UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Simulation der Entstehung von Sternen

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Hendrik Schwanekamp

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Bastian Krayer, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 2018

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.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Zusammenfassung

In dieser Bachelorarbeit wird ein Simulationscode für astrophysikalische Simulationen von Fluiden unter dem Einfluss ihrer eigenen Gravitation entwickelt. Der Code wird hauptsächlich von der *GPU* ausgeführt. Leichte Vereinfachungen der physikalischen Modelle und einige Parameter zum Steuern von Genauigkeit und Rechenaufwand ermöglichen das Simulieren mit interaktiver Bildwiederholrate auf den meisten handelsüblichen, modernen Computern mit einer dedizierten Grafikkarte. Der Simulationscode wird verwendet, um die Entstehung von Sternen aus einer Gaswolke zu simulieren. Einige Merkmale der Sternentstehung, wie zum Beispiel Akkretionsscheiben und Fragmentierung, lassen sich selbst bei niedrigen Partikelzahlen beobachten.

Abstract

In this bachelor thesis a code for astrophysical self-gravitating fluid simulation is developed. The code runs mainly on the *GPU*. Minimal simplifications of the physical model and some parameters for accuracy and tuning allow simulations to be performed at interactive framerates on most modern consumer grade computers that feature a dedicated graphics card. It is used to simulate the birth of stars from a turbulent molecular cloud. Multiple features of star formation, like accretion discs and fragmentation, can be observed in the simulation, even when low particle counts are used.

Vielen Dank...

... an Johannes Braun, der Teile seines OpenGL-Frameworks für die Arbeit zur Verfügung gestellt und mich genau wie Max Nilles bei Programmierfragen unterstützt hat.

... an Prof. Dr. Hubert Klahr und Dr. Christoph M. Schäfer für das interessante und sehr motivierende Treffen am Max-Planck-Institut für Astronomie in Heidelberg.

... an meine Familie, Freunde und meine Freundin, die mich unterstützen und dieses Studium ermöglichen.

... an Jan-Moritz Hoffmann, Thuy Linh Nguyen, Dr. Christoph M. Schäfer, Alexander Scheid-rehder, Birgit Schwanekamp, Katja Schwanekamp und Klaus Zimmer, für das Korrekturlesen dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Entstehung von Sternen 3
2.2 Gravitation 6
2.3 Fluiddynamik 7
2.4 Smoothed Particle Hydrodynamics 10
2.5 Anwendung von SPH auf die Gasdynamik 11
2.6 Interpolationskernel 12
2.7 Viskosität 14
2.8 Variabler Kernelradius 15

3 Konzeption **19**
3.1 Ziele der Simulation 19
3.2 Modellannahmen 19
3.3 Verwendete Techniken 20
3.3.1 Diskretisieren der Gaswolke 20
3.3.2 Anfangs- und Randbedingungen 22
3.3.3 Zeitintegration 23
3.4 Zusammenfassung des Simulationsalgorithmus 24
3.5 Strategien zur Beschleunigung 27
3.5.1 GPU Spezifische Techniken 27
3.5.2 Datenstrukturen 29

4 Implementierung **31**
4.1 Aufbau der Software 31
4.2 Handhabung von Partikeln 31
4.3 Generierung von Partikeln 33
4.4 Simulation des Partikelsystems 33
4.4.1 Einstellung des Kernelradius 34
4.4.2 Berechnung der Dichte 34
4.4.3 Berechnung des Drucks 37
4.4.4 Berechnung der Beschleunigung 37
4.4.5 Integration 38
4.5 Visualisierung des Ergebnisses 39
4.6 Benutzereingabe 41

5 Evaluation **42**
5.1 Performance 42
5.1.1 Parametertuning 44
5.2 Ergebnisse 45
5.2.1 Erkennbare Merkmale der Sternentstehung 48

5.2.2 Geringere Partikelzahlen 49
5.2.3 Vergleich mit anderen Simulationen 50
5.2.4 Mögliche Ursachen für Probleme 52

6 Fazit **54**

7 Ausblick **55**

1 Einleitung

Diese Arbeit beschäftigt sich mit der Entwicklung eines partikelbasierten Simulationscodes für astrophysikalische Fluidsimulationen, mit dem sich die Entstehung von Sternen simulieren lässt. Die Nutzung der *graphics processing unit* (*GPU*) ermöglicht das Durchführen der Simulation mit interaktiver Bildwiederholrate.

Der Lebenszyklus der Sterne – von ihrer Entstehung bis zu ihrem Tod – ist ein aktives Forschungsgebiet. Im Inneren von Sternen entstehen einige der uns bekannten Elemente und in ihrer Umgebung bilden sich häufig Planeten. Unsere Sonne ist unter anderem für die Bedingungen verantwortlich, die Leben auf der Erde ermöglichen. Je mehr über die Sterne und ihre Entstehung bekannt ist, um so besser kann das Universum und dessen Entwicklung verstanden werden. Der Vorgang der Sternentstehung ist allerdings sehr schwierig zu beobachten. Viele Gebiete, in denen Sterne entstehen, sind weit entfernt und es können Millionen von Jahren vergehen, bis die Entstehung eines neuen Sterns abgeschlossen ist. Menschen sehen nur einzelne Momentaufnahmen.

Schon seit einiger Zeit werden in der Physik Computersimulationen eingesetzt. Sie ermöglichen es, Phänomene besser zu verstehen, die nur schwer beobachtet werden können. Durch aufwendige Analysen kann die Genauigkeit der verwendeten Simulationsmethoden eingeschätzt werden. Die Ergebnisse werden, wenn möglich, zusätzlich mit Beobachtungen abgeglichen. Computersimulationen der Sternentstehung bieten einen Überblick über den gesamten Ablauf.

Solche Simulationen sind meist sehr aufwendig. Sie laufen längere Zeit auf Supercomputern oder dedizierter Hardware, bevor die Ergebnisse analysiert werden können. Mittlerweile lassen sich handelsübliche Grafikkarten sehr frei programmieren und halten leistungstechnisch mit zehn bis zwanzig Jahre alten Supercomputern mit. Oft werden Grafikkarten bereits genutzt, um bestimmte Teile von Physiksimulationen zu beschleunigen.

Diese Arbeit kombiniert Techniken der Echtzeit-Computergrafik mit denen der numerischen Astrophysik. So entsteht ein Simulationscode, der hauptsächlich auf einer handelsüblichen *GPU* ausgeführt wird und gut an die Leistungsstärke verschiedener Hardware angepasst werden kann. Mit nur wenigen Vereinfachungen in den physikalischen Modellen wird die Entstehung von Sternen bei interaktiver Bildwiederholrate plausibel abgebildet. Interaktiv heißt hier, dass die Visualisierung direkt beim Simulieren flüssig dargestellt wird. Währenddessen kann der Benutzer die virtuelle Kamera frei im Raum bewegen und so die entstehenden Strukturen aus verschiedenen Blickwinkeln betrachten.

In Kapitel 2 werden die Grundlagen behandelt, die zur Entwicklung einer solchen Simulation verstanden werden müssen. Angefangen mit der Theorie der Sternentstehung, über die physikalischen Gesetze der Gravitationskraft und Fluiddynamik bis zu den Vorgehensweisen bei der Simulationstechnik *Smoothed Particle Hydrodynamics* (*SPH*).

Anschließend werden in Kapitel 3 die Ziele der Simulation definiert und ein entsprechendes Modell hergeleitet. Die verwendeten Techniken werden besprochen und Möglichkeiten zum Beschleunigen der Berechnungen diskutiert.

Das vierte Kapitel erläutert, wie genau die zuvor beschriebenen Konzepte auf der *GPU* implementiert wurden. Darauf folgt mit Kapitel 5 eine Bewertung von Performance und Ergebnissen der Simulation. Fazit (6) und Ausblick (7) bilden den Abschluss der Arbeit.

2 Grundlagen

2.1 Entstehung von Sternen

Am Nachthimmel lassen sich eine Vielzahl von Sternen und manchmal sogar Galaxien erkennen. Der Raum zwischen den Sternen wird als *Interstellares Medium* oder *ISM* bezeichnet und erscheint mit bloßem Auge schwarz und leer. Tatsächlich besteht das *ISM* aus 70% Wasserstoff, 28% Helium und 2% anderer Elementen wie Sauerstoff, Kohlenstoff und Stickstoff. Ein Großteil der Materie – etwa 99% – ist gasförmig mit einer sehr geringen durchschnittlichen Dichte von 10^6 Atomen pro Kubikmeter.

Das *ISM* ist nicht gleichmäßig. Es kann in Regionen verschiedener Dichte und Temperatur eingeteilt werden. Besonders interessant für die Sternentstehung sind große Wolken aus kaltem Gas mit hoher Dichte. Diese erscheinen dunkel, da sie neben dem Gas auch noch Staub enthalten. Obwohl die Staubpartikel sehr klein sind und nur etwa 1% der Gesamtmasse des ISM ausmachen, absorbieren sie den Großteil des Lichts in sichtbaren Wellenlängen. Mit Teleskopen, die in anderen Wellenlängenbereichen arbeiten, können die Wolken trotzdem beobachtet werden. Das Innere solcher Wolken besteht hauptsächlich aus H_2 Molekülen, weil der Staub das Eindringen von UV-Strahlung blockiert. Diese würde die Moleküle sonst zerstören. Die Dichte beträgt mehr als 10^2 Moleküle pro Kubikmeter an den dichtesten Stellen. Solche Wolken haben eine Größe von 0,1pc bis zu 100pc (ein *Parsec* (pc) entspricht 3×10^{16} Metern) und sind die größten bekannten Strukturen innerhalb von Galaxien. Sie enthalten Massen von bis zu mehreren Millionen Sonnen (eine Sonnenmasse entspricht 2×10^{30}kg). Daher werden sie auch als *GMC* – *giant molecular cloud* – bezeichnet.

GMCs kommen häufig in den Scheiben von Spiralgalaxien wie unserer Milchstraße, zum Beispiel in den Spiralarmen, vor (siehe Abbildung 2). Sterne können auch an anderen Orten entstehen, zum Beispiel wenn Galaxien kollidieren.

Oft weisen *GMCs* turbulente Strömungen auf. Diese entstehen unter anderem aus der Einwirkung von in der Nähe entstehenden oder sterbenden Sterne (z.B. Druckwellen einer Supernova-Explosion). Die Bewegung des Gases führt zur Bildung von Regionen unterschiedlicher Dichte. Besonders dichte Regionen werden *dense cores* genannt. Diese *cores* entstehen oft in Gruppen und weisen eine Drehgeschwindigkeit und interne Strömungen auf, die aus der Turbulenz der *GMC* entstehen. Hier wird die Eigengravitation des Gases relevant, einzelne Gasteilchen ziehen sich gegenseitig an.

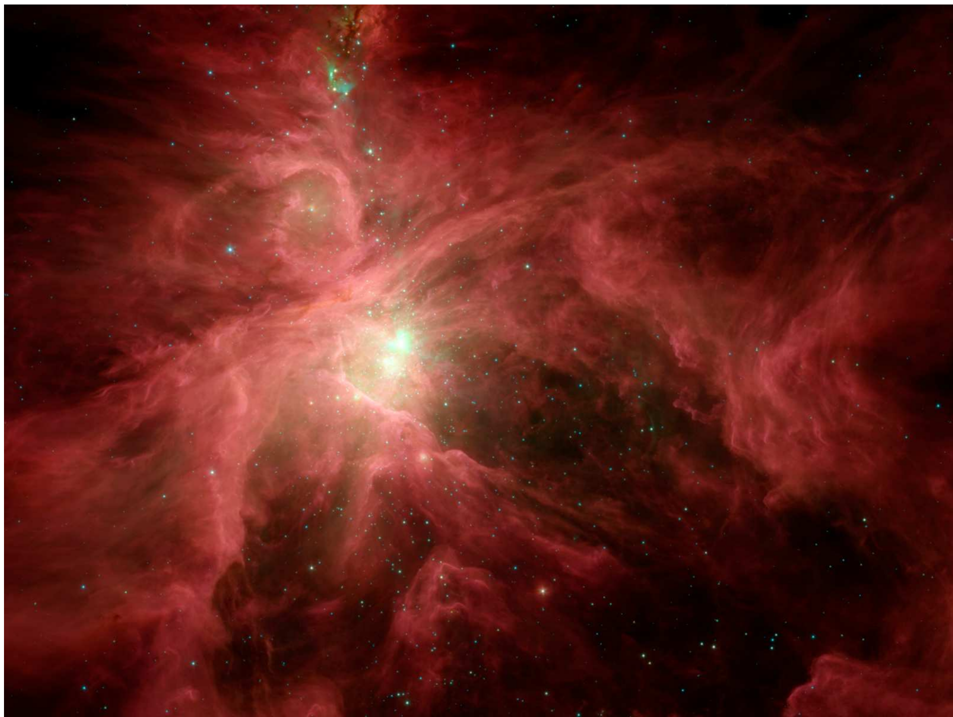


Abbildung 1: Ausschnitt aus einem Infrarotbild des *Orionnebel*, aufgenommen vom *Spitzer Space Telescope*. Der Nebel entstand in einer kalten Gaswolke umgeben von Staub. Er beinhaltet mehr als 1000 junge Sterne und ist mit einer Entfernung von etwa 1450 Lichtjahren das der Erde am nächsten liegende große Sternentstehungsgebiet. `http://www.spitzer.caltech.edu/images/1643-ssc2006-16a-Orion-s-Inner-Beauty`

Wenn Gas solcher Teilregionen unter seiner eigenen Gravitation zusammenfällt, wird es zu einem Stern komprimiert. Die Stärke der Gravitationskraft ist von der Masse abhängig. Dagegen wirkt der temperaturabhängige Druck des Gases, der die Teilchen auseinander drückt. Um ein Stern zu werden, muss der *core* also möglichst kalt sein und viel Masse enthalten. Sind Druck und Gravitation gleich stark, befindet sich die Wolke im Gleichgewicht (*thermodynamic equilibrium*) und fällt nicht zusammen.

Während das Gas zusammenfällt erhöht sich die Dichte und damit auch die Temperatur. Anfangs geschieht das Zusammenfallen langsam und die Wärme kann effektiv durch Strahlung abtransportiert werden. Dadurch bleibt die Temperatur konstant. Das Gas ist *isotherm*. Es entsteht ein druckgestütztes Objekt, ein *Protostern*, im Zentrum einer Hülle aus Gas. Mit steigender Dichte der Hülle kann weniger Energie abtransportiert werden. Das Gas beginnt sich zu erwärmen. Beim Zusammenfallen der Wolke zu *Protosternen* werden Teilregionen instabil, die dann wiederum in sich selbst

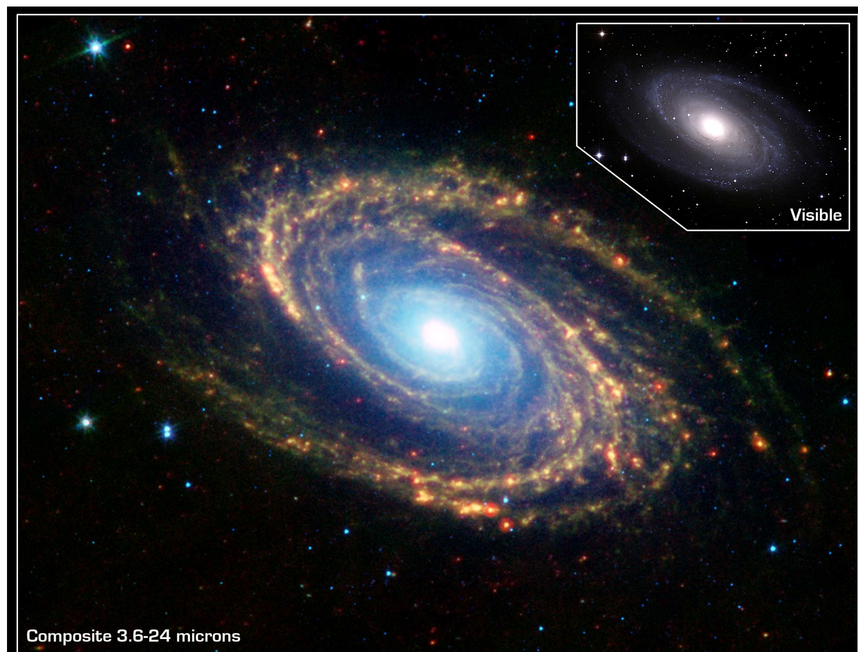


Abbildung 2: Die Spiralgalaxie Messier 18, aufgenommen mit dem *Spitzer Space Telescope*. Im Infrarotbild lassen sich die Regionen in den Spiralarmen erkennen, in denen Sterne entstehen. Im sichtbaren Spektrum (oben rechts) sind sie weniger klar abgebildet.
`http://www.spitzer.caltech.edu/images/1076-ssc2003-06d-Spiral-Galaxy-Messier-81`

zusammenfallen. Dies wiederholt sich rekursiv, solange das Gas *isotherm* bleibt. Der Vorgang wird als *Fragmentierung* bezeichnet (siehe Abbildung 3).

Der *Protostern* nimmt weiter Gas aus der Hülle auf. Dieses fällt allerdings nicht gradlinig nach innen. Durch die anfängliche Drehgeschwindigkeit des *cores* formt es eine rotierende Scheibe um das Objekt im Zentrum. Das Gas fällt dann in einer spiralförmigen Bewegung auf den Äquator des jungen Sterns. In so einer *Akkretionsscheibe* können auch Planeten oder weitere Sterne entstehen. Während dieser Phase nimmt der Stern den Großteil seiner finalen Masse auf. Gleichzeitig sorgt eine Reihe von Mechanismen – die noch nicht vollständig erforscht sind – dafür, dass Drehgeschwindigkeit abgebaut wird. Ansonsten würden *Protosterne* so schnell rotieren, dass ein Zusammenfallen unmöglich wird.

Erreicht der Protostern eine Masse von etwa $0,075$ Sonnenmassen ($1,5 \times 10^{29}$kg), ist sein innerer Kern heiß genug, um Wasserstoff durch Kernfusion in Helium umzuwandeln. Schafft er es nicht, diese Grenze zu erreichen, wird er als *brauner Zwerg* bezeichnet. Der entstandene Stern gibt Wärme an das Gas in seiner Umgebung ab. Die Reste der *GMC* dehnen sich durch die erhöhte Temperatur aus und verschmelzen wieder mit dem Teil des *ISM*, der eine geringere Dichte hat.

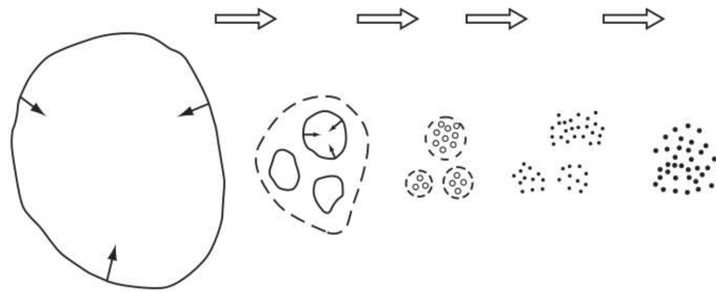


Abbildung 3: Rekursive Fragmentierung einer Gaswolke. [WTW11]

So entstehen Sterne verschiedenster Masse, die sich anhand der Helligkeit und Wellenlänge des abgestrahlten Lichts in Klassen einteilen lassen. Im Laufe seines Lebens fällt ein Stern je nach Masse in mehrere dieser Klassen. Die meisten Sterne durchlaufen die sogenannte *Hauptreihe*, bis schließlich der Großteil des Wasserstoffs in ihrem Kern verbraucht ist. Ist die Masse des Sterns groß genug, explodiert er in einer Supernova, aber auch Sterne mit geringerer Masse beginnen Materie zu verlieren. Die freigesetzten Stoffe werden Teil des *ISM* und der nächsten Generation von Sternen und Planeten. Man spricht auch vom *Lebenszyklus* der Sterne. [WTW11] [MO07]

2.2 Gravitation

Gravitation bezeichnet die Kraft, mit der sich Masse gegenseitig anzieht. Auf kurze Distanzen ist sie wesentlich schwächer als die anderen fundamentalen Kräfte der Physik. Werden allerdings größere Ansammlungen von Materie betrachtet, heben sich anziehende und abstoßende Kräfte gegenseitig auf, da die Anzahl der Positiven und Negativen Ladungen meist gleich ist. Gravitation hingegen wirkt immer anziehend. Das macht Gravitation zu einer der wichtigsten Kräfte bei der Entstehung von Strukturen im Weltall. Annähernd runde Objekte können für die Berechnung der Gravitationskraft wie eine Punktmasse behandelt werden [MH07].

