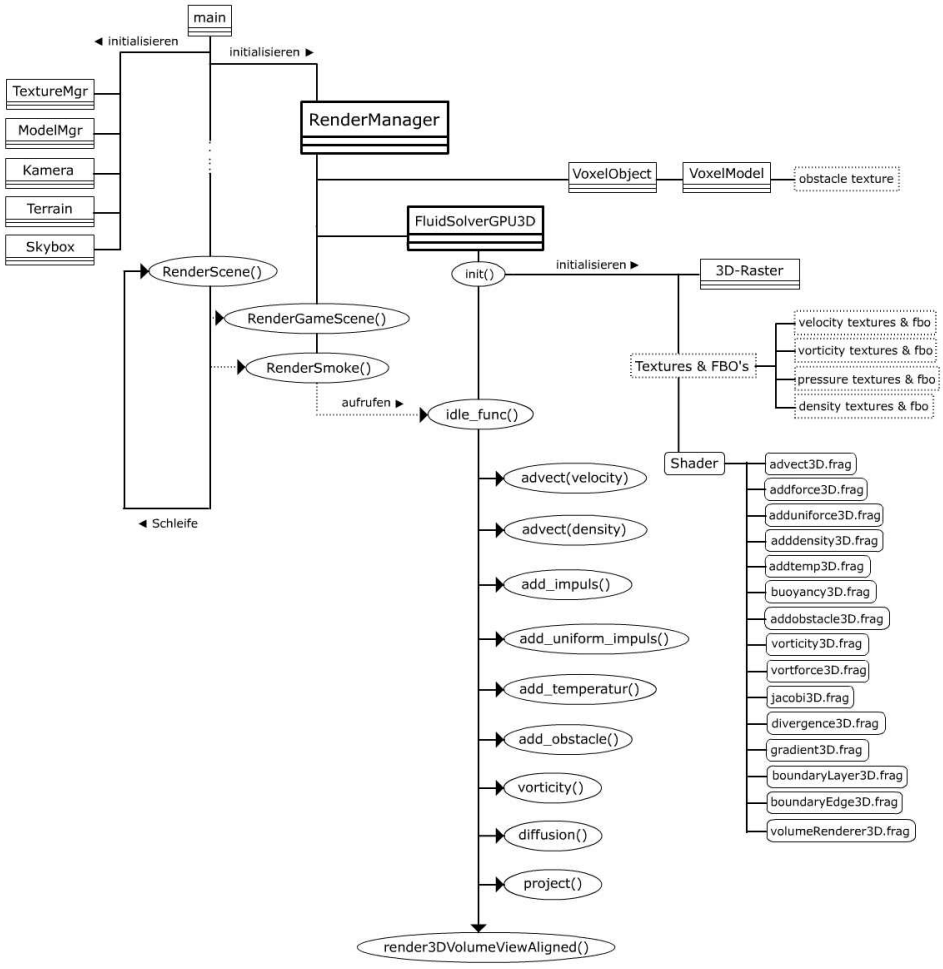

Abbildung 9: Ablaufplan eines Ausschnitts der Anwendung

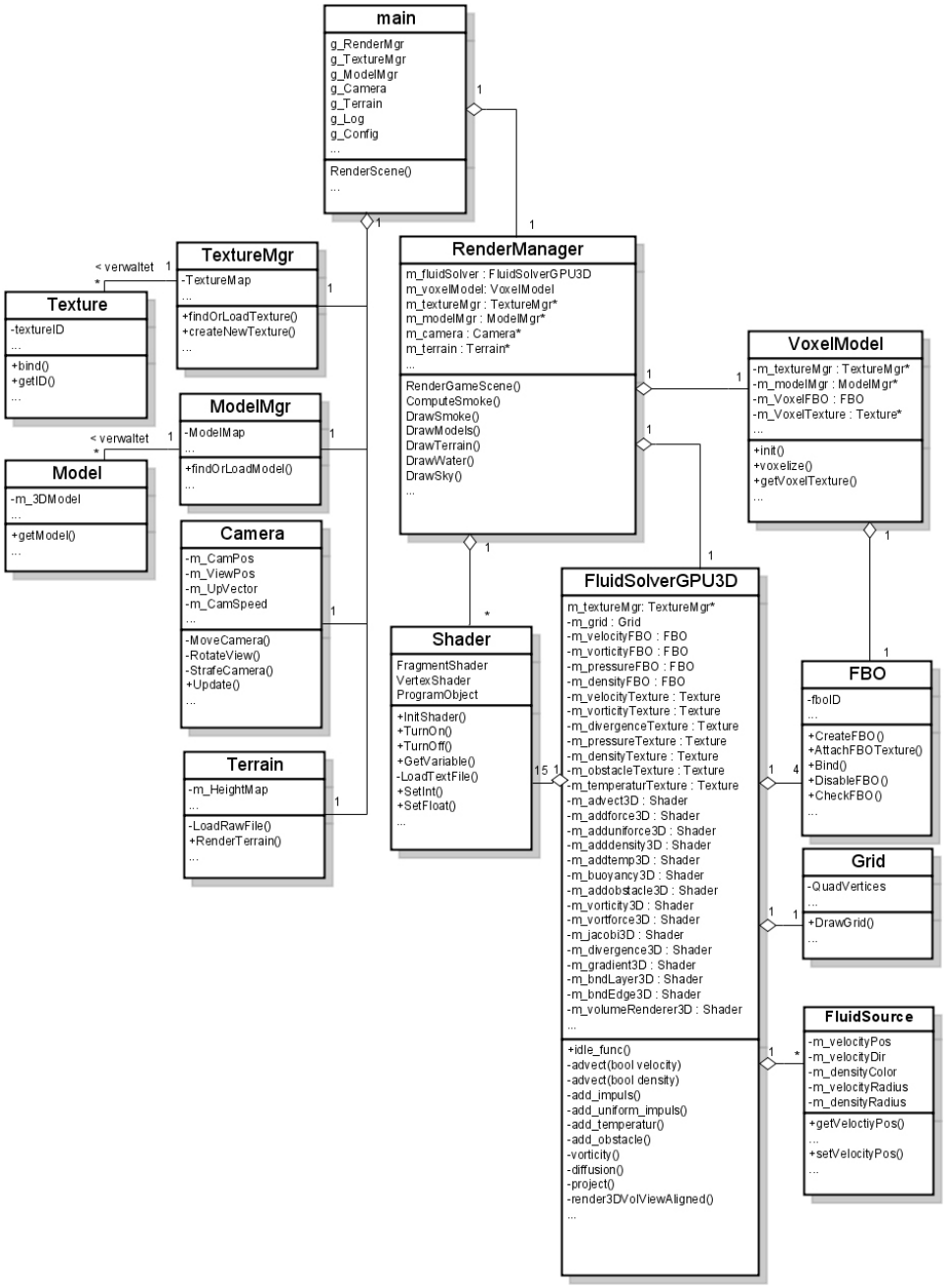

Abbildung 10: Klassendiagramm eines Ausschnitts der Anwendung

6.1 Main

Startpunkt aller Prozesse ist die main.cpp. Von hier aus werden die wichtigsten Objekte und Variablen global angelegt. In der main.cpp werden auch alle notwendigen OpenGL-Funktionen zur Initialisierung des Renderers sowie die Funktion `RenderScene()` aufgerufen. In `RenderScene()` wird in jedem Durchgang die Szene neu berechnet, d.h. von hier aus werden die Funktionen des Rendermanagers aufgerufen.

6.2 Rendermanager

In der `Init()` -Funktion des Rendermanagers, die aus der `main()` einmal aufgerufen wird, werden alle Texturen, Models, die Skybox, das Terrain, einige Shader für die Beleuchtung, der Octree, der Fluidsolver sowie der Voxelizer geladen. Der Voxelizer erzeugt die Flat 3D-Textur für die Objekte, die sich im Fluidvolumen befinden und von denen die Rauchpartikel abprallen sollen und übergibt sie dem Fluidsolver.

In jedem Durchgang der Anwendung wird aus der Funktion `RenderScene()` in der main.cpp die Funktion `RenderGameScene()` des Rendermanagers aufgerufen. In `RenderGameScene()` werden die sortierten Objekte, das Terrain und das Wasser gerendert. Aus `RenderScene()` wird ebenfalls die Funktion `RenderSmoke()` des Rendermanagers aufgerufen. In `RenderSmoke()` werden die Models, die sich im Fluidvolumen befinden, gezeichnet. Ausserdem wird die Funktion `idle_func()` aus dem Objekt `FluidSolverGPU3D` aufgerufen. `idle_func()` enthält dann alle weiteren notwendigen Funktionsaufrufe, um die Strömungsentwicklung des Fluids pro Zeitschritt sowie deren Visualisierung zu berechnen.

6.3 FluidSolverGPU3D

6.3.1 init ()

In der Funktion `Init()` des Fluidsolvers wird nach dem Zuweisen aller Variablen eine Rasterstruktur `GridLayer` als Objekt initialisiert, die beim Setzen von Koordinaten aus einem 3D-Datensatz in einen 2D-Datensatz benötigt wird. Der 2D-Datensatz ist eine Textur. Texturen werden ähnlich wie Arrays für CPU-Programme für GPU-Programme als Datenspeicher genutzt. Die Rasterstruktur hilft uns beim Einsortieren und Wiederfinden der 3D-Daten in einer dann als Flat 3D-Textur[16] (siehe Kap. 6.4.3) bezeichneten Textur. Als nächstes werden alle Texturen und zugehörigen Frame Buffer Objects erzeugt. Einem FBO können mehrere Texturen zugewiesen werden. Wir benötigen hier mindestens zwei Texturen für jedes Update eines Feldes durch ein Fragmentshader, da wir in eine Textur

[16] dt. flache 3D-Textur

nicht schreiben können, wenn sie gerade ausgelesen wird. Eine Textur wie z.B. `vel_tmp` dient als Zwischenspeicher. Es wäre auch möglich gewesen, für alle Texturen einen FBO zu verwenden, was Speicherplatz gespart hätte. Allerdings wären mehr Kopiervorgänge notwendig gewesen, was wiederum zu lasten der Gesamtperformanz gegangen wäre.

Nach dem Erzeugen der Flat 3D-Textures folgt die Initialisierung der Shader, die auf den Texturen arbeiten.