Die Gravitationskraft, welche auf einen Partikel i als Resultat der Interaktion mit einem anderen Partikel j wirkt ist gegeben durch:

$$\boldsymbol{f_{ij}} = G \frac{m_i m_j}{||\boldsymbol{r_{ji}}||^2} \frac{\boldsymbol{r_{ji}}}{||\boldsymbol{r_{ji}}||} \tag{1}$$

Dabei sind m_i und m_j die Massen der beiden Partikel. $\boldsymbol{r_{ji}} = \boldsymbol{r_j} - \boldsymbol{r_i}$ ist der Vektor von Partikel i zu Partikel j, wobei $\boldsymbol{r_i}$ und $\boldsymbol{r_j}$ ihre Positionen bezeichnen. G ist die Gravitationskonstante. In einem System mit N Partikeln ergibt sich die Gesamtkraft, die auf einen Partikel wirkt, aus der Summe seiner

Interaktionen mit allen andern $N-1$ Partikeln:

$$\boldsymbol{F_i} = \sum_{j=1;i\neq j}^{N} \boldsymbol{f_{ij}} = Gm_i \sum_{j=1;i\neq j}^{N} \frac{m_j \boldsymbol{r_{ji}}}{||\boldsymbol{r_{ji}}||^3} \tag{2}$$

Diese Formulierung birgt allerdings einige Herausforderungen für die numerische Simulation. Der Betrag der Gravitationskraft ist antiproportional zum Quadrat der Entfernung. Bei sich annähernden Partikeln wächst die Gravitationskraft unbeschränkt, was zur Instabilität der Simulation führen kann. Um die Bewegung von Partikeln, die sich besonders nah kommen, korrekt zu berechnen, werden sehr kleine Zeitschritte benötigt. Außerdem beträgt der Aufwand einer naiven Implementation von Gleichung 2 $O(N^2)$, da für jeden der N Partikel die Interaktion mit allen anderen Partikeln betrachtet werden muss. Mögliche Lösungen für das letztere Problem werden in Kapitel 3.5.1 und 3.5.2 behandelt.

Um das unbeschränkte Wachsen der Gravitationskraft zu verhindern, wird in der Praxis oft ein Dämpfungsfaktor $\epsilon > 0$ eingeführt und Gleichung 2 wie folgt umgeschrieben:

$$\boldsymbol{F_i} \approx Gm_i \sum_{j=1}^{N} \frac{m_j \boldsymbol{r_{ji}}}{(||\boldsymbol{r_{ji}}||^2 + \epsilon^2)^{3/2}} \tag{3}$$

Hier fällt auch die Bedingung $i \neq j$ weg, da bei gleicher Position der Partikel nicht mehr durch 0 geteilt wird. Diese Art der Dämpfung wird *plummer softening* genannt und basiert auf einem Modell für die Interaktion von kreisförmigen Galaxien [NHP07].

In der Literatur sind noch weitere Dämpfungsverfahren beschrieben, die genauere Ergebnisse liefern. Sie haben dafür einen größeren Berechnungsaufwand (z.B. [DI93] [SYW01]).

Um die Bewegung der Partikel über die Zeit zu verfolgen, wird die aus der Gravitationskraft resultierende Beschleunigung $\boldsymbol{a_i}$ benötigt. Einsetzen von Gleichung 3 in das zweite newtonsche Gesetz $\boldsymbol{a} = \boldsymbol{F}/m$ liefert [NHP07]:

$$\boldsymbol{a}_i^{grav} \approx G \sum_{j=1}^{N} \frac{m_j \boldsymbol{r_{ji}}}{(||\boldsymbol{r_{ji}}||^2 + \epsilon^2)^{3/2}} \tag{4}$$

2.3 Fluiddynamik

Fluid ist der Überbegriff für eine Kategorie von Stoffen, die Flüssigkeiten und Gase umfasst. In der Physik wird eine Substanz als Fluid bezeichnet, die sich unter Einfluss einer Scherungskraft kontinuierlich verformt (Abbildung 4) [Hau08, Seite 7].

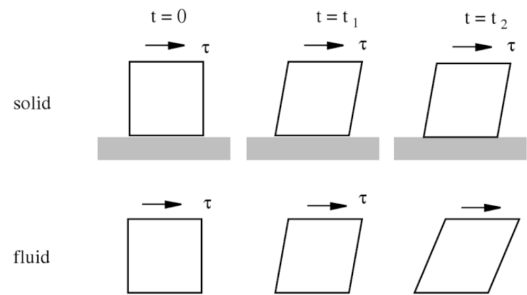


Abbildung 4: Ein Fluid verformt sich unter Einfluss einer Scherungskraft immer weiter. Der Festkörper folgt der Scherung nur bis zu einem bestimmten Punkt [Hau08].

Um den Zustand und das Verhalten eines Fluids zu beschreiben, muss sein gesamtes Volumen betrachtet werden. Für eine numerische Betrachtung wird dieses Volumen diskretisiert. Das bedeutet, es wird ein Modell erstellt, welches die Eigenschaften des Fluids in endlich vielen Werten annähert. Zwei Möglichkeiten haben sich durchgesetzt: der *Lagrangian Approach*, bei dem sich die Punkte, an denen die Eigenschaften gespeichert werden, bewegen können (Partikelsystem) und den *Eulerian Approach*, bei dem die Position der Punkte gleich bleibt und nur die anderen Eigenschaften geändert werden (Unterteilung durch ein Gitter). Abbildung 5 skizziert die beiden Vorgehensweisen.

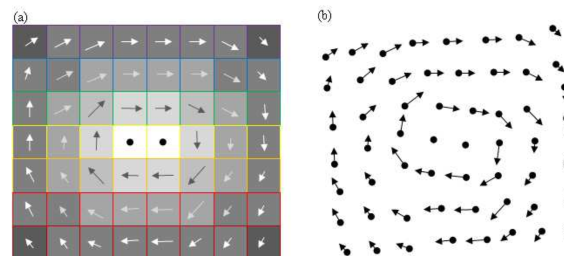


Abbildung 5: Modell eines Fluids nach Euler (a) und Lagrange (b) `software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1--~Mai~2017`

Im Folgenden wird nur auf die Vorgehensweise mit Partikelsystemen eingegangen, da sie einige Vorteile für die Simulation von zusammenfallenden Gaswolken bietet und auf der GPU effizient implementiert werden kann (siehe Kapitel 3.3.1).

Das Verhalten von Gaspartikeln ohne Viskosität ist durch die *Euler Gleichungen* gegeben. Sie beschreiben die Veränderung von Geschwindigkeit $\boldsymbol{v}$ und Position $\boldsymbol{r}$ nach der Zeit. [Mon05]:

$$\frac{d\boldsymbol{v}}{dt} = -\frac{1}{\rho}\nabla p + \boldsymbol{a}^{other} \tag{5}$$

$$\frac{d\boldsymbol{r}}{dt} = \boldsymbol{v} \tag{6}$$

Dabei ist ρ die Dichte und p der Druck des Fluids an der Position des Partikels. $\boldsymbol{a}^{other}$ beschreibt die Beschleunigung durch weitere Kräfte, beispielsweise Gravitation. Der Nabla Operator $\nabla = \left(\frac{\partial}{\partial x} \; \frac{\partial}{\partial y} \; \frac{\partial}{\partial z} \right)^T$ wird zum Bilden von räumlichen Ableitungen verwendet. ∇p ist der *Gradient* des Drucks [Ngu07, Kapitel 30]. Das heißt ein Vektor, der für jede Koordinatenachse die Änderungsrate enthält und in Richtung des größten Anstieges zeigt. Aus Gleichung 5 resultiert eine Beschleunigung entgegen des Druckgradienten. Partikel in einem Fluid bewegen sich also von Orten höheren Drucks zu Orten niedrigeren Drucks.

Der Druck selbst wird durch eine sogenannte *Zustandsgleichung* berechnet. Eine Möglichkeit ist das Verwenden der Gleichung für ideale Gase. Dabei ist der Druck eine Funktion von Dichte und thermischer Energie. In vielen Fällen ist es allerdings nicht notwendig die thermische Energie zu betrachten. Wenn ein Gas zum Beispiel während einer langsamen Kompression schnell genug abkühlt, kann es als *isothermal* betrachtet werden. Es hat überall die gleiche Temperatur. In diesem Fall lässt sich die Zustandsgleichung vereinfachen, sodass der Druck nur noch von der Dichte ρ abhängig ist [Mon05] [BLRY07].

Eine Zustandsgleichung, die häufig verwendet wird, ist [RL99] [Spr02] [BBB03] [Spr05]:

$$p = A\rho^{\gamma} \tag{7}$$

Wobei A ein Parameter für die *Entropie pro Masse* ist und γ den *adiabatischen Index* des Gases bezeichnet. Bei idealen Gasen in drei Dimensionen ist $\gamma = \frac{5}{3}$. Der berechnete Druck ist immer positiv, was sich in rein abstoßenden Kräften zwischen den Partikeln äußert. Das Ergebnis ist ein kompressibles Fluid, welches sich natürlich im Raum ausbreitet. Je höher A, um so größer werden Druck und Druckunterschiede. Entsprechend steigen auch die abstoßenden Kräfte.

Für inkompressible Fluide, die eine konstante Dichte in Raum und Zeit haben, wird meist eine Ruhedichte ρ_0 festgelegt. Die Zustandsgleichung wird dann so formuliert, dass das Fluid dazu neigt, seine Ruhedichte zu halten. Folgende Gleichungen kommen häufig in der Literatur vor [MCG03] [Mon05] [IOS$^+$14]:

$$p = k(\rho - \rho_0) \tag{8}$$

$$p = k\left(\left(\frac{\rho}{\rho_0}\right)^7 - 1\right) \tag{9}$$

2.4 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrondynamics oder kurz *SPH* ist ein Interpolationsverfahren, was es erlaubt, Werte von durch Partikel dargestellten Feldern an beliebigen Punkten zu bestimmen (auch wenn sich dort kein Partikel befindet) [MCG03]. Außerdem können räumliche Ableitungen über solche Felder berechnet werden. *SPH* wurde 1977 von Monaghan und Gingold für Simulationen in der Astrophysik entwickelt [GM77]. Die Methode ist aber sehr allgemein formuliert und wurde seitdem für Simulationen in vielen anderen Bereichen eingesetzt [Mon05] [IOS$^+$14] [DG96].

Zum Bestimmen des Wertes von Feld A an der Stelle $\boldsymbol{r}$ wird ein Integral über die Umgebung gebildet, wobei alle Werte durch eine Kernelfunktion gewichtet werden. Die Gewichtung erfolgt je nach Abstand zum Punkt $\boldsymbol{r}$. Für numerisches Arbeiten wird das Integral durch eine Summe über alle umliegenden Partikel ersetzt.

$$A(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{10}$$

Dabei läuft die Summe über alle Partikel und der Index j markiert die Eigenschaften des aktuell betrachteten Partikels. Die Funktion $W(\boldsymbol{x}, h)$ ist der Interpolationskernel mit Radius h. Zur Veranschaulichung kann er wie eine Gausfunktion betrachtet werden, auch wenn in der Praxis meist andere Funktionen Verwendung finden (siehe Abschnitt 2.6) [Mon92].

Um Ableitungen des Feldes zu bestimmen, werden die Ableitungen der Kernelfunktion gebildet. Der *Gradient* ∇A eines Skalarfeldes A an der Stelle $\boldsymbol{r}$ berechnet sich durch:

$$\nabla A(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{11}$$

$\boldsymbol{V}$ ist ein Vektorfeld und ρ_r die Dichte an der Stelle $\boldsymbol{r}$. Dann lassen sich analog zu 11 *Divergenz*, *Curl* und der *Laplace-Operator* bestimmen. Folgende Formulierungen liefern allerdings bessere Ergebnisse [Mon92] [MCG03]:

$$\nabla \cdot \boldsymbol{V}(\boldsymbol{r}) = \frac{1}{\rho_r} \sum_j m_j \left(\boldsymbol{V}_j - \boldsymbol{V}_r\right) \cdot \nabla W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{12}$$

$$\nabla \times \boldsymbol{V}(\boldsymbol{r}) = \frac{1}{\rho_r} \sum_j m_j \left(\boldsymbol{V}_r - \boldsymbol{V}_j\right) \times \nabla W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{13}$$

$$\nabla^2 A(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{14}$$

2.5 Anwendung von SPH auf die Gasdynamik

Mit Hilfe von SPH kann ein Gas durch ein Partikelsystem modelliert und simuliert werden. Jeder Partikel i erhält anfänglich eine Masse m_i und eine Position $\boldsymbol{r}_i$. Die Dichte der Partikel wird über Gleichung 10 berechnet. Beim Einsetzen von ρ für A fällt ρ_j durch Kürzen weg:

$$\rho_i = \sum_j m_j W(\boldsymbol{r}_{ij}, h) \tag{15}$$

Dabei ist $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$. Mit der Dichte ρ wird der Druck p über eine der in Abschnitt 2.3 vorgestellten Zustandsgleichungen berechnet.

Um die aus dem Druck resultierende Beschleunigung der Partikel zu bestimmen, wird Gleichung 5 gelöst. Ihre SPH-Form sieht folgendermaßen aus:

$$\frac{d\boldsymbol{v}_i}{dt} = -\frac{1}{\rho_i} \sum_j m_j \frac{p_j}{\rho_j} \nabla W(\boldsymbol{r}_{ij}, h) \tag{16}$$

Bei Betrachtung der Wechselwirkung zwischen nur zwei Partikeln fällt auf, dass beide eine unterschiedliche Beschleunigung erhalten. Das verstößt gegen das dritte newtonsche Gesetz, nachdem jede Kraft zwischen zwei Körpern mit einer gleich starken entgegengesetzten Kraft einhergeht. Das Ergebnis wäre eine Verletzung der Impulserhaltung. [Mon05]

Es gibt verschiedene Ansätze, um Gleichung 16 in eine symmetrische Form zu bringen. Zum Beispiel

$$\frac{d\boldsymbol{v}_i}{dt} = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) \nabla W(\boldsymbol{r}_{ij}, h) \tag{17}$$

aus [Mon92], was von vielen anderen Autoren übernommen wurde. Eine alternative Form wird von Müller et al. in [MCG03] vorgeschlagen:

$$\frac{d\boldsymbol{v}_i}{dt} = -\frac{1}{\rho_i} \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\boldsymbol{r}_{ij}, h) \tag{18}$$

Hernquist & Katz [HK89] verwenden das Geometrische Mittel:

$$\frac{d\boldsymbol{v}_i}{dt} = -\sum_j m_j \frac{2\sqrt{p_i + p_j}}{\rho_i \rho_j} \nabla W(\boldsymbol{r}_{ij}, h) \tag{19}$$

Zu der so berechneten Beschleunigung des Partikels können dann weitere Beschleunigungen addiert werden. Diese stammen zum Beispiel aus der Interaktion des Benutzers oder weiteren Physikalischen Effekten wie Gravitation. Ist die Gesamtbeschleunigung für jeden Partikel bestimmt, wird ein Verfahren zum numerischen Integrieren von gewöhnlichen Differenzialgleichungen verwendet, um Geschwindigkeit und Position zu bestimmen. Viele *SPH*-Codes verwenden das *Leapfrog*-Schema [MH07] [Spr05] (siehe Kapitel 3.3.3).

Der Pseudocode 1 zeigt beispielhaft den Ablauf einer einfachen *SPH* Simulation. Zur Veranschaulichung hier mit Euler-Integration.

Algorithmus 1 : Beispielhafter Ablauf einer einfachen *SPH* Simulation

foreach *Partikel i* **do**
  Setze Startwerte für m_i und $\boldsymbol{r}_i$
end
while *Simulation läuft* **do**
  foreach *Partikel i* **do**
    Berechne ρ_i mit Gleichung 15
    Berechne p_i aus ρ_i mit Gleichung 7, 8 oder 9
  end
  bei Paralleler Ausführung: warte bis alle Partikel fertig
  foreach *Partikel i* **do**
    Berechne $\frac{d\boldsymbol{v}}{dt}$ mit Gleichung 17 oder 18
    Addiere eventuelle weitere Beschleunigungen
  end
  bei Paralleler Ausführung: warte bis alle Partikel fertig
  foreach *Partikel i* **do**
    $\boldsymbol{v}_i = \boldsymbol{v}_i + \frac{d\boldsymbol{v}}{dt}\Delta t$
    $\boldsymbol{r}_i = \boldsymbol{r}_i + \boldsymbol{v_i}\Delta t$
  end
end

2.6 Interpolationskernel

Von der Wahl der Kernelfunktion $W(\boldsymbol{x}, h)$ hängen Stabilität, Genauigkeit und Geschwindigkeit der *SPH*-Methode ab. Meist werden Kernel so definiert, dass $W(\boldsymbol{x}, h) = 0$ für $||\boldsymbol{x}|| > h$ oder $||\boldsymbol{x}|| > 2h$ gilt. Partikel, die weiter als h beziehungsweise $2h$ vom Interpolationspunkt $\boldsymbol{r}$ entfernt sind, fließen so nicht in die Berechnung ein. Das erhöht die Geschwindigkeit der

Implementierung erheblich und erlaubt das Verwenden von Beschleunigungsdatenstrukturen (siehe Kapitel 3.5.2).

Der Kernel sollte normalisiert und symmetrisch sein, also die folgenden beiden Gleichungen erfüllen [MCG03]:

$$\int W(\boldsymbol{r}) d\boldsymbol{r} = 1 \tag{20}$$

$$W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h) \tag{21}$$

Häufig wird ein auf Splinefunktionen basierender Kernel verwendet. Zum Beispiel [Mon92] [Spr05]

$$W_{spline}(\boldsymbol{x}, h) = \frac{8}{\pi h^3} \begin{cases} 1 - 6\left(\frac{||\boldsymbol{x}||}{h}\right)^2 + 6\left(\frac{||\boldsymbol{x}||}{h}\right)^3, & 0 \leq \frac{||\boldsymbol{x}||}{h} \leq \frac{1}{2} \\ 2\left(1 - \frac{||\boldsymbol{x}||}{h}\right)^3, & \frac{1}{2} < \frac{||\boldsymbol{x}||}{h} \leq 1 \\ 0, & \frac{||\boldsymbol{x}||}{h} > 1 \end{cases} \tag{22}$$

Es ist auch möglich für verschiedene Partikeleigenschaften unterschiedliche Kernelfunktionen zu verwenden. Müller et al. schlagen in [MCG03] die Verwendung folgenden Kernel vor:

$$W_{poly6}(\boldsymbol{x}, h) = \frac{315}{64\pi h^9} \begin{cases} \left(h^2 - ||\boldsymbol{x}||^2\right)^3, & 0 \leq ||\boldsymbol{x}|| \leq h \\ 0, & ||\boldsymbol{x}|| > h \end{cases} \tag{23}$$

$$W_{spiky}(\boldsymbol{x}, h) = \frac{15}{\pi h^6} \begin{cases} (h - ||\boldsymbol{x}||)^3, & 0 \leq ||\boldsymbol{x}|| \leq h \\ 0, & ||\boldsymbol{x}|| > h \end{cases} \tag{24}$$

Der Vorteil von W_{poly6} ist, dass $\boldsymbol{x}$ nur als $||\boldsymbol{x}||^2$ vorkommt. Bei der Distanzberechnung fällt das Wurzelziehen weg, was Rechenzeit einspart. Der Gradient von W_{poly6} geht im Mittelpunkt gegen null. Wird er zum Berechnen des Druckgradienten verwendet, stoßen sich sehr nahe Partikel nicht stark genug ab. Im Fluid bilden sich unnatürliche Klumpen. Der Kernel W_{spiky} aus [DG96] löst dieses Problem, sein Gradient geht im Mittelpunkt nicht gegen null. Er wird nur für die Berechnung des Druckgradienten verwendet.

Zur Vollständigkeit hier die Gradienten von W_{poly6} und W_{spiky}:

$$\nabla W_{poly6}(\boldsymbol{x}, h) = \frac{945}{32\pi h^9} \begin{cases} \left(h^2 - ||\boldsymbol{x}||^2\right)^2 \boldsymbol{x}, & 0 \leq ||\boldsymbol{x}|| \leq h \\ 0, & ||\boldsymbol{x}|| > h \end{cases} \tag{25}$$

$$\nabla W_{spiky}(\boldsymbol{x}, h) = -\frac{45}{\pi h^6} \begin{cases} (h - ||\boldsymbol{x}||)^2 \frac{\boldsymbol{x}}{||\boldsymbol{x}||}, & 0 \leq ||\boldsymbol{x}|| \leq h \\ 0, & ||\boldsymbol{x}|| > h \end{cases} \tag{26}$$

2.7 Viskosität

Mit dem bis hier gezeigten lassen sich Gase bereits korrekt simulieren, sofern keine Schockeffekte auftreten und keine externe Erwärmung stattfindet. Als Schock bezeichnet man eine Diskontinuität im Fluss des Fluids. Solche Diskontinuitäten entwickeln sich schnell, wenn ideale Gase betrachtet werden [Spr05]. Um Schockeffekte zu behandeln und die Simulation durch Ausgleichen von Integrationsfehlern stabiler zu machen, wird ein Term für künstliche Viskosität eingeführt. [Mon05]

Theoretisch könnte die Viskosität direkt über die *SPH*-Methode bestimmt werden, was aber die Erhaltung von Impuls und Drehimpuls nicht garantiert [Mon05]. In der Computergrafik verwendet man häufig den Laplace Operator des Geschwindigkeitsfeldes (z.B. [MCG03], [IOS+14]). Monaghan und Gingold [MG83] stellen stattdessen einen Viskositätsfaktor Π_{ij} vor, der zu den Drucktermen in Gleichung 17 addiert wird:

$$\frac{d\boldsymbol{v}_i}{dt} = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2} + \Pi_{ij} \right) \nabla W(\boldsymbol{r}_{ij}, h) \tag{27}$$

Der Viskositätsfaktor selbst ist dabei definiert durch

$$\Pi_{ij} = \begin{cases} -\alpha \frac{h\bar{c}_{ij}}{\bar{\rho}_{ij}} \frac{\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij}}{||\boldsymbol{r}_{ij}||^2 + \epsilon h^2}, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} > 0 \\ 0, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} \leq 0 \end{cases} \tag{28}$$

mit $\epsilon \sim 0,001$ um Probleme zu vermeiden, wenn $r_{ij} = 0$ ist. $\bar{\rho}_{ij} = (\rho_i + \rho_j)/2$ ist das arithmetische Mittel der Dichte beider Partikel und $\bar{c}_{ij}$ das ihrer Schallgeschwindigkeit. Die Fallunterscheidung sorgt dafür, dass die Viskosität nur aktiv ist, wenn sich die Partikel aufeinander zu bewegen. Entfernen sie sich voneinander, ist die Viskosität abgeschaltet, um die Ausdehnung des Gases nicht zu behindern. α ist ein Eingabeparameter, der die Stärke der Viskosität einstellt. Die Schallgeschwindigkeit berechnet sich durch [RL99]:

$$c_i = \sqrt{\frac{\gamma p_i}{\rho_i}} \tag{29}$$

Die Formulierung des Viskositätsfaktors wurde mit der Zeit weiterentwickelt bis hin zu [Mon97]:

$$\Pi_{ij} = -\frac{\alpha}{2} \frac{w_{ij} v_{ij}^{sig}}{\rho_i} \tag{30}$$

mit

$$v_{ij}^{sig} = c_i + c_j - 3w_{ij} \tag{31}$$

$$w_{ij} = \begin{cases} \frac{\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij}}{||\boldsymbol{r}_{ij}||}, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} > 0 \\ 0, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} \leq 0 \end{cases} \tag{32}$$

Springel [Spr05] erklärt, dass die neuere Form der Viskosität (Gleichung 30) in all seinen Tests gleich gute oder bessere Ergebnisse im Vergleich mit der älteren Version (Gleichung 28) geliefert habe.

Diese Viskosität verschwindet während Scherströmungen nicht komplett. Eine Scherströmung liegt vor, wenn Fluid-Partikel sich parallel zueinander mit unterschiedlicher Geschwindigkeit bewegen. Diese Eigenschaft kann die Bildung von Akkretionsscheiben behindern. Besonders wenn diese nur durch wenige Partikel repräsentiert werden (ab etwa 1000 Partikeln), kommt es zu starken Störungen [Ste96]. Um das zu verhindern, wird Π_{ij} mit $(f_i^{visc} + f_j^{visc})/2$ multipliziert. Der Faktor

$$f_i^{visc} = \frac{||\nabla \cdot \boldsymbol{v}_i||}{||\nabla \cdot \boldsymbol{v}_i|| + ||\nabla \times \boldsymbol{v}_i|| + 0.0001 c_i/h} \tag{33}$$

nach [Bal95] gibt die Stärke der Scherung in der Strömung an. In einer scherungsfreien Strömung ist $\nabla \cdot \boldsymbol{v}_i \neq 0$ und $\nabla \times \boldsymbol{v}_i = 0$, f_i^{visc} wird 1 und die Viskosität wird komplett nach Gleichung 30 berechnet. In einer reinen Scherströmung ist es genau umgekehrt. Hier ist $\nabla \cdot v_i = 0$ und $\nabla \times v_i \neq 0$, f_i^{visc} wird 0 und unterdrückt die Viskosität komplett. $\nabla \cdot v_i$ und $\nabla \times v_i$ werden mit dem *SPH*-Formalismus berechnet, wie in Abschnitt 2.4 (Formel 12 und 13) beschrieben.

Die hier dargestellte Formulierung der Viskosität kann zusätzlich in der Gleichung für thermische Energie verwendet werden, wenn das Aufwärmen des Gases durch die Viskosität betrachtet werden soll, siehe [Mon05].

2.8 Variabler Kernelradius

Die Auflösung einer SPH Simulation wird durch die Anzahl der Partikel und den Kernelradius h bestimmt. Strukturen kleiner als der Kernelradius werden stark weichgezeichnet und daher nicht gut abgebildet [SM93]. Außerdem wird die Simulationsberechnung aufwendiger. Bei der Kompression des Gases erhalten die Partikel in den Bereichen mit hoher Dichte immer mehr Nachbarn, für die Kernelfunktionen berechnet werden müssen. Es scheint logisch einen sehr kleinen Wert für h zu verwenden. Dies führt allerdings zu Problemen in Bereichen geringer Dichte, wo die Abstände zwischen den Partikeln groß sind. Partikel haben dann nur sehr wenige oder sogar gar keine Nachbarn, wodurch sie sich nicht mehr wie Gas verhalten. [HK89]

Die Lösung liegt darin, h in Raum und Zeit zu variieren. Jedem Partikel wird ein eigener Kernelradius zugewiesen. Dieser wird dann in jedem Simulationsschritt angepasst, um die Anzahl der Nachbarn in etwa konstant zu halten. Für die gewünschte Anzahl der Nachbarn $\mathscr{N}_s$ wird meist ein Wert zwischen 20 und 100 gewählt. Häufig liest man von etwa 50 Nachbarn (z.B. [BBB03]), was auch in den in Abschnitt 5 beschriebenen Simulationen gute Ergebnisse lieferte.

Es gibt verschiedene Formeln die verwendet werden können um h anzupassen. Hernquist und Katz [HK89] verwenden:

$$h_i^n = h_i^{n-1} \frac{1}{2} \left[1 + \left(\frac{\mathcal{N}_s}{\mathcal{N}_i^{n-1}} \right)^{\frac{1}{3}} \right] \tag{34}$$

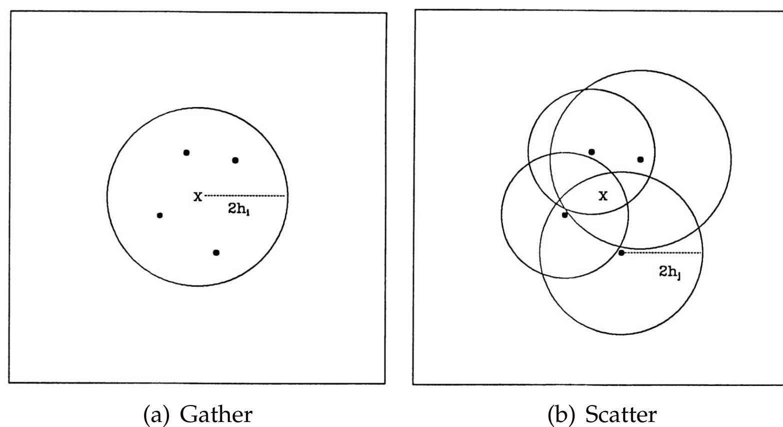
h_i^n ist dabei der neue und h_i^{n-1} der vorherige Kernelradius. $\mathcal{N}_i^{n-1}$ bezeichnet die Anzahl der Nachbarn im vorherigen Simulationsschritt. Eine andere Möglichkeit ist es h_i von der Dichte abhängig zu machen. [Mon05]

$$h_i = 1,3 \left(\frac{m_i}{\rho_i} \right)^{\frac{1}{3}} \tag{35}$$

Es kann sinnvoll sein, einen Maximalwerte für h festzulegen, um die Interaktion zwischen Gebieten mit sehr hoher und sehr niedriger Dichte zu verhindern [Mon05]. Ein Minimalwert kann Instabilitäten reduzieren, die bei besonders kleinen Kernelradien auftreten.

Steinmetz und Müller schlagen in [SM93] einen verbesserten, aber recht komplexen, Algorithmus vor, mit dem sich Instabilitäten und Oszillationen noch weiter reduzieren lassen.

Es existieren verschiedene Interpretationen davon, was ein *SPH*-Partikel mit dem Kernelradius h tatsächlich darstellt. Zum einen, die in Kapitel 2.4 beschriebene *Gather* Interpretation, bei der ein Partikel die Werte aller anderen Partikel in seinem Kernelradius gewichtet aufsummiert. Zum anderen die *Scatter* Interpretation, bei der die Attribute eines Partikels in einer Kugel mit Radius h verschmiert sind [HK89]. Die beiden Varianten sind in Abbildung 6 dargestellt.



(a) Gather (b) Scatter

Abbildung 6: *Gather* und *Scatter* Interpretation von *SPH*. [HK89]

Haben alle Partikel den selben Radius sind beide Interpretationen identisch. Bei unterschiedlichen Radien zeigt sich die Wahl der Interpretation darin, ob für die Berechnung einer Kernelfunktion h_i oder h_j verwendet wird. Für die Berechnung der Dichte gibt es zwei Möglichkeiten:

$$\rho_i = \sum_j m_j W(\boldsymbol{r}_{ij}, h_j) \tag{36}$$

$$\rho_i = \sum_j m_j W(\boldsymbol{r}_{ij}, h_i) \tag{37}$$

Wobei Formel 36 der Scatter und 37 der Gather Interpretation entspricht.

Bei der Berechnung der Beschleunigungen tritt ein Problem auf (was in einem ähnlichen Kontext schon in Abschnitt 2.5 behandelt wurde), da beide Interpretationen nicht symmetrisch sind. Überträgt man die Ansätze direkt auf die Beschleunigungsberechnung, führt dies zur Verletzung des dritten newtonschen Gesetzes und die Impulserhaltung ist nicht mehr garantiert.

Die einfachste Variante der Symmetrisierung ist das Verwenden des arithmetischen Mittels der beiden Radien. Die Kernelfunktion für den Druckgradienten wird dann mit $h = (h_i + h_j)/2$ berechnet. Für die Bestimmung der Dichte wird die asymmetrische Form beibehalten. Hernquist und Katz [HK89] berichten, dass es dadurch zu Fehlern bei der Integration gekommen sei. Sie schlagen stattdessen vor, $W(\boldsymbol{r}_{ij}, h)$ durch $\overline{W}_{ij} = \frac{1}{2}\left[W(\boldsymbol{r}_{ij}, h_i) + W(\boldsymbol{r}_{ij}, h_j)\right]$ zu ersetzen. Dies kann sowohl für den Druckgradienten als auch die Dichte eingesetzt werden.

Wird ein variabler Kernelradius verwendet, erfordert dies in der Theorie das Betrachten von ∇h Termen in den *SPH*-Gleichungen. In der Praxis verbessert das Einbeziehen dieser Terme die Erhaltung von Energie und Entropie. Die Implementierung ist aber recht aufwendig und hat sich daher in der Astrophysik nicht durchgesetzt. Springel und Hernquist stellen in [Spr02] eine neue Formulierung zur Berechnung der Beschleunigung vor.

$$\frac{d\boldsymbol{v}_i}{dt} = -\sum_j m_j \left[f_i^{press} \frac{p_i}{\rho_i^2} \nabla W(\boldsymbol{r}_{ij}, h_i) + f_j^{press} \frac{p_j}{\rho_j^2} \nabla W(\boldsymbol{r}_{ij}, h_j) \right] \tag{38}$$

Dabei wird f_i definiert durch:

$$f_i^{press} = \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i}\right)^{-1} \tag{39}$$

Die ∇h Terme sind hier implizit enthalten. Der Kernelradius wird so angepasst, dass die Masse unter der Kernelfunktion konstant bleibt. Er ist beschrieben durch:

$$\frac{4\pi}{3} h_i^3 \rho_i = \overline{m} \mathcal{N}_s \tag{40}$$

Dabei ist $\overline{m}$ die durchschnittliche Masse eines Partikels. Die Beschleunigung durch Viskosität wird zusätzlich berechnet durch:

$$\left.\frac{d\boldsymbol{v}_i}{dt}\right|_{visc.} = -\sum_j m_j \Pi_{ij} \nabla \overline{W}_{ij}(\boldsymbol{r}_{ij}, h_j) \tag{41}$$

Hier ist Π_{ij} der in Abschnitt 2.7 beschriebene Viskositätsfaktor und $\overline{W}_{ij} = \frac{1}{2}\left[W(\boldsymbol{r}_{ij}, h_i) + W(\boldsymbol{r}_{ij}, h_j)\right]$ wie oben beschrieben.

3 Konzeption

3.1 Ziele der Simulation

Im diesem Kapitel wird auf Basis der in Abschnitt 2 eingeführten Grundlagen ein Konzept für den Simulationscode erarbeitet. Ziel ist es, möglichst viele der physikalischen Prinzipien und Vorgänge plausibel abzubilden und die Ergebnisse intuitiv verständlich zu visualisieren. Das Programm sollte auf einem handelsüblichen Computer interaktiv ausführbar sein, also einige Simulationsschritte – inklusive Visualisierung – pro Sekunde durchführen.

Um dies zu erreichen, beschränkt sich die Simulation auf das Anfangsstadium der Sternentstehung wie in Abschnitt 2.1 beschrieben. Die Ausgangslage ist eine rotierenden Gaswolke mit turbulentem Geschwindigkeitsfeld. Der Zusammenfall unter der eigenen Gravitationskraft und die damit einhergehende Fragmentierung bis zur Entstehung von *Protosternen* soll gut abgebildet sein. Danach werden möglichst viele Merkmale der folgenden Akkretionsphase erkennbar gemacht.

Der weitere Lebenszyklus der entstandenen Sterne ist nicht Teil der Simulation. Die innere Entwicklung der Sterne wird nicht berücksichtigt. Auch der Tod einzelner Sterne oder das Zurückkehren von Materie in das *ISM* kann hier nicht simuliert werden.

Außerdem sollte sich die Auflösung und Genauigkeit der Simulation leicht an die verfügbare Rechenleistung anpassen lassen.

3.2 Modellannahmen

Um etwas so komplexes wie die Entstehung von Sternen am Computer abbilden und simulieren zu können, muss ein Modell erstellt werden, das mit den Zielen der Simulation im Einklang steht. Viele der hier getroffenen Vereinfachungen finden sich an verschiedenen Stellen in der Literatur wieder.

Im Modell beginnt die Entstehung von Sternen mit einer sphärischen Gaswolke. Dichte und Temperatur sind überall gleich. Die Gaswolke erhält eine anfängliche Rotationsgeschwindigkeit und ein turbulentes Geschwindigkeitsfeld. Die für diesen Zustand verantwortlichen Effekte werden nicht betrachtet.

Zwischen den unterschiedlichen Bestandteilen der Gaswolke wird nicht unterschieden. Es wird wie ein reines Gas behandelt. Diese Annahme scheint aufgrund der in Abschnitt 2.1 beschriebenen Zusammensetzung des *ISM* logisch.

Wenn Teile einer *GMC* beginnen zusammenzufallen, kann die entstandene Wärme zunächst durch Strahlung abtransportiert werden. Das Gas wird im Modell daher als *isothermal* behandelt und thermische Energie nicht in den Berechnungen berücksichtigt. Im späteren Verlauf spielt der aus

steigender Temperatur entstehende Druck eine wichtige Rolle. Um den Effekt zu approximieren wird die Gleichung zur Druckberechnung ab einer festgelegten Dichte verändert (siehe Abschnitt 3.3.1). In [BBB03] wird eine ähnliche Methode angewandt.

Dichte, Druck und das Geschwindigkeitsfeld der Gaswolke werden durch das Modell beschrieben. Das Aktualisieren der Werte geschieht auf Basis von Gravitation, Gasdruck und Viskosität. Andere Eigenschaften und Effekte, die das Verhalten des Gases beeinflussen könnten (z.B. Magnetfelder), werden nicht modelliert.

Außerhalb der gleichmäßigen Gaswolke ist die Dichte des Gases in diesem Modell sehr gering. Die Umgebung wird als perfektes Vakuum angenommen.

3.3 Verwendete Techniken

Zum Visualisieren der simulierten Daten wird *OpenGL* genutzt, da es unabhängig von verwendetem Betriebssystem und Grafikkarte ist.

Aktuelle Versionen von *OpenGL* unterstützen das Durchführen von allgemeinen Berechnungen auf der *GPU* durch sogenannte *compute shader*. Die massiv parallele Architektur von *GPUs* ermöglicht die Beschleunigung von vielen Algorithmen. Dafür müssen einige Einschränkungen umgangen werden. Besondere Sorgfalt bei der Planung und Umsetzung der Anwendung ist erforderlich. Das Auslagern von Berechnungen auf die *GPU* lohnt sich um so mehr, wenn die entstehenden Daten danach auf dem Bildschirm dargestellt werden sollen. Hier wird zusätzlich die Datenübertragung vom Hauptspeicher in den Grafikspeicher eingespart.

Der in dieser Arbeit beschriebene Simulationscode nutzt *compute shader* für einen Großteil der Berechnungen. Als Programmiersprache für die Anwendungsteile, welche auf der *CPU* ausgeführt werden, wurde *C++* gewählt.

3.3.1 Diskretisieren der Gaswolke

Das Gas wird durch ein Partikelsystem dargestellt. Zum Modellieren der Gaseigenschaften wird *SPH* (siehe Kapitel 2.4) verwendet. *SPH* Partikelsimulationen haben in diesem Szenario einige Vorteile gegenüber rasterbasierten Simulationstechniken.

Besonders interessant sind Gasregionen mit hoher Dichte, da dort Sterne entstehen. Eine hohe Dichte in *SPH* bedeutet, dass sich viele Partikel in dieser Region befinden. Die Auflösung erhöht sich also automatisch in den interessanten und verringert sich in den uninteressanten Regionen. Berechnungen werden außerdem nur für Teile des Simulationsbereiches durchgeführt, in denen sich auch Gas befindet. Um diesen Effekt bei einer

rasterbasierten Simulation zu erreichen, muss das Raster dynamisch angepasst werden. Partikelsimulationen sind besonders für das Implementieren auf der *GPU* geeignet. Alle Partikel können parallel – jeder in seinem eigenen Thread – bearbeitet werden. Zusätzlich gestaltet sich die Visualisierung von Partikeln einfacher. Die Rendering-Pipeline der Grafikkarte kann direkt genutzt werden, um an der Position jedes Partikels einen Punkt, eine Scheibe oder eine Textur darzustellen.

Zum Erreichen einer hohen Auflösung, auch bei kleinen Partikelzahlen, ist der Kernelradius variabel. Er wird für jeden Partikel in jedem Simulationsschritt aktualisiert. Dafür wird der Formalismus von Springel und Hernquist [Spr02] übernommen. Der Kernelradius ist durch Gleichung 40 definiert. Für die Berechnung der Dichte wird – wie in [Spr05] – die *Gather* Interpretation (Gleichung 37) gewählt. Diese lässt sich auch effizienter implementieren als Gleichung 36 oder eine symmetrische Formulierung, da das für jeden Partikel unterschiedliche h_j nicht berücksichtigt werden muss.

Da das Gas im Modell *isotherm* ist, wird die thermische Energie in der Zustandsgleichung weggelassen. Sie nimmt dann die Form von Gleichung 7 an. Um den Effekt von erhöhtem Druck durch steigende Temperatur in den späteren Phasen des Zusammenfalls der Wolke abzubilden, wird γ in Abhängigkeit der Dichte ρ wie in [BBB03] angepasst:

$$\gamma = \begin{cases} \gamma_{low}, & \rho < \rho_{thres} \\ \gamma_{high}, & \rho \geq \rho_{thres} \end{cases} \tag{42}$$

Dabei sind γ_{low}, γ_{high} und ρ_{thres} Eingabeparameter.