- `advect3D.frag`
  Advektionsprogramm.
- `addforce3D.frag`
  Hinzufügen externer Kräfte, z.B. ein Ventilator.
- `adduniforce3D.frag`
  Hinzufügen uniformer externer Kräfte wie die Schwerkraft.
- `addtemp3D.frag`
  Hinzufügen von Wärmequellen in die Temperaturtextur.
- `buoyancy3D.frag`
  Berechnen und Hinzufügen der Auftriebskraft.
- `addobstacle3D.frag`
  Hinzufügen von Hindernissen im Fluidvolumen.
- `vorticity3D.frag`
  Shader zur Berechnung der Wirbelstärken.
- `vortforce3D.frag`
  Shader zur Berechnung der Wirbelstärkenkraft und dem Hinzufügen dieser Kraft zum Geschwindigkeitsfeld.
- `jacobi3D.frag`
  Berechnet die Diffusion anhand des Viskositätsfaktors. Kann bei sehr kleinem Viskositätsfaktor weggelassen werden. Das Programm wird auch im Projektionsschritt eingesetzt, um aus dem Divergenzfeld das Druckfeld zu berechnen.
- `divergence3D.frag`
  Berechnet das Divergenzfeld aus dem divergenten Geschwindigkeitsfeld.
- `gradient3D.frag`
  Berechnet aus dem Druckfeld und den Gradienten und zieht ihn vom divergenten Geschwindigkeitsfeld ab. Danach ist das Geschwindigkeitsfeld divergenzfrei.

- `boundaryLayer3D.frag`
  Berechnet die Werte an den Grenzen des Fluidvolumens.
- `boundaryEdge3D.frag`
  Berechnet die Werte an den äußeren Ecken des Fluidvolumens.
- `volumeRender3D.frag`
  Sorgt für die Visualisierung des Fluids in einem dreidimensionalen Volumen.

6.3.2 idle_func()

Die `idle_func()` wird in jedem Durchgang aufgerufen. Sie enthält alle Teilschritte der Fluidberechnung. Zuerst wird der Viewport an die Auflösung der Flat 3D-Textures angepasst und die Kamera so ausgerichtet, das sie orthogonal auf die gesamte Textur schaut. Jetzt können die Shader die Texturen auslesen und mit Hilfe der FBO's in die Texturen schreiben. Danach folgen alle Funktionen, die die einzelnen Berechnungsschritte der Fluidentwicklung enthalten.

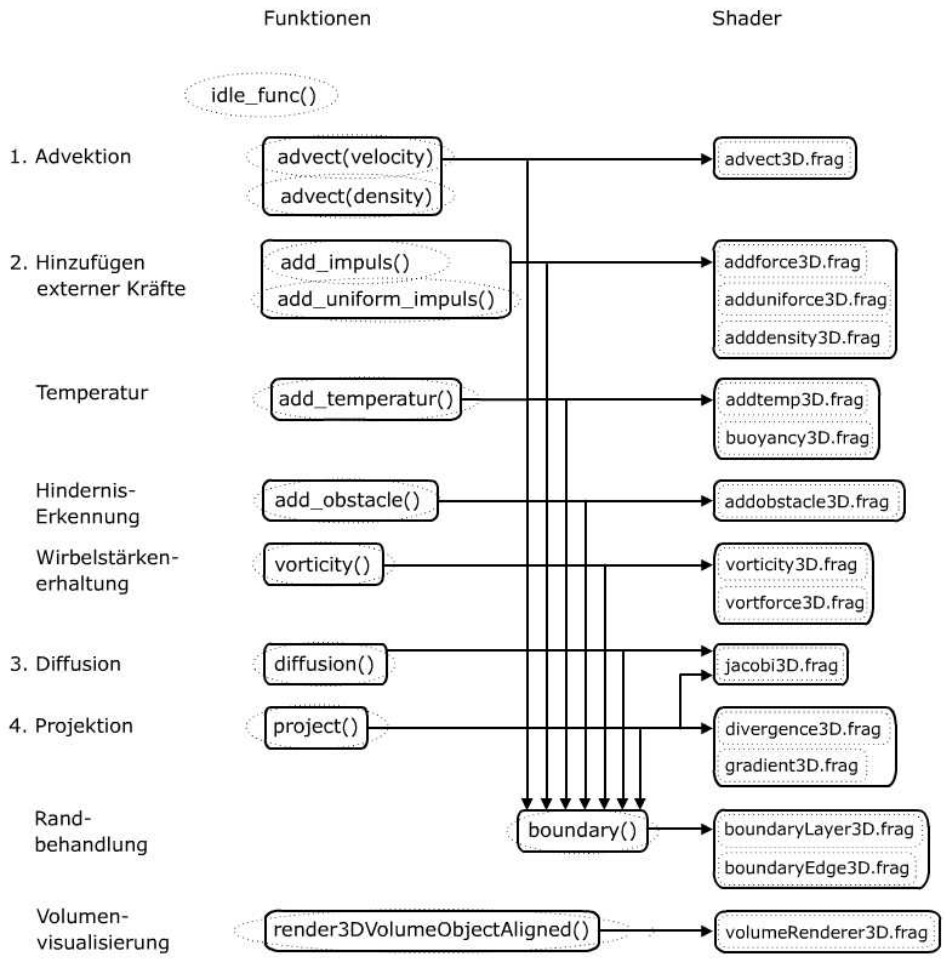

Abbildung 11: Ablauf idle_func(): Funktionen und Shader

In Abb. 11 ist der Ablauf der Funktionsaufrufe in der `idle_func` und die in der jeweiligen Funktion aufgerufenen Shader dargestellt.

6.4 Implementierung

Der Renderer nutzt die Programmiersprache C++ und die Grafik-Bibliothek OpenGL 2.1 mit einigen Erweiterungen[17]. Für die Shader wird die gut mit OpenGL kombinierbare und von ATI und Nvidia unterstützte Sprache GLSL verwendet.

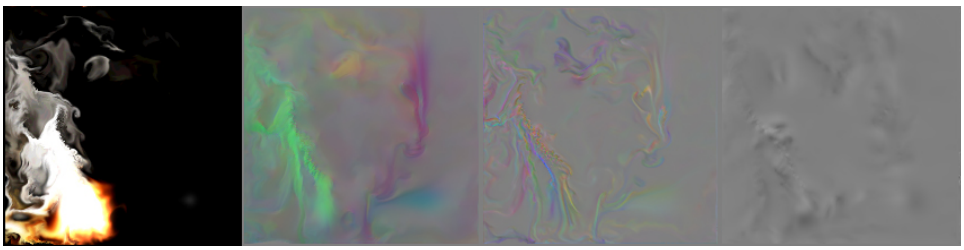

Abbildung 12: Die einzelnen in Texturen gesepeicherten Statusfelder des Fluids. Ausschnitte der Texturen von links nach rechts: Partikelfeld, Geschwindigkeitsfeld (grau ist keine Geschwindigkeit, grün ist Geschwindigkeit in positiver y-Richtung, lila in negativer y-Richtung), Wirbelstärkenerhaltung, Druckfeld (grau ist kein Druck, weiss ist hoher Druck und schwarz niedriger Druck).

6.4.1 GPU - CPU

Zum Berechnen der einzelnen Entwicklungsschritte des Fluids auf der GPU werden wie in Abb. 12 zu sehen Texturen als Datenspeicher und Fragmentshader als Schleifen, die auf den Pixeln der Texturen arbeiten, genutzt. Vergleichbar ist das mit einem CPU-Programm, in denen Schleifen auf Arrayeinträgen arbeiten. Der große Vorteil der GPU ist dabei, dass die Pixel parallel verarbeitet werden, anstatt der Reihenfolge nach, wie es bei einem CPU Programm der Fall wäre. Es können jedoch nur maximal soviele Pixel parallel verarbeitet werden, wie Shadereinheiten dafür zu Verfügung stehen.

Ein solches Textur-Update verläuft laut [Har05] in zwei Schritten.

1. Render-to-texture
   Eine Textur wird als Framebuffer genutzt. Die GPU kann somit direkt in die Textur schreiben.

2. Copy-to-texture
   Der Framebuffer wird in eine Textur kopiert (Bei Verwendung eines FBOs fällt dieser Schritt weg)

[17] z.B. für FBO's

Listing 1: Render-to-Texture Beispiel: Meine copy-Function

```
1
2  void CFluidGPU3D::copy(TexturePtr tmp, TexturePtr x, CFBO* x_fbo, int texID)
3  {
4          x_fbo->Bind();
5          glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT+texID);
6          glClearColor(0.0, 0.0, 0.0, 0.0);
7          glClear(GL_COLOR_BUFFER_BIT);
8
9          tmp->bind(GL_TEXTURE_RECTANGLE_ARB,0);
10         glBegin(GL_QUADS);
11           glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f,     0.0f);     glVertex2f(-1, -1);
12           glMultiTexCoord2fARB(GL_TEXTURE0_ARB, fullResX, 0.0f);     glVertex2f( 1, -1);
13           glMultiTexCoord2fARB(GL_TEXTURE0_ARB, fullResX, fullResY); glVertex2f( 1,  1);
14           glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f,     fullResY); glVertex2f(-1,  1);
15         glEnd();
16
17         x_fbo->DisableFBO();
18         glActiveTextureARB(GL_TEXTURE0_ARB);
19         glDisable(GL_TEXTURE_RECTANGLE_ARB);
20 }
```

Mit Hilfe der Funktion `copy(...)` in Listing 1 kann der Inhalt einer Textur in eine andere kopiert werden. Dabei wird Render-to-Texture mit Hilfe eines Frame Buffer Objects (kurz: FBO) eingesetzt. Ein FBO kann man anstelle des Bildschirm-Framebuffers zum Rendern in eine Textur verwenden. An einen FBO kann man mehrere Texturen, in die gerendert werden darf, binden. Mit Hilfe der Anweisungen in Zeile 4 bis 6 legen wir den zu verwendenden FBO und die Textur, in die gerendert werden soll, fest. Anschließend löschen wir alle Werte im Color Buffer des FBOs. In Zeile 9 bis 15 wird ein Viewport (Bildschirm)-füllendes Quad mit der Textur, die kopiert werden soll, gezeichnet. Man könnte jetzt noch einen Shader auf das Quad mit der Textur anwenden. Das Quad wird direkt in die im FBO aktivierte Textur geschrieben. In Zeile 17 wird der FBO ausgeschaltet und der Framebuffer für den Bildschirm eingeschaltet. Nachdem der FBO ausgeschaltet wurde, kann man die Textur, in die geschrieben wurde, wieder lesen. Der Copy-to-Texture Schritt fällt weg.