Die Beschleunigung der Partikel besteht aus drei Komponenten: Druck, Viskosität und Gravitation. Um die aus dem Druckgradienten folgende Beschleunigung zu bestimmen wird Gleichung 38 verwendet. Dies erfordert die Berechnung von $\frac{\partial \rho_i}{\partial h_i}$, was hier durch die *Zentraldifferenzmethode* geschieht.

Die Beschleunigung durch Viskosität berechnet sich über Gleichung 41. Der Viskositätsfaktor Π_{ij} ist – wie in Kapitel 2.7 erläutert – definiert durch:

$$\Pi_{ij} = -\frac{\alpha}{2} \frac{\left(f_i^{visc} + f_j^{visc}\right)}{2} \frac{w_{ij} v_{ij}^{sig}}{\rho_i} \tag{43}$$

Zum Lösen der *SPH*-Gleichungen für Druck und Viskosität wird der Kernel W_{spiky} (Gleichung 24) beziehungsweise dessen Gradient verwendet. In allen anderen Fällen der Kernel W_{poly6} (Gleichung 23). Die Auswahl erfolgte hauptsächlich aufgrund der effizienteren Implementierung. W_{poly6} arbeitet nur mit dem Quadrat der Distanz, sodass das Wurzelziehen entfällt. Außerdem ist der Vorfaktor ($\frac{315}{64\pi h^9}$ beim W_{poly6}) nur vom Kernelradius h abhängig. Beim Lösen der Gleichungen 37, 38 und 41 wird unter anderem die Kernelfunktion mit Radius h_i für alle Nachbarn berechnet. Der Vorfaktor kann dann einmalig im Vorfeld berechnet und abgespeichert werden, was Rechenzeit einspart.

Die aus Gravitation resultierende Beschleunigung wird durch Gleichung 4 berechnet. Ist dabei der Dämpfungsfaktor ϵ zu groß, verhindert dies das Zusammenfallen der Wolke. Ist er zu klein, führt dies zur Bildung von unnatürlichen Gasklumpen. Die Lösung ist ϵ für jeden Partikel an dessen SPH Kernelradius h anzupassen. Damit ist die Auflösung der Gravitationskraft abhängig von der Auflösung des Gasdrucks. Die Anpassung geschieht hier durch:

$$\epsilon_i = h_i \epsilon_{scale} \tag{44}$$

Dabei ist ϵ_{scale} ein Eingabeparameter

Um asymmetrische Gravitationskräfte zwischen Partikeln mit unterschiedlichem ϵ_i zu vermeiden wird in der Berechnung das geometrische Mittel $\sqrt{\epsilon_i \epsilon_j}$ verwendet. Dies ist kein Problem für die Performance, da in der Gleichung zum Berechnen der Gravitationskräfte ϵ nur als ϵ^2 vorkommt.

Alle drei Beschleunigungen werden addiert, um die neue Geschwindigkeit und Position der Partikel zu berechnen (siehe Abschnitt 3.3.3).

3.3.2 Anfangs- und Randbedingungen

Am Anfang der Simulation sind die Partikel gleichmäßig in einem kugelförmigen Volumen verteilt. Der Radius der Kugel, sowie die Masse und Anzahl der Partikel sind Eingabeparameter (alle Partikel erhalten die selbe Masse). Ihre Anfangsgeschwindigkeit besteht aus einer Drehgeschwindigkeit und einem turbulenten Geschwindigkeitsfeld.

Für die Drehgeschwindigkeit ist eine Rotationsachse $\boldsymbol{\theta}$ als Eingabeparameter gegeben. Die Geschwindigkeit der Partikel wird über das Kreuzprodukt berechnet:

$$\boldsymbol{v}_i^{rot} = \boldsymbol{\theta} \times \boldsymbol{r}_i \tag{45}$$

Dazu wird die Geschwindigkeit aus dem turbulenten Geschwindigkeitsfeld addiert. Bate et al. [BBB03] generieren ein divergenzfreies, zufälliges Geschwindigkeitsfeld mit einem bestimmten Leistungsspektrum. Spätere Arbeiten [Bat09] zeigen aber, dass das Leistungsspektrum für die Eigenschaften der entstehenden Sterne nicht relevant ist.

In dieser Simulation wird das anfängliche, turbulente Geschwindigkeitsfeld über *curl-noise* generiert. Für *curl-noise* wird zunächst ein sogenanntes Potentialfeld $\boldsymbol{\Psi}$, ein Vektor mit drei Komponenten, benötigt. Er wird für jeden Partikel durch das Aufsummieren von *perlin-noise* verschiedener Frequenzen bestimmt. Über dem Potentialfeld wird der Curl-Operator $\nabla \times \boldsymbol{\Psi}$ berechnet. Das Ergebnis ist ein divergenzfreies Vektorfeld, dass als Geschwindigkeit genutzt werden kann [BHF07]. Die Berechnung des Curl-Operators erfolgt durch die SPH-Methode (Gleichung 13).

Da die Umgebung als perfektes Vakuum angenommen wird, erfolgt keine gesonderte Randbehandlung. Entfernen sich einzelne Partikel zu weit vom Zentrum der Gaswolke, erfahren sie kaum noch Beschleunigung durch das Gas und werden von der Gravitation wieder angezogen.

3.3.3 Zeitintegration

Um aus der Beschleunigung der Partikel eine neue Geschwindigkeit und Position zu berechnen, müssen die Differentialgleichungen der Bewegung integriert werden. Dazu wird hier der sogenannte *Leapfrog*-Algorithmus in seiner *kick-drift-kick* Form verwendet. Algorithmen wie *Runge-Kutta-4* haben zwar eine höhere Ordnung, liefern aber langfristig schlechtere Ergebnisse als *Leapfrog*, da letzterer *symplektisch* ist. Dies wird in [Spr05] ausführlich diskutiert. Außerdem ist *Leapfrog* einfach zu implementieren und benötigt nur wenig Rechenzeit.

Algorithmus 2 : Der Leapfrog Algorithmus in kick-drift-kick Form

Setze Startwerte $\boldsymbol{r}_0$ und $\boldsymbol{v}_0$
Berechne $\boldsymbol{a}_0$
foreach *Simulationsschritt i* **do**
1 $\boldsymbol{v}_{i+1/2} = \boldsymbol{v}_i + \boldsymbol{a}_i \frac{\Delta t}{2}$
2 $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i + \boldsymbol{v}_{i+1/2} \Delta t$
3 Berechne $\boldsymbol{a}_{i+1}$
4 $\boldsymbol{v}_{i+1} = \boldsymbol{v}_{i+1/2} + \boldsymbol{a}_{i+1} \frac{\Delta t}{2}$
end

Wie bei allen Algorithmen zum Integrieren der Bewegungsgleichungen, wird die Beschleunigung für die Zeit Δt (Zeitschritt) als konstant angenommen. Die daraus entstehenden Fehler werden größer, um so länger der Zeitschritt gewählt wird. Generell erfordern dichtere Regionen, in denen Gravitationskräfte, Druck und Viskosität größer sind, einen kleineren Zeitschritt, damit die Simulation stabil läuft. Regionen mit geringer Dichte kommen mit wesentlich größeren Zeitschritten aus. Wird am Anfang der Simulation ein Zeitschritt festgelegt, ist er vermutlich zu grob um die dichten Regionen im späteren Verlauf zu händeln. Ist der Zeitschritt von Anfang an sehr klein, kostet das viel Rechenzeit.

Besser ist es, die Länge des Zeitschrittes im Laufe der Simulation an die aktuellen Bedingungen anzupassen. In der *kick-drift-kick* Form ist dies auch beim *Leapfrog* möglich, solange Zeile 1 und 4 nicht zusammengefasst werden. Um den mindestens benötigten Zeitschritt für einen Partikel zu bestimmen, werden zwei Kriterien verwendet. Das erste berücksichtigt die Gravitationskraft, das zweite die Gasdynamik [Spr05].

$$\Delta t_i^{grav} = \sqrt{\frac{2\eta\epsilon_i}{||\boldsymbol{a}_i||}} \tag{46}$$

$$\Delta t_i^{hyd} = \frac{C_{courant} h_i}{\max_j \left(v_{ij}^{sig} \right)} \tag{47}$$

Dabei ist $\max_j$ das Maximum unter Berücksichtigung aller Partikel innerhalb des Kernelradius von i. η und $C_{courant}$ sind Eingabeparameter und stellen die Genauigkeit der Simulation ein. Kleinere Werte führen zu kleineren Zeitschritten und damit zu einer genaueren, aber langsameren Simulation. Für den Zeitschritt eines Partikels wird der kleinere der beiden Werte gewählt. Zusätzlich kann eine Ober- und Untergrenze festgelegt werden.

Ist der größte zulässige Zeitschritt für jeden Partikel berechnet muss das Minimum aller Zeitschritte gefunden werden. Dieses wird im nächsten Simulationsschritt als Zeitschritt für alle Partikel verwendet.

Während der Simulation kann der Unterschied von dichten zu weniger dichten Regionen sehr groß werden. Noch effizienter wäre es daher, jeden Partikel mit seinem individuellen Zeitschritt zu integrieren. Dies ist möglich, wurde hier allerdings nicht implementiert. Für eine ausführlichere Beschreibung wird erneut auf [Spr05] verwiesen.

3.4 Zusammenfassung des Simulationsalgorithmus

Der Pseudocode 3 und 4 zeigt den möglichen Ablauf eines Zeitschrittes wenn die oben beschriebene Techniken angewendet werden. Es werden alle zu berechnenden Werte inklusive der Formeln in einer sinnvollen Reihenfolge angegeben. Bei den *foreach* Schleifen kann jeder Partikel i von einem eigenen Thread bearbeitet werden. Die Punkte, an denen parallel laufende Threads synchronisiert werden müssen, sind ebenfalls markiert.

Dabei muss beachtet werden, dass der erste Zeitschritt etwas anders abläuft. In diesem müssen zunächst die Anfangswerte für alle Partikeleigenschaften eingestellt werden.

Algorithmus 3 : Aktualisieren der Simulation Teil 1

foreach *Partikel i* **do**

```
/* Geschwindigkeit und Position aktualisieren */
```

$\boldsymbol{v}_i = \boldsymbol{v}_i + \boldsymbol{a}_i \frac{\Delta t}{2}$

$\boldsymbol{r}_i = \boldsymbol{r}_i + \boldsymbol{v}_i \Delta t$

```
/* Anpassen des Kernelradius */
```

$h_i = \left(\frac{3\overline{m}\mathcal{N}_s}{4\pi\rho_i}\right)^{1/3}$

$h_i = \text{clamp}\left(h_i, h_{min}, h_{max}\right)$

end

Warte bis alle Partikel fertig

foreach *Partikel i* **do**

```
/* Berechnen der Dichte */
```

$\rho_i = \sum_j m_j W(\boldsymbol{r}_{ij}, h_i)$

$\rho_i^p = \sum_j m_j W(\boldsymbol{r}_{ij}, h_i + \epsilon_{zdm})$

$\rho_i^m = \sum_j m_j W(\boldsymbol{r}_{ij}, h_i - \epsilon_{zdm})$

$\frac{\partial \rho_i}{\partial h_i} = \frac{\rho_i^p - \rho_i^m}{2\epsilon_{zdm}}$

$f_i^{pres} = \left(1 + \frac{h_i}{3\rho_i}\frac{\partial \rho_i}{\partial h_i}\right)^{-1}$

```
/* Druck und Schallgeschwindigkeit */
```

$\gamma = \begin{cases} \gamma_{low}, & \rho < \rho_{thres} \\ \gamma_{high}, & \rho \geq \rho_{thres} \end{cases}$

$p = A\rho^\gamma$

$c_i = \sqrt{\frac{\gamma p_i}{\rho_i}}$

end

Warte bis alle Partikel fertig

foreach *Partikel i* **do**

```
/* Korrekturfaktor für die Viskosität */
```

$\nabla \cdot \boldsymbol{v}_i = \frac{1}{\rho_i} \sum_j \left(\boldsymbol{v}_j - \boldsymbol{v}_i\right) \cdot \nabla W(\boldsymbol{r}_{ij}, h)$

$\nabla \times \boldsymbol{v}_i = \frac{1}{\rho_i} \sum_j m_j \left(\boldsymbol{v}_{ij}\right) \times \nabla W(\boldsymbol{r}_{ij}, h)$

$f_i^{visc} = \frac{||\nabla \cdot v_i||}{||\nabla \cdot v_i|| + ||\nabla \times v_i|| + 0.0001 c_i / h}$

end

Warte bis alle Partikel fertig

Algorithmus 4 : Aktualisieren der Simulation Teil 2

foreach *Partikel i* **do**

```
/* Beschleunigung */
```

$$\boldsymbol{a}_i^{pres} = -\sum_j m_j \left[f_i^{pres} \frac{p_i}{\rho_i^2} \nabla W(\boldsymbol{r}_{ij}, h_i) + f_j^{pres} \frac{p_j}{\rho_j^2} \nabla W(\boldsymbol{r}_{ij}, h_j) \right]$$

$$\boldsymbol{a}_i^{visc} = -\sum_j m_j \Pi_{ij} \nabla \overline{W}_{ij}(\boldsymbol{r}_{ij}, h_j)$$

Mit:

$$\Pi_{ij} = -\frac{\alpha}{2} \frac{\left(f_i^{visc} + f_j^{visc}\right)}{2} \frac{w_{ij} v_{ij}^{sig}}{\rho_i}$$

$$v_{ij}^{sig} = c_i + c_j - 3w_{ij}$$

$$w_{ij} = \begin{cases} \frac{\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij}}{||\boldsymbol{r}_{ij}||}, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} > 0 \\ 0, & \boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} \leq 0 \end{cases}$$

Finde dabei $\max_j \left(v_{ij}^{sig} \right)$

$$\boldsymbol{a}_i^{grav} = G \sum_j \frac{m_j \boldsymbol{r_{ji}}}{\left(||\boldsymbol{r_{ji}}||^2 + \epsilon^2\right)^{3/2}}, \quad \epsilon^2 = h_i h_j \epsilon_{scale}^2$$

$$\boldsymbol{a}_i = \boldsymbol{a}_i^{pres} + \boldsymbol{a}_i^{visc} + \boldsymbol{a}_i^{grav}$$

end

Warte bis alle Partikel fertig

foreach *Partikle i* **do**

```
/* Geschwindigkeit */
```

$$\boldsymbol{v}_i = \boldsymbol{v}_i + \boldsymbol{a}_i \frac{\Delta t}{2}$$

```
/* Zeitschritt anpassen */
```

$$\Delta t_i^{grav} = \sqrt{\frac{2\eta h_i \epsilon_{scale}}{||\boldsymbol{a}_i||}}$$

$$\Delta t_i^{hyd} = \frac{C_{courant} h_i}{\max_j \left(v_{ij}^{sig}\right)}$$

$$\Delta t_i = \min \left(\Delta t_i^{grav}, \Delta t_i^{hyd} \right)$$

$$\Delta t_i = \text{clamp} \left(\Delta t_i, \Delta t_{min}, \Delta t_{max} \right)$$

end

Warte bis alle Partikel fertig

$\Delta t = \min(\Delta t_i)$

3.5 Strategien zur Beschleunigung

Sowohl die Berechnung der Gravitationskräfte als auch die Implementierung jeder einzelnen *SPH*-Formel hat einen Aufwand von $O(N^2)$ (N ist die Anzahl der Partikel). Die Nachbarn eines Partikels für jeden Teil der SPH Berechnung sind die selben. Eine erste Überlegung zur Beschleunigung wäre, während des Berechnens der Gravitationskräfte eine Liste von Nachbarn zu erstellen, die anschließend von der *SPH* Berechnung verwendet wird.

3.5.1 GPU Spezifische Techniken

Bei *compute shadern* sind Threads in sogenannten *work groups* gruppiert. Threads in einer *work group* werden auf der *GPU* dem selben Prozessor zugeteilt. Bis zu 32 Threads bearbeitet ein Prozessor gleichzeitig (auf Nvidia *GPUs*). Die Größe der *work groups* kann einen großen Einfluss auf die Performance der Anwendung haben. Welche Werte für einen bestimmten Shader die besten Ergebnisse erzielen, unterscheidet sich von *GPU* zu *GPU*. Die Größe der *work groups* ist daher ein veränderbarer Eingabeparameter. Auf Nvidia *GPUs* sollte sie ein Vielfaches von 32 sein.

Wird Code ausgeführt, in dem Teile der Threads innerhalb einer *work groups* verschiedene Pfade nehmen (z.B. eine if-Abfrage), so werden alle Pfade nacheinander berechnet. Dabei werden die Threads abgeschaltet, welche einen anderen Pfad nehmen. Verzweigungen sollten daher möglichst vermieden oder so kurz wie möglich gehalten werden. Besonders in den Programmteilen, die N^2 mal ausgeführt werden.

Das Laden von Daten aus dem globalen Speicher der Grafikkarte dauert einige Zeit. In dieser Zeit kann der Thread nicht weiter bearbeitet werden. Im schlimmsten Fall wartet der Prozessor anstatt etwas zu berechnen. Speicherzugriffe aus mehreren Threads innerhalb einer *work group* können zusammengefasst werden, sofern die angeforderten Daten im Speicher hintereinander liegen. Die Partikeleigenschaften werden daher als *float*, *vec2* oder *vec4* gruppiert. Der Typ *vec3* ist aufgrund des *Alignment* weniger geeignet. Außerdem sollte ein Array von Strukturen vermieden werden, wenn auf einzelne Variablen innerhalb der Struktur zugegriffen wird. Die Daten liegen dann nicht mehr hintereinander im Speicher (siehe Abbildung 7). Für mehr Informationen wird auf den *CUDA Programming Guide* [Nvi17] verwiesen.

Eine weitere Möglichkeit den Code zu beschleunigen ist es, Daten im *shared memory* zwischenzuspeichern. Jeder Prozessor der *GPU* hat einen eigenen Speicherbereich den sich alle Threads innerhalb einer *work group* teilen. Speicherzugriffe in den *shared memory* sind wesentlich schneller als die in den globalen Speicher. In Programmteilen, in denen jeder Partikel mit jedem anderen Partikel interagiert, wird dies genutzt um die Anzahl der Zugriffe in den globalen Speicher zu reduzieren [NHP07].

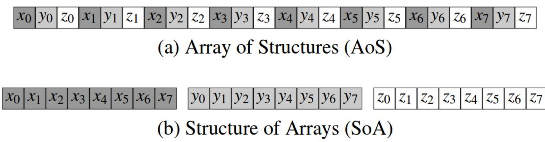


Abbildung 7: Verschiedene Speicherlayouts für sieben Gruppen mit jeweils drei Elementen. Grafik aus [LHW12].

Jeder Thread ist für die Aktualisierung der Eigenschaften eines einzelnen Partikels verantwortlich. Diese werden zunächst im lokalen Speicher des Threads abgelegt. Dann lädt jeder Thread innerhalb einer *work group* einen unterschiedlichen Partikel vom globalen Speicher in den *shared memory*. Bei einer Größe der *work groups* von 32 befinden sich die Partikel 0 bis 31 im *shared memory*. Jeder Thread berechnet dann die Interaktionen von seinem eigenen Partikel mit allen 32 Partikeln im *shared memory*. Danach werden die Daten im *shared memory* mit Partikel 32 bis 63 überschrieben und die Interaktionen werden erneut berechnet. Das wiederholt sich so lange bis die *work group* mit allen N Partikeln interagiert hat. Jeder Thread legt die neuen Eigenschaften seines Partikels zum Schluss wieder im globalen Speicher ab. Es ist hilfreich wenn N ohne Rest durch die Größe der *work groups* teilbar ist. Ansonsten wird eine gesonderte Behandlung der letzten Iteration notwendig. Abbildung 8 stellt das Verfahren grafisch da.

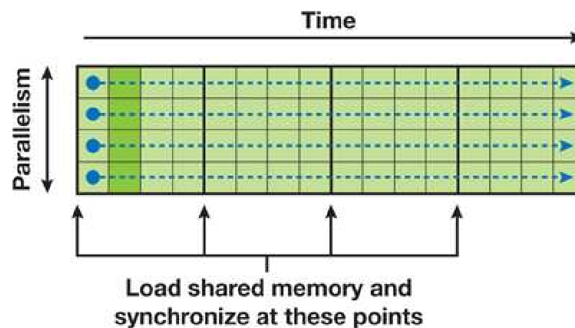


Abbildung 8: Threads einer *work group* berechnen Interaktionen zwischen Partikeln. Jedes Kästchen steht für eine Interaktion, jede Zeile ist ein Thread. [NHP07]

Moderne *GPUs* haben eine Vielzahl von Prozessorkernen, die gleichzeitig jeweils eine *work group* ausführen. Um die Rechenleistung voll auszunutzen, muss allen Kernen eine *work group* mit mindestens 32 Threads (auf Nvidia *GPUs*) zugewiesen werden. Es kann sogar sinnvoll sein, noch mehr *work groups* zu starten. Prozessoren können dann zwischen mehreren Threads wechseln, während sie auf die Ausführung von Speicherzugriffen warten. Außerdem läuft die Anwendung auf zukünftigen *GPUs* mit noch mehr Prozessorkernen schneller. Um auch bei geringer Partikelzahl genug

Threads bereit zu stellen, damit die gesamte *GPU* arbeiten kann, wird die Handhabung eines Partikels auf mehrere Threads aufgeteilt. Ein Thread lässt dann Partikel i mit den Partikeln 0 bis $N/2-1$ interagieren, während ein anderer Thread (in einer anderen *work group*) die Interaktionen von i mit $N/2$ bis N übernimmt. Sind beide Threads beendet, erfolgt eine globale Synchronisation und ein weiterer Thread wird gestartet um die beiden Ergebnisse zu summieren. Der zusätzliche Aufwand, den dies verursacht, beeinträchtigt eine Simulation mit großer Partikelanzahl kaum.

3.5.2 Datenstrukturen

Bei steigenden Werten für N stoßen die Verfahren aus dem vorherigen Abschnitt an ihre Grenzen. Die Verwendung von Datenstrukturen kann den Aufwand der Berechnungen von $O(N^2)$ auf $O(N \log N)$ oder sogar $O(N)$ reduzieren. In dem in dieser Arbeit vorgestellten Simulationscode wurde keine Datenstruktur implementiert. Trotzdem werden in diesem Kapitel verschiedene mögliche Datenstrukturen, sowie ihre Vor- und Nachteile im Hinblick auf eine kompressible *SPH* Simulation auf der Grafikkarte diskutiert.

In der Computergrafik werden *SPH* Partikel oft in ein gleichmäßiges Raster (*uniform grid*) einsortiert. Voxel in diesem Raster haben die Kantenlänge h. Bei der Nachbarschaftssuche müssen dann nur Partikel in angrenzenden Voxeln betrachtet werden. Sind die Partikel gleichmäßig im Raum verteilt führt dieses Vorgehen zu einem Aufwand von $O(N)$, wobei das Aufbauen und Aktualisieren des Rasters zusätzliche Zeit kostet [IOS+14]. In der Computergrafik werden meist inkompressible Fluide simuliert, sodass die Partikel tatsächlich annähernd gleichmäßig im Volumen verteilt sind. Bei einer zusammenfallenden Gaswolke hingegen befindet sich schon nach kurzer Zeit ein Großteil der Partikel in einem sehr kleinen Teil des ursprünglichen Volumen. Im Extremfall, bei dem sich alle Partikel im selben Voxel des Rasters befinden, ist der Aufwand der Nachbarschaftssuche dann wieder $O(N^2)$. Zusätzlich verbrauchen viele leere Voxel unnötig Speicher. Variable Kernelradien sind ein zusätzliches Problem.

Beim Simulieren von kompressiblen Fluiden unter dem Einfluss von Gravitation wird in der Astrophysik oft eine Baumstruktur verwendet. Das Verfahren nennt sich *TREESPH* und ermöglicht das Beschleunigen von Nachbarschaftssuche und Gravitationsberechnung mit einer einzigen Baumstruktur [HK89]. Ein Partikel ist ein Blatt des Baumes und Knoten werden nur für Bereiche gespeichert, in denen sich tatsächlich Partikel befinden. Für jeden Knoten des Baumes wird die Gesamtmasse, der Massenschwerpunkt, der räumliche Mittelpunkt und die Größe rekursiv berechnet und gespeichert.

Die Berechnung der Gravitationskraft für einen bestimmten Partikel beginnt mit dem obersten Knoten. Der Abstand zwischen Partikel und Knoten wird mit der Größe des Knoten verglichen. Ist er weit genug entfernt und klein, wird seine Gesamtmasse und sein Massenschwerpunkt zum Berechnen der Gravitationskraft verwendet. Ist der Knoten zu nah für seine Größe wird er geöffnet und die Kindknoten werden in gleicher Weise behandelt. Ein Toleranzparameter steuert, ab wann Knoten geöffnet werden und gewichtet Geschwindigkeit gegen Genauigkeit.

Zum Finden der Nachbarn für die SPH Berechnungen wird der Baum wieder traversiert. Der Kernelradius h_i des Partikels stellt dabei eine Kugel um seine Position $\boldsymbol{r}_i$ dar. Eine weitere Kugel entsteht aus dem Mittelpunkt und der Größe eines Knotens. Überschneiden sich diese beiden Kugeln, wird der Knoten geöffnet. Mit den Kindern wird genauso verfahren, bis alle Partikel in einem Radius von hi um den Punkt $\boldsymbol{r}_i$ gefunden sind. Das ganze hat Ähnlichkeit zu einer in der Computergrafik genutzten *bounding volume hierarchy*.

Ist die Nachbarschaftssuche auch vom Kernelradius des anderen Partikels abhängig, wird für jeden Knoten zusätzlich der längste darin vorkommende Kernelradius gespeichert. Das Kriterium zum Öffnen wird dann so angepasst, dass alle für die Rechnung benötigten Partikel gefunden werden [Spr05].

Sowohl das Traversieren als auch das Generieren von Baumstrukturen ist auf der GPU kompliziert. Rekursive Funktionsaufrufe sind nicht möglich und Verzweigungen sehr teuer. Auch atomare Speicherzugriffe und das Synchronisieren von parallel laufenden Threads kostet wertvolle Rechenzeit. Eine vollständige GPU-Implementation einer Baustruktur wird von Bédorf et al. [BGP12] gezeigt. Sie lohnt sich aber laut den Autoren erst ab Simulationen mit etwa 10^5 Partikeln.

Hegeman et al. [HCM06] hingegen definieren ihre Baumstruktur so, dass ein veralteter Baum zwar langsamer ist, aber weiterhin korrekte Ergebnisse liefert. Der Baum wird dann auf der *CPU* erzeugt und aktualisiert, während die GPU immer weiter simuliert. Sobald ein neuer Baum verfügbar ist, wird er an die GPU weiter gereicht und die CPU beginnt wieder den Baum mit neuen Partikelpositionen zu aktualisieren. Dieser Ansatz klingt besonders interessant, da die Rechenleistung von beiden Prozessoren voll ausgenutzt wird und ihre individuellen Stärken berücksichtigt werden.

4 Implementierung

4.1 Aufbau der Software

Der Großteil der Simulation erfolgt in einer Reihe von *compute shadern* auf der *GPU*. Die *CPU* verwaltet dabei die Einstellungen und steuert die Simulation auf der Grafikkarte. Sie übernimmt außerdem die Kommunikation mit dem Benutzer.

Der *CPU*-Code nutzt dabei drei Klassen. Der `ParticleBuffer` repräsentiert das Partikelsystems auf der *GPU*. Die Klasse `ParticleSpawner` übernimmt das Generieren von Partikeln vor der Simulation, `ParticleRenderer` ist für die Visualisierung zuständig. Das Starten der *compute shader* für die Simulation geschieht direkt aus der `main()`-Funktion.

Um den Umgang mit *OpenGL* zu vereinfachen, wurden einige Wrapper-Klassen und Hilfsfunktionen verwendet. Dieses grundlegende Framework ist in Zusammenarbeit mit Johannes Braun entstanden. Die Quelldateien sind entsprechend gekennzeichnet.

4.2 Handhabung von Partikeln

Um in *compute shadern* auf Daten zugreifen zu können, müssen sie in sogenannten *Shader Storage Buffer Objects* (*SSBO*) gespeichert werden. Diese liegen im globalen Speicher der *GPU* und können über *OpenGL* initialisiert und konfiguriert werden. Alle Partikeleigenschaften sowie Zwischenergebnisse der Simulation, die über einen globalen Synchronisierungspunkt hinweg gespeichert werden sollen, müssen in so einem *SSBO* abgelegt werden.

SSBOs erlauben das Anlegen von Strukturen ähnlich wie in der Programmiersprache C. Theoretisch wäre es möglich alle Eigenschaften eines Partikels in einer solchen Struktur zu bündeln und dann in einem *SSBO* einen Array von N solcher Strukturen abzuspeichern. Wie in Kapitel 3.5.1 angesprochen nimmt dieses Speicherlayout der *GPU* die Möglichkeit, mehrere Speicherzugriffe innerhalb einer *work group* zusammenzufassen. Dies ist besonders wichtig, da nicht immer alle Partikeleigenschaften auf einmal gelesen oder geschrieben werden müssen.

Besser ist es, wenn die Positionen aller Partikel hintereinander im Speicher liegen, dann alle Geschwindigkeiten und so weiter. Um dies zu erreichen, erhält jede Partikeleigenschaft ein eigenes *SSBO*. Eigenschaften, die oft zusammen in den Berechnungen auftauchen, werden zu **`vec2`** oder **`vec4`** gruppiert. Der Simulationscode verwendet folgende *SSBOs*:

positionBuffer Typ: **vec4**, die ersten drei Komponenten enthalten die aktuelle Position eines Partikels, in der vierten Komponente steht seine Masse.

velocityBuffer Typ: **vec4**, die ersten drei Komponenten enthalten die aktuelle Geschwindigkeit eines Partikels als Vektor, in der vierten Komponente wird die Schallgeschwindigkeit für die Bestimmung der Viskosität gespeichert.

accelerationBuffer Typ: **vec4**, die ersten drei Komponenten enthalten die aktuelle Beschleunigung eines Partikels als Vektor, die vierte Komponente speichert $\max_j \left(v_{ij}^{sig} \right)$, was für die Berechnung des Zeitschrittes benötigt wird (siehe Abschnitt 3.3.3.

hydrodynamicsBuffer Typ: **vec4**, die erste Komponente enthält die Dichte des Gases ρ an der Position dieses Partikels, die zweite Komponente enthält den Druck p, in der dritten Komponente ist der Korrekturfaktor für Viskosität bei Scherströmung f^{visc} gespeichert (siehe Abschnitt 2.7), die vierte Komponente enthält den Korrekturfaktor zur Druckberechnung bei veränderlichem Kernelradius f^{pres} (siehe Abschnitt 2.8).

smlengthBuffer Typ: **float**, enthält den Kernelradius h eines Partikels.

timestepBuffer Typ: **float**, enthält den Zeitschritt, der anhand der Zeitschritt-Kriterien für diesen Partikel berechnet wurde.

Jeder dieser Buffer enthält N Elemente und die Eigenschaften eines Partikels i liegen in `positionBuffer[i]` bis `timestepBuffer[i]`. Eine Ausnahme bilden der **acceleration**- und **hydrodynamicsBuffer**. Hier beträgt die Anzahl der Elemente ein Vielfaches von N, wenn die Berechnung von Beschleunigung oder Dichte auf mehrere Threads pro Partikel aufgeteilt wird (wie in Kapitel 3.5.1 beschrieben). Alle Zwischenergebnisse für alle Partikel werden dann gesammelt und später addiert.

Damit die Verwaltung von so vielen einzelnen *SSBOs* auf der *CPU*-Seite nicht aufwendig und unübersichtlich wird, wird sie von der Klasse `PartikelBuffer` übernommen. Sie speichert die Buffer-ID für alle benötigten Buffer, sowie die Anzahl der enthaltenen Partikel. Ihre Methoden ermöglichen ein einfaches Erstellen und Binden der *SSBOs* und verwalten Zugriffe von der *CPU* Seite. Soll eine andere Klasse oder Funktion Operationen auf den Partikeln ausführen, kann ein Objekt von `PartikelBuffer` übergeben werden. Der *CPU*-code bleibt dadurch genauso aufgeräumt, als wenn nur ein einzelnes *SSBO* verwendet werden würde.

In derselben Datei gibt es eine Liste von Konstanten für alle *SSBO*-Bindungspunkte. Die Shader-Datei `common.glsl` enthält eine Kopie dieser Liste, sodass alle *SSBOs* immer an der richtigen Stelle gebunden werden.

4.3 Generierung von Partikeln

Das Generieren der Partikel erfolgt in der Klasse `ParticleSpawner`. Zunächst muss ein Objekt von `ParticleBuffer` übergeben werden. Dann stehen verschiedene Methoden zur Verfügung, um die Partikel zu erzeugen. Sie können gleichmäßig in einer Kugel verteilt werden, aber auch das Definieren mehrerer Bereiche unterschiedlicher Dichte ist möglich. Dabei kann die Gesamtmasse des Systems sowie der anfängliche Kernelradius angegeben werden. Die Generierung erfolgt durch *compute shader* direkt auf der Grafikkarte. Der `ParticleSpawner` verwaltet lediglich gewählte Einstellungen und die Ausführung der Shader.

Ist die Erzeugung der Partikel abgeschlossen, kann eine zweite Gruppe von Methoden verwendet werden, um eine Anfangsgeschwindigkeit für die Partikel festzulegen. Die Ausführung mehrerer Methoden nacheinander führt zur Addition der Geschwindigkeitswerte. Implementiert ist (wie in Kapitel 3.3.2 beschrieben) eine Drehgeschwindigkeit um eine beliebige Achse sowie *curl-noise*, bei dem eine beliebige Anzahl von Frequenzen mit unterschiedlicher Intensität eingestellt werden kann. Auch der Seed für den Zufallsgenerator kann manuell festgelegt werden. Es gibt außerdem die Möglichkeit *simplex-noise* direkt als Anfangsgeschwindigkeit zu verwenden.

Der `ParticleSpawner` ist hauptsächlich eine Sammlung von Methoden, die Shader aufrufen. Entsprechend kann er sehr einfach erweitert werden, um beliebige Anfangsbedingungen zu ermöglichen.

4.4 Simulation des Partikelsystems

Um den Algorithmus 3 und 4 auf der *GPU* zu implementieren, müssen an mehreren Stellen alle Threads synchronisiert werden. Innerhalb eines *compute shaders* gesetzte Synchronisationspunkte beziehen sich nur auf die eigene *work group*. Ein globales Synchronisieren aller Threads ist nicht möglich. Als workaround wird der Algorithmus in mehrere Shader aufgeteilt. Zwischen den einzelnen Aufrufen der `dispatch()`-Funktion – die *compute shader* startet – wird dann sichergestellt, dass alle Threads des vorherigen Aufrufs die Ausführung beendet haben. In der `main()`-Funktion werden die Shader compiliert und dann zu einem Lambda zusammen gefasst. Die Simulationseinstellungen werden über **uniform**-Variablen übergeben. Listing 1 zeigt die Lambda-Funktion `simulate()`. Sie ruft alle benötigten Shader auf um die Simulation einen Zeitschritt voranzubringen. Dabei wird die Größe der *work groups* `wgSize` und die Gesamtzahl der Partikel übergeben (möglich durch die Erweiterung `GL_ARB_compute_variable_group_size`). Für jeden Partikel wird dann eine Instanz des Shaders in einem eigenen Thread gestartet. Ist die Größe der *work group* bereits im Shader angegeben, wird stattdessen die Anzahl der zu startenden *work groups* berechnet und übergeben.

```
auto simulate = [densityShader,pressureShader,wgSize,
    hydroAccum,integrator,adjustH](){
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  adjustH.dispatch(NUM_PARTICLES,wgSize);
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  densityShader.dispatch(NUM_PARTICLES*
      DENSITY_THREADS_PER_PARTICLE/DENSITY_WGSIZE);
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  hydroAccum.dispatch(NUM_PARTICLES,wgSize);
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  pressureShader.dispatch(NUM_PARTICLES*
      ACCEL_THREADS_PER_PARTICLE/PRESSURE_WGSIZE);
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  integrator.dispatch(NUM_PARTICLES,wgSize);
};
```

Listing 1: Funktion zur Durchführung eines Simulationsschrittes.

Im Folgenden werden die Aufgaben der einzelnen Shader erklärt. Wie zu erwarten ist, orientiert sich die Aufteilung der Berechnungen an den in Algorithmus 3 und 4 markierten Synchronisationspunkten.

4.4.1 Einstellung des Kernelradius

Im Shader `adjustH` wird der Kernelradius jedes Partikels aktualisiert.

```
vec4 hydi = hydro[gl_GlobalInvocationID.x];
float hi = clamp(pow(3.0f*num_neighbours*mass_per_particle
    / (hydi.y*4.0f*PI),1.0f/3.0f),hmin,hmax);
smlength[gl_GlobalInvocationID.x] = hi;
```

Listing 2: Shader zur Anpassung des Kernelradius.

4.4.2 Berechnung der Dichte

Der `densityShader` berechnet die Dichte der Partikel. Außerdem löst er den von der Dichte unabhängigen Teil der Divergenz und des Curl-Operators.

Um den Vorgang zu beschleunigen wird wie in Kapitel 3.5.1 beschrieben vorgegangen. Die Partikel, mit denen interagiert wird, werden in *tiles* aufgeteilt. Ein *tile* hat immer die selbe Größe wie die *work group*. Sollten die Interaktionen eines Partikels auf mehrere Threads aufgeteilt werden, wird der Wert TILES_PER_THREAD vom Hauptprogramm entsprechend eingestellt und die Anzahl der gestarteten *work groups* erhöht. Jeder Thread muss also zunächst berechnen, für welchen Partikel er verantwortlich ist und bei welchem *tile* er mit der Berechnung der Interaktion beginnt.

Danach werden die Eigenschaften des Partikels geladen, der zu diesem Thread gehört. Außerdem werden einige Werte berechnet, die während der Interaktion mit allen anderen Partikeln konstant bleiben. Dazu zählt zum Beispiel der Vorfaktor des W_{poly6}-Kernels $\left(\frac{315}{64\pi h^9}\right)$ und das Quadrat des Kernelradius.

Alle *tiles* werden nacheinander in einer Schleife abgearbeitet. Jeder Thread in der *work group* lädt die benötigten Parameter von einem Partikel des aktuellen *tiles* in den *shared memory*. Nachdem die Threads in der *work group* synchronisiert wurden, beginnt eine Schleife, in der die Interaktionen des eigenen Partikels mit allen Partikeln des *tiles* behandelt wird. Eine erneute Synchronisation verhindert das vorzeitige Überschreiben des *shared memory* durch einen anderen Thread.

```
const uint idxi = gl_GlobalInvocationID.x % NUM_PARTICLES; // calculate index of this particle
const uint startTile = TILES_PER_THREAD * uint(gl_GlobalInvocationID.x / NUM_PARTICLES);

// load parameters and calculate constants
// ...

for(uint tile = 0; tile < TILES_PER_THREAD; tile++)
{
    // fill fields in shared memory
    uint idx = gl_WorkGroupSize.x * (startTile + tile) + gl_LocalInvocationID.x;
    pos[gl_LocalInvocationID.x] = positions[idx];
    vel[gl_LocalInvocationID.x] = velocities[idx].xyz;

    memoryBarrierShared();
    barrier();

    for(uint j=0; j<gl_WorkGroupSize.x; j++)
    {
        // calculate interaction
        // ...
    }

    memoryBarrierShared();
    barrier();
}
```

Listing 3: Aufbau des Dichte-Shaders.

Bei der Berechnung einer einzelnen Interaktion im Inneren der Schleife werden die im *shared memoy* gespeicherten Eigenschaften sowie die am Anfang berechneten Konstanten verwendet.

```
const vec4 posj = pos[j];
const vec3 velj = vel[j];
const vec3 rij = posi.xyz - posj.xyz;
const float r2 = dot(rij, rij);

// calculate all densities
density += posj.w * (Wpoly6(r2, hiPoly6Factor, hi2));

densityPlus += posj.w * (Wpoly6(r2, hipePoly6Factor, hipe2
    ));
densityMinus += posj.w * (Wpoly6(r2, himePoly6Factor,
    hime2));

// calculate vorticity and divergence of the velocity
vec3 gradi = Wpoly6Grad(rij,r2,hiPolyGrad6Factor,hi2);
vorticity += posj.w * cross(veli - velj, gradi);
divergence += dot(velj - veli,gradi);
```

Listing 4: Berechnung der Dichte.