6.4.2 Randpixel - Innenpixel

Da wir die Pixel am Rand (Abb. 13) einer Textur anders behandeln als die Innenpixel, benötigen wir für jedes Texturupdate zwei Durchgänge. Im ersten Durchgang werden alle Innenpixel aktualisiert und im zweiten Durchgang alle Randpixel mit Hilfe eines eigenen Fragmentshaders für die Randbehandlung.

6.4.3 3D Textur - Flat 3D-Textur

Ältere GPUs unterstützten zwar das Rendern von 3D-Texturen, konnten aber keine Daten in eine 3D-Textur schreiben. Daher etablierten sich die Flat 3D-Texturen. Flat 3D-Texturen sind eine praktikable und effiziente Methode zum Arbeiten mit 3D-Datensätzen auf der GPU, daher werden sie in dieser Arbeit verwendet. Allerdings unterstützen aktuelle GPUs auch das Schreiben in echte 3D-Texturen.
Da wir mit einem Fluidvolumen arbeiten, müssen wir den jeweiligen Fragmentshader für jede Schicht im Volumen ausführen. Wir verwenden eine 2D-Textur, die unsere 3D-Textur repräsensiert, eine Flat 3D-Textur (Abb. 14), in der alle N

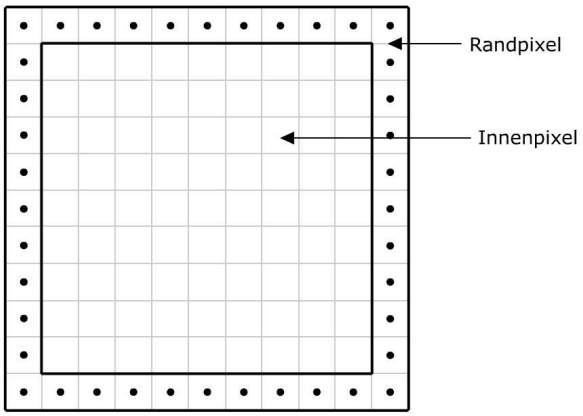

Abbildung 13: Randpixel - Innenpixel nach [Dep08]

Schichten des Volumens enthalten sind. Wir nutzen zum Ausführen eines Fragmentshaders auf einer Schicht eine Schleife über alle Schichten N. N entspricht dabei der Auflösung der Tiefe z des Volumens, d.h. bei einem würfelförmigen Volumen mit Kantenlänge 64 gibt es auch 64 Schichten. Die Flat 3D-Textur wäre dann $(64 \times 8) \times (64 \times 8) = 512 \times 512$ Pixel.[Dep08]

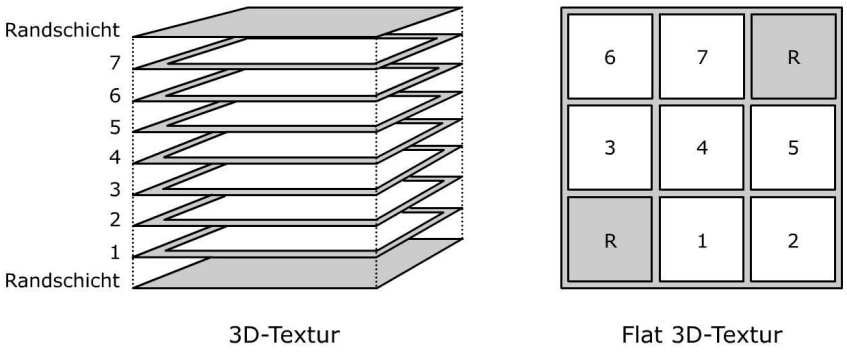

Abbildung 14: 3D-Raster in einer Flat 3D-Textur nach [Dep08]

6.4.4 Early-Z Culling

mit Hilfe des Early-Z Culling Features der GPU lassen sich die Anzahl der Operationen der Jacobi-Iterationen reduzieren. An Stellen im Fluid mit wenig Druck bzw. an Stellen mit relativ geringen Unterschieden der Nachbarschaftswerte kann die Berechnung der Werte mit Hilfe der Poisson-Gleichung übersprungen werden, da sich keine Veränderung der Werte ergibt. Die Zahl der zu berechnenden Frag-

mente auf der GPU werden durch Ausschluss mittels des Early-Z Culling weniger. Nur die wichtigsten Regionen mit stark unterschiedlichen Nachbarschaftswerten werden noch berechnet. Dadurch wird die Anzahl der Operationen wärend des vergleichsweise aufwendigen Berechnungsprozess der Poisson-Gleichung minimiert [Dep08].

6.5 Fragmentshader

6.5.1 Advektion

Das Advektionsprogramm ist der erste Fragmentshader innerhalb der `idle_func()`. Es transportiert die in der Textur `vel` gespeicherte Geschwindigkeit und speichert sie mit Hilfe des Framebuffer `vel_fbo` in Buffer-Textur `vel_tmp`, die anschließend mit Hilfe der in Kapitel Render-to-texture beschriebenen Funktion `copy()` wieder in die Textur `vel` geschrieben wird. Die Randbehandlung wird gesondert mit Hilfe der Funktion `boundary()` durchgeführt. Als Eingabedaten werden bei der Selbst-Advektion der Geschwindigkeit das Geschwindigkeitsfeld `vel` und als zu transportierendes Medium ebenfalls `vel` verwendet. Das Advektionsprogramm wird analog zur Selbst-Advektion noch ein zweites mal zum Transport der Partikel genutzt. Als Medium wird das Partikelfeld `dens` verwendet.

Listing 2: advect3D.frag

```
1  uniform float dt;                    // Zeitschritt = 0.1
2  uniform float adj;                   // Korrekturfaktor = 0.944
3  uniform float dissipation;           // Verlustfaktor dissipation: 0.0997
4  uniform float l;                     // aktuelle Schicht entlang der z-Achse
5  uniform vec3  resolution;
6  uniform sampler2DRect velocity;      // Geschwindigkeitsfeld
7  uniform sampler2DRect media;         // Medium
8  uniform sampler2DRect offsetTexture; // Hilfstextur um die richtige Schicht
9                                       // in z-Richtung der Flat 3D-Textur zu finden
10
11 vec4 texture2DRectbilerp(in sampler2DRect media, in vec2 s)
12 {
13         vec4 uv;
14         uv.xy = floor(s-0.5)+0.5;
15         uv.zw = uv.xy + 1.0;
16
17         vec2 t = (s - uv.xy);
18
19         vec4 tex11 = texture2DRect(media, uv.xy); //(-1,-1)
20         vec4 tex21 = texture2DRect(media, uv.zy); //( 1,-1)
21         vec4 tex12 = texture2DRect(media, uv.xw); //(-1, 1)
22         vec4 tex22 = texture2DRect(media, uv.zw); //( 1, 1)
23
24         // bilineare Interpolation
25         return mix(mix(tex11, tex21, t.x), mix(tex12, tex22, t.x), t.y);
26 }
27
28 vec4 advect3Drect()
29 {
30         //Rückverfolgen des Geschwindigkeitsfelds
31         vec3 vel = -(texture2DRect(velocity, gl_TexCoord[0].xy).rgb
32                    - vec3(0.5, 0.5, 0.5)* adj) * dt * resolution;
33
34         float zValue = l + vel.z;
35
36         float zVal1 = floor(zValue);
37         float zVal2 = zVal1 + 1.0;
38
39         vec2 zOffset1 = texture2DRect(offsetTexture, vec2(zVal1, 0.5)).rg;
40         vec2 zOffset2 = texture2DRect(offsetTexture, vec2(zVal2, 0.5)).rg;
41
42         vec2 xyVal = gl_TexCoord[0].zw + vel.xy;
43         xyVal = min(max(vec2(-1.0,-1.0),xyVal),resolution.xy + 1.0);
44
```

```
45          vec4 iSample1 = texture2DRectbilerp(media, zOffset1 + xyVal);
46          vec4 iSample2 = texture2DRectbilerp(media, zOffset2 + xyVal);
47
48          //trilineare Interpolation
49          vec4 newVal = mix(iSample1, iSample2,
50                            (zValue/resolution.z - zVal1/resolution.z)*resolution.z);
51
52          return newVal;
53  }
54
55  void main()
56  {
57          gl_FragColor = advect3Drect();
58  }
```

Die Funktion `advect3Drect()` führt die Advektion durch. Das Geschwindigkeitsfeld und das Feld mit dem Transportmedium wird als Flat 3D-Textur `velocity` und `media` übergeben. `dt` gibt die Zeitspanne eines Durchgangs an. Um numerische Genauigkeitsunterschieden zwischen verschiedenen GPU's entgegenzuwirken, verwende ich noch ein zur Laufzeit justierbaren zusätzlichen Parameter `adj`. Im Parameter `l` ist die aktuelle Rasterschichtnummer gespeichert. Zusammen mit der vorberechneten Textur `offsetTexture`, in der zu jeder Rasterschichtnummer die zugehörigen Start-Texturkoordinaten der Flat 3D-Textur gespeichert sind, kann der Fragmentshader jeden beliebigen Voxel im Raster erreichen. Der Verlustfaktor `dissipation` bestimmt die Reichweite des zu transportierenden Mediums bevor es verschwindet.

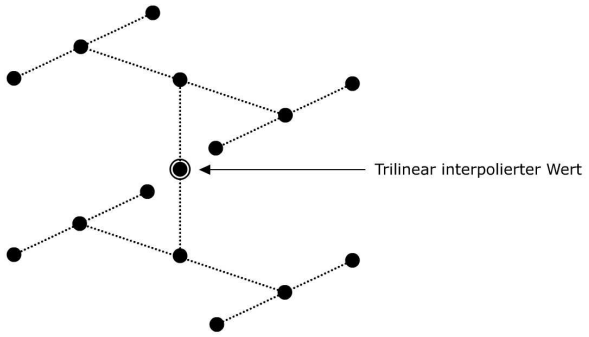

Abbildung 15: Trilineare Interpolation

In `resolution` ist die Auflösung der Flat 3D-Textur gespeichert. Da aktuelle GPU's noch keine billineare Interpolation von floating-point Texturen unterstützen, wird diese mit Hilfe der Funktion `texture2DRectbilerp(...)` durchgeführt. Dabei werden die vier nächstgelegenen Texel der übergebenen Texturkoordinate billinear interpoliert. Anschließend wird in `advect3Drect()` noch entlang der dritten Koordinate interpoliert.

6.5.2 Beschleunigung

Der Einfluss externer Kräfte findet nach der Advektion des Geschwindigkeitsfeldes statt. Ein Impuls einer externen Kraftquelle wird beispielsweise mit Hilfe des

Fragmentshaders in Listing 3 dem Geschwindigkeitsfeld hinzugefügt. Eine uniforme Kraft wie die Schwerkraft kann ebenfalls Einfluss nehmen. Analog dem Hinzufügen eines Impulses zum Geschwindigkeitsfeld werden auch neue Partikel mit Hilfe des Fragmentshaders adddensity3D.frag dem Partikelfeld zugeführt.

Listing 3: addforce3D.frag

```
1
2  uniform vec3  sourcepos;      // Position der Kraftquelle im Fluid
3  uniform vec4  sourceDir;      // Staerke der Kraftquelle
4  uniform vec3  windowDims;
5  uniform float sourceRadius;   // Radius der Kraftquelle
6  uniform float l;              // Aktuelle Schicht im Fluid
7
8  uniform sampler2DRect v;
9
10 float gaussian(in vec3 pos, in float radius)
11 {
12         return exp(-dot(pos,pos)/radius);
13 }
14
15 vec4 splat()
16 {
17         vec3 pos  = windowDims * sourcepos - vec3(gl_TexCoord[0].zw,l);
18         float rad = windowDims.x * sourceRadius;
19         vec4 c    = vec4(sourceDir.rgb, clamp(dot(sourceDir,vec4(1.0,1.0,1.0,1.0)),0.0,1.0));
20
21         return texture2DRect(v,gl_TexCoord[0].xy) + c * gaussian(pos,rad);
22 }
23
24 void main()
25 {
26         gl_FragColor = splat();
27 }
```

6.5.3 Temperatur

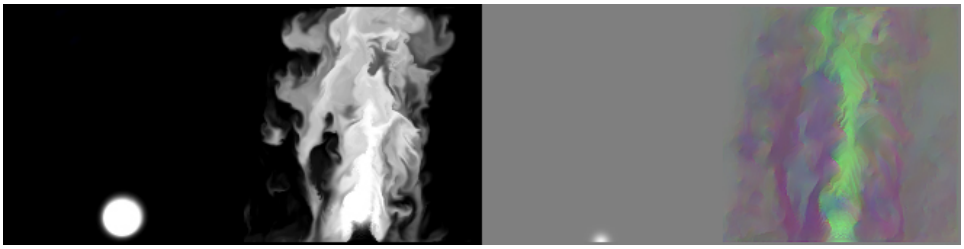

Abbildung 16: Von links nach rechts: Partikelquelle, Dichte (Partikel), Wärmequelle, Geschwindigkeitsfeld. Im Geschwindigkeitsfeld ist der Auftrieb grün zu erkennen. Lila eingefärbte Pixel markieren eine Abwärtsbewegung, verursacht durch die Gravitation, die auf die Partikel wirkt.

Wie bereits in Kap. 4.6.2 beschrieben, wird eine Temperaturtextur erstellt, in die zunächst die Wärmequellen mit Hilfe des Fragmentshaders in Listing 4 geschrieben werden. Anschließend wird die Auftriebskraft $\vec{F}_{buo}$ aus Gleichung 4 im Fragmentshader in Listing 5 mit Hilfe der Werte aus Temperaturtextur und Dichtetextur berechnet und dem Geschwindigkeitsfeld hinzugefügt (siehe Abb. 16). Die Umgebungstemperatur und die Skalierungsfaktoren σ und μ können zur Laufzeit verändert werden.

Listing 4: addtemp3D.frag

```
1
2  uniform vec3  sourcePos;      // Position der Wärmequelle
3  uniform float T;              // Temperatur der Wärmequelle
4  uniform vec3  windowDims;
5  uniform float sourceRadius;   // Radius der Waermequelle
6  uniform float l;              // Aktuelle Schicht im Fluid
7
8  uniform sampler2DRect t;      // Temperaturtextur
9
10 float gaussian(in vec3 pos, in float radius)
11 {
12         return exp( -dot(pos, pos) / radius );
13 }
14
15 vec4 splat()
16 {
17         vec3 pos  = windowDims * sourcePos - vec3(gl_TexCoord[0].zw, l);
18         float rad = windowDims.x * sourceRadius;
19         vec4 c    = vec4(T, T, T, clamp(dot(sourceTemp, vec4(1.0, 1.0, 1.0, 1.0)), 0.0, 1.0));
20
21   return texture2DRect(t, gl_TexCoord[0].xy) + c * gaussian(pos, rad);
22 }
23
24 void main()
25 {
26         gl_FragColor = splat();
27 }
```

Listing 5: buoyancy3D.frag

```
1
2  uniform float ambientTemp;   // Umgebungstemperatur
3  uniform float sigma;         // konstanter Skalierungsfaktor
4  uniform float mu;            // konstanter Skalierungsfaktor für die Partikelmasse
5  uniform vec3  dir;           // Zeigt die vertikale Richtung
6  uniform sampler2DRect t;     // Temperaturfeld
7  uniform sampler2DRect d;     // Dichtefeld
8  uniform sampler2DRect v;     // Geschwindigkeitsfeld
9
10 vec4 buoyancy()
11 {
12         //Temperatur
13         float tC = texture2DRect(t, gl_TexCoord[0].xy).r - 0.5;
14
15         //Dichte
16         vec4 dC = texture2DRect(d, gl_TexCoord[0].xy);
17
18         //Geschwindigkeit
19         vec4 vC = texture2DRect(v, gl_TexCoord[0].xy) - vec4(0.5,0.5,0.5,0.0);
20
21         //Berechnet die Auftriebskraft
22         vec4 fbuo = (-mu*dC.a + sigma*(tC - ambientTemp) ) * dir;
23
24         //Fuegt die Auftriebskraft dem Geschwindigkeitsfeld hinzu
25         vec4 u = vC + fbuo;
26
27         return u + vec4(0.5,0.5,0.5,0.0);
28 }
29
30 void main()
31 {
32         gl_FragColor = buoyancy();
33 }
```

6.5.4 Hindernisobjekte

Der Ablauf der Fluidsimulation verändert sich, wenn sich Hindernisobjekte im Fluidvolumen befinden. Die Partikeldichte der Voxel im Hindernisobjekt beträgt Null, die Geschwindigkeitswerte dieser Voxel entsprechen der Geschwindigkeit des Objekts. Die Bedingungen an den Rändern des Hindernisobjekts werden gesondert behandelt. In der Simulation wird die free-slip Bedingung verwendet. Sie verhindert, dass das Fluid in das Hindernisobjekt eindringt. Es kann stattdessen,

wie in Abb. 17 dargestellt, frei entlang der Oberfläche des Hindernisobjekts fließen, wobei die Reibung hier ignoriert wird. Die free-slip Randbehandlung kann nach der Druckberechnung im Projektionsschritt angewandt werden. Das Ergebnis des Projektionsschritts wird korrigiert. Wenn das aktuelle Voxel dem Rand eines Hindernisobjekts angehört, verwenden wir dessen Geschwindigkeitskomponente für das benachbarte Voxel im Fluid. Das setzt die Erkennung der Voxel, die innerhalb des Hindernisobjekts liegen sowie die Bestimmung ihrer Geschwindigkeit voraus. Die Voxel innerhalb eines Hindernisobjekts werden daher vor dem Ablauf der Simulation bestimmt und in eine Flat 3D-Textur gespeichert. Diesen Prozess nennt man Voxelisierung. Die Flat 3D-Textur mit den Voxelisierungsdaten kann anschließend wie eine externe Kraft einfach dem Geschwindigkeitsfeld und dem Partikelfeld hinzugefügt werden [Dep08].

Abbildung 17: Hindernisobjekt im Fluid

6.5.5 Voxelisierung eines Objekts

Für einen Teil der Voxelisierung eines Objekts (Innen-Außeninformationen) verwenden wir den Voxelisierungsalgorithmus aus [Dep08]. Dabei wird ein separates Raster in Form einer RGBA Flat 3D-Textur erstellt, welches die Voxel, die sich innerhalb eines Objekts befinden, speichert. Dafür rendert man das Objekt in einer Schleife über jede Schicht des Rasters mit Hilfe einer orthografischen Projektion. Die far clipping plane wird auf den Wert unendlich eingestellt und die near clip-

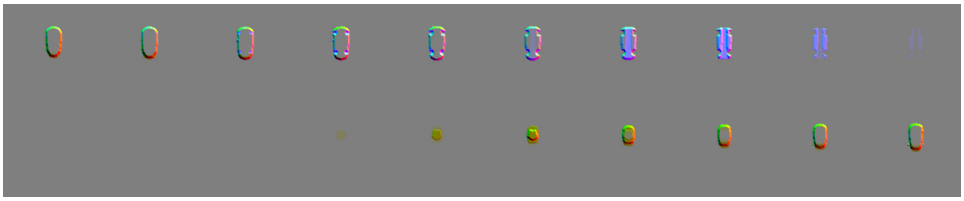

Abbildung 18: Die einzelnen Schichten der Voxeltextur als Ergebnis der Voxelisierung eines Autos. Die Normalen (als Farben zu sehen) werden mit Hilfe eines Fragmentshaders berechnet, der einen Gradienten auf die Innen-Außeninformationen aus dem Alpha-Kanal der Voxeltextur anwendet.

ping plane auf den Tiefenwert der aktuellen Schicht. Pro Schicht wird das Objekt zweimal gerendert. Zuerst wird das Objekt mit back face culling gerendert und ein Shader schreibt in jedes Voxel, in dem das Objekt zu sehen ist den Wert -1 in den Alpha-Kanal. Im zweiten Schritt wird das Objekt noch einmal mit front face culling gerendert und mit Hilfe des Shaders wird in jedes Voxel, in dem das Objekt zu sehen ist, der Wert 1 in den Alpha-Kanal geschrieben. Jetzt steht in jedem Voxel der aktuellen Schicht, das sich innerhalb des Objekts befindet, eine 1 im Alpha-Kanal. In allen übrigen steht eine 0 im Alpha-Kanal. Damit das Verfahren korrekt funktioniert, muss das Objekt eine geschlossene Hülle besitzen. Die Normalen und die Geschwindigkeit eines Objekts können wir in den ersten drei Komponenten RGB der Flat 3D-Textur speichern (siehe Abb. 18). Die Geschwindigkeit eines starren Körpers kann man über die Differenz der Vertexpositionen eines Zeitschritts ermitteln [Dep08]. Für deformierbare Objekte benötigt man eine speziellere Behandlung (siehe dazu [CLT07]).