Sind alle Interaktionen in allen *tiles* abgehandelt, werden die Ergebnisse im `hydrodynamicsBuffer` abgespeichert. Außerdem wird $\frac{\partial \rho}{\partial h}$ berechnet. Im Speicher liegen die Ergebnisse aus dem ersten Thread für alle Partikel hintereinander, dann die Ergebnisse aus dem zweiten Thread für alle Partikel und so weiter. Speicherzugriffe einer *work group* können dadurch weiterhin zusammengefasst werden.

```
float vort = length(vorticity);
hydro[gl_GlobalInvocationID.x] = vec4(vort, density,
    divergence, (densityPlus-densityMinus)/(2*eps));
```

4.4.3 Berechnung des Drucks

Der hydroAccum-Shader wird pro Partikel in einem Thread ausgeführt. Wurden im vorherigen Shader mehrere Threads pro Partikel verwendet, müssen zunächst die Teilergebnisse aufsummiert werden. Danach wird der Druck, die Schallgeschwindigkeit c und die Korrekturfaktoren f^{visc} und f^{pres} berechnet.

```
vec4 sum = vec4(0);
for(uint i=0; i < THREADS_PER_PARTICLE; i++)
{
    sum += hydro[NUM_PARTICLES * i + gl_GlobalInvocationID.
        x];
}
float hi = smlength[gl_GlobalInvocationID.x];

const float ac = (sum.y<frag_limit) ? ac1 : ac2;
const float pressure = k * pow(sum.y,ac);

float ci = sqrt(ac*pressure/sum.y);

float vorticity = (sum.x/sum.y);
float divergence = (sum.z/sum.y);
float fiv = divergence / (divergence + vorticity+0.001*ci/
    smlength[gl_GlobalInvocationID.x]);

float fip = 1.0/(1.0+ ((hi/(3*sum.y)) * sum.w));
hydro[gl_GlobalInvocationID.x] = vec4(pressure, sum.y, fiv,
     fip);
velocities[gl_GlobalInvocationID.x].w = ci;
```

Listing 5: Aufsummierung der Teilergebnisse.

4.4.4 Berechnung der Beschleunigung

Die Berechnung der Beschleunigung im pressureShader funktioniert analog zur Berechnung der Dichte in Abschnitt 4.4.2. Die Partikel werden ebenfalls in *tiles* aufgeteilt und die Berechnung erfolgt dann unter Nutzung des *shared memory*. Der Unterschied liegt in den benötigten Parametern und der Berechnungen, die während einer einzelnen Interaktion durchgeführt werden.

Listing 6 zeigt nur den Code der für jedes Partikelpaar ausgeführt wird. Hier wird die Beschleunigung berechnet, die aus Schwerkraft, Druckgradient und Viskosität resultiert und $\max_j \left(v_{ij}^{sig}\right)$ bestimmt.

```
const vec3 rij = posi.xyz - posj.xyz; // vector from i to j
const float r2 = dot(rij, rij);
const float dist = sqrt(r2); // distance from i to j

// gravity
accGravity.xyz += posj.w * -rij / sqrt(pow(r2+(hi*hj*
    eps_factor2),3));

// pressure
const float pod2j = hydroj.x / (hydroj.y*hydroj.y);
const vec3 gradi = WspikyGrad(rij,dist,hi,hiSpikyGradFactor
    );
const vec3 gradj = WspikyGrad(rij,dist,hj);

accPressure -= posj.w * (hydroi.w*pod2i* gradi + hydroj.w*
    pod2j* gradj);

// viscosity
const float vfij = (hydroi.z+hydroj.z)*0.5;

float wij = dot(veli.xyz - velj.xyz, rij)/dist;
wij = (wij <0) ? wij : 0;
const float vsig = veli.w+velj.w - 3.0*wij;
const float rhoij = (hydroj.y + hydroi.y)*0.5;
const float II = vfij * -alpha*0.5 * vsig * wij / rhoij;

maxVsig = max(maxVsig,vsig);
accPressure -= posj.w * II * (gradi+gradj)*0.5f;
```

Listing 6: Berechnung der Beschleunigung.

4.4.5 Integration

Der Shader `integrator` summiert analog zu 4.4.3 die einzelnen Teilergebnisse des `pressureShader` auf. Danach wird aus der Beschleunigung die neue Geschwindigkeit und Position mithilfe des *Leapfrog* Algorithmus berechnet (Kapitel 3.3.3). Außerdem wird die maximale Länge des Zeitschritts bestimmt, der für den Partikel im nächsten Simulationsschritt verwendet werden kann.

Der **timestepBuffer** wird anschließend auf die *CPU* heruntergeladen. Das Minimum wird bestimmt und als Zeitschritt für den nächsten Simulationsschritt eingestellt.

4.5 Visualisierung des Ergebnisses

Nach dem Abschluss eines Simulationsschrittes werden die Partikel an ihren neuen Positionen gerendert. Die Bildwiederholrate ist hauptsächlich von der Dauer der Berechnung eines Simulationsschrittes abhängig. Ändert sich der Zeitschritt, scheint es als verlangsame oder Beschleunige sich die Bewegung der Partikel. Um zu verhindern, dass die Simulation am Anfang zu schnell läuft und die Ergebnisse nicht richtig sichtbar werden, kann der maximal erlaubte Zeitschritt kleiner eingestellt werden.

Für das Visualisieren ist die Klasse ParticleRenderer zuständig. Die Partikel werden durch runde *Sprites* mit weichem Rand dargestellt. Die OpenGL Einstellungen

```
1 glBlendFunc(GL_ONE, GL_ONE);
2 glEnable(GL_BLEND);
3 glDisable(GL_DEPTH_TEST);
```

Listing 7: OpenGL-Einstellungen zum Rendern

sorgen dafür, dass die Farbwerte von Partikeln, die hintereinander liegen, aufsummiert werden. Regionen mit höherer Dichte lassen sich so durch hellere Farben identifizieren. Partikelgröße, Helligkeit und Farbe können vom Benutzer eingestellt werden. So kann beispielsweise die Partikelgröße reduziert werden, um Details in dichten Regionen sichtbar zu machen (Abbildung 9).

Gerendert wird mit der Einstellung GL_POINTS, wobei der positionBuffer direkt als Datenquelle verwendet wird. Das Einschalten von GL_PROGRAM_POINT_SIZE ermöglicht das Festlegen der Punktgröße im Vertexshader. Dieser wird für jeden Partikel ein Mal aufgerufen. Die oben beschriebenen Einstellungen erhält er zusammen mit der Projektions- und Model-View-Matrix der virtuellen Kamera als *uniform-variable*. Außerdem wird die Größe des Viewports übergeben. Um die Größe der einzelnen Punkte entsprechend ihrer Entfernung von der Kamera festzulegen, muss die gesamte perspektivische Projektion manuell durchgeführt werden.

```
1 void main()
2 {
3     gl_Position = model_view_projection * input_position;
4     gl_PointSize = viewport_size.y * projection[1][1] *
          particle_size / gl_Position.w;
5     center = (0.5 * gl_Position.xy/gl_Position.w + 0.5) *
          viewport_size;
6     radius = gl_PointSize / 2;
7 }
```

Listing 8: Der Vertexshader.

center und radius sind Ausgabevariablen, die an den Fragmentshader weiter gegeben werden.

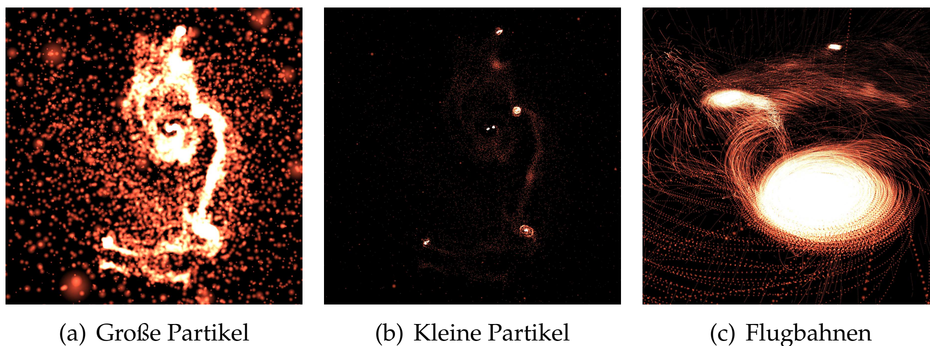
Für jedes *Fragment*, aus dem die gerenderten Partikel bestehen, wird nun ein Fragmentshader gestartet. Die Punkte sind zunächst quadratisch. Um sie abzurunden, werden alle Fragmente verworfen die einen zu großen Abstand zum Mittelpunkt haben. Dann wird – wieder basierend auf dem Abstand zum Mittelpunkt – die Farbe des *Fragments* abgedunkelt. Nach der Addition von überlappenden Farbwerten erscheint der Partikel dadurch zum Rand hin transparent. Listing 9 zeigt die Implementierung im Fragmentshader.

```
1 void main()
2 {
3     vec2 coord = (gl_FragCoord.xy - center) / radius;
4     float distFromCenter = length(coord);
5     if(distFromCenter > 1.0)
6         discard;
7     vec4 falloffColor = color * (1 - distFromCenter);
8     fragment_color = falloffColor * brightness;
9 }
```

Listing 9: Der Fragmentshader.

Das Ergebnis ist eine sehr simple Darstellung der Simulationsdaten, die wenig Rechenzeit in Anspruch nimmt. Sie ist optisch ansprechend und ermöglicht das Erkennen von Strukturen wie Sternen und der Bewegung des Gases. Allerdings ermöglicht sie keine direkten Rückschlüsse auf die tatsächliche Dichte oder eine der anderen simulierten Eigenschaften.

Um die Flugbahnen der einzelnen Partikel besser nachverfolgen zu können, wird das Leeren des Bildpuffers beim Drücken und Halten von Taste 3 ausgesetzt. Die Partikel erscheinen dann als Linien, die sich immer weiter verlängern (Abbildung 9(c)). Beim Loslassen der Taste wird der Bildpuffer wieder vor jedem Rendervorgang geleert und Partikel erscheinen wie zuvor als Punkte.



(a) Große Partikel (b) Kleine Partikel (c) Flugbahnen

Abbildung 9: Mit großen Partikeln *(a)* lassen sich Regionen höher und niedriger Dichte grob voneinander unterscheiden. Kleiner dargestellte Partikel *(b)* machen einzelne Sterne sogar in Binärsystemen sichtbar. In *(c)* werden die Flugbahnen der einzelnen Partikel visualisiert.

4.6 Benutzereingabe

Eine grafische Benutzeroberfläche existiert nicht. Alle Parameter und Optionen können in der Datei `Settings.h` eingestellt werden. Nur zum Verändern der Anfangsbedingungen muss die `main()`-Funktion oder die Klasse `ParticleSpawner` direkt bearbeitet werden. Die Kamera wird mit *W, A, S, D, Q* und *E* gesteuert. Ein Klick mit der rechten Maustaste fängt den Mauszeiger im Fenster und ermöglicht das Einstellen der Blickrichtung mit der Maus. Mit + und - kann die Geschwindigkeit der Kamerabewegung kontrolliert werden. Taste *1* startet und *2* pausiert die Simulation. Taste *3* visualisiert die Flugbahnen der einzelnen Partikel. *P* schreibt Debug-Informationen zum aktuellen Simulationsschritt in die Konsole. Mit +,-,*8* und *9* auf dem Nummernblock lassen sich Partikelgröße und Helligkeit einstellen.

Die Eingabebehandlung erfolgt mittels *glfw*, durch *Polling* (zyklisches Abfragen) in der Hauptschleife der `main()`-Funktion.

5 Evaluation

In diesem Kapitel werden einige Simulationen mit dem bisher beschriebenen Simulationsprogramm durchgeführt und die Ergebnisse diskutiert. Untersucht wird dabei die Dauer der Simulation auf verschiedener Hardware. Außerdem werden die Ergebnisse der Simulation optisch auf die in Kapitel 2.1 beschriebenen Merkmale untersucht und mit anderen Computersimulationen verglichen.

5.1 Performance

Um die Performance des Simulationscodes bewerten zu können, wurden eine Reihe von Simulationen mit verschiedenen Parametern auf Systemen mit unterschiedlichen Grafikkarten durchgeführt. Gemessen wurde die durchschnittliche Dauer eines Simulationsschrittes inklusive Visualisierung innerhalb der ersten $3,5$ Zeiteinheiten der Simulation. Zwischen den Messungen variiert wurde die Partikelanzahl, die Größe der *work groups* und die Anzahl der Threads pro Partikel. Die gewünschte Anzahl der SPH-Nachbarn ist auf 50 eingestellt. Um Abweichungen – zum Beispiel aus der Nutzung der Hardware durch das Betriebssystem – zu minimieren wurde jeder Test mehrmals durchgeführt und der Mittelwert gebildet. Als Hardware diente ein Workstation PC mit einer *Nvidia gtx 1080 GPU*, ein Workstation PC mit einer *Nvidia gtx 980 GPU* und ein Notebook mit einer *Nvidia gtx 950M GPU*. Das Betriebssystem war bei allen Tests *Ubuntu 16.04*.

Tabelle 1 bis 3 zeigen die Dauer eines Simulationsschrittes in Millisekunden für verschiedene Partikelzahlen auf den unterschiedlichen Grafikkarten. Jeweils mit den schnellsten Einstellungen für die übrigen Parameter. Dabei ist *wgs* die Größe der *work groups*, *tppd* die Anzahl der Threads pro Partikel bei der Dichteberechnung und tppb die Anzahl der Threads pro Partikel bei der Berechnung der Beschleunigung. Die letzte Spalte gibt die langsamste gemessene Zeit (auch in Millisekunden) bei nicht optimaler Parameterwahl an (es wurden nur durch 32 teilbare *work group* Größen verwendet). Der Anteil der Visualisierung an der Gesamtzeit beträgt selbst bei den größeren Partikelzahlen nur wenige Millisekunden.

N	t_{min} (ms)	wgs	$tppd$	$tppb$	t_{max} (ms)
4096	$1,1$	32	64	32	$3,0$
8192	$3,0$	32	32	32	$5,6$
16384	$9,2$	64	32	32	$12,0$
32768	$32,2$	128	64	16	$42,7$
65536	$121,2$	128	32	32	$144,7$
131072	$472,9$	256	32	16	$508,8$
262144	$1885,7$	256	16	16	$1938,6$
524288	$7453,6$	512	16	16	*

Tabelle 1: Dauer eines Simulationsschrittes bei verschiedenen Partikelzahlen unter Verwendung einer *Nvidia gtx 1080*.
** aus Zeitgründen wurde für* 524288 *Partikel nur eine einzelne Simulation mit den angegebenen Parameter durchgeführt.*

N	t_{min} (ms)	wgs	$tppd$	$tppb$	t_{max} (ms)
4096	$1,7$	128	4	4	$3,3$
8192	$4,6$	256	32	32	$5,8$
16384	$16,2$	512	32	32	$20,0$
32768	$59,7$	512	64	64	$78,9$
65536	$231,7$	512	128	128	$268,8$
131072	$951,8$	512	64	16	$1005,4$
262144	$3801,1$	512	32	32	*

Tabelle 2: Dauer eines Simulationsschrittes bei verschiedenen Partikelzahlen unter Verwendung einer *Nvidia gtx 980*.
** aus Zeitgründen wurde für* 262144 *Partikel nur eine einzelne Simulation mit den angegebenen Parameter durchgeführt.*

N	t_{min} (ms)	wgs	$tppd$	$tppb$	t_{max} (ms)
4096	$6,1$	16	16	32	$7,9$
8192	$19,1$	32	16	256	$23,7$
16384	$64,8$	32	32	512	$72,1$
32768	$243,9$	16	8	512	$258,5$
65536	$954,4$	16	8	512	$979,3$

Tabelle 3: Dauer eines Simulationsschrittes bei verschiedenen Partikelzahlen unter Verwendung einer *Nvidia gtx 950M*.

Es wird direkt ersichtlich, dass für jede Grafikkarte und Partikelzahl andere Einstellungen optimale Ergebnisse liefern. Im schlimmsten Fall dauert die Simulation bei ungünstig gewählten Parametern fast drei mal länger. Werden die Parameter korrekt eingestellt, erlaubt die *gtx 1080* eine interaktive Simulation mit bis zu 65536 Partikeln. Bei 131072 Partikeln dauert ein Simulationsschritt dann bereits so lange, dass es schwer ist die Evolution der Gaswolke als flüssige Bewegung wahrzunehmen. Auf dem Notebook mit der *gtx 950M* können immerhin bis zu 16384 Partikel interaktiv simuliert werden. Grafik 10 zeigt die Ergebnisse als Diagramm. Hier ist deutlich zu erkennen, dass die Dauer eines Zeitschrittes quadratisch von der Partikelanzahl abhängt. Die *Nvidia gtx 980* braucht bei gleicher Partikelzahl fast doppelt so lange wie die neuere *gtx 1080*. Der Vorteil wird bei sinkender Partikelzahl kleiner.

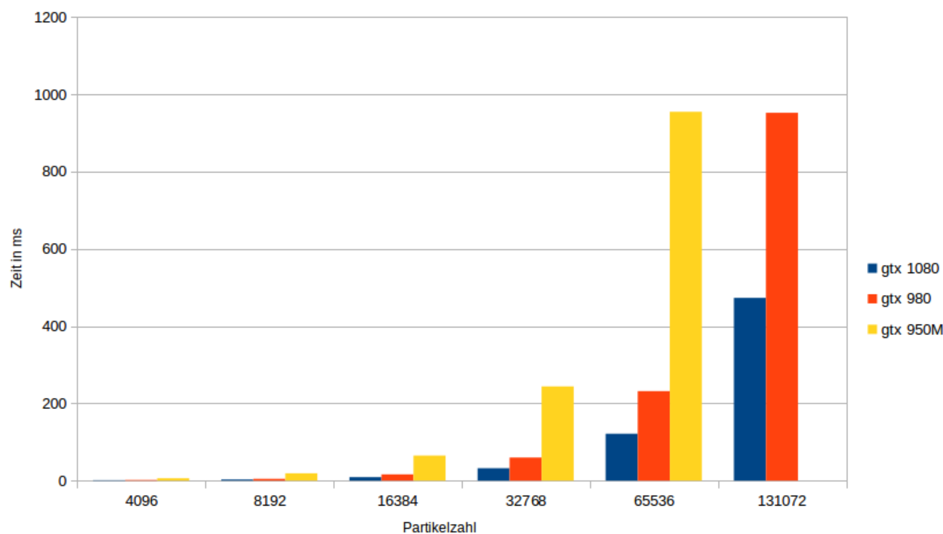


Abbildung 10: Dauer eines Simulationsschrittes bei verschiedenen Partikelzahlen auf allen getesteten Grafikkarten.

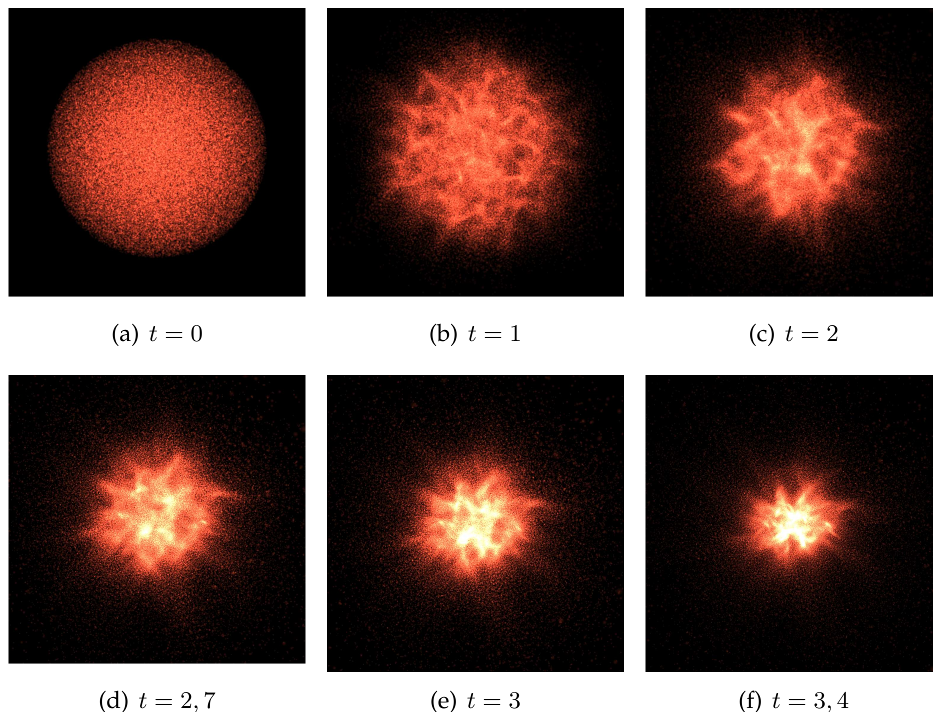
5.1.1 Parametertuning

Bei kleinen Partikelzahlen können nicht genug Threads gebildet werden, um alle Prozessoren der Grafikkarten komplett auszulasten. Kleinere *work groups* liefern hier bessere Ergebnisse. So wird die Arbeit auf möglichst viele Prozessorkerne aufgeteilt. Bei mehr Partikeln lohnt es sich dann die Größe der *work groups* zu erhöhen, was die Anzahl langsamer Zugriffe in den globalen Grafikspeicher reduziert. Wo diese Grenzen liegen, ist abhängig von der Anzahl der Prozessorkerne und der Geschwindigkeit des Grafikspeichers der genutzten *GPU*. Nach oben hin wird die Größe der *work groups* vom verfügbaren *shared memory* begrenzt.

Das Verwenden von unterschiedlich vielen Threads pro Partikel ist eine weitere Möglichkeit, die Berechnung auf mehr Prozessorkerne aufzuteilen, was je nach Hardware bei den meisten Partikelzahlen sinnvoll ist. Ab einer bestimmten Threadanzahl dauert das nachträgliche Aufsummieren der einzelnen Werte – was hier iterativ implementiert ist – so lange, dass ein weiteres Erhöhen keine Vorteile mehr bringt.

5.2 Ergebnisse

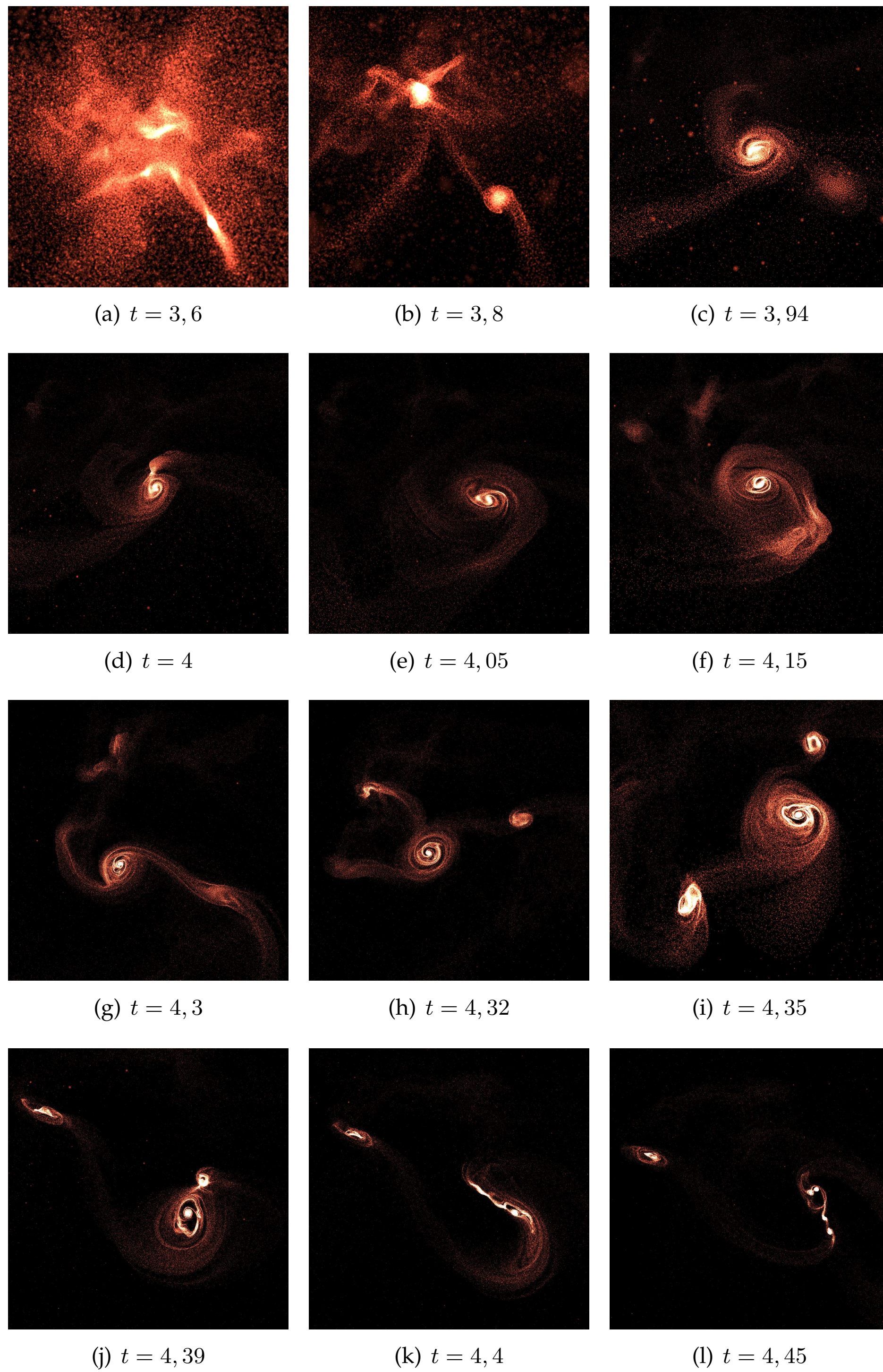
Um die Qualität der Simulationsergebnisse zu bewerten, werden im Folgenden die Ergebnisse einer Simulation mit 262144 Partikeln präsentiert. Die Gesamtmasse beträgt 20 Gewichtseinheiten, der Radius der Wolke ist 6 Längeneinheiten. Sie erhält außerdem eine anfängliche Rotationsgeschwindigkeit und ein turbulentes Geschwindigkeitsfeld (wie in Kapitel 3.3.2 beschrieben). Der Parameter A für die *Entropie* (Kapitel 2.3) ist auf $0,08$ gesetzt, während $\rho_{thres} = 80$, $\gamma_{low} = 1$ und $\gamma_{high} = 7/5$ (Kapitel 3.3.1).



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

(d) $t = 2,7$ (e) $t = 3$ (f) $t = 3,4$

Abbildung 11: Die Abbildungen zeigen die Evolution der Gaswolke zu Beginn der Simulation. Nach etwa $3,4$ Zeiteinheiten hat sich das Gas weit genug zusammengezogen, um einen dichten Kern zu bilden.



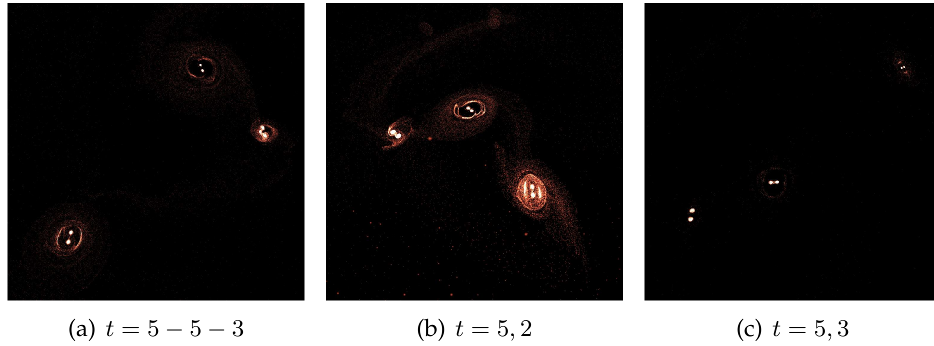
(a) $t = 3,6$ (b) $t = 3,8$ (c) $t = 3,94$

(d) $t = 4$ (e) $t = 4,05$ (f) $t = 4,15$

(g) $t = 4,3$ (h) $t = 4,32$ (i) $t = 4,35$

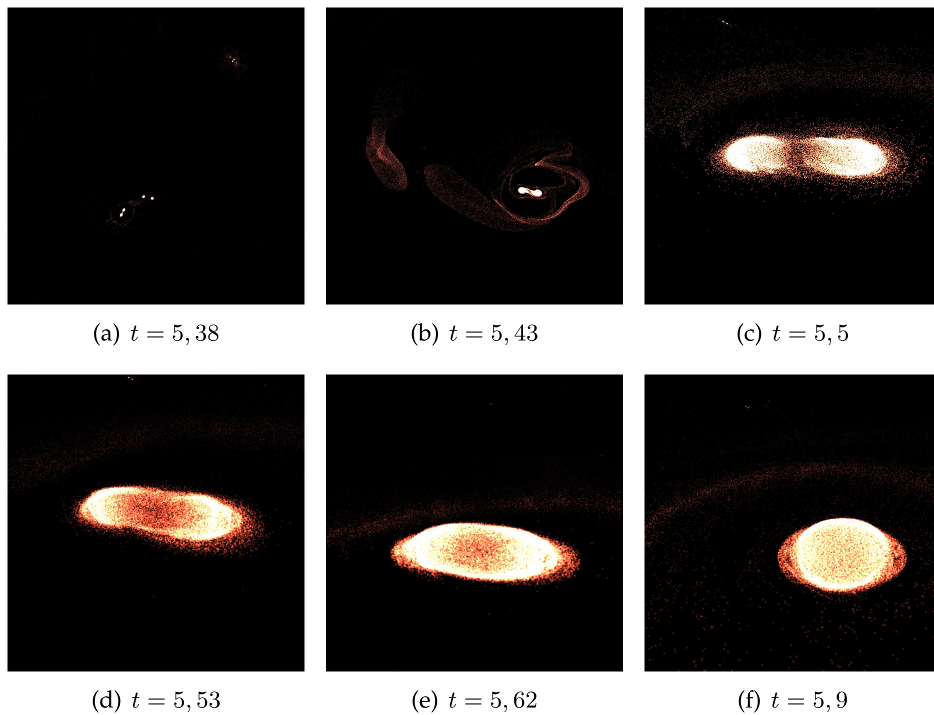
(j) $t = 4,39$ (k) $t = 4,4$ (l) $t = 4,45$

Abbildung 12: Im dichten Kern, der sich in der Gaswolke gebildet hat, beginnen Sterne zu entstehen. Mehrere Fragmentierungen und beinahe-Kollisionen sind erkennbar. Die Größe der Partikel wurde zwischen den Bildern variiert, um Details besser sichtbar zu machen.



(a) $t = 5 - 5 - 3$ (b) $t = 5,2$ (c) $t = 5,3$

Abbildung 13: Eine beinahe-Kollision zwischen drei Sternensystemen. Die Wechselwirkung zwischen den Sternen schleudert eines der Systeme davon.



(a) $t = 5,38$ (b) $t = 5,43$ (c) $t = 5,5$

(d) $t = 5,53$ (e) $t = 5,62$ (f) $t = 5,9$

Abbildung 14: Die anderen beiden Systeme kollidieren. Der Doppelstern im Zentrum der Wolke absorbiert das andere System und wird instabil. Die Abbildung zeigt Nahaufnahmen der verschmelzenden Sterne.

5.2.1 Erkennbare Merkmale der Sternentstehung

In der zuvor gezeigten Simulation lassen sich einige der in Kapitel 2.1 beschriebenen Phasen und Merkmale der Sternentstehung beobachten. Von Anfang an ist die Gaswolke in Bewegung. Nach etwa 3 Zeiteinheiten ist ein Großteil der anfänglichen turbulenten Geschwindigkeiten durch die Viskosität abgebaut worden. Die Wolke beginnt sich besonders im inneren Teil zusammenzuziehen, bis ein dichter Kern entstanden ist. Sie teilt sich dabei durch Fragmentierung in mehrere Regionen auf. Bei $t = 3,8$ ist das Gas so stark komprimiert, dass die Temperatur ansteigt. Der erhöhte Druck verhindert zunächst ein weiteres Zusammenfallen. In Bild 12(b) ist der *Protostern* gut sichtbar, umgeben von einer Hülle aus Gas. Er nimmt immer mehr Gas aus seiner Hülle auf, während sich um ihn eine große Akkretionsscheibe bildet. Auch andere Fragmente des Kerns werden durch ihren eigenen Gasdruck gestützt. Die Akkretionsscheibe des ersten *Protosterns* beginnt selbst Fragmente zu bilden, was von den Interaktionen mit den anderen Objekten begünstigt wird.

Über die Entstehung von Sternen durch die Fragmentierung einer massereichen Akkretionsscheibe wurde auch von anderen Autoren berichtet [WCB+95]. Besonders in Clustern, die nur wenige Sterne enthalten, können Binärsysteme durch die Interaktion von einem Objekt mit der Akkretionsscheibe eines anderen entstehen [MC95].

Es bilden sich mehrere Sternensysteme, von denen einige durch die chaotischen Interaktionen verschmelzen (Abbildung 14), während andere aus dem Cluster herausgeschleudert werden (Abbildung 13). Übrig bleiben in diesem Fall ein einzelner Stern und zwei kleinere in einem Binärsystem. Dies stimmt mit Beobachtungen unseres Universums überein. Etwa zwei Drittel der Hauptreihensterne, die eine ähnliche Masse haben wie unsere Sonne, sind Teile von binären oder multiplen Systemen. Jüngere Sterne kommen sogar noch öfter zusammen vor [WTW11].

Nach 6 Zeiteinheiten haben die verbleibenden Sterne eine annähernd sphärische Form und deutlich definierte Oberfläche ausgebildet. Einige Regionen höherer Dichte, sowie die Akkretionsscheiben sind immer noch zu erkennen. Diese bleiben auch nach längerer Zeit erhalten, da die in den Sternen freigesetzte thermische Energie nicht simuliert wird. In der Wirklichkeit würden die Sterne das überschüssige Gas erwärmen, sodass es sich ausdehnt und wieder mit den Teilen des Interstellaren Mediums verschmilzt, die eine geringere Dichte haben.

Abbildung 15 zeigt die gesamte Gaswolke am Ende der Simulation.

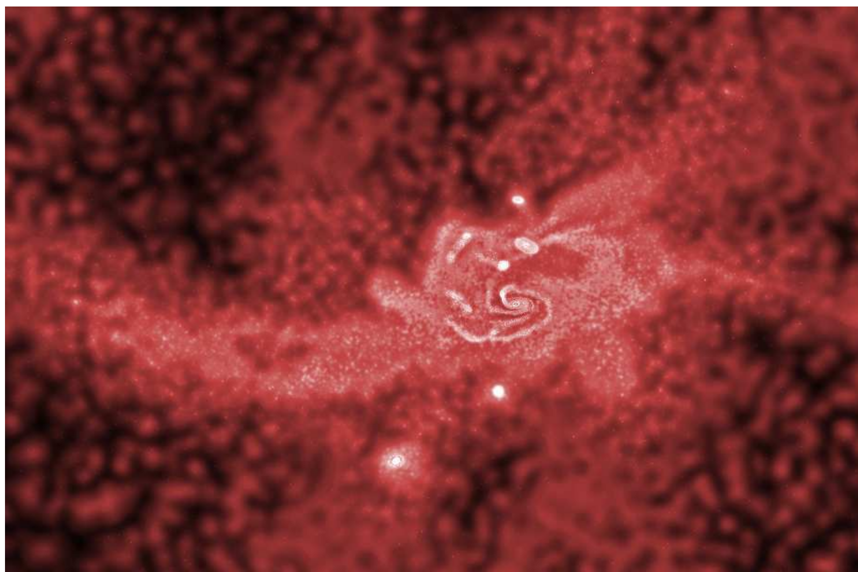
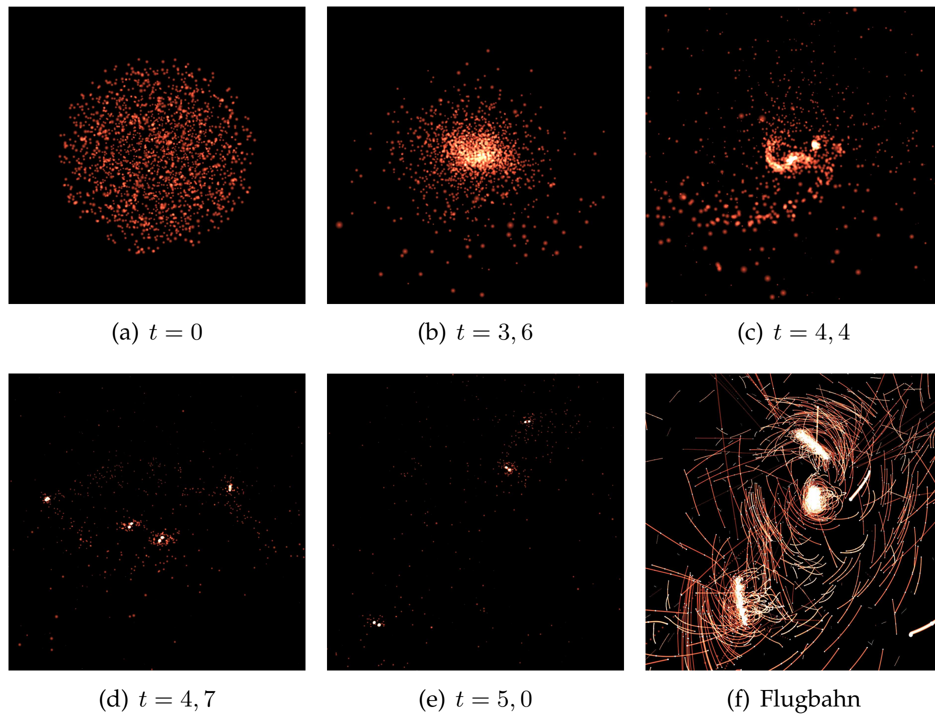


Abbildung 15: Unterschiedliche Detailstufen wurden mittels Photoshop zu einem Bild vereint. Es zeigt die Verteilung des Gases sowie die entstandenen Sterne am Ende der Simulation. Der einzelne Stern im Zentrum ist in Abbildung 14 aus einem Doppelstern entstanden. Oberhalb davon befindet sich das weg geschleuderte Sternensystem aus Abbildung 13. Einige weitere Strukturen sind in den äußeren Fragmenten der Gaswolke entstanden und nicht auf den anderen Bildern zu sehen.

5.2.2 Geringere Partikelzahlen

Viele dieser Merkmale können sogar noch in Simulationen mit 2048 und weniger Partikeln erkannt werden. Die Qualität der Visualisierung lässt allerdings nach. Abbildung 16 zeigt eine solche Simulation. In Bild (a) ist die gesamte Wolke abgebildet. Bei so wenigen Partikeln fällt es schwer sie als kontinuierliches Gas zu sehen. Nach den ersten $3,5$ bis 4 Zeiteinheiten hat sich der innere Teil der Wolke zu einem dichten Kern zusammengezogen (Bild (b)). Er teilt sich durch Fragmentierung in mehrere Bereiche, in denen dann bei $t = 4,7$ nach erneuter Fragmentierung Binärsterne zu erkennen sind. Durch die Interaktion der vier Binärsysteme verschmelzen zwei miteinander und ein weiteres wird aus dem Cluster herausgeschleudert. Zum Auflösen von Akkretionsscheiben reichen die wenigen Partikel nicht. Sie lassen sich höchstens in Bild (f) erahnen, wenn die Flugbahnen der einzelnen Partikel visualisiert werden.



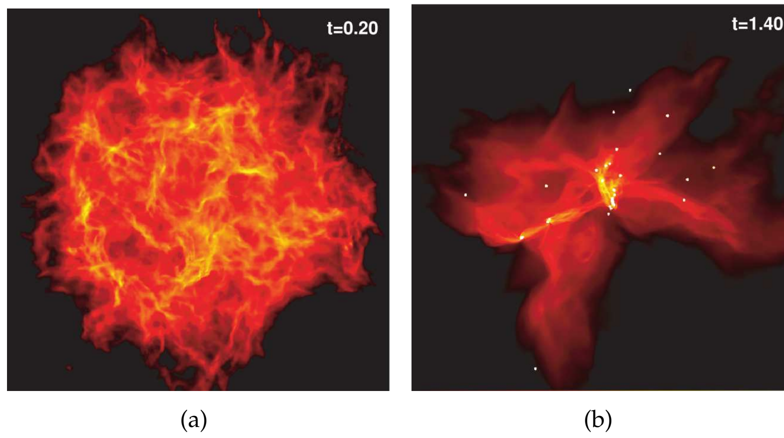
(a) $t = 0$ (b) $t = 3,6$ (c) $t = 4,4$

(d) $t = 4,7$ (e) $t = 5,0$ (f) Flugbahn

Abbildung 16: Der Ablauf einer Simulation mit nur 2048 Partikeln. Die Evolution der Gaswolke zeigt Ähnlichkeiten zu den Simulationen mit mehr Partikeln. Einige Merkmale sind immer noch zu erkennen.