6.5.6 Wirbelstärke

Eine spezielle Form externer Kraftquellen ist die in [FSWJ01] beschriebene Wirbelstärkenkraft. mit Hilfe dieser Kraft können wir die kleineren detaillierten Verwirbelungen wieder zum Geschwindigkeitsfeld hinzufügen, die sonst durch den Advektionsschritt verloren gehen würden. In Listing 6 ist der Fragmentshader zum Berechnen der Wirbelstärke des Geschwindigkeitsfeldes zu sehen. Listing 7 zeigt den Fragmentshader für die Berechnung der Wirbelstärkenkraft.

Listing 6: vorticity3D.frag

```
1  uniform sampler2DRect u;    //Geschwindigkeitsfeld
2
3  vec4 vorticity3D()
4  {
5          vec4 L = texture2DRect(u, gl_TexCoord[0].xy - vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0);
6          vec4 R = texture2DRect(u, gl_TexCoord[0].xy + vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0);
7          vec4 T = texture2DRect(u, gl_TexCoord[0].xy + vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0);
8          vec4 B = texture2DRect(u, gl_TexCoord[0].xy - vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0);
9          vec4 U = texture2DRect(u, gl_TexCoord[1].xy) -vec4(0.5,0.5,0.5,0.0);
10         vec4 D = texture2DRect(u, gl_TexCoord[1].zw) -vec4(0.5,0.5,0.5,0.0);
11         vec4 C = texture2DRect(u, gl_TexCoord[0].xy);
12
13         vec3 vort = vec3((T.z-B.z)-(U.y-D.y), (R.z-L.z)-(U.x-D.x), (R.y-L.y)-(T.x-B.x));
14
15         return vec4(vort+vec3(0.5, 0.5, 0.5), C.a);
16 }
```

```
17
18 void main()
19 {
20         gl_FragColor = vorticity3D();
21 }
```

Listing 7: vortForce3D.frag

```
1  uniform float epsilon;       // Skalierung der Wirbelstärkenkraft
2  uniform float timestep;
3  uniform sampler2DRect vort;  // Wirbelstärke
4  uniform sampler2DRect u;     // Geschwindigkeit
5
6  vec4 vortForce3D()
7  {
8          //Auslesen der Werte Wirbelstärke in den benachbarten Voxeln
9          vec4 L = abs(texture2DRect(vort, gl_TexCoord[0].xy - vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0));
10         vec4 R = abs(texture2DRect(vort, gl_TexCoord[0].xy + vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0));
11         vec4 T = abs(texture2DRect(vort, gl_TexCoord[0].xy + vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0));
12         vec4 B = abs(texture2DRect(vort, gl_TexCoord[0].xy - vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0));
13         vec4 U = abs(texture2DRect(vort, gl_TexCoord[1].xy) -vec4(0.5,0.5,0.5,0.0));
14         vec4 D = abs(texture2DRect(vort, gl_TexCoord[1].zw) -vec4(0.5,0.5,0.5,0.0));
15         vec4 C = texture2DRect(vort, gl_TexCoord[0].xy) -vec4(0.5,0.5,0.5,0.0);
16
17         // Berechne den Gradienten des Wirbelstärkenfeldes
18         vec3 gradVort = vec3(R.x-L.x, T.y-B.y, U.z-D.z);
19
20         vec3 V = vec3(0.0,0.0,0.0);
21         vec3 vorticityForce = vec3(0.0,0.0,0.0);
22
23         float w = dot(gradVort,gradVort);
24
25         if(w!=0)
26         {
27                 V = normalize(gradVort);
28                 vorticityForce = epsilon * cross(V, C.xyz);
29         }
30
31         vec3 uNew = texture2DRect(u, gl_TexCoord[0].xy).rgb-vec3(0.5,0.5,0.5);
32         uNew += timestep * vorticityForce;
33
34         return vec4(uNew.r+0.5, uNew.g+0.5, uNew.b+0.5, C.a);
35 }
36
37 void main()
38 {
39         gl_FragColor = vortForce3D();
40 }
```

6.5.7 Druck und Diffusion

In Kap. 4.7.5 zur Lösung der Poisson-Gleichung werden die Gleichung 7 für Druck und Gleichung 12 für Diffusion als diskretisierte Poisson-Gleichungen vorgestellt, die sich beide auch als lineares Gleichungssystem der Form $A\vec{x} = \vec{b}$ schreiben lassen. Dabei ist A ein Matrix, $\vec{x}$ der Vektor von Unbekannten und $\vec{b}$ der Vektor mit den bereits bekannten Werten. Ich verwende die in Kap. 4.7.5 bereits vorgestellte einfach Jacobi-Iteration als Technik zum Lösen der Gleichungssysteme. Ein iterativer Schritt zum Lösen der beiden Gleichungen für den Druck und die Diffusion kann wie in der Form der Gleichung 14

$$\vec{x}^{n+1}_{i,j,k} = \frac{\vec{x}^{n}_{i-1,j,k} + \vec{x}^{n}_{i+1,j,k} + \vec{x}^{n}_{i,j-1,k} + \vec{x}^{n}_{i,j+1,k} + \vec{x}^{n}_{i,j,k-1} + \vec{x}^{n}_{i,j,k+1} + \alpha\vec{b}_{i,j,k}}{\beta}$$

aus Kap. 4.7.5 beschrieben werden. In die Druckgleichung setzen wir für $\vec{x} = p \cdot t$, für $\vec{b} = \nabla \cdot \vec{w}$, für $\alpha = (\delta x)^2$ und für $\beta = 6$ ein. In die Diffusionsgleichung werden

für $\vec{x}$ und $\vec{b}$ das Geschwindigkeitsfeld $\vec{w}$ eingesetzt. α wird auf $\frac{(\delta x)^2}{\nu \delta t}$ und β auf $6+\alpha$ gesetzt [Dep08].

Listing 8: jacobi3D.frag

```
uniform float alpha;
uniform float rBeta;
uniform float obstacle; //Hindernisserkennung an/aus

uniform sampler2DRect x;
uniform sampler2DRect b;

vec4 jacobi()
{
    //Auslesen der Werte der benachbarten Voxel
    vec4 xL = texture2DRect(x, gl_TexCoord[0].xy - vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0);
    vec4 xR = texture2DRect(x, gl_TexCoord[0].xy + vec2(1.0, 0.0))-vec4(0.5,0.5,0.5,0.0);
    vec4 xT = texture2DRect(x, gl_TexCoord[0].xy + vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0);
    vec4 xB = texture2DRect(x, gl_TexCoord[0].xy - vec2(0.0, 1.0))-vec4(0.5,0.5,0.5,0.0);
    vec4 xU = texture2DRect(x, gl_TexCoord[1].xy) -vec4(0.5,0.5,0.5,0.0);
    vec4 xD = texture2DRect(x, gl_TexCoord[1].zw) -vec4(0.5,0.5,0.5,0.0);

    //Lese den Wert von b aus dem aktuellen Voxel
    vec4 bC = texture2DRect(b, gl_TexCoord[0].xy)-vec4(0.5,0.5,0.5,0.0);

    //Schreibe Null in die Hindernisobjekt-Voxel
    if(obstacle==1){
      vec4 xC = texture2DRect(x, gl_TexCoord[0].xy)-vec4(0.5,0.5,0.5,0.0);
      if(xL.a == 1) xL = xC;
      if(xR.a == 1) xR = xC;
      if(xT.a == 1) xT = xC;
      if(xB.a == 1) xB = xC;
      if(xU.a == 1) xU = xC;
      if(xD.a == 1) xD = xC;
    }

    //Auswerten der Jacobi-Iteration
    vec4 jac = (xL + xR + xT + xB + xU + xD + alpha * bC) * rBeta;

          return jac + vec4(0.5, 0.5, 0.5, 0.0);
}

void main()
{
          gl_FragColor = jacobi();
}
```

In Listing 8 ist der zuständige Fragmentshader für die Berechungen der Jacobi-Iteration zu sehen. Die Vektorfelder von $\vec{x}$ und $\vec{b}$ werden als Flat 3D-Texturen, α und β als uniforme floating-point Parameter übergeben. Um das skalare Druckfeld p berechnen zu können benötigen wir noch die Divergenz des Geschwindigkeitsfeldes. Die Divergenz wird im Fragmentshader in Listing 9 berechnet.

Listing 9: divergence3D.frag

```
uniform sampler2DRect w;    // Vectorfeld, Geschwindigkeit

vec4 divergence()
{
        vec4 L = texture2DRect(w, gl_TexCoord[0].xy - vec2(1.0, 0.0));
        vec4 R = texture2DRect(w, gl_TexCoord[0].xy + vec2(1.0, 0.0));
        vec4 T = texture2DRect(w, gl_TexCoord[0].xy + vec2(0.0, 1.0));
        vec4 B = texture2DRect(w, gl_TexCoord[0].xy - vec2(0.0, 1.0));
        vec4 U = texture2DRect(w, gl_TexCoord[1].xy);
        vec4 D = texture2DRect(w, gl_TexCoord[1].zw);

        vec4 C = texture2DRect(w, gl_TexCoord[0].xy);

        float div = (R.x - L.x + T.y - B.y + U.z - D.z) * 0.5;

        return vec4(div+0.5, 0.5, 0.5, C.a);
}

void main()
{
        gl_FragColor = divergence();
}
```

6.5.8 Projektion

Im Kap. 4.7.1 zur Helmholtz-Hodge Zerlegung erhält man laut Gleichung 6 ein divergenzfreies Geschwindigkeitsfeld, wenn man von einem divergenten Geschwindigkeitsfeld den Gradienten eines Skalarfeldes p abzieht. Verwenden wir das Druckfeld als Skalarfeld p, können wir die geforderte Masseerhaltung gewährleisten und den Druck-Term von den Unbekannten der Navier-Stokes-Gleichung zur Impulserhaltung wie in Kap. 4.7.1 beschrieben entfernen. Listing 10 zeigt den Fragmentshader zu diesem Projektionsschritt.

Listing 10: gradient3D.frag

```
uniform sampler2DRect p; // Druckfeld
uniform sampler2DRect w; // Geschwindigkeitsfeld

vec4 grad()
{
        float pC = texture2DRect(p, gl_TexCoord[0].xy).r - 0.5;
        float pL = texture2DRect(p, gl_TexCoord[0].xy - vec2(1.0, 0.0)).r - 0.5;
        float pR = texture2DRect(p, gl_TexCoord[0].xy + vec2(1.0, 0.0)).r - 0.5;
        float pT = texture2DRect(p, gl_TexCoord[0].xy + vec2(0.0, 1.0)).r - 0.5;
        float pB = texture2DRect(p, gl_TexCoord[0].xy - vec2(0.0, 1.0)).r - 0.5;
        float pU = texture2DRect(p, gl_TexCoord[1].xy).r - 0.5;
        float pD = texture2DRect(p, gl_TexCoord[1].zw).r - 0.5;

        // Auslesen der Werte im Zentrumsvoxel und aus den benachbarten Voxeln
        // Verwendung der Werte aus den benachbarten Voxeln fuer die free-slip Randbedingung
        // an den Raendern der Hindernisobjekte
        vec4 wC = texture2DRect(w, gl_TexCoord[0].xy) - vec4(0.5,0.5,0.5,0.0);
        vec4 wL = texture2DRect(w, gl_TexCoord[0].xy - vec2(1.0, 0.0)) - vec4(0.5,0.5,0.5,0.0);
        vec4 wR = texture2DRect(w, gl_TexCoord[0].xy + vec2(1.0, 0.0)) - vec4(0.5,0.5,0.5,0.0);
        vec4 wT = texture2DRect(w, gl_TexCoord[0].xy + vec2(0.0, 1.0)) - vec4(0.5,0.5,0.5,0.0);
        vec4 wB = texture2DRect(w, gl_TexCoord[0].xy - vec2(0.0, 1.0)) - vec4(0.5,0.5,0.5,0.0);
        vec4 wU = texture2DRect(w, gl_TexCoord[1].xy) - vec4(0.5,0.5,0.5,0.0);
        vec4 wD = texture2DRect(w, gl_TexCoord[1].zw) - vec4(0.5,0.5,0.5,0.0);

        // Geschwindigkeitsfeld und Maske des Geschwindigkeitsfelds durch
        // die free-slip Randbedingung
        vec3 obstV = vec3(0,0,0);
        vec3 vMask = vec3(1,1,1);

        // 1. Elimination des unkorrekten Druck-Gradient Terms
        //    (d.h. Druck Werte im Hindernisobjekt werden ignoriert)
        // 2. berechne Geschwindigkeit durch free-slip Randbedingung
        if(wL.a == 1){pL = pC; obstV.x = wL.x; vMask.x = 0;}
        if(wR.a == 1){pR = pC; obstV.x = wR.x; vMask.x = 0;}
        if(wB.a == 1){pB = pC; obstV.y = wB.y; vMask.y = 0;}
        if(wT.a == 1){pT = pC; obstV.y = wT.y; vMask.y = 0;}
        if(wU.a == 1){pU = pC; obstV.z = wU.z; vMask.z = 0;}
        if(wD.a == 1){pD = pC; obstV.z = wD.z; vMask.z = 0;}

        // Gradient
        vec3 gradP =  vec3(pR - pL, pT - pB, pU - pD);

        // Projektion
        vec4 u = vec4(0.0,0.0,0.0,0.0);
        u.xyz = wC.xyz - gradP;

        // Geschwindigkeitskorrektur durch free-slip Randbedingung
        u.xyz = (vMask * u.xyz) + obstV;

        return u + vec4(0.5,0.5,0.5,0.0);
}

void main()
{
        gl_FragColor = grad();
}
```

In das Fluidvolumen können Hindernisse in Form von Objekten platziert werden. Die Bedingungen an den Rändern eines solchen Objekts werden gesondert behandelt. Für das Druckfeld verwenden wir die Neumann-Randbedingung. Der Gradient des Druckfeldes entlang der Oberflächennormalen des Hindernisobjekts sollte

dabei den Wert Null annehmen. Beim Geschwindigkeitsfeld verwenden wir eine free-slip Randbedingung, damit die Geschwindigkeit des Objekts auf das Geschwindigkeitsfeld übertragen werden kann [Dep08].

7 Ergebnisse

In der Studienarbeit konnte gezeigt werden, dass die physikalisch basierte Simulation von dreidimensionalen Fluiden in Echtzeit auf aktuellen GPU's möglich ist. Die Fluidsimulation wurde erfolgreich in eine Echtzeitanwendung integriert und kann mit Objekten interagieren. Die Echtzeitanwendung ist ein kleiner OpenGL-Renderer, mit dessen Hilfe man Terrain, Objekte, Wasser usw. rendern kann. Die Anwendung erlaubt die Manipulation der Parameter der Fluidsimulation und das Speichern der aktuellen Konfiguration. Der Nutzer kann sich mit den vielfältigen Möglichkeiten und faszinierenden Effekten einer Fluidsimulation vertraut machen. Je nach GPU kann die Rasterauflösung des Fluids angepasst werden. Auf dem Testsystem mit einem Grafikbeschleuniger von Nvidia (GeForce GTX260) erreichte die Simulation bei einer Rasterauflösung von $100 \times 100 \times 100$ Pixeln durchschnittlich 31 fps.

Die folgenden Abbildungen sind Screenshots aus der eigens erstellten Echtzeitanwendung in Form eines OpenGL-Renderers, in die die Fluidsimulation integriert wurde. Sie demonstrieren einige Effekte wie Explosionen (Abb. 20), Flammen (Abb. 21), Nebel (Abb. 25, Abb. 24, Abb. 23), Rauchschwaden (Abb. 22, Abb. 19, Abb. 26, Abb. 27) oder das Verhalten des Fluids an Hindernisobjekten (Abb. 28, Abb. 29) die mit einer Fluidsimulation visualisiert werden können.

8 Ausblick

Die vorliegende Fluidsimulation bietet vielfältige Erweiterungsmöglichkeiten.

8.1 Visualisierung

Für eine verbesserte Volumenvisualisierung könnte beispielsweise das Raycasting-Verfahren aus [CLT07] implementiert werden. Eine größere Herausforderung wäre auch die Suche nach einem geeigneten Beleuchtungsverfahren. Das Kapitel „Cloud Illumination and Rendering“ aus der Dissertation von [Har03] liefert dazu ausführlichere Informationen. Beim View-aligned Volume Slicing sind an den Stellen, an denen die Slices die Geometrie der Szene schneiden, Artefakte, wie z.B. unnatürlich scharfe Kanten, zu sehen. Zur Vermeidung dieser Artefakte könnte man die „Soft Particles“-Methode aus [Lor] verwenden.

Abbildung 19: Visualisierung eines Vulkanausbruchs

8.2 Simulation

Neben der Simulation von Rauch- und Gaseffekten kann die Fluidsimulation auch als Basis für die Realisierung von Feuer oder Wassereffekten verwendet werden [CLT07]. Im Beschleunigungsschritt kann man zudem eine Temperatursimulation [Har03] hinzufügen. Nach der Beschleunigung und vor der Projektion kann für die Simulation von Dynamischen Wolken ein Thermodynamik-Schritt wie in [Har03] beschrieben eingefügt werden. Für deformierbare Objekte ließe sich ein verbessertes Voxelisierungsverfahren [CLT07] unter Verwendung des Geometry Shaders realisieren. Der Renderer der Applikation bietet zusätzlich zahlreiche Experimentiermöglichkeiten für den Einsatz von Fluiden in Echtzeitanwendungen.

8.3 Performanz

Im Projektionsschritt lassen sich durch eine Vektorisierung der skalaren Druckwerte kleinere Texturen verwenden. Dazu werden immer vier skalare Werte in einen RGBA-Vektor gepackt. So erhält man weniger Fragmente, die der Shader wärend der Jacobi Iteration berechnen muss. Mehr dazu in [Har03].