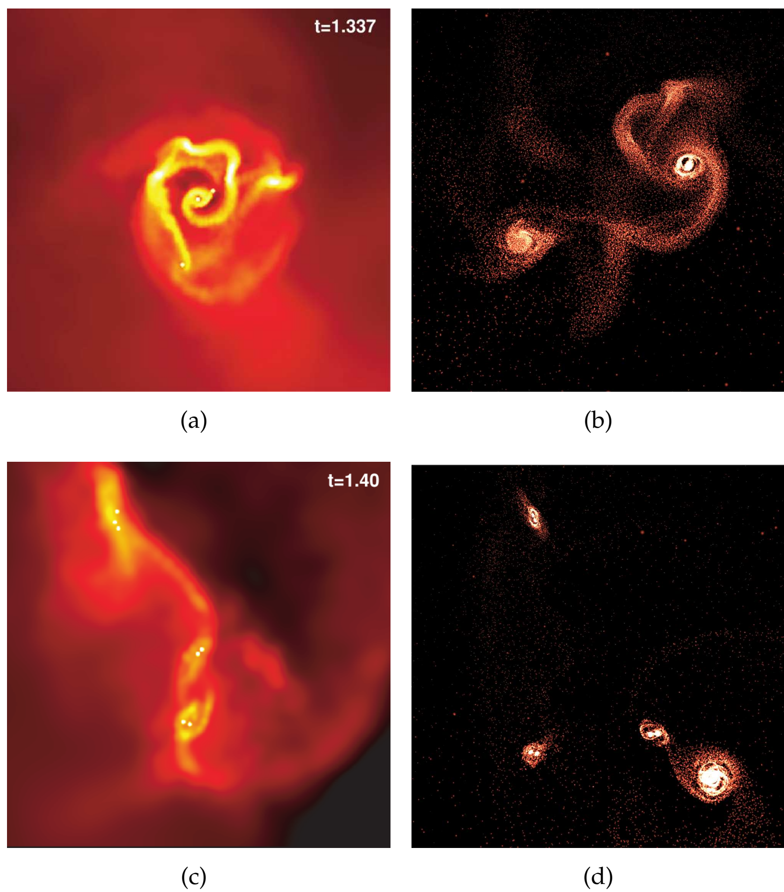
5.2.3 Vergleich mit anderen Simulationen

Bate et al. führen in [BBB03] eine Simulation durch, die einige Ähnlichkeiten zu dem hier vorgestellten Code aufweist. Sie verwenden zum Beispiel dieselbe Zustandsgleichung für die Druckberechnung. Allerdings werden Partikel in bereits entstandenen Sternen zusammengefasst. Dies spart Rechenzeit, sorgt aber auch dafür, dass junge Sterne sich nicht weiter aufteilen oder verschmelzen können. Bate et al. führen die Simulation mit 3500000 Partikeln durch. Hier entstehen mehrere dichte Kerne in der Wolke (Abildung 17). Betrachtet man Detailaufnahmen der Sternentstehungsgebiete, lassen sich durchaus Gemeinsamkeiten erkennen, wie Abbildung 18 zeigt.



(a) (b)

Abbildung 17: Die Abbildung zeigt die Gaswolke zu Beginn (a) und am Ende (b) der Simulation von Bate et al. [BBB03].



(a) (b)

(c) (d)

Abbildung 18: Das zweite Sternentstehungsgebiet aus [BBB03] (links) weist einige Ähnlichkeiten mit der hier vorgestellten Simulation (rechts) auf.

Genau wie in der hier gezeigten Simulation, bildet sich auch bei Bate et al. oft zuerst ein großer Stern, in dessen Akkretionsscheibe dann weitere Sterne entstehen. Dies geschieht hauptsächlich durch die Fragmentierung der Scheibe. Sie berichten auch von den Einflüssen anderer Objekte, die mit der Akkretionsscheibe interagieren. Durch diese Störung im Fluss des Gases entstehen weitere Strukturen.

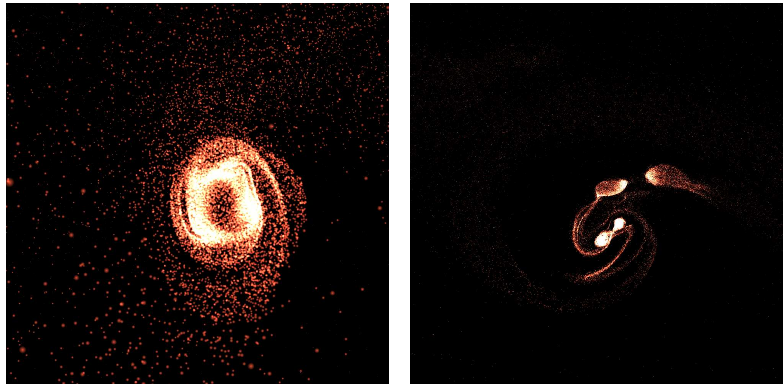
Bate et al. fanden in einer späteren Simulation heraus, dass das anfängliche Geschwindigkeitsfeld keinen Einfluss auf die Eigenschaften der entstandenen Sterne, insbesondere ihre Masse, hat [Bat09]. Dasselbe lässt sich auch mit dem hier vorgestellten Simulationscode beobachten. Wenn man die Parameter zum Generieren des anfänglichen turbulenten Geschwindigkeitsfeldes abändert entstehen in etwa die gleichen Sterne. Um andere Ergebnisse zu erzielen, muss die Rotationsgeschwindigkeit der Wolke verändert werden.

5.2.4 Mögliche Ursachen für Probleme

Neben den beschriebenen Merkmalen der Sternentstehung treten hin und wieder noch einige Artefakte und Probleme auf. In einigen Simulationen bilden sich dichte Ringe aus Gas, die schnell genug rotieren, um nicht zusammenzufallen. Auch in Akkretionsscheiben lassen sich oft dichte Ringe von Gas erkenne, die den Stern auf einer stabilen Umlaufbahn umkreisen. Die Mechanismen, die zum Abbau des Drehimpulses während der Sternentstehung beitragen, sind noch nicht vollständig erforscht. Einige Theorien lassen jedoch vermuten, dass Magnetfelder dabei eine Rolle spielen [Ray12]. Die Effekte von Magnetfeldern werden in dieser Simulation nicht berücksichtigt, wodurch sich einige Objekte möglicherweise schneller drehen als in der Wirklichkeit. Auch die manchmal beobachteten ovalen Sterne können dadurch erklärt werden.

Eine andere Ursache könnte die genutzte Methode zur Dämpfung der Gravitationskraft sein (siehe [DI93]). In vielen Partikelsimulationen wird daher für die Dämpfung der Gravitationskraft eine Splinefunktion verwendet.

Besonders in dichten Regionen treten auch Probleme mit Instabilitäten während der Simulation auf. Es kann vorkommen, dass einzelne Partikel nach besonders nahen Begegnungen mit anderen Objekten mit hoher Geschwindigkeit aus der Wolke hinausgeschleudert werden. Ist die Viskosität zu stark eingestellt, werden weitere Partikel mitgerissen und im schlimmsten Fall löst sich die gesamte Gaswolke in einer Kettenreaktion auf. Instabilitäten treten auch auf, wenn die Untergrenze des Kernelradius h_{min} zu klein gewählt wird. In sehr dichten Regionen kommt es dann zu Oszillationen, die zum Explodieren einzelner Sterne führen können. Steinmetz et al. beschäftigen sich in [SM93] mit möglichen Lösungen zu diesem Problem.



(a) ein Gasring (b) Sterne explodieren

Abbildung 19: Die Simulation ist nicht komplett frei von Problemen. Manchmal entstehen merkwürdige Objekte oder es kommt zu spontan explodierenden Sternen.

Eventuell kann ein intensives Testen verschiedener Parameter eine komplett stabile Simulationen erzeugen. Es ist aber auch denkbar, dass die Algorithmen zum Berechnen der Viskosität und des Kernelradius noch verbessert werden müssen. Im Programmteil, der für die Berechnung des Zeitschrittes verantwortlich ist, wurden einige Kompromisse zugunsten der Rechengeschwindigkeit gemacht, die ebenfalls Instabilitäten hervorrufen könnten. Außerdem könnte die Verwendung von *double precision* statt der momentan verwendeten *single precision* Gleitkommazahlen Genauigkeit und Geschwindigkeit der Simulation verbessern. Wobei die Verwendung des Größeren Datentyps die Berechnungen am meisten verlangsamen würde.

6 Fazit

In dieser Arbeit wurde ein partikelbasierter, astrophysikalischer Simulationscode entwickelt. Mit diesem wurde die Entstehung von Sternen in einer Gaswolke simuliert. Durch die Nutzung der *GPU* lassen sich diese Simulationen mit interaktiver Bildwiederholrate durchführen.

Die verwendeten *OpenGL compute shader* sind unabhängig von Grafikkarte und Betriebssystem. Verschiedene Partikelanzahlen sowie einige Parameter zum Anpassen des Algorithmus auf die verfügbare Hardware ermöglichen interaktives Simulieren auf den meisten modernen Computern mit einer dedizierten Grafikkarte. Workstation Computer ermöglichen dabei detaillierte Simulationen mit mehr als 65536 Partikeln, während ein Notebook immerhin noch 16384 Partikel interaktiv simulieren kann.

Die Simulation bildet den Entstehungsvorgang dabei plausibel und optisch ansprechend ab. Einige bekannte Phasen und Merkmale der Sternentstehung finden sich wieder. Dazu zählt zum Beispiel die Fragmentierung der Gaswolke, die Bildung von Akkretionsscheiben und die bevorzugte Entstehung von binären und multiplen Sternensystemen. Einige dieser Merkmale sind auch noch in Simulationen mit sehr geringer Partikelzahl (2048 Partikel) zu erkennen.

Außerdem können die Anfangsbedingungen durch kleine Anpassungen im Code beliebig gewählt werden, sodass sich beliebige Phänomene simulieren lassen, solange sie durch Gravitation und *SPH* modelliert werden können.

Mit dem hier gezeigten Code lässt sich die Entstehung von Sternen also sehr schnell auf günstigen Computern simulieren. Auch wenn die meisten der physikalischen Modelle dieselben sind, die auch bei großen wissenschaftlichen Simulationen verwendet werden, lässt sich eine so hohe Geschwindigkeit nicht erreichen, ohne einige Kompromisse einzugehen. Nicht alle physikalischen Effekte, die für die Entstehung von Sternen relevant sind, wurden simuliert. Nicht berücksichtigt wurde zum Beispiel die Evolution der thermischen Energie sowie Magnetfelder. Es wurde außerdem auf schnellere *single precision* Gleitkommazahlen sowie einfacher zu berechnende Kernelfunktionen zurückgegriffen.

Eine komplette Analyse im Hinblick auf die Genauigkeit der Simulation nach physikalischen Gesichtspunkten wurde nicht durchgeführt. Ohne Modifikationen ist der Code vermutlich nicht geeignet, um neue Forschungsergebnisse abzuleiten. Er könnte allerdings in der Lehre oder in Ausstellungen eingesetzt werden, wenn es um Entstehung von Strukturen im Weltraum geht. Wird die Anwendung noch um eine grafische Benutzeroberfläche erweitert, könnten Besucher selbst mit verschiedenen Anfangsbedingungen experimentieren.

Auch denkbar wäre es, einzelne Bestandteile des hier entwickelten Simulationscodes zu verwenden, um große Physiksimulationen zu beschleunigen.

7 Ausblick

Es gibt einige Möglichkeiten, um die Performance des Simulationscodes noch weiter zu steigern und so Simulationen mit größeren Partikelzahlen zu ermöglichen. Das Implementieren einer Baumstruktur und einer wiederverwendbaren Liste der Nachbarn, wie in Kapitel 3.5.2 beschrieben, ist eine davon. Auch das Verwenden von individuellen Zeitschritten für jeden Partikel (Kapitel 3.3.3) könnte Rechenzeit einsparen. Ein weiteres interessantes Thema ist die Aufteilung der Berechnung auf mehrere *GPUs* in einem System, oder sogar auf mehrere Computer innerhalb eines Clusters. Zusätzlich wäre es möglich, Partikel in sehr dichten Regionen zusammenzufassen, um so die Anzahl der Partikel im Verlauf der Simulation zu reduzieren. Es wäre außerdem hilfreich, einige Aufgaben auf die *CPU* auszulagern, die momentan nicht ausgelastet ist.

Um die Ergebnisse der Simulation zu verbessern, können auch weitere physikalische Effekte mit einbezogen werden. Zum Beispiel die Entwicklung der thermischen Energie während des Zusammenfalls der Wolke. Dadurch könnten die Werte für den Gasdruck genauer bestimmt werden. In der Realität spielen auch Magnetfelder bei der Entstehung von Sternen eine Rolle, da sie den Drehimpuls reduzieren können [Ray12]. Auch diese könnten in einer SPH Simulation berücksichtigt werden. Die aktuelle Simulation erzeugt gute Ergebnisse bis zu dem Zeitpunkt, an dem die ersten Sterne entstehen. Könnte man die Sterne erkennen und die Vorgänge in ihrem Inneren modellieren, wäre es möglich, die Auswirkungen auf das Gas zu simulieren. Dazu zählt zum Beispiel die Erwärmung durch die bei der Fusionsreaktion freigesetzte Energie. Die Simulation würde so im späteren Verlauf realistischere Ergebnisse liefern. Um die Stabilität der Simulation zu verbessern, sollten außerdem die in Kapitel 5.2.4 beschriebenen Probleme genauer untersucht werden.

Um ein genaueres Untersuchen der Ergebnisse zu ermöglichen, wäre eine Funktion hilfreich, mit der die Partikeleigenschaften nach jedem Simulationsschritt gespeichert werden. Eine aufwendige Simulation kann dann später mehrfach angeschaut oder mathematisch analysiert werden.

Um die Nutzung der Simulation und ein Experimentieren mit verschiedenen Einstellungen für jeden zu ermöglichen, kann eine grafische Benutzeroberfläche erstellt werden. Diese sollte dann das Einstellen verschiedener Anfangsbedingungen und Parameter erlauben, sowie die entstandenen Strukturen markieren und deren Eigenschaften anzeigen.

Auch das Verfahren zur Visualisierung kann noch weiterentwickelt werden, um das Aussehen der Gaswolke an den aus Teleskopaufnahmen bekannten Look anzupassen. Die Verwendung von Techniken aus dem Volumenrendering wäre eine Möglichkeit. Beim Betrachten der Simulation wäre es außerdem hilfreich, die Kamera auf eine entstandene Struktur fokussieren zu können und ihre Bewegung zu verfolgen.

Literatur

[Bal95] Dinshaw S Balsara. Von neumann stability analysis of smoothed particle hydrodynamics—suggestions for optimal algorithms. *Journal of Computational Physics*, 121(2):357–372, 1995.

[Bat09] Matthew R Bate. The dependence of star formation on initial conditions and molecular cloud structure. *Mon. Not. R. Astron. Soc*, 248:232–248, 2009.

[BBB03] Matthew R. Bate, Ian A. Bonnell, and Volker Bromm. The Properties of Stars and Brown Dwarfs. *Galactic Star Formation Across the Stellar Mass Spectrum*, 287:427–432, 2003.

[BGP12] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231(7):2825–2839, 2012.

[BHF07] Robert Bridson, Jim Hourihan, and Tweak Films. Curl-Noise for Procedural Fluid Flow. *ACM Trans. Graph. Article*, 26(10), 2007.

[BLRY07] Peter Bodenheimer, Gregory Laughlin, Michal Rozyczka, and Harold Yorke. Numerical Methods in astrophysics: An Introduction. chapter 4, page 352. 2007.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed Particles: A new paradigm for animating highly deformable bodies. pages 61–76, 1996.

[DI93] Charles C Dyer and Peter S S Ip. Softening in N-body simulations of collisionless systems. *Astrophysical Journal*, 409:60–67, 1993.

[GM77] Robert A Gingold and Joseph Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 1977.

[Hau08] Guillermo Hauke. *An introduction to fluid mechanics and transport phenomena*, volume 86. Springer, 2008.

[HCM06] Kyle Hegeman, Nathan A. Carr, and Gavin S P Miller. Particle-based fluid simulation on the GPU. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3994 LNCS:228–235, 2006.

[HK89] Lars Hernquist and Neal Katz. TREESPH - A unification of SPH with the hierarchical tree method. *The Astrophysical Journal Supplement Series*, 70:419, 1989.

[IOS+14] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH Fluids in Computer Graphics. *Eurographics*, (2):21–42, 2014.

[LHW12] Roland Leißa, Sebastian Hack, and Ingo Wald. Extending a c-like language for portable simd programming. *ACM SIGPLAN Notices*, 47(8):65–74, 2012.

[MC95] John McDonald and Cathie J Clarke. The effect of star-disc interactions on the binary mass-ratio distribution. *Monthly Notices of the Royal Astronomical Society*, 275:671, 1995.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-Based Fluid Simulation for Interactive Applications. *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, (5):154–159, 2003.

[MG83] Joseph Monaghan and Robert A Gingold. Shock simulation by the particle method sph. *Journal of computational physics*, 52(2):374–389, 1983.

[MH07] Junichiro Makino and Piet Hut. Moving Stars Around. *The Art of Computational Science*, 1, 2007.

[MO07] Christopher F. McKee and Eve C. Ostriker. Theory of Star Formation. 45, 2007.

[Mon92] Joseph Monaghan. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543–574, 1992.

[Mon97] Joseph Monaghan. Sph and riemann solvers. *Journal of Computational Physics*, 136(2):298–307, 1997.

[Mon05] Joseph Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, 2005.

[Ngu07] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. *Simulation*, 3(1):677–696, 2007.

[Nvi17] Nvidia. CUDA C Programming Guide, 2017.

[Ray12] Tom Ray. *Losing spin: The angular momentum problem*. Oxford University Press, oct 2012.

[RL99] Frederic A Rasio and James C Lombardi. Smoothed Particle Hydrodynamics Calculation of Stellar Interactions. *Journal of Computational and applied Mathematics*, 109:213–230, 1999.

[SM93] Matthias Steinmetz and Ewald Mueller. On the capabilities and limits of smoothed particle hydrodynamics. *Astronomy and Astrophysics*, 268(1):391–410, 1993.

[Spr02] Springel, Volker and Hernquist, Lars. Cosmological smoothed particle hydrodynamics: the entropy equation. *Monthly Notices of the Royal Astronomical Scociety*, 333(January):649–664, 2002.

[Spr05] Volker Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[Ste96] Matthias Steinmetz. Grapesph: cosmological smoothed particle hydrodynamics simulations with the special-purpose hardware grape. *Monthly Notices of the Royal Astronomical Society*, 278(4):1005–1017, 1996.

[SYW01] Volker Springel, Naoki Yoshida, and Simon D.M. White. GADGET: A code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, 2001.

[WCB+95] Anthony P Whitworth, Steven Chapman, Amardeep Bhattal, Mike Disney, H Pongracic, and Jacob Turner. Binary star formation: accretion-induced rotational fragmentation. *Monthly Notices of the Royal Astronomical Society*, 277:727–746, 1995.

[WTW11] Derek Ward-Thompson and Anthony P. Whitworth. *An Introduction to Star Formation*. Cambridge University Press, 2011.

Abbildungsverzeichnis

1 Junge Sterne im Orionnebel 4
2 Sternentstehung in den Spiralarmen von Messier 18 5
3 Rekursive Fragmentierung einer Gaswolke 6
4 Vergleich Festkörper und Fluid 8
5 Partikel- und Gitterbasierte Simulation 8
6 *Gather* und *Scatter* Interpretation von *SPH*. [HK89] 16
7 Verschiedene Speicherlayouts 28
8 Aufteilung in *work groups* 28
9 Einstellungen der Visualisierung 40
10 Dauer eines Simulationsschrittes 44
11 Evolution der Gaswolke 45
12 Sternentstehung im *dense core* 46
13 Weggeschleudert 47
14 Verschmelzung eines Doppelsterns 47
15 Ende der Simulation 49
16 Simulation mit nur 2048 Partikeln 50
17 Bilder aus der Simulation von Bate et al. 51
18 Direkter Vergleich der Simulationen 51
19 Fehler in der Simulation 53

Algorithmenverzeichnis

1 Beispielhafter Ablauf einer einfachen *SPH* Simulation 12
2 Der Leapfrog Algorithmus in kick-drift-kick Form 23
3 Aktualisieren der Simulation Teil 1 25
4 Aktualisieren der Simulation Teil 2 26

Quellcodeverzeichnis

1 Funktion zur Durchführung eines Simulationsschrittes. 34
2 Shader zur Anpassung des Kernelradius. 34
3 Aufbau des Dichte-Shaders. 35
4 Berechnung der Dichte. 36
5 Aufsummierung der Teilergebnisse. 37
6 Berechnung der Beschleunigung. 38
7 OpenGL-Einstellungen zum Rendern 39
8 Der Vertexshader. 39
9 Der Fragmentshader. 40