Abbildung 20: Visualisierung einer Explosion

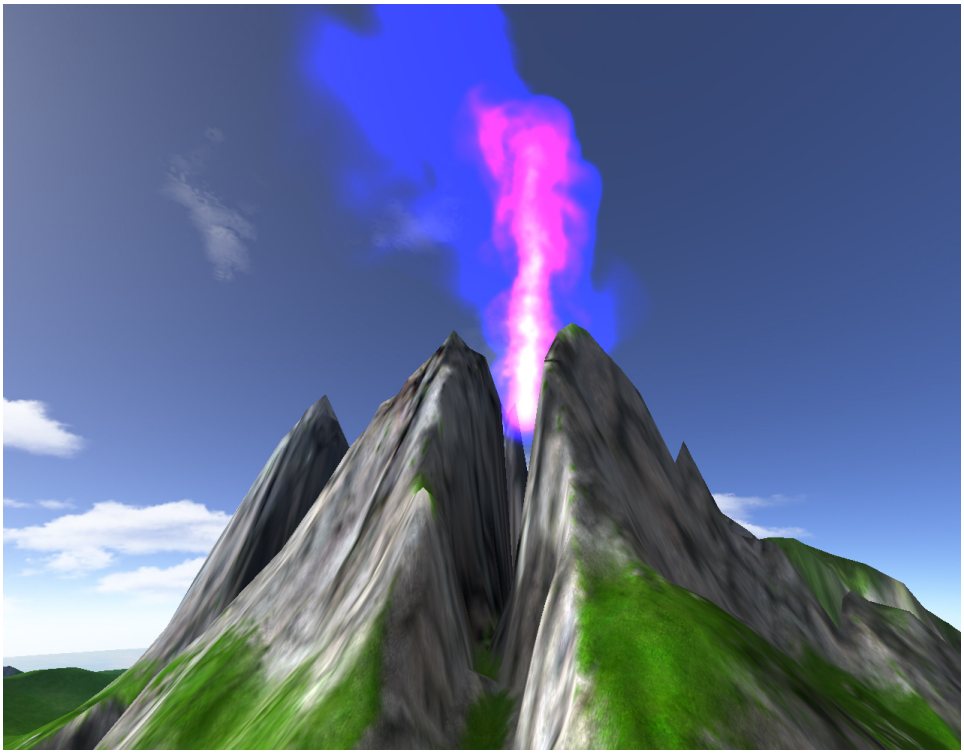

Abbildung 21: Visualisierung einer bunten Flamme

Abbildung 22: Visualisierung von Rauchschwaden

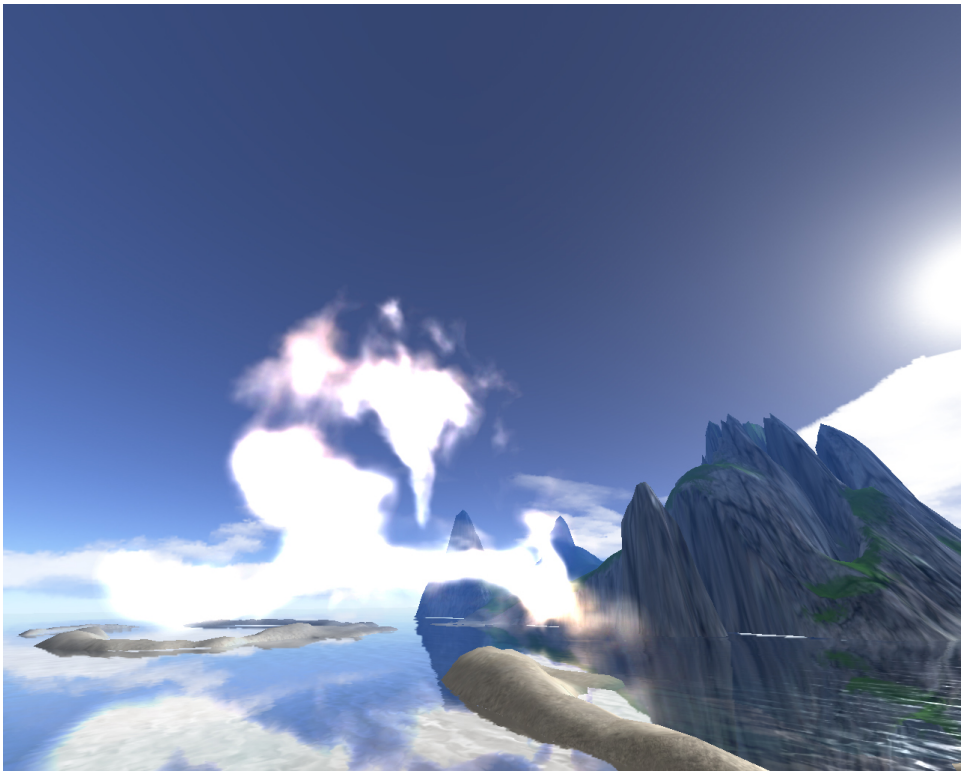

Abbildung 23: Verändert man den Blendmode, erscheinen die Rauchschwaden wie Dampf

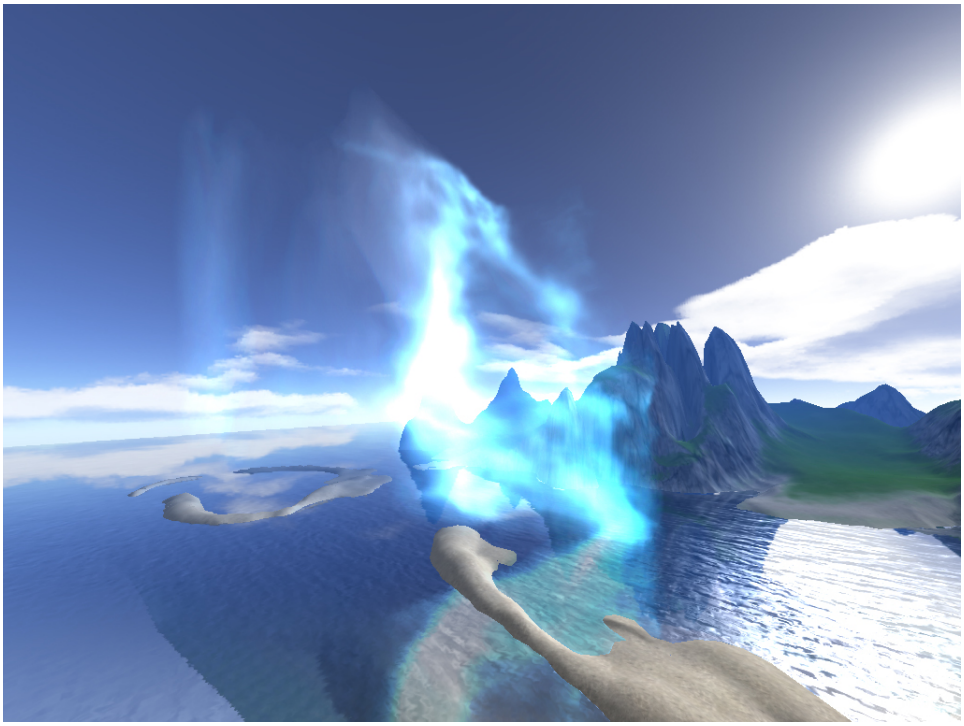

Abbildung 24: Visualisierung bläulich schimmernder Nebelschwaden

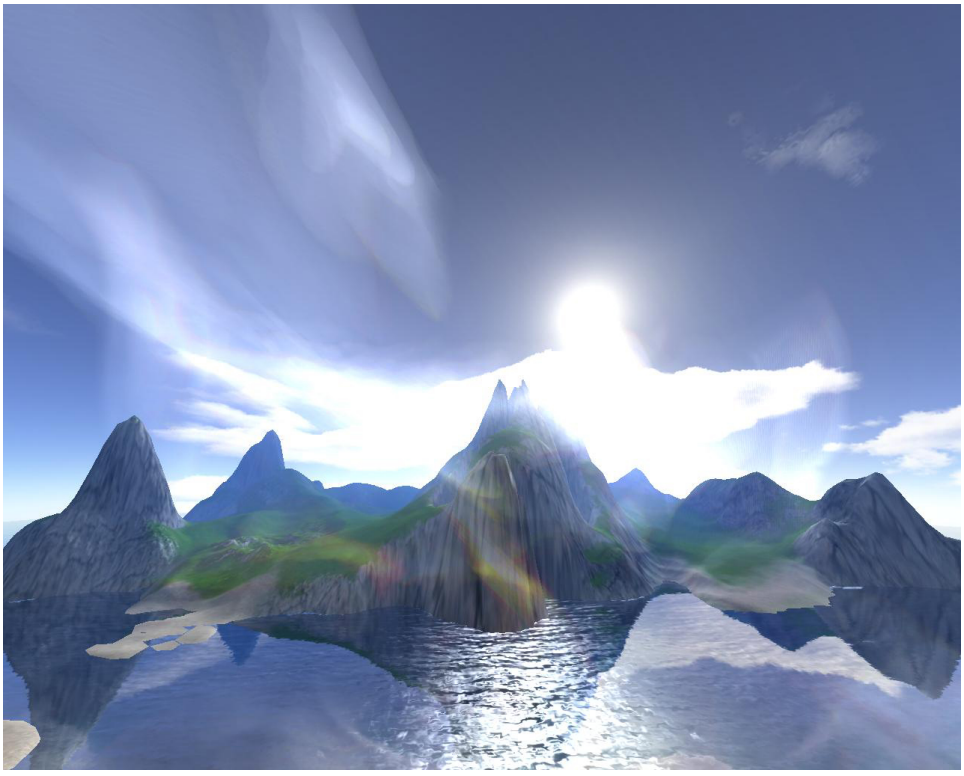

Abbildung 25: Farbige Nebelschwaden

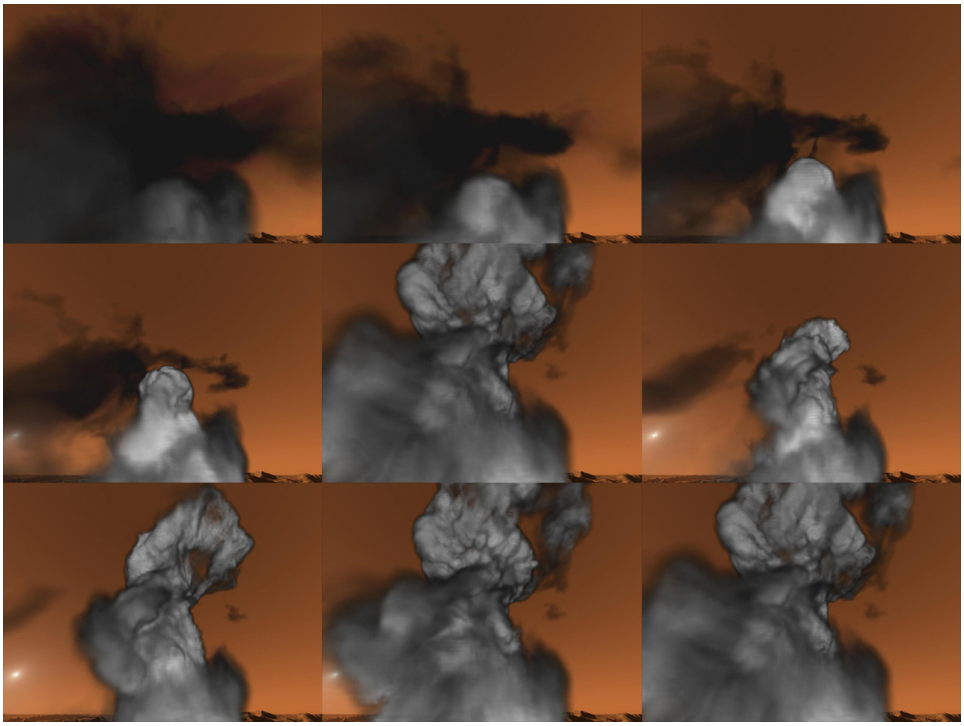

Abbildung 26: Die Entwicklung eines Rauchpilzes in einzelnen Schritten in einem $150 \times 150 \times 150$ großen Grid.

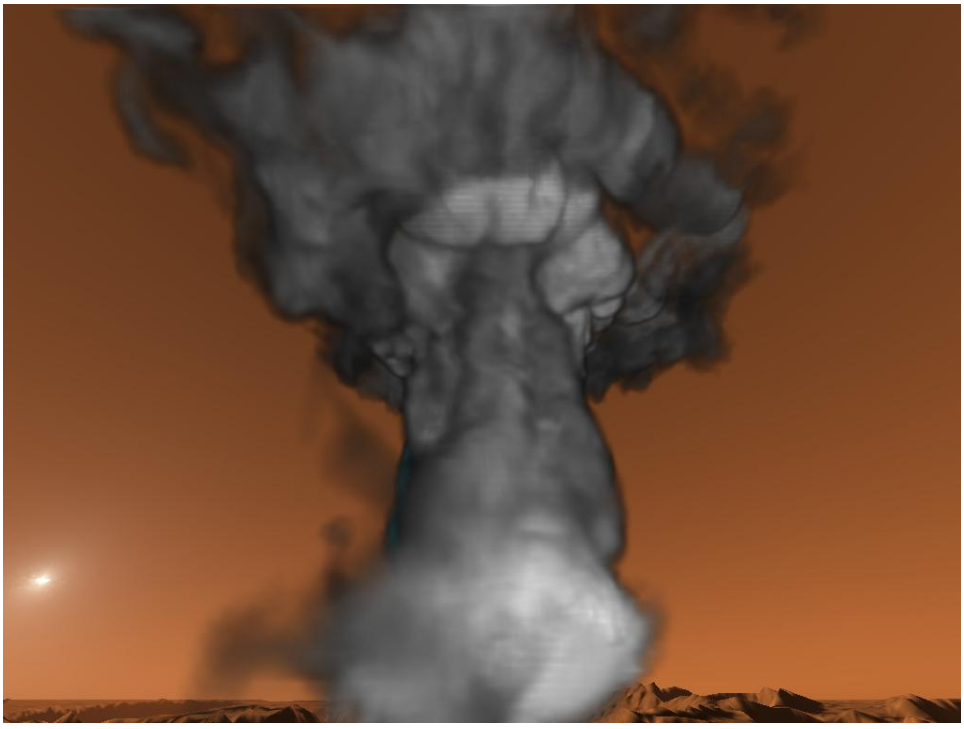

Abbildung 27: Visualisierung eines Rauchpilzes

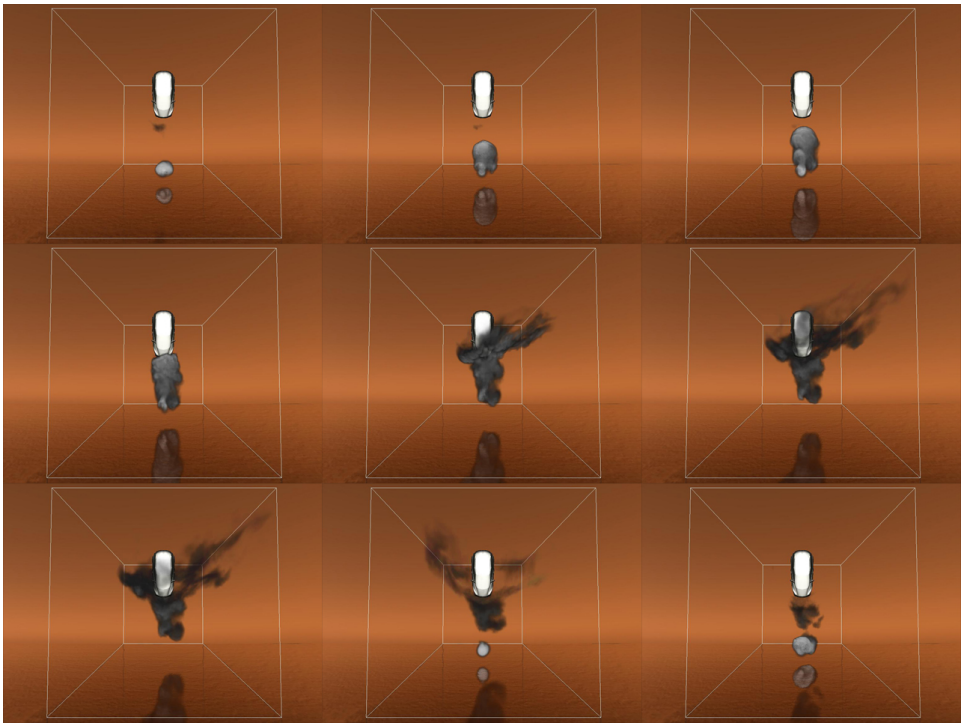

Abbildung 28: Hinderniserkennung in einzelnen Zeitschritten

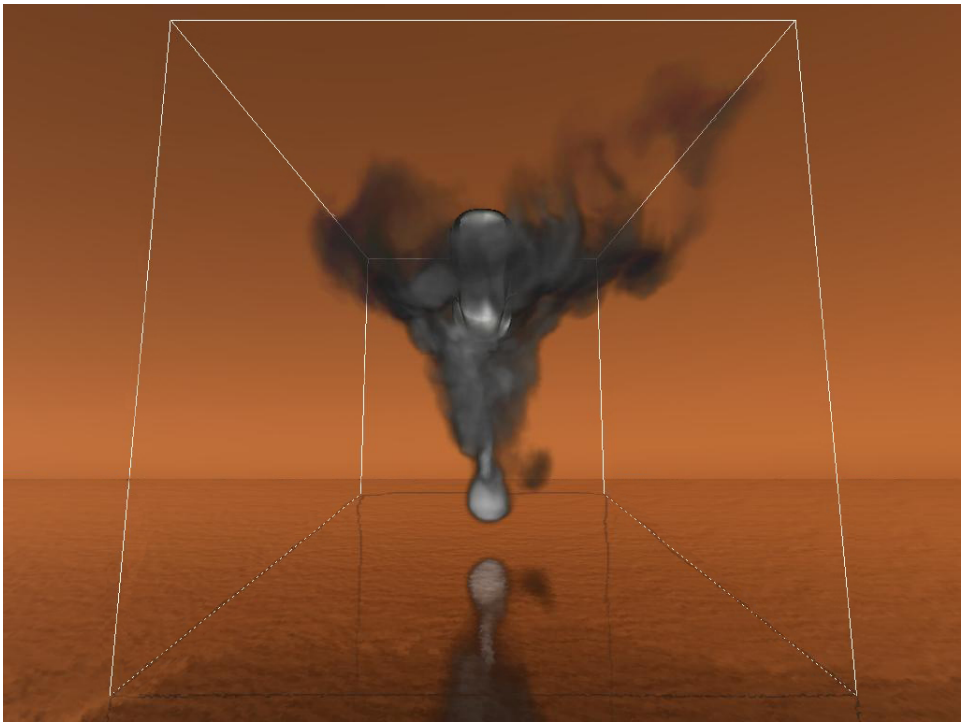

Abbildung 29: Hindernisobjekt im Fluid

Abbildungsverzeichnis

Listings

Literatur

[CLT07] CRANE, Keenan ; LLAMAS, Ignacio ; TARIQ, Sarah: Real-Time Simulation and Rendering of 3DFluids. In: NGUYEN, Hubert (Hrsg.): *GPU Gems 3*. Addison Wesley Professional, August 2007, Kapitel 30

[Dep08] DEPARTMENT OF CONTROL ENGINEERING AND INFORMATION TECHNOLOGY: Gridfluid - 3D Grid Based Fluid Simulator for Hewlett-Packard Scalable Visualization Array / Budapest University of Technology and Economics. 2008. – Forschungsbericht

[FSWJ01] FEDKIW, Ronald ; STAM, Jos ; WANN JENSEN, Henrik: Visual Simulation of Smoke. In: *Proceedings of ACM SIGGRAPH 2001*, 2001 (Computer Graphics Proceedings, Annual Conference Series), S. 15–22

[GSC08] GSC GAME WORLD: *S.T.A.L.K.E.R.: Clear Sky*. `http://stalker.deepsilver.com`. Version: 2008. – S.T.A.L.K.E.R.: Clear Sky HD DirectX10 Feature Demo Video

[Har03] HARRIS, Mark J.: *Real-Time cloud simulation and rendering*, University of North Carolina at Chapel Hill, Diss., 2003

[Har05] HARRIS, Mark J.: Fast fluid dynamics simulation on the GPU. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM Press, 2005

[Hei07] HEINECKE, Berthold: *Physikalische Rauchsimulation auf Partikelbasis in Echtzeit mit der PhysX-Engine*. September 2007. – Belegarbeit an der Technischen Universität Dresden

[Lor] LORACH, Tristan: *Soft Particles*. – NVIDIA DirectX 10 SDK

[MCG03] MÜLLER, Matthias ; CHARYPAR, David ; GROSS, Markus: Particle-based fluid simulation for interactive applications. In: *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2003. – ISBN 1–58113–659–5, S. 154–159

[Sta99] STAM, Jos: Stable Fluids. In: ROCKWOOD, Alyn (Hrsg.): *Siggraph 1999, Computer Graphics Proceedings*. Los Angeles : Addison Wesley Longman, 1999, S. 121–128

[Ste08] STELZMANN, Robert: *PPartikelsysteme und Fluidsimulationen*. August 2008. – Proseminar Computergrafik an der Technischen Universität Dresden

[TL07] TARIQ, Sarah ; LLAMAS, Ignacio: *Real-Time Volumetric Smoke using D3D10*. Proceedings of the Game Developer Conference 2007, März 2007. – NVIDIA Developer Technology

[Wik09a] WIKIPEDIA: *Eulersches Polygonzugverfahren— Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?tit\le=Eulersches_Polygonzugverfahren\&oldid=58725467. Version: 2009. – [Online; Stand 6. April 2009]

[Wik09b] WIKIPEDIA: *Navier-Stokes-Gleichungen— Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=\Navier-Stokes-Gleichungen\&oldid=58713318. Version: 2009. – [Online; Stand 5. April 2009]

[Wik09c] WIKIPEDIA: *Smoothed Particle Hydrodynamics— Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Smooth\ed_Particle_Hydrodynamics\&oldid=56812835. Version: 2009. – [Online; Stand 6. April 2009]

[Wil] WILLKOMM, Dennis: *Visuelle Effekte mit volumetrischen Shadern*. – Studienarbeit an der Universität Koblenz-Landau